ČESKÉ VYSOKÉ UČENÍ TECHNICKÉ V PRAZE
FAKULTA INFORMAČNÍCH TECHNOLOGIÍ

ZADÁNÍ DIPLOMOVÉ PRÁCE

Název:	C++ knihovna pro tvorbu her a intermediálních aplikací
Student:	Bc. Adam Vesecký
Vedoucí:	Ing. Martin Půlpitel
Studijní program:	Informatika
Studijní obor:	Webové a softwarové inženýrství
Katedra:	Katedra softwarového inženýrství
Platnost zadání:	Do konce letního semestru 2016/17

Pokyny pro vypracování

Cílem práce je návrh a implementace knihovny v C++ pro tvorbu multimediálních aplikací, především her, pomocí komponentově orientované architektury. Dále pak návrh a implementace prototypu 2D on-line hry pro více hráčů, která bude tuto knihovnu využívat.
Požadavky na obsah knihovny:
- správa herních objektů, komponent, zpráv a událostí,
- komponenty pro práci s uživatelským vstupem, s grafikou, hudbou, zvuky, s daty (SQLite, XML) a komunikací (HTTP, TCP).
Požadavky na technologické prvky 2D hry:
- bude využívat některý z prvků umělé inteligence (pravděpodobnostní algoritmy, stavové prostory, rozhodovací procesy),
- bude implementovat komunikaci s nízkou latencí pro přenos herního modelu mezi zařízeními (multiplayer).
Podmínky:
- knihovna bude využívat framework OpenFrameworks a bude moci být distribuována jako její dodatek (addon, viz: http://www.ofxaddons.com/categories).
Návrh knihovny i prototyp hry dokumentujte pomocí metod SI, obojí implementujte, dokumentujte a otestujte.

Seznam odborné literatury

https://raw.githubusercontent.com/surjikal/cbgos-experiment/master/doc/nicolasporter-cbgos-paper.pdf
http://www.gdcvault.com/free/gdc-canada-09

[redacted]

Ing. Michal Valenta, Ph.D.
vedoucí katedry

[redacted]

prof. Ing. Pavel Tvrdík, CSc.
děkan

V Praze dne 10. února 2016

České vysoké učení technické v Praze
Fakulta informačních technologií
Katedra Softwarového inženýrství

Diplomová práce

C++ knihovna pro tvorbu her a intermediálních aplikací

Bc. Adam Vesecký

Vedoucí práce: Ing. Martin Půlpitel

9. května 2016

Poděkování

Děkuji svému vedoucímu, Ing. Martinu Půlpitelovi, za to, že mi poskytl možnost zrealizovat tento projekt a osvojit si techniky z širokého spektra různých oblastí. Rovněž děkuji své úžasné rodině za podporu při psaní této práce.

Prohlášení

Prohlašuji, že jsem předloženou práci vypracoval(a) samostatně a že jsem uvedl(a) veškeré použité informační zdroje v souladu s Metodickým pokynem o etické přípravě vysokoškolských závěrečných prací.

Beru na vědomí, že se na moji práci vztahují práva a povinnosti vyplývající ze zákona č. 121/2000 Sb., autorského zákona, ve znění pozdějších předpisů, zejména skutečnost, že České vysoké učení technické v Praze má právo na uzavření licenční smlouvy o užití této práce jako školního díla podle § 60 odst. 1 autorského zákona.

V Praze dne 9. května 2016 .....

Odkaz na tuto práci

Vesecký, Adam. *C++ knihovna pro tvorbu her a intermediálních aplikací.* Diplomová práce. Praha: České vysoké učení technické v Praze, Fakulta informačních technologií, 2016.

Abstrakt

Tato práce se zabývá obecnou problematikou spojenou s programováním počítačových her a herních enginů. Výhody komponentově orientovaného přístupu, umožňujícího jednoduché přidávání nových funkcionalit, demonstruje vlastní herní engine, napsaný v jazyce C++ jako nadstavba frameworku OpenFrameworks a použitelný pro tvorbu desktopových i mobilních her. Součástí práce je rovněž stručný úvod do problematiky multiplayeru a umělé inteligence, jež byly použity při implementaci realtime strategické hry, která zároveň testuje použitelnost navrženého enginu.

Klíčová slova herní engine, komponentová architektura, multiplayer, umělá inteligence

Abstract

This thesis deals in general with programming of computer games and game engines. It introduces advantages of the component-oriented approach, allowing the programmer to simply add new functionalities. An own engine is also a part of the thesis. It is written in C++ language as an extension of the Openframeworks framework and is suitable for making desktop and mobile games. The other part of this work briefly describes multiplayer and artificial intelligence techniques that were used during the implementation of a real-time strategy game. The game also tests the applicability of the proposed engine.

Keywords game engine, component architecture, multiplayer, artificial intelligence

Obsah

Seznam obrázků

Seznam tabulek

Úvod

Počítačové hry tvoří již dlouhá léta významný prvek zábavního průmyslu. Od roku 1972, kdy firma Atari Inc. vydala první komerčně úspěšnou hru Pong [1], urazily dlouhý kus cesty. Společně s rozvojem digitálních technologií se vyvíjela i jejich propracovanost a rozmanitost, což můžeme sledovat např. u herních sérií jako Need for Speed či NHL, kde časový odstup mezi jednotlivými díly není větší než pár let, a přesto je na každém dalším produktu citelně znát technologický pokrok. Z hlediska grafického zpracování je na dnešních počítačích možné dosáhnout téměř fotorealistického zobrazení herního světa.

Tento pokrok však měl negativní dopad na poli malých nezávislých her (*Indie games*), a to především na konci devadesátých let, kdy stále se zvyšující nároky hráčů začaly převyšovat možnosti malých vývojářských týmů.

To se však změnilo v průběhu poslední dekády, kdy došlo ke globálnímu nárůstu počtu uživatelů připojených k internetu a s tím spojené tvorbě malých onlinových her, ale také k nástupu chytrých telefonů a tabletů, které otevřely herní svět pro nenáročné uživatele.

Jak rostl počet prodaných chytrých telefonů, zvyšoval se i počet aplikací pro tyto platformy. V listopadu roku 2015 obsahovala distribuční služba Google Play více než 1 750 000 aplikací [2], přičemž necelou polovinu tvořily hry.

Byly to právě hry pro mobilní aplikace, jednoduchost distribuce a poměrně nízká časová náročnost na vývoj, co zapříčinilo nový rozkvět Indie games. Za vývojem jednoduché hry, která se těší popularitě velkého množství lidí, stojí mnohdy i jednočlenný tým.

Komunita tvůrců nezávislých her se začala rozšiřovat a s ní i počet technologií, které ke své práci mohou využít. Vývoj hry obvykle nezačíná na zelené louce, neboť je zde k dispozici celá řada hotových řešení, od podpůrných nástrojů a knihoven až po komplexní frameworky, některé i s vlastním vývojovým prostředím. Těmto nástrojům se říká *herní enginy* a právě o nich bude řeč, především o těch, co jsou využívány pro tvorbu mobilních her.

Herní engine

První hry jako *Pac-Man* či *Tetris* byly implementovány přímo pro daný hardware a kód nebylo možno použít pro vývoj dalších her. To se změnilo až s nástupem moderních programovacích jazyků, umožňujících vyšší úroveň abstrakce. Již před vznikem samotných herních enginů usilovali vývojáři o oddělení herní logiky od programové základny. Začaly se

objevovat konfigurační struktury (např. INI soubory ve hře *Civilization II*) s definicí herních prvků jako jsou životy, pozice a mapy úrovní.

Za zlomový je považován rok 1993, kdy společnost id Software vydala hru *Doom* [3]. Tehdejší vedoucí vývoje, John Carmack, navrhnul herní model tak, aby byla veškerá data jako mapy, grafika, zvukové efekty a hudba, v oddělených souborech - tzv. WAD (*Where's All the Data*), čehož mohla využít amatérská scéna. Začaly vznikat tzv. *MOD komunity* - vývojáři vytvářející hry modifikací existujících her. Sady nových úrovní pro hru Doom, jsou vytvářeny dodnes.

V následujících letech vzniklo mnoho dalších enginů; některé byly vytvářeny společně s komerční hrou, aby mohly být jejich funkce a možnosti spolehlivě otestovány. Mezi nejznámější z toho období patří např. *Build* se hrou *Duke Nukem 3D*, *Quake engine* a *GoldSRC* se hrou *Half-Life*. Dalším populárním enginem se stal *Unreal Engine*, poprvé použit ve hře *Unreal* na konci 90. let.

Dnes existuje celá řada komerčně úspěšných enginů, více či méně orientovaných na konkrétní herní žánr. Přílišná univerzálnost však má své nevýhody - design je obvykle úzce spjat s typem her, které jej budou využívat. Dá se říct, že univerzálnější herní enginy jsou méně vhodné pro vývoj konkrétního typu hry než ty specializované.

To platí především pro malé hry pro mobilní platformy, které možnosti daného enginu zdaleka nevyužijí, přesto se do cílové aplikace jeho komponenty zaintegrují, což má negativní dopad jak na výkon, tak i na celkovou velikost aplikace.

Cílem této práce je návrh takového enginu, který poskytne vývojářům her, především těch mobilních, užitečnou sadu nástrojů pro vývoj dvourozměrných her. Engine bude modulární, snadno rozšiřitelný a optimalizovaný pro běh i na starších zařízeních.

Cíle práce

Techniky používané při vývoji počítačových her pokrývají obrovskou teoretickou oblast a není v silách jednotlivce ovládat je všechny. Klíčem k dobře navržené hře je především pochopení základních principů architektury těchto systémů, která se od tradičních aplikací značně liší.

Jedním z těchto principů je *komponentově orientované programování*, umožňující snadné přidávání nových funkcionalit bez nutnosti zásahu do jiných částí aplikace. Právě na tomto principu bude založen herní engine, jehož prototyp tvoří první část této práce.

Druhá část bude pojednávat o technikách používaných při samotné implementaci her jako jsou algoritmy umělé inteligence (hledání cest, stavový automat, prohledávání stavového prostoru) a přenos herního modelu po síti (multiplayer). Tyto techniky budou na konci zaintegrovány do prototypu realtime strategické hry (*Real-time strategy*, RTS), která bude tento engine využívat.

V následujících kapitolách bude předpokládáno, že čtenář zná základy objektového programování, jazyka UML, XML a C++, je seznámen s teorií grafů a stavových prostorů, lineární algebrou a ISO/OSI komunikačním modelem.

Obsah práce

V kapitole 1 budou popsány vlastnosti komponentově orientované architektury a její srovnání s objektovým přístupem. Také bude diskutováno, v jakých podobách se tato architektura vyskytuje u vybraných herních enginů a jakou cestou se bude ubírat vlastní řešení.

Kapitola 2 se bude věnovat návrhu a implementaci architektury vlastního enginu.

V kapitole 3 budou představeny grafické komponenty enginu, způsob, jakým jsou řešeny transformace, animace a vykreslování obecně.

Kapitola 4 popíše ostatní části enginu jako je konfigurace, skriptování, práce se zvuky a asynchronní procesy.

Kapitola 5 představí základní principy přenosu herního modelu po síti a návrh vlastního řešení, které bude taktéž součástí enginu.

V kapitole 6 budou zmíněny různé techniky používané při implementaci umělé inteligence a jejich integrace do enginu.

Kapitola 7 bude pojednávat o návrhu a implementaci prototypu real-time strategie, která bude engine a jeho komponenty využívat.

V kapitole 8 budou prezentovány výsledky testování funkčnosti enginu a jeho kvalit.

V závěrečné kapitole budou zhodnoceny výsledky práce a diskutována možná rozšíření.

Kapitola 1

Herní enginy a komponentová architektura

Ludologie je věda, která se snaží odpovědět na otázku, co vlastně dělá hru hrou a jaké vlastnosti ji definují. Chceme-li vymezit jejich klíčové prvky z pohledu IT, je třeba nejprve pochopit, čím se odlišují od klasických systémů.

U aplikací jako je CMS či DMS se setkáváme s něčím, čemu říkáme *request a response* mechanismus. Uživatel zadá požadavek - *request*, systém jej zpracuje a vrátí výsledek - *response*. Tím práce systému končí, dokud uživatel nezadá další požadavek. O těchto systémech můžeme mluvit jako o nástrojích na zpracování dat. Tato data navíc nejsou nutně spjata se samotným systémem a mohou existovat i mimo něj. Různé instance téhož programu mohou navíc obsahovat zcela odlišná data.

Oproti tomu hry jsou simulátory - simulátory fiktivního světa, který se neustále mění, ať už samovolně nebo na základě interakce s hráčem. Každá hra obsahuje něco, čemu se říká *gameplay*. Jedná se o sadu pravidel a herních postupů, které definují, jaké akce je možno u konkrétního stavu hry provést a jaké jsou odměny či penalizace za jejich provedení. Na základě těchto znalostí provádí hráč v každém okamžiku volbu následné akce. Tento princip platí u všech her - deskových, logických, strategických i akčních. Na tato pravidla je možno nahlížet jako na model celé hry, jelikož existují samostatně a herní svět je pouze projektuje pomocí grafiky, zvuků, hudby a vyprávění.

Kromě *gameplaye* obsahuje hra také *content*, což jsou samotná data. Na rozdíl od klasických aplikací tato data nevytváří uživatel ale jsou pevnou součástí hry. Některé hry sice nabízí možnost editace *contentu* prostřednictvím editorů vlastních misí a kampaní, tím ale povznáší roli hráče na roli návrháře či vypravěče.

1.1 Hra jako realtime systém

Vezmeme-li si jako příklad typickou hru z žánru adventur, kde hráč ovládá postavu, jež navštěvuje různé lokace, sbírá předměty a mluví s jinými postavami, můžeme ji implementovat mnoha způsoby: jako trojrozměrnou hru s grafickými efekty, bohatou animací a dramatickou doprovodnou hudbou, ale také jako obyčejnou konzolovou aplikaci, která aktualizuje herní model pouze poté, co uživatel zadá příkaz (např. *Jdi na sever*).

Zatímco gameplay je v obou případech stejný, obě aplikace projektují hru odlišným způsobem. V implementaci s grafickým prostředím existuje tzv. *herní smyčka*, která v jednotlivých časových intervalech aktualizuje herní model - právě tento princip má zásadní dopad na herní zážitek, jelikož systém běžící v reálném čase může být v reálném čase i vizualizován, animován apod.

Tato kategorie her je souhrnně nazývána jako *real-time interactive agent-based computer simulators* [4]. Dále se práce bude zabývat právě touto kategorií.

1.2 Základní prvky herního systému

Základním kamenem každé realtime hry je herní smyčka, která mimo jiné v pravidelných časových intervalech provádí následující akce:

- Zpracování vstupu od uživatele
- Aktualizace herního modelu
- Překreslení herní scény

V závislosti na implementaci může taková smyčka obsahovat i další akce (např. aktualizace zvukového bufferu). Důležité je, aby jednotlivé iterace této smyčky netrvaly příliš dlouho. Filmová videa obvykle dosahují rychlosti 24 snímků za sekundu (perioda 42 ms) a nižší frekvence již může mít negativní dopad na herní zážitek. Jako doporučená frekvence se obvykle uvádí 30-60 snímků za sekundu.

1.2.1 Zpracování vstupu od uživatele

U klasických aplikací, vyvíjených nad nějakým frameworkem vyšší vrstvy, se pro zpracování vstupů registrují metody, které tato vrstva sama zavolá (*listenery* v jazyce Java či *eventy* v C#). Pokud zde taková komponenta není, je nutné zpracovat vstup explicitně. Během každé iterace herní smyčky dojde k ověření aktuální kombinace stisknutých kláves a pozice myši či dotyku. Tyto údaje je nutné zpracovat a například pro klik myši identifikovat objekty v herní scéně, které jsou danou událostí zasaženy.

1.2.2 Aktualizace herního modelu

V této části smyčky obvykle probíhají všechny akce, které nesouvisí přímo s vykreslováním. Může se jednat o výpočet pohybu herních objektů, fyzikální simulaci, komunikaci po síti nebo plánování strategie.

Některé komponenty vyžadují pro zachování granularity vyšší frekvenci aktualizace (např. fyzikální model), některé je potřeba aktualizovat jen sporadicky (např. kontrola plnění úkolů). Proto je potřeba v hlavní smyčce provádět aktualizaci co nejčastěji a nechat jednotlivé komponenty, aby si frekvenci své vlastní aktualizace nastavily dle potřeby.

1.2.3 Překreslení herní scény

Vše, co nesouvisí s herním modelem, ale má bezprostřední dopad na výslednou podobu grafického výstupu, se provádí v této části. Pokud je například pro vykreslení potřeba objekty

nejprve seřadit, spočítat normálové vektory, načíst textury nebo spustit *shadery*, mělo by k tomu docházet zde.

Některé frameworky používají k překreslení samostatnou smyčku, což je možno využít např. pro *vertikální synchronizaci* - techniku, která sesynchronizuje překreslovací frekvenci scény s frekvencí monitoru, čímž odstraní některé nežádoucí artefakty, způsobené při rychlém pohybu objektů.

1.3 Architektura hry

V raných dobách byly hry vytvářeny přímo na míru cílovému typu hardware. Znovupoužitelnost kódu ztěžovala nejrůznější paměťová a výkonnostní omezení, kterými daná platforma trpěla. Příkladem může být zdrojový kód hry *Prince of Persia*, dostupný zde: [5].

Teprve s nástupem objektových programovacích jazyků, nabízejících vyšší úroveň abstrakce, se stal vývoj mnohem pohodlnější.

Pro návrh herní architektury je možno využít celou řadu přístupů. Nyní budou popsány dva z nich - objektový a komponentový.

1.3.1 Objektový přístup

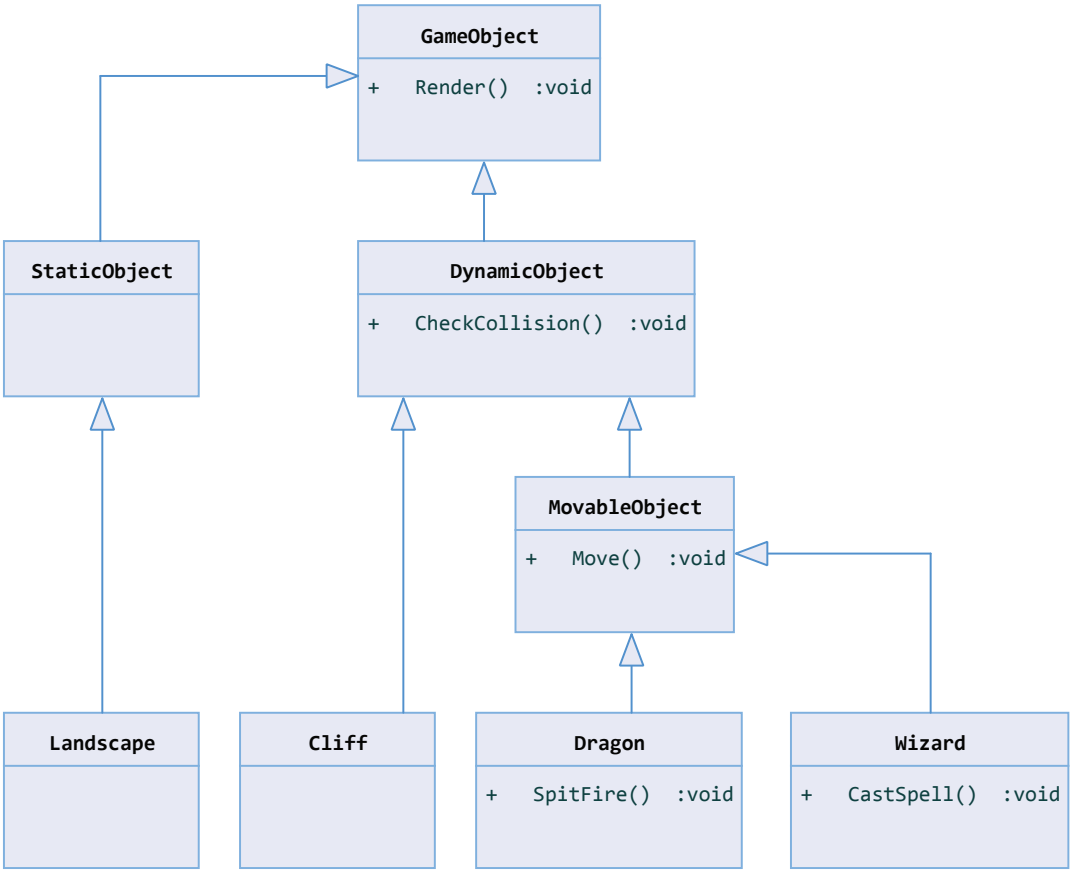
V tradičních informačních systémech je hojně využíváno konceptu dědičnosti, kdy se na vyšší úrovni nacházejí abstraktní třídy, od kterých jsou odvozovány stále více specializované třídy. Typickým příkladem je GUI v technologii .NET WPF (*Windows Presentation Foundation*), kde jsou všechny grafické elementy odvozeny od společné třídy `UI Element`, ta je zase odvozena od třídy `Dispatcher Object` [6].

Představme si jednoduchou scénu, ve které máme draka a kouzelníka, jak znázorňuje obrázek 1.1.

Drak může chrlit oheň, kouzelník dokáže kouzlit. Oba se mohou pohybovat a interagovat s jinými objekty - např. s kamenným útesem, na kterém stojí. Ten pak musí obsahovat něco jako detekci kolizí, aby se nepropadli dolů.

Obrázek 1.1: Příklad herní scény

Možnou podobu objektové hierarchie ukazuje obrázek 1.2. Všechny objekty jsou potomky obecné třídy `GameObject`, obsahující metodu `Render()` pro vykreslování objektů. Útes (`Cliff`) je potomkem třídy `DynamicObject` a dokáže tedy detekovat kolize. Drak s kouzelníkem se mohou pohybovat, jelikož jsou potomky specializované třídy `MovableObject`.



Obrázek 1.2: Příklad použití hierarchie tříd

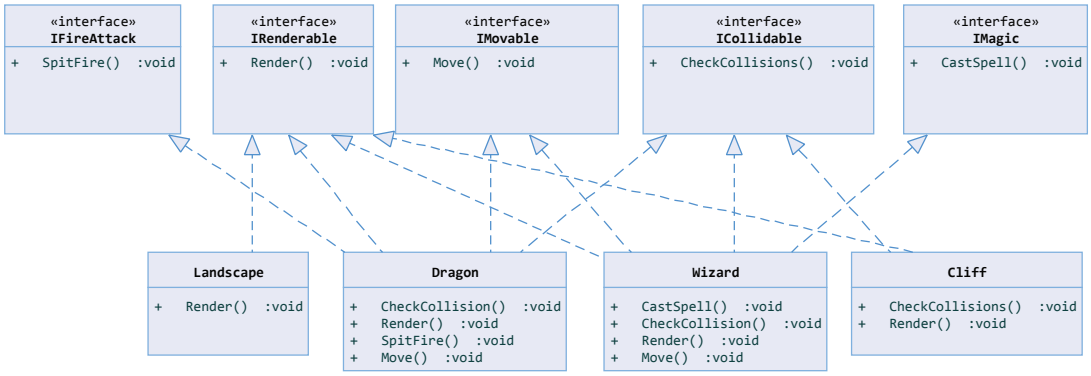
Takový přístup je použitelný pro jednoduché hry s několika málo pravidly, avšak u složitějších her brzy narazí na problém se škálovatelností.

Pokud bychom chtěli vytvořit draka, který umí kouzlit, musel by podědit vlastnosti jak od dosavadní třídy `Dragon`, tak od třídy `Wizard`. Od útesu můžeme v budoucnu vyžadovat, aby se po určité akci rozpadnul na dvě části, čímž by z jedné instance třídy `Cliff` vznikly dvě. Kouzelník se může proměnit ve draka - prakticky libovolná změna v herních pravidlech by vyžadovala přestavění této hierarchie.

Problém dědičnosti tkví v tom, že se jedná o velmi nepružnou relaci, která je zadrátována ve zdrojovém kódu již při kompilaci. Navíc ne každá množina závislostí se dá vyjádřit jako orientovaný acyklický graf.

Trochu jiné řešení nabízí použití rozhraní, jak ukazuje obrázek 1.3. Každé rozhraní představuje nějakou vlastnost, kterou může příslušný objekt implementovat. Pokud bychom chtěli

vytvořit kouzlícího draka, implementoval by navíc rozhraní `IMagic` s metodou `CastSpell()`. Kdybychom vyžadovali pohybující se objekty, které s nikým neinteragují, implementovaly by pouze `IMovable`. Ty interagující by pak navíc implementovaly `ICollidable`.



Obrázek 1.3: Příklad použití rozhraní

I když je toto řešení z určitého hlediska o něco flexibilnější, naráží na řadu jiných problémů: rozhraní ze své podstaty žádnou implementaci neobsahují, jedná se pouze o předpis pojmenované funkcionality, kterou pak zprostředkovávají třídy toto rozhraní implementující.

V předchozím příkladě mohla být například metoda `CheckCollisions()` implementována pouze jednou ve třídě `DynamicObject` a ostatní objekty by tuto funkcionalitu převzaly. V tomto řešení by ji musely implementovat všechny tři třídy, což by pro vyhnutí se duplikaci vyžadovalo vytvoření obslužných tříd.

Dalším problémem, které žádný z předchozích dvou přístupů nedokáže elegantně řešit, je možnost proměny kouzelníka na draka či rozpadnutí útesu na několik částí - takových problémů tu můžeme najít celou řadu.

Výhodou objektového přístupu je bezesporu jeho jednoduchost a snadnost ladění, neboť vazby mezi všemi objekty jsou známy již během kompilace. Vytváření hierarchie objektů pomocí dědičnosti je navíc přirozené a intuitivní.

Nevýhodou je nízká flexibilita a špatná údržba. Herní konstrukty jsou navíc ponechány v režii programátora.

1.3.2 Komponentový přístup

Pojem *komponentová architektura* není v literatuře definován zcela jednoznačně. Různé články, které se této architektuře věnují, přizpůsobují její design konkrétnímu účelu, ke kterému je určena (např. automatizované systémy na letištích [7] či generické algoritmy [8]).

Z pohledu UML je komponenta definována jako *modulární část systému, která ukrývá svůj obsah a jejíž projev může být okolním prostředím nahrazen* [9].

Další definici poskytuje článek [8], který o komponentovém programování mluví jako o generickém návrhu knihoven, které mohou být aplikovány v libovolné doméně. Jako příklad uvádí knihovnu *Standard Template Library* jazyka C++, která obsahuje kontejnery, algoritmy, iterátory a alokátory, to vše nezávislé na typech objektů, se kterými pracují.

U her je situace trochu odlišná - komponenty zde nepředstavují části systému ale funkční bloky vztažené ke konkrétním entitám. Chování takové entity je pak definováno množinou komponent, které obsahuje.

U tohoto typu komponent můžeme vyjmenovat následující klíčové vlastnosti:

- Komponenta může být za běhu přidána či odstraněna
- Komponentu je možno použít opakovaně
- Komponenta zpravidla nezávisí na ostatních komponentách

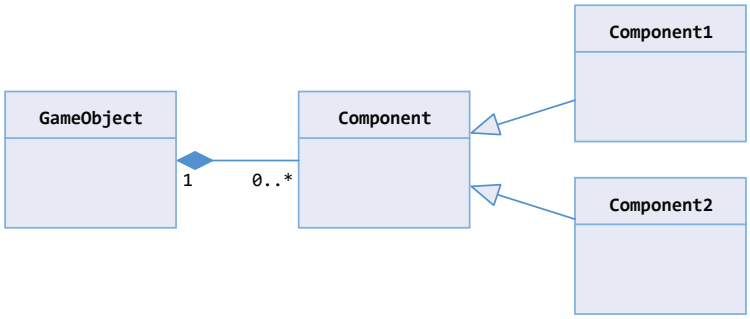
O komponentové architektuře použitelné pro vývoj her byla sepsána celá řada článků. V této části budou zmíněny dva z nich.

1.3.2.1 Varianta Nicolase Portera

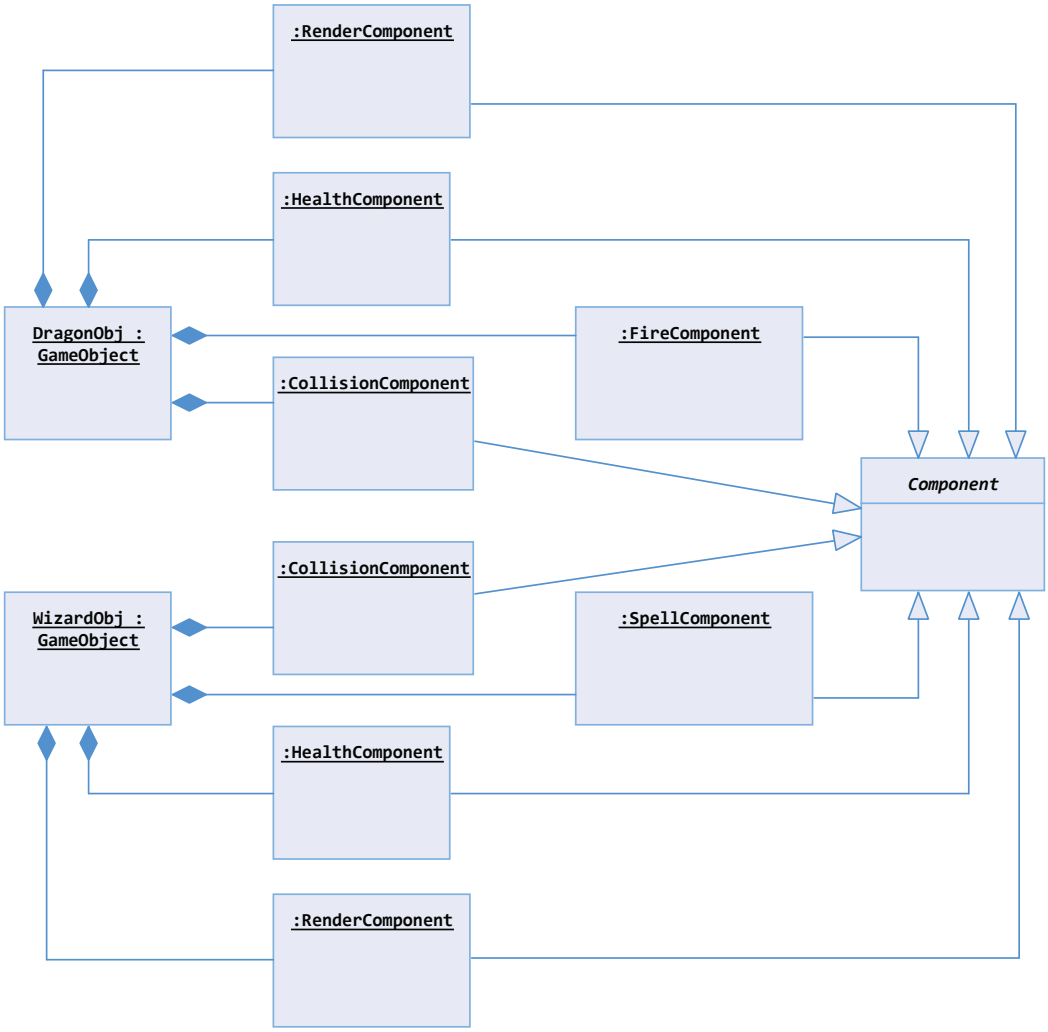
Nicolas Porter ve své práci [10] zmínil výhody a nevýhody jak komponentové, tak i hybridní architektury, využívající některé prvky objektové hierarchie. Herní objekt zde představuje pouhou přepravku komponent, které mezi sebou komunikují prostřednictvím zpráv.

Právě posílání zpráv je to, co umožňuje nahrazení libovolné komponenty okolním prostředím. Díky absenci přímého volání metod definuje chování komponenty pouze způsob její reakce na tyto zprávy.

Architektura je znázorněna na obrázku 1.4 a možná implementace příkladu s drakem a kouzelníkem na obrázku 1.5.



Obrázek 1.4: Komponentový přístup, varianta 1



Obrázek 1.5: Komponentový přístup, varianta 1: příklad

Ze systémového hlediska zde není žádný drak ani kouzelník, pouze dva pojmenované objekty typu `GameObject` s přiřazenými komponentami. Chrlení ohně u draka definuje komponenta `FireComponent`, sesílání kouzel u kouzelníka komponenta `SpellComponent`.

Oba objekty obsahují detektor kolizí `CollisionComponent`, komponentu udržující stav jejich zdraví `HealthComponent` a vykreslovací komponentu `RenderComponent`.

Komunikace probíhá tím způsobem, že příslušná komponenta pošle zprávu a ostatní na ni zareagují. Pokud například zasáhne kouzelník draka, `CollisionComponent` vyšle zprávu o zasažení objektu. Na tu zareaguje `HealthComponent`, která sníží hodnotu `health`.

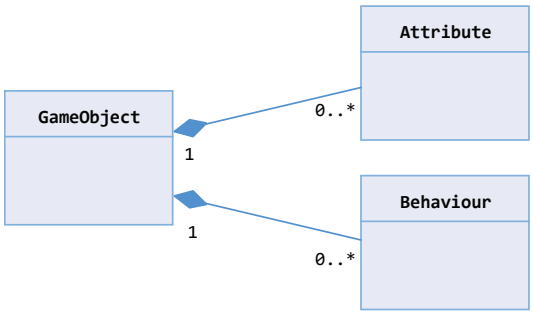
Všechny problémy, které byly diskutovány u objektového přístupu, se zde dají bez problémů řešit. Proměna kouzelníka ve draka by znamenala pouze prohození dvou komponent. Aby drak získal schopnost kouzlit, přidala by se mu komponenta `SpellComponent`.

Obrovskou výhodou tohoto přístupu je vysoká modularita a snadné přidávání nových funkcí. Na druhou stranu je tato hierarchie mnohem obtížnější na ladění, neboť je poskládána až za běhu. Systém posílání zpráv navíc povoluje, aby kdokoliv reagoval na jakoukoliv zprávu, což při špatném programátorském přístupu může znamenat pohromu.

1.3.2.2 Varianta Marcina Chady

Marcin Chady, který se vývoji her a umělé inteligenci věnuje již dlouhá léta, nastínil na jedné ze svých přednášek [11] možnou podobu komponentové architektury, obdobné jako u předchozího příkladu.

Komponentové třídy jsou pojmenovány jako `Behaviour` a na rozdíl od předchozího řešení jsou zde odděleny i atributy, což umožňuje jednotlivým komponentám sdílet data a nejen to, tato data mohou být k herním objektům libovolně přiřazována.



Obrázek 1.6: Komponentový přístup, varianta 2

V tomto případě tedy dochází k modularizaci obojího - funkcionality i dat. U příkladu s drakem a kouzelníkem by to vypadalo podobně jako u předchozího návrhu s tím rozdílem, že atribut `health` by byl přiřazen k herním objektům a komponenta by se k němu dostala například zavoláním metody `GetAttribute(HEALTH)`.

Protože posílání zpráv je mnohem náročnější na výkon než přímé volání metod, je zde diskutováno několik způsobů optimalizace. Tím nejjednodušším je tzv. *subscribe* technika, kdy komponenta projeví zájem o zasílání konkrétního typu zprávy tím, že se zaregistruje jako *subscriber*.

1.3.2.3 Shrnutí

Komponentový přístup se vydává zcela jinou cestou než ten objektový. Jeho modularita umožňuje přímo za běhu vytváření nových atributů, ale také celých objektů, neboť je každý objekt definován pouze sadou atributů a komponent, které obsahuje. Dědičnost je zde nahrazena kompozicí a celá architektura je tak datově orientovaná.

Polymorfismus má zde zcela odlišný význam - zatímco u objektového přístupu je pomocí rozhraní možno modularizovat chování komponenty podle konkrétního typu, který dané rozhraní implementuje, zde je polymorfismem míněn samotný způsob zpracování přijatých zpráv.

Modularita je největší výhodou, ale zároveň největší slabinou komponentového přístupu. Protože se celá architektura komponent a objektů sestavuje až v době běhu, není možné špatně sestavenou hierarchii odhalit v době kompilace.

Pokud u objektového přístupu použije programátor na některém místě objekt, který neimplementuje potřebné rozhraní, kompilátor vypíše chybu. U komponentového přístupu by taková chyba nastala podstrčením nesprávné komponenty a aplikace by se sice zkompilovala, ale nemusela by se chovat korektně.

1.4 Analýza herních enginů

Než dojde k vlastnímu návrhu komponentové architektury, který bude součástí herního enginu, je třeba nejprve vymezit, čím se typický herní engine vyznačuje.

Herní engine můžeme definovat jako sadu nástrojů a knihoven, obsahujících obecnou funkcionalitu pro vývoj her. Různé úrovně abstrakce této programové základny pak udávají použitelnost a sofistikovanost celého enginu, přičemž klíčovým faktorem je jeho flexibilita.

Julian Gold, softwarový inženýr v Microsoft Research, definoval herní engine takto: *„Herní engine je množina modulů a rozhraní, umožňující vývojovému týmu soustředit se na samotný gameplay nežli na technické aspekty hry“* [12].

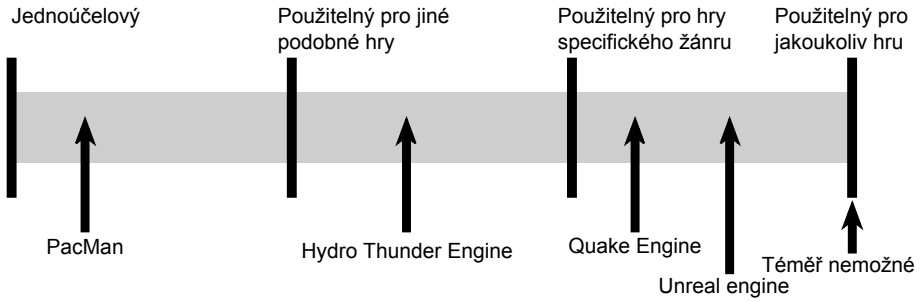
V knize [4] je o něm zase pojednáváno jako o té části hry, která nezprostředkovává *gameplay* ani s ním nijak nesouvisí.

Na světě existuje celá řada herních enginů. Většina z nich se zaměřuje jen na konkrétní herní žánry (např. *Unreal Engine*), jiné se snaží svou univerzálností uspokojit potřeby co nejširšího kruhu vývojářů (*Unity, C4 engine*). Volba vhodného enginu pro vývoj konkrétní hry tak může být poměrně tvrdý oříšek, neboť v pokročilé fázi vývoje se na jinou technologii bude přecházet jen těžko. Důležitým prvkem při výběru nejsou jen technologické aspekty dané aplikace, ale také programovací jazyk, cena za licenci, podpora a velikost komunity.

Jak bylo poznamenáno již v úvodu, čím je engine univerzálnější, tím méně vhodným se stává pro konkrétní typ hry. Například rozhraní pro zápasy bude vypadat zcela jinak než rozhraní pro hry závodní. U masivních online her (MMORPG) se klade důraz na synchronizaci herního modelu po síti s co nejmenší latencí, u adventur je potřeba zpracovávat dialogové stromy pro interakci s jednotlivými postavami, hry typu *First Person Shooter* (FPS) kladou důraz na fotorealistické vykreslení herního prostředí, ve strategických hrách zase obvykle dochází k vykreslování velkého množství objektů najednou.

Jason Gregory ve své knize *Game Engine Architecture* nastínil problematiku modularity enginů, jehož interpretace je znázorněna na obrázku 1.7.

Vytvořit herní engine, který by splňoval požadavky všech typů her, je stejné jako snažit se vytvořit jednotnou aplikaci pro úpravu bitmap, vytváření 3D modelů, vektorových animací, střih videa a generování syntetické hudby. Taková aplikace by se velmi těžko ovládala, nehledě na to, že by nikdo nedokázal využít potenciál všech funkcí v rámci jednoho projektu.



Obrázek 1.7: Modularita enginů

V následující části budou popsány vybrané herní enginy z kategorie multiplatformních enginů s podporou pro mobilní systémy. Diskutována bude také podobnost s komponentovou architekturou.

1.4.1 Marmalade

Marmalade je považován za nejrychlejší C++ multiplatformní engine. Aplikace je možno vyvíjet přímo ve vývojovém prostředí Visual Studio, ve kterém se nalinkuje *Marmalade SDK*. Součástí tohoto balíčku jsou knihovny, ukázkové kódy, dokumentace a nástroje pro vývoj.

Vývoj nativně v jazyce C++ navíc umožňuje snadnou integraci knihoven třetích stran. Skriptovacím jazykem je zde Lua.

Tento engine je určen především pro vývojáře, kteří chtějí mít nade všemi prostředky plnou kontrolu (správa paměti, optimalizace). Díky absenci frameworků jako je tomu v případě *Mono* u Unity má instalační APK soubor pro systém Android s prázdnou scénou pouhých 1.2 MB.

Protože zde nevznikají žádné logické vazby mimo programovací kód (na rozdíl od Unity, kde se vše nastavuje v editoru), může vývojář bez větších obtíží provést migraci na jinou technologii i v pozdější fázi vývoje.

Engine není striktně komponentově orientovaný. Obsahuje sice sofistikovaný manažer herní scény, ale způsob návrhu vlastních projektů nechává na vývojářích. Dá se říct, že se jedná spíše o sadu nástrojů (*toolkit*) a abstraktní vrstvu, která vše spojuje dohromady.

```
CSprite* background = new CSprite(); // Create background sprite
background->m_X = (float)IwGxGetScreenWidth() / 2;
background->m_Y = (float)IwGxGetScreenHeight() / 2;
background->SetImage(g_pResources->getGameBG());
background->m_W = background->GetImage()->GetWidth();
background->m_H = background->GetImage()->GetHeight();
```

Ukázka kódu Marmalade

1.4.2 Unity

Již v době svého vzniku v roce 2005 se stal tento engine velice oblíbeným. To především díky robustnímu editoru, ve kterém je možné přímo za běhu zobrazit herní scénu se všemi objekty, manipulovat s nimi a měnit jejich atributy.

Nejnovější verze 5 navíc přinesla plnou podporu pro 2D hry, jejichž vývoj byl až do této verze poněkud těžkopádný.

V Unity bylo vytvořeno mnoho populárních her. Mezi ty známé patří například *Deus Ex*, *Kerbal Space Program* a *Zombieville*.

Hlavním skriptovacím jazykem je zde C#, který běží pod frameworkem *Mono*. Díky tomu je zde možné využívat technologie jako LINQ či lambda výrazy. Dalšími podporovanými jazyky jsou JavaScript a Boo.

Během analýzy byla vytvořena testovací aplikace s jednou scénou bez jediného objektu. Velikost instalačního APK souboru pro platformu Android dosahovala 18.5 MB.

O Unity lze mluvit jako o komponentovém systému - každý herní objekt má přiřazenu množinu komponent a skriptů, které programuje vývojář. Skript zde připomíná `Behaviour` komponenty z článku Marcina Chady, které definují chování příslušného herního objektu. Také se tu odehrává komunikace prostřednictvím zasílání zpráv, ta ale probíhá přes reflexi jazyka C#, kdy odesílatel specifikuje název metody, která má být zavolána nad každou komponentou daného objektu. Svým způsobem se jedná o kompromis mezi zmíněnou *subscribe* technikou a přímým voláním metod.

I když Unity vykazuje některé aspekty komponentové architektury, ztrácí se zde její největší výhoda, a tou je modularita. Jednotlivé komponenty mezi sebou mohou komunikovat napřímo a v exemplárních příkladech, které tvůrci systému připravili, se tak hojně děje. Patrně tak chtějí nabádat začínající programátory k objektovému přístupu, kde je ladění chyb snazší.

```
//Awake is always called before any Start functions
void Awake()
{
   //Get a component reference to the attached BoardManager script
   boardScript = GetComponent<BoardManager>();
      boardScript.foodCount++;
      FoodController.SendMessage("OnFoodAdded", 1);
      //Call the InitGame function to initialize the first level
      InitGame();
}
```

Ukázka kódu v Unity

1.4.3 Atomic Game Engine

Jedná se o poměrně nový multiplatformní engine, určený pro 2D i 3D hry. Stejně jako Unity obsahuje editor, ten však slouží spíše jako správce zdrojových souborů a médií, případně pro vytváření herních scén. Jeho nespornou výhodou je fakt, že nabízí ke stažení kompletní zdrojové kódy.

Kromě nativního jazyka C++ podporuje také C#, JavaScript a TypeScript. Engine samotný je napsaný v C++. Velikost instalačního APK souboru s prázdnou scénou dosahuje příjemných 4.6 MB.

Na rozdíl od Unity jsou zde skripty spravovány odděleně od herních objektů. Práce s komponentami je podobná - každý objekt má přiřazenu sadu komponent (`MassComponent`, `PhysicsComponent`, ...), komunikace však probíhá přímým voláním metod. Zasílání zpráv je zde nahrazeno registrací globálních událostí.

```
self.start = function() {
    self.rigidBody = self.getComponent("RigidBody2D");
    self.subscribeToEvent("PhysicsBeginContact2D", function(data){
        //get an collidable object
        var other = (data.nodeA == self.node) ? data.nodeB : data.nodeA;
        //check collision for a brick and remove it
        if (other.name.indexOf("Brick") > -1) Atomic.destroy(other);
    });
}
```

Ukázka kódu v Atomic Game Engine

1.4.4 Porovnání enginů

Následující tabulka zobrazuje seznam diskutovaných enginů, programovací jazyky a podporované platformy.

Engine	**Jazyk**	**Platforma**	**Komponentová architektura**
Marmalade	C++, Lua	Android, iOS, OSX, Windows	ne
Unity	C#, JavaScript, Boo	Android, Browser, iOS, Linux, OSX, Windows	ano
Atomic Game Engine	C++, C#, JavaScript	Android, Browser, iOS, Linux, OSX, Windows	částečně

Tabulka 1.1: Porovnání herních enginů

Atomic Game Engine i Unity jsou robustní nástroje, které však vyžadují, aby se jim vývojář přizpůsobil. Vývojové prostředí je nutné používat ve spolupráci s editorem a některé druhy vazeb je možno definovat pouze v tomto editoru. Atomic Game Engine má tu výhodu v podobě otevřeného kódu, kdy vývojář může nahlédnout dovnitř a snadněji tak porozumí architektuře celého systému, což Unity neumožňuje.

U Unity je znatelná jeho snaha o univerzálnost, která se podepisuje také na velikosti cílového projektu, jak bylo uvedeno v předchozí části.

Marmalade zde vystupuje spíše jako sada nástrojů než jako klasický herní engine. Na jednu stranu to přidělává vývojářům práci, jelikož jsou tak nuceni si spoustu věcí naprogramovat sami, na druhou stranu nejsou tolik svázaní architekturou systému.

1.5 Vlastní řešení

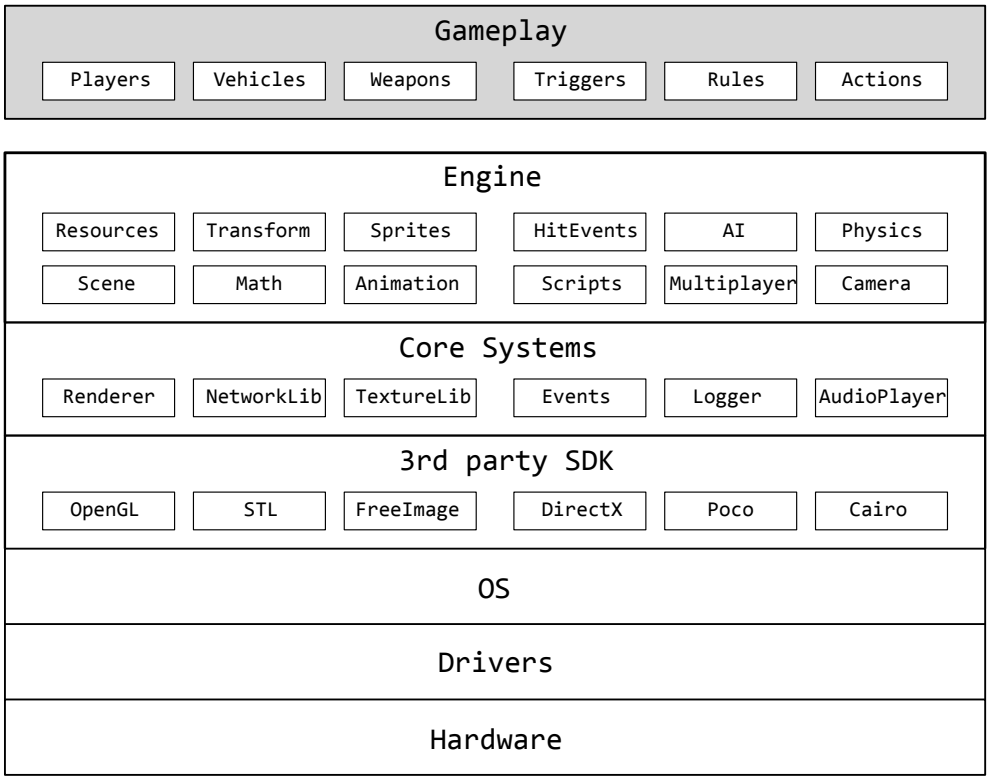
I když různé žánry her využívají různé nástroje, podíváme-li se na ně jako na simulátory fiktivního světa, nalezneme mnoho problémových domén, které jsou na žánrech nezávislé - téměř každá hra potřebuje nějakou knihovnu pro vizualizaci animací, přehrávač zvuků, správce dat (*assetů*) apod. Diagram na obrázku 1.8, který částečně vychází z diagramu v knize [4], definuje sedm vrstev herní architektury.

Na nejspodnější vrstvě se nachází samotný hardware, ovladače a operační systém. Poté následují knihovny třetích stran pro nízkoúrovňovou práci s grafikou a assety obecně. Sem můžeme zařadit např. grafickou knihovnu *glut*, knihovnu pro práci s jazykem XML *tinyxml* nebo *Standard Template Library* jazyka C++.

Poté následuje pátá vrstva, která představuje sadu nástrojů pro snadnou spolupráci mezi čtvrtou a šestou vrstvou - systém vstupních a výstupních událostí, síťová knihovna s možnostmi serializace objektů, komponenta pro načítání multimédií a jejich přehrávání či vykreslování.

Šestou vrstvou je samotný herní engine, obsahující vše, co potřebuje vrstva sedmá, kterou je samotná hra. Je zde správce herní scény, kamery, synchronizační komponenta pro multiplayer, fyzikální engine, algoritmy umělé inteligence, animační engine a transformace.

Šestá vrstva by ideálně měla obsahovat vše, co nepatří do sedmé vrstvy, ale je touto vrstvou vyžadováno, tedy vše, co je možno využít pro více než jednu hru.



Obrázek 1.8: Vrstvy herního enginu

Engine, který je součástí této práce, bude vznikat společně s prototypem hry, díky čemuž zde může proběhnout jakási průběžná korelace mezi šestou a sedmou vrstvou.

Základem bude komponentová architektura, na které bude založena struktura herní scény. Tato architektura v sobě zkombinuje některé přístupy z části 1.3.2 a využije také poznatky z analýzy existujících řešení, především z enginu Unity, kde je tato architektura zaintegrována do herní scény.

V průběhu vývoje hry pak vzniknou požadavky na další funkcionalitu a potřebnou sadu komponent, které budou postupně utvářet výslednou podobu enginu.

Ve výsledku by měl nabídnout nástroje pro vykreslování, přehrávání zvuků a hudby, přenos dat a synchronizaci po síti, ale také grafové algoritmy pro hledání cest, stavové automaty a jiné konstrukty z oblasti umělé inteligence.

Aby nemusel stavět na zelené louce a pracovat s knihovnami nižších úrovní jako je *glut* či *FreeImage*, pokryje engine šestou vrstvu zmíněné architektury. Prototyp hry bude představovat vrstvu sedmou a vše ostatní obstará multimediální framework.

Oním frameworkem je OpenFrameworks, o kterém bude řeč nyní.

1.5.1 OpenFrameworks

OpenFrameworks [13] není ani tak herní engine jako komplexní sada nástrojů pro vývoj real-time systémů v jazyce C++, zaměřených na generování a zpracování zvuku či grafiky. Klade si za cíl být kolaborativní, jednoduchý, konzistentní, intuitivní a snadno rozšiřitelný. Řídí se tzv. DIWO (*Do it with others*) filozofií.

Podporovanými platformami jsou Windows, Mac OS X, Linux, iOS, Android a od verze `0.9.x` také HTML5. Dá se považovat za alternativu k frameworku *Processing* [14], který má podobné zaměření, programovacím jazykem je však Java a nepodporuje tolik platforem.

OpenFrameworks obsahuje celou řadu knihoven jako *FreeImage* a *OpenCV* pro zpracování obrazu či *glew* pro přístup k OpenGL rozhraní. To vše je slepené dohromady sadou komponent, tvořících abstraktní vrstvu nad těmito knihovnami.

Tato abstrakce zjednodušuje vývojářům práci s jednotlivými technologiemi jako je analýza zvuku, *computer vision* či přenos dat po síti. Kromě toho bylo pro tento framework vytvořeno více než tisíc doplňků [15], které programovou základnu ještě rozšiřují.

Možnost integrace vlastního enginu jako OpenFrameworks doplňku byl jeden z důvodů, proč byl tento framework vybrán. V současné době není k dispozici téměř žádný doplněk věnující se tomuto tématu a proto by se pro tento framework mohl herní engine stát významným přínosem, neboť by zde byla možnost jednoduchého rozšiřování prostřednictvím integrace s dalšími doplňky.

Framework byl vytvořen v roce 2005, současná verze v době psaní této práce je `0.9.3`, která v porovnání s verzemi `0.8.x` obsahuje mnoho výrazných změn - došlo k předělání celého jádra a přibyla podpora pro export do HTML5 s využitím technologie *Emscripten*. Nevýhodou těchto změn je fakt, že nová verze již není plně kompatibilní se staršími projekty a pro vývoj vyžaduje kompilátor podporující standard C++ 14.

Framework je šířen jako open-source pod MIT licencí a je možno jej využít pro vývoj komerčních projektů.

Mezi nejznámější projekty vytvořené v OpenFrameworks patří interaktivní instalace *Body Paint*, mapovací projekce *No_Thing* a generátor textilních vzorků *Pixtil*. Na poli vývoje her je zřejmě nejznámější arkáda *Ridiculous Fishing* [16].

Kapitola 2

Engine

Navržený engine byl pojmenován *CogEngine*, což je zkratka slov *Component Oriented Game Engine*. V první fázi bude navržen jednoduchý prototyp na základě předchozích zjištění z různých komponentových architektur, který bude obsahovat vše potřebné k definici herní scény.

Jelikož je součástí práce i vytvoření prototypu hry, která poběží pod tímto enginem, bude probíhat vývoj druhé fáze, obsahující již grafické rozhraní, společně s tímto prototypem a velká část rozšiřitelných komponent bude navržena právě dle potřeb této hry.

2.1 Požadavky na aplikaci

Na základě poznatků z předchozí kapitoly byly definovány funkční a systémové požadavky pro oba projekty.

2.1.1 Funkční požadavky

2.1.1.1 Funkční požadavky na engine

- **FP01: Engine bude využívat komponentový přístup**

  Všechny stěžejní části aplikace budou implementovány jako komponenty, které mezi sebou budou komunikovat prostřednictvím zpráv. Chování herního objektu pak budou definovat komponenty, které bude obsahovat.

- **FP02: Engine bude obsahovat graf herní scény**

  Herní scénu bude tvořit stromová struktura uzlů, kde se transformace každého uzlu bude odvíjet od transformace jeho předků. Herní uzel pak může představovat cokoliv od zobrazitelných objektů až po spouštěče událostí (*triggery*) či kontejnery na atributy a funkce.

- **FP03: Engine bude umožňovat psaní skriptů ve skriptovacím jazyce**

  Díky skriptovacímu jazyku bude možno naprogramovat chování objektů bez nutnosti neustálé rekompilace.

- **FP04: Herní scénu bude možno definovat v konfiguračních souborech XML**

  Zatímco pomocí skriptů bude možno definovat chování objektů, jazyk XML bude sloužit pro definici atributů jako je transformace, animace a konfigurační proměnné.

- **FP05: Engine bude obsahovat komponentu pro vykreslování sprite sheetů**

  Sprite sheet je textura obsahující jiné textury. Aby grafická karta nemusela v rámci jedné scény vykreslovat každý objekt zvlášť, pošle se do ní v rámci jednoho vykreslovacího kroku sprite sheet a seznam souřadnic deklarujících, které pod-textury (sprity) mají být vykresleny a kde, což významně snižuje časovou složitost zpracování.

- **FP06: Engine bude obsahovat komponentu pro přehrávání animací**

  Animace, ať už obrázkové či transformační, bude možno definovat staticky v konfiguračním XML souboru. Konfigurovatelná bude rovněž rychlost jejich přehrávání a bude je možno libovolně poskládat i vnořovat.

- **FP07: Engine bude umožňovat synchronizaci dat po síti**

  Každá hra obsahující multiplayer potřebuje nějakým způsobem synchronizovat svůj model mezi několika zařízeními. Engine nabídne komponenty, které se postarají jak o navázání spojení, tak o serializaci a deserializaci zpráv do proudu bajtů a synchronizaci spojitých veličin pomocí interpolace.

- **FP08: Engine bude obsahovat komponenty pro umělou inteligenci**

  Pro vývoj her je velice často využívanou komponentou algoritmus A* na hledání cest. Pro přesun objektů po mapě je také potřeba animovat jejich pohyb, k čemuž poslouží tzv. *steering behaviors*, což je sada funkcí pro simulaci realistického pohybu. Dále engine nabídne stavový automat a algoritmy prohledávání stavového prostoru, které je možno použít například pro strategická rozhodování.

- **FP09: Engine bude umožňovat přehrávání hudby a zvuků**

- **FP10 Engine bude umožňovat ukládání dat do databáze**

  Engine využije standardní knihovnu *SQLite*, ke které pouze zprostředkuje tenkou vrstvu pro jednodušší práci s databázovými objekty.

2.1.1.2 Funkční požadavky na prototyp hry

- **FG01: Hra bude spadat do žánru RTS**

  Bude se jednat o strategii, odehrávající se na nějaké dvourozměrné mapě, ve které bude cílem porazit protihráče prováděním různých akcí.

- **FG02: Hra bude využívat algoritmy umělé inteligence**

  Ve hře se budou vyskytovat autonomní jednotky, které budou využívat simulátory pohybu (*steering behaviors*) a algoritmy hledání cest na mapě. Pro strategická rozhodnutí, vedoucí k poražení protihráče, budou použity randomizované algoritmy prohledávání stavového prostoru.

- **FG03: Hra bude obsahovat multiplayer pro dva hráče**

  Každý hráč bude hrát na svém vlastním zařízení, která spolu budou komunikovat po síti. Jednotlivé komponenty pak zajistí synchronizaci herních modelů a hladké přechody spojitých veličin (transformace) na základě interpolace hodnot z aktualizačních zpráv.

2.1.2 Obecné požadavky

- **OP01: Engine i hra budou napsány v jazyce C++**

  Kompilace do nativního kódu přinese enginu mnoho výhod, především z výkonnostního hlediska.

- **OP02: Prototyp hry bude zkompilovatelný pro Windows a Android**

  Ačkoliv bude možno oba projekty zkompilovat pro obě platformy, z pohledu *user experience* bude prototyp hry optimalizován především pro platformu Android, jelikož bude cílit na uživatele mobilních zařízení. Hratelný bude ale i pod Windows. Minimální podporovaná verze pro systém Android bude 4.2 (API 17).

- **OP03: Engine bude moci být distribuován jako oficiální doplněk frameworku OpenFrameworks**

  OpenFrameworks zde bude sloužit jako programová základna pro architekturu enginu. Poskytne základní nástroje pro zpracování herní smyčky, registraci obslužných metod na vstupy od uživatele, práci s fonty a multimediálními soubory.

  Díky distribuci jako doplněk OpenFrameworks bude engine snadno přístupný ostatním vývojářům pro tento framework vyvíjející a budou tak moci poskytnout zpětnou vazbu pro budoucí rozšiřování.

- **OP04: Engine i hra budou šířeny pod MIT licencí**

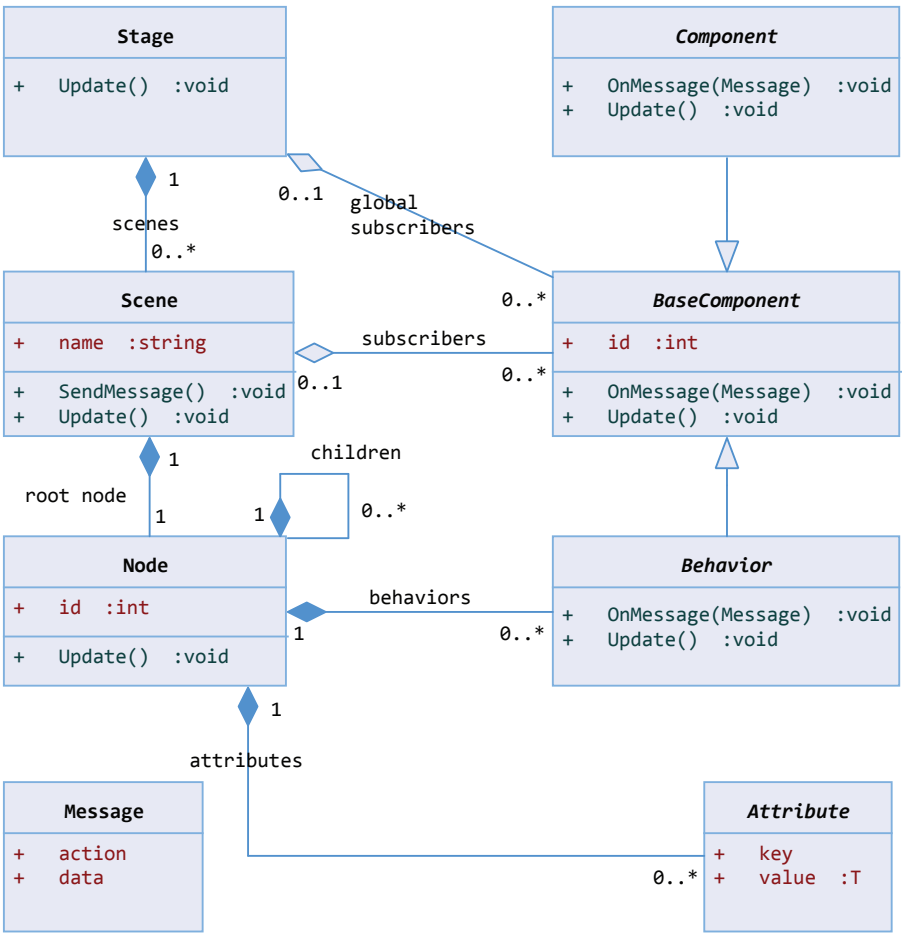
2.2 Návrh enginu

2.2.1 Návrh komponentové architektury

Návrh architektury vychází z poznatků získaných během analýzy existujících řešení. Prostředí je rozděleno na několik scén, tvořících stromovou strukturu entit, které v dané scéně figurují. Tyto objekty obsahují kolekci atributů a chování, jak ukazuje diagram 2.1.

Základním prvkem je zde třída `Node`, představující objekt v herní scéně se sadou atributů, reprezentovaných třídou `Attribute`. Kolekce jednotlivých scén pak tvoří dohromady `Stage`, která bude mít na starosti jejich aktualizaci a přepínání. Komponenty jsou zde dvojího druhu - třída `Component` pro globální komponenty, existující po celý běh aplikace, a `Behavior` pro komponenty přiřazené ke konkrétnímu hernímu objektu.

Zatímco ty globální budou mít na starosti aktivity týkající se celé aplikace, například komunikace s databází, ty lokální budou svým způsobem definovat chování daného objektu, například reakce tlačítka na stisk.



Obrázek 2.1: Návrh architektury herní scény

Obě dvě třídy `Component` a `Behavior` jsou navrženy jako abstraktní. Teprve jejich konkrétní implementace (např. `HealthBehavior`) bude definovat jejich význam.

Chování daného objektu `Node` tedy definuje kolekce příslušných `Behavior` objektů a sada atributů. Samotný objekt `Node` žádnou logiku nevykonává, což z něj činí prostý kontejner.

Vzhledem k důležitosti pochopení rozdílu mezi globální a lokální komponentou bude v následujících částech práce použita následující terminologie:

- pojem *komponenta* označuje globální komponentu (třída `Component`). Může existovat po celou dobu běhu aplikace a není závislá na žádném herním objektu.
- pojem komponenta (bez kurzívy) značí komponentu z pohledu UML
- pojem *behavior* označuje komponentu, která je součástí herní scény (třída `Behavior`) a vztahuje se ke konkrétnímu objektu. *Behavior* nemůže existovat vně herní scény.

2.2.1.1 Aktualizace modelu

Všechny třídy herní scény mají metodu `Update`. Tato metoda bude volána v rámci aktualizace herní smyčky - nejprve se zavolá u třídy `Stage`, která pak pomocí iterátoru projde všechny aktivní scény a globální komponenty, které zaktualizuje.

Každá scéna pak zavolá metodu `Update` nad kořenovým uzlem `Node`. Ten pak totéž provede u všech svých `Behavior` a nad svými potomky. Takto se postupně zaktualizuje celý herní strom.

Zda budou jednotlivé *komponenty* či *behaviors* tuto metodu implementovat, záleží na jejich účelu. Například *behavior* detekující kolizi objektů ji implementovat bude, jelikož bude provádět opakovaně to samé ověřování, zatímco *behavior* upravující hodnotu zdraví (atribut `Health`) podle situace na herní scéně ji implementovat nebude.

2.2.1.2 Posílání zpráv

Posílání zpráv je navrženo tak, aby měl každý potomek třídy `BaseComponent` možnost sám deklarovat zájem o posílání zpráv daného typu tím, že se zaregistruje jako *subscriber*. *Komponenty* se budou registrovat ve třídě `Stage` a *behaviors* budou zaregistrovány ve scéně, ke které náleží jejich objekt `Node`.

Jakmile dojde k zaslání zprávy, vybere scéna či `Stage` objekty, které se zaregistrovaly jako posluchači pro zprávy daného typu. Tento typ bude ve zprávě `Message` představovat proměnná `action`. Poté se nad každým posluchačem zavolá metoda `OnMessage`, ve které bude implementována reakce na tuto zprávu.

Zpráva může obsahovat libovolná data (proměnná `data`).

2.2.1.3 Atributy

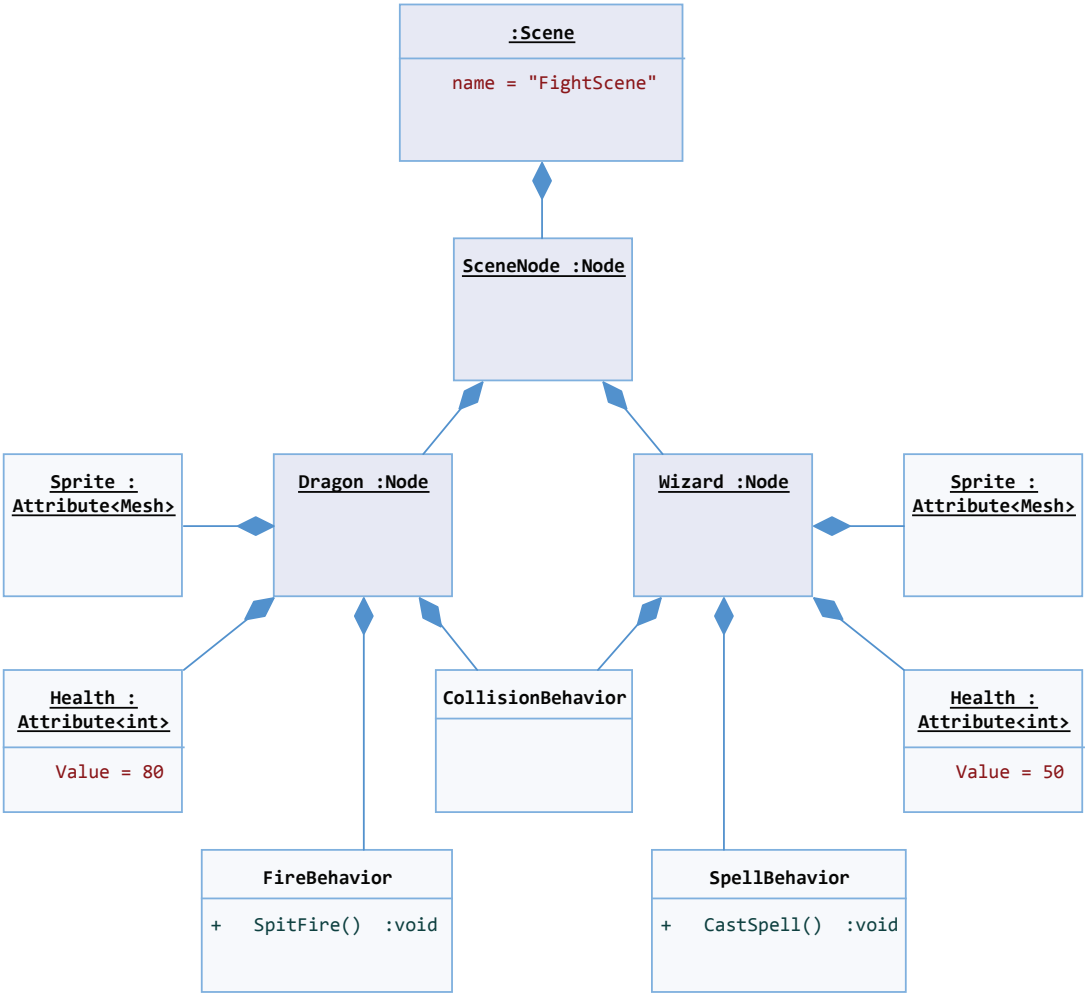
Všechny atributy daného objektu `Node` budou obsaženy ve zvláštní kolekci generických objektů `Attribute`, identifikované nějakým klíčem.

Do těchto atributů se budou ukládat všechny sdílené proměnné. Ostatní proměnné mohou být definovány jako klasické proměnné tříd u konkrétních implementací `Behavior`. Například proměnnou `Health`, obsahující údaje o zdraví nějaké postavy, je vhodné uložit do kolekce atributů, zatímco `StartPosition`, představující počáteční pozici transformace nějaké animace `AnimBehavior`, může být uložena jako členská proměnná třídy `AnimBehavior`, jelikož ji žádné další *behavior* využívat nebude.

2.2.1.4 Příklad scény

Následující diagram znázorňuje, jak by vypadala dle tohoto návrhu scéna s drakem a kouzelníkem. Pro přehlednost jsou zde vynechány některé objekty jako útes či pozadí a společný předek `Behavior` jednotlivých *behaviors*.

Herní scéna obsahuje kořenový uzel s názvem `SceneNode` se dvěma potomky. Grafické objekty jsou ukryty v atributech `Sprite`. Oba objekty mají hodnotu zdraví v atributu `Health` a jejich herní logiku definují behaviors `FireBehavior` pro draka, resp. `SpellBehavior` pro kouzelníka.



Obrázek 2.2: Příklad scény

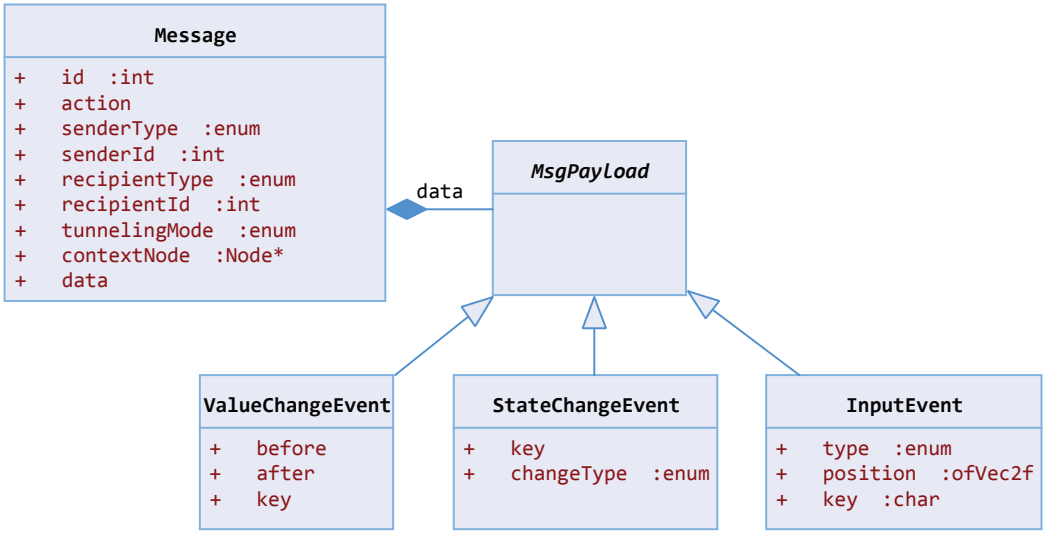
Pokud bychom chtěli, aby hráč ovládal kouzelníka, vytvořili bychom novou *komponentu* s názvem např. `InputAction`, která bude mapovat uživatelský vstup na konkrétní akci.

Jakmile by hráč stiskl určitou kombinaci kláves, odeslala by se zpráva o této akci, kterou by zachytil `SpellBehavior`.

Pokud by daná akce byla validní (kouzelník je schopen kouzlit), vytvořil by se nový objekt `Node` (s názvem např. `Fireball`), představující vystřelenou ohnivou kouli.

2.2.2 Posílání zpráv

Diagram 2.3 znázorňuje návrh třídy `Message`. V rámci zasílacího procesu se vytvoří nová instance, která bude po odeslání zahozena. Hodnoty jednotlivých atributů pak budou určovat samotný způsob zaslání.



Obrázek 2.3: Návrh zprávy

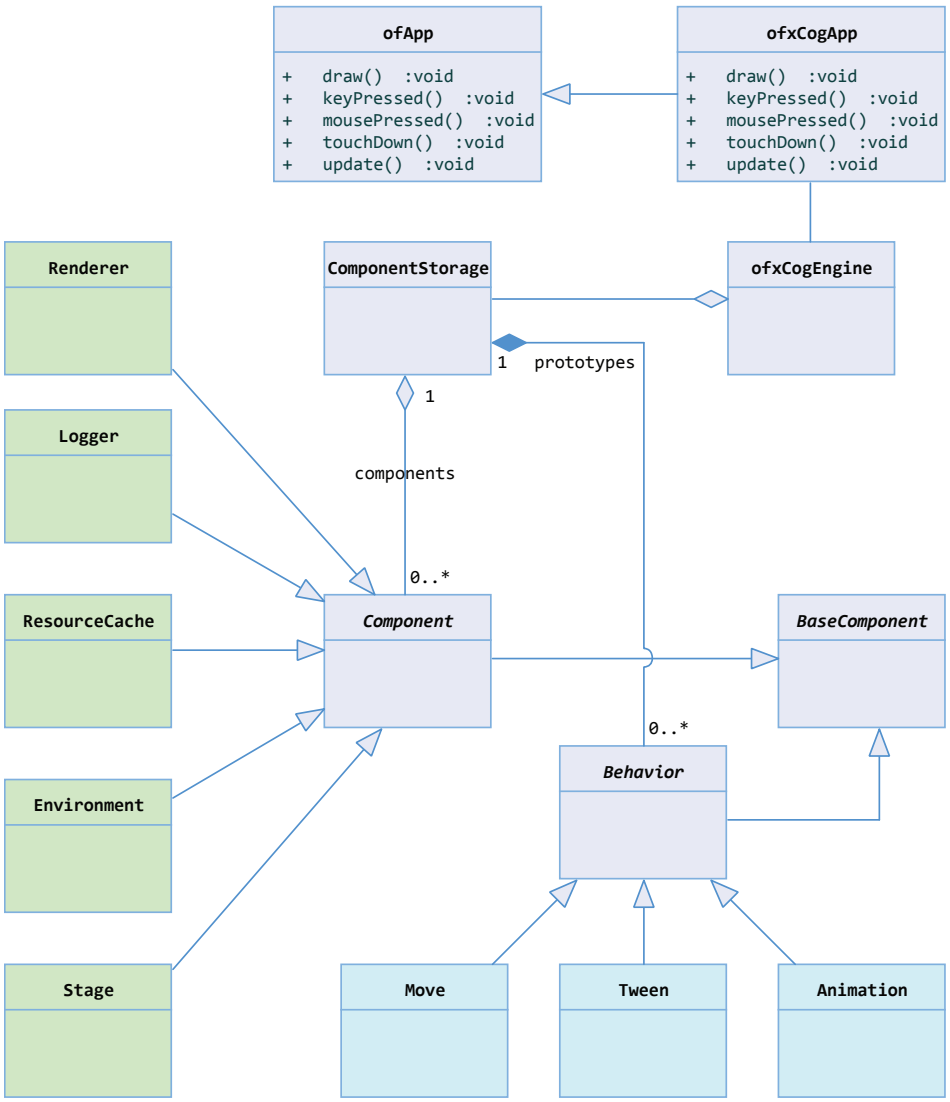
Atributy:

- **id** - identifikátor, inkrementován s každou novou instancí
- **action** - typ zprávy, akce či události
- **senderType** - typ odesílatele (*behavior*, *komponenta* nebo externí třída)
- **senderId** - identifikátor odesílatele (pokud nějaký má)
- **recipientType** - typ příjemce. Pomocí výčtu zde bude možno konfigurovat, zda se má zpráva poslat jen *komponentám* a *behaviors*, které mají danou akci zaregistrovanou, nebo bude traverzovat herním stromem od nějakého výchozího uzlu a postupně bude zaslána všem zaregistrovaným objektům. Tento přístup je vhodný především u zpráv, kde je vyžadována určitá posloupnost příjemců (např. pokud chceme zaslat zprávu potomkům konkrétního uzlu).
- **recipientId** - identifikátor příjemce; tento atribut bude nastavený pouze v případě, že bude zpráva určena jednomu konkrétnímu příjemci.
- **tunnelingMode** - v případě, že bude zpráva traverzovat herním stromem a postupně bude zaslána jednotlivým objektům, hodnota atributu **tunnelingMode** stanoví, zda se mají posílat směrem od listů (*bubbling*) nebo od kořene (*tunneling*). Tento mechanismus používá např. technologie WPF pro určení posloupnosti zpracování událostí.
- **contextNode** - uzel, kterého se zpráva týká.
- **data** - tělo zprávy. Diagram obsahuje příklad tří možných tříd - `ValueChangeEvent` pro změnu hodnoty nějakého atributu, `StateChangeEvent` pro změnu stavu a `InputEvent` pro událost vstupního zařízení jako stisk myši či klávesy.

2.2.3 Architektura enginu

Během návrhu architektury enginu bylo potřeba vyřešit, jakým způsobem bude engine napojen na samotný OpenFrameworks. V něm je definována abstraktní třída `ofBaseApp` pro Windows, resp. `ofxAndroiApp` pro Android. Tuto třídu je nutné podědit a implementovat metody jako `setup()` pro nastavení frameworku, `update()` pro aktualizaci herní smyčky a `draw()` pro vykreslování. Také jsou zde metody na ošetření vstupu od uživatele jako `mousePressed()` či `keyPressed()`.

Následující diagram ukazuje návrh architektury jádra enginu. Obě zmíněné třídy zde implementuje třída `ofxCogApp`, která přeposílá data třídě `ofxCogEngine` a ta zase příslušným komponentám.



Obrázek 2.4: Návrh architektury enginu

OpenFrameworks vyžaduje pro své addony konvenci pojmenování hlavních tříd s prefixem *ofx*. Z tohoto důvodu budou dvě hlavní třídy - `ofxCogApp` a `ofxCogEngine`, přes které bude možno přistupovat ke všem ostatním, pojmenovány dle této konvence.

Třída `Stage` a s ní i celá herní scéna zde figuruje jen jako jedna z *komponent*, díky čemuž se z třídy `ofxCogEngine` stává prostý delegát, představující společně s třídou `ofxCogApp` jakýsi most mezi událostmi, které vyvolává OpenFrameworks, a ostatními částmi aplikace.

V diagramu jsou popsány také některé *komponenty* (zelenou barvou), důležité pro běh enginu. `Renderer` bude mít na starosti vykreslování, `Logger` logování zpráv, `ResourceCache` bude uchovávat *assety* jako obrázky, hudbu a textury a `Environment` poskytne informace o aktuálně stisknutých klávesách, pozici myši či velikosti okna.

Pro názornost obsahuje diagram také některé typy *behaviors* - `Move` pro pohyb objektů na základě nějaké výslednice sil, `Tween` pro tweening (přesouvání) scén a `Animation` pro spouštění animací.

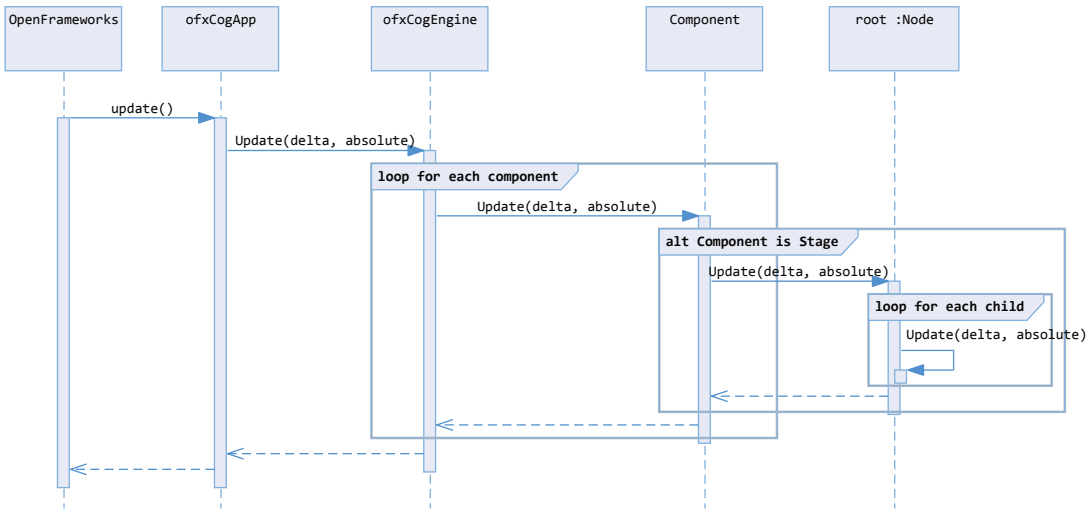
Pro vytvoření nové *komponenty*, resp. *behavior*, bude potřeba pouze přidat novou třídu a zaregistrovat jí do `ComponentStorage`.

2.2.4 Aktualizační smyčka

V rámci aktualizace zavolá nejprve OpenFrameworks metodu `update()` ve třídě `ofxCogApp`. Ta přidá dodatečné informace jako absolutní čas uplynulý od spuštění aplikace a časový rozdíl mezi poslední iterací - zde je možnost s těmito časy různě pracovat, např. zpomalit a zrychlit, což se bude hodit zejména při testování.

Poté se zavolá metoda `Update(delta, absolute)` ve třídě `ofxCogEngine`. Ta pouze iterací projde všechny *komponenty* a zavolá nad nimi stejnou metodu.

K aktualizaci herní scény dojde v okamžiku, kdy bude zavoláno `Update` nad *komponentou* `Stage`, jak ukazuje obrázek 2.5. Ta bude obsahovat ukazatel na kořen celého herního stromu, jehož potomky budou i jednotlivé scény.



Obrázek 2.5: Aktualizace enginu

2.3 Realizace

Implementace enginu probíhala současně s implementací prototypu hry, o které bude řeč až v kapitole 7. V této části bude popsána implementace jeho architektury a jednotlivých komponent. Některé z nich pak budou podrobněji popsány v následujících kapitolách.

2.3.1 Konvence

Engine byl naimplementován v jazyce C++. Jako způsob notace byl zvolen *Pascal case* - názvy proměnných jsou psané první malým písmenem, názvy metod prvním velkým písmenem. Všechen kód se nachází ve jmenném prostoru `Cog`. Dokumentace metod a proměnných je psána v *Javadoc* stylu, kdy je komentář uvozen dvěma počátečními hvězdičkami.

V enginu byl hojně využit sdílený ukazatel, *shared pointer*, jehož deklarace byla pro jednodušší použití přepsána pomocí direktivy `using` takto:

```
template<typename T>
using spt = std::shared_ptr<T>;

// example
auto myObj = spt<MyObject>(new MyObject());
```

2.3.2 Použité technologie

Kromě samotného OpenFrameworks bylo využito několik knihoven třetích stran. Zde je jejich výčet:

- **Catch** - knihovna pro automatizované testy, distribuována jako jeden hlavičkový soubor. Umožňuje jednoduchou deklaraci scénářů, test-casů a asercí pomocí maker [17]. Použitá verze je 1.2.1.
- **Lua** - interpreter jazyka Lua pro C++ [18]. Použitá verze je 5.1.5.
- **Luabridge** - rozšiřující knihovna pro mapování funkcí a tříd mezi Lua a C++ [19].
- **SQLite** - knihovna implementující transakční SQL databázový engine [20]. Použitá verze je 3.6.16.
- **Tinyxml** - knihovna určená pro parsování značkovacího jazyka XML [21]. Použitá verze je 2.5.3.
- **ofxXmlSettings** - OpenFrameworks addon, nadstavba knihovny *Tinyxml*, obsahuje funkce pro jednoduché iterování a získávání atributů a hodnot.
- **ofxNetwork** - OpenFrameworks addon určený pro síťovou komunikaci (TCP, UDP)
- **ofxSQLite** - OpenFrameworks addon, objektová nadstavba knihovny `SQLite`.
- **ofxTextLabel** - OpenFrameworks addon pro zarovnávání textů

2.3.2.1 Android a JNI

Android je open-source operační systém postavený na linuxovém jádře, určený především pro mobilní zařízení. V současné době je spravován společností *Google*. Jeho architektura se dělí na pět vrstev: Linux Kernel, knihovny, Android Runtime (virtuální stroj Dalvik), Application Framework a aplikace.

Vývoj aplikací probíhá obvykle s pomocí balíčku *Android SDK* za použití programovacího jazyka Java. Tento balíček obsahuje sadu nástrojů jako debugger, softwarové knihovny, emulátor, ukázkové kódy a tutoriály [22].

Pro Android je také možno vyvíjet v nativním kódu (C a C++) pomocí *Android NDK* balíčku. Jeho součástí je *Java Native Interface* (JNI), což je knihovna umožňující aplikacím napsaných v Javě interagovat s aplikacemi napsanými v nativním jazyce.

OpenFrameworks již JNI při kompilaci nativní části využívá. Addon `ofxAndroid`, který je součástí standardního instalačního balíčku frameworku, obsahuje vše potřebné pro zkompilování celé aplikace pro systém Android.

2.3.3 Realizace herní smyčky

Při aktualizaci herního modelu se předává parametr `delta`, určující počet milisekund od poslední aktualizace. Zde jsou možné dva přístupy: buďto bude tento čas vždy konstantní a v případě vyšší zátěže dojde ke zpomalení, nebo bude proměnlivý tak, aby hra běžela stále stejnou rychlostí - zde pak v případě vyšší zátěže dojde ke ztrátě přesnosti, především u animací.

Počet aktualizací a vykreslení za jednu sekundu je standardně nastaven na 60. Hodnota delta by tedy měla být v průměru 16 milisekund.

Engine počítá standardně s proměnlivým časem pro interval [d,2d], kde d je perioda aktualizace. Mimo tento interval bude nastavena fixní hodnota jeho krajního bodu, což bude mít za následek při vyšším výkonu zpomalení aplikace.

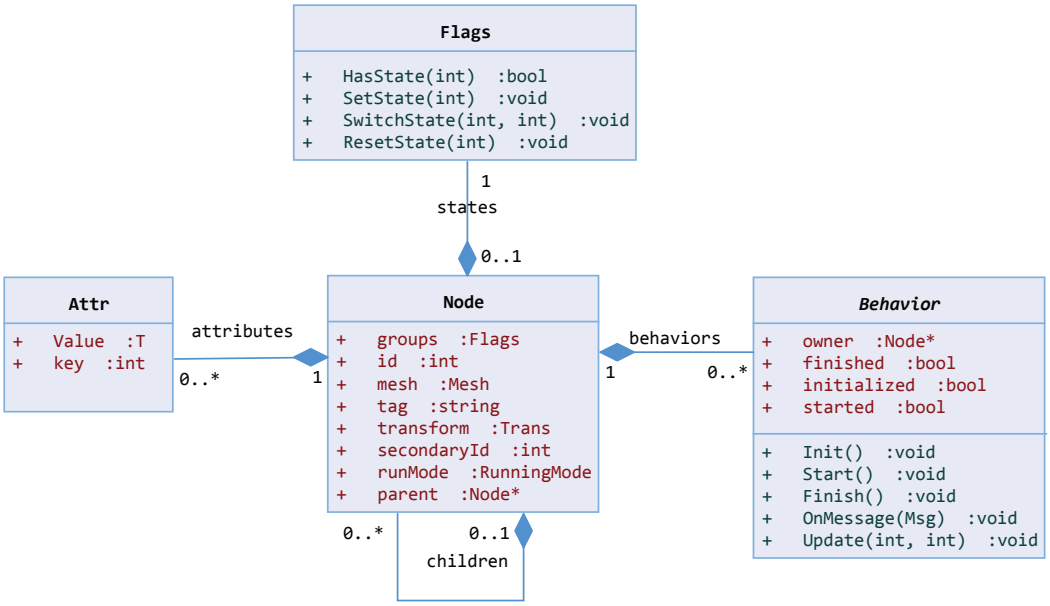
2.3.4 Herní scéna

Každý herní objekt `Node` obsahuje kolekci *behavior* a atributů. **Během implementace byly odděleny tři nejpoužívanější atributy** - transformace, grafický objekt (*mesh*) a objekt `Flags` jako klasické členské proměnné třídy `Node`.

`Flags` představuje bitové pole, do kterého je možno ukládat stavy indexované přirozeným číslem. `Mesh` může představovat obrázek, font, barevný obdélník nebo *sprite*.

Kromě těchto atributů má každý herní objekt název (atribut `tag`), sekundární identifikátor, který je na rozdíl od primárního nastavitelný, dále seznam skupin, ve kterých se nachází, a odkaz na svého předka a potomky.

Parametr `runMode` se používá pro nastavení viditelnosti herního objektu či jeho reakci na události. Aktivní uzel má tuto hodnotu nastavenou na `RUNNING`. Hodnota `DISABLED` jej zcela zneaktivní, hodnota `PAUSED` vypne aktualizaci a hodnota `INVISIBLE` vypne vykreslování.



Obrázek 2.6: Herní objekt

2.3.5 Posílání zpráv

Pro poslání zprávy stačí vytvořit objekt `Msg` a poslat jej pomocí metody `SendMessage(msg)` aktuální scéně. *Komponenty* a *behaviors* mají vlastní metody, které zprávu automaticky seskládají a proces zasílání tak výrazně zjednoduší.

Následující příklad ukazuje poslání zprávy o nastavení nového stavu objektu:

```
void Node::SetState(unsigned state) {
   this->states.SetState(state);
   SendMessage(ACT_STATE_CHANGED, spt<FlagChangeEvent>(new
       FlagChangeEvent(FlagChangeType::SET, state)));
}
```

Ukázka poslání zprávy

2.3.6 Implementované komponenty

Zde je stručný popis všech *komponent*, tedy globálních komponent ve smyslu komponentové architektury, které byly implementovány jako součást enginu. Některé z nich budou předmětem dalších kapitol.

2.3.6.1 CogDatabase

Jedná se o jednoduchý wrapper nad databázovým enginem `SQLite`. Jelikož C++ neobsahuje reflexi, je nutné všechny atributy z databáze namapovat na příslušné objekty ručně.

Pro snazší práci s objekty byl využit existující addon `ofxSQLite`, ke kterému byla doimplementována třída `SQLiteEntity`, reprezentující abstraktního předka všech databázových entit.

2.3.6.2 Interpolator

Komponenta pro aktualizaci spojitých veličin na základě synchronizačních datagramů, přijatých buďto sítí a odchycených pomocí systému zasílání zpráv, nebo předaných pomocí metody. O této problematice bude řeč v kapitole 5.

2.3.6.3 Environment

`Environment` slouží k uchovávání stisknutých kláves, pozic myši a dotyků a právě přehrávaných zvuků. Tyto kolekce v průběhu aktualizační smyčky upravuje podle aktuálního stavu - pokud např. dojde k uvolnění tlačítka, aktualizuje se objekt, který informaci o původně stisknutém tlačítku obsahoval.

2.3.6.4 InputHandler

`InputHandler` je využíván pro přeposílání spuštěných vstupních událostí. Protože objekty v herním stromu nejsou nijak řazeny a tento strom se vytváří podle toho, jak do něj byly jednotlivé herní uzly vloženy, hodí se tato *komponenta* pro ukládání všech objektů, které detekovaly jednu konkrétní událost.

Pokud například uživatel stiskne tlačítko myši, na které může zareagovat větší množství objektů najednou, tato komponenta je všechny zaregistruje a přednostně pošle zprávu tomu v popředí (s nejvyšším z-indexem).

2.3.6.5 Logger

Logování je možné buďto do konzole nebo do souboru. Komponenta může být inicializována pomocí globálního nastavení v konfiguračním XML souboru nebo ručně.

```
CogLogError("Environment","Error while parsing aspect ratio for;
expected format xx/yy, found %s", aspectRatio.c_str());
```

Příklad logování

2.3.6.6 LuaScripting

Tato *komponenta* inicializuje interpreter jazyka Lua a pomocí knihovny `luabridge` nastaví mapování všech objektů, ke kterým se bude uvnitř skriptu přistupovat. O skriptování bude řeč v kapitole 4.

2.3.6.7 NetworkCommunicator

`NetworkCommunicator` slouží k synchronizaci dat po síti s využitím tzv. aktualizačních a informačních zpráv. O této *komponentě* bude pojednávat kapitola 5.

2.3.6.8 Renderer

`Renderer` se stará o vykreslování. Podrobněji bude popsán v kapitole 3

2.3.6.9 ResourceCache

V této *komponentě* je uložena objektová reprezentace všech souborů jako jsou konfigurační XML soubory, zvuky, animace, sprite sheety, obrázky a fonty.

2.3.6.10 SceneSwitchManager

Stará se o přepínání scén. Přepínání je možno učinit okamžitě nebo pomocí animace. V takovém případě je nutné po krátký časový úsek vykreslovat obě dvě scény, dokud se ta druhá nenasune do popředí.

2.3.6.11 Stage

`Stage` má na starosti správu herní scény. Jakmile je zavolána metoda `Update`, dojde k postupné aktualizaci celého herního stromu skrz tuto *komponentu*.

2.3.7 Implementované behaviors

Zde je výčet všech implementovaných *behaviors*, které je možno použít jako součást herního stromu prostým přiřazením jednotlivým objektům `Node`.

2.3.7.1 AttribAnimator

Tento *behavior* má na starosti atributové animace. Animovat je možno například rotaci, pozici či velikost. Definice takové animace může být poměrně komplexní a je diskutována v sekci 3.3.1.1.

2.3.7.2 Button

Behavior, které z daného uzlu udělá tlačítko. Musí existovat společně s `HitEvent`, který jej bude notifikovat v případě, že uživatel na tlačítko kliknul.

2.3.7.3 CompositeBehavior

Tento *behavior* umožňuje v sobě seskupovat jiné *behaviors*, díky čemuž tak vzniká stromová struktura. Aktualizace je provedena nad celým stromem, stejně jako zaslání zprávy.

2.3.7.4 DelayAction

`DelayAction` umožňuje poslat zprávu se zpožděním. Jako parametr přebírá zprávu, která se má poslat, a přesný čas v milisekundách, ve kterém k tomu má dojít. Tento čas je odvozen od počtu milisekund od inicializace enginu a vždy se posílá v rámci aktualizace modelu jako parametr metody `Update`.

2.3.7.5 FloatingScene

Toto *behavior* funguje podobně jako prohlížeč obrázků - zobrazí objekt, který může být větší než velikost displeje a pomocí myši či *pinch* gesta umožní jeho posouvání a přibližování či oddalování. Tento *behavior* byl použit v prototypu hry, o které bude řeč v kapitole 7.

2.3.7.6 Goal

Jedná se o implementaci *goal-driven behavior* pro definici chování autonomních agentů na základě kompozitní struktury cílů. Tato technika bude předmětem kapitoly 6, věnující se umělé inteligenci.

2.3.7.7 HitEvent

Na události jako pohyb kolečka myši či stisk klávesy je možno zareagovat prostou registrací konkrétního typu zprávy. U událostí, ke kterým je vázána nějaká pozice na displeji (tlačítko myši, dotyk), je potřeba detekovat, na který objekt uživatel kliknul.

K tomu slouží `HitEvent` a musí být přiřazen všem objektům, se kterými uživatel může interagovat. Obecně se kontrola provádí prostým zjištěním, zda se místo dotyku nenachází uvnitř obdélníku, který obaluje daný objekt. V případě obrázků je možné provést přesnější test, kdy se porovná barva stisknutého pixelu.

Samotný test se provádí tak, že se pozice dotyku vynásobí inverzní transformační maticí daného objektu - díky tomu se souřadnice dotyku počítají v soustavě souřadnic tohoto objektu.

2.3.7.8 Move

Tento *behavior* pracuje s atributem typu `Move`, což je kolekce sil působících na daný objekt. Z nich se vypočte výslednice, aktuální zrychlení a rychlost, a podle těchto hodnot se pak nastaví transformace objektu.

Rychlost je definována jako počet jednotek souřadného systému objektu za jednu milisekundu.

2.3.7.9 MultiAnim

`MultiAnim` se používá pro běh většího množství animací najednou. Vždy je aktivní jen jedna animace a jakmile skončí, zaktivní se následující v řadě.

2.3.7.10 MultiSelection

`MultiSelection` se chová podobně jako zaškrtávací políčko (*checkbox*). Všechny objekty, které jsou součástí nějaké skupiny, budou mít přiřazeno toto *behavior* s odkazy na obrázky zaškrtnutého a nezaškrtnutého tlačítka a identifikátoru příslušné skupiny.

2.3.7.11 Selection

Funguje podobně jako `MultiSelection` s tím rozdílem, že je možno označit právě jeden objekt z dané skupiny (*radiobutton*).

2.3.7.12 SheetAnimator

Behavior, které spouští obrázkové animace. Postupuje animačním stromem směrem do hloubky a to tak, že nejprve u každého uzlu spustí animaci uzlu samotného, poté animace jeho potomků a nakonec animaci svých sourozenců.

Způsob procházení animačního stromu je znázorněn v sekci 3.3.1.

2.3.7.13 State

Behavior, který je možno použít pro implementaci stavu. Bude podrobněji popsán v kapitole 6.

2.3.7.14 StateMachine

Behavior reprezentující stavový automat. Bude podrobněji popsán v kapitole 6.

2.3.7.15 SteeringBehavior

Implementace *steering behaviors*, sady funkcí pro simulaci realistického pohybu herních agentů, kterým se bude věnovat kapitola 6.

2.3.7.16 TransformAnim

`TransformAnim` umožňuje provést animaci z jednoho transformačního objektu do druhého. Transformaci je možno nastavit jako překrývací (hodnoty jsou pevně nastaveny) či aditivní (hodnoty se přičtou k aktuálním hodnotám transformace objektu).

2.3.7.17 Tween

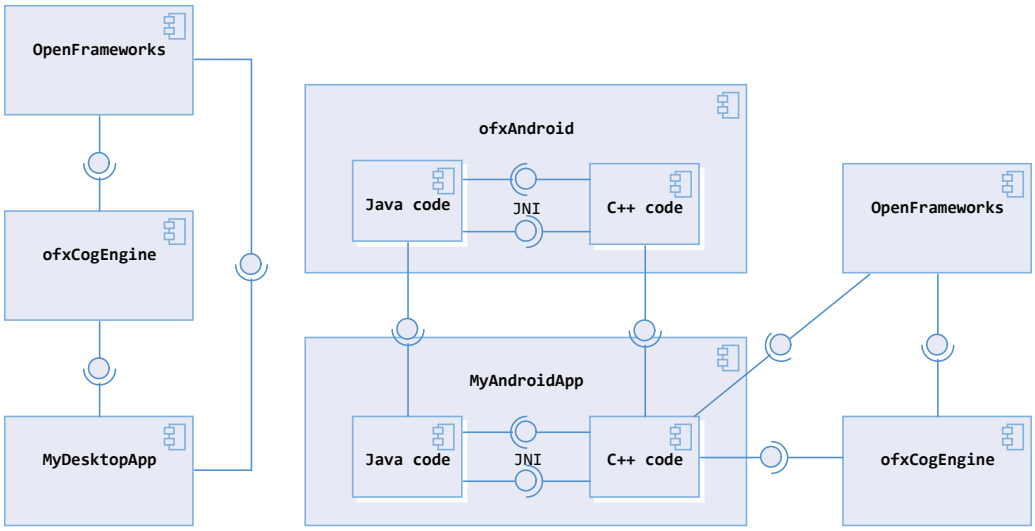
Jedná se o *behavior* používaný k posouvání scén. Posun může být do čtyř různých směrů a v rámci konfigurace je možno nastavit rychlost posunu a tzv. *easing* funkci, která modifikuje rychlost animace v různých úsecích a zajišťuje tak estetičtější přechod.

2.4 Sestavení

Pomocí direktiv pro preprocessor je možné engine jednoduše zkompilovat pro obě platformy, Windows i Android. I když se v OpenFrameworks s uživatelskými vstupy pro obě platformy (myš a dotyk) pracuje odlišně, engine umožňuje tyto události zpracovávat jednotně.

Diagram 2.7 ukazuje schéma sestavení. OpenFrameworks je distribuován v podobě zdrojových kódů, pro engine to bude platit také.

V případě Windows stačí u vlastního projektu nareferencovat OpenFrameworks a CogEngine, u platformy Android je situace o něco složitější: zdrojové kódy projektu jsou rozděleny na C++ část a Javovskou část a kromě OpenFrameworks je potřeba ještě nareferencovat doplněk *ofxAndroid*, který obsahuje metody pro komunikaci s frameworkem a načítání projektových souborů. Další podrobnosti budou popsány v manuálu.



Obrázek 2.7: Diagram sestavení pro Windows (vlevo) a Android (vpravo)

2.5 Ukázkové příklady

Pro účely demonstrace různých funkcí byla vytvořena sada ukázkových příkladů, které budou distribuovány spolu s enginem. Nachází se také na přiloženém CD ve složce `Examples`

Seznam příkladů:

- **CogEngineLab** - prázdný projekt, který je možno použít pro různá testování. Je možné jej zkompilovat a spustit pod systémy Windows i Android
- **Network** - synchronizace rotačního objektu po síti
- **Network2** - posílání informací po síti
- **Network3** - synchronizace pohybu velkého množství objektů po síti
- **Scenes** - ukázka šesti statických scén, definovaných v XML
- **SceneSwitch** - přepínání dvou scén
- **Scripting** - přepínání obrázků pomocí skriptu
- **Scripting2** - pohyb objektu řízený skriptem
- **Scripting3** - posílání zpráv mezi dvěma skripty
- **Sprites** - vykreslování velkého množství objektů
- **Transforms** - dvacet scén s různým nastavením transformace

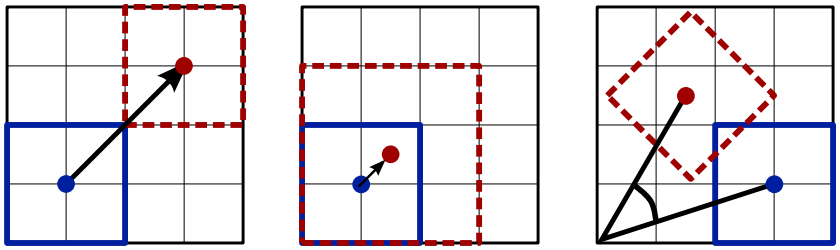
Kapitola 3

Grafika

Tato kapitola se věnuje komponentám enginu, které přímo či nepřímo souvisí s grafikou. Bude zde objasněn způsob deklarace a výpočtu transformací objektů, možné způsoby vykreslování a definice animací.

3.1 Transformace

V počítačové grafice se obvykle pro vyjádření lineárních transformací používají matice, přičemž rozlišujeme tři typy fundamentálních transformací: translaci, změnu měřítka (škálování) a rotaci. Pro zadání těchto transformací by stačila matice velikosti 3×3. Aby ale bylo možno vyjádřit i perspektivní projekci, používá se matice 4×4. Ke třem souřadnicím (x, y, z), se pak přidává čtvrtá souřadnice w, označovaná jako váha [23].

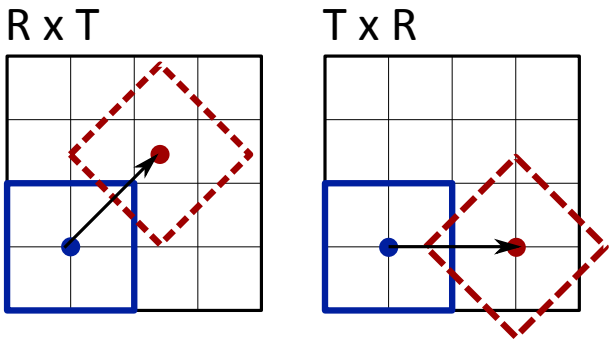


Obrázek 3.1: Transformace (translace, změna měřítka a rotace)

Při zobrazování objektů existují dva druhy projekce - *paralelní*, kdy všechny paprsky svírají s průmětnou stejný úhel, a *perspektivní*, kdy jsou vedeny do středu promítání.

Výhodou zadávání transformací pomocí matic je fakt, že se dají jednoduše skládat. Složením transformací vyjádřených maticemi M_1 a M_2 vznikne transformace $M_3 = M_1 \times M_2$. Protože maticové násobení není komutativní, různé pořadí dílčích matic dává různé výsledky.

Pokud bychom měli translační matici T a rotační matici R, matice vzniklá vynásobením $T \times R$ by objekt zobrazila na pozici, kam by ji posunula translační matice, a otočila by jej podle rotační matice. Oproti tomu násobek matic $R \times T$ by objekt posunul ve směru úhlu, který udává rotační matice. Tento fenomén znázorňuje obrázek 3.2.



Obrázek 3.2: Transformace v různém pořadí

3.1.1 Návrh

V knihovně OpenGL, kterou využívá OpenFrameworks, se využívají tři typy matic - *ModelView* matice slouží pro umístění objektů ve scéně, *CameraView* se používá pro nastavení polohy a natočení kamery a *ProjectionView* se používá pro nastavení perspektivní či paralelní projekce. Dále zde může dojít také k transformaci pracoviště (*viewport*), což je velikost a posun výřezu, který se zobrazí na obrazovce.

Protože engine se zaměřuje na 2D hry, bude ke grafice přistupováno odlišně než by tomu bylo v případě 3D - nebude zde žádná kamera a s transformačními maticemi se bude pracovat pouze uvnitř enginu. Jako inspirace zde posloužila deklarace transformací v technologii CSS, kde je možné volit různé způsoby pozicování objektů (absolutní, relativní). Každý herní objekt `Node` bude mít transformaci definovanou ve svém lokálním systému souřadnic a při vykreslování se tyto transformace budou skládat (absolutní transformace tak bude závislá na umístění objektu v grafu scény), bude je však možno definovat vícero způsoby a do lokálního systému se pak přepočítají.

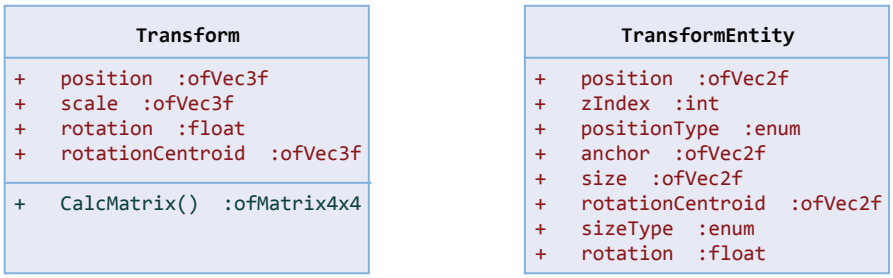
Na obr. 3.3 je návrh dvou objektů - `Transform`, se kterým bude pracovat vykreslovací *komponenta*, a `TransformEntity`, která bude sloužit pro obecné zadávání transformací.

OpenFrameworks nerozlišuje mezi pozicemi a vektory - obojí je reprezentováno strukturou `ofVec2f` pro 2D, resp. `ofVec3f` pro 3D, proto budou i pozice udávány ve vektorech.

Pozici bude reprezentovat trojrozměrný vektor. Třetím rozměrem zde bude *z-index*, díky kterému bude možno objekty umisťovat do popředí či pozadí.

Rotace a změna měřítka budou počítány ve dvourozměrném kartézském souřadném systému. V počítačové grafice, kde se tento systém používá, je obvykle osa y převrácená. To přináší řadu výhod - například při vkládání víceřádkového textu neznámé délky. Tuto konvenci bude dodržovat i CogEngine: počátek souřadného systému bude umístěn v levém horním rohu obrazovky a pozice objektů budou pozicemi jejich levých horních rohů. Úhly budou počítány ve stupních, neboť se budou lépe zapisovat v konfiguračních souborech.

Transformace, reprezentována objektem `Transform`, se bude skládat ze čtyř atributů - pozice, změny měřítka, rotace a počátku rotace. Z těchto čtyř atributů se pak během vykreslování vygeneruje transformační matice.



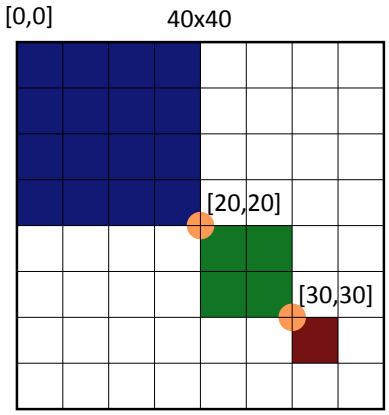
Obrázek 3.3: Transformační entity

3.1.1.1 Výpočet transformace

Výpočet absolutní pozice a měřítka bude probíhat podle následující rovnice:

```
absScale = parent.absScale*scale;
absPosition = parent.absPosition+position*parent.absScale;
```

Pro objasnění závislosti transformací na umístění objektů v grafu scény uvažujme následující příklad: na obrázku 3.4 jsou vyobrazeny tři čtverce různé velikosti na různých pozicích, nacházející se ve scéně velikosti 40x40 jednotek. Tabulka 3.1 pak ukazuje pro každý čtverec nastavení jeho pozice a měřítka v závislosti na uspořádání tak, aby absolutní pozice a měřítko odpovídaly tomu, co je na obrázku.



Obrázek 3.4: Příklad transformace

Uspořádání	**Modrý**	**Zelený**	**Červený**
scéna -> všichni	[0,0],1	[20,20],0.5	[30,30],0.25
scéna -> M -> Z -> Č	[0,0], 1	[20,20], 0.5	[20,20], 0.5

Tabulka 3.1: Hodnoty pozic a měřítka pro různá uspořádání

Pokud by všechny tři čtverce měly jako předka celou scénu, jejich lokální pozice by odpovídala absolutním pozicím. Pokud bychom ale nastavili zelený čtverec jako potomka modrého a červený jako potomka zeleného, situace se mění - díky měřítku 0.5 u zeleného čtverce je systém souřadnic scény a červeného čtverce jiný - laicky řečeno, pokud bychom červený čtverec chtěli posunout o 10 jednotek v ose x, museli bychom přičíst k jeho aktuální pozici hodnotu 20. Kdyby naopak měl zelený čtverec měřítko 2 (byl by $2x$ větší než modrý čtverec), posunuli bychom ten červený jen o 5 jednotek.

Změna měřítka má tedy vliv na pozicování potomků daného objektu. Tento přístup má jednu nesmírnou výhodu - umožňuje libovolnou část grafu scény přemisťovat a měnit její měřítko, aniž by bylo potřeba přenastavovat transformace jednotlivých objektů v tomto podgrafu. Kdybychom chtěli transformovat všechny tři čtverce najednou, v prvním případě, kdy je každý čtverec přímý potomek scény, bychom museli transformovat všechny tři. Ve druhém případě, kdy jsou tyto objekty umístěny v grafu pod sebou, by stačilo transformovat pouze modrý čtverec.

3.1.1.2 Zadávání transformace

Díky entitě `TransformEntity` bude možno transformaci zadávat mnohem jednodušeji, jelikož nebude nutné vycházet z absolutní transformace daného objektu, aby bylo možno vypočítat lokální transformaci.

Kromě pozice, velikosti, rotace a rotačního středu obsahuje další tři atributy: `positionType`, `sizeType` a `anchor`.

`Anchor` bude sloužit k relativnímu posunu objektu, pro který je pozice počítána. Pokud bychom tedy chtěli, aby se na dané pozici nacházel střed objektu, stačí tento atribut nastavit na hodnotu $[0.5, 0.5]$. Pro pravý horní roh by to bylo $[1.0, 0]$ a pro levý horní zase $[0, 0]$.

`PositionType` a `SizeType` stanoví způsob počítání transformace pro pozici, resp. měřítko.

Výčet možných hodnot `PositionType` a `SizeType`:

- **absolutní** - hodnota bude nastavena podle systému souřadnic scény
- **lokální** - hodnota bude nastavena podle systému souřadnic objektu
- **procentuální** - hodnota bude počítána jako relativní k předkovi. Pro pravý dolní roh předka to bude hodnota $[1, 1]$.
- **absolutně procentuální** - hodnota bude počítána jako relativní k celé scéně. Pro pravý dolní roh scény to bude hodnota $[1, 1]$.
- **grid** - engine umožní definovat pro scénu tzv. virtuální velikost (např. 70x40), nezávislou na velikosti displeje. Výsledná hodnota pak bude počítána v těchto jednotkách.

Pokud tedy bude scéna definována jen na základě relativních transformací, nebudou pozice jednotlivých objektů záviset na rozlišení displeje.

3.1.2 Implementace

Implementace probíhala podle návrhu, přičemž objekt `Transform` byl přejmenován na `Trans` pro kratší zápis v kódu.

Aby mohly být transformace zadávány i v konfiguračním XML souboru, bylo potřeba definovat způsob zápisu pro každý typ jednotky (relativní, absolutní apod.).

Způsob zápisu jednotek:

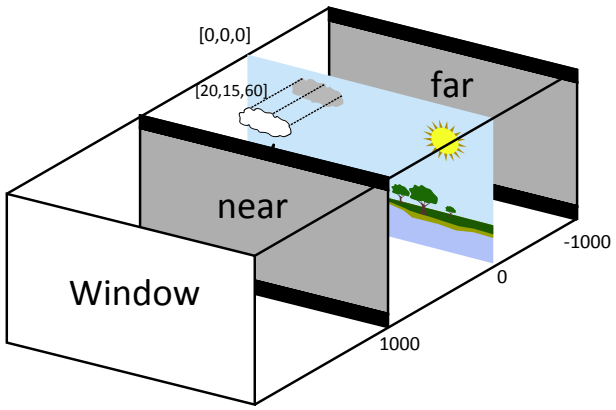
- **absolutní** - písmeno „un“, například `12un`
- **lokální** - bez písmene, například `12`
- **procentuální** - písmeno „r“, například `0.7r`
- **absolutně procentuální** - dvojice písmen „rp“
- **grid** - dvojice písmen „gr“, například `50gr`

Velké množství ukázek možného nastavení transformací se nachází v doprovodných příkladech - projekt *Transforms*.

3.2 Vykreslování

Protože OpenFrameworks nerozlišuje mezi 2D a 3D, bude engine vykreslovat dvourozměrné objekty s využitím mechanismů aplikovatelných na 3D prostor. Pro korektní vykreslení je potřeba nastavit matici *ProjectionView* pro paralelní promítání. Vzdálenost objektů od kamery tak nebude mít vliv na jejich pozici a velikost na displeji, tudíž můžeme bez obav využít třetí rozměr (hloubku) jako z-index.

Na obrázku 3.5 je znázorněna paralelní projekce, kterou engine bude využívat. Ořezové roviny mají vzdálenosti -1000 a 1000, což jsou zároveň minimální a maximální hodnoty z-indexů. V případě, že je v enginu nastaven fixní poměr stran a zařízení používá jiný poměr, budou roviny shora a zdola oříznuté, což je zde vidět v podobě černých pruhů.



Obrázek 3.5: Paralelní projekce

3.2.1 Vykreslování obrázků

O vykreslování objektů se stará *komponenta* `Renderer`. Funguje tím způsobem, že při zavolání metody `Draw` nad každým herním objektem se tento objekt zaregistruje do této komponenty, která je pak setřídí podle z-indexu.

Samotný vykreslitelný objekt je uložen jako atribut `Mesh` ve třídě `Node`. Jeho transformaci pak určuje atribut `Transform`, ze kterého se vygeneruje a uloží transformační matice - tímto způsobem mohou být vykresleny obrázky, obdélníky a texty. Způsob vykreslení zajišťuje OpenFrameworks - každá vykreslovací entita obsahuje metodu `draw()`.

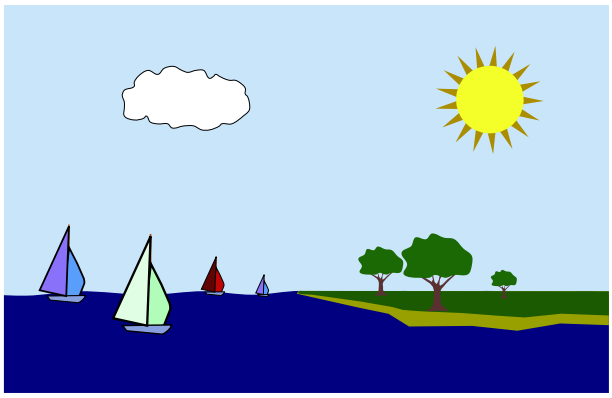
Pro každý obrázek je potřeba uložit transformační matici, načíst texturu a provést vykreslení, což je poměrně náročný proces, který naráží na rychlost přesunu dat z operační paměti do paměti grafické karty. Pří větším počtu obrázků se tak díky pomalému přenosu dat začne snižovat vykreslovací frekvence.

Z tohoto důvodu se u her místo samostatných obrázků využívají tzv. *sprite sheety*, což je textura obsahující množinu menších textur, které se mají vykreslit v rámci jednoho procesu. Sprite sheety nejsou součástí OpenFrameworks, proto bylo potřeba tuto komponentu naprogramovat.

3.2.2 Vykreslování spritů

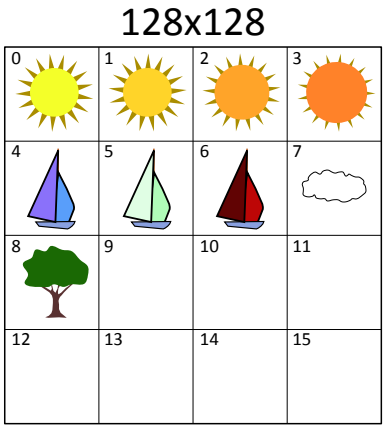
Pro vykreslování spritů byla vytvořena třída `SpriteSheetRenderer`, která je součástí třídy `Renderer` a bude využita pro průběžné ukládání spritů a jejich jednotné vykreslování.

Obrázek 3.6 zobrazuje scénu, která má být vykreslena - pozadí i moře jsou statické obrázky a vše ostatní má být načteno ze sprite sheetu.



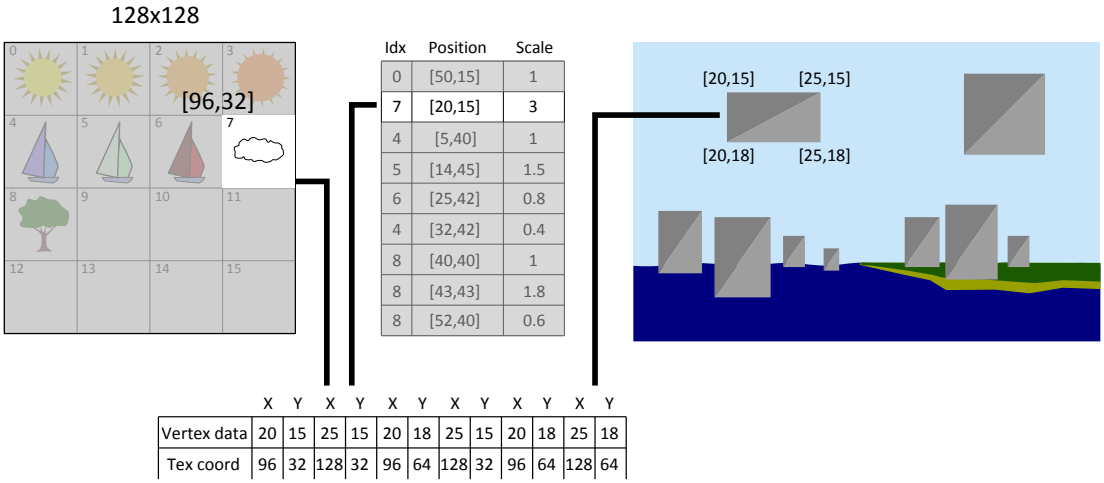
Obrázek 3.6: Scéna s několika sprity

Pro snadné zpracování grafickou kartou by měl mít každý sprite sheet čtvercový rozměr mocniny 2. Následující obrázek ukazuje příklad sprite sheetu o rozměrech 128x128. Všechny sprity mají velikost 32x32 a z jejich indexu je možno odvodit přesnou pozici v textuře.



Obrázek 3.7: Ukázka sprite sheetu

Během vykreslení je potřeba projít všechny sprity ve scéně, podle indexu spočítat souřadnice v textuře a absolutní pozici. Pokud má některý ze spritů nenulovou rotaci, je potřeba tuto rotaci provést i se souřadnicemi v textuře. Velikost části textury se poté vynásobí velikostí spritu - z toho se pak odvodí absolutní pozice obdélníka, na který se textura vykreslí, jak je znázorněno na obrázku 3.8.



Obrázek 3.8: Vykreslování spritů

Výstupem tohoto procesu jsou dvě pole - pole vrcholů a pole souřadnic pro texturu (známé jako *UV souřadnice*). Protože grafická karta pracuje pouze s trojúhelníky a všechny obecné n-úhelníky jsou na ně přepočítávány [24], byl `SpriteSheetRenderer` navržen tak, aby rovnou ukládal trojúhelníky. Nevýhodou je zde větší spotřeba paměti, jelikož dva ze tří vrcholů jsou v poli uloženy dvakrát.

Porovnání rychlosti vykreslování obrázků a spritů bude diskutováno v části 8.4.1.

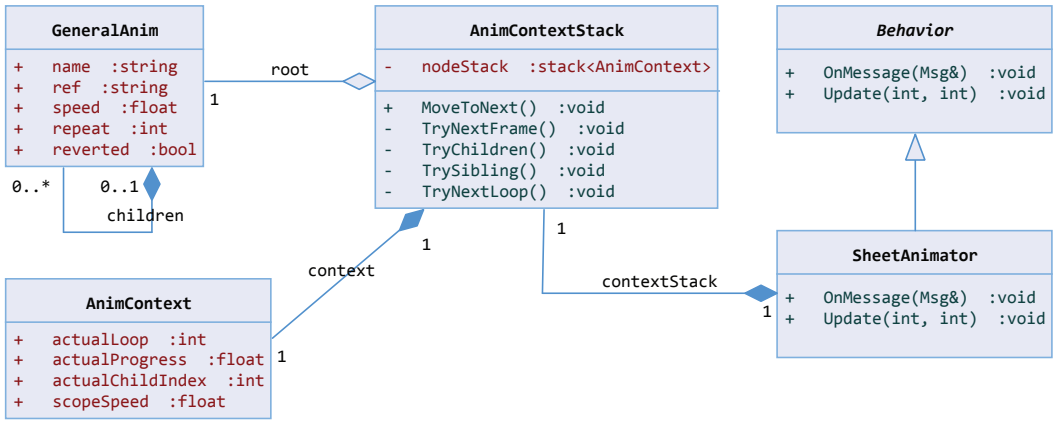
3.3 Animace

Animace byly naimplementovány dvojího typu - obrázkové animace pro přehrávání sady obrázků a atributové animace pro kontinuální změnu transformačních atributů. Obojí je možno definovat v konfiguračním XML souboru.

3.3.1 Obrázkové animace

Obrázkovou animaci může představovat buďto sekvence obrázků nebo sekvence spritů v jednom sprite sheetu. Každá animace má unikátní jméno a může se skládat z libovolného množství dalších animací. Animace tak budou tvořit animační strom, který bude možno rekurzivně procházet a spouštět jednotlivé animace.

Diagram 3.9 znázorňuje architekturu animační komponenty - *behavior* typu `SheetAnimator` zde drží referenci na `AnimContextStack`, pomocí které pak prochází animační strom - metoda `MoveToNext()` přehraje postupně všechna okna dané animace a poté rekurzivně pokračuje v potomcích, po návratu přehraje další iteraci (pokud je počet opakování větší než 1) a nakonec se přepne na další uzel v pořadí.

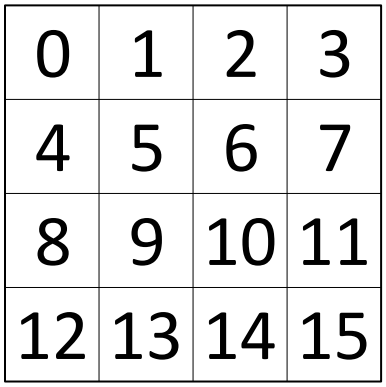


Obrázek 3.9: Návrh animační komponenty

Jako názorný příklad je možno uvést konkrétní definici animace v XML struktuře, kde je animační strom dobře viditelný: na obr. 3.10 je zobrazen sprite sheet s šestnácti sprity. Každý sprite obsahuje číslo, které zároveň značí jeho index. Kdybychom chtěli z tohoto sprite sheetu vytvořit animaci, která přehraje všechna čísla od 0 do 16, vypadala by konfigurace takto:

```
<anim name="numberAnim" repeat="1" sheet="numbers.png" frames="4"
 lines="4" start="0" end="15" increment="1" />
```

Je zde deklarována animace s názvem *numberAnim* o čtyřech řádcích a čtyřech sloupcích. Bude začínat na spritu s indexem 0 a končit bude na indexu 15. Atribut `repeat` je nastaven, aby se animace spustila právě jednou a atribut `increment` deklaruje, že nebudou žádná čísla vynechána.



Obrázek 3.10: Animační sprite sheet

Animace je možno libovolně skládat prostým vnořováním příslušných XML elementů, jak ukazuje následující příklad:

```
<anim name="numberAnim2">
  <anim ref="numberAnim" />
  <anim ref="numberAnim" increment="2" />
  <anim ref="numberAnim" start="1" increment="2" />
</anim>
```

Atribut `ref` odkazuje na animaci `numberAnim`, která v rámci animace `numberAnim2` bude spuštěna celkem třikrát - přehrají se všechna čísla, poté všechna sudá a nakonec všechna lichá čísla.

Další animace je možno nalézt v doprovodných příkladech (projekt *Scenes*).

3.3.1.1 Atributové animace

Atributové animace se skládají z deklarace atributu, který bude animován, počáteční a konečné hodnoty a délky trvání. V následujícím příkladu je ukázána rotace o 360:

```
<attranim name="myAnim" attr="rotation" from="0" to="360" duration="1000"
repeat="1" />
```

Princip je stejný jako u obrázkových animací - opět je lze libovolně skládat a odkazovat se na již definované animace pomocí atributu `ref`.

Kapitola 4

Komponenty

V této kapitole budou popsány zbývající části enginu, kromě komponent týkajících se přenosu dat po síti a umělé inteligenci - o těch bude řeč v kapitolách 5 a 6.

4.1 Konfigurace

Statické objekty herní scény, nastavení konstant, animací a transformací je možno nadefinovat v XML souboru. Cesta k němu se pak předá enginu během inicializace.

4.1.1 Nastavení

V konfiguračním souboru je možno definovat dva typy nastavení - globální a projektové.

Globální nastavení je určeno obecně pro *komponenty*, které existují v průběhu celého životního cyklu aplikace. Je zde možno nastavit např. úroveň logování, preferovaný poměr stran či různé počáteční hodnoty. Toto nastavení je načteno během inicializace jednotlivých *komponent* a je zcela v jejich režii, jaké atributy se zde mohou nacházet.

Projektové nastavení engine přímo nevyužívá. Slouží jako úložiště libovolných *key-value* záznamů pro vlastní aplikaci. Obě nastavení jsou uchovávána v `ResourceCache`.

```
<app_config>
<global_settings>
  <setting name="logger">
    <item key="level" value="DEBUG" />
  </setting>
  <setting name="myComponent">
    <item key="multiple_values">
      <value>value1</value>
      <value>value2</value>
    </item>
  </setting>
</global_settings>
</app_config>
```

Ukázka globálního nastavení

4.1.2 Definice scén

Scény jsou definovány v kořenovém elementu `scenes`. Každá scéna pak může mít kolekci herních objektů `Node`. Jejich transformaci určuje element `transform`, grafický objekt zase element `Node`. Jednotlivé *behaviors* se pak definují pomocí elementu `behavior`, kde atribut `type` značí jejich konkrétní typ (název třídy).

```
<scenes initial="scene1"> <!-- scene1 is initial -->
  <scene name="scene1"> <!-- first scene -->
    <node img="background.png">
     <transform pos_x="0" pos_y="0" />
   </node>
    </scene>
   <scene name="scene2"> <!-- second scene -->
     <node img="sky.png">
       <transform pos_x="0" pos_y="0" />
     </node>
     <node img="water.png">
       <transform pos_x="0" pos_y="320" />
     </node>
     <node img="yacht.png">
       <transform pos_x="120" pos_y="290" />
      <behavior type="YachtBehavior"> <!-- behavior for yacht -->
        <setting>
          <item key="speed" value="120"/>
        </setting>
      </behavior>
     </node>
   </scene>
  </scenes>
```

Ukázka definice scény

Velké množství konfigurací je možno nalézt v doprovodných příkladech - téměř každý z nich obsahuje nějaký konfigurační soubor. Projekt `Scenes` pak obsahuje šest scén s animacemi definovanými v XML.

4.2 Skriptování

Zatímco pro definici scén, nastavení atributů a pozic objektů bude využit konfigurační soubor XML, pro definici herní logiky nabídne engine skriptovací jazyk.

Skriptovací jazyky jsou v oblasti vývoje her velice populární. Změna ve skriptu totiž nevyžaduje rekompilaci celého projektu, což významně ušetří čas během ladění. Navíc pracují na vyšší vrstvě než C++ a mohou tak nabídnout techniky jako reflexi, *garbage collector*, dynamické typování apod.

Využívají je především herní návrháři, kteří neřeší technické záležitosti hry, ale herní logiku, příběh, mapy úrovní atd.

4.2.1 Výběr skriptovacího jazyka

Pro engine byl vybírán skriptovací jazyk ze dvou kandidátů - JavaScript a Lua. Oba jsou poměrně často používány v různých herních enginech (např. Lua v Marmalade a JavaScript v Unity), JavaScript je však pro své časté používání ve webových aplikacích mnohem známější.

4.2.1.1 JavaScript

JavaScript je skriptovací jazyk primárně určený pro webové stránky. V různých modifikacích jej implementují jednotlivé prohlížeče, nicméně jazyk jako takový je standardizován pod názvem *ECMAScript* [25].

JavaScript byl vytvořen v roce 1995 jako součást prohlížeče *Netscape*. Je objektově orientovaný a dynamicky typovaný. Obsahuje 6 datových typů (*boolean, null, undefined, number, string, symbol*) a typ *Object*. K definici strukturovaných dat se používá JSON (*JavaScript Object Notation*).

Mezi běžně používané JavaScriptové interpretery patří například V8 a Duktape.

4.2.1.2 Jazyk Lua

Lua je open source skriptovací jazyk, jehož interpreter je napsán v ANSI C a kompatibilní s C++. Sám kompiluje zdrojové kódy do bajtkódu, který je následně interpretován ve virtuálním stroji [26]. Velikost knihovny je pouhých 63 KB.

Lua byl vytvořen v roce 1993 v Brazílii původně pro akademické účely, dnes se běžně používá pro vývoj her - z těch známějších je možno jmenovat např. MDK 2, Escape from Monkey Island nebo Baldur's gate.

Obsahuje jen malé množství typů a konstruktů - nenajdeme zde žádné objekty ani dědičnost, nicméně toto vše je možno implementovat v jazyce samotném. Obsahuje 8 datových typů (*nil, boolean, number, string, functions, user-data, thread* a *table*).

4.2.1.3 Zvolený jazyk

V případě JavaScriptu byl vyzkoušen interpreter Duktape. Instalace je velice jednoduchá, neboť je distribuován v podobě dvou zdrojových souborů. Stačí je tedy nalinkovat do enginu a spustit kompilaci.

Lua byl vyzkoušen z oficiální distribuce, jejíž součástí je vlastní virtuální stroj. Instalace probíhá stejně jako v případě Duktape s tím rozdílem, že je zde více zdrojových souborů.

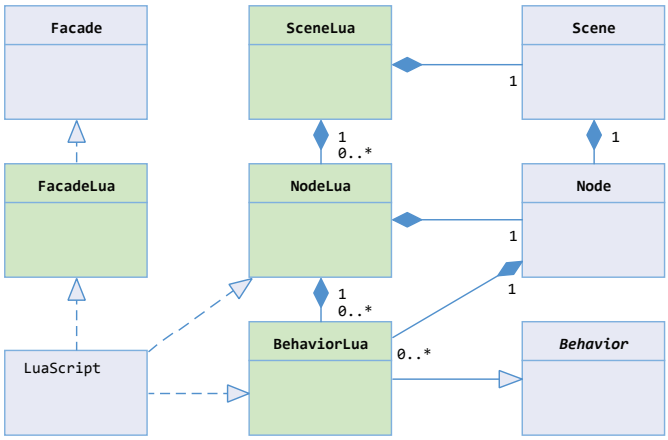
Z těchto dvou jazyků byl nakonec vybrán Lua - především kvůli jeho jednoduchosti a mnohem snazšímu ladění při mapování na C++ API. Lua má navíc o něco menší velikost knihovny - přibližně 63 kB, zatímco Duktape má přes 170 kB [27].

4.2.2 Mapovací komponenta

V Lua se pro komunikaci s jazykem C++ používá zásobník - pokud chce programátor poslat do Lua nějakou hodnotu, vloží ji do zásobníku a zavolá API funkci. Pokud chce naopak z Lua hodnotu získat, zavolá metodu `popStack()`. Pro zjištění konkrétního typu proměnné slouží pomocné metody jako `lua_istable()`.

Pro mapování proměnných a tříd byla využita knihovna *luabridge* [19], která původní knihovnu rozšiřuje o řadu užitečných funkcí. Kromě samotného mapování nabízí také jednoduchý přístup k proměnným a funkcím bez nutnosti použití zásobníku.

Jednoduché objekty s několika atributy jako vektor `ofVec2f`, transformace `Trans` či zprávy `Msg` mohou být mapovány přímo. U těch komplexních je nutné použít zástupné objekty, které budou volání funkcí řešit explicitně. Diagram 4.1 zobrazuje mapování objektů herního stromu.



Obrázek 4.1: Zástupné objekty pro Lua skripty

Scéna, herní uzel i *behavior* zde mají svůj zástupný objekt. Globální metody, které v enginu zprostředkovává fasáda `Facade`, jsou mapovány přes třídu `FacadeLua`.

Jedním z důvodů použití zástupných objektů je fakt, že mapování nepodporuje přetížené ani generické metody - například místo metody `GetAttribute<type>` ve třídě `Node` byla pro každý známý typ vytvořena zvláštní metoda ve třídě `NodeLua`, např. `GetAttributeFloat`, `GetAttributeString` atd.

Samotné mapování provádí *komponenta* `LuaScripting`. Jednotlivé skripty jsou načteny a zkompilovány při startu aplikace.

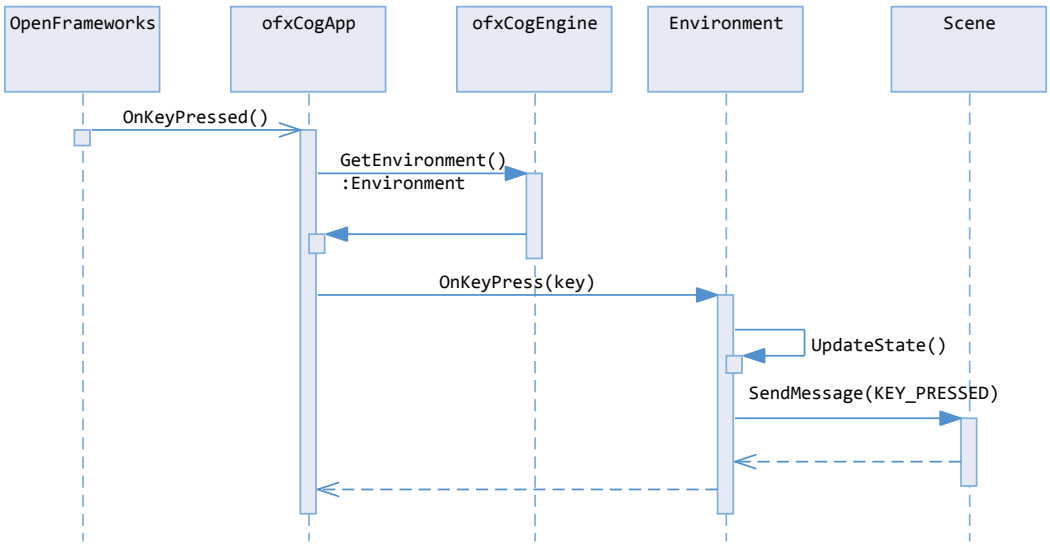
```
getGlobalNamespace(L)
   .beginClass<Flags>("Flags")
   .addConstructor<void(*)(StrId)>()
   .addFunction("HasState", &Flags::HasState)
   .addFunction("SetState", &Flags::SetState)
   .addFunction("SwitchState", &Flags::SwitchState)
   .addFunction("ResetState", &Flags::ResetState)
   .endClass();
```

Ukázka mapování objektu do Lua

Použití skriptů v herní scéně je možno nalézt v doprovodných příkladech - projekty *Scripting*, *Scripting2* a *Scripting3*.

4.3 Vstupní události

Veškeré vstupní události jako je změna velikosti okna, stisk klávesy či změna pozice dotyku, vyvolává sám OpenFrameworks. Posloupnost akcí při stisku klávesy je znázorněna v diagramu 4.2.



Obrázek 4.2: Zpracování událostí

Všechny vstupní události jsou uchovávány v *komponentě* `Environment`, která notifikuje herní scénu odesláním zprávy. Příslušný *behavior* pak může na událost ihned zareagovat ve své metodě `OnMessage`.

4.4 Hashovaný string

Většina objektů jako klíče atributů, názvy stavů a typy akcí zpráv, jsou identifikovány pro rychlé vyhledávání v hashovacích tabulkách pomocí přirozeného čísla (*integer*). Protože by ale bylo v případě definice nových hodnot potřeba dohledávat, jakého rozsahu nabývají aktuální hodnoty, je mnohem výhodnější pro snazší ladění použít řetězce (*string*).

Problém je v tom, že vyhledávání v poli řetězců je mnohem pomalejší než vyhledávání v poli čísel, jelikož při porovnávání řetězců dochází k porovnávání jednotlivých znaků.

Z tohoto důvodu byla vytvořena třída `StrId`, která transformuje *string* na přirozené číslo pomocí rychlého hashovacího algoritmu SDBM [28]. Tuto třídu je možno použít jako klíč hashovacích tabulek a navíc se s ní dá pracovat jako s obyčejnými čísly - konstruktor bere jako parametr typ *string* i *integer*.

Pro účely ladění byla do kódu přidána direktiva, která každý *string* v *debug* verzi ukládá do zvláštní kolekce. Díky tomu je možno pro každý hash dohledat jeho vzorový řetězec a případně jej zapsat do logu.

4.5 Asynchronní procesy

Engine umožňuje spouštět asynchronní procesy, což není nic jiného než vlákno, které běží odděleně od hlavního vlákna aplikace. Každý takový proces musí dědit od třídy `Job` a implementovat metodu `threadedFunction`. Spuštění se provede velice jednoduše zavoláním metody `CogRunProcess(process)`.

Jedním z těchto procesů je třída `SceneLoader`, umožňující asynchronní načítání scény z XML souboru. To je vhodné především u komplikovaných scén, kde se zobrazí nejprve *progress bar*, než dojde k přepnutí načtené scény.

4.6 Přehrávání zvuků

Pro přehrávání zvuků byla vytvořena třída `Soundfx`, která k přehrávání používá objekt `ofSoundPlayer`, jenž je součástí OpenFrameworks. Přehrání souboru je možno provést jednoduše zavoláním metody `CogPlaySound(file)`.

Všechny aktuálně přehrávané zvuky jsou uchovávány ve třídě `Environment` a po skončení přehrávání jsou odstraněny.

Kapitola 5

Multiplayer

V devadesátých letech, kdy ještě nebyl internet příliš rozšířený, se pro hru více hráčů hojně používala technologie známá jako *split screen*, kdy je obraz rozdělen na několik částí a každá tato část představuje herní instanci pro jednoho hráče.

S nástupem internetu a síťových technologií však začal být stále více využíván online multiplayer, ve kterém každý hráč používá své vlastní zařízení. Tato kapitola bude pojednávat právě o tomto typu multiplayeru.

5.1 Síťový model TCP/IP

Počítačová síť je tvořena zařízeními, která spolu komunikují pomocí rodiny protokolů TCP/IP (RFC 1122 [29] a 1123 [30]). Mezi nejznámější protokoly této rodiny patří *Transmission Control Protocol* (TCP), *User Datagram Protocol* (UDP) a *Internet Protocol* (IP).

5.1.1 Internet Protocol

Internet Protocol, popsán v RFC 791 [31], poskytuje komunikační službu bez spojení, která probíhá pomocí datových jednotek, zvaných datagramy. Protože tyto datagramy putují sítí nezávisle, mohou být doručeny v různém pořadí nebo dokonce zduplikovány.

Datagram se skládá z hlavičky a těla, přičemž u protokolu verze 4 (IPv4) má tato hlavička velikost 20 B a u protokolu verze 6 (IPv6) 40 B. Data mohou mít proměnnou velikost.

5.1.2 Transmission Control Protocol

Protokol TCP, popsán v RFC 793 [32], je spojově orientovaný a data přenáší ve formě proudu. Data jsou rozdělena na segmenty, které jsou předány IP protokolu k přepravě.

Protokol zajišťuje spolehlivé doručení dat ve správném pořadí. Pokud se nějaká data ztratí, jsou po uplynutí určité doby odeslána znovu, přičemž druhá strana je průběžně potvrzuje. V případě výpadků pak může TCP samo regulovat rychlost přenosu.

Obě komunikující stanice jsou identifikovány IP adresou a portem, což je celé číslo v rozmezí (0-65535).

Nevýhodou tohoto protokolu je možné zpoždění během čekání na nepotvrzená data a jeho velikost - povinná část hlavičky zabírá 20 B.

5.1.3 Protokol UDP

Protokol UDP, popsán v RFC 768 [33], identifikuje komunikující stanice stejně jako protokol TCP. Nenavazuje ale spojení ani nezajišťuje spolehlivé doručení dat.

Výhodou tohoto protokolu jsou nižší režie na přenos dat, neboť hlavička má velikost pouze 8 B. Z tohoto důvodu se tento protokol často používá pro hru více hráčů po síti, přičemž navázání spojení a potvrzování zpráv je implementováno v aplikační logice.

5.2 Principy multiplayeru

Jak bylo řečeno v kapitole 1, hry jsou simulátory univerza, které se v čase mění. Aby běžely plynule, musí mít aktualizační frekvenci v řádu několika desítek jednotek za sekundu.

V případě síťového multiplayeru pracuje zařízení každého hráče s vlastní instancí tohoto univerza. Jejich synchronizace představuje poměrně velký problém, neboť rychlost přenosu dat po síti má mnohem větší omezení co do rychlosti i kapacity přenosu dat než rychlost přenosu mezi jednotlivými částmi hardware jednoho počítače. Zatímco u operačních pamětí je velikost latence v řadu desítek nanosekund [34], u síťového připojení je to již v řádu desítek mikrosekund [35]. Nemluvě o případech, kdy jsou oba hráči vzdáleni stovky kilometrů a data musí projít přes velké množství směrovačů.

5.2.1 Topologie

5.2.1.1 Client-server model

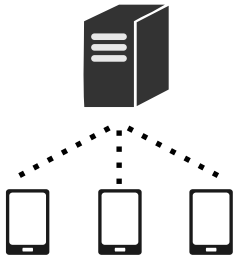
V této topologii existuje centrální jednotka (server), ke které se jednotliví klienti připojují. Tito klienti pak vytváří tzv. *action messages*, které posílají na server. Ten je zpracuje a notifikuje klienty pomocí *update messages* o změně stavu herního modelu.

Komunikace probíhá pouze mezi serverem a klienty, nikoliv mezi klienty samotnými.

Ve *Steam enginu*, který používá client-server topologii, simuluje server hru v diskrétních krocích [36]. Během jednoho kroku zpracuje vstupy, spustí simulaci a aktualizuje stavy. Po aktualizaci pošle klientům notifikaci o změně stavu.

Výhodou této topologie je nižší zatížení sítě a možnost simulovat část akcí pouze na serveru, což může ušetřit výkon u klientů a znesnadnit případné podvádění, neboť žádný z klientů nemusí mít kompletní informace o herním světě.

Nevýhodou jsou vysoké nároky na výkon serveru, protože jedno zařízení musí komunikovat se všemi klienty a aplikovat změny na herní model.



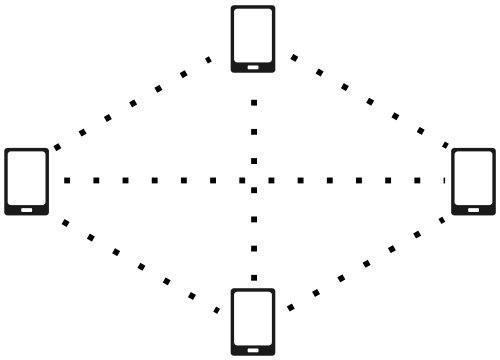
Obrázek 5.1: Client-server topologie

5.2.1.2 Peer-to-peer model

V případě peer-to-peer topologie komunikují všechna zařízení mezi sebou. U tohoto řešení dojde k rozprostření zátěže při simulaci, nicméně zde nastane mnohem vyšší spotřeba šířky pásma. Ta dle [37] roste s každým připojeným klientem exponenciálně.

Navíc zde mohou nastat problémy, kdy dva klienti provedou tutéž operaci ve stejném čase, jejichž soustavné provedení se vzájemně vylučuje. V takovém případě je potřeba rozhodnout, která operace bude provedena a která bude zamítnuta.

Tato topologie byla využita například u realtime strategií jako *Age of Empires* a *Starcraft*, kde hráč ovládá velké množství jednotek.



Obrázek 5.2: Peer-to-peer topologie

5.2.2 Latence

Latenci je možno definovat jako zpoždění mezi odesláním požadavku a jeho přijetím cílovou stanicí, případně mezi odesláním požadavku a přijetím potvrzení o jeho obdržení.

Vysoká míra této hodnoty má negativní dopad na herní zážitek - Michael Powers ve svém článku [37] dokonce poznamenal, že tento zážitek neovlivňuje nic tak významným způsobem jako právě latence.

Ne všechny interakce hráče jsou však k latenci stejně senzitivní. Například odstřelovač střílející na pohybující se cíl bude latencí ovlivněn mnohem významněji než hráč, který chce poslat označené jednotky na určenou pozici.

V článku [38] je problém latence rozdělen na čtyři domény: konzistence herního světa, vizuální konzistence, využití sítě a férovost. V této práci budou diskutovány první dvě.

5.2.3 Konzistence herního světa

U prvních her hratelných po síti jako *Doom*, kde se využívala peer-to-peer topologie, byly vstupy uživatele (stisk klávesy) odeslány všem ostatním klientům a než bylo možno provést simulaci dalšího kroku, bylo nutné počkat na vstupy od všech hráčů, což mělo za následek značné zpomalení hry v důsledku zatížení sítě.

Client-server architektura tento problém vyřešila centralizovaným bodem pro zpracování herní logiky - klient v podstatě nespouštěl žádný kód, pouze odeslal na server vstupy

z klávesnice a myši, server je zpracoval a aktualizoval stav hry. Klient pak po obdržení aktualizační zprávy změnil pozici hráče [39].

I když přítomnost serveru snížila zátěž sítě, bylo nutné vyřešit další problém - pokud například hráč stiskl tlačítko vpřed, musel klient čekat na potvrzení od serveru, než aktualizoval pozici hráče. Toto zpoždění bylo znatelné především u rychlých akcí hráče.

John Carmack, který stál za vývojem kultovních her jako *Doom* či *Quake*, navrhnul predikční model, kdy klient bezprostředně po vstupní akci uživatele provede aktualizaci hráče a po přijetí aktualizační zprávy od serveru pak tuto pozici upraví.

I když by klient mohl pozici hráče nastavovat sám a serveru pouze posílat aktualizaci pozice, takový přístup je nepřípustný, neboť by tak bylo velmi snadné ve hře podvádět. Jedinou autoritativní jednotkou musí být vždy server.

5.2.4 Vizuální konzistence

Protože aktualizace herního modelu probíhá v řádu desítek jednotek za sekundu, je potřeba tuto frekvenci dodržet i v případě synchronizace mezi klientem a serverem.

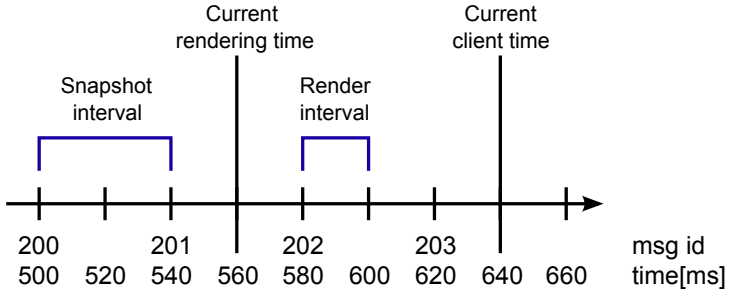
Zasílání celého herního modelu v tomto intervalu však není u běžných her možné, neboť naráží na limit šířky pásma. Proto se místo celého modelu posílají jen rozdílové informace oproti minulým stavům. I tak je ale počet takový zpráv odeslaných za jednu sekundu mnohem menší než aktualizační frekvence hry - například server ve *Steam enginu* posílá aktualizace 20 krát za sekundu, zatímco model je aktualizován v průměru 60 krát za sekundu.

Drobné zpoždění herních akcí jako příkaz k pohybu jednotek či spuštění animace nemá na herní zážitek významný vliv, problém však nastává u objektů, jejichž atributy se významně mění po každé aktualizaci, například pohyb.

Klient ani server nemohou obsahovat ve stejný okamžik totožné informace o stavu herního světa. Tento problém není možné z fyzikálních principů vyřešit a techniky, které se používají, slouží pouze k zakrytí těchto nedostatků před zraky hráčů.

Jednou z nich je takzvaná interpolační rezerva - stav herního modelu na klientovi je oproti stavu na serveru uměle opožděn. Díky tomu klient během vykreslení herní scény ví, v jakém stavu se bude scéna nacházet o několik desítek milisekund později.

Na obrázku 5.3 je zobrazeno časové okno s čísly jednotlivých aktualizačních zpráv, které odeslal server. Zatímco hráč vidí herní model v čase 560 ms, skutečný čas je 640 ms. Díky tomu může klient příštích 100 ms vycházet z údajů podle zpráv 202 a 203. Navíc pokud by během přenosu došlo ke ztrátě zprávy 202, můžeme použít údaje ze zpráv 201 a 203.



Obrázek 5.3: Umělé zpoždění při synchronizaci

5.2.4.1 Interpolace a extrapolace

Společně s touto technikou se při aktualizaci často měnících se veličin používají metody známé jako *interpolace* a *extrapolace*.

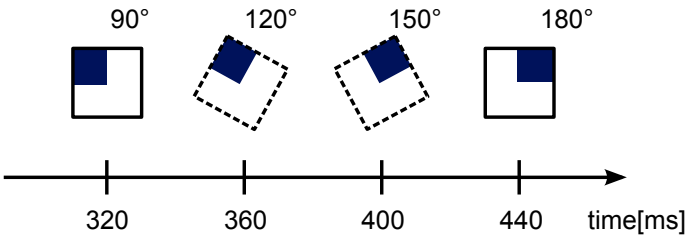
Interpolace je metoda, která dokáže určit hodnotu veličiny v intervalu, jehož krajní body jsou předem známy. Na rozdíl od aproximace musí hledaná křivka procházet všemi známými body.

Máme-li funkci $f(x)$, pro kterou jsou známy hodnoty $f(x_0), f(x_1), \ldots, f(x_n)$, znamená interpolace nalezení hodnoty $f(x)$ takové, že platí $x_0 \leq x \leq x_n$.

Nejjednodušším typem interpolace je lineární interpolace, spočívající v proložení sousedních bodů přímkou. Pokud známe hodnoty funkce $f(x_0) = y_0$, $f(x_1) = y_1$, platí pro hodnotu funkce f v bodě x, kde $x_0 \leq x \leq x_1$, následující rovnice:

$$f(x) = y_0 + (x - x_0) \cdot ((y_1 - y_0)/(x_1 - x_0)) \tag{5.1}$$

Pokud měl například objekt v čase 320 ms rotaci o hodnotě 90° a v čase 440 ms 180° , můžeme spočítat jeho rotaci v tomto časovém okénku, jak ukazuje obr. 5.4.



Obrázek 5.4: Příklad interpolace

Použití interpolace předpokládá, že klient zná stav modelu v čase pozdějším než je čas vykreslování. Pokud však dojde k výpadku některých zpráv a informace o budoucím stavu chybí, použije se extrapolace.

Extrapolace funguje podobně jako interpolace s tím rozdílem, že hledaná hodnota leží mimo interval známých hodnot. Pokud není pohyb herních objektů příliš chaotický, dokáže extrapolace spolehlivě předpovědět jejich pohyb o několik milisekund dopředu.

5.2.4.2 Problém teleportace

Při synchronizaci je důležité rozlišit, které hodnoty jsou spojité a které nikoliv. Představme si objekt, který se plynule pohybuje z levé části okna do pravé části. Jakmile dorazí do pravé části, bude teleportován zpět do levé části.

Při interpolační synchronizaci by tato animace na přijímací stanici vypadala tak, že se objekt rychle přesune do levé části okna. Proto je důležité, aby klient při synchronizaci detekoval, zda má být hodnota interpolována či nastavena přímo.

5.2.5 Potvrzování zpráv

Protokol UDP je nespolehlivý. Datagramy mohou dorazit zpřeházené, zduplikované, případně se mohou ztratit, s čímž musí aplikace počítat.

Přijímací stanice musí všechny přijaté zprávy potvrdit vysílací stanici. Pokud nejsou potvrzeny v určitém časovém intervalu, vysílací stanice je musí odeslat znovu. Velikost okénka, představující počet zpráv, na jejichž potvrzení stanice čeká, než odešle další, je možno zvolit podle potřeby aplikace.

U multiplayeru můžeme zprávy rozdělit do dvou kategorií - akční a aktualizační.

Akční zprávy (například vytvoření nového objektu, změna stavu apod.) jsou kritické pro konzistenci herního modelu a musí být vždy potvrzeny. Oproti tomu aktualizační zprávy, ze kterých je možno zrekonstruovat aktuální stav modelu, potvrzovány být nemusí. Pokud vysílací stanice odesílá hodnoty veličin objektů v čase X, není již potřeba, aby odesílala i hodnoty v čase menším než X.

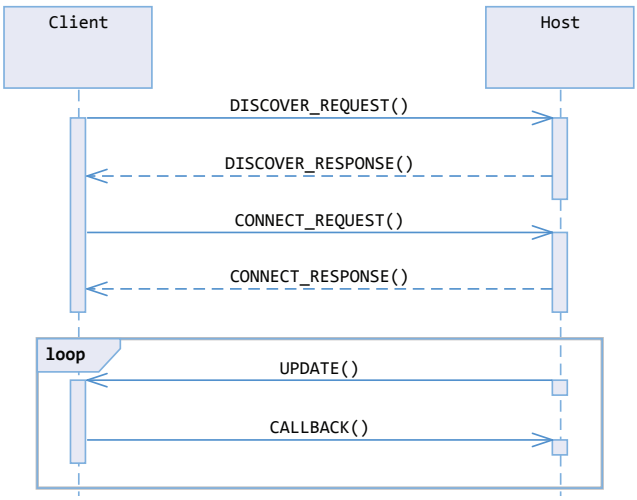
5.3 Návrh řešení

Návrh probíhal společně s návrhem síťové komponenty pro prototyp hry, o které bude řeče v kapitole 7.

Z hlediska multiplayeru byla zvolena architektura peer-to-peer, neboť řešení bylo navrženo pro hru dvou hráčů, z nichž každý sdílí částečný model herního světa. Pro přenos dat bude využit protokol UDP.

5.3.1 Navázání spojení

Navázání spojení je znázorněno na obrázku 5.5. Komunikační stanice může být dvojího typu: klient, který se připojuje k hostiteli, a hostitel, který čeká na připojení klienta. Po navázání spojení se až do jeho ukončení obě stanice chovají totožně.



Obrázek 5.5: Navázání spojení

Na začátku se klient nachází ve stavu `Discovering` a hostitel ve stavu `Listening`. Komunikaci iniciuje klient posláním zprávy všem stanicím (*broadcast*). Jakmile hostitel jeho zprávu obdrží, pošle odpověď, která už může obsahovat nějaká data (např. zvolená frakce či název hrací mapy).

Jakmile se klient bude chtít připojit, pošle zprávu `CONNECT_REQUEST` a přepne se do stavu `Connecting`. Když hostitel obdrží od klienta zprávu o tom, že se chce připojit, odešle zprávu `CONNECT_RESPONSE` a přepne se do stavu `Communicating`. Do tohoto stavu se poté přepne i klient, jakmile obdrží od hostitele odpověď.

Hostitel může klienta kdykoliv odpojit, případně se klient může odpojit sám zasláním zprávy `DISCONNECT`. K automatickému odpojení může dojít po vypršení časového intervalu.

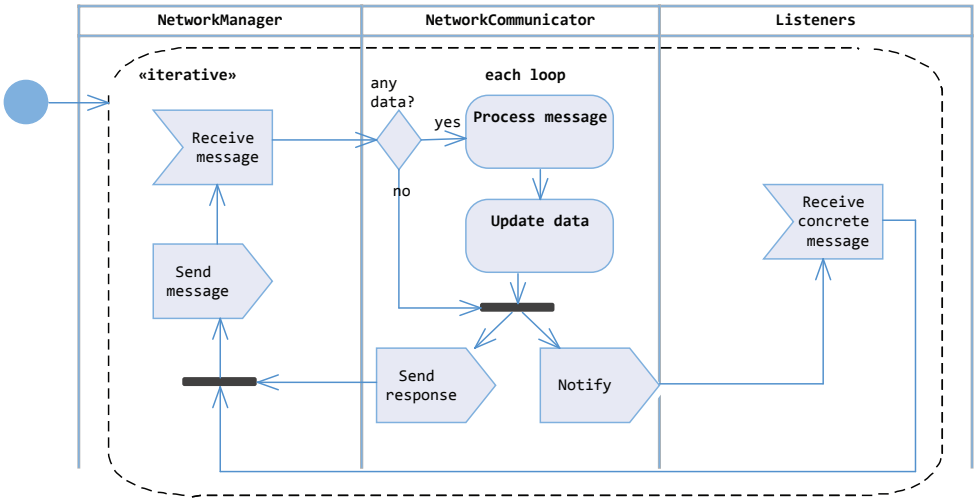
5.3.2 Komunikace

Zpracování zpráv bude probíhat ve třech vrstvách, které jsou popsány v diagramu 5.6. Každá zpráva se bude skládat ze dvou částí - hlavičky, kterou bude zpracovávat druhá vrstva pro aktualizaci stavu komunikace a výměny zpráv, a data, kterou bude zpracovávat konkrétní aplikace.

První vrstva, `NetworkManager`, bude sloužit k odesílání a přijímání zpráv s využitím daného protokolu. Na této vrstvě dojde také k deserializaci proudu přijatých bajtů na obecnou zprávu.

Druhá vrstva, `NetworkCommunicator`, bude zpracovávat hlavičku zprávy - aktualizuje seznam zpráv, které již byly potvrzeny a odešle data určena k odeslání. Zároveň o přijaté zprávě notifikuje třetí vrstvu, kterou bude představovat jakákoliv *komponenta* či *behavior* na tuto zprávu reagující. Zde může dojít k deserializaci datové části zprávy a jejímu dalšímu zpracování.

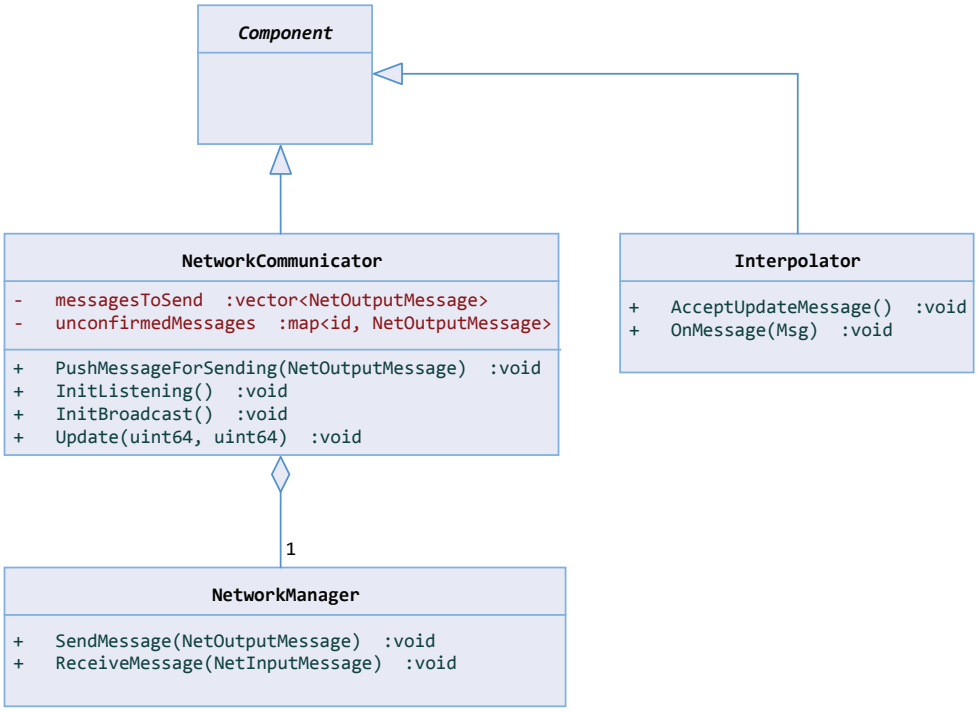
Aby bylo udrženo spojení, vysílací stanice bude odesílat průběžné zprávy i v případě, kdy nedostala od vyšší vrstvy žádná konkrétní data.



Obrázek 5.6: Komunikace síťových komponent

5.4 Implementace

Základní synchronizaci dat zprostředkovávají tři třídy, z nichž každá reprezentuje jednu vrstvu popsanou ve fázi návrhu: `NetworkManager`, který přijímá a odesílá zprávy, *komponenta* `NetworkCommunicator`, zajišťující navázání spojení a synchronizaci, a `Interpolator` pro interpolaci spojitých veličin.



Obrázek 5.7: Architektura síťové komunikace

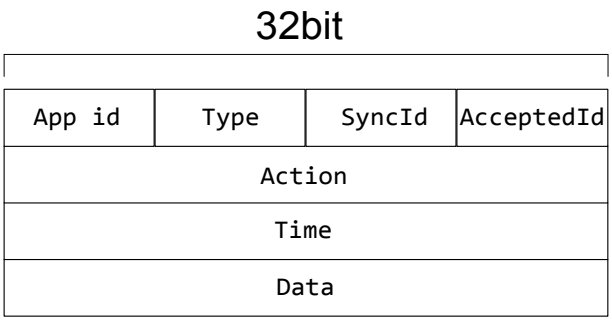
5.4.1 NetworkManager

Třída `NetworkManager` přistupuje ke knihovnám implementujících různé síťové protokoly. Tuto třídu je možno použít pro otevření či navázání spojení přes protokoly UDP, TCP a HTTP, nicméně pro účely multiplayeru bude využit pouze protokol UDP.

Při přijetí zprávy je tato deserializována z proudu bajtů na objekt, obsahující parametry dle obrázku 5.8. Každá zpráva má bez datové části velikost 12 B a obsahuje následující atributy:

- **App id** - identifikátor aplikace; slouží pro odlišení jednotlivých aplikací komunikujících přes stejný port.
- **Type** - typ zprávy; jednotlivé typy jsou popsány níže.
- **SyncId** - synchronizační identifikátor, který generuje vysílací stanice. Přijímací stanice pak musí tento identifikátor uvést v nějaké budoucí zprávě, aby zprávu potvrdila.

- **AcceptedId** - identifikátor přijaté zprávy. Je nastaven, pokud je potřeba nějakou přijatou zprávu potvrdit.
- **Action** - akce, kterou zpráva představuje. Podle tohoto atributu je pak možno deserializovat datovou část na konkrétní typ, pokud je k dispozici. Tento atribut využívá především třetí vrstva.
- **Time** - čas, ve kterém byla zpráva odeslána. Udává počet milisekund od spuštění aplikace.
- **Data** - libovolná data , deserializaci musí provést třetí vrstva.



Obrázek 5.8: Tvar zprávy

Typy zpráv:

- **DISCOVER_REQUEST** - posílá klient, když chce zkontaktovat hostitele.
- **DISCOVER_RESPONSE** - odpovídá hostitel na DISCOVER_REQUEST
- **CONNECT_REQUEST** - posílá klient, když se chce připojit k hostiteli.
- **CONNECT_RESPONSE** - odpovídá hostitel na CONNECT_REQUEST
- **UPDATE** - posílá kdokoliv, kdo potřebuje druhé stanici zaslat aktualizační informace.
- **ACCEPT** - posílá kdokoliv, kdo nemá žádná data k zaslání, ale zároveň potřebuje nějaká přijatá data potvrdit.
- **DISCONNECT** - posílá hostitel, když odpojí klienta, případně klient, pokud se odpojí od hostitele.

5.4.2 NetworkCommunicator

Tato třída implementuje protokol, který obohacuje UDP o synchronizaci a potvrzování. Od ostatních *komponent* přijímá data k odeslání.

Jakmile je potřeba nějaká dříve přijatá data potvrdit, vyplní se u právě odeslané zprávy atribut `AcceptedId`. Vysílací stanice si uchovává zprávy, které doposud nebyly potvrzeny a v pravidelných intervalech je odesílá znovu.

Jak bylo již poznamenáno, u multiplayeru se zprávy rozlišují na akční a aktualizační. Akční musí být potvrzeny vždy, nicméně u těch aktualizačních, kde v případě ztráty zprávy nehrozí trvalá nekonzistence synchronizovaného modelu, se na potvrzování nečeká.

Jelikož akční zprávy do jisté míry reflektují aktivitu uživatele či akce na herní scéně, je pravděpodobné, že aktualizačních zpráv, které nemusí být potvrzovány, bude daleko více.

5.4.3 Interpolator

Třetí vrstvu může tvořit jakákoliv *komponenta*, která bude reagovat na zprávy přijaté ze sítě. Do enginu byla implementována jedna z nich - `Interpolator`, kterou je možno použít obecně pro interpolaci spojitých veličin.

Třída si uchovává dva poslední vzorky zpráv, které obsahují hodnoty spojitých veličin v určitém čase. Mezi těmito vzorky provádí interpolaci, přičemž rychlost této interpolace může mírně zrychlit i zpomalit v závislosti na tom, s jakým zpožděním zprávy přichází.

Kapitola 6

Umělá inteligence

Ačkoliv se hry po grafické stránce neustále zlepšují, mnohdy až na fotorealistickou úroveň, na poli umělé inteligence není pokrok až tak markantní a i dnes je možné setkat se s hrami, ve kterých je její úroveň natolik slabá, že kazí herní zážitek (příklad porovnání: [40]).

Pro umělou inteligenci doposud neexistuje obecně přijímaná definice, spíše se jedná o volné sdružení různorodých teorií a technik. Jeden z jejích průkopníků, Marvin Minsky, tento pojem definoval jako *vědu o vytváření systémů, které budou při plnění úkolu volit takový postup, který bychom považovali za projev inteligence, kdyby jej prováděl člověk* [41].

V oblasti her se pod pojmem *umělá inteligence* většinou rozumí ta část programu, která řídí chování herních entit. Takový program může využívat neuronové sítě, strojové učení, prohledávací algoritmy ale také sadu předpřipravených skriptů a jednoduchých pravidel.

Tato kapitola probírá jednotlivé techniky používané při vývoji her, jejichž cílem je naprogramovat chování herních objektů. Dále bude o umělé inteligenci pojednáváno pod zkratkou AI (ze slov *Artificial Intelligence*).

Nejprve bude ve stručnosti uveden žánr RTS her, do kterého bude spadat prototyp hry, kterému je věnována kapitola 7. Z hlediska AI budou diskutovány problémové domény, kterým budou věnovány ostatní části kapitoly a které budou rovněž implementovány jako součást enginu.

6.1 Úvod do strategických her

Real-time strategie jsou hry běžící v reálném čase, ve kterých se hráč snaží maximalizovat svůj zisk prováděním různých akcí. Může se jednat o budovatelské hry (simulátory), kde se snaží nakládat s aktivy ve prospěch svého světa, nebo válečné hry (šarvátky), kde je cílem získat strategickou převahu v oblasti jednotek a surovin, případně zničit protivníka.

Drtivá většina takových her se odehrává na nějaké mapě, která reprezentuje herní svět. Na této mapě se mohou nacházet zdroje surovin, cesty a nepřístupné oblasti (pohoří, moře) - v každém případě se jedná o terén, který musí být hráčem a AI analyzován, na základě čehož pak mohou naplánovat svoji strategii.

Dále hra může obsahovat suroviny, jejichž nedostatek může mít za následek zablokování stavby jednotek a proto je potřeba vhodně volit priority během hry, aby nedošlo k ekonomickému vyčerpání hráče.

Dalším elementem jsou obvykle jednotky. Pohyb jednotek po mapě vyžaduje znalost

terénu a dostupnost jednotlivých oblastí. Každá jednotka může mít svůj specifický účel (boj, těžba) a v některé fázi hry se vyplatí budovat jiné jednotky než na jejím začátku. Například velké množství těžebních jednotek může hráči přinést ekonomickou výhodu, ale také slabší strategické postavení. Pokud by protihráč rychle postavil válečné jednotky, snadno by zničil slabšího protivníka.

Velké množství her obsahuje také základnu. Jedná se o síť budov, používanou za účelem stavby nových jednotek a těžby surovin. Budování základny může být poměrně komplikovaný proces, obzvlášť když se odehrává v nepravidelném terénu.

6.1.1 Problémové domény

Žánr RTS nabízí mnoho zajímavých domén z hlediska výzkumu umělé inteligence, od možností kooperace herních entit až po strojové učení strategie. Zde je seznam těch, kterými se bude zabývat tato práce:

- **Hledání cest** - toto je pravděpodobně jedna z nejdůležitějších komponent pro strategické hry, u které je navíc kladen velký důraz na optimalizaci z důvodu frekventovaného používání.

- **Pohyb objektů** - v této práci bude představena jedna z oblíbených metodik: *steering behaviors*

- **Rozhodovací procesy** - zatímco simulaci pohybu a hledání cest provádí i jednotky hráče, rozhodovací procesy jsou tím typem AI, který supluje jeho skutečnou aktivitu. Existuje mnoho variant jak rozhodovací procesy implementovat, v této práci budou popsány tři z nich - konečné automaty, *goal-driven behavior* a algoritmy prohledávání stavových prostorů.

6.2 Hledání cest

Problém hledání cest je úzce spjat s teorií grafů. Herní mapa je obvykle transformována do množiny uzlů a vrcholů a problém nalezení cesty na mapě je pak přenesen na problém nalezení cesty v grafu.

Ohodnocený orientovaný graf (G, w) je orientovaný graf G spolu s reálnou funkcí $w : E(G) \to (0, \infty)$. Je-li e hrana grafu G, číslo $w(e)$ se nazývá její *ohodnocení* nebo *váha*.

Algoritmy na hledání cest v grafu je možno rozdělit na informované a neinformované [42]. Neinformované algoritmy na rozdíl od informovaných neberou v potaz váhu jednotlivých hran a nemají k dispozici žádné dodatečné informace, které by jim usnadnily cestu k cíli. Procházení uzlů tak musí provádět systematicky, dokud nenaleznou řešení. U informovaných algoritmů určuje pořadí prohledávání heuristika.

Mezi neinformované patří například algoritmus prohledávání do šířky (*breadth-first search* BFS) a do hloubky (*depth-first search* DFS).

K těm informovaným patří především Dijkstrův algoritmus na nalezení nejkratší cesty v grafu a algoritmus A*, který využívá navíc odhad vzdálenosti.

6.2.1 Neinformované prohledávací algoritmy

Algoritmus prohledávání do šířky začne procházet graf z daného uzlu. Pokud z uzlu vede více hran, uloží se do fronty a postupně se prozkoumají jednotlivé hrany. Díky frontě jsou uzly zpracovávány v pořadí daném jejich vzdáleností od kořene.

Oproti tomu algoritmus prohledávání do hloubky používá pole stavů, kde každý uzel je v počátku nastaven jako FRESH, po nalezení uzlu se jeho stav změní na OPEN a po návratu z tohoto vrcholu se tento nastaví jako CLOSED.

Oba algoritmy jsou úplné - vždy naleznou řešení, pokud existuje. Při prohledávání však může být prozkoumáno neúměrně více uzlů, než je potřeba k řešení.

6.2.2 Dijkstrův algoritmus

Dijkstrův algoritmus, popsaný Nizozemcem *Edsgerem Dijkstrou*, slouží k nalezení nejkratší cesty v orientovaném grafu s nezáporným ohodnocením hran. Lze na něj pohlížet jako na zobecněné prohledávání do šířky, kde místo klasické fronty využívá prioritní frontu.

V této frontě mají větší přednost uzly, které jsou blíže ke zdroji. Algoritmus v každém kroku vybere uzel s nejvyšší prioritou a prozkoumá jeho nezpracované potomky, pro které ověřuje vzdálenost. Jakmile jsou zpracovány všechny uzly, algoritmus skončí.

6.2.3 A* algoritmus

Dijkstrův algoritmus dokáže najít nejkratší cestu, plýtvá ale časem procházením uzlů ve špatném směru, což není na škodu v případě, kdy hledáme cestu do všech uzlů. Pokud ale hledáme cestu mezi dvěma konkrétními uzly, je vhodné expanzi uzlů provádět jen v jednom směru, který považujeme za správný.

Algoritmus A* pracuje s prioritní frontou velmi podobně jako Dijkstrův algoritmus s tím rozdílem, že pro řazení priorit používá funkci, která je součtem heuristické funkce posledního uzlu cesty a její zbývající délky.

Hodnota heuristické funkce musí být nižší nebo rovna skutečné vzdálenosti z daného uzlu do cíle. Obvykle se volí *Manhattanská* nebo *Euklidovská* vzdálenost [42].

Asymptotická složitost závisí na použité heuristické funkci a na průměrném počtu následníků. V případě optimální heuristiky je složitost $O(n)$, kde n je délka cesty.

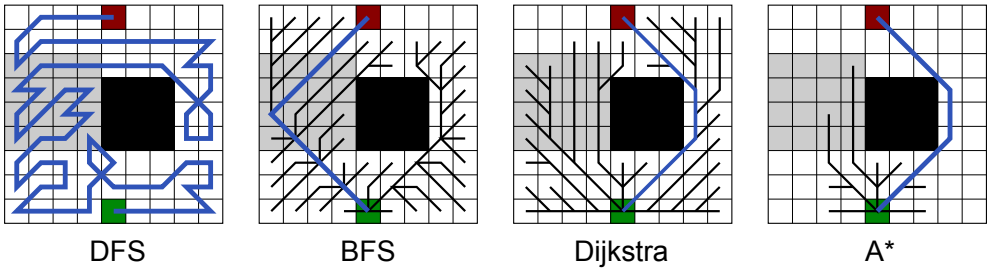
```
queue.add(start)

while(!queue.empty)
 actual = queue.pop()
 for each neighbor in actual.neighbors
   if neighbor is goal then return
   neighbor.g = actual.g + distance(neighbor, actual)
   neighbor.h = distance(neighbor, goal)
   neighbor.f = neighbor.g + neighbor.h
   queue.add(neighbor)
  end
end
```

Pseudokód algoritmu A*

6.2.4 Porovnání algoritmů

Na obrázku 6.1 je zobrazen mřížkový graf o rozměrech 10x10 a průchody jednotlivých algoritmů při vyhledávání. Buňky mřížky představují uzly grafu, černé spojnice značí expanzi jednotlivých uzlů a nalezená cesta je vyznačena modře. Neprůchodnou oblast představují černé čtverce a šedé čtverce mají cenu za překročení dvakrát vyšší než bílé čtverce.



Obrázek 6.1: Hledání cest v grafu

Je vidět, že DFS ani BFS neberou v potaz váhy. Dijkstrův algoritmus a A* dospěly ke stejnému výsledku, ten Dijkstrův ale potřeboval prohledat více uzlů.

6.2.5 Implementace

Pro implementaci byl zvolen algoritmus A*, jelikož se díky heuristické funkci skvěle hodí pro hledání cest na herní mapě - mapou můžeme rozumět mřížkový graf s osmi následníky pro každý uzel.

Algoritmus A* implementuje třída `AStarSearch` s metodou `Search`, která nalezne cestu mezi dvěma uzly mřížkového grafu. Heuristickou funkcí je Manhattanská vzdálenost.

Pokud cesta neexistuje nebo je algoritmus limitovaný maximálním počtem iterací, vrátí dosažitelný uzel z počátečního uzlu, který byl během prohledávání nejblíže koncovému uzlu.

6.3 Steering behaviors

Steering behaviors je sada metod popisujících realistický pohyb autonomních agentů pomocí kombinace silových vektorů. Idea byla navržena Craigem Reynoldsem v roce 1987, kdy publikoval článek *Flocks, Herds and Schools: A distributed Behavioral Model* [43].

Autonomní agent je entita existující v nějakém prostředí se schopností autonomně plnit určité cíle na základě vnímání prostřednictvím senzorů. Zároveň dokáže dle svých možností toto prostředí měnit, aby se přibližoval k naplnění svých cílů.

Chování takových agentů je možno rozdělit do tří vrstev: *action selection* pro plánování strategie, *steering* pro zvolení cesty k cíli a *locomotion* pro způsob dosažení cíle.

Příkladem může být těžba surovin v nějaké strategické hře: *action selection* zde bude přesun jednotky ze základny do oblasti s minerály, *steering* zase zvolení konkrétní cesty (po silnici, po moři) a *locomotion* způsob pohybu (vozidlem, pěšky).

Steering behaviors se zaměřuje především na druhou a částečně i třetí vrstvu - *steering* a *locomotion*.

6.3.1 Návrh pohybového modelu

Pro manipulaci s objektem ve 2D prostoru je potřeba entita, která bude reprezentovat jeho pohybový model. Pozice již je v herní scéně součástí atributu `Transform` - k ní bude potřeba vytvořit atribut se dvěma proměnnými: rychlost a množina sil na objekt působících.

Pokud neuvažujeme hmotnost (resp. hmotnost bude jednotková), výslednice působících sil bude udávat aktuální zrychlení objektu a. Rychlost a pozici je pak možno vypočítat pomocí rovnic klasické mechaniky:

$$\begin{aligned} v &= v_0 + at \\ s &= s_0 + v_0 t \end{aligned} \tag{6.2}$$

Pokud bychom uvažovali proměnnou *delta* udávající čas uplynulý od poslední iterace, *velocity* pro aktuální rychlost objektu, *position* pro pozici a *acceleration* pro aktuální zrychlení, vypadal by výpočet přírůstku pozice a rychlosti takto:

```
velocity += acceleration*delta;
position += velocity*delta;
```

Podle vektoru rychlosti by poté měla být nastavena také radiální rychlost rotace objektu, aby se otáčel ve směru této rychlosti.

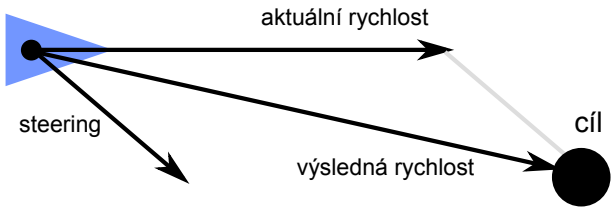
6.3.2 Druhy steering behaviors

Craig Raynolds ve svém článku [44] popsal celkem 15 druhů steering behaviors, od jednoduchých funkcí popisujících pohyb jednotlivých agentů až po komplikovaná pravidla pro vyhýbání se překážkám a kooperaci ve skupině. Zde bude popsáno pět základních, která budou později implementována.

6.3.2.1 Seek

Funkce *seek* vrací sílu působící na agenta, která ji navádí směrem k cílové pozici. Na obrázku 6.2 je tato síla znázorněna vektorem *steering*, který upravuje aktuální rychlost tak, aby objekt směřoval k cíli.

Jakmile se objekt dostane za cíl, začne zpomalovat, poté se otočí a znovu zamíří k cíli.



Obrázek 6.2: Seek

6.3.2.2 Arrive

Funkce *arrive* vylepšuje funkci *seek* o zastavovací efekt. Kromě cíle je zde parametrem také vzdálenost, od kdy má agent začít zpomalovat. Zpomalení zajišťuje síla *arrival force*. Aby nezastavil okamžitě, je potřeba spočítat zpomalení podle poloměru zpomalující oblasti.

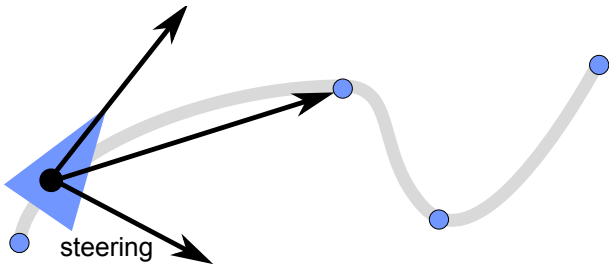
6.3.2.3 Flee

Tato funkce je inverzní funkcí k *seek*. Zatímco *seek* navádí agenta k cíli, funkce *flee* se jej snaží dostat pryč.

6.3.2.4 Follow

Funkce *follow* navádí agenta podél cesty, kterou tvoří tzv. záchytné body (*waypoints*). Z herního pohledu je možno tuto funkci použít např. v RTS pro hlídkování jednotky na vyznačené cestě.

Pomocí funkce *seek* je agent naváděn k jednotlivým záchytným bodům, po jejichž dosažení dojde k přepnutí na další bod. U posledního se pak použije funkce *arrive* pro jeho zastavení.



Obrázek 6.3: Follow

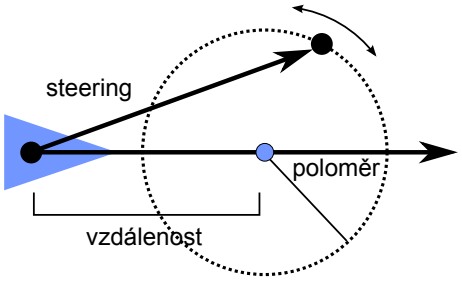
6.3.2.5 Wander

Jedná se o funkci simulující náhodný pohyb. Aby byl zároveň realistický, je potřeba rychlost náhodné změny vektoru rychlosti korigovat.

Jednou z možných implementací, kterou navrhnul Craig Raynolds, je domnělá kružnice, která se nachází před agentem, jak ukazuje obrázek 6.4. Na této kružnici se nachází bod, který určuje směr vektoru rychlosti.

Tento bod se náhodně pohybuje po kružnici a s ním i vektor rychlosti. Výsledné chování pohybu agenta pak určuje poloměr kružnice, rychlost změny pozice bodu na kružnici a vzdálenost kružnice od agenta.

Pokud je žádoucí, aby objekt téměř neměnil směr, nastavíme kružnici do delší vzdálenosti s malým poloměrem a nízkou rychlostí pohybu bodu po kružnici. Pokud naopak potřebujeme častou ale nepatrnou změnu směru, nastavíme kružnici do menší vzdálenosti s větším poloměrem a nízkou rychlostí pohybu bodu.

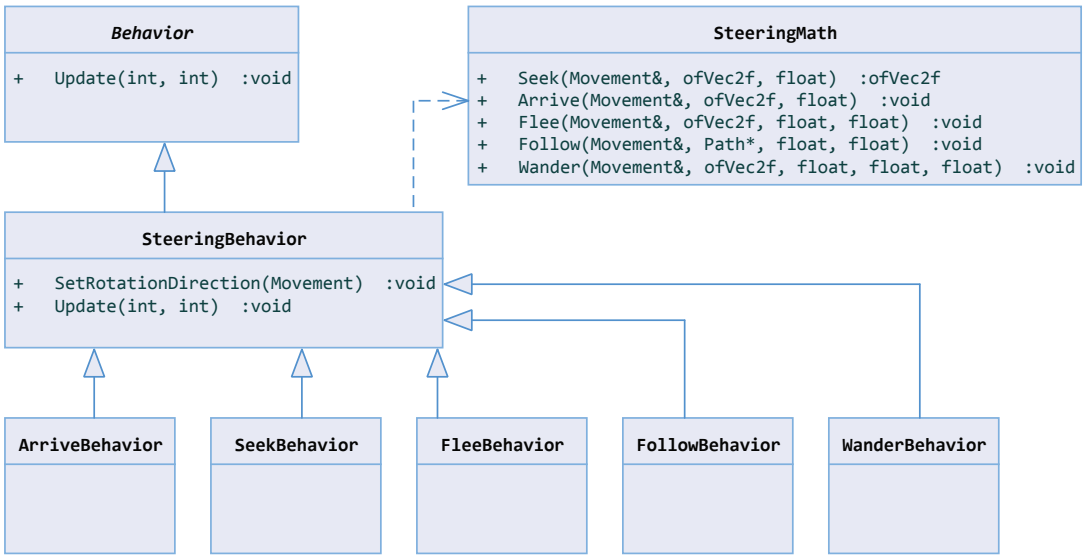


Obrázek 6.4: Wander

6.3.3 Implementace

Steering behaviors byly zaintegrovány do komponentové architektury v podobě *behaviors*, jak ukazuje diagram 6.5.

Všechny tyto objekty pak využívají pro samotný výpočet sil třídu `SteeringMath` s příslušnými funkcemi. Díky tomu mohou být tyto funkce použity i jinými třídami.



Obrázek 6.5: Realizace steering behavior

U funkce *follow* uvažoval původní návrh od Craiga Raynoldse úsečky, které objekt následoval přímo jako vlak na koleji, což není vždy žádané. Pro tuto funkci byla vytvořena třída `Path`, sestávající z kolekce segmentů a podpůrných metod, umožňujících nalézt nejbližší bod na segmentu k aktuální pozici agenta. Okamžik, kdy je agent naváděn k dalšímu záchytnému bodu, je možno konfigurovat. Díky tomu jsou jednotlivé body brány spíše jako reference, které musí agent v určité minimální vzdálenosti minout.

6.4 Řídící procesy

Řídící procesy jsou důležité především pro mikromanagement jednotek, které dostávají příkazy od vyšší vrstvy AI. Ta provádí strategická rozhodnutí, ze kterých plynou jednotlivým entitám určité úkoly (např. stavba mostu, přesun do jiné lokace). Ve hrách, kde strategická vrstva AI není (např. FPS), jednají jednotky obvykle jen na základě svých vlastních senzorů.

Nyní budou popsány dvě techniky, kterými je možné řídící procesy definovat - konečný automat a tzv. *goal-driven behavior*.

6.4.1 Konečný automat

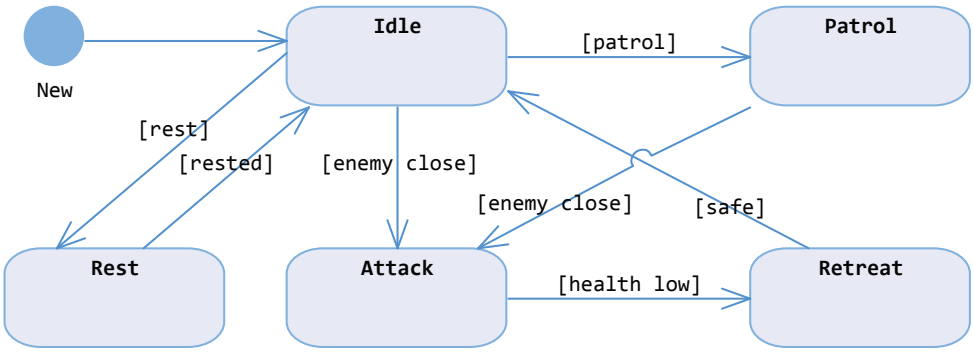
Konečný automat (*finite state machine* FSM) je definován jako *množina stavů, vstupních a výstupních událostí a přechodových funkcí. Přechodová funkce vezme jako parametr aktuální stav, vstupní událost a vrátí množinu nových výstupních událostí a nový aktuální stav* [45].

Ve hrách jsou FSM využívány převážně pro ovládání nehratelných jednotek (*non-playable character*, NPC) a jejich reakce na herní prostředí [46].

FSM se skládá z množiny stavů reprezentujících chování a množiny přechodů specifikujících reakci NPC na herní událost - tzv. *if-this-then-that* proces. Každý stav pak může obsahovat předskriptovanou logiku pro chování agenta, díky čemuž je FSM jednoduchý na porozumění i na implementaci a ladění, neboť je specifikace chování rozložena do malých, snadno spravovatelných částí.

FSM byl použit v mnoha hrách, například *ghost behavior* ve hře *Pac-Man* a NPC jednotky ve hrách *Quake* a *Warcraft* [42].

Na obrázku 6.6 je zobrazen příklad stavového automatu válečníka z nějaké RTS. Pokud nemá zadaný žádný úkol, nachází se ve stavu *Idle*. Na různé události pak reaguje překlopením do příslušného stavu.



Obrázek 6.6: Příklad stavového automatu

Pro menší počet stavů je správa FMS velmi snadná, nicméně s každým dalším stavem jeho složitost narůstá. Kdybychom přidali do diagramu například stav *Dead*, do kterého by se bylo možno dostat z jakéhokoliv stavu, stal by se diagram velmi rychle nepřehledným.

Nevýhodou FSM je tzv. *kombinatorická exploze*, kdy s růstem složitosti systému roste i složitost FSM. K této složitosti přispívá i netečnost vůči sdíleným funkcionalitám - FSM

si neumí poradit s podobnými stavy. Tento problém částečně řeší tzv. hierarchický FSM představený v článku [47], který umožňuje vytvoření kompozitních stavů.

6.4.1.1 Implementace

Do enginu byl naimplementován klasický FSM s možností tzv. globálních stavů a některé jeho nedostatky byly vyřešeny zkombinováním s technikou *goal-driven behavior*, což bude využito při implementaci prototypu hry.

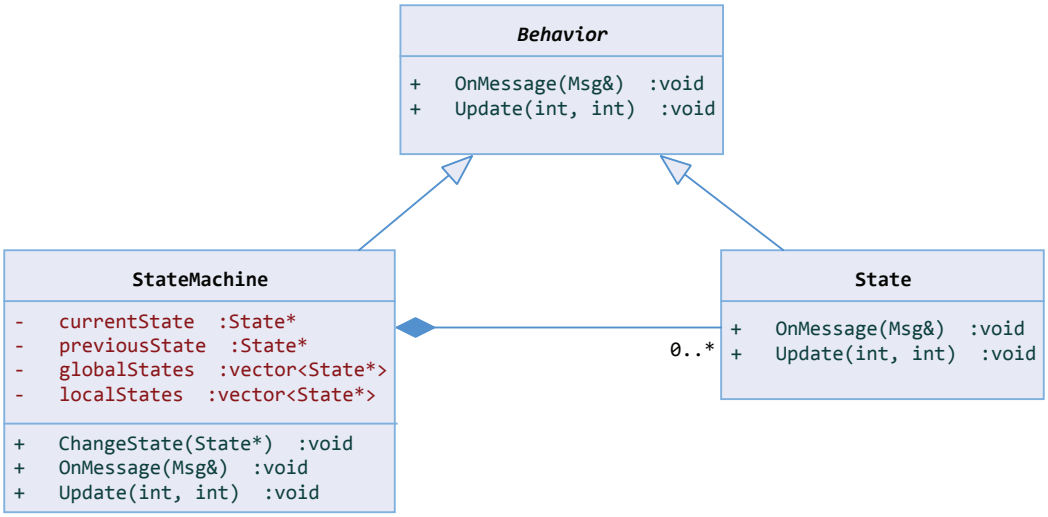
FSM je možno naimplementovat několika způsoby - jako obrovský `switch` s obslužnými metodami pro každý stav, jako přechodovou tabulku nebo množinu tříd, kde každá reprezentuje konkrétní stav.

Pro engine byl zvolen takový způsob implementace, který umožňoval FSM přímo zaintegrovat do komponentové architektury, aby bylo možno využít aktualizační smyčku a systém posílání zpráv. Tento způsob implementace se velmi podobá návrhovému vzoru *Stav*, popsanému v *GoF* [48].

Obě třídy, `StateMachine` i `State`, jsou potomky třídy `Behavior`. `StateMachine` v sobě drží kolekci lokálních stavů, z nichž právě jeden je aktivní - tomuto stavu jsou předávány zprávy metodou `OnMessage` a je aktualizován metodou `Update`.

Krom toho je zde také kolekce globálních stavů, které jsou aktivní vždy a na základě různých událostí mohou kdykoliv změnit aktuální lokální stav.

Veškerá funkcionalita stavu je implementována v konkrétním potomku třídy `State`. K přechodu na jiný stav pak dojde zavoláním funkce `ChangeState` ve třídě `StateMachine`.



Obrázek 6.7: Realizace FSM

6.4.2 Goal-driven behavior

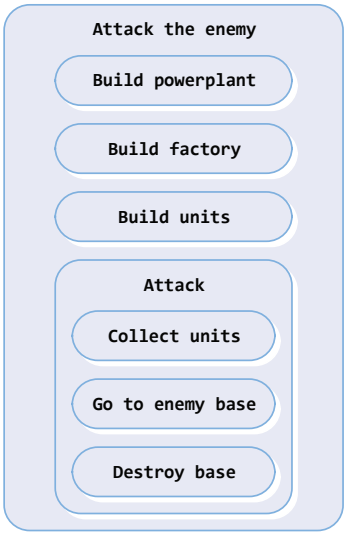
Zatímco FSM definuje chování herních agentů na základě stavů, *goal-driven behavior* používá kompozici tzv. úkolů.

Tato metodika byla zmíněna v knize [42] a je možno ji považovat za kompromis mezi klasickým a hierarchickým FSM. Nese v sobě také některé prvky rozhodovacích stromů a *behavior trees*, popsaných v knize [49]. Z hlediska objektové metodiky se podobá návrhovému vzoru *Composite*.

Každý úkol může být buďto atomický nebo složený - atomické určují jednu konkrétní akci, např. *jdi na pozici X* nebo *seber předmět Y*. Složený úkol je možno nazvat *plánem*, který v sobě sdružuje jednotlivé pod-úkoly. Skládáním pak vzniká strom, kde úkoly tvoří listy a *composite* pak vnitřní uzly.

Jedná se o poměrně intuitivní mechanismus, ve kterém je možno spatřit metodu *rozděl a panuj* - každý plán lze rozložit na podúkoly a ty pak na triviální úkoly. Každá část úkolového stromu se pak dá znovu využít v nějakém dalším plánu.

Na obrázku 6.8 je vidět příklad takového plánu: úkoly na spodnější úrovni je možno dále škálovat, například *Go to enemy base* by v sobě mohl zahrnovat ještě naplánování cesty či seřazení jednotek do formace, *Build units* bychom mohli rozložit na vytváření různých typů jednotek apod.



Obrázek 6.8: Příklad goal-driven behavior

Řešení popsané v knize [42] předpokládá, že po dokončení předchozího úkolu dojde k automatickému přepnutí na úkol následující a pokud dojde k chybě, je provádění celého plánu přerušeno.

Někdy je však žádoucí, aby provádění bylo přerušeno za jiných podmínek - pokud například v nějaké hře definujeme plán pro hledání vhodného řešení k překonání překážky, chceme, aby tento plán skončil, jakmile je řešení nalezeno. Pokud řešením rozumíme sadu triviálních úkolů, z nichž každý se věnuje průzkumu jednoho z řešení, chceme plán ukončit poté, co skončí první úkol, který řešení nalezl - na tomto principu jsou založeny *behavior trees*, popsané v knize [49]. Ty definují dva typy vnitřních uzlů - *selektor*, který provede návrat po prvním splněném úkolu, a *sekvencer*, který provede návrat po prvním nesplněném.

6.4.2.1 Implementace

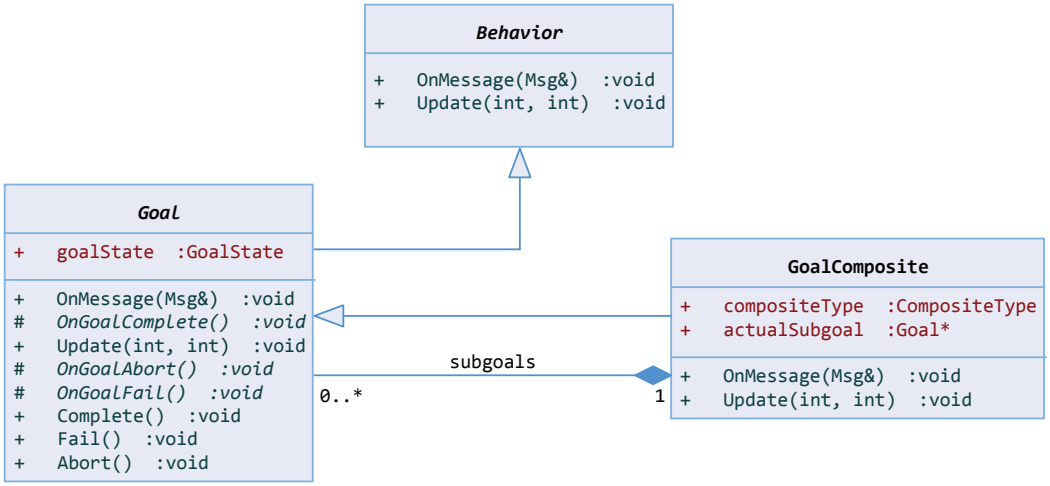
Ve fázi implementace byly zkombinovány obě techniky, *goal-driven behavior* a *decision trees*. Řešení je stejně jako u FSM integrováno do komponentové architektury.

Triviální úkol představuje abstraktní třída `Goal`, od které budou dědit konkrétní úkoly. Logiku zde stejně jako u FSM bude zastávat metoda `Update` a pro doplňující akce je možno přetížit metody `OnGoalComplete`, `OnGoalFail` a `OnGoalAbort`.

Složený úkol reprezentuje třída `GoalComposite`, přičemž atribut `compositeType` specifikuje, jak má reagovat na ukončení podúkolu.

Zde je výčet možných hodnot:

- **SERIALIZER** - úkol skončí, jakmile jsou všechny podúkoly ve stavu COMPLETED nebo FAILED.
- **SELECTOR** - úkol skončí po překlopení prvního podúkolu do stavu COMPLETED
- **SEQUENCER** - úkol skončí, jakmile se první podúkol překlopí do stavu FAILED.



Obrázek 6.9: Architektura goal-driven behavior

6.5 Rozhodovací procesy

Obě předchozí techniky jsou velmi dobře použitelné pro management jednotek a plnění úkolů s předem daným postupem. V RTS hrách je ale také potřeba přítomnost AI, která provádí strategická rozhodnutí.

Možností a technik, jak napodobit chování reálného hráče, existuje celá řada. Všechny techniky vychází z nějaké analýzy prostředí, kterým může být buďto celý herní model nebo jeho abstrakce. Po analýze takového prostředí je z množiny proveditelných akcí vybrána ta, která se zdá být pro danou situaci nejvýhodnější.

6.5.1 Skriptování

Rozhodovací AI může být naprogramována pomocí skriptu či systému pravidel, které řídí akce počítačového hráče. Předem se definuje chování pro dané prostředí a reakce na určité události. Zde se však předpokládá, že hráč bude hrát způsobem, který se od něj očekává.

Skripty pak mohou dosahovat i statisíce řádků kódu, aby pokryly všechny možnosti chování protihráče. Navíc je potřeba zásah expertů, kteří tyto skripty nadefinují a otestují. Přesto se stále hojně využívají ve strategických hrách v kombinaci s dalšími podpůrnými metodami. Příkladem může být hra *Starcraft* se skriptovaným managementem jednotek, která byla zkoumána v rámci mnoha prací, snažících se o rozšíření AI o obecnější techniky jako neuronové sítě [50] či *monte carlo* řízení [51].

Výhodou skriptů je poměrně snadné ladění, neboť se dá jednoduše vytrasovat, proč a na základě čeho se počítačový hráč rozhodl danou akci provést.

Přístupy založené na obecné analýze, které jsou známé především u deskových her, pracují obvykle s tzv. *stavovým prostorem*.

6.5.2 Stavový prostor

Stavový prostor je n-tice $\langle S, k, C, A\rangle$, kde S je množina stavů, $k \in S$ je počáteční stav, $C \subseteq S$ je množina koncových stavů a A je konečná množina operátorů (akcí) [52].

Nalézt řešení úlohy znamená najít takovou posloupnost operátorů $a_1, a_2, \ldots, a_n$, jejichž aplikováním z počátečního stavu k dostaneme nějaký koncový stav $s \in C$.

Stavový prostor lze reprezentovat orientovaným grafem $G = (V, E)$, kde uzel reprezentuje stav a hrana reprezentuje přechod mezi stavy.

Na základě znalosti stavového prostoru pak můžeme vytvářet *herní strom*, což je množina všech možných her, které mohou být z daného stavu (kořene) odehrány.

Příklad stavového prostoru je rozestavění figurek na šachovnici. Počátečním stavem pak bude původní rozmístění figurek, koncovým stavem prohra jednoho z hráčů či remíza a operátory představují jednotlivé tahy.

6.5.3 Heuristická funkce

Protože cílem prohledávání stavového prostoru není vždy nalezení akce, která vede k vítězství hráče, ale akce, která mu zajistí výhodnější strategickou pozici, je potřeba každý stav stavového prostoru ohodnotit.

K tomu slouží heuristická funkce, která vrací číselnou hodnotu popisující, jak je daný stav výhodný či nevýhodný [53]. Na stanovení jejích hodnot pro daný stav hry neexistuje žádný obecný předpis a proto je při její definici potřeba vycházet ze zkušeností a pozorování.

V případě šachů se může jednat o vážený součet počtu figurek, jejich mobility, zabezpečení krále a kontroly nad středem šachovnice.

6.5.4 Algoritmus minimax

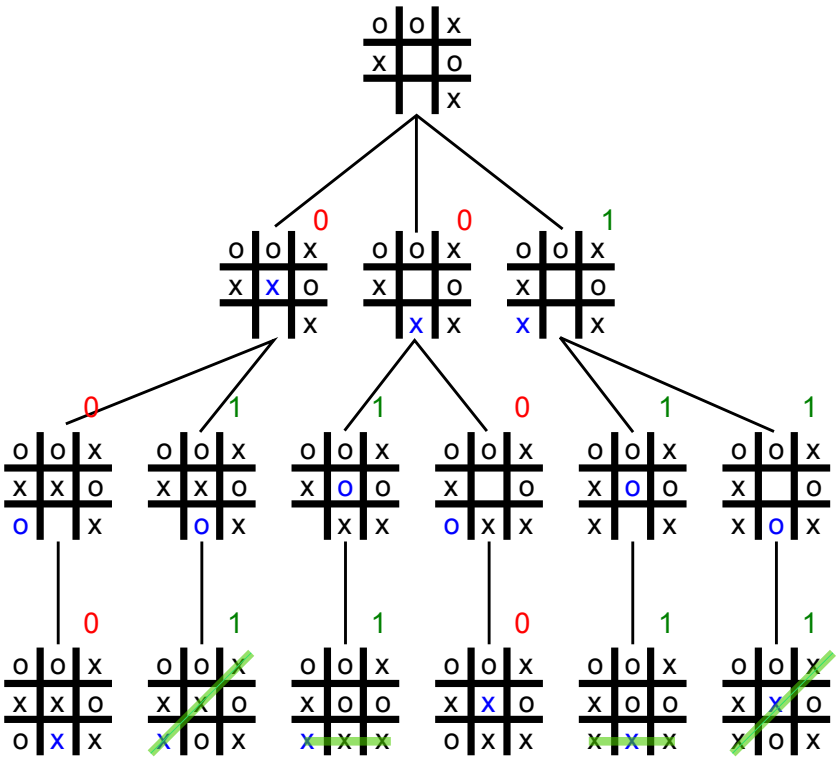
Algoritmus minimax je založen na prohledávání stavového prostoru do určité hloubky. Řídí se předpokladem, že každý hráč volí vždy nejlepší tah.

Název je odvozen od pravidelného střídání dvou hráčů - MAX hráče, který v každém kroku maximalizuje svůj zisk, a MIN hráče, který naopak svůj zisk minimalizuje [53].

Algoritmus prohledává herní strom do určité hloubky, zjistí hodnoty heuristické funkce pro všechny listy a z nich pak vybere strategicky nejvýhodnější akci.

Na obr. 6.10 je uveden příklad takového prohledávání - algoritmus nalezl šest koncových stavů, z toho čtyři vítězné. Nad aktuálním stavem je možno provést tři akce, z toho jedna vede vždy k vítěznému stavu.

Problém tohoto algoritmu spočívá v tom, že u větších stavových prostorů či u herních stromů s vysokým faktorem větvení běží neúměrně dlouho - například u deskové hry *go*. U real-time her je velikost tohoto prostoru natolik enormní, že činí minimax pro tyto účely nepoužitelným.



Obrázek 6.10: Příklad minimaxu na hře piškvorky

6.5.5 MonteCarlo prohledávání

Metoda *MonteCarlo* je založena na odhadování distribuce vlastností ve velkých stavových prostorech na základě pseudonáhodných vzorků. Jedná se o poměrně obecný pojem s velkým množstvím matematických aplikací. Na rozdíl od metodik *Las Vegas* je doba běhu deterministická a nezaručuje optimální hodnotu na výstupu. Je vhodná pro situace, kdy je čas běhu něčím limitovaný.

Obecně tato metoda funguje následovně: během hry algoritmus odehraje z daného stavu náhodnou simulovanou partii - *playout*. Jakmile dorazí do koncového stavu, spočítá se výsledek podle pravidel dané hry. Těchto náhodných simulací se provede několik a výsledek se poté zprůměruje. K simulaci obecně nepotřebuje heuristickou funkci na rozdíl od mini-

maxu, díky čemuž je vhodná pro prohledávání obecných stavových prostorů bez dodatečných informací.

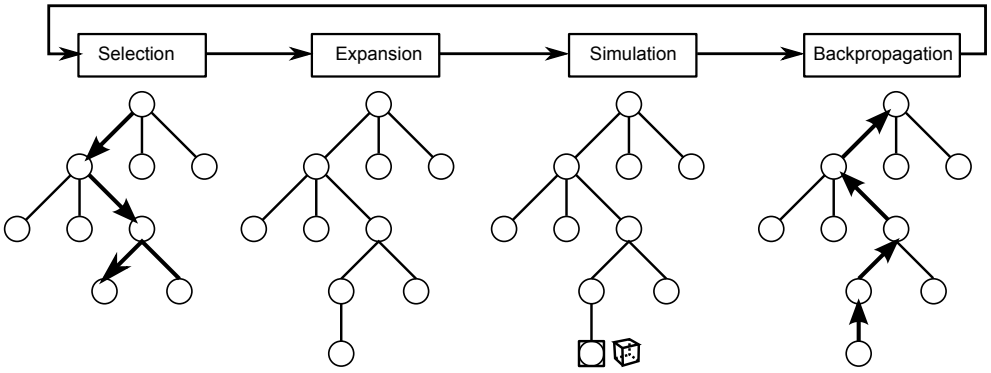
V posledních letech byla tato metoda úspěšně aplikována do různých her jako např. *Scrabble*, *Go* nebo *Poker* s překvapivě dobrými výsledky. Velký úspěch zaznamenala u hry *go*, což byla doposud jedna z mála tradičních her, u které selhávalo použití známých heuristických funkcí, neboť má vysoký větvící faktor a hluboký strom [54].

6.5.5.1 Monte-Carlo Tree Search

Monte-Carlo Tree Search (MCTS), je *best-first* algoritmus, vedený výsledky z předchozích simulací. Tradiční *Monte-carlo* metodiku rozšiřuje o možnost klást větší důraz na tahy, které vykazují lepší výsledky [55]. Tomuto fenoménu se říká *exploration vs exploitation*, kde *exploration* znamená procházení všech možných tahů a *exploitation* se zaměřuje jen na slibné tahy.

Algoritmus vytváří v průběhu simulace strom možných stavů v průběhu čtyř kroků:

- **Selekce** - strom je procházen od kořene na základě *exploration* formule
- **Expanze** - rozvinutí listu (vygenerování potomků pro každou odsimulovanou hru)
- **Simulace** - proběhne simulace od daného počátečního stavu
- **Zpětná propagace** - výsledek simulace se uloží do uzlů, které byly součástí selekce



Obrázek 6.11: Princip MonteCarlo tree-search

6.5.6 UCT algoritmus

Vyváženost mezi *exploration* a *exploitation* závisí na formuli, která se používá během fáze selekce. To mimo jiné naráží na problém mnohorukého bandity (*Multi-armed bandit problem*), kde je na každý vrchol MCTS stromu s n potomky pohlíženo jako na n-rukého banditu.

Jedním z nejznámějších algoritmů z rodiny MCTS řešící tento problém je algoritmus UCT (*Upper Confidence bound to Trees*), publikován v práci [56]. K prohledávání stromu využívá algoritmus formuli UCB1, která je definována následovně:

$$UCB1(i) = \frac{w_i}{n_i} + c\sqrt{\frac{\ln t}{n_i}} \tag{6.3}$$

- w_i je počet výher po i-tém tahu
- n_i je počet simulací po i-tém tahu
- c je *exploration* parametr
- t je počet návštěv daného uzlu

Při selekci se vybere ten potomek, pro který dává tato formule nejvyšší hodnotu. Pokud je odměna za každou odehranou hru v intervalu [0,1] (1 pro výhru a 0 pro prohru), je doporučená hodnota parametru c dle [54] a [56] rovna $\sqrt{2}$.

6.5.7 Implementace

Komponenta byla naprogramována tak, aby nebyla závislá na konkrétním vyhledávacím algoritmu a byla jednoduše rozšiřitelná. Při implementaci byly využity pokročilé šablonovací mechanismy jazyka C++, které umožnily obecně definovat stavový prostor a přechodové funkce. Architekturu komponenty znázorňuje diagram 6.12.

Základem každé hry je stav S a množina akcí A, která je z tohoto stavu proveditelná. Tyto dvě entity jsou zároveň parametry šablonovacích tříd.

Akce může být jakéhokoliv typu - v případě šachů by se jednalo o dvojici *figurka-pozice*, v případě piškvorek by to byla pouze souřadnice nově zaplněného políčka příslušným hráčem.

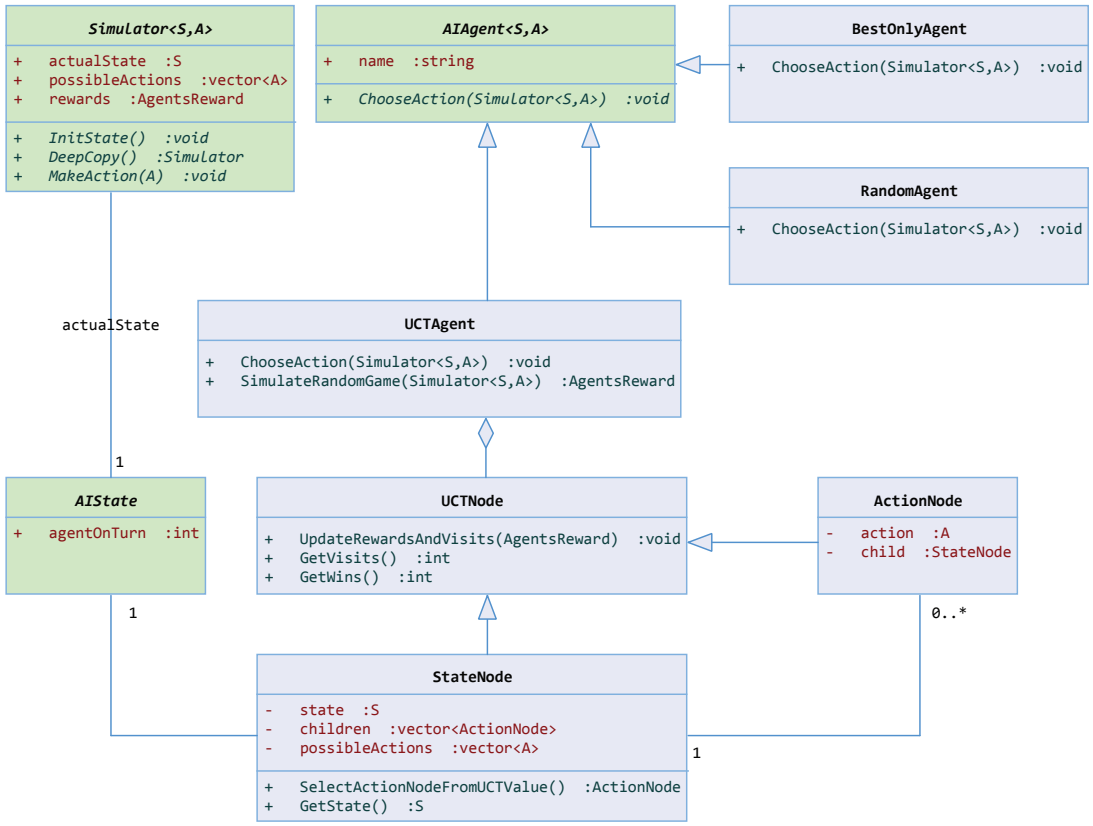
Stav hry může reprezentovat jakákoliv třída, která je potomkem `AIState`. Tato třída musí mít přetížené operátory rovnosti, aby bylo možno provést porovnání s ostatními stavy. Krom toho také obsahuje index aktuálního hráče, který je na tahu.

Třída `Simulator` reprezentuje množinu přechodových funkcí - obsahuje nějaký stav (proměnná `actualState`) a pomocí funkce `MakeAction` je proveden přechod do nového stavu v závislosti na zvolené akci. V terminologii stavového prostoru by to znamenalo aplikování operátoru.

Po provedení akce musí simulátor přegenerovat množinu přípustných akcí pro nový aktuální stav a také odměnu pro každého hráče po provedení akce.

Třída `AIAgent` reprezentuje hráče. Ta dostane na vstupu simulátor s aktuálním stavem a množinou přípustných akcí. Hráč pak musí nějakou akci vybrat.

Součástí komponenty jsou tři agenti - náhodný agent `RandomAgent`, který vybírá z množiny přípustných akcí vždy náhodně, `UCTAgent`, který implementuje UCT algoritmus, a `BestOnlyAgent`, který volí vždy akci s nejvyšší odměnou.



Obrázek 6.12: Návrh simulátoru

6.5.7.1 Aplikace algoritmu

Pro přidání nového agenta stačí vytvořit nového potomka třídy `AIAgent` a implementovat metodu `ChooseAction`.

Pro implementaci nějaké hry je potřeba vytvořit objekt reprezentující stav, objekt reprezentující akci a potomka třídy `Simulator`, který bude provádět přechody mezi stavy na základě zvolených akcí.

Aplikovatelnosti algoritmu UCT na realtime strategie se věnuje řada článků - např. [51], zabývající se managementem jednotek ve hře *Starcraft*. Dalším velmi zajímavým článkem je [57], kde jsou *MonteCarlo* metodiky aplikovány na generování náhodných plánů pro hru *Capture the Flag*.

V rámci této práce byl algoritmus vyzkoušen na prototypu hry, kterému se věnuje kapitola 7. Testování algoritmu je popsáno v kapitole 8.

Kapitola 7

Prototyp hry

Implementace hry probíhala současně s implementací enginu. Díky tomu mohlo v jednotlivých iteracích vývoje dojít k drobným optimalizacím, které si hra vyžádala. Hra byla pojmenována *Hydroq*.

7.1 Princip hry

Jedná se o dvourozměrnou hru spadající do žánru realtime strategií. Celá hra se odehrává na vodní ploše, na které se nachází ostrovy s těžebními věžemi.

Cílem hry je tyto těžební věže zabrat. K tomu slouží robotické jednotky, které každá věž po jejím zabrání začne vytvářet. Aby se tyto jednotky dostaly přes vodu, je nutné stavět přes vodu cesty, po kterých se pak mohou přemisťovat.

Jakmile je postavena cesta až k těžební věži, je možné ji zabrat. Pokud už tuto věž vlastní protihráč, je tak možné učinit až ve chvíli, kdy se kolem ní nachází více hráčových jednotek než jednotek protihráče. Poté přejdou všechny jednotky, která tato věž vytvořila, pod kontrolu toho, kdo ji zabral.

Na začátku hry si hráč volí jednu ze dvou frakcí - červenou nebo modrou. Poté má každý z nich přiřazenu jednu těžební věž, která začne ihned vytvářet jednotky.

Hra končí v okamžiku, kdy jeden z hráčů již nemá žádné jednotky, protože mu druhý hráč zabral jeho těžební věže.

Seznam uživatelských funkcí:

- **Build** - slouží pro stavění nových cest. Hráč označí pozici, na které se má stavět, a k tomuto místu se posléze přesune volná jednotka, která most postaví.
- **Destroy** - slouží pro demolici cest. Hráč označí pozici, která má být zničena a volná jednotka poté cestu přeruší.
- **Forbid** - označí pozici na mapě, kterou by jednotky neměly překročit. K překročení může dojít, pokud jednotka plní úkol a neexistuje jiná cesta.
- **Attract** - vytvoří na herní mapě *atraktor*, ke kterému se pak přesune určité procento jednotek. Pokud je atraktorů rozmístěno více, rozdělí se mezi daný počet jednotek.

7.2 Návrh

7.2.1 Návrh grafického rozhraní

Hra byla navržena pro cílovou platformu Android s orientací na šířku. Při grafickém návrhu byl kladen důraz na minimalismus a přesnost linií. Dominantní barvou je matně tmavě modrá a její doplňkové barvy, geometrie je převážně tvořena ostrými úhly, křivky se objevují sporadicky.

Z hlediska stylu se u jednotlivých objektů mísí *pixel art* a tradiční vektorová grafika. Jako font byl využit volně dostupný *Motion Control* od autora Harryho Wakamatsu [58].

Na obrázku 7.1 je zobrazen grafický návrh pro hlavní menu a herní scénu - všechny obrazovky menu mají ucelený styl, pro přechod do herní scény bylo navrženo statické okno s nápisem *Loading*.

Obrázek 7.1: Návrh grafického rozhraní

7.2.2 Návrh funkcí

Funkce byly navrženy tak, aby mohla být hra ovládána jedním prstem, případně *pinch* gestem pro přiblížení a oddálení herní mapy. Všechny podrobnosti o ovládání a ovládacích prvcích jsou popsány v příručce, která je součástí této práce.

7.2.2.1 Menu

V hlavním menu je možno pomocí dvou tlačítek vybrat buďto hru proti počítači (*Single Player*) nebo hru dvou lidských hráčů (*Multiplayer*).

V případě hry proti počítači si hráč vybere frakci, mapu a tlačítkem *Play* hru spustí. Počítač dostane automaticky druhou frakci.

V případě hry dvou hráčů se hráč buďto připojí k existujícímu hostiteli, který se mu zobrazí, nebo si zvolí mapu, frakci a tlačítkem *Host* pak vyčká, až se k němu někdo připojí.

7.2.2.2 Herní pole

Herní pole je rozděleno na čtyři části: levý a pravý panel, horní lišta a herní scéna. Levý a pravý panel je možno odsunout z obrazovky pryč.

V levé části je zobrazena mapa, počet jednotek a zabraných věží. Miniaturu mapy je možno použít pro rychlé přesouvání. V horní části se nachází lišta zobrazující aktuální vyváženost sil na mapě. V pravé části se vyskytuje panel s funkcemi pro stavění, bourání, zákaz přístupu a funkce *atraktor*.

Herní scénu je možno přibližovat a oddalovat.

7.2.3 Návrh architektury

Architektura byla navržena se snahou o zkombinování komponentového přístupu a tradičních objektových metodik, především vzoru *model-view-controller*.

Samotné oddělení modelu a view na základě různých skupin objektů typu *behavior* nestačí - pokud by model operoval nad reálnými herními objekty, byl by závislý na jejich transformačních objektech, které se mohou měnit se změnou grafických prvků, s různým rozložením herní plochy apod.

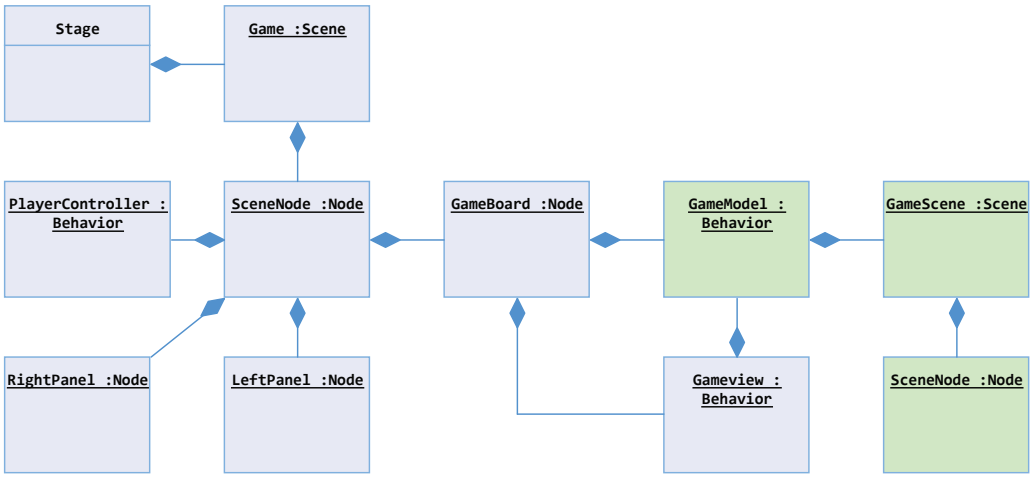
Jak bylo řečeno v části 2.2.3, základem herního stromu je komponenta `Stage`, která má pod sebou jednotlivé herní scény. Tyto scény drží odkaz na strom všech objektů obvykle pro jedno okno obrazovky. Zde vznikl nápad vytvořit scénu, která nebude součástí *komponenty* `Stage`, ale poběží odděleně. Tuto skutečnost zachycuje příklad na obrázku 7.2.

Scéna `Game` obsahuje všechny grafické objekty, které vidí uživatel. *Behavior* `GameModel`, které je součástí této scény, ale obsahuje svou vlastní scénu `GameScene`.

V této scéně jsou uchovávány modelové reprezentace všech objektů hry - mapa, jednotky, těžební věže apod. To umožňuje oddělenou správu transformací (které mohou být i v jiném souřadném systému), ale také posílání zpráv a reakcí na ně.

`GameView` pak bude mít na starosti reakci na zprávy od modelu a aktualizaci zobrazených objektů podle hodnot jejich atributů, které budou součástí modelu.

Jako controller je zde navržena třída `PlayerController`, která události odchytne pomocí systému zpráv (např. stisk tlačítka v pravém panelu) a předá ostatním *komponentám*.



Obrázek 7.2: Oddělené herní scény pro model a view

7.2.4 Multiplayer

Multiplayer byl navržen pro hru dvou hráčů, kde oba sdílejí část herního modelu. Jak bylo diskutováno v části 5.3.2, komunikace se odehrává na třech vrstvách. První a druhou vrstvu zajišťuje engine, třetí vrstvu bylo potřeba narvhnout pro potřeby hry.

7.2.5 Navázání spojení

Navázání spojení bude probíhat podle schematu z části 5.3.1. Na třetí vrstvě dojde k následujícím krokům:

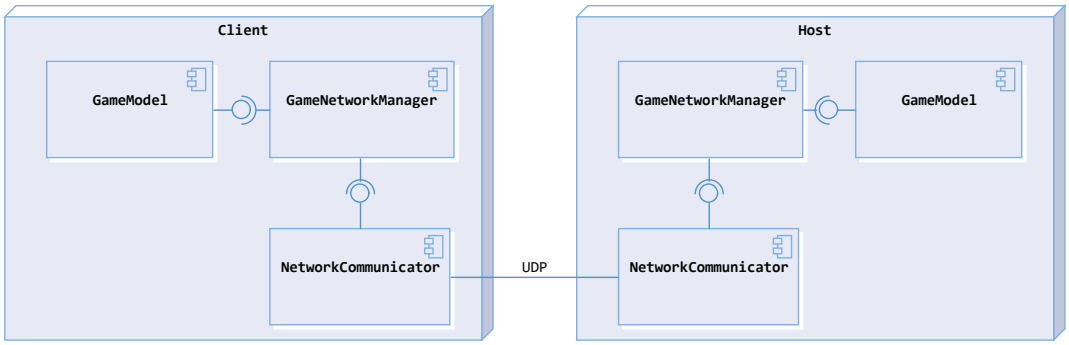
- hostitel si vybere mapu a frakci
- hostitel začne naslouchat na určitém portu
- klient pošle broadcast a najde dostupné hostitele
- hostitel odešle uvítací zprávu (zvolenou mapu a frakci)
- klient se připojí k hostiteli
- hostitel klienta akceptuje a spustí hru
- klient obdrží akceptační zprávu a spustí hru

Broadcast bude odeslán na tři subnety - `192.168.0.255`, `10.16.0.255` a pro účely ladění také na `127.0.0.1`. Obě stanice tedy musí být připojeny na nějaké lokální síti.

7.2.6 Komunikace

Díky komponentové architektuře bude možno synchronizaci herních modelů provést jednoduše pomocí mezivrstvy, která na základě přijatých zpráv upraví herní model.

Uživatelské akce, které budou v herní scéně posílány prostřednictvím zpráv, si tato vrstva odchytí a odešle druhé stanici. Pro interpolaci spojitých veličin bude využita komponenta enginu `Interpolator`.



Obrázek 7.3: Princip komunikace

`NetworkCommunicator` je *komponenta* enginu, která zajišťuje potvrzování přijatých zpráv a odesílání těch nepotvrzených. Zprávy odesílá v předem nakonfigurovaném časovém intervalu, přičemž typ zprávy nastavují ostatní komponenty, které tyto zprávy vkládají.

Komponenta před odesláním přibalí k této zprávě čas odeslání a pokud je potřeba nějakou z předchozích zpráv potvrdit, nastaví její identifikátor jako `AcceptId`.

Pokud zde není žádná zpráva, která by mohla být odeslána, vytvoří se obecná zpráva, která jen notifikuje druhou stanici a případně potvrdí přijatá data. Množství odeslaných zpráv a zatížení sítě během hry je diskutováno v části 8.5.1.

`GameNetworkManager` je komponenta třetí komunikační vrstvy a skládá se ze dvou tříd: `HydNetworkSender` a `HydNetworkReceiver`. Ty se starají o samotnou synchronizaci na úrovni herní logiky. Jakmile `HydNetworkReceiver` obdrží přijatou zprávu, identifikuje její typ, případně deserializuje datovou část do příslušného objektu. Pokud se jedná o *action message*, zavolá příslušnou metodu nad herním modelem (např. `SpawnWorker` pro vytvoření nové jednotky).

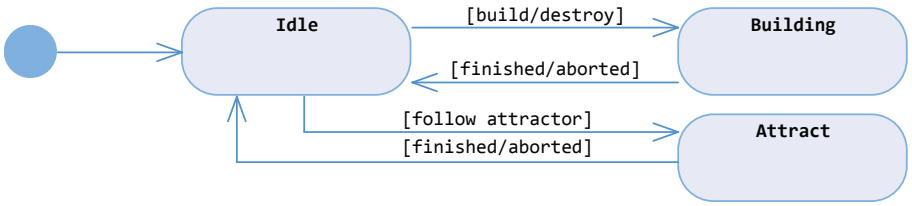
Protože je potřeba odlišit objekty, které vznikly v lokálním herním modelu, a jednotky, které vznikly na druhé stanici, bude využit parametr `secondaryId` objektu `Node` - ten bude obsahovat identifikátor objektu, který byl vytvořen na druhé stanici.

Chování pohybujících se jednotek je definováno jejich vlastními *behaviors*. Jednotky, které vznikly na druhé stanici, žádné funkční *behavior* obsahovat nebudou. Pouze bude docházet k průběžné aktualizaci jejich parametrů (pozice, rotace) na základě hodnot z *komponenty* `Interpolator`.

7.2.7 Umělá inteligence

Umělá inteligence bude pracovat ve třech vrstvách - management jednotek, plánování úkolů a strategické rozhodování.

Všechny jednotky budou mít svůj vlastní stavový automat, který je znázorněn na obrázku 7.4. Každý stav pak bude definovat množinu úkolů, které povedou k jeho naplnění - např. stav *Building*, který zahrnuje stavění mostu, vytvoří dva podúkoly - dostat se do cílové pozice a provést stavbu. V případě dokončení všech podúkolů nebo selhání prvního z nich dojde k přepnutí do stavu *Idle*.



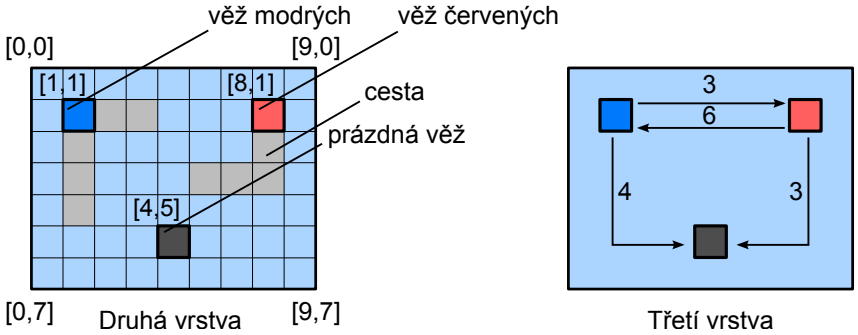
Obrázek 7.4: Stavy jednotek

Další entitou zde bude přidělovač úkolů. Třetí vrstva provede strategické rozhodnutí (stavba cesty směrem k určité těžební věži, zabrání prázdné věže apod.) Přidělovač pak tato rozhodnutí zanalyzuje a naplánuje úkoly, které přiřadí volným jednotkám v dosahu. Jakmile bude mít jednotka přiřazenou množinu úkolů, může se ještě na základě svých vlastních informací rozhodnout, zda tento úkol provede či zda jej odmítne.

7.2.7.1 Abstrahovaný herní model

Na třetí vrstvě bude pracovat strategická AI. Ta pomocí algoritmu UCT a abstrahovaného stavového prostoru hry vybere vhodnou akci pro následující časový interval. Po uplynutí tohoto intervalu dojde k přeplánování strategie.

Na této vrstvě nebude docházet k analýze terénu ani pozice jednotek. Algoritmus dostane na vstupu pozice jednotlivých těžebních věží, jejich vlastníka a dosažitelné vzdálenosti mezi nimi - tuto skutečnost zachycuje diagram na obr. 7.5



Obrázek 7.5: Herní model druhé a třetí vrstvy AI

Zde je například mezi modrou a prázdnou věží nejbližší políčko, kam se jednotka může dostat, na souřadnici $[1, 4]$. Manhattanská vzdálenost od prázdné věže je 4, protože je potřeba postavit 3 políčka na souřadnicích $[1, 5], [2, 5], [3, 5]$.

Třetí vrstva také bere v potaz, jaké bylo její předchozí rozhodnutí a za častou změnu strategie může být penalizována.

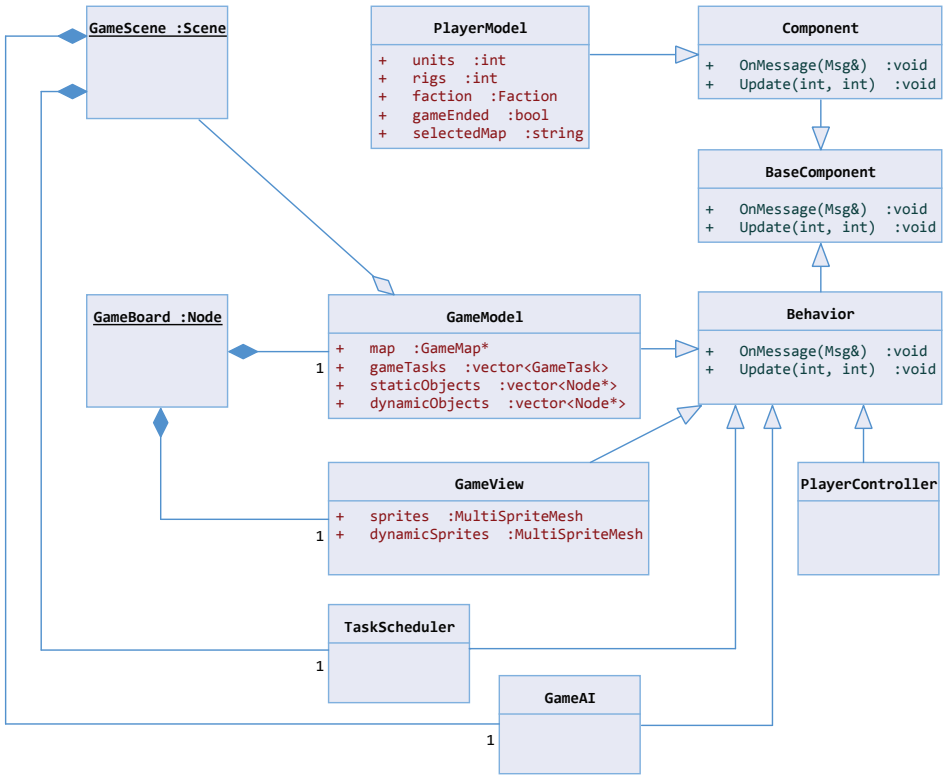
Jakmile dojde k výběru konkrétní akce, předá ji druhé vrstvě, která pak na základě kompletních informací o herním modelu rozhodne, jakým způsobem bude daná akce splněna. Cestu je možno stavit několika způsoby, stejně tak přesun jednotek k věži.

7.3 Realizace

Během realizace byl průběžně upravován engine na základě požadavků, které si hra vyžádala. Docházelo k refaktorování a rozšiřování jednotlivých *komponent* a zároveň bylo opraveno mnoho chyb a nedostatků, které byly objeveny až při řádném otestování na skutečném projektu. Některé z poznatků budou uvedeny v části 8.8.

7.3.1 Architektura

Následující diagram zobrazuje architekturu hry, která byla zaintegrována do komponentového modelu enginu.



Obrázek 7.6: Architektura hry

Samotná hra se skládá ze dvou scén: `GameBoard` je součástí grafu scény v komponentě `Stage` a je aktualizována společně s celou aplikací. Scéna `GameScene` oproti tomu existuje pouze ve třídě `GameModel` a je odstíněna od zbytku aplikace.

Jedinou globální *komponentou* je zde `PlayerModel`, který existuje ve všech herních scénách, jelikož jsou v něm obsaženy informace jak o situaci hráče, tak i o zvolené frakci a mapě, což nastavuje scéna `SinglePlayerMenu`, případně `MultiPlayerMenu`.

Uživatelské události má na starosti `PlayerController`, který pak danou akci předá hernímu modelu - třídě `GameModel`.

Zobrazování objektů a přepínání jednotlivých spritů má na starosti `GameView`. Nenachází se v něm žádná herní logika, pouze získává od modelu pomocí zpráv informace o nastálých změnách, které se pak vizuálně projeví.

Třída `GameModel` udržuje celý model hry - mapu, aktuální úkoly a herní objekty. Správu těchto úkolů má pak na starosti plánovač `TaskScheduler`, který figuruje jako *behavior* v kořenovém objektu scény `GameScene`. O strategická rozhodování se stará `GameAI`.

Všechny tyto třídy, `GameModel`, `GameView`, `GameAI` a `TaskScheduler` jsou potomky třídy `Behavior` a zachází se s nimi stejně jako s ostatními částmi komponentové architektury.

7.3.2 Herní scéna

Všechny scény jsou definovány v konfiguračním souboru *config.xml*, který je předán hernímu enginu při startu aplikace. V tomto souboru jsou definovány všechny statické objekty, jejich *behaviors* a transformace. Krom toho jsou zde také specifikovány všechny sprite sheety, cesty ke skriptům a globální nastavení.

```
<node name="toppanel" img="game/gm_toppanel.png" >
  <behavior type="TopPanel" />
<transform pos_x="50gr" pos_y="0gr" anchor_x="0.5" z_index="10" />

  <node name="blueicon" img="game/gm_blueicon.png">
    <transform pos_x="40gr" pos_y="0.5gr" anchor_x="0.5" anchor_y="0" />
  </node>
  <node name="redicon" img="game/gm_redicon.png">
    <transform pos_x="60gr" pos_y="0.5gr" anchor_x="0.5" anchor_y="0" />
  </node>
  <node name="scorebar_blue">
    <transform pos_x="43gr" pos_y="0.6gr" anchor_x="0" anchor_y="0"
        width="1gr" height="1.3gr" />
  <shape type="plane" color="0x23B9FF" width="10" height="10" />
  </node>
<node name="scorebar_red">
    <transform pos_x="57gr" pos_y="0.6gr" anchor_x="1" anchor_y="0"
        width="1gr" height="1.3gr" />
  <shape type="plane" color="0xFF2323" width="10" height="10" />
  </node>
</node>
```

Ukázka konfigurace herní scény (horní panel)

7.3.3 Měřítko

Všechny scény mají nastavenu referenční velikost, podle které se pak nastavují měřítka jednotlivých obrázků. Pozice jednotlivých objektů jsou navíc deklarovány relativně a díky tomu je hra zcela nezávislá na velikosti displeje cílového zařízení.

Podporovaný poměr stran je 16:9. V případě jiného poměru dojde k oříznutí okrajů displeje, jak je znázorněno v části 3.2.

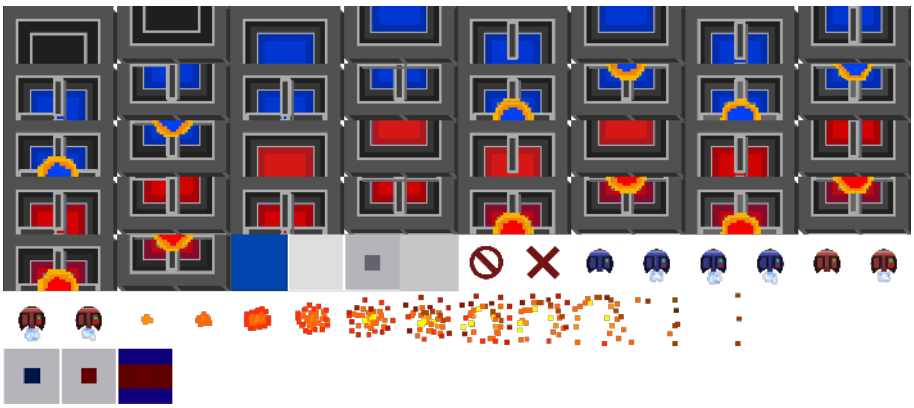
7.3.4 Mapy

Pro hru byly vytvořeny čtyři mapy - *Alpha, Beta, Gamma* a *Delta* s různými úrovněmi obtížnosti a různou velikostí. Specifikace objektů v mapě a jejich mapování na sprity je definována v konfiguračním souboru *mapconfig.xml*. Jednotlivé mapy jsou pak uloženy jako bitmapy a daná barva pak slouží jako identifikátor objektu, který se na dané pozici nachází.

Například mapa *Alpha* je definována v souboru *map_1.png*. Má velikost 45x40 pixelů, což znamená 45x40 polí. První pixel má černou barvu - `0x000000`. V souboru *mapconfig.xml* je pro tuto barvu nastaven sprite *water*.

7.3.5 Sprity

Objekty jako panely, miniatura mapy a ikony akcí se načítají jako obrázky. Oproti tomu vše na herní scéně (věže, jednotky, cesty) je načítáno z jednoho sprite sheetu, který je zobrazen na obr. 7.7. Indexy jednotlivých spritů pak uchovává konfigurační soubor *mapconfig.xml*. Porovnání rychlosti obrázků a spritů bude diskutováno v části 8.4.1.



Obrázek 7.7: Mapa spritů

7.3.6 Skripty

Některé komponenty byly napsány jako skripty v jazyce Lua - jednalo se především o *behaviors*, které reagovaly na stisk nějakého tlačítka a poté přepnuly herní scénu.

Všechny použité skripty jsou definovány v souboru *config.xml* a kompilují se při startu aplikace.

Kapitola 8

Testování

V této kapitole je shrnuto testování enginu, jeho komponent a prototypu hry. Jsou zde také zmíněny poznatky, kterými hra přispěla k průběžné optimalizaci enginu.

8.1 Testovací zařízení

K testování byla použita tři zařízení: notebook, tablet a chytrý telefon. Jejich konfiguraci zobrazuje tabulka 8.1. V následujících částech kapitoly budou již zmiňovány pouze jako *desktop*, *tablet* a *telefon*.

Tablet i telefon mají nainstalován operační systém Android, na tabletu je verze 4.1.1 a na telefonu verze 5.0.2, na desktopu je nainstalován Windows 8 64bit.

Zařízení	**CPU**	**GPU**	**RAM**
Notebook MSI GE70	Intel Core i7 2.4 GHz	nVidia GeForce GTX 960M	8GB DDR III
Samsung S6 edge	Exynos 8-core 4x2.1 + 2x1.5 GHz	Mali-T760 MP8	3GB LPDDR 4
Prestigio Multipad 7	ARM Cortex A9 1.5GHz	Mali-400	1GB DDR III

Tabulka 8.1: Testovací zařízení

8.2 Automatizované testy

Pro CogEngine bylo napsáno přes 200 automatických testů, seskupených do 14 skupin. Pro testování byl využit framework *catch* a k jejich spouštění byl vytvořen projekt *Tests*, nacházející se na přiloženém CD.

Seznam testů:

- **CoroutineTest** - testuje spouštění lambda výrazů
- **EngineTest** - testuje simulaci aktualizační smyčky enginu
- **FlagsTest** - testuje třídu `Flags` pro uchovávání stavů
- **GoalTest** - testuje komponenty z *goal-driven behavior*
- **LuaTest** - testuje komunikaci se skriptovacím jazykem *Lua*
- **MathTest** - testuje algoritmy vyhledávání v grafech
- **MeasureTest** - testuje komponentu na měření času
- **MonteCarloTest** - testuje algoritmus UCT
- **NetworkTest** - testuje komunikaci po síti a serializaci objektů
- **SettingsTest** - testuje XML konfiguraci
- **SQLTest** - testuje entitu pro přístup k databázi
- **StateMachineTest** - testuje komponentu pro stavový automat
- **StrIdTest** - testuje komponentu pro hashování řetězců
- **TransformTest** - testuje transformační entity

8.3 Výkonnostní testy

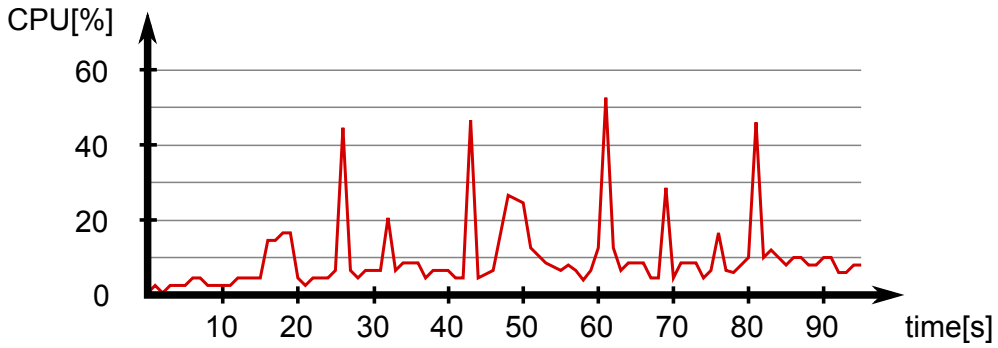
Součástí výkonnostních testů bylo měření procesorové zátěže, vytížení paměti a rychlost vykreslování.

8.3.1 Vytížení procesoru

Na obr. 8.1 je zobrazen graf vytížení procesoru na desktopu během odehrané hry. Průměrná spotřeba se pohybovala okolo hodnoty 7%.

V grafu je vidět několik extrémů - ty menší byly způsobeny situací, kdy větší množství jednotek použilo *pathfinding*. Největší zátěž (40-60%) byla naměřena v okamžiku spuštění simulace UCT algoritmu, kdy dochází k plánování strategie.

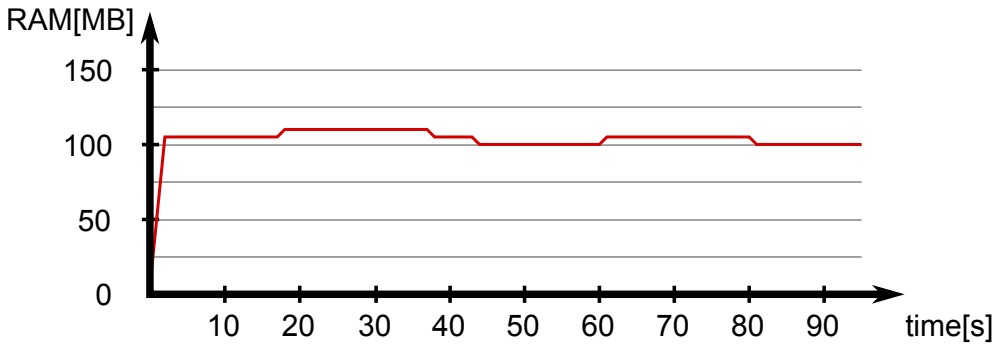
U ostatních zařízení dopadl test podobně; pouze na nejslabším tabletu byla pozorována občasná zamrznutí hry, kdy v rámci jedné aktualizace došlo jak k přeplánování strategie, tak k opakovanému použití *pathfinding*.



Obrázek 8.1: Test vytížení procesoru

8.3.2 Vytížení paměti

Spotřeba paměti byla téměř po celou dobu hraní konstantní - mezi 100 a 115 MB. Nutno podotknout, že drtivou většinu paměti zaujímaly načtené textury a obrázky, které byly vytvořeny pro zařízení s maximálním rozlišením 2560x1440 pixelů. Z tohoto testu vzešel poznatek pro budoucí rozšíření enginu, kde by byla možnost, podobně jako u standardní Androidí aplikace, definovat obrázky s různými velikostmi, které by se poté načetly dle rozlišení zařízení.



Obrázek 8.2: Test vytížení paměti

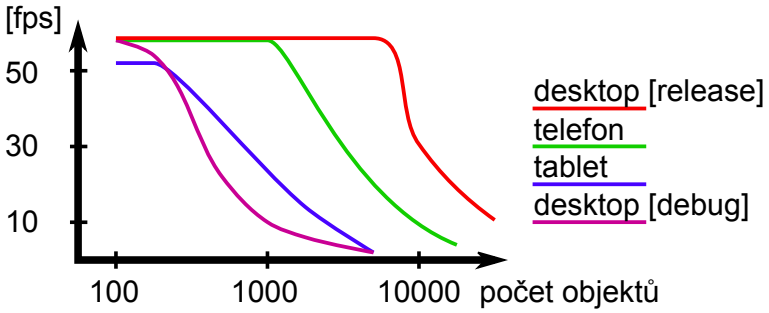
8.4 Rychlost vykreslování

Pro test výkonu byl vytvořen projekt *Sprites*, který je součástí sady příkladů na přiloženém CD. Během testu byl vytvořen určitý počet objektů a sledována průměrná obnovovací frekvence (počet snímků za sekundu). Tyto objekty se pohybovaly po ploše pomocí náhodné změny rychlostního vektoru. Objekty byly načteny jako sprite sheet.

Na desktopu byla porovnána verze pro *debug* i *release*, neboť v jazyce C++ se jejich výkonová náročnost dramaticky liší.

Počet objektů	Tablet	Telefon	Desktop debug	Desktop release
100	52	58	58	58
200	52	58	54	58
500	38	58	22	58
1000	22	57	10	58
2000	15	45	5	57
5000	4	20	1	57
7000	-	9	-	44
10000	-	6	-	30

Tabulka 8.2: Test rychlosti vykreslování



Obrázek 8.3: Test rychlosti vykreslování - graf

OpenFrameworks se snaží vždy dodržet nastavenou vykreslovací frekvenci, která má v tomto případě hodnotu 58, bez ohledu na zátěž. Jakmile však čas potřebný na vykreslení přesáhne délku vykreslovací periody, začne celková frekvence klesat, což je zřetelně vidět na grafu 8.3.

Nejlepší výsledky byly naměřeny na desktopu v *release* verzi. *Debug* verze dopadla nejhůř, což je způsobeno povahou jazyka C++ a standardních knihoven, které jsou díky složitým kontrolám (např. *Debug Iterator Support* v STL) v tomto módu mnohem pomalejší.

Přesto je možné konstatovat, že vykreslování spritů zvládá engine velice slušně. I starší tablet dokázal plynule zobrazit stovky pohybujících se objektů.

8.4.1 Spritesheet vs bitmapy

Pro porovnání byl ještě proveden stejný test, kdy se místo sprite sheetu použilo přímé vykreslování bitmap jedna za druhou. Test byl proveden pouze u desktopového zařízení v *release* verzi.

Dle výsledků uvedených v tabulce 8.3 nastává zlom v momentě, kdy dosahuje počet objektů přibližně tisícovky - druhá varianta přestane stíhat a frekvence rychle klesá. Oproti tomu první varianta, využívající sprite sheet, zvládne vykreslit objektů při stejné frekvenci pětkrát tolik.

Počet objektů	Sprite sheet	Obrázky
100	58	58
200	58	58
500	58	55
1000	58	43
2000	57	25
5000	57	13
7000	44	9
10000	30	6
12000	26	4

Tabulka 8.3: Test obou typů vykreslování

8.5 Přenos dat po síti

K testu přenosu dat byl využit projekt *Network3*, který je součástí sady příkladů na přiloženém CD. Tento příklad je potřeba spustit na dvou stanicích - vysílací a přijímací. Test probíhal na desktopu a telefonu, obojí připojené přes wi-fi.

Vysílací stanice vytvořila herní scénu s daným počtem pohybujících se objektů. Tyto objekty byly poté přeposílány na přijímací stanici, která je následně zobrazila. Během testu byla průběžně měněna frekvence odesílání.

Na základě pozorování bylo vyhodnoceno, že k plynulému zobrazení objektů na přijímací stanici bez častého zamrzávání dochází při frekvenci v intervalu [5,20].

Následující tabulka shrnuje vytížení sítě pro daný počet objektů a frekvenci odesílání. Jelikož aktualizační datagramy nepotřebují být potvrzovány, odesílala přijímací stanice pouze pravidelné notifikace. Protože minimální délka zprávy činí 12 B, což je i s velikostí UDP hlavičky celkem 20 B na datagram, činil objem notifikačních zpráv pouhých 0.78 Kb/s.

Počet objektů	Přenos dat pro frekvenci 5	Přenos dat pro frekvenci 20
10	10 kbps	40 kbps
30	28 kbps	112 kbps
50	48 kbps	192 kbps
100	94 kbps	376 kbps
200	188 kbps	752 kbps
500	468 kbps	1872 kbps
1000	938 kbps	3752 kbps

Tabulka 8.4: Testování síťové zátěže

U nejnižší přijatelné frekvence 5 zpráv za sekundu dosahovala rychlost přenosu dat i u tisícovky objektů přibližně 1000 kbps, což je mnohem méně než např. doporučená rychlost pro přenos videa v HD rozlišení.

8.5.1 Přenos dat ve hře

Počet aktualizačních zpráv pro hru *Hydroq* byl nastaven na hodnotu 10. Jelikož se u jednotek stejně jako v předchozím testu synchronizuje pouze pozice a rotace, je možné očekávat stejný objem dat pro daný počet jednotek jako v předchozím testu. Počet jednotek je závislý na celkovém počtu těžebních věží, nicméně během testování nikdy nepřekročil číslo 150.

Během několika testů byly naměřeny následující hodnoty:

- průměrná hra trvá 5 min
- bylo odesláno v průměru 6537 paketů
- průměrná velikost paketu byla 256 B
- maximální naměřená velikost paketu byla 4838 B
- celkové množství odeslaných dat činilo v průměru 1634 KB
- průměrná zátěž sítě byla 5.45 KB/sec (43.6 kbps)

Pro hru dvou hráčů je možno považovat průměrnou rychlost přenosu dat 5.45 KB/s vzhledem k průměrné rychlosti síťového připojení jako naprosto dostačující.

8.6 Testování UCT algoritmu

UCT algoritmus byl implementován jako součást strategické AI ve hře *Hydroq*. Protože nebyl k dispozici skript, proti kterému by mohla být změřena účinnost UCT agenta, byla AI otestována jako celek v rámci testování s uživateli, které bude popsáno v části 8.7.

Testování algoritmu jako takového je součástí automatických testů, pro které byl vytvořen jednoduchý model hry *piškvorky*.

8.6.1 Piškvorky

Piškvorky (anglicky *Tic-tac-toe*) je možno považovat za strategickou hru, jejíž pravidla netřeba představovat. Stavový prostor nemá tak vysoký faktor větvení jako hra *go*, proto je vhodný i pro systematické prohledávání algoritmy jako *minimax* či *negamax*.

Celý model se nachází v souboru `MonteCarloTestAssets.h`. Hrací deska má velikost 8×8 políček a k vítězství je potřeba vytvořit přímou nebo diagonální řadu pěti označených polí.

Simulátor `TicTacToeSimulator`, který obsahuje generátor možných akcí a přechodové funkce, odměňuje herního agenta jednoduše podle délky nejdelší řady.

Otestovány byly dvě varianty - UCT agent proti náhodnému agentovi a UCT agent proti agentovi, který volí akci s nejvyšší odměnou.

Následující tabulka ukazuje výsledky partií UCT agenta proti náhodnému agentovi pro různý počet simulací tohoto algoritmu, počet vítězství obou agentů a průměrný počet tahů. Testů bylo provedeno celkem deset a počet tahů celé partie byl zprůměrován.

UCT simulace	**Score**	**Prům. počet tahů**
1	4:6	25
10	4:6	25
100	5:5	17
200	7:3	14
500	8:2	9
1000	10:0	7
2000	10:0	5
5000	10:0	5

Tabulka 8.5: UCT agent proti náhodnému agentovi

Až u počtu simulací 100 a více začal UCT agent náhodného agenta porážet. Protože hrací deska obsahuje 64 políček, je potřeba prohledat minimálně stejné množství stavů, než bude mít algoritmus k dispozici relevantní informace.

V případě, kdy proti sobě hráli UCT agent a `BestOnlyAgent`, který volí vždy akci s nejvyšší odměnou, vítězil `BestOnlyAgent` až do počtu simulací 5000, jelikož UCT agent nebyl schopný předpovědět blížící se porážku.

Tento problém byl vyřešen zmenšením stavového prostoru hry pomocí filtru - do třídy `Simulator` byla přidána abstraktní metoda `ApplyFilter`, která vezme jako parametr objekt typu `ActionFilter` a ten může po přechodu do následujícího stavu upravit množinu povolených akcí.

Tento filtr byl naimplementován pro simulátor piškvorek tak, aby bylo možno hrát vždy nejdále jedno políčko od již označeného políčka. Tento filtr může být předán UCT agentovi, který jej bude používat při vlastních simulacích.

Následující tabulka ukazuje výsledky partií UCT agenta proti agentovi `BestOnlyAgent` s filtrem a bez filtru. Díky filtraci vykazoval UCT agent mnohem lepší výsledky, neboť byly v každém kroku vypuštěny strategicky nezajímavé akce.

UCT simulace	**Score s filtrem**	**Score bez filtru**
1	0:10	0:10
10	0:10	0:10
100	0:10	0:10
200	0:10	2:8
500	0:10	4:6
1000	0:10	10:0
2000	1:9	10:0
5000	3:7	10:0

Tabulka 8.6: UCT agent proti best-only agentovi

8.7 Testování s uživateli

Testování s uživateli proběhlo ve dvou fázích: v první fázi byl vytvořen testovací scénář, podle kterého měli uživatelé postupovat, aniž by znali pravidla hry. Ve druhé fázi již byli s pravidly obeznámeni a měli odehrát několik her proti počítači.

Testovanou skupinou lidí byli příležitostní hráči mobilních her ve věku 22-28 let.

8.7.1 První fáze

Testovací scénář:

1. vyberte si frakci a mapu
2. spusťte hru
3. prozkoumejte herní pole
4. postavte cestu k těžební věži
5. přesuňte jednotky k této věži

Následující tabulka ukazuje postup jednotlivých testerů testovacím scénářem.

Krok	**Tester 1**	**Tester 2**	**Tester 3**
1	bez problému	bez problému	bez problému
2	bez problému	bez problému	bez problému
3	nevěděl, že je možné mapu oddálit	bez problému	bez problému
4	bez problému	nevěděl, jak stavět	chvíli tápal
5	nevěděl, jak funguje atraktor	nevěděl, jak funguje atraktor	po chvíli úkol splnil

Tabulka 8.7: Průchod testerů scénářem

Většina úkolů byla splněna bez problémů. Největší problém představoval bod 5, kdy dva testeři nepochopili funkci atraktoru. Ani jeden z nich nepřišel na to, že atraktor je možno vytvořit v různých velikostech.

Ve většině RTS her dochází k ovládání jednotek přímo a atraktory nejsou příliš známé. Přesto existují hry, kde je tento princip hojně využíván (např. hra *Globulation*). Řešením by mohla být interaktivní nápověda, která by prostřednictvím poloprůhledné ikony hráči při prvním hraní napověděla, jak hru ovládat.

8.7.2 Druhá fáze

Během druhé fáze každý tester odehrál celkem 20 her, z toho po pěti na každé mapě. Mapa *Alfa* obsahovala dvě těžební věže, *Beta* byla nejkomplikovanější z hlediska velkého

počtu věží, *Gamma* nevyžadovala žádné stavění ale rychlý přesun jednotek, *Delta* obsahovala velké množství cest, po kterých se mohly jednotky pohybovat.

Mapa	**Tester 1 score**	**Tester 2 score**	**Tester 3 score**
Alfa	2:3	3:2	3:2
Beta	1:4	0:5	2:3
Gamma	5:0	3:2	3:2
Delta	3:2	3:2	3:2

Tabulka 8.8: Odehrané hry

Pouze na mapě *Gamma*, která nevyžaduje stavění žádných cest, si jeden z testerů našel strategii, která vedla vždy k porážce počítačového hráče. AI totiž nebyla schopna zabrat větší množství věží najednou a při přesunu jednotek odkryla svou vlastní těžební věž, ke které se hráč mohl dostat obloukem. Celkově nepatrně převažoval počet výher lidských hráčů.

8.7.3 Zhodnocení

Všichni testeři hodnotili hru velmi kladně, během testování byly také odhaleny některé nedostatky počítačového protihráče:

1. AI staví některé cesty zbytečně, když se poblíž již nějaká nachází
2. občas se stane, že si AI rozmyslí svou strategii
3. AI často nechává některé své věže opuštěné

Protože třetí vrstva AI používá abstrahovaný herní model, nedokáže některé akce předvídat. S tím se musí vyrovnat druhá vrstva, která obecně akce transformuje na sadu úkolů pro plánovač.

Poznatek č. 1 by mohla vyřešit komplexnější analýza herní mapy či průběžná optimalizace při stavění cest.

Poznatek č. 2 byl vyřešen úpravou druhé vrstvy, která od vrstvy třetí přijímá strategická rozhodnutí - pokud vyhodnotí změnu strategie za příliš rizikovou, ponechá si tu aktuální až do příští iterace.

Poznatek č. 3 byl vyřešen úpravou funkcionality atraktorů - AI nikdy nepošle všechny jednotky k cizí věži, ale vždy si ponechá část u té své.

Protože se hra svým principem postupného zabírání věží odlišuje od známých strategických her, není na ni možné aplikovat žádné obecně známé postupy. Pro nalezení optimální strategie, kterou by pak bylo možno naprogramovat do AI, by vyžadovala velké množství testování. Zajímavým přínosem by zde mohla být implementace strojového učení, díky kterému by hra svou strategii volila na základě dat sesbíraných z minulých her.

8.8 Poznatky z testování enginu

Engine byl otestován především díky hře *Hydroq*, která v něm byla naimplementována. Během této fáze došlo k několika optimalizacím komponentové architektury - především se zjednodušilo posílání zpráv přidáním patřičných metod, které zprávu poskládají samy a jako parametr dostanou jen podmnožinu parametrů.

Mezi nevýhodami komponentové architektury, které byly diskutovány v části 1.3.2.1, nejvíce převažovala dynamičnost celé architektury, která znesnadňovala ladění. Pro tento účel byla vytvořena metoda `WriteInfo` do všech tříd, která vypíše do logu kompletní strukturu celé herní scény.

Dalším problémem byly neznámé typy atributů - pokud je atribut například definován jako *string* a v následujících iteracích vývoje je předefinován na *integer*, musí být takto předefinován ve všech zdrojových souborech, které s tímto atributem pracují. Pokud by se tak nestalo, program by se přesto zkompiloval a docházelo by k problémům. Pro snazší ladění byly do všech metod, které vrací objekt neznámého typu, přidány aserce na kontrolu tohoto typu, které případné chyby zapisovaly do logu i s číslem řádky, kde k chybě došlo.

Dále byly optimalizovány některé vyhledávací metody - např. třída `Scene`, která obsahuje metody `FindNodeById` a `FindNodeByTag`, používá několik hashovacích tabulek, do kterých ukládá všechny uzly, které se ve scéně nachází.

8.8.1 Velikost balíčku

Součástí zkompilované knihovny enginu je i OpenFrameworks a všechny použité knihovny třetích stran. Knihovna enginu zkompilovaná pod MSVC má velikost 3.2 MB. Velikost aplikace zkompilované pro platformu Android s prázdnou scénou činí 3.6 MB. Při srovnání s ostatními enginy, diskutovanými v části 1.4, z toho vychází CogEngine poměrně slušně.

Počet řádků kódu v enginu dosahuje 16 500. Prototyp hry jich obsahuje 4 800.

Závěr

Zhodnocení práce

Hlavním cílem této práce bylo vytvořit engine použitelný pro vývoj dvourozměrných her a otestovat jej na prototypu vlastní hry, určené pro platformu Android. Engine měl navíc disponovat komponentovou architekturou, která by umožnila snadné přidávání nových funkcionalit a dynamickou správu herních objektů.

Dalším cílem bylo implementovat základní sadu komponent běžně se vyskytujících v herních enginech jako vykreslování *spritů*, synchronizace dat po síti, hledání cest a algoritmy umělé inteligence.

Všechny požadavky byly splněny a výsledkem práce je funkční herní engine nabízející rozličnou sadu nástrojů, dále pak hratelný prototyp strategické hry s multiplayerem a umělou inteligencí, s přívětivým designem a minimalistickým uživatelským rozhraním.

Součástí enginu je rovněž sada doprovodných příkladů, které demonstrují jeho možnosti. Jednotlivé komponenty včetně herní scény mohou být konfigurovány v jazyce XML a kromě nativního vývoje v C++ může být pro definici herní logiky použit skriptovací jazyk Lua.

Engine je napsán v jazyce C++ jako doplněk OpenFrameworks frameworku a je zkompilovatelný pro platformy Windows a Android. Na rozdíl od ostatních enginů dosahuje poměrně kompaktní velikosti 3.2 MB a díky MIT licenci může být použit pro vývoj komerčních produktů.

Kromě her je také použitelný pro vývoj multimediálních aplikací, neboť umožňuje přehrávání zvuků a animací, přístup k SQLite databázi a snadnou integraci dalších komponent.

Otestován je více než dvěma sty automatickými testy, obsahuje pečlivě dokumentovaný zdrojový kód a je připraven k distribuci jako oficiální doplněk OpenFrameworks [1]. Jeho aktuální verze je k dispozici na portálu GitHub [2].

Realizaci každé části enginu předcházel stručný teoretický úvod do dané problematiky a práce tak může posloužit i jako zdroj informací pro zájemce o vývoj vlastního řešení z dané oblasti.

[1] Pro distribuci je potřeba pouze dodržet jmenné konvence, vložit projekt na GitHub a podat žádost k zařazení do oficiální kolekce addonů. Žádost byla podána dne 9.5.2016.

[2] `https://github.com/dormantor/ofxCogEngine`

Náměty k rozšíření

Existuje nespočet možností, jak rozšířit herní engine. Některá rozšíření jdou přímo na ruku konkrétním herním žánrům, některá však mohou být využita obecně v jakékoliv hře. Zde jsou popsána tři nejzajímavější, které CogEngine postrádá:

- **Fyzikální engine** - simulátorů fyziky je k dispozici mnoho, některé z nich je možno najít i mezi doplňky OpenFrameworks frameworku. Integrace takového simulátoru do enginu by umožnila vývoj složitějších her, stavěných na mechanických principech.
- **Grafický editor scén** - protože jsou nyní scény definovatelné v XML souborech, je myšlenka jejich vizuální editace přímo na místě. Grafický editor by významně urychlil prototypování a co víc, možnost editace herní scény přímo za běhu aplikace podobně jako v enginu Unity by mu dala zcela nový rozměr.
- **Fuzzy logika** - herní agenti, jejichž rozhodování řídí jednoduché podmínky s pevnými hodnotami (např. *zaútoč, pokud je základna blíž než 10 jednotek mapy*), se chovají nerealisticky. Fuzzy logika, která umožňuje díky stupni příslušnosti hladký přechod mezi jednotlivými stavy, by do enginu vnesla nové možnosti rozhodování.

Literatura

[1] Stephen Daultrey: 1972: First Commercially Successful Arcade Computer Game. [cit. 2016-05-01]. Dostupné z: http://www.guinnessworldrecords.com/news/60at60/2015/8/1972-first-commercially-successful-arcade-computer-game-392971

[2] Statista: Number of available applications in the Google Play Store from December 2009 to November 2015. [cit. 2016-05-01]. Dostupné z: http://www.statista.com/statistics/266210/number-of-available-applications-in-the-google-play-store/

[3] norman: Doom to Dunia: A Visual History of 3D Game Engines. [cit. 2016-05-01]. Dostupné z: http://www.maximumpc.com/doom-to-dunia-a-visual-history-of-3d-game-engines/

[4] Gregory, J.: *Game Engine Architecture*. CRC Press, druhé vydání, 2015.

[5] Jordan Mechner: Prince of Persia. [cit. 2016-05-05]. Dostupné z: https://github.com/jmechner/Prince-of-Persia-Apple-II

[6] Microsoft: WPF Architecture. [cit. 2016-05-07]. Dostupné z: https://msdn.microsoft.com/en-us/library/ms750441%28v=vs.100%29.aspx

[7] Wenbin Dai: *A Component-Based Design Pattern for Improving Reusability of Automation Programs*. 2013.

[8] Harald Gall, Mehdi Jazayeri, René Klösch: *The Architectural Style of Component Programming*. 2013.

[9] Arlow., J.: *UML 2 and the unified process : practical object-oriented analysis and design*. Upper Saddle River, 2005.

[10] Porter, N.: *Component-based game object system*. 2012.

[11] Marcin Chady: Theory and Practice of Game Object Component Architecture. [cit. 2016-05-03]. Dostupné z: http://twvideo01.ubm-us.net/o1/vault/gdccanada09/slides/marcinchadyGDCCanada.ppt

[12] Julian Gold: *Object-oriented Game Development*. 2004, ISBN 978-0321176608.

[13] OpenFrameworks. [cit. 2016-05-03]. Dostupné z: http://openframeworks.cc/

[14] Processing. [cit. 2016-05-04]. Dostupné z: http://www.processing.org/

[15] OpenFrameworks addons. [cit. 2016-05-08]. Dostupné z: http://ofxaddons.com/categories

[16] Ridiculous fishing. [cit. 2016-05-02]. Dostupné z: http://www.ridiculousfishing.com/

[17] Catch library. [cit. 2016-05-04]. Dostupné z: https://github.com/philsquared/Catch/

[18] Lua. [cit. 2016-05-08]. Dostupné z: https://www.lua.org

[19] LuaBridge. [cit. 2016-05-08]. Dostupné z: https://github.com/vinniefalco/LuaBridge

[20] SQLite. [cit. 2016-05-08]. Dostupné z: https://www.sqlite.org/

[21] TinyXML. [cit. 2016-05-08]. Dostupné z: https://sourceforge.net/projects/tinyxml/

[22] Android OS. [cit. 2016-05-08]. Dostupné z: http://developer.android.com/

[23] Fletcher Dunn, I. P.: *3D Math Primer for Graphics and Game Development*. 2002.

[24] David Luebke, Greg Humphreys: How GPUs Work. [cit. 2016-05-04]. Dostupné z: http://www.cs.virginia.edu/~gfx/pubs/Luebke_2007_HGW/luebke2007.pdf

[25] JavaScript. [cit. 2016-05-04]. Dostupné z: https://www.javascript.com/

[26] Man, K.-H.: *A No-Frills Introduction to Lua 5.1 VM Instructions*. 2012.

[27] Duktape. [cit. 2016-05-08]. Dostupné z: http://duktape.org/

[28] SDBM Hash. [cit. 2016-05-08]. Dostupné z: http://www.cse.yorku.ca/~oz/hash.html

[29] RFC 1122 - Requirements for Internet Hosts - Communication Layers. [cit. 2016-05-07]. Dostupné z: https://tools.ietf.org/html/rfc1122

[30] RFC 1123 - Requirements for Internet Hosts - Application and Support. [cit. 2016-05-03]. Dostupné z: https://tools.ietf.org/html/rfc1123

[31] RFC 791 - Internet Protocol. [cit. 2016-05-04]. Dostupné z: https://tools.ietf.org/html/rfc791

[32] RFC 793 - Transmission Control Protocol. [cit. 2016-05-06]. Dostupné z: https://tools.ietf.org/html/rfc793

[33] RFC 768 - User Datagram Protocol. [cit. 2016-05-04]. Dostupné z: https://tools.ietf.org/html/rfc768

[34] Memory performance speed latency. [cit. 2016-05-04]. Dostupné z: http://www.crucial.com/usa/en/memory-performance-speed-latency

[35] Introduction to Ethernet Latency. [cit. 2016-05-06]. Dostupné z: http://www.qlogic.com/Resources/Documents/TechnologyBriefs/Adapters/Tech_Brief_Introduction_to_Ethernet_Latency.pdf

[36] Source Multiplayer Networking. [cit. 2016-05-03]. Dostupné z: https://developer.valvesoftware.com/wiki/Source_Multiplayer_Networking

[37] Michael Powers: Mobile Multiplayer Gaming. [cit. 2016-05-06]. Dostupné z: http://www.oracle.com/technetwork/java/index-140739.html

[38] Alf Inge Wang: Experiences from Implementing a Mobile Multiplayer Real-Time Game for Wireless Networks with High Latency. [cit. 2016-05-07]. Dostupné z: http://www.hindawi.com/journals/ijcgt/2009/530367/

[39] What every programmer needs to know about networking. [cit. 2016-05-04]. Dostupné z: http://gafferongames.com/networking-for-game-programmers/what-every-programmer-needs-to-know-about-game-networking/

[40] Ashley Reed: Gaming's worst AI companions. [cit. 2016-05-04]. Dostupné z: http://www.gamesradar.com/gamings-worst-ai-companions/

[41] Jack Copeland: What is Artificial Intelligence? [cit. 2016-05-04]. Dostupné z: http://www.alanturing.net/turing_archive/pages/referencearticles/whatisai.html

[42] Buckland, M.: *Programming game AI by example*. 2010, ISBN 978-1556220784.

[43] Reynolds, C. W.: *Flocks, herds and schools: A distributed behavioral model*. 1987, ISBN 0-89791-227-6.

[44] Reynolds, C. W.: Steering Behaviors For Autonomous Characters. 1999, [cit. 2016-05-04]. Dostupné z: http://www.red3d.com/cwr/steer/gdc99/

[45] Finite state machine in Technology. [cit. 2016-05-05]. Dostupné z: http://www.dictionary.com/browse/finite-state-machine

[46] Ferdinand Wagner, T. W. P. W., Ruedi Schmuki: *Modeling software with Finite State Machine: A practical approach*. 2006.

[47] Malluk, W.: *An Object-Oriented Approach for Hierarchical State Machines*. 2006, ISBN 85-7669-098-5.

[48] Erich Gamma, R. J. J. V., Richard Helm: *Design Patterns: Elements of Reusable Object-Oriented Software*. 2000, ISBN 978-0201633610.

[49] Millington, I.: *Artificial Intelligence for Games*. 2009, ISBN 978-0123747310.

[50] Amirhosein Shantia, M. W., Eric Begue: Connectionist Reinforcement Learning for Intelligent Unit Micro Management in StarCraft. [cit. 2016-05-08]. Dostupné z: http://www.ai.rug.nl/~mwiering/GROUP/ARTICLES/StarCraft.pdf

[51] Wang Zhe, R. T., Kien Quang Nguyen: Using Monte-Carlo Planning for Micro-Management in Starcraft. [cit. 2016-05-08]. Dostupné z: http://www.ice.ci.ritsumei.ac.jp/~ruck/PAP/gameonasia12-wang.pdf

[52] Gergely Kovásznai, G. K.: Artificial Intelligence and its Teaching. [cit. 2016-05-04]. Dostupné z: http://aries.ektf.hu/~gkusper/ArtificialIntelligence_LectureNotes.v.1.0.4.pdf

[53] Allis, V.: *Searching for Solutions in Games and Artificial Intelligence*. 1994, ISBN 90-9007488-0.

[54] Cameron Browne, E. P.: A Survey of Monte Carlo Tree Search Methods. [cit. 2016-05-07]. Dostupné z: http://www.cameronius.com/cv/mcts-survey-master.pdf

[55] Guillaume Chaslot, I. S., Sander Bakkes: Monte-Carlo Tree Search: A New Framework for Game AI. [cit. 2016-05-05]. Dostupné z: https://www.aaai.org/Papers/AIIDE/2008/AIIDE08-036.pdf

[56] Levente Kocsis, C. S.: Bandit based Monte-Carlo Planning. [cit. 2016-05-04]. Dostupné z: http://www.sztaki.hu/~szcsaba/papers/ecml06.pdf

[57] Michael Chung, J. S., Michael Buro: Monte Carlo Planning in RTS Games. [cit. 2016-05-06]. Dostupné z: https://skatgame.net/mburo/ps/mcplan.pdf

[58] Wakamatsu, H.: Motion Control font. [cit. 2016-05-06]. Dostupné z: https://www.behance.net/gallery/22542969/Motion-Control-a-Sturdily-Built-Font

Seznam použitých zkratek

RTS Real-time Strategy

FPS First Person Shooter

XML Extensible Markup Language

UML Unified Modeling Language

CMS Content Management System

DMS Document Management System

WPF Windows Presentation Foundation

SDK Software Development Kit

NDK Native Development Kit

JNI Java Native Interface

CSS Cascading Style Sheets

TCP Transmission Control Protocol

UDP User Datagram Protocol

BFS Breadth-first Search

DFS Depth-first Search

FSM Finite State Machine

NPC Non-Playable Character

MCTS Monte-Carlo Tree Search

UCT Upper Confidence Bound 1 applied to trees

MSVC Microsoft Visual C++

Instalační a uživatelská příručka

Instalační příručka

Data hry i každého ukázkového projektu se nachází ve složce *Data*. Tato složka musí být před spuštěním zkopírována do složky *bin* (VisualStudio provádí automaticky skriptem).

Kompilace enginu a ukázkových příkladů

Engine obsahuje metadata pro otevření v aplikaci Visual Studio 2015. Pro ostatní prostředí je nutné mít k dispozici kompilátor implementující standard C++ 14. Ke spuštění zkompilovaných aplikací je potřeba balíček *Visual C++ Redistributable Packages 2015*.

OpenFrameworks je možné stáhnout zde: `http://openframeworks.cc/download/`, kde se také nachází manuály pro různé platformy a prostředí. Je distribuován v podobě zdrojového kódu, tudíž není potřeba žádné linkování knihoven, pouze stačí naimportovat zdrojové kódy frameworku.

V případě editace v programu Visual Studio 2015 stačí pro kompilaci CogEngine pouze přepsat cesty k OpenFrameworks, aby mohl být projekt načten. Pro kompilaci ukázkových příkladů je nutné ještě nastavit cestu k enginu.

Zdrojové soubory enginu se nachází ve složkách *Source* a *3rdParty*.

Instalace hry

Pro platformu Android stačí nahrát do mobilního zařízení instalační soubor APK a ten spustit. Pro platformu Windows se hra neinstaluje, je ale nutné mít nainstalován balíček *Visual C++ Redistributable Packages 2015*

Kompilace hry pro platformu Android

Zkompilovat hru pro systém Android je poměrně náročný proces, který vyžaduje velké množství prerekvizit. Postup je však stejný jako u všech ostatních doplňků pro OpenFrameworks, je tedy možno využít manuál ze stránek `http://openframeworks.cc/download/`.

Jsou potřeba následující aplikace a balíčky:

- MinGW_w64
- MSYS
- Android NDK r10e x86
- JDK (1.8.0_45)
- JRE (1.8.0_45)
- Android SDK pro API 17
- Ant
- addon ofxAndroid

Aplikace se zkompiluje pomocí příkazu `make -j1 Release PLATFORM_OS=Android` .

Kompilace hry pro platformu Windows

Ke kompilaci je možno využít Visual Studio 2015 stejně jako u ukázkových příkladů. Kromě MSVC je však možno také využít kompilátor MinGW. K tomu je potřeba dodržet následující postup:

- nainstalovat msys2
- otevřít *shell* a napsat `pacman -noconfirm -needed -Sy bash pacman pacman-mirrors msys2-runtime`
- stáhnout OpenFrameworks distribuci pro msys
- otevřít msys shell a spustit OF/scripts/win_cb/msys2/install_dependencies.sh
- příkazem *make* ve složce OF/libs/openFrameworksCompiled/project zkompilovat OpenFrameworks
- nastavit proměnnou OF_ROOT na adresář OF/openframeworks/msys
- spustit ve složce projektu příkaz `make Release PLATFORM_OS=MINGW32_NT` a zkompilovat projekt

Manuál k enginu

Pro vývoj vlastního projektu je potřeba inicializovat engine. Inicializaci provádí třída, která musí dědit od `ofxCogApp` a zároveň musí být předána OpenFrameworks frameworku, který na ni naváže své vlastní komponenty.

Následující kód inicializuje engine pro obě platformy, Android i Windows. Inicializaci je možno provést buďto načtením XML souboru nebo procedurálně.

```
class ExampleApp : public ofxCogApp {

   void InitEngine() {
      ofxCogEngine::GetInstance().Init("config.xml");
      ofxCogEngine::GetInstance().LoadStageFromXml(spt<ofxXml>(new
          ofxXml("config.xml")));
   }

};

#ifdef ANDROID
#include <jni.h> // include java native interface
#endif

int main() {
   ofSetupOpenGL(800, 450, OF_WINDOW);
   ofRunApp(new ExampleApp());
   return 0;
}

#ifdef ANDROID
extern "C" { // this method is called from OFAndroid.java class
   void Java_cc_openframeworks_OFAndroid_init(JNIEnv* env, jobject thiz) {
      main();
   }
}
#endif
```

Inicializace enginu

Způsob práce se scénami, XML soubory a skripty je názorně ukázán v sadě doprovodných příkladů, které se nachází ve složce *Examples*. Ve složce *CogEngineLab* se nachází aplikace pro testovací účely, kterou je možno zkompilovat pro obě platformy.

Makra

Globální komponentu je možno získat pomocí makra `GETCOMPONENT(type)`. Pro aserce slouží makro `COGASSERT(condition, module, message)`. Pro logování je možno využít metody `CogLogDebug`, `CogLogInfo` a `CogLogError`, případně makro `COGLOGDEBUG`.

Fasáda

Pro často používané metody z různých *komponent*, které existují po celý běh aplikace a jsou nedílnou součástí enginu (`Logger`, `ResourceCache` apod.) byla vytvořena fasáda `Facade`. Všechny její metody začínají prefixem *Cog*, např. `CogGetScreenSize()` a `CogGetFrameCounter()`.

Logování

Logování je možné buďto do konzole nebo do souboru. Komponenta může být inicializována pomocí globálního nastavení v konfiguračním XML souboru nebo ručně.

```
CogLogError("Environment","Error while parsing aspect ratio for;
expected format xx/yy, found %s", aspectRatio.c_str());
```

Pomocí fasády je možno zalogovat příslušnou zprávu zavoláním jedné ze tří metod - `CogLogError` pro chybové zprávy, `CogLogDebug` pro ladící zprávy a `CogLogInfo` pro informační zprávy. Také je možno využít makro `COGLOGDEBUG(msg)`, které v release módu nic nedělá a šetří tak procesorový čas.

Součástí každé zprávy je název modulu, ze kterého zpráva pochází. Pro snazší ladění je pak možno libovolnou zprávu odfiltrovat, k čemuž slouží kolekce `includedModules` a `excludedModules` v komponentě `Logger`.

Měření času

Pro jednoduché měření času slouží třída `TimeMeasure`. Začátek měření se deklaruje pomocí makra `COGMEASURE_BEGIN(scopeName)` a konec pomocí `COGMEASURE_END(scopeName)`. Výstupem je pak souhrnný přehled o tom, kolik procent času jaká část kódu trvala, počet volání, délka provádění a frekvence volání za sekundu.

```
Report:: total 23796 ms
Engine_Draw:  Total [18%], Calls [1371],Dur[4319 ms],Freq[57.21]
Engine_Update: Total [32%], Calls [1371],Dur[7721 ms],Freq[57.21]
Update_Trans: Total [ 7%], Calls [315330],Dur[1716 ms],Freq[13250.64]
```

Ukázka výstupu měření

Automatické testy

Pro spuštění automatických testů stačí pouze nareferencovat příslušné hlavičkové soubory a spustit knihovnu *catch* - to vše provádí projekt, nacházející se ve složce *Tests*.

Herní příručka

Po spuštění hry se zobrazí hlavní menu - zde je možno zvolit buďto hru proti počítači (tlačítko *Single Game*) nebo hru dvou hráčů (tlačítko *Multiplayer*).

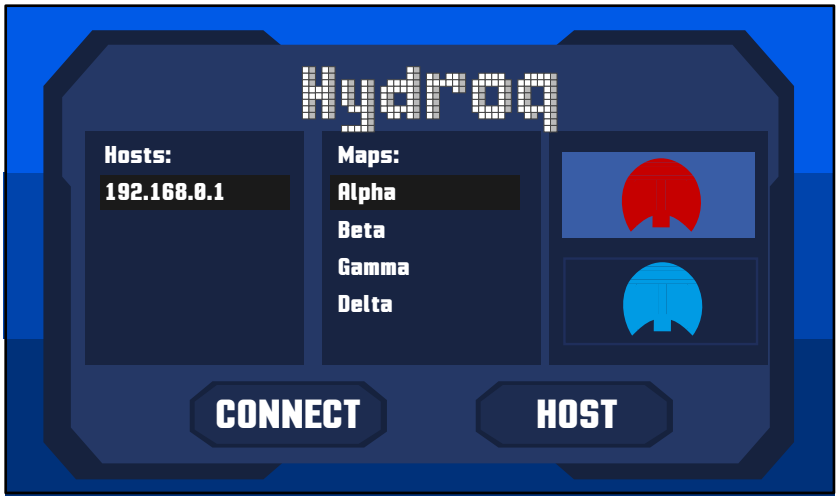
Pro návrat z jakékoliv scény či menu slouží tlačítko *back* u platformy Android a tlačítko *backspace* u platformy Windows.

Obrázek .4: Hlavní menu

V menu hry proti počítači si hráč vybere frakci poklepáním na ikonku s příslušnou barvou, vybere si mapu a tlačítkem *Play* hru spustí.

Obrázek .5: Menu pro hru proti počítači

V menu hry dvou hráčů může hráč buďto zvolit frakci, mapu a tlačítkem *Host* počkat, dokud se k němu nějaký klient nepřipojí. Pokud už je nějaký hostitel k dispozici, zobrazí se jeho IP adresa v levém panelu. Pokud na ni hráč klikne, označí se mu mapa, kterou si hostitel zvolil. Jakmile se hráč bude chtít připojit, stiskne tlačítko *Connect*.

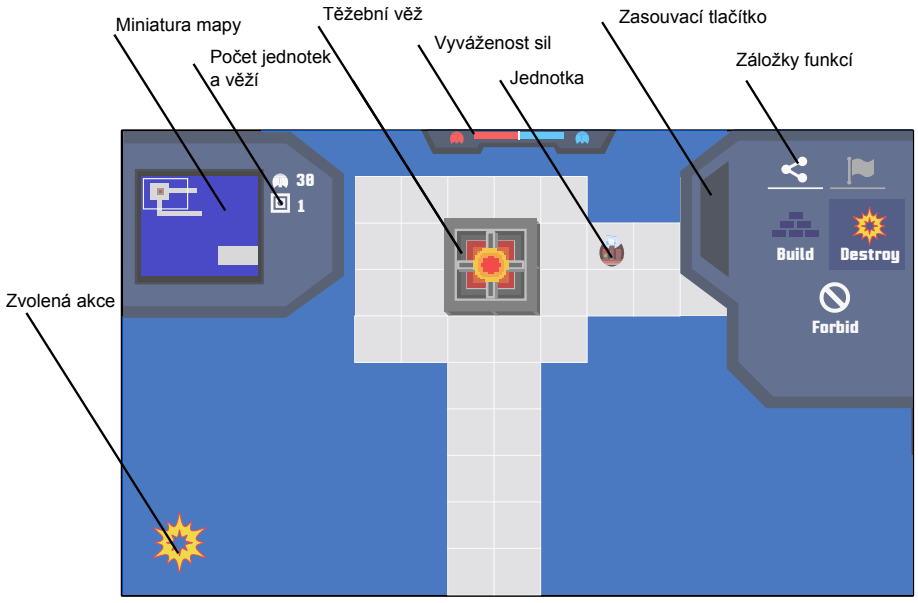


Obrázek .6: Hra dvou hráčů

Na obrázku .7 je vidět rozvržení herní scény. Hráč z pravého menu může zvolit ze dvou záložek čtyři akce - stavění cest, bourání cest, označení nepřístupné pozice a atraktor.

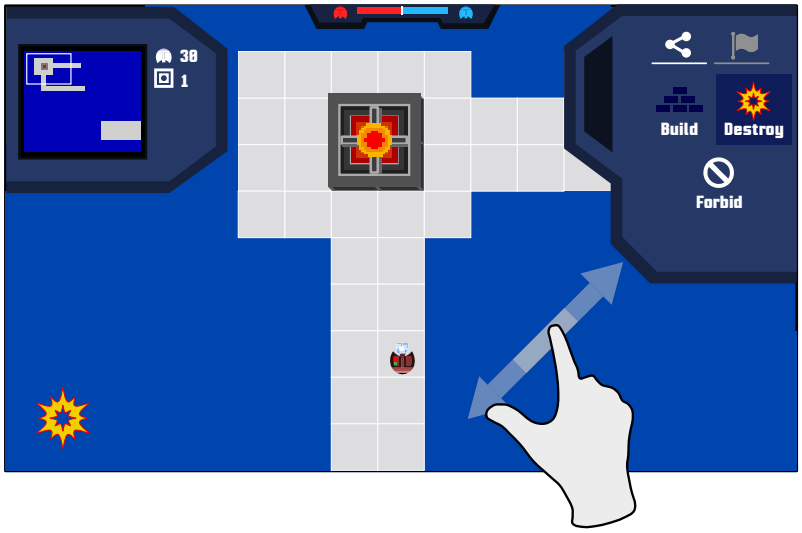
V horní liště se zobrazuje aktuální vyváženost sil podle počtu jednotek pro každou frakci.

Levé menu zobrazuje počet jednotek a těžebních věží daného hráče. Poklikáním na miniaturu mapy je možno se po této mapě přesouvat. Kliknutím na oblast těsně pod číselníky se levý panel zasune. Stejně tak je možné zasunout pravý panel kliknutím na tmavě modré tlačítko v pravém rohu.



Obrázek .7: Rozvržení herní scény

Posouvat mapu je možno buďto pomocí miniatury na levém panelu nebo tažením po herní ploše. Gestem *pinch* u platformy Android a kolečkem myši u Windows je pak možno mapu přibližovat a oddalovat.

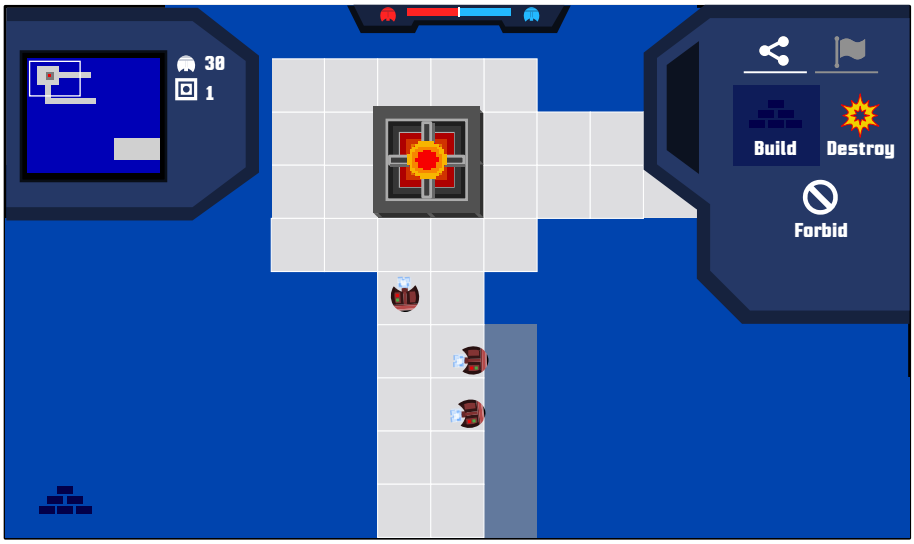


Obrázek .8: Ukázka pohybu po mapě

Stavění cest je možno provést zvolením funkce *build* a kliknutím na příslušné čtverce na mapě. Stejným způsobem je možné cestu zbořit pomocí funkce *destroy*. Funkce *destroy* může být také použita pro zrušení oblasti, kde se má postavit cesta.

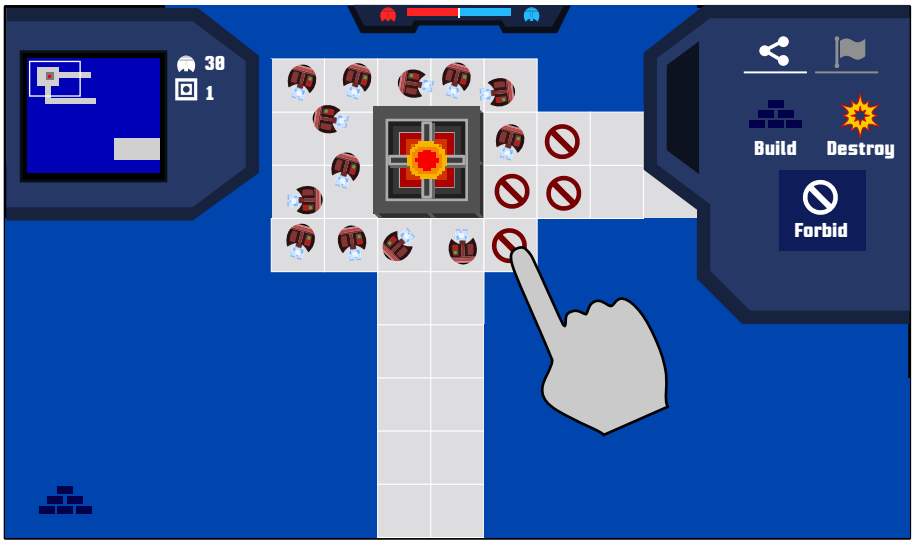
Obrázek .9: Stavění cest

Jakmile je oblast označena, začnou se k ní přesouvat volné jednotky.



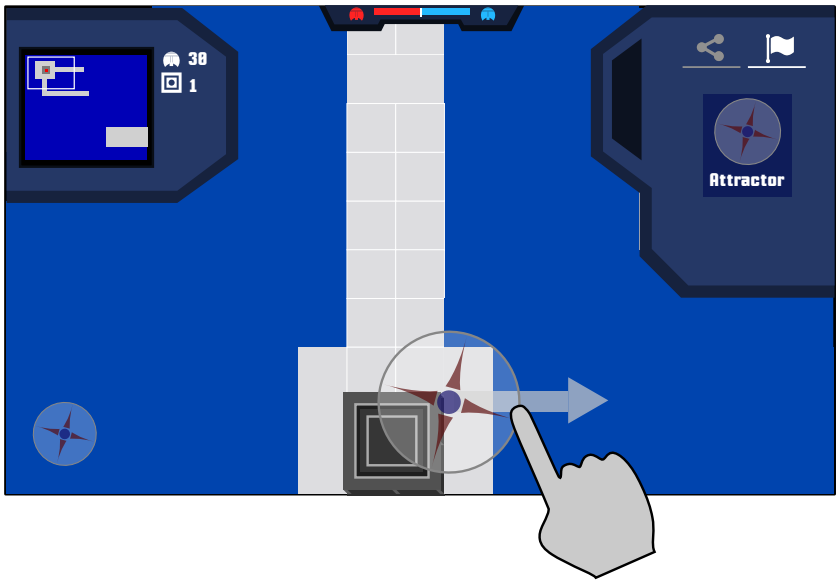
Obrázek .10: Přesun jednotek k označené pozici

Funkcí *Forbid* je možno zakázat pohyb jednotkám po určené oblasti. Tuto oblast mohou jednotky překročit pouze v případě, že mají zadaný úkol a neexistuje žádná jiná cesta.



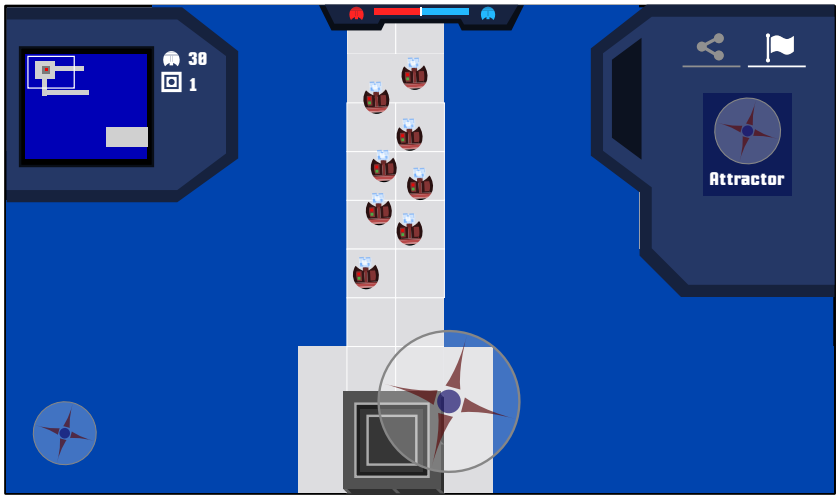
Obrázek .11: Zakázaná oblast

Pomocí funkce *attractor* je možné přikázat jednotkám, aby se přemístily do příslušné oblasti. Atraktor se vytvoří tažením přes určenou oblast - čím dále uživatel táhne, tím bude atraktor větší. Menší atraktor přitáhne méně jednotek a v případě více atraktorů se pak počet jednotek dělí mezi tyto atraktory.



Obrázek .12: Vytváření atraktoru

Atraktor je možno využít pro obsazení těžebních věží - jakmile se k prázdné věži dostane první jednotka, začne věž vytvářet jednotky pro frakci, která ji zabrala.

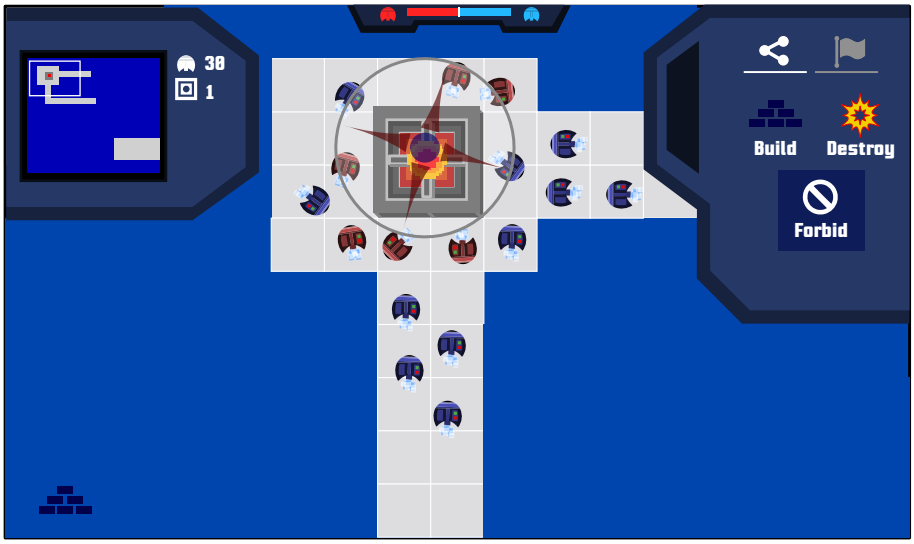


Obrázek .13: Přesun jednotek k atraktoru

Pro zabrání věže, kterou již nějaká frakce vlastní, je situace složitější. Je potřeba, aby se na čtvercích v bezprostřední blízkosti věže nacházelo více jednotek frakce, která chce věž zabrat, než těch, kterým věž patří.

Jakmile je věž zabrána jinou frakcí, přechází všechny jednotky vytvořené danou věží pod tuto frakci.

Hra končí, jakmile jeden z hráčů přijde o všechny jednotky.



Obrázek .14: Obsazování věže

Referenční aplikace

Kromě hry *Hydroq* byly v prvním prototypu enginu vytvořeny ještě dvě další hry - *Noah's Matching* a *CopterDown*.

Noah's Matching

Noah's Matching je logická hra na bázi pexesa. Téma se opírá o příběh Noemovy archy a cílem hráče je posbírat na herní ploše páry zvířat - samečka a samičku.

Hra obsahuje bohaté animace, velké množství obrázků a doplňkových funkcí jako odemykání nových úrovní a hru dvou hráčů.

Obrázek .15: Noah's Matching

CopterDown

CopterDown je klon legendární hry *Paratrooper*, kde je cílem sestřelit všechny vrtulníky a parašutisty, než vyprší čas.

Hra byla vytvořena v rámci prototypování první verze enginu, kdy bylo implementováno přepínání scén a podpora pro platformu Android. Grafika je pouze prozatímní, nicméně do budoucna je plánován další vývoj.

Obrázek .16: CopterDown

Obsah přiloženého CD

- `Readme.txt` ..... stručný popis obsahu CD
- `/Binary`
  - `/Hydroq` ..... spustitelná forma hry pro obě platformy
  - `/Examples` ..... spustitelná forma ukázkových aplikací
- `/Doc` ..... dokumentace
- `/Source`
  - `/CogEngine` ..... herní engine
  - `/CogEngineLab` ..... testovací aplikace
  - `/Examples` ..... sada doprovodných příkladů
  - `/Hydroq` ..... prototyp hry Hydroq
  - `/Tests` ..... projekt spouštící testy
  - `/thesis` ..... zdrojová forma práce ve formátu LaTeX
- `/Text`
  - `thesis.pdf` ..... text práce ve formátu PDF
- `/Video` ..... video-ukázka ze hry