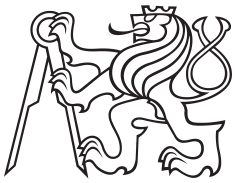
Bakalářská práce



České
vysoké
učení technické
v Praze

F3

Fakulta elektrotechnická
Katedra počítačů

Synchronizace distribuovaného stavu

Matyáš Neuvirt

Vedoucí: RNDr. Ondřej Žára
Studijní program: Softwarové inženýrství a technologie
Květen 2020

ZADÁNÍ BAKALÁŘSKÉ PRÁCE

I. OSOBNÍ A STUDIJNÍ ÚDAJE

Příjmení: **Neuvirt** Jméno: **Matyáš** Osobní číslo: **474460**

Fakulta/ústav: **Fakulta elektrotechnická**

Zadávající katedra/ústav: **Katedra počítačů**

Studijní program: **Softwarové inženýrství a technologie**

II. ÚDAJE K BAKALÁŘSKÉ PRÁCI

Název bakalářské práce:

Synchronizace distribuovaného stavu

Název bakalářské práce anglicky:

Distributed state synchronization

Pokyny pro vypracování:

Nastudujte problematiku synchronizace stavu mezi autoritativním serverem a klienty s různou latencí. Soustřeďte se zejména na algoritmy a přístupy využívané v hrách více hráčů.
Otestujte a naimplementujte algoritmy, které zajistí, aby:
a) klient vždy zobrazoval nejnovější stav serveru
b) klient synchronizoval správně čas simulace a měřil a zobrazoval míru latence
c) klient zvyšoval plynulost zobrazování simulace pomocí extrapolace a volitelně též interpolace
Demonstraci funkčnosti řešení naimplementujte formou hry. Tato bude napsána v JavaScriptu, bude proto fungovat ve webovém prohlížeči (klienti) a na serveru (NodeJS). Ukažte, jak simulace reaguje na nestálé kvalitativní parametry internetového připojení. Pomocí kvalitativního uživatelského testování ověřte funkčnost použitých algoritmů. Uvažte, může-li za této situace simulaci ovlivnit nespolehlivý (potenciálně podvádějící) klient.

Seznam doporučené literatury:

Glazer, Madhav: Multiplayer Game Programming, Addison-Wesley 2015, ISBN 0134034309
Source Multiplayer Networking, https://developer.valvesoftware.com/wiki/Source_Multiplayer_Networking
Ondřej Žára, Koncepty a triky real-time her, https://ondras.zarovi.cz/slides/2017/devel/

Jméno a pracoviště vedoucí(ho) bakalářské práce:

RNDr. Ondřej Žára, Katedra počítačové grafiky a interakce

Jméno a pracoviště druhé(ho) vedoucí(ho) nebo konzultanta(ky) bakalářské práce:

Datum zadání bakalářské práce: **13.02.2020** Termín odevzdání bakalářské práce: **22.05.2020**

Platnost zadání bakalářské práce: **30.09.2021**

RNDr. Ondřej Žára podpis vedoucí(ho) práce	podpis vedoucí(ho) ústavu/katedry	prof. Mgr. Petr Páta, Ph.D. podpis děkana(ky)

III. PŘEVZETÍ ZADÁNÍ

Student bere na vědomí, že je povinen vypracovat bakalářskou práci samostatně, bez cizí pomoci, s výjimkou poskytnutých konzultací.
Seznam použité literatury, jiných pramenů a jmen konzultantů je třeba uvést v bakalářské práci.

. _____
Datum převzetí zadání

_____
Podpis studenta

Poděkování

Děkuji panu RNDr. Ondřeji Žárovi za veškeré odborné konzultace a za vedení bakalářské práce i jí předcházejícího semestrálního projektu.

Prohlášení

Prohlašuji, že jsem předloženou práci vypracoval samostatně, a že jsem uvedl veškeré použité informační zdroje v souladu s Metodickým pokynem o dodržování etických principů při přípravě vysokoškolských závěrečných prací.

V Praze 22. května 2020

Abstrakt

Cílem práce je popsat a překonat výzvy okamžité distribuce stavu simulace ze serveru, která se zpracovává několikrát za sekundu, na několik klientů v reálném čase. Dalším úkolem je zajistit zobrazení plynulého průběhu simulace na straně klienta. Součástí vypracování je také demo využívající principy zmíněné v práci. Serverová část dema je implementovaná v Node.js, klientská část je webová aplikace.

Klíčová slova: node.js, JavaScript, server, klient, extrapolace, interpolace, synchronizace

Vedoucí: RNDr. Ondřej Žára

Abstract

The goal of this thesis is to study and overcome challenges that arise while implementing quick response system from fast paced simulation running on a remote host to multiple clients in real time. Another task is to ensure fluid visualization of said simulation on the client side. Included with this thesis is a demo based on acquired knowledge. Server part is implemented using Node.js. Client part is a web application.

Keywords: node.js, JavaScript, server, client, extrapolation, interpolation, synchronization

Title translation: Distributed state synchronization

Obsah

Testování a závěr

Přílohy

Obrázky

Tabulky

Kapitola 1
Úvod

1.1 Předmluva

Při přidávání síťových funkcionalit do aplikace jakéhokoliv typu mohou i z přímočarých úkolů vyvstat zajímavé problémy, kterými bychom se při lokálním zpracování vůbec nemuseli zabývat. V první řadě jde o navázání spojení s protistranou, v druhé pak vypořádat se s nedostatky použitého způsobu komunikace, změnami v propojování vzdálených komponent a sdílení stavu napříč zařízeními.

Online hry více hráčů v reálném čase se stávají dobrým prostředím pro testování kvality řešení těchto komplikací, neboť bývají, na rozdíl od jiných síťových aplikací, odlišné svými specifickými kvalitativními požadavky. Krátkodobé výpadky toku informací, dlouhé zpoždění v komunikaci, trhaná vizualizace nebo opožděná reakce na vstupy jsou sice obecně nepříjemné neduhy, ale v tomto prostředí se jedná o závažné nedostatky.

V průběhu této bakalářské práce jednotlivé problémy spojené s distribuovanými systémy, hlavně v kontextu videoher, nejdříve teoreticky rozeberu a poté podrobněji popíši ty, které jsem implementoval do jednoduché hry pro demonstraci účinnosti nastudovaných technik. Na závěr analyzuji výsledky uživatelského testování a zhodnotím efektivitu různých implementací a prozkoumám, jakými způsoby by bylo možné finální výstup dále vylepšit.

Zde prezentované mechanismy jsou použitelné kromě her i v širším okruhu různorodých případů. Obzvláště způsoby, kterými můžeme efektivně zvýšit plynulost vykreslování, najdou využití v aplikacích s limitovaným množstvím dat nebo nedostatečnou schopností data zaznamenávat.

1.2 Popis problémů

Během přípravy online dema, jehož cílem je zajistit synchronizaci společně s plynulým vykreslováním, jsem se potýkal s několika problémy. Ty hlavní si nyní zvlášť popíšeme.

1.2.1 Distribuce rychle proměnného stavu

Prvním úkolem je zajistit, aby měl klient vždy k dispozici nejnovější stav herní simulace, která velice rychle mění své vnitřní vlastnosti, jelikož o demu uvažujeme jako o online hře v reálném čase. Klient tak musí ze serveru data získávat s vysokou frekvencí.

1.2.2 Zjištění latence mezi serverem a klientem

Vlivem zpoždění není možné, aby klient viděl stav simulace přesně takový, jaký je ve stejnou chvíli na serveru. Veškerá interakce ze strany klienta reaguje na stav simulace v minulosti, ale na serveru ovlivňuje přítomnost. Zjištění tohoto zpoždění nám při užití různých technik umožní snížit dopady tohoto jevu.

1.2.3 Synchronizace stavu napříč klienty

Protože se snažíme všem klientům v rámci hry zajistit stejné podmínky, je naprosto klíčové, aby každý uživatel viděl simulaci ve stejném stavu jako ostatní. Především mají-li možnost ovlivňovat ostatní hráče nebo na momentální stav hry nějak reagovat. I když dostanou k dispozici stejný stav simulace, ještě to neznamená, že uvidí hru stejně. Na základě jejich *latence* (viz 2.2 Zjištění latence a posunu hodin mezi serverem a klientem) ji vidí jinak zpožděnou.

1.2.4 Plynulé zobrazování průběhu simulace

Pro plynulé vykreslování musíme mít vždy k dispozici nový stav simulace alespoň v takové frekvenci, ve které chceme simulaci zobrazovat. Málokdy se podaří tento stav udržet, ať už kvůli ztrátě dat během přenosu, zpoždění při přenosu nebo protože server negeneruje stavy v dostatečném množství pro plynulé zobrazení. Pro potřeby této práce považujme za přijatelnou vykreslovací frekvenci alespoň 60 snímků za sekundu, která se většině lidí jeví jako plynulá [1].

Část I

Teoretická příprava

Kapitola 2
Principiální řešení hlavních problémů

Jednotlivé problémy lze uspokojivě vyřešit aplikací technik popsaných v této kapitole. Bližším detailům o implementaci se budeme věnovat později, nejdříve si je zde obecněji popíšeme.

V nadcházejícím textu budeme pracovat s následujícími pojmy:

Server – místo, kde probíhá simulace herního světa. Její stav server pravidelně rozesílá klientům.

Klient – program, který komunikuje se serverem, přijímá a zobrazuje nejnovější stav simulace poslaný serverem.

Uživatel – fyzická osoba obsluhující program klienta.

2.1 Distribuce stavu a uživatelských vstupů s vysokou frekvencí

Protože mezi serverem a klientem musí pravidelně proudit množství dat, běžné techniky používané ve webovém prostředí (HTTP GET požadavky, Ajax, longpolling), nejsou pro tento úkol dostatečně robustní. Jelikož stav posílaný serverem požadujeme okamžitě, jakmile je k dispozici, nemá smysl, aby se klient na stav pravidelně serveru dotazoval. Mnohem účelnější je, aby server nově vygenerovaný stav klientovi okamžitě zaslal skrze udržovaný kanál, kdy k navázání spojení dojde pouze jednou. Pak už mohou data libovolně proudit oběma směry.

2.2 Zjištění latence a posunu hodin mezi serverem a klientem

Znalost latence je zásadní informace, pomocí které dokážeme zjistit časový posun mezi hodinami klienta a serveru. Bez těchto parametrů není klient schopný správně pracovat s časovými značkami obdrženými od severu.

Latence – doba trvání cesty dotazu na protistranu + doba trvání cesty odpovědi protistrany.

2.2.1 Měření latence

Měření latence je jednoduchý a přímočarý úkol. Jedna strana spojení vyšle zprávu (tzv. *ping*) a uloží si čas, kdy ji odeslala. Protistrana na dotaz po přijetí neprodleně odpoví vlastní zprávou (tzv. *pong*). Dotazovatel si na základě času příchodu odpovědi a času odeslání dotazu jejich odečtením zjistí, jak dlouho celý proces trval, čímž zjistil svou latenci [2].

$$latence = T_{\text{přijetí}} - T_{\text{odeslání}} \tag{2.1}$$

2.2.2 Synchronizace serverových a klientských hodin

Aby klient mohl synchronizovat svůj čas se serverovým, využije informaci o latenci. Server ale musí k odpovědi (zprávě *pong*) na dotaz klienta připojit navíc i časovou značku označující přítomnost na serveru v době, kdy přijal *ping*. Klient v čase po přijetí zprávy *pong* odečte svoji „přítomnost“ od serverové, a navíc musí přičíst polovinu hodnoty naměřené latence, protože o tento úsek je časová značka označující přítomnost serveru zastaralá (kvůli doby trvání cesty zpět ke klientovi). Pro potřeby výpočtu pracujeme s předpokladem, že latence má symetrický charakter, tzn. trvání cesty zprávy z klienta na server i ze serveru na klienta trvá stejně dlouho [2].

$$\text{časový posun klienta} = T_{\text{klient}} - T_{\text{server}} + \frac{latence}{2} \tag{2.2}$$

Pokud zůstává platný předpoklad o symetrii latence, její hodnota nemá na velikost časového posunu žádný vliv. Ten poté klient používá ke správné interpretaci časových značek přicházejících od serveru.

Pro naše účely je bezpečné očekávat, že na serveru i klientovi plyne čas stejným tempem, jednou naměřený rozdíl je v čase neměnný.

2.3 Synchronizace stavu napříč klienty

Synchronizace stavu simulace je naprosto klíčový prvek, především jestliže její součástí má být několik klientů najednou s možností ovlivňovat ji a tím i sebe navzájem.

Abychom lépe pochopili, jak a proč k synchronizaci dochází a kdy by mohla být ohrožena, popíšeme si několik aspektů spojených s komunikací klientů.

2.3.1 Schéma síťové provozu

Jako schéma propojení všech účastníků simulace využijeme schéma *klient–server*. Toto schéma usnadňuje vytváření spojení a jejich udržování, protože každý klient si udržuje pouze jedno připojení, a to k serveru. Tímto se usnadňuje i distribuce dat, která putují pouze od jednoho zdroje. Případné problémy specifického klienta neovlivní nikoho z ostatních, s těmito problémy se vypořádá pouze server.

2.3.2 Autoritativní server

Pro server, se kterým budou klienti komunikovat, zvolíme koncept *autoritativního serveru*. Takový server je jediným zdrojem pravdy a všichni se mu podřizují. Pokud by klient prováděl vlastní výpočty v simulaci a během jakéhokoliv výpočtu došel k jinému výsledku než server, vždy platí výsledek příchozí ze serveru.

2.3.3 Věrohodnost klienta

Klient na server posílá pouze akce, které by chtěl provádět, nikdy jejich výsledky, protože není serverem považován za důvěryhodného a autoritativní server taková data nepřijímá. I když běžný klient neumožní uživateli vykonávat akce, na které nemá nárok a ani je neposílá na server, server vždy musí u sebe zkontrolovat, jestli k nim byl klient oprávněn. Neoprávněné akce, zaslané ať už podvodným klientem nebo chybou na straně klienta, jsou zahazovány[3].

Jestliže se klient snaží podvádět manipulací stavu, ovlivňuje tak stav pouze u sebe samotného, podvržený stav se na server nerozšíří a neovlivní průběh simulace u jiných klientů ani na serveru. Podvodný klient podvrhuje stav jedině svému uživateli. Například v případě, kdy klient ukazuje uživateli plný počet životů i po utrpěném zásahu, server stále chápe hráče jako zraněného. Při dalším zásahu může být uživatel poražen, přestože mu podvržená prezentace herního světa stále ukazuje dostatek zdraví.

2.3.4 Závěr

Protože klienti čerpají informace o simulaci výhradně od serveru, který má jako jediný možnost ovlivňovat její průběh, je zaručeno, že všichni obdrží stejná data. Jediná možnost, kdy nastane desynchronizace stavu, je na podvodném klientovi, a to pouze u něj samotného. Integrita simulace pro poctivé klienty zůstává v pořádku.

Pro zajištění situace, kdy klienti vidí simulaci ve stejném stavu bez ohledu na jejich rozdílnou latenci, můžeme využít stejné techniky, které použijeme pro zvýšení plynulosti vykreslování hry, viz další sekce.

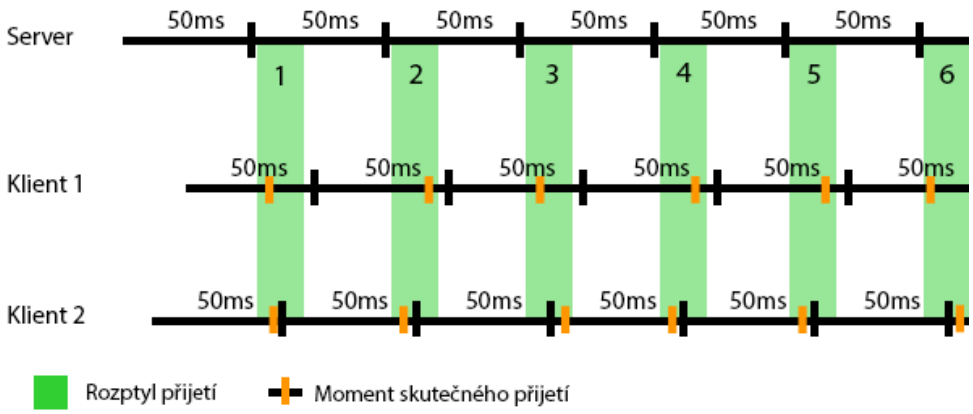
2.4 Zvýšení plynulosti vykreslování

Pro kvalitní uživatelský zážitek ze hry je stěžejní, aby ji hráč vnímal jako plynulou.

Snapshot – otisk stavu simulace v konkrétním čase.

Pro plynulé zobrazení potřebujeme pravidelně překreslovat stav hry. Snažíme se, aby klient viděl nejaktuálnější stav hry, proto vykreslujeme vždy poslední obdržený snapshot. Jestliže by klient měl vykreslovat nové snímky v pravidelných intervalech, nemusí vždy mít k dispozici nový, dosud nevykreslený snapshot, i kdyby frekvence překreslování byla rovna frekvenci, kterou server snapshoty generuje. Na obrazovce bez nově příchozího snapshotu i při

překreslení nedojde k žádné změně, a navíc by se jednalo o zbytečnou zátěž pro hardware. Kvůli nepředvídatelným časovým odchylkám se může snapshot opozdit a dorazit až po příkazu klienta vykreslit nový snímek. Pokud před vykreslením náhle dorazí ještě další snapshot, předchozí se už nikdy nevykreslí, protože již není aktuální. Roli také hraje vzájemný časový posun intervalů, kdy klient vykresluje a server data zasílá. Při nevhodném překrytí může být popsaný problém nejen více znatelný, ale každý klient zároveň pozoruje hru s rozdílným zpožděním, viz obrázek 2.1.



Obrázek 2.1: Demonstrace problému periodického překreslování posledního přijatého stavu

Pro ilustraci uvažujme případ, kdy server každých 50 ms zašle nový snapshot a klient každých 50 ms překreslí obrazovku. Rozptyl přijetí je způsobený kolísáním latence a nepřesností časování game loop (viz sekce 5.1 Game loop). Protože se každý klient k serveru připojil v jiný čas, jsou vůči serveru rozdílně časově posunutí. Klient 1 pravidelně obdrží nový stav před vykreslením. Nestává se mu, že by zbytečně obrazovku překresloval, ale v době vykreslení vidí stav, který je už několik milisekund zastaralý. Klient 2 je shodou okolností časově posunut nepříznivým způsobem. Stav 1 ze serveru přijal těsně před vykreslením a vidí tak aktuálnější stav než Klient 1[1]. Situace se opakuje pro stav 2. Stav 3 ale dorazil příliš pozdě. Klient 2 tak dvakrát po sobě vidí stejný stav hry – simulace se na chvíli jeví jako „zamrzlá“, až poté dorazí stav číslo 3. Než by se měl stav 3 zobrazit, dorazí i nový stav 4. Klient 2 tak nikdy neuvidí stav 3 a obraz poskočí rovnou do stavu 4. Tím se pocit zamrznutí simulace ještě více prohloubí. Uživatelský zážitek tímto problémem výrazně degraduje.

Z obrázku je také patrné, že překreslit stav pouze při obdržení nového snapshotu také není ideální, protože nemusí přicházet v pravidelných rozestupech. V horším případě se může snapshot po cestě sítí i úplně ztratit, přičemž by

[1] Časový rozdíl je sice tak malý, že výhoda Klienta 2 je zanedbatelná, ale i tento problém bychom rádi vyřešili.

k překreslení obrazovky vůbec nedošlo. Počet snímků za sekundu by kolísal a obraz se nejevil jako plynulý.

Pokud si ovšem klient pamatuje více než jenom poslední stav, je možné odhadnout dočasný mezistav využitý pouze pro potřeby vykreslení, a to z informací v předchozích snapshotech. Takto můžeme simulaci zobrazovat ve více snímcích, než je počet obdržených stavů.

V kontextu klient–server, kdy klient zobrazuje stavy posílané serverem, jsou techniky pro počítání mezistavů velice užitečným nástrojem, jak překonat nižší frekvenci zásobování klienta stavy simulace serverem, než je jeho požadovaná vykreslovací frekvence. Užití nacházejí, i když dojde ke ztrátě snapshotu nebo jeho zpoždění. Vygenerováním očekávaného mezistavu můžeme ztrátu/zpoždění snapshotu zamaskovat. Způsobem, jak takové mezistavy generovat, může být *extrapolace* nebo *interpolace*. Podrobněji jsou popsané v samostatných kapitolách. Především jimi umožňujeme zvýšení vykreslovací frekvence v libovolné výši a, i přes zastaralost snapshotů v době vykreslení, můžeme zkusit zobrazit aktuální stav na serveru odvozením z informací v minulých stavech.

Kapitola 3

Extrapolace

Extrapolace je jednoduchá technika, kterou lze využít k předpovědi velikosti dané hodnoty v budoucnosti.

3.1 Princip

Hlavním principem extrapolace je předpověď stavu objektu na základě jeho předchozích známých stavů. Klient si kromě nejnovějšího přijatého stavu musí pamatovat ještě ten předchozí.

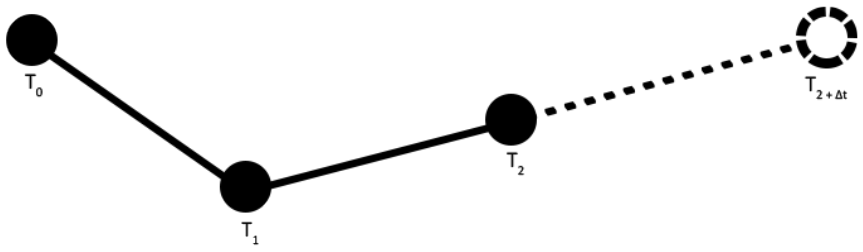
Jestliže se objekt v čase T_0 nachází na pozici $x_0 = 0$ a v čase T_1 na pozici $x_1 = 1$ a pohybuje se spojitě, můžeme předpovědět, kde bude v čase T, kdy $T_0 < T_1 < T$, na základě parametru t $(0, \infty)$.

$$t = \frac{(T - T_1)}{(T_1 - T_0)} \tag{3.1}$$

Parametr t popisuje, kolikrát do extrapolace promítáme velikost posunu.

$$x_e = (x_1 - x_0) * t + x_1 \tag{3.2}$$

Zjistíme velikost posunu během T_0 a T_1 a v závislosti na parametru t ho přičteme k poslední známé pozici.



Obrázek 3.1: Znázornění extrapolace. Čárkovaně je znázorněna projekce vycházející ze stavu T_1 a T_2 (na rozdíl od rovnice, kterou popisujeme s časy T_0 a T_1)

3.2 Využití

Extrapolací můžeme zvýšit plynulost zobrazování simulace, pokud nový stav od serveru vůbec nepřijde nebo dorazí pozdě, nebo pokud klient vykresluje simulaci rychleji, než server stavy generuje a distribuuje.

Nejčastěji predikovanou vlastností objektů bývá jejich pozice ve světě. Pokud víme, kde se objekt nacházel v minulosti (a klient se zpožděním vidí herní svět vždy v minulosti), můžeme provést projekci na přímku procházející jeho dvěma posledními známými pozicemi a předpovědět, kde se nachází jakkoli daleko do budoucnosti. Pokud se vektor rychlosti objektu nezměnil, predikce bude přesná.

Aby klient zjistil, jak vypadá stav hry na serveru, může předpovědět stav hry v čase $T + \frac{latence}{2}$. Takto může i klient se zpožděním relativně přesně pozorovat aktuální stav herního světa.

3.3 Výhody a nevýhody

Výhodou extrapolace je její relativně snadná implementace. Ve většině případů poskytuje přijatelný způsob, jak uživateli zamaskovat ztrátu nebo zpoždění obdržení stavu simulace ze serveru. Dalším pozitivem je také, že pro její uspokojivé využití stačí upravit pouze klientský kód. V implementaci serveru není nutné provádět jakékoliv změny.

Nevýhodou bývá nepřirozený posun objektů, jestliže během chybějícího stavu, který se snažíme zamaskovat, dojde k výrazné změně v rychlosti nebo směru pohybu. S větší změnou těchto atributů dochází k větší odchylce mezi predikcí a skutečným stavem. Pří nápravě pozice objektu je pak znatelnější náhlé posunutí na správnou pozici.

Příkladem, kdy extrapolace vytváří méně výrazné artefakty je použití u závodních her. U vozidla v pohybu se nepředpokládá viditelně náhlá změna v rychlosti nebo směru pohybu, dochází k nim relativně plynule (vyjma srážky, kdy auto mění své zrychlení nárazově). Chyba v předpovědi způsobená zastaralou informací o momentální rychlosti vozidla je tak méně znatelná.

Další nevýhodou je závislost kvality extrapolace na latenci klienta. Při vyšší latenci predikujeme dále do budoucnosti bez znalosti skutečného stavu, čímž vzniká větší prostor pro chyby.

Kapitola 4
Interpolace

Velice podobně extrapolaci, i u interpolace se jedná o projekci na základě známých stavů a času k vytvoření dočasného mezistavu využitého pro vykreslení.

4.1 Princip

Hlavním rozdílem oproti extrapolaci je, že místo využití předchozích známých stavů k předpokladu, kde objekt bude, můžeme vykreslit, kde objekt *byl*.

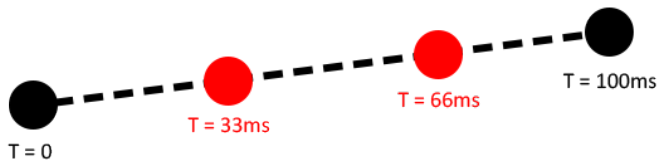
Jestliže se objekt v čase T_0 nachází na pozici $x = 0$ a v čase T_1 na pozici $x = 1$ a objekt se pohybuje spojitě, není problém určit, kde se nacházel kdykoliv v čase T, kdy $T_0 < T < T_1$, s využitím parametru t, který nabývá hodnot v intervalu (0,1).

$$t = \frac{(T_1 - T)}{(T_1 - T_0)} \tag{4.1}$$

Parametr t relativně popisuje, jaký čas mezi T_0 a T_1 nás zajímá.

$$x_i = (x_1 - x_0) * t + x_0 \tag{4.2}$$

Zjistíme velikost posunu během T_0 a T_1 a v závislosti na parametru t ho přičteme k pozici v T_0. Protože $0 < t < 1$, pohybujeme se pouze v rozmezí pozic v T_0 a T_1.



Obrázek 4.1: Znázornění interpolace. Černě jsou znázorněné známé stavy, červeně interpolované pozice.

Jestliže není k dispozici stav před časem T a po čase T, nemáme pro interpolaci dostatek informací. V této chvíli je vhodné pro zachování plynulosti dynamicky přejít zpět k extrapolaci.

4.2 Typy interpolace

Typů interpolací je několik, nejjednodušším typem je lineární interpolace (popsána rovnicí 4.1 a 4.2), která je nejvšestrannější volbou, nemáme-li ke konstrukci mezistavů další informace.

Uvažujme o tělesu pohybujícím se po kružnici. Protože server poskytuje informace o pozici v omezeném počtu snapshotů, při vysoké frekvenci vykreslování sice můžeme zobrazit perfektně plynulý pohyb, ale kvůli omezenému množství informací se bude jednat o pohyb po n-úhelníku.

Jedním ze způsobů, jak skutečně zobrazit pohyb po kružnici s lineární interpolací je změnit způsob reprezentace z bodových souřadnic na polární – pozici x a y nyní chápeme jako střed kružnice, dále posíláme vzdálenost od středu a úhel. Lineární interpolací úhlu jsme schopni zobrazit přesný pohyb po kružnici. Problémem této metodiky je potřeba nového souřadnicového systému. Pokud by objekt byl schopný se pohybovat i mimo kružnici, vznikají další komplikace v rozhodování, jak interpretovat souřadnice poskytnuté serverem.

Druhým řešením je použít místo lineární interpolace jiný typ. Abychom mohli správně vyhodnotit, jaká interpolace je nejvhodnější, musíme mít k dispozici další informace o způsobu pohybu objektu. Při správně zvolené metodě interpolace a dostatku informací o zrychlení je možné dosáhnout kvalitních výsledků pohybu po křivkách už při pouhých 10 snapshotech za sekundu [4]. Tyto techniky interpolace ale nejsou v rozsahu práce a dále už se jimi zabývat nebudeme.

Další možností zpřesnění vykreslování je zvednutí frekvence generování snapshotů zasílaných serverem. Pokud je frekvence dostatečně vysoká, odchylky od skutečného pohybu tělesa po křivce se i přes lineární interpolaci mezi vybranými body křivky stávají méně viditelnými.

Problém s přesností se nemusí týkat pouze pohybu po křivce, ale také náhlých změn v pohybu tělesa, které netrvaly dostatečně dlouho, aby je server zaznamenal do snapshotu. Například, jestliže objekt nárazově změnil svůj směr pohybu v čase $t_0 = 0\ ms$, $t_1 = 25\ ms$, $t_2 = 50\ ms$, ale snapshoty se odesílají každých 50 ms a obsahují pouze stav v daný okamžik, uživatel uvidí zjednodušený pohyb po úsečce mezi místy, kde se objekt nacházel v čase t_0 a t_2. Na výsledku pohybu se sice nic nemění, vidíme ho ovšem v menším detailu. Ve hře, kdy hráč nemá možnost ovlivnit rychlost svého pohybu, ale pouze směr, by se v popsaném příkladě náhle jevil, jako by zpomalil nebo se vůbec nehnul, jednalo by se o dva navzájem protichůdné pohyby. Tato situace není ideální a může vést ke zmatení ostatních hráčů.

Možným řešením je neposílat absolutní stav simulace v čase pořízení snapshotu, ale celou sekvenci pozic v intervalu předchozího a nynějšího snapshotu. Jinou a jednodušší možností je zpracovat směr hráče pouze na začátku daného

kroku a počítat s tímto směrem po celou dobu jeho trvání [1].

4.3 Využití

U klienta je kvůli latenci nemožné zobrazit stav simulace přesně tak, jak se v dané chvíli nachází na serveru. Tomuto se můžeme pouze přiblížit technikou extrapolace, kdy ale čas od času dochází k chybám a viditelným artefaktům. Možným způsobem, jak zobrazit simulaci plynuleji, než dostáváme snapshoty ze serveru, a vyvarovat se chyb způsobených extrapolací, je zobrazovat hru v minulosti a pomocí interpolace doplnit chybějící mezistavy.

4.4 Výhody a nevýhody

Nespornou výhodou interpolace je její přesnost a uživatelsky přívětivé zvýšení plynulosti za předpokladu, že využijeme mechanismy na kompenzaci interpolačního zpoždění (viz sekce4.5 Kompenzace interpolačního zpoždění). Kvůli náročnosti implementace ale ukázková hra tyto mechanismy neobsahuje.

Nevýhodou je, že si musíme pamatovat minimálně dva poslední snapshoty a až poté můžeme zobrazovat stav simulace ve chvíli mezi nimi. Abychom mohli konzistentně interpolovat, musíme zajistit, abychom vždy měli dopředný snapshot v zásobě. Pro tento účel na straně klienta vložíme úmyslné *interpolační zpoždění* – nově přijatý snapshot tak prozatímně pouze uchováváme v zásobě. Počet snapshotů nutných mít v zásobě je závislý na zavedeném interpolačním zpoždění, latenci a frekvenci generování snapshotů serverem. Jakmile byl snapshot „plně prointerpolovaný“, můžeme ho uvolnit z paměti. Aby žádný klient nebyl v nevýhodě, musí u sebe každý zavést stejné zpoždění.

Pokud je latence klienta větší než interpolační zpoždění, nemůže interpolaci využít, protože kvůli velké latenci jsou i nově přijaté stavy pro dané interpolační zpoždění příliš zastaralé. Po zvýšení interpolačního zpoždění u takového klienta je sice možné opět interpolovat, ale potom by svět viděl oproti ostatním klientům pozadu a dostal by sám sebe do nevýhody. I když bychom interpolační zpoždění mohli pro klienty, kteří jej latencí jen drobně překračují, navýšit, vhodnější je využít extrapolaci.

Uvažujeme-li o klientovi s latencí $T_l = 110\ ms$. Při běžných podmínkách bychom museli extrapolovat 110 ms dopředu, protože ale primárně interpolujeme se zavedeným zpožděním T_i, nechť se $T_i = 100\ ms$, stačí nám extrapolovat pouze do času $T_l - T_i = 10\ ms$. Extrapolace 110 ms dopředu je velice náchylná k chybám, v případě pouhých 10 ms nejsou problémy výrazné.

Důležité je správně zvolit délku zpoždění v závislosti na periodě, v jaké stavy běžně přicházejí ze serveru. Rozumnou hodnotou zpoždění interpolací pro simulaci běžící 30krát za sekundu je 100 až 150 ms. Latence 100 ms je hodnota, do které se většina uživatelů vejde a zároveň je toto zpoždění stále únosné pro relativně pohodlné ovládání hry, obzvláště při využití mechanismu kompenzace zpoždění. Navíc nám tato prodleva poskytuje dostatek prostoru

[1] Tento způsob využívá přiložené demo.

nashromáždit nutné množství snapshotů (které v ideálním případě dorazí každých 33.3 ms, pokud simulace běží s frekvencí 30krát za sekundu), i pokud by došlo k náhodné ztrátě nějakého z nich. [5]

4.5 Kompenzace interpolačního zpoždění

V prostředí rychlé online hry je nežádoucí sledovat její průběh zpožděný, kromě latence, navíc ještě interpolací. Prohlubuje se tak problém, kdy se pohybující objekty u klienta jeví na místě, kde z pohledu serveru již nějakou dobu nejsou. I jednoduchý úkol, jako pouze kliknout na pohybující se objekt myší by se mohl stát obtížným. Uživatel ze svého úhlu pohledu může kliknout přesně na objekt, avšak na straně serveru se může nacházet úplně jinde. Znalý uživatel, který ví, že vidí minulost, může předvídat, kde se objekty nachází v „přítomnosti“. Jestliže jsou možnosti uživatele ovlivnit průběh simulace minimální, nemusí nás zpoždění tolik trápit, neboť má méně možností, jak si zpoždění uvědomit.

Abychom překonali tuto značnou překážku při využívání interpolace, server si musí pamatovat více stavů hry a přetáčet čas zpět do doby, kdy uživatel u sebe akci vykonal. Kdy akce z pohledu klienta nastala zjistíme z jím přiložené časové značky[2]. Server tak sám může dočasně přetočit a interpolovat stav simulace do chvíle, jak ji viděl klient ze svého úhlu pohledu, a správně vyhodnotit důsledky. [5]

Pokud akce nastala příliš nazpátek v minulosti, například měl-li klient zrovna velké zpoždění, akci nezpracováváme. Důvodem mohou být již chybějící snapshoty. Na serveru si ukládáme předchozí stavy pouze v omezené míře, abychom šetřili výpočetní zdroje. Příliš starou akci tak ani nemusíme mít možnost zpracovat. Ale hlavním důvodem zahodit příliš zastaralou akci je možnost vzniku situace, kdy pro její řádné zpracování je nutné vzít zpět už jiné, potvrzené akce, a výrazně přepočítat dosavadní průběh hry. Z pohledu uživatele se může jednat o nepřípustně invazivní zákrok, jako například zpětné snížení již jednou dosáhnutého skóre.

Jedná se o citlivou problematiku, kdy cílem je dojít ke kompromisu, který umožní uživatelům s vyšším zpožděním hrát bez znatelných nevýhod, a zároveň nezpůsobovat problémy pro klienty s nízkou latencí ve snaze umožnit pomalým klientům férovou hru. Klienti s nižším zpožděním budou ale ve hrách v reálném čase vždy ve výhodě.

[2]Klient musí časovou značku přizpůsobit serveru, tedy přičíst k ní rozdíl svých a serverových hodin.

Kapitola 5

Rychlost zpracovávání simulace

Protože extrapolaci a interpolaci budeme demonstrovat na příkladu online hry v reálném čase, požadujeme, aby simulace neustále běžela tempem reálného světa. Pro zpracovávání a kontrolu rychlosti běhu simulace použijeme tzv. *game loop*.

5.1 Game loop

Game loop je nekonečná herní smyčka. Jejím úkolem je na základě uplynulého času kontrolovat rychlost simulace. V každé její iteraci může dojít ke zpracování uživatelského vstupu, úpravě herního stavu, nebo vykreslení [6]. Například na serveru, jehož úkolem je simulaci zprostředkovávat klientům, nic vykreslovat nemusíme.

Tick – konkrétní krok simulace.

Tickrate – frekvence zpracovávání simulace.

Timestep – vyjádření časového rozdílu mezi dvěma ticky.

5.2 Timestep typy

Timestep označuje časový rozdíl mezi dvěma kroky game loop. Podle výkonnosti hardware, na kterém program běží, a podle účelnosti, může být timestep simulace fixní, proměnný v čase nebo různě kombinovaný. [7]

5.2.1 Variable timestep

Mezi jednotlivými iteracemi smyčky s *variable timestep* může být libovolná prodleva. Smyčka zaznamenává časový rozdíl Δt mezi iteracemi, který se musí započítat do výpočtů během krokování. Jinak by simulace v závislosti na výkonnosti hardware běžela rozdílným tempem, protože výkonnější počítač vykoná za stejný časový úsek více iterací. Pro výpočet nové hodnoty souřadnice x se zrychlením $accX$ použijeme vzorec:

$$x = x + accX * \Delta t \tag{5.1}$$

Tato smyčka udržuje stálou rychlost simulace. Jestliže je hardware pomalejší, jednotlivé ticky budou od sebe více vzdálené na časové ose, ale kvůli Δt se herní simulace vždy posune dopředu stejnou rychlostí jako reálný čas.

Hlavní nevýhodou takové smyčky je nepřesnost výpočtů při dlouhé prodlevě mezi iteracemi. Uvažujme pohyb tělesa danou rychlostí s vysokým Δt. Těleso během jednoho kroku posuneme o znatelně větší vzdálenost. Během tohoto velkého posunu se může stát, že se předmět octne za možnou překážkou, aniž bychom detekovali kolizi. Pro zamezení podobného problému můžeme použít *fixed timestep*, nebo krokování označené jako *semi-fixed timestep*.

5.2.2 Fixed timestep

Smyčka s fixed timestep se snaží udržovat mezi ticky fixní časový rozdíl. Stav simulace se vždy posune o pevně daný úsek. Simulace se tak stává předvídatelnější, více deterministickou a je snazší rekonstruovat její přesný průběh. Pro výpočet pozice pohybujícího se tělesa použijeme následující vzorec.

$$x = x + accX \tag{5.2}$$

Do rovnice nemusíme zakomponovat Δt[1], protože z principu se bude jednat vždy o stejnou hodnotu.

Reálně je nemožné zajistit, aby stroj vykonal iterace vždy se skutečně stejným časovým rozdílem, pokaždé se vyskytne odchylka v řádu milisekund, nehledě na programově zadanou prodlevu. I přes tyto nepřesnosti vykonáme posun předmětů, jako by došlo k ideálnímu časování. Tento problém vynahradíme oddálením nebo přiblížením další iterace smyčky tak, aby průměrná hodnota rozdílu odpovídala očekávanému *tickrate*.

Jestliže očekáváme tickrate 10 – jednu iteraci každých 100 ms – a mezi posledními dvěma byl časový rozdíl 120 ms, stále posuneme předměty o vzdálenost, kterou by urazily za 100 ms. Další krok smyčky naplánujeme nikoliv na očekávaných 100 ms, ale 80 ms. Nechť se další krok reálně vykoná za 85 ms místo předpokládaných 80 ms. Předměty opět posuneme o vzdálenost, kterou by urazily za 100 ms a další krok smyčky naplánujeme za 95 ms, abychom zachovali celkový průměr rozdílu 100 ms.

Problém této smyčky je, že klademe nárok na hardware, aby byl schopný simulaci zpracovávat s rychlostí, na jakou je game loop nastavena. Pomalejší stroje, které nestíhají dokončit iteraci před začátkem další, budou postupně nabírat zvětšující se zpoždění. Důsledkem je simulace, která běží pomaleji než reálný čas.

Její výhodou je naopak předvídatelnost. V každém kroku simulace postupujeme o stejný časový úsek dopředu. Takto pomáhá předcházet možným problémům s fyzikálními výpočty, které se mohou vyskytovat u smyčky s *variable timestep*.

[1] Deltu je do rovnice stále vhodné zakomponovat, abychom umožnili konfigurovat tickrate simulace. Δt ale bude, narozdíl od variable timestep, konstantní po celou dobu trvání simulace.

5.2.3 Semi-fixed timestep

Tato implementace je variací *variable timestep*. Jestliže je prodleva mezi iteracemi příliš dlouhá, neuděláme v simulaci jeden dlouhý krok, ale rozdělíme jej na několik menších, které pustíme okamžitě po sobě v rámci jednoho *ticku*.

Hypoteticky, jestliže by prodleva mezi dvěma iteracemi trvala 200 ms, v *ticku* provedeme postupně například 4 změny herního stavu s $\Delta t = 50\ ms$.

Část II

Implementace

Kapitola 6
Obecný přehled

Před popisem konkrétních částí dema si projdeme jeho celkovou strukturu.

Doplňkovým materiálem pro popis struktury jednotlivých částí jsou tři diagramy obsažené v příloze B.

6.1 Charakteristika dema

Demo je online hra s implementovaným serverem i klientem. Jedná se o 2D střílečku s pohledem shora. Cílem je dosáhnout nejvyššího skóre ze všech účastníků. Každý hráč ovládá svou vlastní postavu a skóre získává vyřazováním postav ostatních hráčů ze hry. Hráči s vyřazenou postavou se po krátkém intervalu automaticky přidělí nová. Pro zpestření hra obsahuje několik map s různě rozmístěnými překážkami a možnost sbírat náhodně generované vybavení ze země.

6.2 Použité technologie

Použité technologie pro program klienta a serveru se částečně liší. Obě ale sdílí programovací jazyk JavaScript a využívají nástroje webpack – pro komprimaci zdrojového kódu – a npm, který se stará o správu závislostí a použitých knihoven, které jsou především:

Express - framework usnadňující, mimo jiné, práci s HTTP serverem

ws - implementace WebSocket serveru

earcut - knihovna pro triangulaci polygonů

collisions - knihovna pro jednoduchou detekci kolizí

6.2.1 Server

Server funguje na základě softwaru Node.js[1], který je interpret JavaScriptu. Pro chod serveru je využit framework *express*, jenž slouží pro zjednodu-

[1] Doporučená verze alespoň 12.x.x.

šení vytváření HTTP serveru včetně náležitostí, jako je posílání statického i generovaného obsahu klientům a routování.

6.2.2 Klient

Klient je webovou aplikací fungující ve webovém prohlížeči, který, podobně jako Node.js, interpretuje JavaScriptový kód. K vytvoření uživatelského rozhraní využívá HTML a CSS.

6.2.3 Zdůvodnění výběru

Podstatným kritériem pro výběr technologie byla jednoduchost jejího užití. JavaScript je relativně snadně pochopitelný, interpretovaný programovací jazyk, pro který existuje dostatek nástrojů a knihoven pro rychlé vytvoření základní struktury dema. Můžeme se tak rychle začít věnovat funkčním částem programu namísto složité přípravy prostředí. Webový prohlížeč například nabízí už v základu způsob pro přehrávání zvuků a vykreslování tvarů na obrazovku, knihovny pro Node.js se správnou šablonou okamžitě poskytují funkční základ serveru.

Obrovskou výhodou je, že klient i server využívá stejný programovací jazyk. Mohou tak mezi sebou sdílet části zdrojového kódu a programátorovi stačí znát jeden programovací jazyk.

6.3 Základní struktura zdrojového kódu

Přestože JavaScript sám o sobě nebyl vhodný pro objektově orientované programování, s jeho verzí ES6 se tomuto konceptu více přiblížil a umožnil strukturovat kód dema do jednotlivých tříd s pomocí nástroje webpack[2].

Třídy, každá ve vlastním souboru, jsou tematicky seskupené do složek (paralela s balíčky v jazyce Java). Nástroj webpack poté všechny sloučí do jednoho velkého souboru, tzv. *bundle*. Bundly se hodí, protože u webových prohlížečů je problematičtější pracovat s kódem napříč několika soubory, a ne všechny podporují moduly. Na server se zase snáze a rychleji nasazuje jeden velký než spousta malých.

6.4 Základní funkční struktura

Dvěma stěžejními a zcela samostatnými celky funkční struktury je herní simulace a poté skupina tříd zaštiťujících vykreslování simulace, její ozvučení a zprostředkování interakce s uživatelem.

Herní simulace neobsahuje žádné informace o tom, jak svůj stav vykreslit, ozvučit, jak zpracovat klientské vstupy, jak jim svůj stav rozeslat nebo ani jak

[2]Koncept tříd sice jde emulovat i ve starších verzích JavaScriptu, avšak ES6 hlavně využíváme pro koncept modulů.

rychle se má zpracovávat. Toto mají na starosti přilehlé třídy, jejichž implementaci lze nahrazovat. Důvodem pro tuto abstrakci je umožnění rozběhnutí simulace v různých prostředích. Hra je primárně konstruovaná tak, aby běžela na serveru, klient pouze přijímá snapshoty o jejím stavu a simulaci u sebe sám nemusí spouštět. Díky zmíněné abstrakci je ale možné ji u klienta snadno spustit v režimu jednoho hráče, kdy si simulace a třídy zpracující audiovizuální části vyměňují data napřímo v programu klienta, namísto transformace dat do formy vhodné pro síťové protokoly na přenos dat a následné rekonstrukce na straně příjemce.

Základním stavebním kamenem simulace je její schopnost provést iteraci a posunout svůj stav o specifikovaný časový úsek. Jí nadřazená třída funguje jako libovolná game loop řídící rychlost běhu.

Simulace na konci každé iterace veřejně poskytne snapshot svého stavu a zároveň vyvolá různé události, které ke stavu vedly. Třída pozorující její průběh poté na události reaguje. Její hlavní funkcí je rozvrhnout, jaké informace ze zachycených událostí je nutné předat klientům. Ty poté přepošle třídě starající se o distribuci stavu, která obstará přenos dat způsobem vhodným pro prostředí, ve kterém hra běží. Audiovizuální třídy u klienta poté přijaté informace zpracují a aktualizují prezentaci hry.

Kapitola 7

Struktura herní simulace

Třída reprezentující herní simulaci (dále jen hra) obsahuje seznam všech aktivních entit, informace o hráčích, různé pomocné metody a hlavně metodu pro úpravu herního stavu s parametrem Δt, o který se posune vpřed.

7.1 Krokování

Požadavek na provedení kroku o délce Δt posílá hře nadřazená herní smyčka, která si na hru uchovává referenci. Pro zpomalení nebo zrychlení simulace postačuje, aby game loop jako parametr Δt neposílala skutečně uběhlý čas mezi svými iteracemi, ale kratší/delší úsek, podle potřeby.

V každém ticku hry se provedou tyto kroky v daném pořadí:

1. Provedení naplánovaných akcí z předchozích iterací.
2. Aktualizace všech[1] entit.
3. Aktualizace systému pro detekci kolizí, jejich detekování a vyhodnocení.
4. Odstranění označených entit.
5. Posun herních stopek.
6. Vygenerování nového snapshotu simulace.

7.2 Entity

Každý objekt, který udržujeme uvnitř stavu hry, považujeme za entitu. Druhů entit hra uznává několik, podle nutnosti pořadí, v jakém by se měli zpracovat jejich aktualizace a případně, zda se tak má stát během každého ticku, nebo jestli toto zajišťujeme jiným způsobem.

Mezi entity řadíme „fyzicky“ přítomné objekty – hráčské postavy, projektily, překážky, bonusové vybavení – ale i skryté bez viditelné reprezentace, jakým je manažer generování vybavení.

[1] Kromě specificky vyčleněných.

7.3 Komponenty

Entity v sobě obsahují seznam svých *komponent*. Komponenta v rámci hry je znovupoužitelný balíček metod a vlastností, které entitám přidávají funkcionalitu. Entity komponenty využívat nemusí a mohou všechny své metody a nutné vlastnosti obsahovat samy v sobě. Avšak když jich několik sdílí společnou vlastnost, měli bychom ji vyčlenit jako samostatnou komponentu, aby se stejná funkcionalita nemusela implementovat na několika místech, nebo aby se předešlo řetězení dědění.

7.4 Objekt hráče

Datová struktura reprezentující hráče (v sekci dále jen hráč) není entitou a není začleněna v jejich seznamu, protože při začátku každého kola se hra kompletně vynuluje, včetně seznamu entit. Hráči mezi koly ale setrvávají.

Všichni jsou ve hře reprezentováni vlastní entitou (postavou). Na začátku každého kola hra zkontroluje aktuální hráče, připraví jim nové postavy a vynuluje skóre. Hráči se mohou do hry zapojit i kdykoli během rozehraného kola. Z perzistentního stavu hry jsou odebráni pouze na konci kola a pouze pokud s nimi spřažené spojení v té době není aktivní. To znamená, že se může během kola odpojit a zase vrátit, aniž by přišel o své skóre.

7.5 Řešení kolizí

Správné vyhodnocování kolizí není triviálním úkolem a uspokojivá a optimalizovaná řešení není snadné vytvořit. Proto hra k tomuto účelu využívá existující knihovnu *collisions*. Nejedná se o skutečný fyzikální systém, ale detekci intersekcí konvexních polygonů a kruhů. Složité konkávní polygony je pro správné detekování kolizí v systému knihovny prvně nutné rozdělit na polygony konvexní. Proces triangulace je k takové transformaci ideální, protože trojúhelníky jsou vždy konvexním tvarem. Tuto funkcionalitu obstarává knihovna *earcut*.

Polygonálními překážkami jsou zdi, jejichž triangulace proběhne vždy při načtení konkrétní herní mapy. Ostatní entity využívají pro zjednodušení kruhová kolizní tělesa.

7.6 Interakce s uživateli

Zprávy reprezentující uživatelské akce se zpracovávají zvlášť ve třídě mimo objekt hry, přestože přímo ovlivňují její stav. Klient uživatelské vstupy, hned jak je zaznamená, zasílá na server, kde po přijetí dochází k okamžitému uplatnění jejich důsledků.

Kvůli zjednodušení zpracovávání uživatelských akcí každá akce trvá po dobu celého ticku. Dorazí-li v mezičase zpracování dvou kroků protichůdné

akce, dojde k jejich přepisování. Z více protichůdných akcí se projeví vždy ta, která byla přijata jako poslední.

Příklad: nechť je tickrate simulace 20 (jeden tick každých 50 ms). Kdyby dorazil požadavek „jdi doleva“ a „jdi doprava“ 40 ms, respektive 20 ms před tickem, vyhraje poslední přijatý – „jdi doprava“. Hráč se posune, jako by šel doprava, a to po dobu celého ticku, tedy 50 ms.

Důsledek každé akce je kromě latence, a případně interpolačního zpoždění, také zpomalený navíc až o maximální délku prodlevy mezi ticky. Jedná se o podobný princip zpoždění jako u přijímání snapshotů klientem, viz popis aktuálnosti stavů pod obrázkem 2.1 Demonstrace problému periodického překreslování posledního přijatého stavu

7.7 Systém událostí

Události se spravují z jednoho bodu uvnitř simulace. Naslouchání událostem nebo jejich propagace se vždy provádí centrálně. Reagovat na události konkrétní entity vyžaduje registraci posluchače na všechny relevantní události a z připojených informací v nich zjistit, zdali vychází z pro nás zajímavé entity.

7.8 Časovač úkolů

Úkoly, které potřebujeme vykonat s prodlevou se neukládají do asynchronních volání se zpožděním[2], ale registrují se do speciální třídy uvnitř hry, do které vložíme dvojici callback a časovou značku, kdy se má provést. Časovač nepracuje s aktuálním systémovým časem, ale stopkami uvnitř herní simulace, které se posouvají pouze s krokováním o hodnotu Δt. V každém kroku se zkontroluje fronta úkolů, kdy se spustí všechny s časovou značkou mladší, než je aktuální hodnota stopek.

Časové odchylky v game loop nemají vliv na tyto stopky a prodleva uvnitř simulace funguje konzistentně[3].

[2] Například nativní metoda `setTimeout`

[3] Kdyby hra běžela poloviční tempem a používali bychom metodu `setTimeout`, museli bychom jako parametr zpoždění uvádět dvojnásobnou hodnotu.

Kapitola 8
Program serveru

Smyslem programu serveru je udržovat u sebe instanci hry v nekonečné smyčce a distribuovat její stav všem připojeným klientům.

8.1 Komunikace mezi klientem a serverem

Na stroji, který má fungovat jako server, program nejprve spustí HTTP server a na něj napojí WebSocket server. To umožňuje spustit oba servery na stejném portu, každý ale obsluhuje jiný protokol. Server, se kterým je třeba navázat komunikaci se určí protokolem v URL adrese.

Demo využívá protokol HTTP pouze pro distribuci programu klienta uživatelům. Pro frekventované zasílání dat ze serveru klientům a naopak HTTP nestačí. Využijeme proto WebSocket server. Ten mezi nimi zprostředkuje udržovaný obousměrný komunikační kanál.

Komunikace probíhá pomocí JavaScriptových objektů generovaných přes factory methods. Každý obsahuje svůj unikátní identifikátor a relevantní přiložená data. Pro potřeby síťového přenosu je během odeslání převeden do formátu JSON a po přijetí protistranou rekonstruován.

Pro snazší správu aktuálních WebSocket připojení je WebSocket server obalený v dekorátoru, který usnadňuje rozesílání zpráv a umožňuje je ukládat do fronty a frontu poté odeslat jako jeden balíček, aby se minimalizoval počet zpráv klientům. Třída se také stará o mapování objektů reprezentujících WebSocket spojení na objekt hráče uvnitř hry a ukládá si ke spojením i další data, jako například, zdali má spojení administrátorská privilegia.

Při navázání komunikace klient nejdříve pošle na server zprávu s žádostí o přístup do hry a připojí svoje jméno. Server jméno zpracuje [1] a pokud je jméno unikátní, vytvoří ve hře nový objekt hráče. Spojení poté namapuje na objekt hráče, aby šlo příchozí klientské akce spojit s konkrétním hráčem. Navíc pak server také začne pravidelně posílat informace o stavu hry klientovi.

[1] Zajistí, aby jméno neobsahovalo nepovolené znaky, bylo v rámci dané délky, atd.

8.2 Struktura zpráv o herním stavu

Stav hry je pro potřeby vykreslení složitý a obsahuje mnoho zbytečných parametrů. Na klienty proudí pouze zjednodušená reprezentace snapshotu, který obsahuje informace o typu a pozici graficky reprezentovatelných objektů. Těmi jsou postavy hráčů, projektily a bonusové vybavení. Snapshot vždy obsahuje absolutní stav, tedy včetně informací o objektech, které se nijak nezměnily.

Protože klient nezná herní pravidla, ale pouze slepě vykresluje podle informací ve snapshotu, nechápe koncept, jako je zásah projektilem, výstřel nebo odstranění herního objektu ze hry. Ve snapshotu se pouze náhle objeví nebo zmizí informace o herní entitě. Zprávy o událostech vedoucích ke změnám v herním stavu, u kterých požadujeme, aby na ně reagoval i klient, jsou posílané zvlášť. Ukázkou budiž zpráva o zásahu postavy hráče s novým stavem životů nebo požadavek na přehrání zvuku specifické zbraně v reakci na událost výstřelu.

Na základě těchto událostí například operuje ukazatel životů hráče. Informace o životech nedostává klient s každým snapshotem, ale pouze pokud se v průběhu hry náhle změnily. Zamezujeme tak zbytečnému překreslování ukazatele, když nedochází k pro něj relevantním změnám.

Obdobně funguje tabulka se skóre. Při prvním zapojení hráče do hry dostane o skóre kompletní informace. Poté už jenom zprávy o změnách.

8.3 Game loop

Kvůli svým vlastnostem se pro udržení hry na serveru v chodu velmi hodí game loop s fixed timestep. Klienty chceme zásobovat novými stavy periodicky a předvídatelně. Na serveru se ve výchozím nastavení simulace zpracovává 50krát do sekundy. Daná frekvence poskytuje dostatečnou kvalitu fyzikálního modelu a zpracovávání klientských událostí[2]. Server klientům posílá nový stav vždy po každém dokončení iterace.

Hra je připravena ke startu okamžitě po zapnutí serveru, ale game loop se spustí až po úspěšném navázání spojení s prvním hráčem.

[2]Při tickrate 30 během testování občas proletěly projektily skrze tenké překážky nebo se nesprávně registrovaly zásahy hráčů.

Kapitola 9
Program klienta

Úkolem klienta je zprostředkovat interakci uživatele s hrou a její stav vykreslit na obrazovku a ozvučit. Klient podporuje hru jednoho hráče, kdy herní simulace běží lokálně, nebo hru více hráčů, kdy simulaci zprostředkovává server.

Klienta uživatel spustí zadáním URL adresy serveru do prohlížeče[1].

9.1 Obecná struktura

Klient je postaven na návrhovém vzoru stavů. Primárními dvěma stavy je stav hlavního menu a stav poskytující prezentaci hry. Každý stav funguje zcela autonomně a spolupracuje s manažerem stavů a manažerem uživatelských vstupů. Detekování interakce uživatele zajišťuje centrální manažer uživatelských vstupů, který každou registrovanou akci přepošle do právě aktivního stavu. Stav poskytne manažeru zpětnou vazbu, zdali je na akci nutné reagovat, a pokud ano, tak jak. Pokud je akce z pohledu stavu nedůležitá, reakce na ni se přenechává webovému prohlížeči.

Mimo tyto dva stavy je pak zvlášť vyčleněn editor herních map přístupný na URL adrese serveru rozšířenou o „/editor“. Přestože je součástí webové aplikace, není hlavní náplní práce a dále mu nebude věnována pozornost.

9.1.1 Stav: hlavní menu

Hlavní menu je velice jednoduché, obsahuje krátký popis o ovládání hry, základní nastavení a možnost připojit se do online zápasu nebo spustit režim jednoho hráče.

9.1.2 Stav: režim hry

Do stavu prezentující hru se přechází s parametrem specifikujícím, jestli se jedná o lokální hru nebo online.

V online režimu se klient pokusí připojit k WebSocketu na specifikované URL adrese. Výchozí nastavení URL je adresa serveru, který poskytl program

[1] I pro hru v režimu jednoho hráče musí být uživatel připojený k internetu alespoň, dokud mu server nezašle program klienta.

klienta. Pokud se podaří úspěšně navázat spojení, klient začne vykreslovat a přijímat snapshoty ze serveru a propagovat uživatelské akce na server. Jestliže během připojení nastane chyba, klient zobrazí chybovou hlášku a automaticky se přepne zpět do stavu hlavního menu.

Režim jednoho hráče kromě procesu vykreslování také spustí lokální instanci hry. Veškerá komunikace mezi herní simulací a audiovizuálními třídami probíhá pomocí stejných zpráv jako v případě online hry, ale vynechává se krok převedení zpráv do JSON a pak zpětné transformaci, která probíhá při odesílání, respektive přijímání zpráv přes WebSocket.

9.2 Gameloop klienta

Klient pro hru v režimu jednoho hráče a více hráčů využívá lehce rozdílné game loops. V obou figuruje nativní JavaScript funkce *requestAnimationFrame(callback)*. Pokud rekurzivně odkazuje sama na sebe, automaticky zařídí, aby metoda poskytnutá v callback byla volána podle výkonnosti stroje, kde prohlížeč běží, s maximální frekvencí 60 volání za sekundu [8].

Funkce se perfektně hodí k implementaci game loop s variable time step, protože sama zařídí, aby se na výkonném stroji častěji krokovalo a tím se zvýšila frekvence vykreslování, případně preciznost výpočtů v simulaci v lokální hře.

9.2.1 Režim více hráčů

Protože při hraní online běží simulace a všechny výpočty s ní spojené na serveru, nemusíme se obávat problémů s fyzikou při dlouhých prodlevách mezi iteracemi game loop. Klientovi v tomto případě stačí jenom vykreslovat příchozí snapshoty ze serveru, proto ani nemusí znát uplynulý čas mezi iteracemi. Vykreslování mu totiž stačí jenom čas během nynější iterace a časové značky posledních snapshotů. Z toho důvodu klient používá obyčejný variable timestep implementovaný funkcí *requestAnimationFrame(callback)*. Uvnitř callback metody pouze spouštíme překreslování.

9.2.2 Režim jednoho hráče

V režimu jednoho hráče uživateli s výkonným zařízením díky variable timestep umožňujeme simulaci krokovat častěji s kratšími časovými úseky. Pokud by uživatel měl nevýkonný hardware, problémům s fyzikou kvůli dlouhým prodlevám klient zamezuje tím, že využije semi-fixed timestep principu. Uvnitř callback metody ve funkci *requestAnimationFrame(callback)*, nejdříve posouváme herní stav v krocích o maximálně 20 ms[2] a až pak překreslíme herní stav.

[2]Emulujeme tickrate 50, při kterém jsou minimální problémy s fyzikou.

9.3 Vykreslování

Herní stav se zobrazuje pomocí HTML elementu *plátna* (anglicky canvas). Pro vykreslování používáme celkově dvě rozdílná plátna kvůli optimalizaci. První obsahuje statické objekty, které zpravidla stačí vykreslit pouze jednou při prvním načtení herní plochy. K překreslení dochází výjimečně v případě změny velikosti okna prohlížeče. Druhé, v popředí prvního se statickým obsahem, vykresluje při každé iteraci game loop dynamické objekty. Protože každý vykreslený tvar je zátěží pro hardware, rozdělením a pravidelným zpracováváním pouze druhého plátna s pohyblivými objekty omezujeme počet překreslených tvarů a šetříme výpočetní zdroje. Překreslování probíhá v každém ticku game loop klienta, nehledě na použitou strategii.

9.3.1 Princip strategií vykreslování

V hlavním menu si uživatel může vybrat jeden ze tří způsobů – strategií – vykreslování. Hlavní třída starající se o vizuální prezentaci (dále renderer) pouze předává instrukce plátnům, jak objekty zobrazit a kdy (uživatelské rozhraní jsou samostatné HTML elementy v popředí pláten, které spravuje herní stav). Ohledně otázky, *kde* objekty zobrazit, spoléhá na odpověď jedné ze zvolených strategií.

Renderer prvně předá pokyn strategii, aby se připravila, a vyžádá si od ní seznam všech entit, se kterými může pracovat. Pak se pro každou entitu ze seznamu zvlášť ptá, jestli má smysl se ji pokusit vykreslit. Pokud ano, až poté se začne dotazovat na hodnoty konkrétních vlastností. Především se jedná o pozici ve světě.

V rámci dotazu renderer posílá aktivní strategii metodu, která má za úkol dohledat a vrátit uloženou hodnotu z poskytnutého snapshotu. Touto metodou se strategie pokusí získat hodnoty ze snapshotů, které jsou relevantní pro její vnitřní implementaci, a poté analýzou a zpracováním získaných hodnot vrátit rendereru finální výsledek.

Metoda přístupu k hodnotě vlastnosti v sobě musí obsahovat ošetření situací, kdy daná vlastnost ve snapshotu neexistuje. Pokud tento případ nastane, metoda musí vrátit hodnotu *false* místo čísla[3]. Strategie pak na vrácenou hodnotu *false* z metody reaguje jako na chybový stav a dostává prostor pro použití záložních řešení. K tomuto stavu dojde pouze, počítá-li strategie s tím, že může nastat, a je připravena na něj reagovat. Jinak by zamezila jakýmkoliv konkrétním dotazům negativní odpovědí na předchozí otázku rendereru, zdali má smysl entity vykreslovat (viz předchozí odstavec). Konkrétně tento mechanismus využívá například extrapolační strategie pro entity, pro které zatím nemá informace alespoň ze dvou stavů.

[3]Tento způsob nám umožňuje JavaScript díky své netypovosti. Zároveň „číslo“ je v JavaScriptu skutečný datový typ.

9.3.2 Strategie poslední snapshot

Nejprimitivnější je strategie posledního snapshotu, která na každý dotaz rendereru odpoví přesným kopírováním hodnoty v posledním přijatém snapshotu. Efektivní frekvence vykreslování je nanejvýš rovna té, kterou snapshoty poskytuje server. Uživatelé navíc vidí herní stav zastaralý o polovinu hodnoty své latence a doby, než klientská game loop dá pokyn k překreslení. Jedná se o problém ilustrovaný obrázkem 2.1 Demonstrace problému periodického překreslování posledního přijatého stavu.

9.3.3 Strategie extrapolace

Při extrapolační strategii pro vykreslení zvažujeme pouze entity obsažené v posledním přijatém snapshotu. Ty má v rámci této strategie vždy smysl vykreslit, protože předpokládáme jejich existenci i v příštím snapshotu a chceme je zobrazit co nejdříve, tedy i před dostatkem informací k extrapolaci.

Je-li záznam o entitě i v předposledním snapshotu, můžeme její pozici extrapolovat a přiblížit se tak jejímu skutečnému umístění na serveru. Když záznam chybí, vykreslíme ji technikou posledního snapshotu, dokud neobdržíme další informace o stavu hry. Takto sice každý objekt nejdříve zobrazujeme staticky a po doplnění informací náhle poskočí vpřed (a až pak se začne pohybovat plynule). Zabráníme tak ale případům, kdy by entity existující pouze po dobu jednoho ticku uživatel nikdy nespatřil.

Během přípravy si strategie zjistí, jestli jsou celkově k dispozici alespoň dva snapshoty (jedná se především o stav při prvním připojení do hry). Pokud ne, automaticky přepne do režimu poslední snapshot a o extrapolaci se nepokouší, jinak si připraví parametr t podle rovnice 3.1. Abychom správně interpretovali přiložené časové značky, za přítomnost (čas T) považujeme čas synchronizovaný s časem serveru.

Při sestavení odpovědi na dotaz rendereru se strategie v režimu extrapolace nejdříve podívá na hodnotu vlastnosti v posledním snapshotu a pak ji zkusí najít i v předposledním. Pokud v něm přístupová metoda vrátí *false*, znamená to, že neobsahuje informaci o dané vlastnosti a tudíž nelze extrapolovat, protože nemáme dostatek dat – jako odpověď vracíme přímo hodnotu z posledního snapshotu. Jestliže ale metoda vrátí číslo, dosadíme ho do rovnice 3.2 společně s ostatními parametry a vrátíme její výsledek.

Pro názornost si rovnice spojené s extrapolací opět uvedeme. Spodní index 0 označuje hodnoty předposledního přijatého snapshotu a spodní index 1 posledního.

$$t = \frac{(T - T_1)}{(T_1 - T_0)} \qquad \text{(duplikát 3.1)}$$

Parametr t popisuje, kolikrát do extrapolace promítáme velikost posunu.

$$x_e = (x_1 - x_0) * t + x_1 \qquad \text{(duplikát 3.2)}$$

9.3.4 Strategie interpolace

Interpolační strategie dynamicky přepíná mezi interpolací a extrapolací v závislosti na dostupných informacích.

Protože interpolace pracuje s daty v minulosti, jako přítomnost T považujeme synchronizovaný čas se serverem, od kterého odečteme délku interpolačního zpoždění.

Při přípravě se strategie nejdříve pokusí identifikovat snapshoty, které svými časovými značkami ohraničují čas T. Jestliže takové dva neexistují, kontrolu přebírá funkcionalita extrapolační strategie. V opačném případě pracujeme v režimu interpolace, a to pouze s entitami přítomnými v obou snapshotech. Přístupová metoda tedy nemůže dojít do chybového stavu. V odpovědi rendereru vrátíme výsledek získaný dosazením parametrů do rovnice 4.2, kdy t získáme z rovnice 4.1.

Pro názornost si zde rovnice spojené s interpolací připomeneme. Spodní index 0 označuje hodnoty snapshotu na časové ose nalevo od T a spodní index 1 napravo.

$$t = \frac{(T_1 - T)}{(T_1 - T_0)} \qquad \text{(duplikát 4.1)}$$

Parametr t relativně popisuje, jaký čas mezi T_0 a T_1 nás zajímá.

$$x_i = (x_1 - x_0) * t + x_0 \qquad \text{(duplikát 4.2)}$$

9.4 Umělé navýšení latence

Při testování účinnosti extrapolace a interpolace se hodí vytvářet testovací prostředí s libovolnou mírou latence. Pro tento účel klient umožňuje úmyslně zpozdit odesílání a přijímání zpráv. Pokud si uživatel přeje tuto funkci využít, místo výchozí implementace WebSocketu v prohlížeči klient využije speciální třídy (proxy), která WebSocket obalí. Ta implementuje jeho rozhraní a volání metod přesně mapuje na obalený WebSocket s výjimkou metody pro odeslání zprávy nebo jejího zpracování, do kterých vloží konfigurovatelné zpoždění. To je realizované nativní JavaScript metodou *setTimeout*.

Nakonfigurované zpoždění se v latenci projeví dvojnásobně, protože se uplatní dvakrát. Jednou při odesílání zpráv a podruhé při přijetí.

Část III

Testování a závěr

Kapitola 10

Výstupy uživatelského testování

Hlavním cílem uživatelského testování bylo porovnat jednotlivé vykreslovací techniky a vyhodnotit, které uživatelé považují za nejvíce přívětivé. Sekundárně se hodnotila kvalita zpracování vstupů.

Testování probíhalo online a zúčastnili se jej 4 testeři. Za běžných podmínek se jejich latence pohybovala okolo 20 ms. Pro testování kvality vykreslovacích technik v závislosti na rostoucí latenci testeři využili funkce umělé zvýšení latence klienta. Frekvence vykreslování ani u jednoho testera neklesla pod očekávaných 60 snímků za sekundu, viz sekce 9.2 Gameloop klienta.

Celkově proběhly dva testy. Jednou s tickrate serveru nastaveným na hodnotu 30, podruhé 50. Pro porovnání, herní společnost Valve používá pro své online hry, kde soupeří lidé vzájemně proti sobě, tickrate 66 [5].

10.1 Kvalita zpracování uživatelských vstupů

Při obou testech s rozdílným tickrate byly nepřesnosti způsobené zjednodušením, popsaným v sekci 7.6 Interakce s uživateli, neznatelné.

10.2 Kvalita vykreslovacích technik

10.2.1 Poslední snapshot

Při tickrate 30 se technika posledního snapshotu neosvědčila, obraz nebyl dostatečně plynulý pro pohodlné hraní. Pro tickrate 50 technika působila na uživatele mnohem přívětivěji, byť stále trhaně.

10.2.2 Extrapolace

Extrapolace vždy zajistila plynulý průběh vykreslování, ale s vyšším tickrate a při nízké latenci, do 30 ms, byly problémy s viditelnými artefakty při extrapolaci znatelné v menší míře. S narůstající latencí tickrate nehrál žádnou znatelnou roli a problém s náhlými pohyby při korekci chyb v predikci působil velice rušivě. Při latenci nad 80 ms byly chyby extrémně znatelné – způsobovaly výraznou degradaci uživatelského zážitku až k hranici nehratelnosti.

10.2.3 Interpolace

Na rozdíl od extrapolace zajišťuje interpolace kvalitní a plynulé zobrazení při nízké i velké latenci. Největším jejím problémem je delší prodleva, než uživatel spatří důsledky svých akcí.

10.3 Závěr

Při nízké latenci testeři preferovali extrapolaci nebo interpolaci. Záleželo na osobní preferenci testovaných. Přestože se všichni shodli, že problémy s nápravou chyb v predikci způsobené extrapolací byly minimální, někteří přesto preferovali interpolaci, protože problém s interpolačním zpožděním vnímali jako menší zlo oproti občasným extrapolačním chybám, byť málo výrazným. Technika posledního snapshotu zaostávala vlivem trhaného obrazu.

Se zvyšující se latencí testeři postupně od extrapolace upouštěli ve prospěch interpolace kvůli zvětšujícím se chybám. Při zpoždění 100 ms jeden dokonce preferoval i trhaný obraz posledního snapshotu oproti interpolaci – vyměnil plynulost obrazu za rychlejší odezvu[1]. Nehledě na použitou techniku, při vysoké latenci kvalitu uživatelského zážitku nejvíce snižovala především délka odezvy na akce uživatele. Jakmile byla prodleva znatelná (latence nad 100 ms nebo interpolace), její skutečná délka neměla až tak velký vliv.

Podle výsledků testování vyznívá interpolace jako nejvšestrannější technika, která nachází uplatnění v nejvíce situacích. Dá se předpokládat, že odstraněním její největší nevýhody – přidaným znatelným zpožděním na akce uživatele – by se jednalo z testovaných technik o nejlepší variantu.

[1] V tomto případě je ale nutné mít na paměti, že při technice posledního snapshotu jsou uživatelé s nižší latencí zvýhodnění kromě rychlejší odezvy na své akce i prezentací světa s menším zpožděním. Úkolem práce je ale zajistit, aby všichni uživatelé viděli svět ve stejném stavu.

Kapitola 11

Prostor pro zlepšení

Některé další techniky, které lze ve hrách více hráčů uplatnit jsou popsané v této kapitole. Úmyslně byly vyňaty z části teoretické přípravy, protože nejsou natolik důležité pro zkvalitnění uživatelského zážitku, ani implementované v ukázkové hře.

11.1 Predikce na straně klienta

11.1.1 Princip

Pro ošetření problému s prodlením viditelné reakce hry na akce uživatele serverem v důsledku latence (a případně interpolace) lze u klienta provozovat jeho vlastní herní simulaci souběžně s tou na serveru, se kterou se pravidelně synchronizuje. Ve své vlastní simulaci pak klient může důsledky akce předpovědět a okamžitě poskytnout zpětnou vazbu uživateli během čekání na vyjádření serveru. Zpráva od něj v nejlepším případě pouze potvrdí lokální předpověď [5].

11.1.2 Chyby v predikci

Klient ovšem nemá k dispozici přesný stav hry v době predikce kvůli latence. Chybějící nebo zastaralá data mohou vést ke kritické chybě v předpovědi a rozporu s odpovědí serveru. Protože je server autoritativní, jeho výsledek přepíše lokální predikci.

Náprava chyby může být uživateli viditelná až matoucí a poškozující herní zážitek. Projevit se může například okamžitým přemístěním pozice hráče, místo obvykle plynulého pohybu, podobně jako u interpolace. Aby oprava mezi chybným lokálním stavem a skutečným nově příchozím ze serveru nebyla příliš patrná, nenapravíme ji okamžitě, ale rozprostřeme ji do několika snímků. Pokud se lokální a serverový stav liší jenom drobně, předejdeme tak vzniku viditelných artefaktů.

Pokud na mylně předpovězeném úspěchu akce závisely jiné skutečnosti, je třeba opravit a ošetřit celou kaskádu událostí. Klient si musí pamatovat všechny své předpovědi na akce, které ještě nebyly zpracované serverem. Pokud je specifická akce serverem zpracovaná se stejným výsledkem, není

nutné si ji dále pamatovat. Jestliže je server a klient ohledně výsledku akce v rozporu, klient musí opravit svůj stav podle výsledku ze serveru a znovu u sebe aplikovat všechny uložené, serverem dosud nezpracované, akce.

11.1.3 Rozsah predikce

Ne všechny akce je bezpečné předvídat na klientovi, pokud by jejich napravení bylo pracné nebo vyžadovalo velký zásah do průběhu simulace. Například v kontextu online střílečky je vhodné predikovat výstřely, kvůli okamžité zpětné vazby na vstupy uživatele, včetně zásahu protivníků podle informací dostupných klientovi, aby projektily nemizely až po viditelné prodlevě po zásahu protivníka, než server zásah potvrdí. Ovšem odstranění protivníka při ztrátě všech životů ze hry by bylo bezpečnější učinit až na základě pokynu serveru [9]. Chybně registrovaný nebo ušlý zásah soupeře na straně klienta je v rámci kompromisu klientské přívětivosti a okamžité odezvy bezpečné předvídat. Ovšem oživení padlého soupeře nebo odečtení predikovaného zisku skóre silně zasahuje do průběhu hry a snižuje uživatelský komfort.

11.2 Optimalizace objemu síťového provozu

Baseline snapshot – snapshot, jenž je východiskem pro daný delta snapshot.

Delta snapshot – obsahuje pouze seznam změn vůči svému baseline snapshotu.

11.2.1 Motivace

Snížením objemu posílaných dat můžeme snížit latenci klientů a tím vylepšit uživatelský zážitek. S méně příchozími daty je obvykle také spojeno méně práce nutné k jejich zpracování, šetříme tak i zdroje hardware. V extrémním případě, pokud by objem posílaných dat byl větší, než je v rámci klientova připojení možné přijímat, by nastaly nepředvídatelné problémy s přijímanými daty a hru by nemuselo být možné vůbec hrát.

11.2.2 Balíčkování

Protože každá zpráva po síti cestuje zabalená v rámci paketu, který kromě samotných posílaných dat vždy obsahuje navíc i informace neužitečné pro aplikaci, je vždy lepší poslat stejné množství užitečných dat v co nejmenším počtu paketů. Z tohoto důvodu není vhodné odesílat každou klientskou akci nebo zprávy o stavu hry ze serveru zvlášť, jakmile nastanou, ale slučovat je do větších celků a ty poslat najednou. Každou zprávu ale musíme označit časovou značkou. Po přijetí balíčku je tak možné přesně zrekonstruovat pořadí zpráv a zakomponovat do zpracování časové rozdíly.

Demo popsanou techniku implementuje jenom částečně, a to pouze na zprávy odesílané serverem. Rekonstrukce sekvence klientských akcí server nepodporuje, klient své zprávy ani nebalíčkuje.

11.2.3 Delta snapshot

Velmi efektivní technikou snižující objem posílaných dat je využít tzv. delta snapshot, který vyjadřuje změny vůči výchozímu (baseline) snapshotu. Pokud je v další iteraci simulace ve stejném stavu jako v předchozím, není třeba posílat žádná data. Jestliže se změnil stav pouze některých objektů vůči baseline snapshotu, posíláme informace pouze o jejich změněných vlastnostech.

Nevýhodou techniky je ztížení zajištění synchronizace. Pokud se stav u klienta v jednu chvíli desynchronizuje, nedetekovaná chyba může postupně desynchronizaci nadále prohlubovat, obzvláště využívá-li se predikce na straně klienta.

V primitivních implementacích bez využití delta snapshots, kdy vždy posíláme kompletní absolutní stav, desynchronizace nemůže vydržet déle než do příchodu stavu nového.

Demo obsahuje náznak techniky delta snapshot, a to pouze při aktualizaci uživatelského rozhraní. Nejedná se ovšem o obecné a robustní řešení, ale implementačně specifické reakce na konkrétní zprávy o událostech ve hře.

11.2.4 Vlastní protokol a formát zpráv

Vyspělou formou optimalizace je příprava vlastního protokolu, formátu zpráv nebo interpretace jednotlivých dat. Například místo dosavadního označení jednotlivých položek dat klíčem je možné používat fixní pořadí položek, kdy je každému pořadí určený konkrétní význam.

Velké úspory dosáhneme i změnou formátu zpráv během přenosu z aktuálního otevřeného textu ve formátu JSON na binární data. Zpracování takových dat ale vyžaduje mnohem vyšší úsilí namísto používání jednoduchého JSON API běžného pro JavaScript.

Jako hodně specifický příklad optimalizace lze uvést možnost zkrácení zápisu úhlu v kvaternionu. [1]. Místo obvyklých 4 složek kvaternionu můžeme posílat pouze 3, protože celková hodnota všech 4 složek vždy musí dát jako výsledek jedné specifické operace vrátit číslo 1. Čtvrtou složku kvaternionu tak nemusíme posílat, příjemce si ji dopočítá [10].

[1] Kvaternion je datová struktura, která reprezentuje úhel v 3D prostoru pomocí 4 složek.

Kapitola 12

Závěr a zhodnocení

Během bakalářské práce jsem přímo řešil problémy, s jakými se vývojáři videoher mohou potýkat, ať už jde o hru jako takovou, nebo přidružený kód umožňující hraní online. Přestože uživatelské testování bylo svým způsobem okrajovou záležitostí vzhledem k nutnému času pro jeho dokončení v porovnání s implementací programu, jednalo se o pro mě zajímavou činnost a bavilo mě rozebírat a diskutovat postoje jednotlivých testerů.

Kromě získání nových znalostí o technikách a principech popsaných v práci, z nichž jsem i několik implementoval, jsem si procvičil návrh architektury programu od samého počátku, vzájemné propojování jednotlivých funkčních částí a použití některých návrhových vzorů. Při zprovozňování režimu jednoho a více hráčů jsem skutečně plně ocenil význam abstrakce a výhody oddělování implementací od rozhraní již od samého začátku projektu.

Z prostředí webu jsem si osvojil práci s HTML plátnem (ve 2D kontextu) a WebSocketem, prohloubil si znalosti o JavaScriptu samotném i práci s nástroji spojenými s vývojem webových aplikací jako je npm, kdy jsem se více do hloubky podíval na webpack a jsdoc, včetně jeho alternativy esdoc. Do jisté míry jsem si i zkusil vzdálenou administraci Linux serveru během nasazování dema.

Nahlédl jsem i do knihovny pro psaní testů, bohužel jsem ji nevěnoval dostatek pozornosti a testy dostatečně nevyužíval, čehož jsem později litoval při refaktoringu, kdy se do té doby funkční systémy zhroutily a oprava chyb trvala mnohem déle, než musela.

Byť se to nemusí zdát, cennou zkušeností byla i práce s externími knihovnami a jejich integrace do vlastní implementace i pročítání dokumentace. Za zmínku určitě také stojí práce s interpretem Node.js, který jsem předtím znal pouze velice povrchně.

S výsledkem bakalářské práce jsem osobně spokojený a velice si cením nabytých znalostí ze všech oblastí, kterými jsem se zabýval.

V budoucnu bych určitě rád implementoval kompenzaci zpoždění a predikci na straně klienta, se kterými by se demo stalo skutečným základem pro minimalistickou online hru s responzivním ovládáním.

Literatura

[1] Logical Increments. *Information About Frame Rate* [online]. Poslední aktualizace 3.12.2019 [cit. 4.05.2020] Dostupné z: `https://www.logicalincrements.com/articles/framerate`

[2] Ondřej Žára. *Koncepty a triky real-time her více hráčů* [online]. 2017 [cit. 28.04.2020]. Dostupné z: `https://ondras.zarovi.cz/slides/2017/devel/`

[3] Gabriel Gambetta. *Fast-Paced Multiplayer* [online]. Copyright © Gabriel Gambetta 2019 [cit. 28.04.2020]. Dostupné z: `www.gabrielgambetta.com/client-server-game-architecture.htm`

[4] Glenn Fiedler. *Snapshot Interpolation* [online]. Copyright © Glenn Fiedler, 2004 [cit. 22.04.2020]. Dostupné z: `https://gafferongames.com/post/snapshot_interpolation/`

[5] VALVE. *Source Multiplayer Networking* [online]. [cit. 22.04.2020]. Dostupné z: `https://developer.valvesoftware.com/wiki/Source_Multiplayer_Networking`

[6] Robert Nystrom. *Game Loop!* [online]. Copyright © Robert Nystromr, 2009 - 2014 [cit. 23.04.2020]. Dostupné z: `https://gameprogrammingpatterns.com/game-loop.html#the-pattern`

[7] Glenn Fiedler. *Fix Your Timestep!* [online]. Copyright © Glenn Fiedler, 2004 [cit. 23.04.2020]. Dostupné z: `https://gafferongames.com/post/fix_your_timestep/`

[8] MDN. *Window.requestAnimationFrame()* [online]. Copyright © 2005 [cit. 28.04.2020]. Dostupné z: `https://developer.mozilla.org/en-US/docs/Web/API/window/requestAnimationFrame`

[9] VALVE. *Latency Compensating Methods in Client/Server In-game Protocol Design and Optimization* [online]. [cit. 22.04.2020]. Dostupné z: `https://developer.valvesoftware.com/wiki/Latency_Compensating_Methods_in_Client/Server_In-game_Protocol_Design_and_Optimization`

[10] Glenn Fiedler. *Snapshot Compression* [online]. Copyright © Glenn Fiedler, 2004 [cit. 22.04.2020]. Dostupné z: `https://gafferongames.com/post/snapshot_compression//#optimizing-orientation`

Přílohy

Příloha A

Zdrojový kód

V příloze demo.zip se ve složce *source* nachází veškerý zdrojový kód ukázkové implementace. Kód je k dispozici i v GitLab repositáři na adrese `https://gitlab.fel.cvut.cz/neuvimat/network`, commit ID *c1e30d36*, nahraný dne 18.5. 2020.

Kód je rozšířen o dokumentující komentáře ve formátu jsdoc popisující třídy, metody a parametry. Vygenerovaná HTML dokumentace není k dispozici, protože běžně používané nástroje pro její generování nedostatečně podporují standard modulů JavaScriptu verze ES6. Jsdoc nesprávně vyhodnocuje jména modulů při dědění a duplikuje dokumentaci rodičovských tříd. Esdoc sice správně vyhodnocuje jména modulů, není ale tak flexibilní, přehledný a především není plně kompatibilní s použitými jsdoc tagy.

Součástí složky je i základní návod v souboru *readme.md* (v angličtině), který obsahuje pokyny pro sestavení a spuštění projektu.

Příloha B

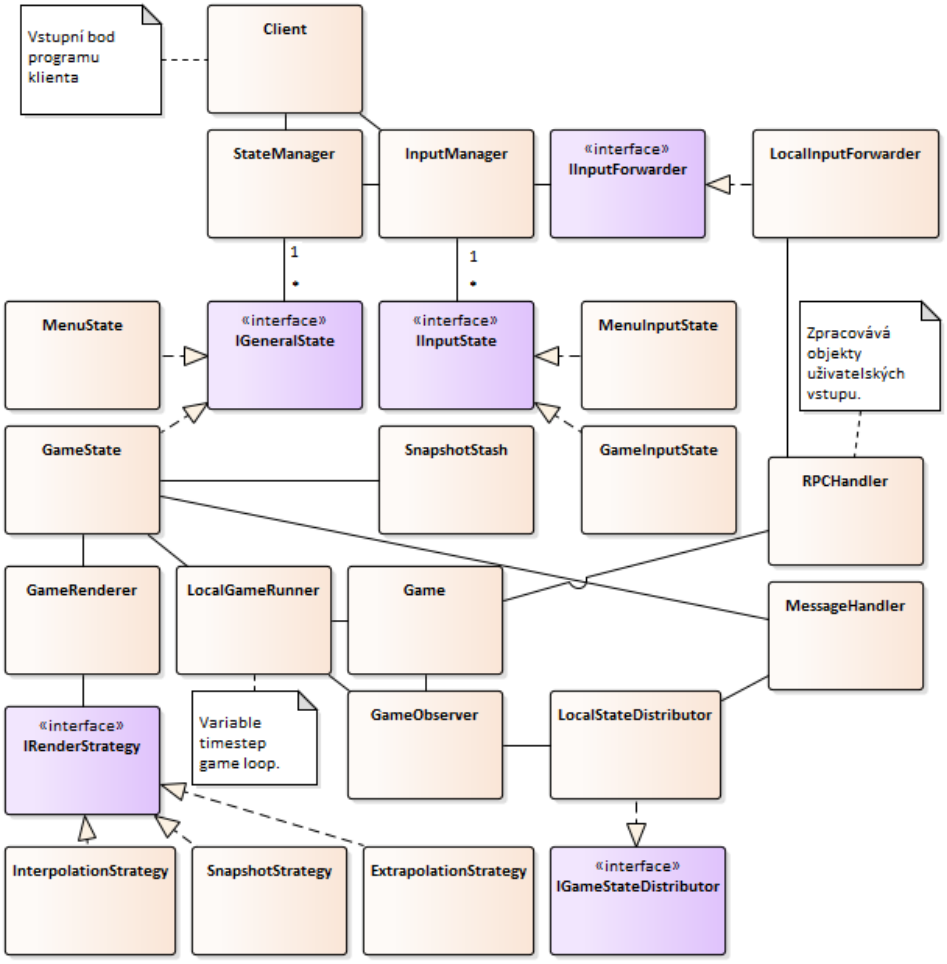
Doménový model

Nadále jsou v příloze demo.zip ve složce *diagrams* zdrojové obrázky pro následující diagramy ve větším měřítku, které zjednodušeně popisují vnitřní strukturu vybraných částí ukázkové implementace:

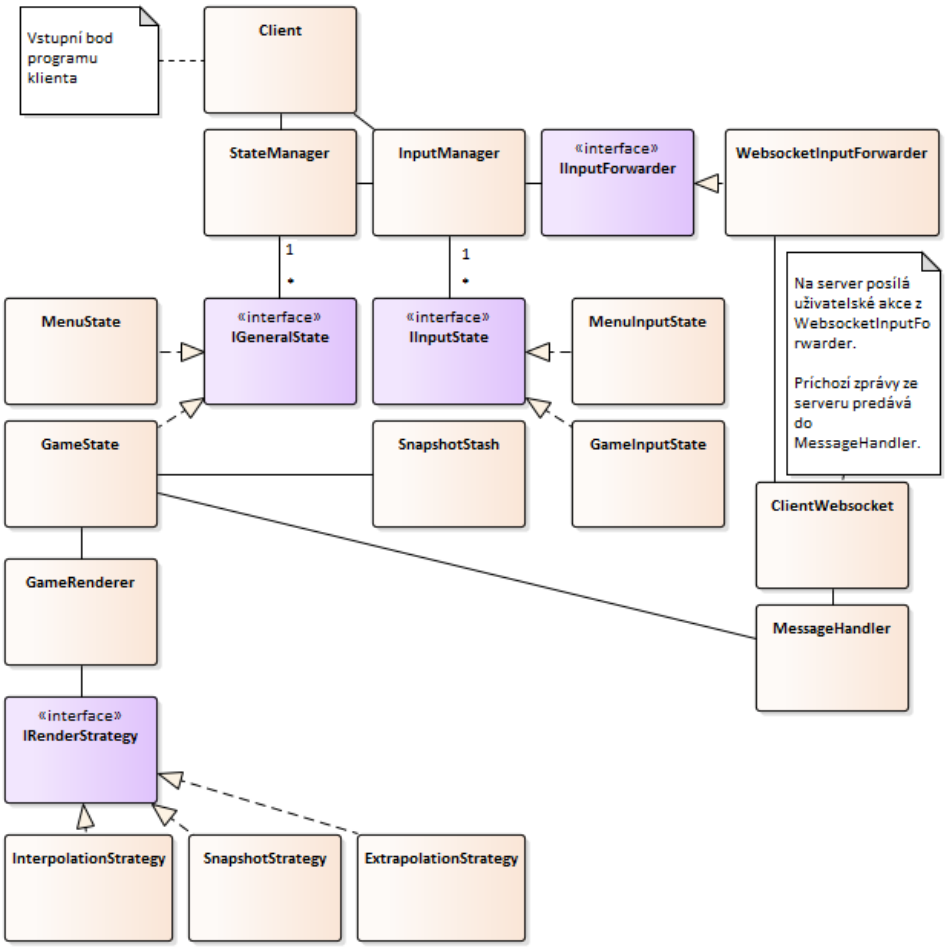
ClientOnline.png – základní třídy a vazby v programu klienta při hře v online režimu.

ClientLocal.png – základní třídy a vazby v programu klienta při hře v režimu jednoho hráče.

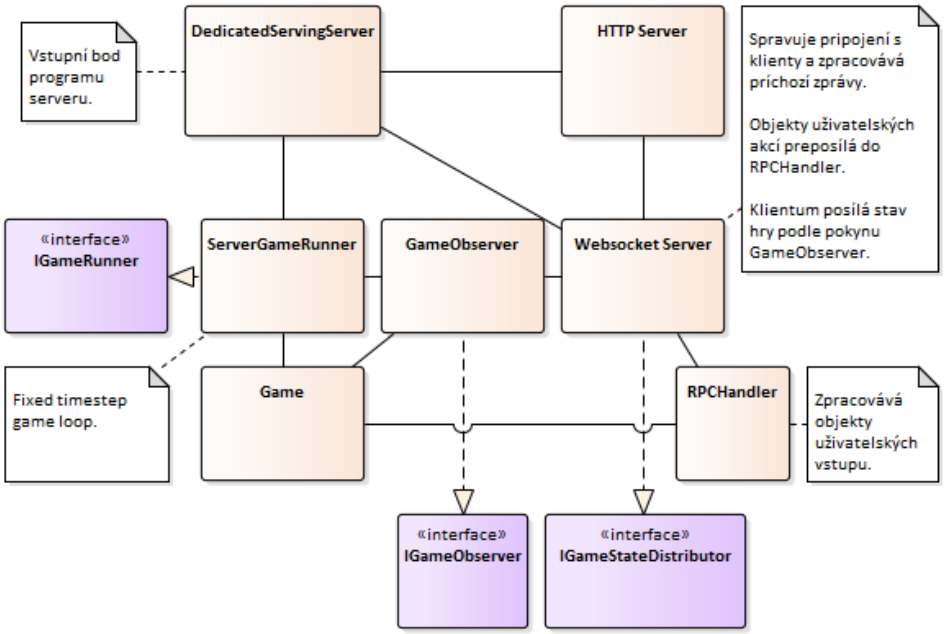
DedicatedServingServer.png – základní třídy a vazby v programu serveru s využitím technologie WebSocket.



Obrázek B.1: ClientLocal.png - znázornění tříd klienta v režimu hry jednoho hráče



Obrázek B.2: ClientOnline.png - znázornění tříd klienta v režimu hry více hráčů



Obrázek B.3: DedicatedServingServer.png - znázornění tříd programu serveru