

ŽILINSKÁ UNIVERZITA V ŽILINE
FAKULTA RIADENIA A INFORMATIKY

BAKALÁRSKA PRÁCA

Študijný odbor: Informatika

Martin Olešnaník

**Vytvorenie real-time hry s
využitím knižnice OpenGL**

Vedúci BP: Ing. Peter Tarábek, PhD.

Reg. č. 384/2013

Ministerské číslo práce: 28360120141384

Dátum zadania práce: 31.10.2013

Dátum odovzdania práce: 6.5.2014

ŽILINSKÁ UNIVERZITA V ŽILINE
FAKULTA RIADENIA A INFORMATIKY

VYTVORENIE REAL-TIME HRY S
VYUŽITÍM KNIŽNICE OPENGL

Bakalárska Práca

Reg. č. 384/2013

Študijný program:	Informatika
Študijný odbor:	9.2.1 Informatika
Školiteľ:	Ing. Peter Tarábek, PhD.
Ministerské číslo práce:	28360120141384

Žilina 2014

Martin Olešnaník

Abstrakt

Práca sa venuje vývoju real-time hry od návrhu herného sveta a hernej logiky až po implementáciu v podobe počítačovej hry s použitím knižnice OpenGL. V prvej časti je práca venovaná analýze súčasného stavu knižnice OpenGL a popisu spolupráce s grafickými kartami. V ďalšej časti pokračujem popisom vytvoreného frameworku Relativity pre podporu a zjednodušenie vykresľovania a komunikácie s rozhraním OpenGL. Práca sa tiež venuje popisu umelej inteligencie, implementácii jednoduchkej AI schopnej nahradiť reálneho protihráča a implementácii systému pre podporu hry viacerých hráčov cez TCP/IP.

Kľúčové slová: Real-time hra, Knižnica OpenGL, Umelá inteligencia, Multiplayer pomocou TCP/IP.

Abstract

The work is dedicated to development of real-time game from designing the game environment and the game logic all the way to implementation in the form of a computer game with the usage of OpenGL library. In the first part the work is focused on current state of the art analysis of OpenGL and description of communication with graphic cards. Later the description of the implemented framework Relativity continues, which was designed as a support platform for rendering and communication with the interface of OpenGL. The work is also dedicated to describing artificial intelligence, implementation of simple AI able to replace real adversary and also implementation of the features allowing multiplayer game-playing through TCP/IP.

Keywords: Real-time game, OpenGL library, Artificial intelligence, Multiplayer through TCP/IP.

Obsah

1 Úvod.....	10
2 Vykresľovanie v počítačových hrách.....	11
2.1 Rozhranie OpenGL.....	12
2.1.1 OpenGL z pohľadu programátora.....	13
2.1.2 Priame vykresľovanie.....	15
2.1.3 Vykresľovanie cez Vertex Buffer Object.....	15
2.1.4 Zobrazenie primitív na obrazovke.....	16
2.1.5 Maticové transformácie.....	16
2.1.6 Perspektívna projekcia.....	17
2.1.7 Rasterizácia primitív.....	19
2.1.8 Shadery.....	20
2.1.9 OpenGL v jazyku Java.....	21
3 Umelá inteligencia v hrách.....	21
3.1 Inteligentný agent.....	23
4 Multiplayer v počítačových hrách.....	24
4.1 Protokoly TCP a UDP.....	25
5 Hra Graviton.....	26
5.1 Úvod do logiky hry.....	26
5.2 Pravidlá interakcie objektov v hre.....	29
5.3 Fyzika planéty.....	31
5.4 Vyhodnocovanie stavu hernej inštancie.....	33
6 Implementácia hry Graviton.....	34
6.1 Framework Relativity.....	35
6.1.1 Základný cyklus aplikácie frameworku Relativity.....	36
6.1.2 Objekty na grafickej karte.....	36
6.1.3 Maticové transformácie.....	37
6.1.4 Texture atlasing.....	37
6.1.5 Vykresľovanie textu.....	38
6.1.6 Časticový systém.....	39

6.2 Grafické užívateľské rozhranie.....	42
6.2.1 Základné komponenty.....	43
6.2.2 9-Patch.....	44
6.3 Umelá inteligencia.....	45
6.4 Podpora hry multiplayer.....	47
6.5 Ostatné balíky implementácie.....	50
7 Záver.....	52

Zoznam skratiek

- **API** – Application Programming Interface
- **GPU** – Graphical Processing Unit
- **LWJGL** – The Lightweight Java Game Library
- **OpenAL** – Open Audio Library
- **OpenCL** – Open Computing Language
- **OpenGL** – Open Graphics Library
- **TCP** – Transmission Control Protocol
- **TCP/IP** – Transmission Control Protocol / Internet Protocol
- **UDP** – User Datagram Protocol
- **VBO** – Vertex Buffer Object

Zoznam obrázkov

Obrázok 1: Zaradenie rozhrania OpenGL v hierarchii softvéru a hardvéru.....	13
Obrázok 2: Prekresľovací cyklus.....	14
Obrázok 3: Ihlan perspektívnej projekcie.....	18
Obrázok 4: Planéta.....	27
Obrázok 5: Graviton.....	27
Obrázok 6: Hranice.....	28
Obrázok 7: Prekážka.....	28
Obrázok 8: Červí systém.....	28
Obrázok 9: Červí priechod.....	28
Obrázok 10: Príťažlivá sila dvoch planét.....	29
Obrázok 11: Pôsobenie gravitonu na planétu.....	30
Obrázok 12: Pôsobenie hraníc na planétu.....	30
Obrázok 13: Prechod červou dierou.....	31
Obrázok 14: Odraz planéty od prekážky.....	31
Obrázok 15: Planéta pred horením.....	33
Obrázok 16: Planéta počas horenia.....	33
Obrázok 17: Textúra častice.....	41
Obrázok 18: Sajúci efekt vytvorený časticovým systémom.....	41
Obrázok 19: Ukážka grafického užívateľského rozhrania.....	42
Obrázok 20: Ukážka tlačidla.....	44
Obrázok 21: Ukážka textového vstupu.....	44
Obrázok 22: Ukážka vertikálneho zoznamu.....	44
Obrázok 23: Ukážka panelu.....	44
Obrázok 24: 9-patch obrázok.....	45
Obrázok 25: Spracovanie objektu abstrakčnou vrstvou.....	49

1 Úvod

Vývoj počítačových hier bol úzko spätý s vývojom počítačov ako takých už od ich začiatkov. V mnohých prípadoch aj v histórii, ale takisto aj dnes, sú práve počítačové hry lídrom v udávaní smeru vývoja nových technológií pre hardvér či softvér. Na trhu každý rok vidíme nové technológie výroby grafických čipov, stále stupňujúce množstvá grafických procesorov (GPU – Graphical Processing Unit) na grafických kartách, ktoré by bez existencie počítačových hier pravdepodobne mimo vedeckých kruhov stratili význam, aký majú dnes.

História počítačových hier sa začala približne v 50. rokoch 20. storočia, kedy sa vývoj počítačov dostal do bodu, kedy boli natoľko výkonné, že dokázali zvládať širokú škálu problémov a mohli byť aplikované do množstva situácií. Niektoré hry, ktoré boli v tomto období vyvinuté, boli však vyvinuté pre demonštráciu problémov napríklad z oblasti umelej inteligencie alebo strategického plánovania vojenských aktivít.

Jednou z prvých komerčne predávaných počítačových hier bola hra Computer Space, vydaná spoločnosťou Nutting Associates na jeseň roku 1971. Jednalo sa o ucelenú hernú konzolu s monitorom, na ktorej mohol hráč hýbať vesmírnou loďou v priestore a mal za úlohu prežiť útoky nepriateľských lodí. Hráč mohol svoju loď rotovať do oboch strán, takisto mohol používať pohon pre zrýchlenie pohybu lode smerom, kam bola otočená.

Počítačové hry sú takisto obrovskou časťou trhu vývoja softvéru. Mnoho zahraničných i domácich spoločností sa venuje explicitne vývoju počítačových hier, na čom postavili základný kameň svojho podnikania. Tieto dôvody vo veľkej miere prispeli ku mojej voľbe vytvoriť vlastnú plnohodnotnú počítačovú hru od návrhu až po konečnú implementáciu vo forme tejto bakalárskej práce.

2 Vykresľovanie v počítačových hrách

Grafická reprezentácia stavu hry a interagovanie užívateľa s hrou cez čo najintuitívnejšie jednoduché rozhranie je jedným z najpodstatnejších dôvodov, prečo hry užívateľov stále bavia.

Vykresliť stav hry (scénu hry) však vždy nebolo tak jednoduché, ako je tomu dnes. V časoch, keď na trh začali prichádzať prvé počítačové hry, mal programátor počítačovej hry dostupné len veľmi jednoduché prostriedky na vypisovanie textu, prípadne zobrazovanie špeciálnych znakov. V najlepšom prípade mal programátor kontrolu nad najmenšou možnou jednotkou pri vykresľovaní (ďalej “pixel”). Programátor mal možnosť určiť, či sa daný pixel zobrazí, a neskôr získal kontrolu aj nad farbou a odtieňom, v ktorom sa daný pixel na obrazovke objaví.

Takéto prostriedky ponúkali programátorom počítačových hier dostatočný priestor a možnosti aby mohli tvoriť počítačové hry, ktoré budú graficky reprezentovať stav hry nielen textom, ale aj grafickým vyzobrazením objektov, ktoré sa v hre nachádzajú. Nevýhodou tohto prístupu bol nedostatok technických prostriedkov a výkonu počítačov, ktorý nedokázal držať krok s dopytom po vykresľovaní čoraz krajších herných scénach, ktoré by dokázali byť bližšie realite, ktorú vidíme okolo seba.

Hlavným problémom takéhoto prístupu vykresľovania bola nutnosť programového riadenia celého vykresľovania hernej scény. Procesor nevyužíval čas len na spracovanie hernej logiky a signálov zodpovedných za spracovanie interakcií od užívateľa, bol tiež zodpovedný za cyklické prekresľovanie scény. Scénu bolo nutné v reálnom čase prekresľovať dostatočne rýchlo, aby boli užívateľovi graficky prezentované všetky podstatné zmeny herného stavu a takisto bola zachovaná plynulosť animácie, ktorou tieto zmeny boli reprezentované.

Postupom času boli vytvorené technické prostriedky (hardvér) špecializované na urýchlenie vykresľovacieho procesu a zníženie času, ktorý musel procesor venovať vykresleniu scény. Časť z týchto technických prostriedkov v dnešnej dobe poznáme pod názvom grafické karty. V dnešnej dobe existuje viacero výrobcov grafických kariet, ktoré sú schopné zabezpečovať rýchle vykreslenie zložitých objektov na obrazovku, kde

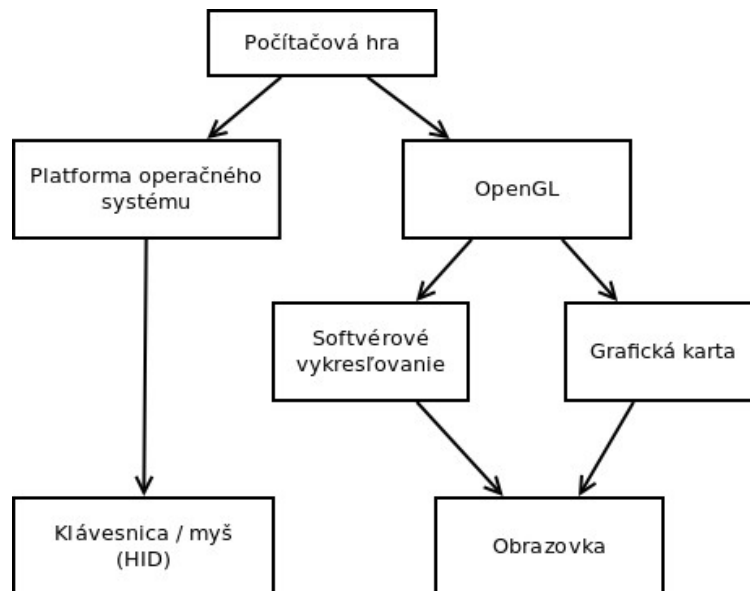
procesor riadi len konkrétne uloženie objektov na obrazovke, ich pohyb apod., avšak neriadi samotné zmeny jednotlivých pixelov na obrazovke.

Hardvér výrobcu každej grafickej karty sa však z určitej časti líši. Tento fakt spôsobuje problém programátorovi počítačových hier, nakoľko by musel pre každú grafickú kartu, s ktorou by sa užívateľ, ktorý by chcel jeho hru hrať, programovať samostatné obslužné časti a vykresľovacie procesy. Ako riešením tohto problému začali postupne vznikať grafické knižnice, ktoré programátorovi poskytujú základne funkcie a primitíva na vykresľovanie scény, zatiaľ čo zahŕňujú konkrétny hardvér (grafické karty), na ktorom sú dané funkcie implementované. Tieto knižnice často nezaručujú, že herná scéna vykreslená pomocou rovnakých volaní funkcií bude na všetkých grafických kartách vyzerat' rovnako (teda že sa jednotlivé pixely nebudú líšiť), avšak vedia zaručiť, že esencia scény, ktorá sa zobrazuje, bude zachovaná s čo najväčším dôrazom na detail.

2.1 Rozhranie OpenGL

Rozhranie (API – Application Programming Interface) OpenGL (Open Graphics Library)[2] bolo vyvinuté za účelom vytvorenia univerzálneho spôsobu vykresľovania grafických objektov na obrazovku. Konkrétna implementácia vykresľovania je závislá na grafickej karte a i jej prítomnosti v počítači. Rozhranie OpenGL môže byť použité dokonca v prípade, ak počítač grafickú kartu nemá zavedenú, kedy sa používa softvérová implementácia vykresľovania a teda sa spätne používa spôsob, kedy procesor riadi celú hierarchiu vykresľovania scény. Týmto spôsobom je hardvér grafických kariet programátorovi zahalený abstrakciou samotného rozhrania, čím sa značne zjednodušuje implementácia vykresľovania v hre a je možné pri vývoji hry viac času venovať hernej logike a interakcii hry s užívateľom.

Základným kameňom rozhrania OpenGL je multiplatformovosť a nezávislosť na hardvéri. V dnešnej dobe je možné používať OpenGL na platformách s Unix-ovým základom, na systémoch Microsoft Windows a mnohých ďalších. Rozhranie OpenGL takisto zaručuje nezávislosť s užívateľským prostredím operačného systému a teda neposkytuje funkcie pre správu signálov z klávesnice alebo myši, prípadne iných nástrojov pre interakciu s užívateľom. Tieto signály musí obsluhovať programátor mimo rozhrania OpenGL.

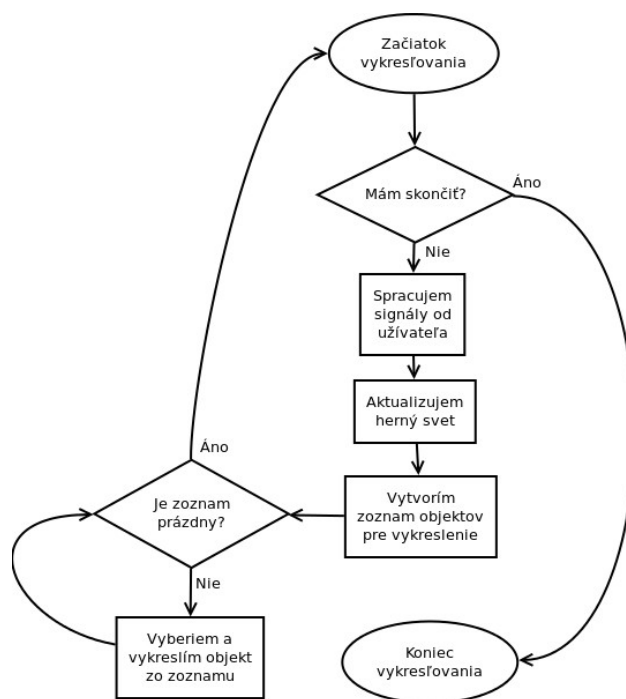


Obrázok 1: Zaradenie rozhrania OpenGL v hierarchii softvéru a hardvéru

2.1.1 OpenGL z pohľadu programátora

Rozhranie OpenGL sa skladá z funkcií, dátových typov a konštánt. Spolu tieto prvky definujú rozhranie, ktorým je programátor schopný vykresľovať hernú scénu a vykresľovanie prebieha pomocou zmien stavového stroja rozhrania OpenGL. Rôznymi postupnosťami volaní sme schopní vykresľovať objekty od najmenších primitív, ako sú napríklad bod, čiara alebo trojuholník až po komplexné scény popisujúce virtuálnu realitu herného sveta. Postupným volaním funkcií rozhrania OpenGL je takisto možné definovať vzhľad a správanie jednotlivých vykresľovaných primitív vzhľadom na ostatné vykreslené primitíva, ako sú napríklad farba, priehľadnosť, použitá textúra apod.

OpenGL je vhodné ako aj na vykreslenie statických scén napríklad pre účely vyuzalíacie architektonických návrhov, v hrách sa však zvyčajne používa na vykreslenie zložitých grafických animácií a zachytenie herného priestoru v reálnom čase.



Obrázok 2: Prekresľovací cyklus

Tradičná obnovovacia frekvencia, teda počet prekreslení scény na obrazovku sa v dnešnej dobe pohybuje medzi 25 až 100Hz. Obnovovacia frekvencia je závislá na rýchlosti samotného hardvéru ako aj zložitosti herných simulácií, ktoré sa musia vykonať medzi jednotlivými prekresleniami scény. Pri každom prekreslení je nutné určiť, ktorý objekt alebo primitívum sa má zobrazíť, ako sa má vykresliť, na akej pozícii a množstvo ďalších parametrov. K týmto výpočtom môže dochádzať asynchrónne, avšak rozhraniu OpenGL musia byť podávané synchrónne, nakoľko OpenGL bežne nepodporuje kreslenie jednej scény z viacerých vlákien / procesov operačného systému.

Existuje niekoľko spôsobov, ktorými môžeme objekty vykresľovať, najčastejšie sa používajú:

- **Priame vykresľovanie (Immediate OpenGL rendering)**, kedy sa každý grafický objekt na hernej scéne rozloží na najjednoduchšie primitíva, a tieto sa postupne v cykle vykresľujú.
- **Vykresľovanie cez Vertex Buffer Object (Non-Immediate OpenGL rendering)**, (ďalej len “Vykresľovanie cez VBO”), kedy celé objekty alebo skupiny objektov rozložíme na primitíva len raz, dáta o pozíciách a tvaroch jednotlivých primitív uložíme do pamäte grafickej karty, odkiaľ ich následne vykresľujeme.

2.1.2 Priame vykresľovanie

Priame vykresľovanie hernej scény je prístupné pomocou rozhrania OpenGL už od prvých vydání, nakoľko sa jedná o najprimitívnejší spôsob vykresľovania objektov na scénu. Výhodou tohto prístupu je jednoduchosť. V jednom cykle prekreslenia sa na procesore musí vykonať nasledovný algoritmus:

1. Pre každý objekt na scéne vykonaj krok č.2
2. Rozložím objekt na primitíva, a pre každé primitívum vykonaj krok č.3
3. Vykresli konkrétne primitívum cez rozhranie OpenGL.

Je zrejmé, že takýto algoritmus má asymptotickú zložitosť lineárne závislú od počtu primitív, na ktoré sa rozdelia všetky objekty potrebné pre prekreslenie scény. Je dobré si uvedomiť, že jeden grafický objekt vykresľovaný na scénu sa môže skladať až z niekoľkých stoviek až tisícok primitív, aby bol dosiahnutý dostatočný grafický a umelecký detail.

2.1.3 Vykresľovanie cez Vertex Buffer Object

Vykresľovanie cez VBO je prístupné cez jadro rozhrania OpenGL od verzie OpenGL 2.1 a funguje v dvoch fázach:

1. Objekty alebo skupiny objektov sú rozložené na primitíva, ktoré sú následne uložené do pamäte grafickej karty. Grafická karta nám ku uloženým dátam poskytne referenciu, vďaka ktorej môžeme následne objekt alebo skupinu objektov vykresliť. Je zrejmé, že zložitosť tejto fázy, rovnako ako pri algoritme priameho vykresľovania, je lineárne závislá od počtu primitív, ktoré je potrebné na vykreslenie scény.
2. Objekty alebo skupiny objektov sú vykresľované pomocou referencií, ktoré boli ku objektom alebo skupinám objektov poskytnuté v prvej fáze vykresľovania cez VBO. Je zrejmé, že zložitosť tejto fázy je lineárne závislá od počtu objektov alebo skupín objektov, ktoré sú potrebné na vykreslenie scény.

Hlavným rozdielom a výhodou vykresľovania cez VBO oproti priamemu vykresľovaniu je oddelenie prvej a druhej fázy, kde iba druhú z nich potrebujeme

vykonávať pri každom prekreslení scény. Síce prvá fáza vykresľovania cez VBO má rovnakú zložitosť ako algoritmus priameho vykresľovania, nakoľko je nutné vykonať ju iba raz pred začatím prekresľovania scény, je z nášho pohľadu ľahko zanedbateľná.

Je zrejmé, že výsledná zložitosť prekreslenia v prípade použitia vykresľovania cez VBO je asymptoticky lineárne závislá od počtu objektov. Takáto zmena by nijak praktický výsledok nezlepšila, pokiaľ by sme ako objekty používali jednotlivé primitíva. Takýto prípad sa v praxi vyskytne však veľmi zriedka, nakoľko v dnešnej dobe sa pohybuje priemerný počet primitív pripadajúcich ku jednému objektu okolo stoviek až tisícok.

2.1.4 Zobrazenie primitív na obrazovke

Pri programovaní počítačových hier sa stretávame so situáciou, kedy veľmi dobre vieme definovať, ako vyzerá naša herná scéna. Vieme povedať, kde sa hra začína, a ktorým smerom, akou veľkosťou sú v trojrozmernom priestore uložené naše herné objekty, prípadne aj samotné primitíva. Z tohto stavu je však ešte veľmi dlhá cesta ku reálnemu zobrazeniu objektov na obrazovke.

Musíme totiž zabezpečiť, aby sme pre každý trojrozmerný vektor našej scény $\vec{v}_s$ vedeli nájsť vektor $\vec{v}_o$, ktorý bude jeho zobrazením na obrazovke. Takýmto spôsobom vieme všetky primitíva zo scény jednoducho transformovať a následne vykresliť na obrazovku.

2.1.5 Maticové transformácie

Ako základný nástroj pre transformovanie vektorov sa používa maticová transformácia. Každá bežne nutná transformácia (či už rotácia, posunutie, skosenie alebo aj perspektívna projekcia) sa dá vyjadriť ako násobenie transformačnej matice typu 4x4 popisujúcej konkrétny typ transformácie naším vektorom (doplneným o štvrtý komponent rovný 1). Výsledný vektor bude mať síce 4 komponenty, no pre nás budú vo finálnom výsledku zaujímavé iba 3 z nich.

Nech máme pôvodný vektor $\vec{v}_s = (1, 0, 0)$ a transformáciu definovanú ako “posun o 2 po prvej osi, o 3 po druhej osi a o 4 po tretej osi”. Matica transformácie posunutia A bude

$$A = \begin{pmatrix} 1 & 0 & 0 & 2 \\ 0 & 1 & 0 & 3 \\ 0 & 0 & 1 & 4 \\ 0 & 0 & 0 & 1 \end{pmatrix} \text{ a transformácia: } \begin{pmatrix} 1 & 0 & 0 & 2 \\ 0 & 1 & 0 & 3 \\ 0 & 0 & 1 & 4 \\ 0 & 0 & 0 & 1 \end{pmatrix} \times \begin{pmatrix} 1 \\ 0 \\ 0 \\ 1 \end{pmatrix} = \begin{pmatrix} 3 \\ 3 \\ 4 \\ 1 \end{pmatrix} \text{ bude popisovať}$$

výsledný vektor $\vec{v}_t = (3, 3, 4)$ po transformácii vektora $\vec{v}_s = (1, 0, 0)$ vyššie zmienanou transformáciou.

Takýmto spôsobom môžeme rôzne transformácie postupne kombinovať, napríklad rotovať, posunúť, a následne znova rotovať. Je však zrejmé, že len niektoré transformácie sú medzi sebou komutatívne, a teda nezáleží na ich poradí (napríklad posunutie).

2.1.6 Perspektívna projekcia

Jednou z najkomplikovanejších maticových transformácií je perspektívna projekcia vektoru z priestoru na obrazovku podobne, ako by sme sa na priestor scény pozerali cez obrazovku, kde vidíme zaznamenaný obraz z kamery zo scény. Zostavenie transformačnej matice pre perspektívnu projekciu sa pre zjednodušenie rozdeľuje na dve fázy:

1. **Vytvorenie pohľadovej transformačnej matice (view matrix) V** , ktorá zachytáva transformáciu kamery, teda otočenie scény a nastavenie pozície kamery.
2. **Vytvorenie projekčnej transformačnej matice (projection matrix) P** , ktorá zachytáva projekciu toho, čo vidí kamera, na plochu obrazovky.

Výslednú transformačnú maticu T zachytávajúcu aj transformáciu kamery ako aj projekciu na obrazovku dostaneme jednoduchým násobením matice $T = P \times V$.

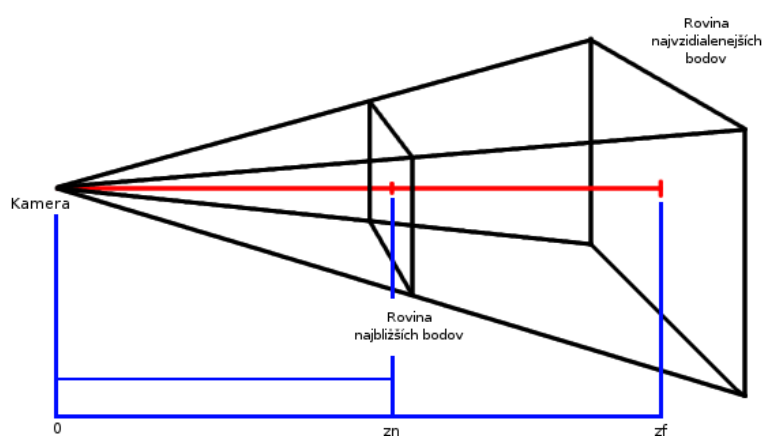
Nastavením kamery pri perspektívnej projekcii rozumieme niekoľko parametrov vstupujúcich do pohľadovej transformačnej matice:

- **pozícia kamery** $\vec{c}_p$ – trojrozmerný vektor popisujúci pozíciu kameru v 3D svete scény, ktorú vykresľujeme,
- **cieľ kamery** $\vec{c}_t$ – trojrozmerný vektor popisujúci cieľ v 3D svete scény, na ktorý je kamera namierená (bod, ktorý vidíme v strede kamery),
- **vektor otočenia kamery (up-vector)** $\vec{c}_u$ – vektor, ktorý popisuje, ktorým smerom v trojrozmernom svete scény by bol namierený vrch kamery, tradične sa

tento vektor definuje ako $\vec{c}_u = (0, 1, 0)$, ak druhú (teda y-novú) súradnicu považujeme za vertikálnu súradnicu a kladné hodnoty popisujú svet nad horizontom.

Majme vektory $\vec{c}_f = \vec{c}_t - \vec{c}_p$, $\vec{c}_r = \vec{c}_f \times \vec{c}_u$, $\vec{c}_w = \vec{c}_r \times \vec{c}_f$, potom pohľadovú

transformačnú maticu definujeme ako:
$$V = \begin{pmatrix} \vec{c}_{rx} & \vec{c}_{wx} & -\vec{c}_{fx} & 0 \\ \vec{c}_{ry} & \vec{c}_{wy} & -\vec{c}_{fy} & 0 \\ \vec{c}_{rz} & \vec{c}_{wz} & -\vec{c}_{fz} & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}.$$



Obrázok 3: Ihlan perspektívnej projekcie

Nastavením obrazovky rozumieme parametre vstupujúce do projekčnej transformačnej matice:

- **uhol pohľadu f** – popisuje uhol medzi rovinami, ktoré vzniknú zobrazením ľavej a pravej hrany obrazovky do priestoru scény, z ktorej zobrazujeme na obrazovku,
- **pomer strán obrazovky r** – výsledok podielu šírky a výšky obrazovky.
- **najväčšia vzdialenosť od kamery z_f** – vzdialenosť najvzdialenejšieho možného zachyteného bodu od kamery, ktorý sa na obrazovku premietne,
- **najmenšia vzdialenosť od kamery z_n** – vzdialenosť najbližšieho možného zachyteného bodu od kamery, ktorý sa na obrazovku premietne.

Následne projekčná matica vyzerá nasledovne:

$$P = \begin{pmatrix} \frac{1}{\tan(f/2)*r} & 0 & 0 & 0 \\ 0 & \frac{1}{\tan(f/2)} & 0 & 0 \\ 0 & 0 & -\frac{z_f + z_n}{z_f - z_n} & -1 \\ 0 & 0 & -\frac{2*z_n*z_f}{z_f - z_n} & 1 \end{pmatrix}$$

Je dobré si uvedomiť, že transformovanie pomocou perspektívnej projekcie (nie však výlučne, tento efekt platí napríklad aj pre ortogonálnu projekciu) nám transformuje vektory z trojrozmerného priestoru scény do trojrozmerného priestoru obrazovky. Tretím rozmerom obrazovky je hĺbka bodu, pomocou ktorej vieme jednoducho rozhodnúť o spôsobe prekrývania primitív vykresľovaných na obrazovke. Ak by sme totiž tento tretí rozmer ignorovali a vykresľovali v poradí od najbližších objektov až po tie najvzdialenejšie, na obrazovke by sme videli nezmysly, pretože by sme videli v popredí objekty pozadia, ktoré majú byť zakryté objektami popredia. [4]

2.1.7 Rasterizácia primitív

Rasterizácia primitív je proces, ktorý transformuje primitíva z trojrozmerného priestoru obrazovky na fragmenty. Fragment je grafická štruktúra, ktorá je pevne viazaná na pixel obrazovky, obsahuje však okrem farby pixelu niekoľko ďalších dôležitých parametrov.

Zaujímavým parametrom fragmentu je okrem farby aj hĺbka fragmentu. Hĺbka fragmentu je, ako bolo vyššie spomenuté, rozhodovacím faktorom pri prekrývaní vykresľovaných primitív. Každé vykresľované primitívum sa na grafickej karte transformuje na fragmenty obrazovky. Parametre konkrétneho nového fragmentu (farba, hĺbka) sa zvyčajne tvoria interpoláciou parametrov jednotlivých vertexov (bodov, pomocou ktorých je definované primitívum). Hĺbka takto vytvoreného nového fragmentu je následne porovnaná s hĺbkou už predošlého vykresleného fragmentu. Podľa definovaného pravidla, ktoré je možné cez rozhranie OpenGL upraviť, sa buď prekryje už vykreslený fragment novým fragmentom, alebo je nový fragment ignorovaný. Týmto spôsobom je rasterizácia schopná na obrazovke veľmi jednoducho vytvoriť dokonca čiastočne prekryté zobrazenia

primitív. [3]

2.1.8 Shadery

Často sa programátor stretne pri programovaní grafickej aplikácie alebo hry so situáciou, kedy potrebuje pre každý vertex primitíva vykonať určitý výpočet, aby upravil jeho pozíciu, farbu, apod. V prípade, kedy by takúto úlohu riešil programovo na procesore, mohla by zložitosť jedného prekreslenia dosiahnuť asymptotickú zložitosť lineárnu v závislosti od počtu vertexov, a teda by sa dostal do situácie, kedy by napríklad vykresľovanie cez VBO stratilo voči priamemu vykresľovaniu svoje hlavné výhody.

Prostriedkom pre vyriešenie tohto problému sú shadery. Shader je jednoduchý program, ktorý nie je spúšťaný na hlavnom procesore počítača, ale na špecializovaných procesoroch grafickej karty. Shader prijíma ako vstup programu dáta tradične malej štruktúry (napríklad jedného primitíva, vertexu alebo fragmentu) a výstupom je podobne malá štruktúra. Štandardne nemá shader prístup k dátam iných štruktúr než štruktúr, ktoré prijal na svojom vstupe. Takéto obmedzenie robí shader ideálnym prístupom pre paralelizáciu výpočtov pre takéto štruktúry. Nakoľko sa pri bežnom vykreslení spracúvajú stovky až tisícky takýchto malých štruktúr, je možné využiť všetok potenciál grafickej karty, ktorá v dnešnej dobe obsahuje stovky až tisícky malých špecializovaných procesorov.

Existuje niekoľko typov shaderov, kde každý z nich vstupuje do inej časti vykresľovacieho procesu. Hlavnými dvoma typmi shaderov sú:

- **Vertex shader:** ako vstup prijíma dáta vertexu, teda bodu, z ktorých je zostrojené primitívum. Výstupom vertex shadera sú dáta toho istého vertexu, avšak počas spustenia shadera môže shader rôzne upraviť pozíciu, farbu vertexu, prípadne k dátam vertexu pripojiť iný dátový parameter.
- **Fragment shader:** ako vstup prijíma dáta fragmentu (bežne po ukončení rasterizácie primitívu), teda pixelu spolu s ďalšími dôležitými parametrami ako sú farba, prípadne hĺbka. Fragment shader môže, takisto ako vertex shader, manipulovať s akýmkoľvek parametrom dát fragmentu. Pridávať ďalší dátový parameter k dátam fragmentu na výstupe fragment shadera však tradične nemá

zmysel, nakoľko fragment shader je posledným shaderom, ktorý sa vo vykresľovaní procese spúšťa.

2.1.9 OpenGL v jazyku Java

Rozhranie OpenGL je navrhnuté ako multiplatformové a nezávislé na programovacom jazyku. V každom programovacom jazyku však musí existovať sada funkcií alebo definícií, ktoré umožňujú prístup ku natívnym implementáciám rozhrania OpenGL.

Programovací jazyk Java, takisto ako OpenGL bol navrhnutý ako platformovo nezávislý jazyk, so sloganom “write once, run anywhere” (*Slovensky: napíš raz, spúšť kdekoľvek*). Platformová nezávislosť jazyka Java a rozhrania OpenGL spolu s vysokoúrovňovou programovacieho jazyka Java sú hlavnými dôvodmi voľby tejto kombinácie pre praktickú časť tejto bakalárskej práce.

Existuje viacero programových knižníc, ktoré sprístupňujú rozhranie OpenGL pre programovací jazyk Java. Jednou z nich je aj knižnica LWJGL (The Lightweight Java Game Library), ktorá bola vyvinutá za účelom sprístupniť programovaciemu jazyku Java tri rozhrania: OpenGL (rozhranie pre grafické funkcie), OpenCL (rozhranie pre paralelné výpočty) a OpenAL (rozhranie pre prácu so zvukovými aparátmi). Hlavným dôvodom voľby knižnice LWJGL v praktickej aplikácii tejto bakalárskej práce je vysoká kvalita implementácie a dokumentácie samotnej knižnice.

3 Umelá inteligencia v hrách

Existuje mnoho definícií umelej inteligencie ako takej. Niektoré z nich sa zameriavajú na porovnanie správania počítačových systémov voči správaniu ľudí, kdežto iné sa zameriavajú na správanie počítačových systémov voči ideálnemu prípadu riešenia daného problému. V modernej literatúre sa vyskytuje niekoľko zaujímavých popisov, čo je to umelá inteligencia.

“Napínavé úsilie naučiť počítače myslieť...”	“Štúdia duševných schopností pomocou počítače s myslou, v plnom a doslovnom výpočtových modelov” (Charniak and
--	--

zmysle” (Haugeland, 1985)	McDermott, 1985)
“Automatizácia aktivít spájaných s ľudským myslením, aktivít ako sú rozhodovanie, riešenie problémov, učenie...” (Bellman, 1978)	“Štúdia výpočtov, ktoré umožňujú vnímať, uvažovať a konať” (Winston, 1992)
“Umenie tvorby strojov, ktoré sú schopné vykonávať funkcie, ku ktorým je potrebná inteligencia v prípade vykonávania ľuďmi.” (Kurzweil, 1990)	“Odbor, ktorý sa snaží vysvetliť a napodobniť inteligentné správanie z hľadiska výpočtových procesov” (Schalkoff, 1990)
“Štúdia o tom, ako naučiť počítače robiť veci, v ktorých sú zatiaľ ľudia lepší” (Rich a Knight, 1991)	“Vetva vedy o počítačoch ktorá sa zameriava na automatizáciu inteligentného správania” (Luger and Stubblefield, 1993)

Rozdelenie definícií umelých inteligencií [1]

V hrách sa najčastejšie používa umelá inteligencia na simuláciu protihráča pre prípad, kedy sa hra buď nedá hrať viacerými reálnymi hráčmi v jednom čase, alebo sa hráč rozhodne hrať proti umelej inteligencii napríklad preto, že umelá inteligencia je pre hráča väčšou výzvou ako bežný protihráč.

Hry sa však zvyknú používať aj pre demonštráciu rôznych algoritmov a prístupov ku tvoreniu umelej inteligencie, nakoľko poskytujú veľkú škálu rôznorodých systémov, v ktorých je simulácia a demonštrácia algoritmov, konceptov a ideí mnohokrát jasnejšia a jednoduchšia, ako ich demonštrovať v reálnych systémoch.

“Hry sú fascinujúce, a písanie programov schopných hrať hry je možno ešte viac fascinujúce. Môžeme povedať, že hranie hier ku umelej inteligencii je ako závody Grand Prix ku automobilovému priemyslu: aj keď špecifickosť úlohy a extrémne kompetitívne podmienky vedú k vývoju systémom ktoré nevyzerajú ako obecný, všadepoužiteľný inteligentný systém, ich výsledkom býva množstvo popredných konceptov a inžinierskych myšlienok.” [1]

Ak potrebujeme vytvoriť umelú inteligenciu pre hru, kedy nie je pre nás najdôležitejšie zamerať sa na optimálnosť konania umelej inteligencie v hre, ale je pre nás najdôležitejší zážitok hráča, ktorý proti umelej inteligencii hrá, riešenie sa častokrát veľmi

líši a výsledok riešenia umelej inteligencie v danej inštancii hry je častokrát neoptimálny a iracionálny.

Na druhej strane je dobré si uvedomiť, že pokiaľ by umelá inteligencia v hre proti reálnemu hráčovi vždy konala iba racionálne a optimálne, v prípade, že sa jedná o hru s férovým začiatkom a pravidlami ktoré nezvýhodňujú konkrétneho hráča, hra by sa pre užívateľa stala veľmi rýchlo neatraktívnou. Reálny užívateľ totiž častokrát sám nevykonáva iba racionálne a optimálne riešenia, čo pre užívateľa veľmi často môže vyústiť do tvrdej prehry.

3.1 Inteligentný agent

Častým prístupom ku tvoreniu umelej inteligencie je tvorba **inteligentných agentov**. **Agentom** sa rozumie program, ktorý dokáže spracovávať vnemy (percepts) pomocou senzorov a následne vykonávať podľa vnemov určité akcie pomocou aktorov (actuators). Senzory teda poskytujú agentovi informácie o okolitom systéme a aktóri dávajú možnosť agentovi vplývať na okolitý systém. Senzory a aktóry sú jediné komunikačné kanály agenta s okolitým systémom. Pre potreby tejto bakalárskej práce si môžeme definovať niekoľko pojmov, ako sú napríklad racionálny alebo inteligentný agent.

Pred definíciou **racionálneho a inteligentného agenta** je dôležité popísať funkciu výsledku. **Funkcia výsledku agenta** v určitom systéme nám popisuje, ako vhodné bolo rozhodovanie agenta (interpretované ako stavy okolitého systému spolu s akciami, ktoré agent vykonal cez aktórov) v danej situácii (stavy okolitého systému a možné akcie, ktoré agent mohol v týchto stavoch vykonať). Funkcia výsledku musí byť funkciou, nad ktorou oborom hodnôt je jasne definovaná relácia “je lepší ako”, ktorá je aspoň čiastočným usporiadaním.

Spolu s takto definovanou funkciou výsledku agenta v určitom systéme je jednoduché popísať **optimálne rozhodovanie agenta** v určitom systéme. Takým rozhodovaním (teda postupnosťou stavov systému a akcií aktórov agenta) rozumieme rozhodovanie, ktoré má zo všetkých možných rozhodovaní najväčšiu možnú funkciu výsledku.

Racionálny agent je taký agent, ktorý zo všetkých možných rozhodovaní

vykonáva také rozhodovanie, ktoré je optimálnym rozhodovaním.

Inteligentný agent je taký agent, ktorý sa zo všetkých možných rozhodovaní snaží vybrať a vykonať také rozhodovanie, ktoré vedie k najvyššej možnej funkcii výsledku agenta. Táto snaha však môže byť obmedzená schopnosťami agenta rozhodovať o vnemoch, o systéme, o súvislostiach, a preto teda nemusí viesť ku optimálnemu rozhodovaniu.

4 Multiplayer v počítačových hrách

Každú hru môžeme klasifikovať podľa počtu hráčov vystupujúcich a interagujúcich v rámci jednej hry na hry jedného hráča, na hry dvoch hráčov alebo na hry viacerých hráčov. Hry jedného hráča sú napríklad Sudoku, Míny alebo Mahjongg. Hrami pre dvoch hráčov sú napríklad šach alebo dáma. Hrami viacerých hráčov sú napríklad človeče nehnevaj sa, kartové hry poker alebo kanasta. Medzi hry viacerých hráčov patria aj také hry, ktoré môžu byť hrané dvoma hráčmi, ale takisto môžu byť hrané viacerými hráčmi.

V počítačových hrách bola prítomná možnosť hrania počítačovej hry (alebo multiplayer) už od ich začiatkov, kedy počítačové hry začali vznikať. Existuje však niekoľko spôsobov, závislých na technologických možnostiach platformy pre počítačové hry, akými môžu hráči počítačových hier zdieľať herný svet:

- **striedanie ťahov na jednom počítači** – spôsob, ktorý sa využíva výhradne pri tzv. ťahových hrách, kedy je hra rozdelená do diskretných okamihov, kedy určitý hráč môže ovplyvniť hru, a následne “je na ťahu” ďalší hráč v závislosti od pravidiel danej hry. V počítačovej hre sa takéto striedanie implementuje postupným striedaním hráča pri užívateľskom prostredí hry, ktoré ponúka hráčovi, ktorý je momentálne na ťahu, možnosť vykonať svoj ťah. Takisto po konci ťahu určuje hráča, ktorý má nasledovať v ďalšom ťahu a teda vykonať svoj ťah cez užívateľské prostredie hry. Takýto spôsob hry viacerých hráčov počítačovej hry nekladie prakticky žiadne technické požiadavky na platformu, pre ktorú je počítačová hra vyvíjaná a celá implementácia je závislá na programátorovi.
- **súvislá hra viacerých hráčov na jednom počítači** – spôsob, ktorý sa používa zvyčajne pri hrách s prostredím reálneho času, kedy musia v jednom momente

vykonávať rozhodnutia v hre viacerí hráči. Takáto hra viacerých hráčov už vyžaduje schopnosť užívateľského rozhrania hry ale aj platformy zvládať vstupy od viacerých hráčov, napríklad byť schopná akceptovať veľa súvislých vstupov z klávesnice alebo podporovať viacero lokalizačných zariadení (myši, joystickov, apod.).

- **súvislá hra viacerých hráčov na viacerých počítačoch** – spôsob, ktorý sa napríklad používa, ak sa hráči hry nenachádzajú na jednom fyzickom mieste a teda nemôžu fyzicky zdieľať hru alebo si princíp hry vyžaduje, aby hráči boli od seba oddelení a teda napríklad bolo zamedzené sledovaniu aktivít druhého hráča v rámci herného sveta. Tento spôsob je najkomplikovanejší, pretože si vyžaduje asynchrónnu komunikáciu jednotlivých procesov hry medzi dvoma alebo viacerými počítačmi a teda vyžaduje nie len technické prostriedky počítača schopné prenášať dáta ale takisto často veľmi komplikovanú implementáciu synchronizácie herného sveta medzi viacerými počítačmi.

4.1 Protokoly TCP a UDP

Jedným z častých spôsobov implementácie komunikácie viacerých procesov, či už v rámci jedného počítača, v rámci lokálnej siete alebo v rámci siete Internet je implementácia na základe protokolov TCP a UDP. Protokoly TCP (Transmission Control Protocol) a UDP (User Datagram Protocol) sú protokolmi transportnej vrstvy ISO/OSI modelu. Oba protokoly sú schopné vytvárať komunikačné kanály medzi dvoma alebo viacerými procesmi, každý z nich má však svoje špecifiká, výhody a nevýhody:

- **Transmission Control Protocol (TCP)** je určený na spojovanú komunikáciu, kedy je zaručená správnosť a následnosť doručených dát tak, ako boli dáta posielané. Je teda výhodné používať protokol TCP v prípade, pokiaľ sú tieto vlastnosti komunikácie nevyhnutné a teda nie je napríklad možné predpokladať stratenie časti komunikácie alebo doručenie dát v inom poradí, v akom boli posielané. Vzhľadom na zaručenie takýchto vlastností je však nevyhnutnosťou zo strany implementácie protokolu TCP režírovať komunikáciu a zaisťovať doručovanie správ aj v prípade, že na nižších vrstvách ISO/OSI modelu takéto doručenie zlyhalo. Preto môže byť použitie protokolu TCP nevýhodné, či už z pohľadu množstva prenesených dát

alebo časových odoziev medzi poslaním a doručením, v prípade, pokiaľ nie je nevyhnutné využiť všetky vyššie spomínané výhody takéhoto protokolu.

- **User Datagram Protocol (UDP)** je na rozdiel od protokolu TCP protokol pre nespojovanú komunikáciu určený na posielanie tzv. paketov, ktorých doručenie a poradie doručenia vzhľadom na ostatné pakety voči odosielaniu nie je nijak zaručená. Táto vlastnosť činí protokol UDP veľmi výhodným protokolom pre dáta, kedy je nutné minimalizovať odozvy medzi posielaním a doručením dát a zároveň nie je nutné zaistenie poradia a istoty doručenia dát druhej strane komunikácie. Absencia akýchkoľvek garancií vzhľadom na doručenie dát takisto zvýhodňuje protokol UDP voči protokolu TCP aj v množstve prenesených dát, nakoľko je potrebné režírovať takúto komunikáciu len minimálne.

5 Hra Graviton

Vždy som mal slabosť pre veľkosť a nedosiahnuteľnosť vesmíru, preto som chcel vytvoriť hru, ktorá by mi tento priestor priblížila. Takisto som nechcel vytvoriť konvenčnú hru, v ktorej užívateľ bude interagovať s prostredím ako človek, ale ako vesmírna sila – gravitácia, tak vznikla hra s pracovným názvom Graviton.

Poznámka autora: Momentálny “oficiálny” názov hry je Critical Impact. Názov Graviton bol však s hrou spätý od vzniku samotnej myšlienky hry, teda všade v texte bakalárskej práce budem hru nazývať Graviton.

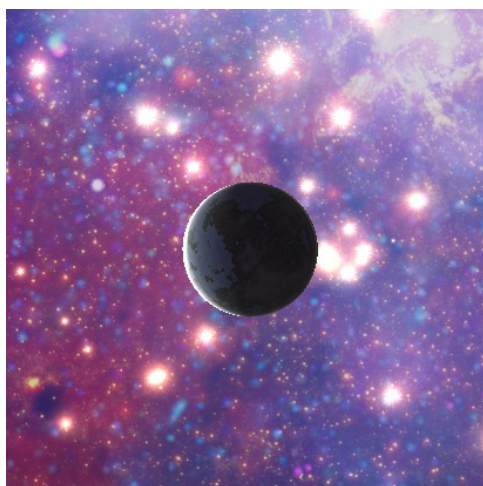
Hra sa odohráva v priestore vesmíru, kde každý hráč má k dispozícii svoju planétu a gravitačné pole, ktorým ovplyvňuje gravitačný priestor okolitého vesmíru. Úlohou každého hráča je udržať existenciu jeho planéty čo najdlhšie a tým zvíťaziť nad ostatnými protihráčmi.

5.1 Úvod do logiky hry

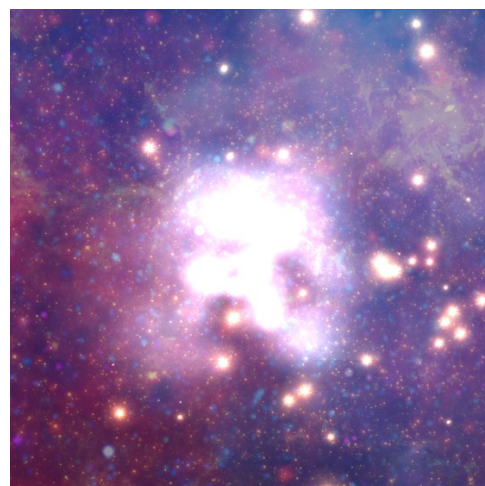
V každej jednej inštancii hry sa nachádza niekoľko základných objektov. Všetky tieto objekty sa nachádzajú na dvojrozmernej ploche, na ktorej spolu interagujú podľa niekoľkých základných pravidiel.

Objekty hry Graviton sú:

1. **Planéty**, ako základné objekty a subjekty rozhodovania o víťazovi danej hry reprezentujú všetku fyzickú hmotu, ktorú môžete v hre nájsť. Plánety majú určitú hmotnosť, príťažlivú silu a podľa nadefinovaných pravidiel a predošlých udalostí nadobúdajú určitú teplotu. V prípade nadmernej teploty v čase postupne túto hmotu spaľujú a vyžarujú do priestoru.
2. **Gravitony**, alebo gravitačné polia, cez ktoré môže hráč interagovať s prostredím hry. Graviton má určenú silu gravitácie, ktorou pôsobí na gravitačné pole. Na rozdiel od reálneho vesírmru však môže graviton okrem príťažlivej sily pôsobiť aj silou odpudivou, ktorá gravitačné pole zakriví presne opačným spôsobom ako sila príťažlivá. Hráč interaguje s prostredím hry pomocou určenia pozície jemu prideleného gravitonu takisto ako s rozhodnutím, akou silou graviton na gravitačné pole pôsobí.



Obrázok 4: Planéta

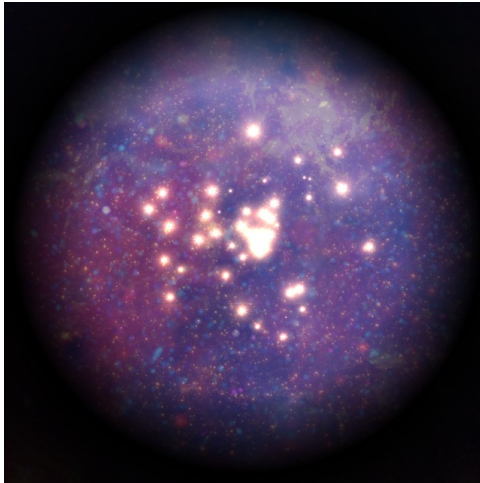


Obrázok 5: Graviton

3. **Hranice herného priestoru (Bounds)**, sú špeciálnym objektom hry, ktorý obmedzuje pohyb planét v rámci hraníc herného priestoru. Každá inštancia hry môže (ale nemusí) obsahovať najviac jednej hranice.
4. **Prekážky (Obstacles)**, sú statické n-uholníky umiestňované v priestore pre rozlčenie priestoru alebo vytvorenie prekážky pre pohyb planét.
5. **Siete červích dier (Wormhole systems)** slúžia na zjednotenie viacerých priechodov červích dier. Planéta, ktorá sa dostane na hranicu priechodu červej diery

v rovnakom momente vyjde z iného náhodne vybraného priechodu červej diery rovnakej siete červích dier.

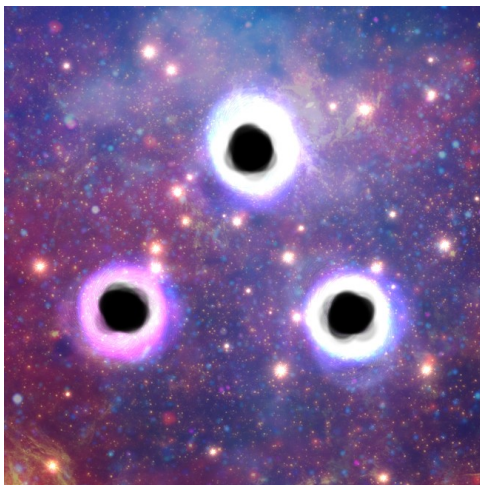
- 6. Priechod červej diery (Wormhole Passage)** je súčasťou konkrétnej siete červích dier.



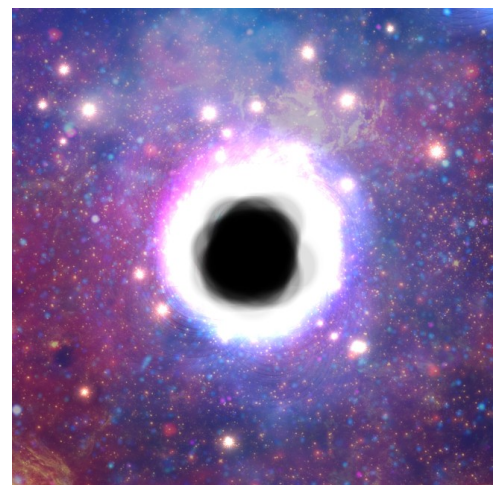
Obrázok 6: Hranice



Obrázok 7: Prekážka



Obrázok 8: Červí systém



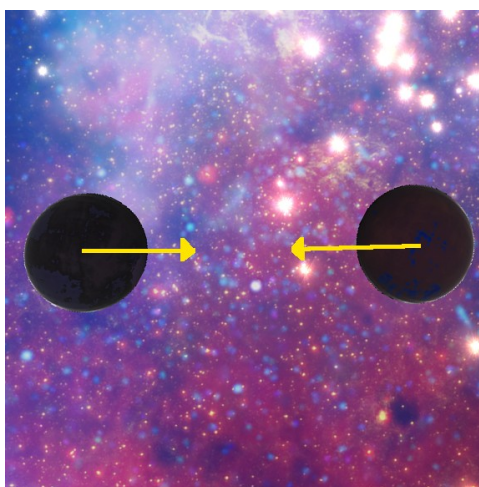
Obrázok 9: Červí priechod

5.2 Pravidlá interakcie objektov v hre

Interakcie medzi hernými objektami sú vždy vyhodnocované iba ako interakcie dvojíc objektov, kde typ interakcie závisí od typu oboch objektov. Ak nie je definované pravidlo medzi určitou dvojicou objektov, objekty na seba nijak herne nevyplývajú.

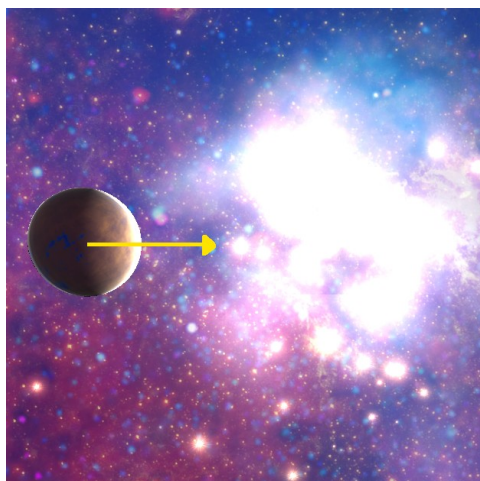
Najkomplikovanejšie pravidlo je pravidlo interakcie **medzi dvoma planétami**. Planéty na seba vplývajú dvoma spôsobmi:

1. **Príťažlivou silou**, ktorá je závislá od ich vzdialenosti a hmotností oboch planét.
2. **Preberaním a odovzdávaním hmotnosti** medzi planétami, pokiaľ medzi planétami došlo ku kolízii. V prípade kolízie medzi planétami odovzdá planéta s menším množstvom hmoty planéte s väčším množstvom hmoty také množstvo, aby už ďalej nedochádzalo ku kolízii. Toto pravidlo, ako je zrejmé, môže vyústiť ku zániku menšej z planét. V prípade kolízie, a teda v prípade odovzdávania hmoty sa takisto medzi planétami odovzdáva (rovnakým smerom) aj teplota hmoty a takisto kinetická energia hmoty.



Obrázok 10: Príťažlivá sila dvoch planét

Medzi planétou a gravitonom je definované pravidlo interakcie pôsobením gravitačnej sily podobne, ako pri interakcii dvoch planét. Planéta na graviton avšak nijak nepôsobí a zároveň gravitačná sila, ktorou pôsobí graviton na planétu nemusí byť len príťažlivá, ale takisto odpudivá. Odpudivá gravitačná sila funguje podobne ako príťažlivá, ale pôsobí na planétu opačným smerom.



Obrázok 11: Pôsobenie gravitonu na planétu

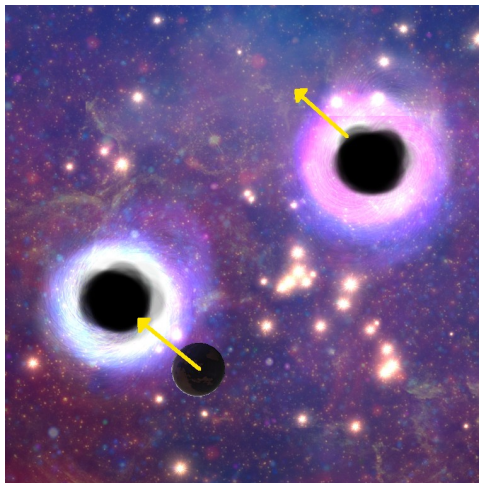
Medzi planétou a hranicami herného priestoru je definované pravidlo, ktorého cieľom je udržať planety v hraniciach herného priestoru. Toto pravidlo je aktívne, ak sa planéta priblíži dostatočne blízko (v lineárnej závislosti od rozsahu hraníc) ku hraniciam herného systému. V takom prípade hranice pôsobia odpudivou silou na planétu, kde veľkosť sily je závislá od “blízkosti” (obrátenej hodnoty vzdialenosti) ku hraniciam. Vektor tejto sily pôsobí takým smerom, aby donútil planétu vzdialiť sa od hraníc smerom ku stredu herného sveta.



Obrázok 12: Pôsobenie hraníc na planétu

Medzi planétou a prekážkou je definované pravidlo odrazu. V prípade, že dôjde ku kolízii planéty a prekážky, planéta je od prekážky odrazená takým spôsobom, aby bola

zachovaná známa rovnosť uhlu dopadu a uhlu odrazu. Odrazenie od prekážky ovplyvňuje planétu jedine zmenou smeru momentálnej rýchlosti, ostatné parametre ostávajú nezmenené a takisto nedochádza k odobratiu kinetickej energie planéty.



Obrázok 14: Prechod červou dierou

Medzi planétou a priechodom červej diery je definované pravidlo skákania do iného prechodu červej diery rovnakého systému červích dier. V momente, kedy sa planéta priblíži na vopred definovanú vzdialenosť od priechodu červej diery, sa vykoná náhodný výber z priechodov červích dier rovnakého systému (rovnomerným rozdelením pravdepodobnosti). Zároveň sa vypočíta relatívna pozícia voči pôvodnému priechodu červej diery. Planéta sa presunie na pozíciu vybraného priechodu červej diery s posunutím o zistenú relatívnu pozíciu voči pôvodnému priechodu červej diery. Pred týmto posunom sa však relatívna pozícia voči pôvodnému priechodu červej diery upraví tak, aby po presunutí planéty nedošlo ku opätovnému presunu medzi tým istým prechodom červej diery.

5.3 Fyzika planéty

Planéta je najkomplikovanejším objektom hry Graviton, nakoľko jej interakcia s okolím je závislá na vonkajších parametroch a tie sú zároveň závislé na vnútorných procesoch prebiehajúcich v planéte:

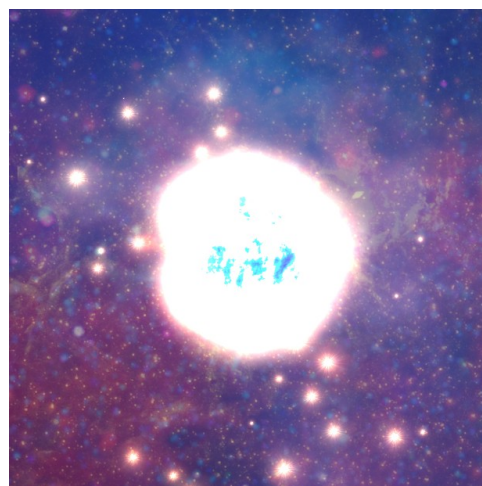
- **Hmotnosť alebo množstvo hmoty** je základným parametrom každej planéty a je takisto rozhodovacím prvkom pre určenie víťaza danej inštancie hry. Hmotnosť

alebo množstvo hmoty planéty sa dá ľahko prirovnať k počtu častíc, z ktorých je planéta zoskupená. Tento parameter je ako jediný vyhodnocovaný pri posudzovaní príťažlivých či odpudivých gravitačných síl.

- **Momentálna teplota planéty** je základným ukazateľom vnútorných procesov prebiehajúcich v planéte. Dá sa ľahko prirovnať ku priemernej miere nestability častíc, z ktorých planéta pozostáva. Zjednodušene povedané, čím vyššia teplota, tým vyššia nestabilita planéty ako takej. Teplota planéty sa udáva v jednotkách K (Gravitonský Kelvin) a jediné, čo má táto jednotka spoločné so stupňami Kelvina ako ich poznáme v štandardizovanom značení je fakt, že obe jednotky nemôžu nadobudnúť hodnoty menšie než 0K a takisto hodnota 0K popisuje absolútny a nedosiahnuteľný klud častíc.
- **Prirodzená teplota planéty** je určená teplota planéty v závislosti od jej hmotnosti. Túto teplotu sa snaží každá planéta dosiahnuť postupným ochladzovaním (vyžarovaním energie do priestoru) alebo otepľovaním (spôsobeným vnútornými kolíziami častíc). Táto teplota sa takisto udáva v jednotkách K.
- **Ochladzovací (resp. otepľovací) efekt** je spôsobený v planéte, ktorej momentálna teplota je vyššia (resp. nižšia) než prirodzená a teda je nutné prebytočnú teplotu v podobe energie vyžiarit' do nekonečného medziplanetárneho priestoru (resp. absorbovať z medzičasticových kolízií). Rýchlosť ochladzovania (resp. otepľovania) v čase je priamo úmerná rozdielu momentálnej a prirodzenej teploty. Z tohto faktu je zrejmé, že sa momentálna teplota priebehom času asymptoticky blíži prirodzenej teplote.



Obrázok 15: Planéta pred horením



Obrázok 16: Planéta počas horenia

- **Efekt horenia** je spôsobený nadmernou momentálnou teplotou planéty. Dá sa prirovnať k voľnému rozpadu častíc planéty na čistú energiu, ktorý je spôsobený nadmernými kolíziami medzi jednotlivými časticami. Planéta začína horieť v prípade, že jej momentálna teplota dosiahla určitú konštantne definovanú hraničnú teplotu. Množstvo vyhorených častíc je priamo úmerné prekročeniu momentálnej teploty nad hraničnú teplotu. Horenie častíc ďalej spôsobuje zvyšovanie teploty vďaka vzniknutej vnútornej energii a teda môže v určitých situáciách vyvolať reťazovú reakciu a tým v krátkom časovom úseku výrazne znížiť celkovú hmotnosť planéty.

5.4 Vyhodnocovanie stavu hernej inštancie

Každá inštancia hry Graviton obsahuje niekoľko (aspoň jedného) hráčov. Hráčom môže byť napríklad užívateľ, ktorý interaguje s hrou cez užívateľské rozhranie hry, alebo takisto umelá inteligencia, ktorá sa snaží simulovať správanie hráča v hre. Každému hráčovi je v hre pridelený graviton a planéta, toto pridelenie sa počas inštancie hry nemení.

Cieľom hráča je eliminovať planéty všetkých protihráčov a zachovať pri tom existenciu svojej planéty.

Tento cieľ je hráč schopný dosiahnuť umiestňovaním svojho prideleného gravitonu na rôzne pozície v hre a určovaním, kedy pôsobí odpudivo, kedy príťažlivo, alebo kedy nepôsobí na gravitačné pole vôbec. Zmeny pozície a pôsobenia gravitonu hráčovi nie sú

nijak obmedzené, ale sila, ktorou graviton pôsobí, je počas celej inštancie hry rovnaká.

Herná inštancia sa považuje za ukončenú, keď nastane moment, kedy:

- existuje posledný hráč, ktorý má existujúcu planétu, za víťaza hernej inštancie sa vtedy považuje tento hráč, alebo,
- zanikli všetky planéty hráčov iných, než hráčov umelých inteligencií, kedy sa za víťaza hernej inštancie považuje ten hráč, ktorý má zo zostávajúcich planét pridelených hráčom planétu s najväčšou hmotnosťou. Tento moment konca bol do hry vložený, nakoľko z pohľadu reálneho hráča nie je zaujímavé čakať na dokončenie hry umelými inteligenciami.

6 Implementácia hry Graviton

Implementáciu hry Graviton, teda praktickú časť mojej bakalárskej práce som programoval z najväčšej časti v jazyku Java, nakoľko poskytoval ideálne prostredie pre prístup ku rozhraniu OpenGL so zachovaním multiplatformovosti implementácie.

Pri vývoji implementácie som použil knižnicu LWJGL (LightWeight Java Graphics Library) zodpovednú za prepojenie jazyka Java a rozhrania OpenGL, vývojové prostredie Eclipse pre zjednodušenie vývoja a spríjemnenie hľadania chýb krokováním.

Implementáciu som rozdelil na niekoľko základných častí:

- **Framework Relativity**, ktorý slúži ako medzičlánok medzi grafickou kartou a zobrazovaním samotných objektov v hre, ako framework užívateľského rozhrania a takisto poskytuje množstvo pomocných tried pre prácu s grafickým rozhraním OpenGL, matematické operácie apod. Samotný framework je nezávislý na implementácii konkrétnej hry a teda sa dá v budúcnosti použiť pre programovanie inej počítačovej hry.
- **Implementáciu hry Graviton**, obsahujúcu logiku hry, užívateľské rozhranie, simulačné jadro pre fyziku.
- **Multiplayer subsystém** zodpovedný za prenášanie objektov cez TCP/IP protokol, ktorý je využívaný implementáciou hry Graviton pre podporu multiplayer.

6.1 Framework Relativity

Nakoľko OpenGL v jazyku Java (prístupné cez knižnicu LWJGL) je stále len sada funkcií a konštánt rozhrania, pre zjednodušenie ďalšieho vývoja hry bolo potrebné vyvinúť medzičlánok, ktorý by zaobalil prácu s objektami na grafike, shadermi, maticami do tzv. POJO (Plain Old Java Objects). Pre uspokojenie takejto potreby som vyvinul framework Relativity, ktorý v svojom základe poskytuje objektový prístup ku vytváraniu objektov na grafike, ich vykresľovaniu, maticovým transformáciám a množstvám doplnkových tried, ktoré slúžia na zjednodušenie vykresľovania zložitých objektov.

Balíky, do ktorých som systematicky rozdelil kód frameworku Relativity, sú nasledovné:

- **sk.nixone.relativity:** Balík obsahujúci podporné triedy frameworku Relativity pre zjednodušenie komunikácie s jadrom frameworku a uľahčenie vykresľovania komplikovaných typov objektov.
- **sk.nixone.relativity.core:** Balík obsahujúci základné jadro pre komunikáciu medzi frameworkom Relativity a vykresľovacím rozhraním OpenGL.
- **sk.nixone.relativity.math:** Balík pre podporu matematických údajových štruktúr a operácií s maticami, vektormi, úsečkami a rovinami.
- **sk.nixone.relativity.sound:** Balík pre podporu zvukových efektov relativity Frameworku.
- **sk.nixone.relativity.test:** Balík pre základné testovanie relativity Frameworku.

Základnou triedou pri vykresľovaní vo frameworku Relativity je trieda sk.nixone.relativity.core.Graphics. Inštancia tejto triedy reprezentuje základné spojenie procesu hry s grafickou kartou cez rozhranie OpenGL. Drvivá väčšina komunikácie s grafickou kartou prebieha práve cez túto inštanciu. Jedinými výnimkami sú niektoré triedy z balíka sk.nixone.relativity.core, ktoré dopĺňajú framework o ďalšie podstatné funkcionality vzhľadom ku rozhraniu OpenGL.

6.1.1 Základný cyklus aplikácie frameworku

Relativity

Pred začatím akéhokoľvek vykresľovania je nutné vytvoriť inštanciu triedy sk.nixone.relativity.core.Screen, ktorá reprezentuje zobrazovaciu plochu pre grafický výstup rozhrania OpenGL. Pre vytvorenie tejto inštancie je nutné zadať titulok okna, rozmery zobrazovacej plochy a ďalšie parametre. Po získaní tohto objektu je možné dodatočne nastaviť niektoré ďalšie parametre vykresľovania, ako je napríklad maximálny počet vykreslených snímkov za sekundu apod. Po ukončení konfigurácie zobrazovacej plochy je nutné súčasne so štartom vykresľovania dodať zobrazovacej ploche objekt tzv. renderer (rozhranie sk.nixone.relativity.Renderer). Nad týmto objektom bude pri každom prekreslení obrazovky zavolaná metóda Renderer.render(Graphics graphics), ktorá je zodpovedná za celé vykreslenie hernej scény.

6.1.2 Objekty na grafickej karte

Vo frameworku Relativity som vytvoril jednoduchý objektový systém na prácu s dátami vertexov a ich usporadúvaním do objektov. Objekt (alebo súbor vertexov) na grafickej karte je vo frameworku Relativity reprezentovaný inštanciou generického objektu sk.nixone.relativity.core.Mesh<V>. Typ V v generickej triede Mesh hovorí o type vertexov, z ktorých je objekt poskladaný. Takýmto spôsobom môžeme grafickej karte (a shaderom) podať o objekte viac informácií, ako len pozíciu a farbu. Vďaka tejto vlastnosti je možné vytvoriť takmer akékoľvek objekty, ktoré sú následne spracovávané vertex a fragment shadermi.

Pre vytvorenie objektu na grafickej karte je nutné vytvoriť inštanciu triedy Mesh<V>. Počas vytvárania (alebo tesne po vytvorení) tejto inštancie je nutné zadať množinu vertexov, alebo vertexových dát, z ktorých je objekt poskladaný. V momente zadefinovania všetkých potrebných vertexov je nutné zavolať metódu Mesh<V>.pack(), ktorá zabezpečí alokáciu a prenesenie vertexových dát na grafickú kartu. Pokiaľ potrebujeme dodatočne počas vykresľovania upravovať vertexové dáta objektu, framework Relativity takúto funkcionality samozrejme podporuje, je však nutné následne volať metódu Mesh<V>.update(), ktorá zabezpečí synchronizovanie vertexových dát z Java

objektov na grafickú kartu.

Framework Relativity v základe poskytuje niekoľko nádstavb triedy Mesh<V>, ktoré zjednodušujú vytváranie jednoduchých pravidelných objektov na grafickej karte, ako sú napríklad štvorec, kruhový výsek z roviny, ale aj zložitejšie objekty, ako napríklad guľa alebo rúra.

6.1.3 Maticové transformácie

Framework Relativity poskytuje zjednodušenie práce s maticovými transformáciami pomocou triedy sk.nixone.relativity.core.ModelTransformation. Bežná situácia pri vykresľovaní objektov hernej scény zahŕňa napríklad stav, kedy vykresľujeme na scénu skupinu objektov, ktorej pozícia je voči svetu scény známa, avšak pohyb objektov prebieha iba relatívne voči skupine takýchto objektov. V takom prípade je ideálne udržiavať transformačnú maticu skupiny voči scéne a takisto transformačnú maticu každého konkrétneho objektu voči skupine. Takýmto spôsobom je možné vytvoriť strom maticových transformácií, kde každá maticová transformácia môže mať odkaz na rodičovskú maticovú transformáciu. Zjednodušuje to situáciu v prípade, keď je takýchto vnorených transformácií viac, a frekvencia zmeny maticovej transformácie nie je rovnaká pri všetkých maticiach. V takomto prípade je možné odkladať medzivýsledky maticových transformácií pre jednotlivé vetvy stromu a teda znížiť počet nutných matematických operácií. Pre tento účel som vyvinul triedu ModelTransformation. Jej hlavnou funkcionalitou je zabezpečiť okamžitú aktuálnosť momentálneho medzivýsledku maticovej transformácie a takisto zabezpečiť jednoduchý prístup k bežným maticovým transformáciám, ako sú napríklad posunutie, rotácia, zväčšenie (resp. zmenšenie) apod.

6.1.4 Texture atlasing

Objekty môžu cez rozhranie OpenGL byť vykresľované s textúrou. Textúra v OpenGL sú obrazové dáta, ktoré môžu byť namapované na objekt (texturovými koordinátami), resp. na jeho vertexy a následne vykreslené tak, akoby miesto povrchu objektu (jeho primitív) boli vykreslené obrazové dáta textúry. Každá textúra môže byť samostatne uložená v pamäti grafickej karty. V niektorých prípadoch je však praktické uložiť viacero rôznych textúr do jedného obrázku a následne adekvátne prepočítat

textúrové koordináty pri vykresľovaní objektov. Takýto prístup sa nazýva atlasing. Je potrebný napríklad v prípade, keď vykresľujeme jeden objekt, ktorý však potrebuje použiť viacero rôznych obrázkov (resp. obrazových dát) súčasne.

Framework Relativity má vbudovanú priamu podporu pre texture atlasing, a teda každá textúra vo frameworku je implementovaná ako zobrazenie cez texture atlasing. Prípad, kedy sa mapuje jeden obrázok na jednu textúru je vo frameworku len špeciálnym prípadom použitia texture atlasingu. Framework má dokonca podporu vnorených textúr, kedy jedna veľká textúra môže byť použitá pre vykreslenie určitého objektu, kdežto časti tejto textúry pomocou atlasingu môžu byť použité ako podkladové textúry pre vykresľovanie iných objektov.

Hlavnou triedou pre podporu textúr a texture atlasingu vo frameworku Relativity je [sk.nixone.relativity.core.Texture](#), ktorá ponúka niekoľko spôsobov, ako zostrojiť jej inštanciu. Jedným zo spôsobov je priame vytvorenie textúry z obrazových dát, ďalším je uvedenie veľkej textúry a mapovania medzi koordinátmi veľkej textúry a koordinátmi nami vytváranej textúry. Týmto spôsobom máme kedykoľvek k dispozícii vytvorenie novej textúry z už existujúcej textúry pomocou atlasingu.

6.1.5 Vykresľovanie textu

Nakoľko v dobe vyvíniu frameworku Relativity nebol dostupný žiaden komplexný framework alebo knižnica pre kombináciu programovacieho jazyka Java a rozhrania OpenGL pre podporu vykresľovania textu cez VBO, musel som túto podporu vyvinúť ako samostatnú súčasť frameworku.

Pri vykresľovaní textu vznikajú dva problémy, ktoré som musel riešiť. Prvým bolo získanie obrazových dát a ich transformácia do textúry pre reprezentáciu všetkých možných znakov, ktoré môžu byť pri vykresľovaní použité. Druhým problémom bolo získanie konkrétnych rozmerov jednotlivých písmen vykresľovaného textu pre zaistenie prirodzeného zobrazenia textu, ktoré napríklad nie je možné pri použití fixného miesta vyhradeného pre vykreslenie akéhokoľvek znaku. Zrejmým riešením tohto problému by bolo použitie fontu, ktorý pre vykreslenie každého znaku potrebuje rovnako veľký priestor (monospaced fonty). Avšak text vykresľovaný takýmto fontom častokrát nepôsobí na užívateľa atraktívne a častokrát nie je ľahko čitateľný. Oba problémy nám pomohol

vyriešiť balík AWT, ktorý je bežne súčasťou Javy, a je taktiež známy ako základ pre často používané GUI rozhranie Swing.

Pred vykresľovaním akéhokoľvek textu sa vytvorí reprezentácia fontu (ako typ fontu sa použije inštancia triedy `java.awt.Font`), ktorým sa text bude vykresľovať. Táto reprezentácia je inštanciou triedy `sk.nixone.relativity.TrueTypeFont`, ktorá v závislosti od typu fontu vytvorí pre všetky možné znaky v pamäti obrazové dáta. Tieto obrazové dáta sa následne prekonvertujú na obrazové dáta textúry, ktorá sa uloží do pamäti grafickej karty.

Pri vykresľovaní sa text rozloží na konkrétne znaky. Tieto znaky sa postupne vykresľujú, kde pre každý znak sa vykreslí adekvátny obdĺžnik, ktorého rozmery sú závislé od rozmerov písmena v danom fonte, ktorým sa vykresľuje. Tento obdĺžnik sa vykresľuje s nadefinovanou textúrou, ktorá je pridelená konkrétnemu znaku. Vždy ďalší znak je vykresľovaný s adekvátnym posunutím vďaka maticovým transformáciám, aby bola dodržaná plynulosť zobrazeného textu.

6.1.6 Časticový systém

Pre vytvorenie rôznych grafických efektov v hre je častokrát potrebné vykresliť v priestore mnoho malých vzájomne nezávislých grafických objektov. Takéto objekty sa vo svete vývoja hier nazývajú častice (anglicky: particles). Častokrát je nutné pre vytvorenie reálne vyzerajúceho efektu (napr. výbuchu, horenia, atď.) vykresliť v jednom momente aj tisíce častíc. Pohyb a prekresľovanie častíc sa môže stať hlavným dôvodom spomalenia vykresľovania scény hry.

Ako riešenie tohto problému som vyvniul časticový systém, ktorý sa snaží eliminovať čo najväčšiu časť úkonov s časticami (pohyb, prekreslenie) na procesore a snaží sa ich preniesť na grafickú kartu, kde môžu byť paralelizované.

Základným kameňom pre efektívne fungovanie časticového systému bolo zaručiť, aby po vytvorení častice a zaradení do vykresľovacieho procesu procesor nemusel zasahovať do akýchkoľvek procesov častice prebiehajúcich počas doby jej existencie. Pre zaručenie tejto vlastnosti je potrebné pri vytvorení častice definovať všetky parametre, ktoré vypovedajú o správaní častice počas doby jej existencie a tieto parametre už procesor počas existencie častice nemôže meniť. Pre zaručenie jednoduchosti vytvárania a

spracovávaní častíc na grafickej karte som zaviedol niekoľko základných obmedzení, ktoré musia spĺňať všetky častice, ktoré sú časticovým systémom vykresľované:

1. **Konštantnosť pohybu:** od počiatku existencie častice musí byť vektor jej pohybu v priestore konštantný, teda sa nesmie počas existencie častice meniť.
2. **Lineárnosť zmeny veľkosti:** je možné definovať veľkosť častice na začiatku a na konci jej existencie, nie je však možné túto veľkosť akokoľvek upravovať počas existencie častice.
3. **Lineárnosť zmeny priehľadnosti:** je možné definovať mieru priehľadnosti na začiatku a na konci existencie častice, nie je však možné túto mieru priehľadnosti akokoľvek upravovať počas existencie častice.
4. **Pevne definovaný čas existencie každej jednej častice:** Každá častica môže mať rozdielny čas začiatku a dobu trvania existencie, tento čas a doba sa však počas existencie častice už nemôže meniť.

Vytvorenie častice pozostáva z definície jej parametrov, ako sú napríklad počiatočná pozícia, vektor pohybu za sekundu, počiatočná a koncová priehľadnosť, počiatočná a koncová veľkosť, textúra, doba existencie apod. Takáto definícia častice je reprezentovaná inštanciou triedy `sk.nixone.relativity.LinearParticleEngine.ParticleBuilder` a je postupne zostavovaná metódami tejto triedy (podľa návrhového vzoru Builder). Definícia je následne vložená do časticového systému, ktorý je reprezentovaný inštanciou triedy `sk.nixone.relativity.LinearParticleEngine`, a od momentu vloženia sa častica považuje za existujúcu. Pri následnom prekreslení časticového systému je častica zaradená do vykresľovacieho procesu ku ostatným časticiam.

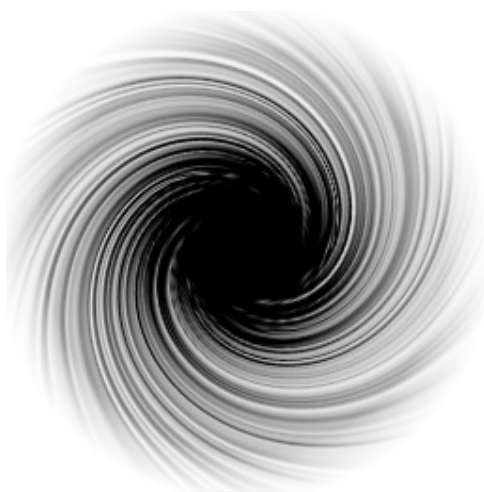
Efektivita časticového systému vzhľadom na veľký počet častíc je výsledkom toho, že od momentu vloženia častice do vykresľovacieho procesu sa s časticou až po moment jej vyradenia z vykresľovacieho procesu (vyradenie je spôsobené vypršaním času existencie častice) na procesore nepracuje.

Implementácia časticového systému zahŕňala niekoľko problémov, ktoré som musel riešiť. Pre efektívne vykreslenie množstva častíc je potrebné vytvoriť objekt na grafickej karte (VBO), ktorý bude reprezentovať určitú časť vykresľovaných častíc (v optimálnom prípade všetky vykresľované častice). Čím väčší počet častíc dokážeme do VBO uložiť,

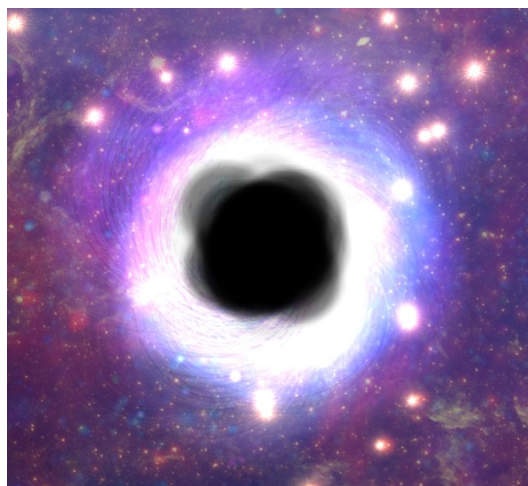
tým zvýšime paralelizovateľnosť vykresľovania jednotlivých častíc. Avšak častice v čase rôzne zanikajú a vznikajú, ich počet sa dynamicky v čase rapídne môže zmeniť a je často veľmi ťažké určiť akékoľvek rozumné ohraničenie počtu častíc. Takisto VBO za bežných podmienok nepodporuje dynamické zmeny v počte vertexov a teda nie je jednoduché do VBO pridávať a odoberať nové vertexy. Z tohto dôvodu bolo potrebné vytvoriť dynamický systém, ktorý bude vytvárať “rozumne veľké” VBO. “Rozumne veľkým” VBO sa v tomto časticovom systéme rozumie dostatočne veľké VBO, aby dokázalo poňať dostatočné množstvo častíc na efektívnu paralelizáciu vykresľovania, avšak dostatočne malé VBO, aby efektívne pracovalo s pamäťovým priestorom grafickej karty.

Počas experimentovania som dospel k relatívne rozumnému nastaveniu počtu častíc pripadajúcich na jeden VBO (1000 častíc / VBO), kedy priemerný počet VBO pri tradičnej hre neprekročil 2.

Nakoľko pohyb častice v čase a zmena jej veľkosti a priehľadnosti nie sú v čase ovládané procesorom, bolo potrebné vytvoriť špeciálny shader na grafickej karte, ktorý bude túto úlohu efektívne vykonávať za procesor. Takto vytvorený shader (umiestnený v adresári hry/res/shaders/particle.vertex) je zodpovedný za všetky úkony, ktoré by vzhľadom na časticu v čase museli byť vykonávané procesorom. V každom prekreslení teda musí vypočítať pozíciu častice (na základe jej počiatočnej pozície, vektoru rýchlosti, začiatočného času existencie a momentálneho času). Takisto musí tento shader zabezpečiť, aby sa nevykresľovali neexistujúce častice, resp. vertexy, ktoré sú alokované na grafickej karte pre častice, ktoré môžu byť do daného VBO ešte vložené.



Obrázok 17: Textúra častice

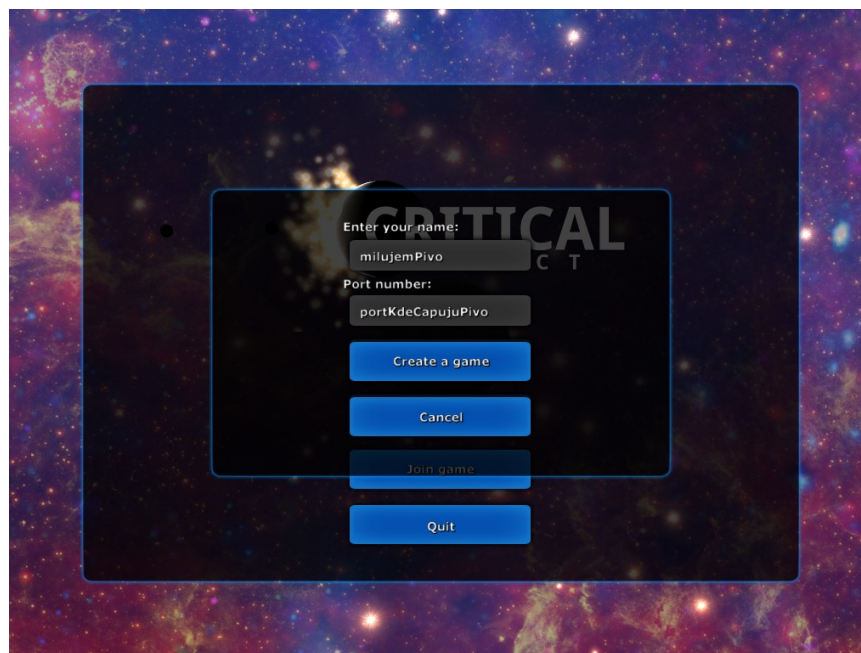


Obrázok 18: Sajúci efekt vytvorený časticovým systémom

Nakoľko sú častice vykresľované ako textúry na štvorcoch, je nutné zabezpečiť, aby v každom prekreslení všetky častice boli otočené správne, a to kolmo na kameru. Takáto metóda otáčania častíc sa nazýva **billboarding**. Pre každú časticu teda musí shader vypočítať tzv. billboardning matrix, teda transformačnú maticu, ktorej úlohou je transformovať štvorec tak, aby bol naklonený kolmo na pozíciu kamery. Ako bolo vyššie v práci spomenuté, celú implementáciu shadera zodpovedného za všetky tieto operácie je možné prezrieť v [adresári hry/res/shaders/particle.vertex](#).

6.2 Grafické užívateľské rozhranie

Nakoľko som nebol schopný nájsť vyhovujúci framework pre vytvorenie užívateľského prostredia v prostredí rozhrania OpenGL, jazyku Java a vykresľovania cez VBO, musel som vyvinuť vlastný framework. Základná štruktúra a komponenty frameworku sa nachádzajú v balíku [sk.nixone.graviton.uifw](#). Pri tvorbe tohto frameworku som sa značne inšpiroval frameworkom Swing pre grafické užívateľské rozhranie v štandardnom balíku Java SE.



Obrázok 19: Ukážka grafického užívateľského rozhrania

Každý komponent grafického užívateľského rozhrania (GUI – Graphical User Interface), ktorý sa nejakým spôsobom zobrazuje, ovplyvňuje iné komponenty alebo je schopný reagovať na vstupy užívateľa je inštanciou alebo dedí inštanciu triedy

sk.nixone.graviton.uifw.Component. Táto trieda implementuje všetku základnú funkcionálnosť každého komponentu, správu veľkosti, správu udalostí apod. Dôležitou časťou každého komponentu je správa pozície v rámci rodičovského komponentu. Takýmto spôsobom je možné nadefinovať konkrétnym komponentom napríklad zarovnanie v ľavom hornom rohu v rámci rodičovského komponentu. Propagáciou rozmerov a zarovnaní sú komponenty schopné dynamicky sa adaptovať veľkosti okna, automaticky upravovať svoju pozíciu apod. Táto schopnosť takisto značne zjednodušuje prácu programátora, nakoľko nie je nútený presne definovať absolútne pozície jednotlivých prvkov a teda len veľmi komplikovane adaptovať rozloženie prvkov rôznym rozmerom zobrazovacej plochy.

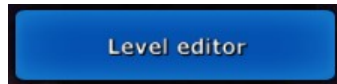
6.2.1 Základné komponenty

Framework pre grafické užívateľské rozhranie obsahuje implementácie niekoľkých základných komponentov, vďaka ktorým je veľmi zjednodušené vytvorenie takmer akéhokoľvek základného grafického užívateľského rozhrania:

- **Kontajner** (trieda sk.nixone.graviton.uifw.Container): Komponent reprezentujúci zoskupenie viacerých ďalších komponentov. Má napríklad schopnosť adaptovať svoju veľkosť v závislosti od rozloženia obsahovaných komponentov ako aj preposielať udalosti, ktoré prijme.
- **Rámec** (trieda sk.nixone.graviton.uifw.Frame): Jednoduchý komponent určený ako základný komponent každého užívateľského rozhrania. Zjednodušuje napojenie na zobrazovaciu plochu a automaticky adaptuje svoju veľkosť v závislosti od veľkosti zobrazovacej plochy.
- **Panel** (trieda sk.nixone.graviton.uifw.Panel): Kontajner určený na grafické zaobalenie ďalších komponentov. Pre svoje vykresľovanie a grafické oddelenie prvkov používa tzv. 9-patch.
- **Vertikálny zoznam** (trieda sk.nixone.graviton.uifw.VerticalList): Kontajner určený na zobrazenie prvkov vertikálne zoradených pod seba. Tento špeciálny kontajner adaptuje svoju veľkosť adekvátne k uloženiu obsahovaných komponentov.
- **Tlačidlo** (trieda sk.nixone.graviton.uifw.Button): Jednoduchý komponent tlačidla,

ktoré má nadefinovaný text a je jednoduché pripojiť obslužný kód stlačenia.

- **Textový vstup** (trieda `sk.nixone.graviton.uifw.Input`): Komponent zodpovedný za získanie jednoduchého textového vstupu od užívateľa.
- **Popisok** (trieda `sk.nixone.graviton.uifw.Label`): Komponent zodpovedný za zobrazenie určitého textu do užívateľského rozhrania. Text a veľkosť textu sa môžu dynamicky meniť.



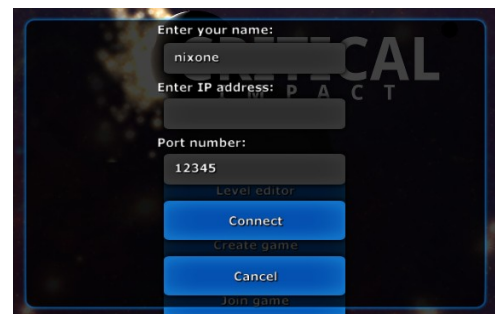
Obrázok 20: Ukážka tlačidla



Obrázok 21: Ukážka textového vstupu



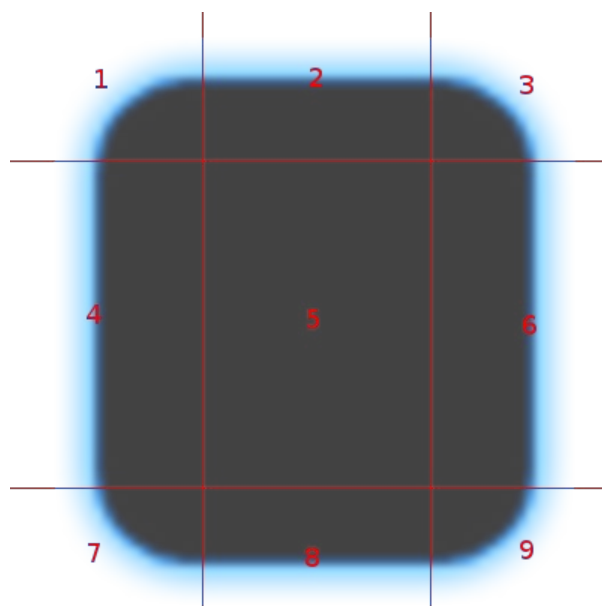
Obrázok 22: Ukážka vertikálneho zoznamu



Obrázok 23: Ukážka panelu

6.2.2 9-Patch

Pre grafické zobrazovanie komponentov s dynamickou veľkosťou sa napríklad v aplikáciách na platforme Android používa takzvaný 9-patch. Na platforme Android sa tým rozumie najčastejšie PNG obrázok s nadbytočným riadkom (resp. stĺpcom) obrazových dát. Tento riadok (resp. stĺpec) označuje možnosť škálovateľnosti daného úseku obrázku po vertikálnej (resp. horizontálnej) osi. Takýmto spôsobom je možné po vytvorení jedného obrázku reprezentujúceho grafické okraje komponentov ich vykresliť bez negatívnych efektov inak vznikajúcich, kedy sa použije rovnaký obrázok pre vykreslenie komponentov rôznych veľkostí.



Obrázok 24: 9-patch obrázok

Ako je možné vidieť na obrázku (9-patch obrázok), časti textúry 1, 3, 7, 9 (rohy vykresľovaného komponentu) sú určené ako nedeformovateľné, teda pri vykreslení akéhokoľvek komponentu ostáva ich veľkosť aj po horizontálnej aj po vertikálnej osi rovnaká. Časti textúry 2 a 8 sú určené ako horizontálne škálovateľné, teda ich šírka sa môže meniť v závislosti od vykresľovaného komponentu, ich výška však vždy ostáva rovnaká. Analogicky, časti textúry 4 a 6 sú určené ako vertikálne škálovateľné, teda ich výška sa môže meniť a šírka ostáva rovnaká. Časť textúry 5, teda telo komponentu, je určená pre vertikálne aj horizontálne škálovanie a teda jej výška a šírka je závislá vždy od veľkosti vykresľovaného komponentu.

V tejto práci som sa inšpiroval týmto systémom a vyvinul som triedu, ktorá je schopná vykresliť obrázok rovnakým spôsobom ako pri systéme 9-patch na platforme Android. Pre takéto vykreslenie je potrebné vytvoriť inštanciu triedy sk.nixone.relativity.NinePatch, ktorá v konštruktoze požaduje textúru obrázku spolu s definíciou škálovateľnosti po osiach vo forme inštancie triedy sk.nixone.relativity.NinePatch.Bounds. Táto trieda dokáže následne vykresliť daný obrázok s akoukoľvek veľkosťou pri zachovaní definovanej škálovateľnosti.

6.3 Umelá inteligencia

Celú implementáciu umelej inteligencie som umiestnil do balíku

Nakoľko sa hra Graviton odohráva v spojitom prostredí, v ktorej aj priebeh hry aj priestor možných akcií každého hráča je spojitý, bolo potrebné vyvinuť dostatočne jednoduchého a efektívneho inteligentného agenta, ktorý bude na druhej strane schopný hrať hru Graviton natoľko inteligentne, aby to bolo pre hráča zábavné.

Jedným z experimentom bolo vytvoriť štvorčekovanú sieť prestretú nad celým priestorom hry, kedy by sa pre každý štvorček tejto siete udržiavali hodnoty získané z herného sveta prislúchajúcemu danému štvorčeku. Agent by sa následne mohol jednoduchšie rozhodovať, nakoľko by jeho rozhodovací priestor už nebol spojitý. Od tohto experimentu som však následne upustil, pretože nebolo možné jednoducho definovať hodnoty, ktoré by sa zo sveta mali získavať a takisto je implementačne náročné tento systém použiť pri neobmedzenom priestore hry.

Nakoľko bolo potrebné vytvoriť efektívneho agenta, ktorý by sa v každom momente hry dokázal rozhodovať a tieto rozhodnutia okamžite v závislosti od stavu hry meniť, bolo potrebné definovať pravidlá pre rozhodovanie a vyhodnocovanie dostatočne jednoducho natoľko, aby bol bežný počítač schopný ich vyhodnotiť aj niekoľko desiatok krát za sekundu.

Agent, ktorý je použitý ako umelá inteligencia v praktickej časti tejto bakalárskej práce sa rozhoduje na základe jednoduchého vyhodnotenia, kde sa nachádza najväčšia predpokladaná hrozba a kde najväčší predpokladaný úžitok pre agenta.

Hrozbou sa pre agenta rozumie akákoľvek planéta, ktorá má schopnosť absorbovať hmotu planéty agenta. Miera takejto hrozby je vyhodnotená na základe vzdialenosti a relatívnej rýchlosti približovania sa k tejto hrozbe. Za najväčšiu hrozbu sa považuje hrozba, ktorej doba do potencionalnej kolízie s planétou agenta je najmenšia.

Úžitkom sa pre agenta rozumie akákoľvek planéta, z ktorej má schopnosť planéta agenta absorbovať hmotu. Miera úžitku je vyhodnotená na základe potencionalne získanej hmoty v pomere ku veľkosti nutnej zmeny rýchlosti, aby došlo ku kolízii. Za úžitok s najväčšou mierou sa teda považuje tá, pre ktorú je potrebné na množstvo hmotnosti najmenej zmeniť trajektóriu planéty agenta.

V každom momente rozhodovania agent vyhodnotí najväčšiu potencionalnu hrozbu

a najväčší potencionálny úžitok. Ak agent vyhodnotí, že miera najväčšej hrozby je dostatočná (podľa stanoveného zostávajúceho času do kolízie), automaticky volí útek pred hrozbou a to zvolením pozície svojho gravitonu tak, aby naviedol svoju planétu mimo kolízny kurz s hrozbou. Ak agent nevyhodnotí dostatočnú mieru najväčšej hrozby, umiestní svoj graviton tak, aby naviedol svoju planétu na kolízny kurz s planétou najväčšieho úžitku.

Jednoduchosť takéhoto rozhodovania znižuje výpočtovú zložitosť rozhodovania agentov a prispieva k možnosti hrania s viacerými agentami umelej inteligencie v jednej hre. Takáto hra sa, na základe niekoľkých experimentov, stáva, aj v prípade takto jednoduchých rozhodovacích pravidiel, veľmi atraktívnou pre hráčov, nakoľko takéto rozhodovacie pravidlá častokrát vyúsťia do situácií a reakcií, ktoré hráč neočakáva.

6.4 Podpora hry multiplayer

Balíky, do ktorých som systematicky rozdelil implementáciu podpory hry multiplayer sú nasledovné:

- **sk.nixone.graviton.game.multiplayer:** Balík zodpovedá za prepojenie samotnej implementácie logiky hry a subsystému pre multiplayer. Obsahuje definície, akým spôsobom sa objekty prenášajú cez sieť, ako sa medzi jednotlivými klientami a serverom posielajú zmeny o svete apod.
- **sk.nixone.messagesystem:** Balík subsystému pre podporu posielania objektov cez protokol TCP/IP.
- **sk.nixone.messagesystem.test:** Balík tried zodpovedných za jednoduché testovanie subsystému pre posielanie objektov cez protokol TCP/IP.

Pri implementácii podpory hry viacerých hráčov (multiplayer) bolo potrebné vyriešiť niekoľko problémov:

Prvým problémom bolo zaistiť **synchornizovanie fyzikálneho sveta a stavu hry** (ďalej len "svet") medzi všetkými hráčmi. Nakoľko sa mohlo stať, že simulácia sveta hry by sa časom u každého hráča rozsynchronizovala iným spôsobom, bolo nutné určiť, svet ktorého hráča bude považovaný za hlavný a takisto zaručiť, aby všetky ostatné svety boli schopné prijať akúkoľvek zmenu hlavného sveta ako svoj nový stav. Tento problém som

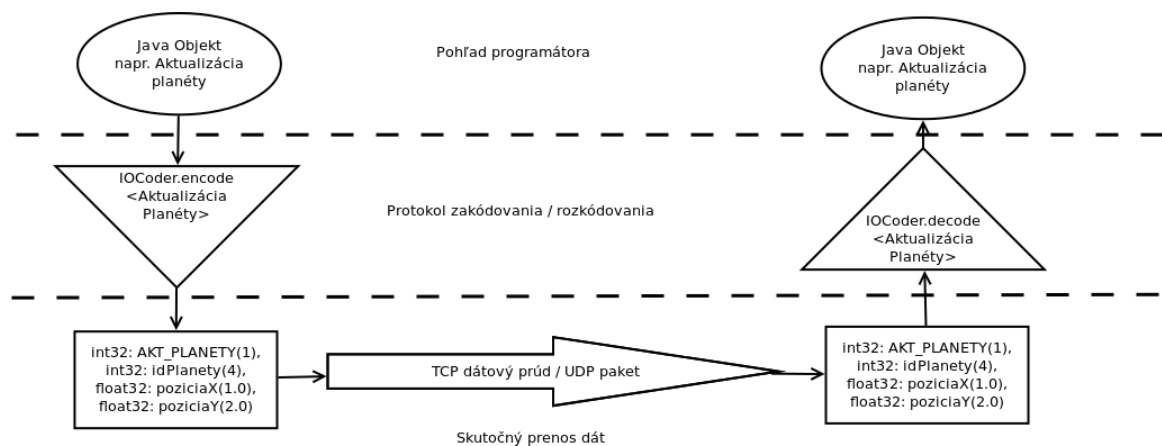
vyriešil vytvorením dedikovaného sveta u hráča, ktorý hru vytvoril. Tento dedikovaný svet následne posiela zmeny stavov sveta cez protokoly TCP a UDP ostatným svetom. Dedikovaný svet prijíma v čase pozície gravitonov konkrétnych hráčov a následne aj tieto pozície posielal ďalším svetom.

Druhým problémom bolo **zaistenie dostatočne rýchleho aktualizovania stavu jednotlivých herných objektov**, aby dochádzalo k čo najmenším odozvám medzi odoslaním údajov o stave herného objektu v dedikovanom svete a jeho spracovaním na strane prijímajúceho sveta. Udalosti, pri ktorých synchronizácii je nutné dodržať správnosť a postupnosť, sú synchronizované správami protokolu TCP, nakoľko nám zaručuje dostatočné garancie doručenia a poradia správ. Nakoľko však vykonáva réžiu pre udržiavanie poradia a úplnosti všetkých správ, vo veľa prípadoch je pomalý a teda nevhodný na aktualizovanie momentálneho stavu herných objektov. Ak by sme totiž dokázali rýchlo posielat' veľa aktualizácií objektov v čase, čiastočná strata alebo porušenie poradia by nepôsobila na synchronizáciu negatívne. Experimentálne som porovnával rýchlosť možnej aktualizácie objektov protokolmi TCP a UDP. Rýchlosť možnej aktualizácie protokolom UDP sa v určitých prípadoch pohybovala až stonásobkom rýchlosti aktualizácie protokolom TCP. Už na príklade s desiatkami aktualizácií je však vidieť výhodnosť využitia protokolu UDP. Pre zážitok z hry je totiž prospešnejšie, ak sa z 10 vyslaných aktualizácií objektu 2 z nich stratia, ale stále doručíme 8 úspešných aktualizácií. Pri protokole TCP by sme síce mali zaručenú správnosť a postupnosť doručenej aktualizácie, boli by sme za ten istý čas poslať však iba jednu miesto ôsmich. Potencionálne zle zoradená aktualizácia by hernému zážitku mohla uškodiť omnoho výraznejšie než nedoručená aktualizácia. S týmto problémom je však jednoduché sa vysporiadať. Postupne priradzujeme a inkrementujeme poradové číslo posielanej aktualizácie. Pri prijímaní aktualizácie jednoducho odignorujeme také aktualizácie, ktorých poradové čísla sú nižšie ako najväčšie poradové číslo doposiaľ prijatej aktualizácie.

Implementácia hry multiplayer pri vytvorení hry cez užívateľské rozhranie na pozadí vytvorí dedikovaný fyzikálny svet hry. Spolu s ním je vytvorený TCP server, ktorý čaká na spojenia od potencionálnych hráčov a UDP socket, ktorý čaká na prichádzajúce UDP pakety od potencionálne pripojených klientov. V momente vytvorenia TCP spojenia server pošle informácie o UDP sockete. Server následne čaká na synchronizačný UDP paket od klienta, aby mohol jasne potvrdiť nadviazanie spojenia. Klient po pripojení na

TCP server cyklicky posiela synchronizačný UDP paket, kým nepotvrdí prijatie synchronizačného UDP paketu potvrdením cez protokol TCP. Takto pripojený klient je pripravený na prenášanie dát aj prostredníctvom protokolu TCP ako aj prostredníctvom protokolu UDP.

TCP protokol je určený na prenos prúdu dát, UDP protokol je určený na prenos balíkov dát (datagramov). Ručné kódovanie aktualizácie svetov a objektov by bolo zbytočne komplikované, preto som vytvoril abstrakčnú vrstvu medzi objektami hry a protokolmi TCP a UDP. Táto abstrakčná vrstva (balíky sk.nixone.messagesystem.*) je schopná prijímať objekty na strane odosielateľa, podľa nadefinovaného protokolu tieto správy zakódovať do prúdu protokolu TCP alebo datagramu protokolu UDP a na strane prijímateľa takto zakódovaný objekt rozkódovať a vytvoriť adekvátnu objektovú reprezentáciu (ideálne identickú s objektovou reprezentáciou odosielateľa), s ktorou programátor môže jednoducho pracovať bez nutnosti riešenia prenosu cez protokoly TCP alebo UDP. Protokol, ktorý definuje spôsob zakódovania a rozkódovania do prúdu alebo datagramu je mapovaný voči triede objektu, ktorý tento protokol dokáže spracovávať a je v implementácii reprezentovaný inštanciou generickej triedy sk.nixone.messagesystem.IOCoder<T>. Typ T v tejto generickej triede definuje triedu



Obrázok 25: Spracovanie objektu abstrakčnou vrstvou

objektu, ktorých inštancie je schopný daný protokol spracovať.

Každý odosielateľ a prijímateľ pevne nadefinuje zobrazenie medzi triedami a protokolmi pre kódovanie a tiež identifikátor, vďaka ktorému sú konkrétne objekty jasne identifikované v prúdoch alebo paketoch dát. Pri odosielaní objektu abstrakčnej vrstve je odosielateľ nútený určiť protokol (TCP alebo UDP), ktorým sa objekt bude odosielať. V

prípade synchronizačných objektov (objekt reprezentujúci začiatok alebo koniec hry, prípadne zoznam hráčov momentálne pripojených v hre, informácie o vzniknutom alebo zaniknutom hernom objekte) sa využíva protokol TCP, pretože je nutné objekt doručiť odosielateľovi a zaručiť správne poradie odosielaných objektov aj za cenu prípadnej odozvy v komunikácii. Ak sa však jedná o objekty reprezentujúce aktuálny stav herného objektu, je možné a taktiež ideálne využiť protokol UDP. Vďaka využitiu protokolu UDP je možné poslať za fixnú jednotku času omnoho väčšie množstvo aktualizácií ako v prípade protokolu TCP. Nakoľko sú jednotlivé aktualizácie herných objektov navrhnuté ako nezávislé, nevzniká problém v prípade nedoručenia niektorých aktualizácií alebo v prípade doručenia aktualizácií v inom poradí než v poradí, ktorom boli odoslané.

Pre prijímanie dát je nutný príjemcať nadefinovať tzv. *handlers* (objekty, ktoré sú zodpovedné za spracovanie prijatých objektov) ku každému typu objektu, ktorý sa pri komunikácii môže vyskytnúť. Okamžite po prijatí, rozkódovaní správy a vytvorení reprezentačného objektu danej správy sa správa posunie prislúchajúcemu handleru danej správy.

6.5 Ostatné balíky implementácie

Súčasťou implementácie hry Graviton boli ďalšie balíky zabezpečujúce podporu pre ostatné systémy ako tiež ďalšie funkcionality, ako sú napríklad simulačné jadro fyziky, podpora pre interaktívny editor herného sveta, apod.

- **sk.nixone:** Balík obsahuje pomocné triedy pre prácu s analytickou geometriou, kolekciami, animácie v abstraktnom ponímaní, prácu s udalosťami, jednoduché nádstavby implicitných vlákien.
- **sk.nixone.graviton.game:** Balík obsahuje samotnú implementáciu veľkej časti logiky hry Graviton, obsahuje takisto triedy a definície zodpovedné za načítavanie a ukladanie definície herného sveta do súborov alebo prúdov.
- **sk.nixone.graviton.game.editor:** Balík obsahuje užívateľský editor zodpovedný za možnosť vytvárania a úpravy herného sveta priamo cez užívateľské rozhranie hry.
- **sk.nixone.graviton.game.space:** Balík obsahuje rôzne typy implementácií gravitačného priestoru pri rôznych presnostiach a zložitostiach.

- **sk.nixone.graviton.input:** Balík obsahuje rôzne typy tried, ktoré reprezentujú spôsoby vnímania vstupov od užívateľa. Jednou z príkladných tried je trieda GravitonController, ktorá prijíma vstupy z myši, a podľa určených pravidiel mení pozíciu gravitonu v závislosti od vstupov užívateľa.
- **sk.nixone.graviton.platform:** Balík obsahujúci všetky základné podporné systémy pre nutný chod aplikácie, od vstupného bodu programu, cez dialógový spúšťač až po základné implementácie rozhraní pre prepojenie s frameworkom Relativity.
- **sk.nixone.graviton.rendering:** Balík obsahujúci triedy, ktoré definujú spôsoby vykreslenia rôznych typov herných objektov hry Graviton ako aj ďalšie triedy, ktoré vykresľujú okolie herného systému, napríklad pozadie alebo častice, z ktorých je tvorená väčšina efektov v tejto implementácii.
- **sk.nixone.graviton.scenes:** Balík obsahujúci rôzne typy scén, s ktorými sa hráč v implementácii môže stretnúť. Je definovaná scéna MainMenuScene, ktorou hra vždy začína, a podľa výberu užívateľa v hlavnom menu sa hra postupne presúva medzi scénami.
- **sk.nixone.monitoring:** Balík obsahujúci doplnky pre monitorovanie času procesoru stráveného na jednotlivých vláknach Java aplikácie.

7 Záver

Cieľom práce bolo navrhnuť a implementovať real-time hru vrátane hernej logiky, navrhnuť a implementovať umelú inteligenciu pre podporu hry proti počítaču, implementovanie podpory hry multiplayer a implementácia hry pomocou knižnice OpenGL.

V prvej časti bakalárskej práce som vysvetlil základy vykresľovania počítačových hier pomocou knižnice OpenGL, popísal základné prvky a procesy vykresľovacieho procesu a takisto teoretické základy, na ktorých stavajú.

V druhej časti práce som popísal navrhnutú a vytvorenú hru Graviton, objekty, ktoré v hre vystupujú a pravidlá, ktoré herný svet ovplyvňujú. Takisto som navrhol systém vyhodnocovania a interakcie hráčov a popísal efekty, ktoré v hre môžu vzniknúť.

V tretej časti práce som sa venoval popisu implementačných detailov. Popísal som mnou vyvinutý framework Relativity, ktorý je zodpovedný za zjednodušenie vykresľovania cez rozhranie OpenGL. Takisto som popísal navrhnutú a implementovanú umelú inteligenciu schopnú imitovať plnohodnotného protihráča a takisto implementáciu multiplayer hry pomocou TCP/IP protokolov transportnej vrstvy TCP a UDP.

Výsledok mojej bakalárskej práce hodnotím úspešne, nakoľko som bol schopný vyvinuť podľa zadania plne funkčnú a zábavnú počítačovú hru, čím je dôkazom aj niekoľko hodín strávených jej hraním počas vývoja nie len na účely experimentovania ale aj zábavy s priateľmi.

Zoznam použitej literatúry

- 1: Stuart Russel, Peter Norvig, Artificial Intelligence - A Modern Approach,
- 2: Silicon Graphics, OpenGLTM Reference Manual, 1994
- 3: Pavel Tišnovský, Grafická knihovna OpenGL (11): vykreslovací řetězec, 2003
- 4: Pavel Tišnovský, Grafická knihovna OpenGL (14): nastavení perspektivní kamery, 2003

Zoznam príloh

- CD médium, skompilovaná implementácia hry Graviton vo formáte JAR archívu