České vysoké učení technické v Praze

Fakulta elektrotechnická

Katedra počítačů

Diplomová práce

Optimalizace síťového provozu pro VRUT

Voříšek Lukáš

Leden 2018

Vedoucí práce: Ing. Kubr Jan

České vysoké učení technické v Praze
Fakulta elektrotechnická

Katedra počítačů

ZADÁNÍ DIPLOMOVÉ PRÁCE

Student: Voříšek Lukáš

Studijní program: Otevřená informatika
Obor: Softwarové inženýrství

Název tématu: Optimalizace síťového provozu pro VRUT

Pokyny pro vypracování:

Virtual Reality Universal Toolkit (VRUT) je univerzální a flexibilní nástroj pro práci s grafickými daty, který je vytvářen ve spolupráci ČVUT, ŠKODA AUTO a.s., MU, MeU, ZCU a VŠB. Nástroj je napsaný modulárním způsobem a umožňuje tak přizpůsobení jednotlivých částí specifickým potřebám. Systém je psaný v jazyce C++.
Prostudujte zapojení VRUT pracoviště ve ŠKODA AUTO a.s. a obvyklé scénáře provozu tohoto pracoviště.
Vytvořte analýzu dostupných možností optimalizovaného přenosu zejména obrazových dat mezi výpočetním clusterem a zobrazovacími uzly. Tato optimalizace bude využívat stávající 10 Gbps linky mezi výpočetním clusterem a zobrazovacími uzly umístěními v Mladé Boleslavi. Na základě analýzy navrhněte a implementujte nástroje pro optimalizaci přenosu. Navrhněte vhodné testy pro otestování implementovaných nástrojů. Tyto testy proveďte a vyhodnoťte.

Seznam odborné literatury:

[1] Václav Kyba. Modulární 3D prohlížeč. Diplomová práce České vysoké učení technické v Praze, Fakulta elektrotechnická, 2008.
[2] Cyril Crassin, David Luebke, Michael Mara, Morgan McGuire, Brent Oster, Peter Shirely, Peter-Pike Sloan, Chris Wyman. CloudLight: A System for Amortizing Indirect Lighting in Real-Time Rendering. Technical Report NVR-2013-001, NVIDIA, 2013.

Vedoucí: Ing. Jan Kubr

Platnost zadání do konce letního semestru 2017/2018

prof. Dr. Michal Pěchouček, MSc.
vedoucí katedry

prof. Ing. Pavel Ripka, CSc.
děkan

V Praze dne 6.2.2017

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 dne 9. 1. 2018

Děkuji panu Ing. Janu Kubrovi za vedení mé diplomové práce a vstřícnost při konzultacích. Také bych rád poděkoval panu Ph.D. Antonínu Míškovi za odborné rady a vstřícné jednání při implementaci změn.

Abstrakt

Cílem této práce je optimalizace datového toku, který je nutný pro distribuovaný výpočet obrazu pomocí aplikace Virtual Reality Universal Toolkit. Tato optimalizace by měla poskytovat možnost výpočtu a zobrazení scény v rámci CAVE projekce a zařízení HTC Vive.

Pro dosažení tohoto cíle bylo navrženo několik možných řešení, které jsou založeny na úpravě architektury komunikace mezi uzly, komprimací a omezením přenášených dat. Dílčí část těchto úprav byla v této práci implementována a popsána ve větším detailu.

Částečná implementace řešení vedla ke snížení datového toku o 37 %, po implementaci ostatních navržených řešení je očekáváno snížení až o 90 %. Při snížení datového toku o 90 %, tedy na 7,25 Gb/s, což je dostačující pro CAVE projekci, ale ne pro zařízení HTC Vive. V rámci implementované optimalizace byla omezena barevná hloubka výsledného obrazu a navržena a implementována komponenta aplikace, jenž distribuuje výpočet na klienty.

Klíčová slova: VRUT, síťová komunikace, vyvažování zátěže, Ray Tracing, komprese dat, ŠKODA AUTO a.s., optimalizace datového toku, distribuovaný výpočet

Abstract

The aim of this thesis is optimisation of data flow which is required for distributed computing of a picture with a use of Virtual Reality Universal Toolkit. This optimisation should provide a possibility of calculation and display a rendered scene under CAVE projection or HTC Vive.

In order to reach the goal, a few possible solutions that were based on the changes of architecture of communication between nodes, compression and reduction of transmitted data, were designed. In order to reach the goal, a few possible solutions that were based on the changes of architecture of communication between nodes, compression and reduction of transmitted data, were designed.

The partial implementation in this thesis leads to 32% reduction of data flow. The rest of proposed changes promises a reduction of data flow by 90 % (7,25 Gb/s), which is sufficient for CAVE projection. However, it does not meet the requirements of HTC Vive devices. Implemented optimalisation was based on a reduction of color depth, a design and implementation of distributing component.

Keywords: VRUT, network communication, load balancing, Ray Tracing, data compression, ŠKODA AUTO a.s., optimisation of network traffic, distributed computing

Obsah

Seznam obrázků

Seznam tabulek

Seznam výpisů

Kapitola 1
Úvod

Tato práce se zabývá analýzou stávající komunikace, návrhem a implementací možných změn vedoucích ke snížení a lepší kontrole datového toku produkovaného aplikací Virtual Reality Universal Toolkit (VRUT) v režimu distribuovaného výpočtu.

VRUT je univerzální a flexibilní nástroj pro práci s grafickými daty, který je vytvářen ve spolupráci ČVUT, ŠKODA AUTO a.s., MU, MeU, ZCU a VŠB. Nástroj je napsaný modulárním způsobem a umožňuje tak přizpůsobení jednotlivých částí specifickým potřebám. Systém je psaný v jazyce C++.

V této práci se budu věnovat převážně datové komunikaci mezi třemi základními uzly VRUTu. Jedná se o *kontrolní uzly*, které řídí stav scény. Tyto uzly například přijímají signály ze vstupních zařízení a transformují tyto signály do akcí scény (přidání objektu, otočení kamery). *Uzly výpočetní*, které počítají obraz scény nebo jeho část a *uzly zobrazovací*, které vypočtenou scénu zobrazují uživateli.

Zadavatel ŠKODA AUTO a.s. chce posunout možnosti zobrazení a jeho kvality tak, aby byl stále schopný konkurence a udržel se ve špičce svého oboru. Pro zobrazení dat proto chce používat náročných grafických algoritmů, včetně sledování paprsků[1] a sledování cest[2]. Jelikož jsou tyto algoritmy náročné na výpočetní výkon, rozhodl se zadavatel ŠKODA AUTO a.s. pro přesun výpočetní části do *clusteru*, který provozuje v jiné budově, než je budova s řídícími a zobrazovacími uzly (momentálně i uzly výpočetními).

Zadavatel se však obává, že datový přenos, jenž bude vyžadovat stávající implementace, bude přesahovat kapacity sítě, které má k dispozici. Pokud by tomu tak bylo, bude nutno upravit modul VRUTu, který data přenáší tak, aby se snížil datový tok, ale zároveň bylo docíleno rychlosti dostačující pro plynulý obraz.

[1] Spíše známo pod anglickým ray tracing
[2] Anglický termín path tracing

1.1 Definice pojmů

Kamera je objekt scény, kterému jsou přiděleny určité vlastnosti (úhel, pozice, ohnisková vzdálenost, ...), a je využíván pro výpočet obrazu scény.

Zobrazovací klient/server zobrazuje vypočtenou scénu uživateli. Také slouží pro načtení scény a její distribuci výpočetním klientům.

Výpočetní klient/klient počítá obraz pro celou a nebo část scény. Nemusí disponovat (v některých případech) grafickým čipem. Výpočetní klient scénu většinou nezobrazuje.

Uzel/node je označení pro zařízení zapojené do clusteru nebo s přístupem do clusteru. Pokud se v textu tedy vyskytuje zobrazovací uzel nebo výpočetní uzel, tak je myšlen výpočetní nebo zobrazovací klient zapojený v rámci clusteru nebo využívající cluster pro výpočet.

Dlaždice/tile Je blok grafických dat pro specifickou část scény. Jinak řečeno dlaždice je částí celkového obrazu scény.

Kapitola 2
Analýza

Jak bylo napsáno v úvodní kapitole, jedním z požadavků je plynulost výsledného obrazu. Po konzultaci se zadavatelem proto uvažuji hodnoty 15, 24, 30 a 60 snímků za sekundu (FPS). 24 snímků za sekundu odpovídá množství snímků používanému ve filmech[1], proto by bylo ideální cílit na tuto hodnotu pro klasický (dvourozměrný) obraz. Hodnoty 60 snímků by bylo vhodné dosahovat při použití 3D technologií (které často vyžadují dvojnásobnou nebo vyšší zobrazovací frekvenci[2]).

Zadavatel má k dispozici několik počítačů umístěných přímo v objektu, které nyní používá jak pro výpočet, tak pro vykreslování scény. Jedná se o osm počítačů, které scénu počítají a vykreslují, a jeden řídící počítač, který přijímá data ze vstupních zařízení a informuje o změnách scény výpočetní klienty.

Kromě těchto počítačů je k dispozici výpočetní cluster, kde je možné zažádat až o 160 uzlů. Tyto uzly ovšem neobsahují grafickou kartu, a tak musí veškeré operace provádět nad CPU.

Technologie a přenosové kapacity sítě jsou popsány v tabulce 1.

Spojení v rámci	Rychlost	Technologie	Poznámka
HPC cluster	56 Gb/s	Infiniband	až 160 uzlů
Master - HPC	10 Gb/s	Ethernet	
VR klienti	1 Gb/s	Ethernet	

Tabulka 1: Kapacita linek

Kromě přenosové kapacity je známo, jaký druh projekce zadavatele zajímá. Výčet v tabulce 3 není úplný, ale zejména CAVE projekce[3] je momentálně upřednostňována, proto se jí bude týkat i většina analýzy.

Pro přenos jedné dlaždice je potřeba přenést (v testovaném případě) 32×32 obrazových bodů, které jsou reprezentovány pomocí typu `Vector4`. Velikost dlaždice ve skutečnosti záleží na zvoleném algoritmu výběru dlaždic. Může se tak jednat i o 8×8 obrazových bodů. `Vector4` je implementován jako čtyři hodnoty datového typu `float`. V jazyce C a C++ je minimální hodnota počtu bitů, které reprezentují datový typ

[3]CAVE je rozhraní virtuální reality, které se skládá ze stěn, stropu a podlahy obklopujících pozorovatele scény. Návrh tohoto rozhraní překonává mnoho problémů, se kterými se potýkají jiné systémy virtuální reality.[8]

`float`, rovna čtyřem bajtům.[20] Jedna dlaždice tak obsahuje:

$$32 \cdot 32 \cdot 4 \cdot 4 = 16384\ B = 16\ \text{kB}$$

Pro zjištění počtu přenesených dlaždic za jeden snímek (tabulka 2) je použit vzorec:

$$\frac{C \cdot T \cdot 128}{S}$$

kde C je kapacita linky v Gb/s, T je doba přenosu v milisekundách a S je velikost přenášených bloků v kB.

FPS	čas přenosu	1 Gb/s	10 Gb/s
15	67 ms	536	5360
24	42 ms	336	3360
30	34 ms	272	2720
60	17 ms	136	1360
100	10 ms	80	800

Tabulka 2: Počet přenesených 16 kB bloků za snímek

Projekce	Rozlišení	Počet
CAVE	2430x2400	8
Powerwall	2048x1080	4

Tabulka 3: Projekce

Dle zadavatele je pro CAVE potřeba přibližně 6600 bloků na jednu stěnu. Tedy celkem 26 400 bloků[4], které je potřeba přenést do 36 (nejhůře 50) ms. V případě HTC Vive[5] je potřeba přibližně 5000 bloků, ale obraz je potřeba vykreslit do 10 milisekund.

U CAVE projekce je potřeba $16\ \text{kB} \cdot 6600 = 105600\ \text{kB} \doteq 103\ \text{MB}$ na stěnu. Při použití čtyř stěn se jedná o $105600\ \text{kB} \cdot 4 = 422400\ \text{kB} = 413\ \text{MB}$ každých 50 milisekund.

Pro plynulý zážitek je potřeba přenést alespoň 24 snímků za sekundu, což odpovídá přibližně 42 ms na snímek. Ve výpočtech jsem vypustil časovou rezervu, která je v provozu potřeba pro zpracování na straně serveru. Během jedné sekundy se tak jedná o $422400\ \text{kB} \cdot 24 = 10137600\ \text{kB} = 9900\ \text{MB} \doteq 10\ \text{GB}$ za sekundu.

Kapacita linky směrem ke clusteru je 10 Gb/s a přenášená data dosahují hodnoty 78 Gb/s. Objem přenášených dat tedy musí být minimálně 8× menší než by byl v současné chvíli.

[4]Při použití pouze přední stěny, obou bočních a podlahy.

[5]HTC Vive je zařízení virtuální reality, které se umisťuje na hlavu pozorovatele. Toto zařízení obsahuje senzory pro monitorování pohybu hlavy hráče, které přenáší do scény, kterou zobrazuje. Zařízení bylo vývojářům představeno v roce 2015. Zobrazovací rozlišení tohoto zařízení je 1080×1200 obrazových bodů na jedno oko. Tedy 2160×1200 obrazových bodů pro celé zařízení. [9]

Kapitola 3
VRUT

VRUT (Virtual Reality Universal Toolkit) je aplikace pro vizualizaci a editaci 3D dat vytvářená ve spolupráci ŠKODA AUTO a.s., ČVUT, MU, MeU, ZCU a VŠB, která si klade za cíl:

1. využití nových technologií,
2. poskytování zvláštní funkcionality,
3. podporu systémů a formátů ŠKODA AUTO a.s.,
4. vysokou rychlost,
5. použitelnost pro výuku a experimenty.

Projekt vznikl v rámci spolupráce katedry počítačové grafiky a interakce ČVUT FEL s firmou ŠKODA AUTO a.s. Jeho podstatou je zobrazení grafických dat a podpora modulů.[34]

V této kapitole popíši, jaké třídy a jakým způsobem jsou využity při síťové komunikaci modulem *Cluster2* ve spolupráci s modulem *RayTracer*.

3.1 Komunikace

Tato práce staví na vyvíjeném modulu *Cluster2*, který je následovníkem modulu *Cluster*. Modul *Cluster* fungoval tak, že server sbíral informace ze vstupních zařízení a změny polohy a scény hlásil klientům. Klienti následovně počítali a zobrazovali scénu. Toto chování odpovídá variantě Serverem řízení klienti z kapitoly 4.1.1 (obrázek 7 na straně 13).

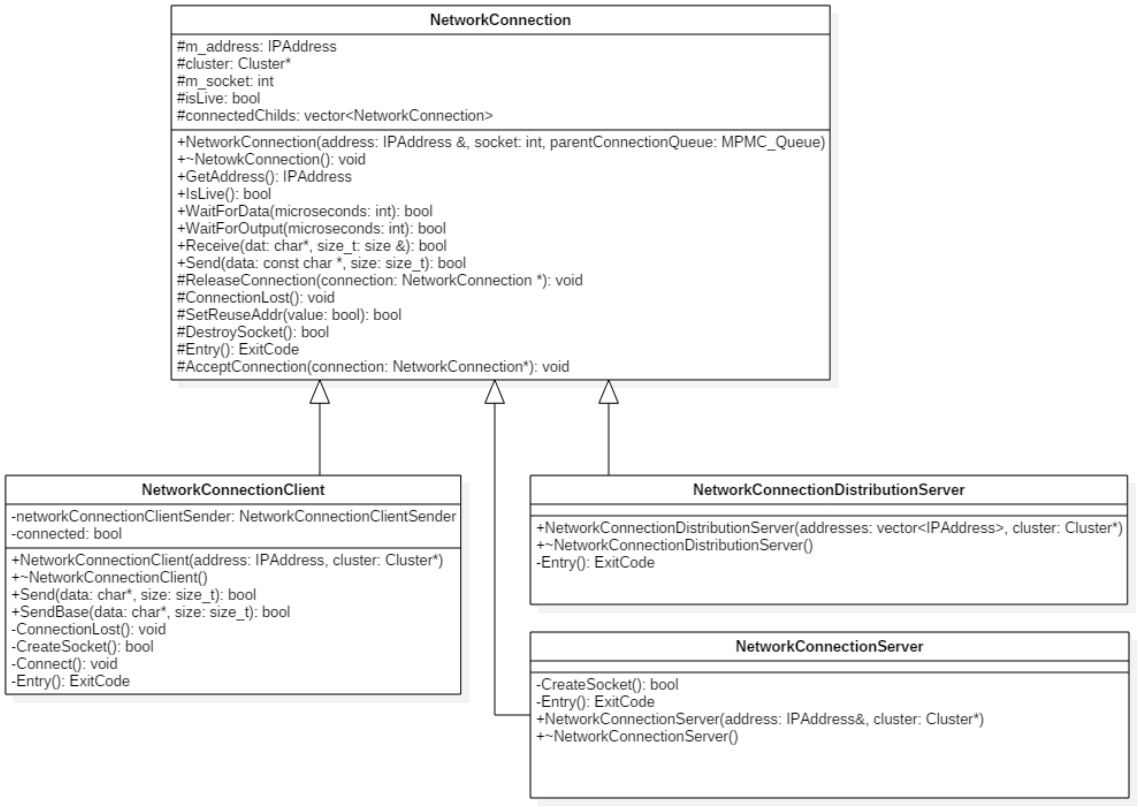
Modul *Cluster2* měl k chování modulu *Cluster* přidat možnost distribuovat výpočet na více počítačů a tím umožnit vyšší rozlišení a obnovovací frekvencí pro dosažení lepšího zážitku z rozšířené reality (CUBE, HTC Vive). Modul by tedy měl fungovat tak, že zvolenou scénu rozdistribuuje na výpočetní uzly a rozdělí jim výpočet scény. Jednotlivé uzly provedou výpočet a vrátí serveru výsledky, ten z vypočtených bloků sestaví výsledný obraz (obrázek 8 na straně 15), který následně zobrazí.

Samotný modul *Cluster2* ovšem nic nezobrazuje ani nepočítá, je pouhým prostředníkem, který se stará o přenos a distribuci dat. Výpočet a zobrazení mohou provádět libovolné kompatibilní moduly, ale v této práci se věnuji pouze vyvíjenému modulu *RayTracer*, pro který je cluster navrhován.

3.1.1 Modul Cluster2

Cluster2 musí přenášet scénu, její změny, požadavky na vykreslení i výsledná data (vykreslené části obrazu). K tomuto účelu byla v modulu zřízena množina tříd, které tuto funkcionalitu zprostředkovávají. Třídy a jejich hierarchie jsou zobrazeny na obrázku 1.

Základní třídou je třída `NetworkConnection`, která poskytuje funkcionalitu pro navázání spojení, udržování spojení, odeslání dat, příjem dat a získání údajů o spojení. Od této třídy se odvozují třídy `NetworkConnectionClient`, `NetworkConnectionServer` a `NetworkConnectionDistributionServer`, které její funkcionalitu rozšiřují a modifikují.



Obrázek 1: Diagram tříd Network Connection

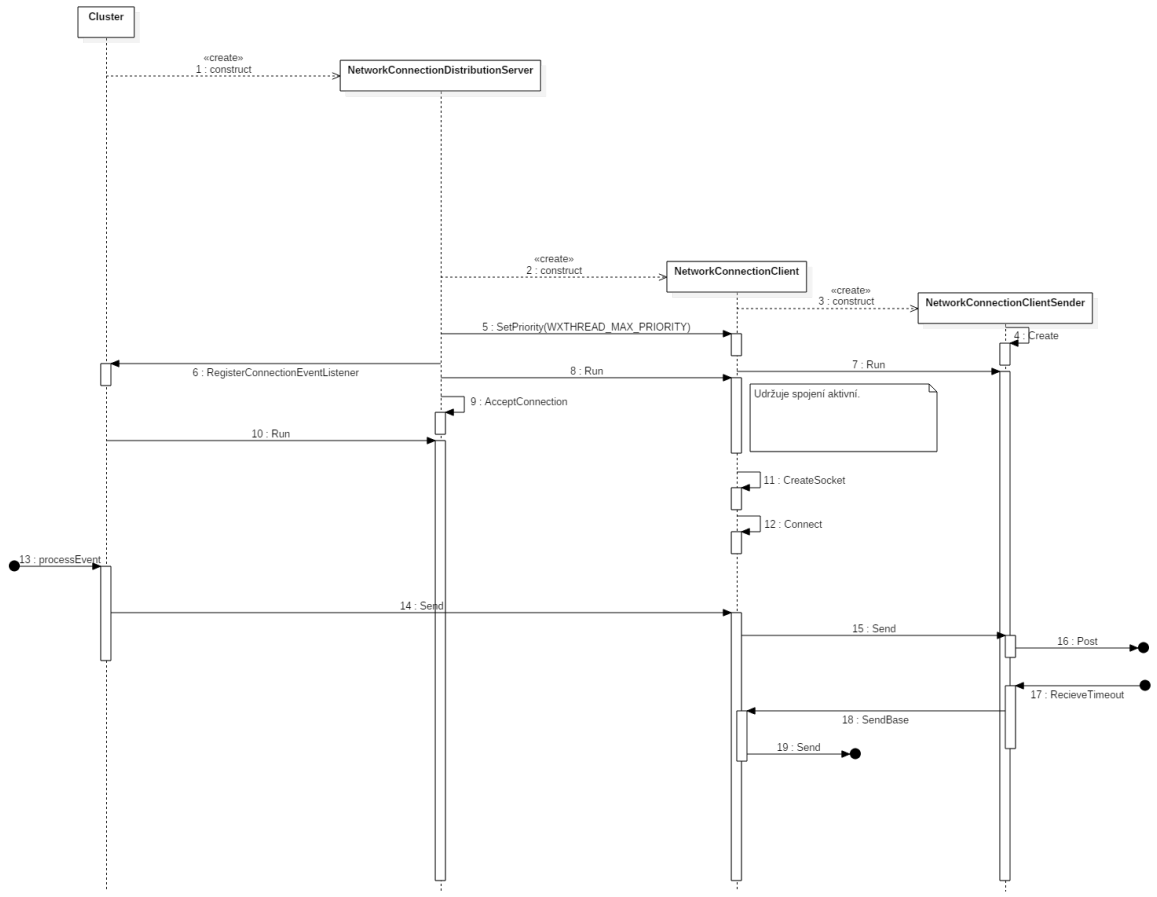
Cluster2 využívá instancí těchto tříd k zasílání a přijímání informací. Inicializace klientů a zaslání zprávy přijaté modulem z jádra je znázorněna na obrázku 2. Vidíme, že nejprve probíhá inicializace modulu. V této fázi se z konfiguračního souboru[6] načte seznam adres výpočetních klientů, se kterými bude server komunikovat, a jejich nastavení. Seznam těchto adres se předá konstruktoru třídy `NetworkConnectionDistributionServer`, který pro každého výpočetního klienta vytvoří instanci třídy `NetworkConnectionClient` popisující informace o spojení a přihlásí se k odběru událostí, které modul umí přenést ke klientům.

[6]Ve výchozím nastavení se jedná o soubor ClusterConfig.xml

`NetworkConnectionClient` dále zodpovídá za udržování spojení s klientem, odesílání a příjem zpráv v rámci popisovaného spojení. Pro udržování spojení a příjem zpráv je vyhrazeno samostatné vlákno, které je spuštěno metodou `Run`.

Příjem zpráv probíhá tak, že jedno vlákno čte vstupní proud a přijaté zprávy řadí do fronty. Druhé vlákno čte tuto frontu, a pokud v ní je dostupná nějaká položka, tak ji zpracuje. Konkrétně pomocí volání metody `ReadItem` třídy `VRUTDeSerializer` je přečtena jedna položka, která je deserializována a zpracována jako událost.

Odesílání zpráv začíná v momentě přijetí události (metoda `processEvent`). Zde se rozhodne o typu události a serializují se potřebné údaje (`VRUTSerializer`). Tyto serializované údaje jsou zapsány do byte bufferu, který je použit jako argument metody `Send` pro každého připojeného klienta. `NetworkConnectionClient` následně zavolá metodu `Send` na třídě `NetworkConnectionClientSender`, která provádí samotné odesílání.

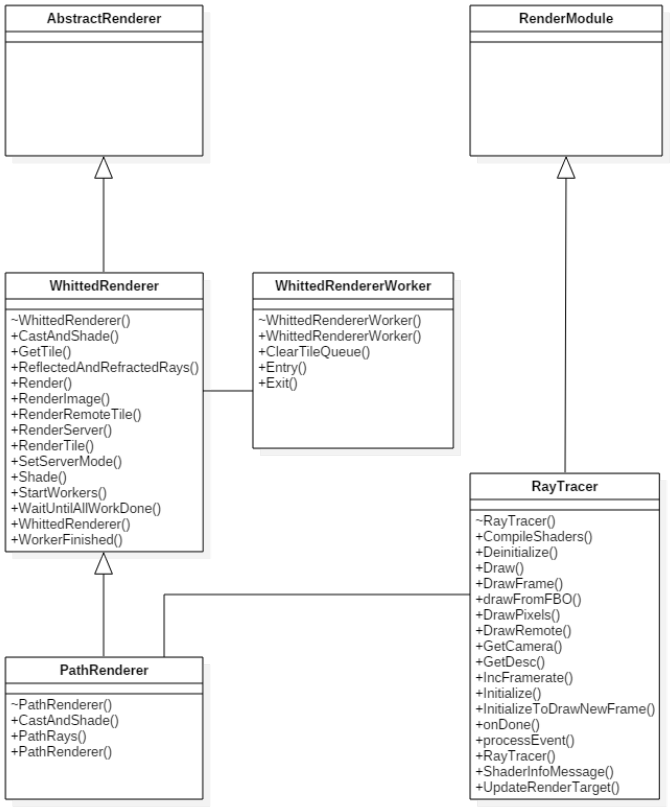


Obrázek 2: Odeslání dat Cluster2 (mód serveru)

3.1.2 Modul RayTracer

V této práci byla řešena pouze spolupráce modulu *Cluster2* s modulem *RayTracer*. Modul *RayTracer* zprostředkovává možnost vykreslovat scénu pomocí vrhání paprsků. Vrháním paprsků chce ŠKODA AUTO a.s. docílit lepší kvality obrazu než při použití OpenGL. Data přenesená přes síť jsou deserializována a převedena na událost (popsáno výše).

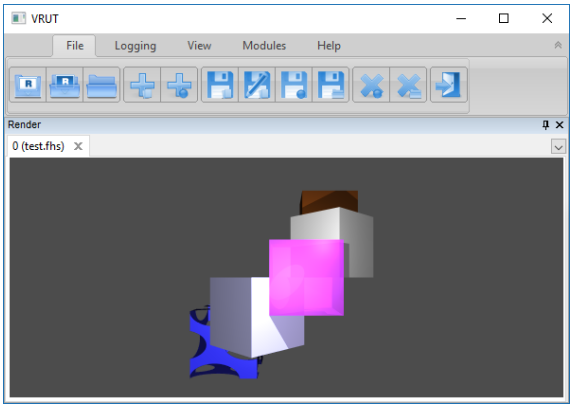
Modul *RayTracer* se skládá z množství tříd, které spolu navzájem komunikují. Na obrázku 3 jsou zobrazeny pouze základní třídy, které se přímo týkají tvorby a zpracování požadavků na výpočet a vykreslení scény.



Obrázek 3: Základní třídy modulu RayTracer

Základní třídou modulu *RayTracer* je třída `RayTracer`. Tato třída umí zpracovávat události z jádra a distribuovat je mezi jednotlivé komponenty modulu *RayTracer*. Také je potomkem (implementací) třídy `RenderModule`, což zajišťuje možnost vykreslovat výsledný obraz do hlavního okna aplikace (obrázek 4).

Pokud je zobrazeno hlavní okno (VRUT pracuje v grafickém režimu) a je aktivní oblast pro vykreslení dat, tak jsou na instanci třídy `PathRenderer` zasílány požadavky na výpočet dané scény. Oblast, která je zachycena kamerou, je rozdělena na menší části (v analýze na straně 3 jsem používal dlaždice o velikosti 32x32 obrazových bodů), které jsou postupně předávány výpočetním vláknům. Každé výpočetní vlákno počítá v danou chvíli přesně jednu takovouto oblast – dlaždici. Každá vypočtená dlaždice je vykreslena na pozici, která jí náleží (v celkovém obrazu scény z pohledu zvolené kamery).



Obrázek 4: Hlavní okno aplikace VRUT

Při použití modulu *Cluster2* je důležité nastavit modulu *RayTracer*, kolik dlaždic může přenechat jiným modulům. V případě, že je tato hodnota větší než-li nula, je místo přímého výpočtu dlaždic zaslána jádru událost **Event::EVT_RENDER_TILE**, která obsahuje informace o dlaždici.

Modul *RayTracer* umí spolupracovat s modulem *Cluster2* pomocí standardizovaných událostí aplikace VRUT. Pro výpočet obrazových dat používá zejména události **Event::EVT_RENDER_TILE** a **Event::EVT_RENDER_TILE_DONE**[7].

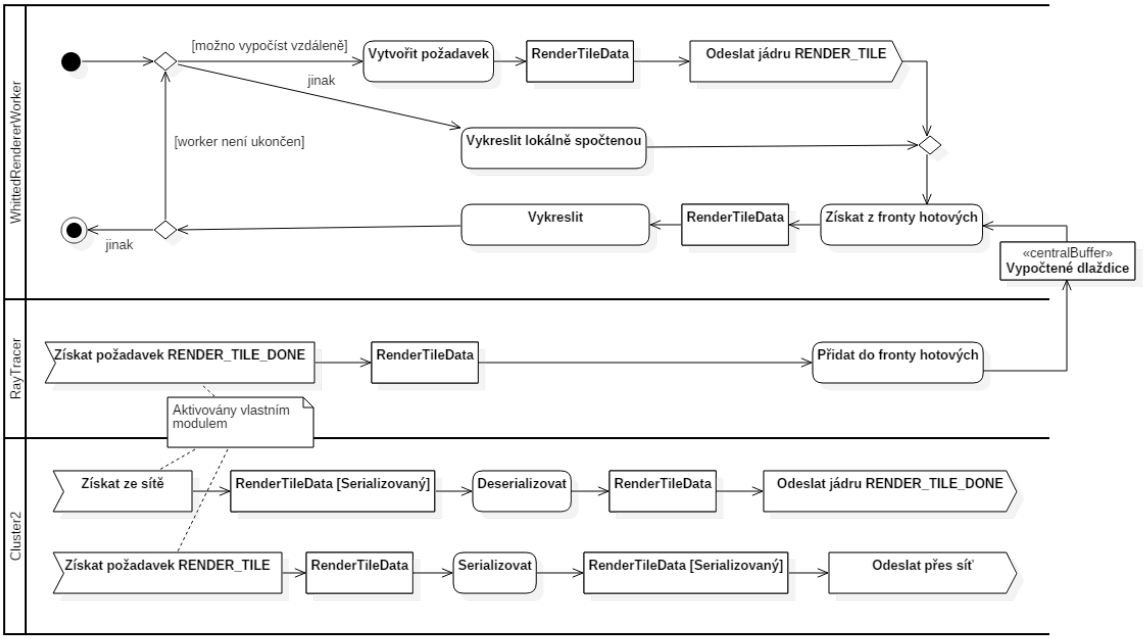
Modul *RayTracer* zašle jádru událost **Event::EVT_RENDER_TILE** v případě, že žádá o výpočet dlaždice jiný modul. Jádro aplikace následně tuto událost rozešle všem modulům, které jsou přihlášeny k jejímu odběru. Mezi moduly, které událost odebírají spadá i modul *Cluster2* (strana 6). Touto událostí dává modul *RayTracer* možnost jinému modulu zpracovat podmnožinu zobrazovaných dlaždic. Respektive každá takováto událost nese jednu žádost o zpracování obsahující informace o dlaždici, kameře a rámci, pro kterou má být obraz vypočten.

Event::EVT_RENDER_TILE_DONE je událost, která informuje modul, který je přihlášen k jejímu odběru, o tom, že dlaždice byla vypočtena. *RayTracer* zašle tuto zprávu jádru v případě, že spočte jednu z přidělených dlaždic. Tuto událost přijme (mimo jiné) modul *Cluster2* a odešle ji všem členům clusteru. Na straně příjemce instance třídy `VRUTDeSerializer` zpracuje zprávu z bufferu a deserializuje ji jako událost **Event::EVT_RENDER_TILE_DONE**, která je odebírána modulem *RayTracer*. Modul *RayTracer* ji zařadí do fronty vypočtených dlaždic, které čekají na vykreslení.

V jednom z vláken, které počítají scénu, se pak tyto vypočtené dlaždice přečtou a přenesou do sdíleného prostoru obrazových dat, který je následně vykreslen uživateli.

[7]Množina událostí podporovaných aplikací VRUT byla během práce na tomto projektu rozšířena zadavatelem o událost **Event::EVT_RENDER_TILE_DONE_SYNC**, která je téměř shodná s událostí **Event::EVT_RENDER_TILE_DONE**. Rozdíl mezi těmito událostmi je ten, že **Event::EVT_RENDER_TILE_DONE_SYNC** vyžaduje běh v hlavním vlákně aplikace VRUT a obchází tak některé standardní cesty zpracování událostí. Ve většině případů jsou tyto dvě zprávy zaměnitelné, proto v textu budu používat jen událost **Event::EVT_RENDER_TILE_DONE**.

3.1.3 Průběh komunikace



Obrázek 5: Zobrazovací server

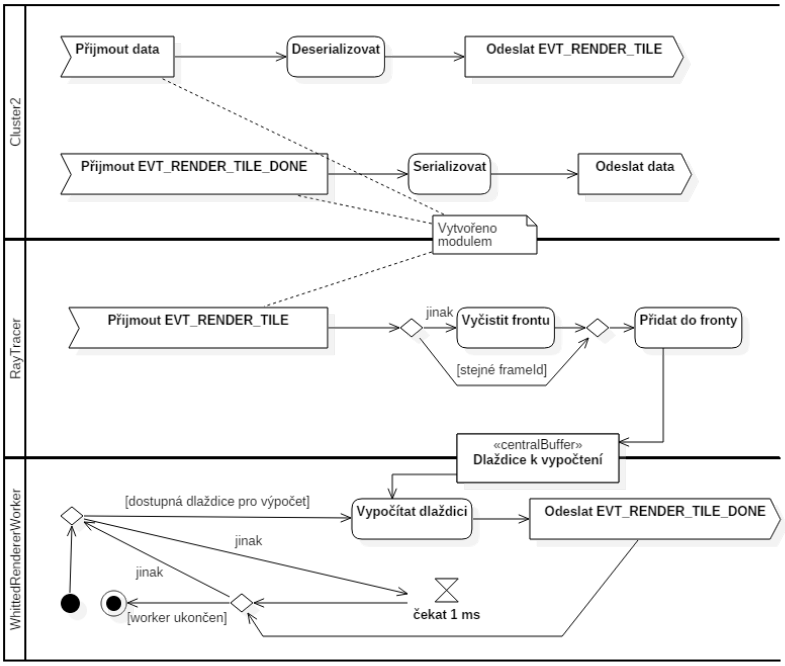
Na obrázku 5 je znázorněno, jak probíhá zobrazení scény v jednom z výpočetních vláken zobrazovacího serveru (Master).

Nejprve se tedy zjistí, zda je možné vytvořit požadavek na vzdálený výpočet; toto rozhodnutí se řídí nastavením modulu.

- Pokud *není možné* vytvořit požadavek na vzdálený výpočet, je dlaždice vykreslena a zobrazena lokálně. Následně jsou vykresleny dlaždice, které byly vypočteny vzdáleně v některém z předchozích kroků.
- Pokud *je možné* požadavek na vzdálený výpočet vytvořit, je vytvořen objekt třídy `RenderTileData`, který je nositelem informace potřebné k výpočtu konkrétní dlaždice a je součástí nově vytvořené události **Event::EVT_RENDER_TILE**. K odběru této události je přihlášen modul *Cluster2* a při jejím přijetí ji serializuje a přenese přes síť klientům (popsáno v kapitole Cluster2 na straně 6).

V případě, že je použita možnost vzdáleného výpočtu, výpočetní klienti (obrázek 6) provedou deserializaci a výpočet dlaždic, a zašlou serveru zpět serializovanou událost **Event::EVT_RENDER_TILE_DONE**. Modul *Cluster2* data deserializuje pomocí instance třídy `VRUTDeSerializer`u a deserializovanou událost předá jádru VRUTu ke zpracování.

Modul *RayTracer* jako odběratel události **Event::EVT_RENDER_TILE_DONE** zařadí vypočtenou dlaždici do fronty, kterou zpracovávají vykreslovací vlákna. Libovolné běžící



Obrázek 6: Výpočetní klient

vykreslovací vlákno získá z fronty objekt typu `RenderTileData`, který nyní již obsahuje data vypočteného bloku, a vykreslí jej.

Kapitola 4
Možná řešení

V této kapitole se budu zabývat možným řešením snížení a kontroly datového toku v rámci clusteru a podrobněji popíši navrhovanou komunikaci mezi výpočetními a zobrazovacími uzly. Kapitola je rozdělena do dvou hlavních částí, kdy jedna se zabývá architekturou komunikace, ve které navrhuji a porovnávám různé přístupy výpočtu a přenosu obrazu v rámci clusteru a přenosu informací (scéna, rotace, vypočtené zobrazení) směrem ke clusteru a zpět. Druhá část se pak zabývá možnostmi snížení datového toku při komunikaci v navržených architekturách, zejména pomocí komprese obrazových dat.

Kontrola datového toku je jedna z hlavních částí této práce. V současné době totiž žádná kontrola datového toku neexistuje (alespoň pro modul Cluster2) a zobrazovací klient dokáže pracovat pouze s jedním výpočetním uzlem. Pokud je výpočetních uzlů více, tak "soutěží"o to, který úlohu vyřeší dříve.

4.1 Architektura

Návrhy architektury komunikace vycházejí z ověřených a obecně známých architektonických návrhu mezi něž patří: samostatné stanice, klient-server (tlustý i tenký klient), peer-to-peer, multi-tier, broker.[3, 4]

Také bylo uvažováno fungování Gaming as a Service (GaaS)[5, 6, 7], které má velmi podobné požadavky na přenos obrazu a rychlost odezvy jako zadavatel.

Gaming as a Service (GaaS) se rozmohl zejména díky pokročilým technologiím v cloudových technologiích. V nejjednodušší formě se jedná o vzdálené vykreslování a ovládání interaktivních her a přenos obrazu zpět ke hráči pomocí internetu. Tento princip fungování je výhodný pro méně výkonné zařízení, které jsou jinak (díky svému malému výpočetnímu výkonu) neschopné spustit moderní hry. Provozovatel GaaS musí dbát na to, aby dosáhl co nejlepší odezvy systému (latence) při co největší obrazové kvalitě s využitím běžného připojení k internetu. Průměrná rychlost připojení k internetu (downstream) v České Republice je v současné době (2017) 33 Mb/s.[10]

Při rychlosti 10 Mb/s byli autoři článku *Cloud Gaming: Architecture and Performance* [7] schopni dosáhnout obrazové kvality PSNR: 33.85 dB a SSIM: 0.94[8]. Vybraný poskytovatel Onlive používal kódování H.264 s hardwarovou podporou a proprietární verzi protokolu *Real Time Transport Protocol* (RTP). Testována byla hra s rozlišením 1280×720 obrazových bodů (720p) a uzamčeným FPS na 30 snímků za sekundu.

[8]PSNR a SSIM jsou vysvětleny v kapitole 5.5 Porovnávané metriky na straně 22.

4.1.1 Serverem řízení klienti

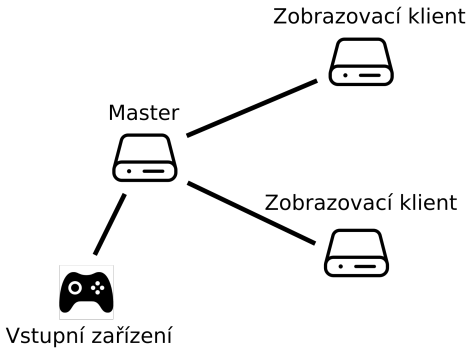
Aktuální stav, který je využíván v produkčním prostředí firmou ŠKODA AUTO a.s., je založen na přítomnosti *serveru*, také označovaného jako master, a osmi *vizualizačních klientů*.

Server sbírá informace z trackovacích zařízení a jiných vstupních zdrojů, na základě kterých modifikuje scénu. Tyto modifikace scény (například posun kamery) distribuuje *klientům*, kteří provádějí výpočet a zobrazení scény nad přidělenou kamerou[9]. Každá stěna CAVE projekce má vlastní server pro výpočet a zobrazení (respektive každá stěna má dva klienty a každý klient počítá její polovinu). (Obrázek 7)

Klient počítá obraz pro svoji kameru a jakmile je obraz připravený k vykreslení, zašle zprávu *serveru*. Při přijetí zprávy od všech klientů dá server příkaz k překreslení obrazu (synchronizace vykreslení obrazu je nutná i proto, aby všechny kamery vykreslovali stejný rámec[10]).

Jedná se o nejjednodušší variantu, která je výhodná v případě, že máme scénu, kterou jsme schopni vykreslit v dostatečném rozlišení a s dostatečnou obnovovací frekvencí[11] na jedné pracovní stanici, a nebo při rozdělení kamery na více pracovních stanicích s přibližně stejným výpočetním výkonem.

Výhodou této varianty je velmi rychlá odezva na akce pozorovatele a téměř zanedbatelný síťový provoz v době projekce[12].



Obrázek 7: Serverem řízení klienti

[9] Kamera je definována v části Definice pojmů na straně 2.

[10] Anglicky frame. VRUT používá pro identifikaci rámce číselnou hodnotu a členskou proměnnou *frameId*.

[11] V případě zadané CAVE projekce tedy 2430×2400 obrazových bodů s obnovovací frekvencí nejméně 24 snímků za sekundu.

[12] Po přenesení scény klientům posíláme pouze informace o změnách ve scéně. Nepřenáší se obrazová data.

4.1.2 Serverem řízení vyvažovaní klienti

Server načte scénu a pošle ji klientům. Následně vytvoří několik kamer (objektů scény), které distribuuje klientům. Kameru pro operátora a kameru pro každého *vizualizačního klienta*. Ty následně slouží k zobrazování v CAVE projekci. Dále zpracovává vstupy ze senzorů, které snímají například pozici hlavy, a jiných vstupních zařízení (myš, klávesnice, ...) a případné změny scény, které následně posílá vizualizačním klientům. Server sleduje vytížení klientů, a pokud není nějaký klient vytížen, může požádat jiného (méně vytíženého) klienta o pomoc s výpočtem části obrazu.

Klient počítá obraz pro svoji kameru a jakmile je obraz připravený k vykreslení, zašle zprávu *serveru*. Při přijetí zprávy od všech klientů dá server příkaz k překreslení obrazu (synchronizace vykreslení obrazu je nutná i proto, aby všechny kamery vykreslovali stejný rámec).

Oproti variantě *serverem řízení klienti* zde zastává server navíc funkci monitorovacího systému, který umožňuje instruovat klienta, aby část svého výpočtu předal jinému klientu. Tato funkce urychluje vykreslování scén, které nejsou výpočetně optimálně rozložené. Příkladem takové scény by mohla být například chodba sídla u moře, kde se pozorovatel pohybuje podél prosklené stěny s výhledem na moře. Část obrazu (dejme tomu vlevo), která je zabrána touto prosklenou stěnou, je o mnoho výpočetně náročnější než část s čistou stěnou s malým odrazem. V takovém případě by po spočtení pravé strany mohl server nařídit klientům počítajícím levou část scény, aby část svých nevypočtených dlaždic přiřadili k výpočtu klientům počítajícím pravou (výpočetně méně náročnou) část scény.

Tato varianta je také výhodná v případě, kdy máme množinu počítačů, v níž je pár hardwarově méně výkonných kusů. Pak můžeme například v CAVE projekci promítat obraz na čtyři stěny s využitím dvou výkonných a dvou méně výkonných počítačů. Dlaždice méně výkonných strojů jsou z části počítány na strojích s lepší konfigurací.

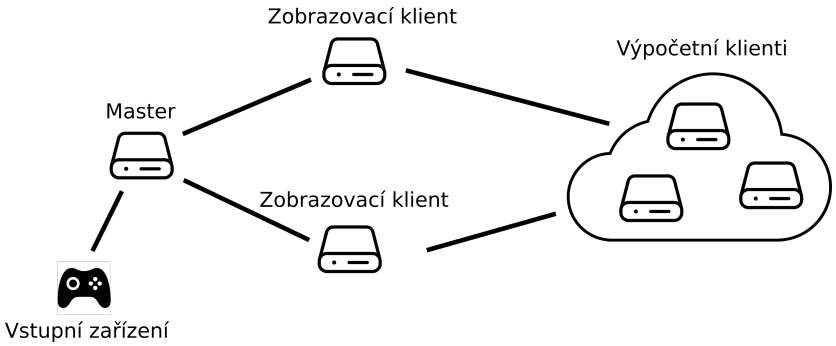
4.1.3 Distribuovaný výpočet

Server načte scénu a pošle ji klientům v clusteru. Následně vytvoří jednu kameru a zpracovává informace ze senzorů (například pohyb hlavy) a jiných vstupních zařízení (myš, klávesnice, ...). Také rozdělí obraz na bloky (v analýze na straně 3 jsem použil bloky o velikosti 32x32 pixelů), tyto bloky rozešle výpočetním klientům k výpočtu.

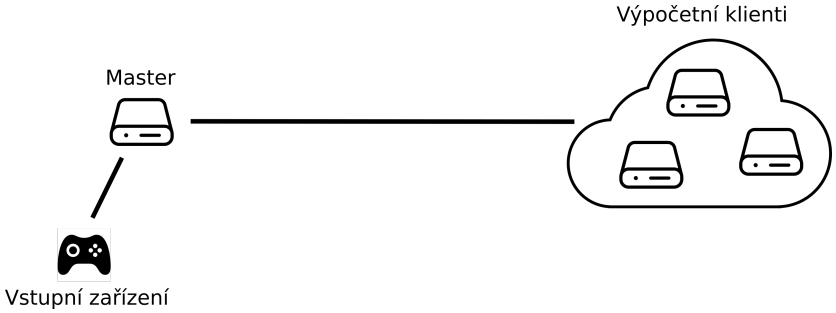
V tomto případě je server hlavním členem řídícím výpočet a zobrazování (oproti předchozím případům, kdy za výpočet a zobrazování zodpovídal klient). Server nejen rozdělí scénu do bloků, ale také zodpovídá za distribuci těchto bloků klientům. Sbírá statistické údaje o klientech a na základě těchto údajů optimalizuje rozdělení mezi dostupné klienty.

Popsané chování odpovídá obrázku 8b, ale snadno jej modifikujeme předřazením masteru na chování znázorněné na obrázku 8a, kde master informuje server o tom, jakou scénu má načíst a k jakým změnám ve scéně dochází. Tyto informace pak již zpracovává

server.



(a) Cluster se zobrazovacími klienty



(b) Cluster bez zobrazovacích klientů

Obrázek 8: Výpočetní cluster

4.1.4 Peer-to-peer

Server je ekvivalentní *klientu* pouze s tím rozdílem, že je u něj aktivní grafické rozhraní. Kterýkoli uzel (tedy ať server nebo klient) může vytvořit, distribuovat a nebo modifikovat scénu. Každou provedenou modifikaci rozešle všem členům své skupiny.

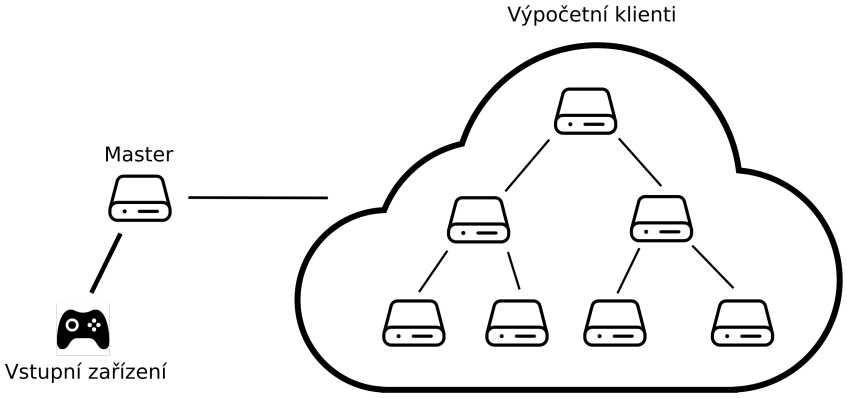
Všechny uzly se podílejí na výpočtu a zároveň o sobě umí poskytnout údaje ostatním členům skupiny. Mezi údaje poskytované ostatním uzlům patří identifikace uzlu, identifikace skupuny, průměrná doba zpracování, počet dlaždic čekajících na zpracování, počet zpracovaných dlaždic a počet dlaždic, který je uzel ochoten přijmout k výpočtu.

Ostatní členské uzly se na základě těchto informací mohou rozhodnout, zda některému jinému uzlu předají některou z dlaždic, jenž si alokovali pro sebe, a nebo zažádat o zaslání vypočteného bloku.

4.1.5 Výpočet ve více vrstvách

Prozatím se vždy dlaždice počítaly lokálně a nebo byly distribuovány na nějaký přímo připojený uzel, který je počítal. Výpočet ve více vrstvách spočívá v postupném dělení scény na menší bloky a jejich následném sestavení. Podle implementace a konfigurace

pak může například server žádat o výpočet čtyři jiné uzly clusteru, které přidělené dlaždice rozdělí mezi své „potomky“ (Obrázek 9). Potomkem jsou pak míněny uzly, které jsou nakonfigurované jako výpočetní uzly pro daný uzel.



Obrázek 9: Výpočet ve více vrstvách

Tato architektura poskytuje několik základních možností distribuce dlaždic.

- Server rozdělí scénu na předem definované dlaždice (například 32×32 obrazových bodů) a ty distribuuje výpočetním klientům. Tito výpočetní klienti vezmou část (nebo všechny) sobě přidělených bloků a zažádají o výpočet své „potomky“.

  V tomto případě je server zodpovědný za rozdělení celé scény na bloky. Dle zvolené distribuční metody může být možné sestavit z vypočtených bloků dílčí obraz scény při zpětném průchodu stromu vzniklého z výpočetních uzlů.
- Server rozdělí scénu na předem definované dlaždice (například 256×256 obrazových bodů) a ty distribuuje výpočetním klientům. Každý výpočetní klient, který není listem stromu výpočetních klientů, rozdělí takto přijatou dlaždici na dlaždice menší (například tedy 64×64 obrazových bodů). Během průchodu stromem se budou dlaždice nadále zmenšovat až na minimální rozměry (například 8×8 obrazových bodů). Pokud má výpočetní klient potomky a přijatá dlaždice již dosáhla minimálních rozměrů, již pouze distribuuje jemu přidělené dlaždice na potomky. Po dokončení výpočtu dlaždice výpočetním klientem odešle klient událost obsahující vypočtená data svému předku. Pokud předek předal klientu k výpočtu dlaždici větší, než klient vypočetl a nebo obdržel vypočtenou, je klient povinen nejprve sestavit dlaždici o požadované velikosti z vypočtených bloků.

  Výhodou této varianty je, že režije spojená s dělením dlaždic je rozložena na více členů clusteru. Zároveň je možné při zpětném průchodu stromu (od určité úrovně[13]) sestavovat dílčí obraz scény. Mimo jiné tato varianta umožňuje aplikovat kompresi obrazových dat na dlaždice specifické velikosti (například jen na dlaždice větší než 64×64 obrazových bodů).

[13] Záleží na nastavení velikosti dlaždic a jejich dělení.

4.1.6 Kombinace lokálního a distribuovaného výpočtu

Výše uvedené architektury se dají do určité míry kombinovat. Kombinace lokálního a distribuovaného výpočtu je kombinací *serverem řízených klientů* (strana 13) a *distribuovaného výpočtu* (strana 14) popřípadě *výpočtu ve více vrstvách* (strana 14), která umožňuje serveru, jenž řídí distribuci dlaždic, přiřadit některé dlaždice k lokálnímu výpočtu. Díky údajům o výpočetních klientech a distribuci dlaždic je pak server schopen rozhodnout o poměru lokálních a vzdálených dlaždic a navíc může optimalizovat distribuci tak, aby posledních několik dlaždic (dle rychlosti výpočtu) bylo vždy počítáno lokálně, a tak minimalizovat zpoždění vykreslení obrazu v důsledku latence na síti a režije síťové komunikace.

4.2 Snížení datového toku

V následujícím textu bude jako *master* označovat počítač, který řídí zobrazování, komunikuje s periferiemi a obraz vypočtené scény zprostředkovává uživateli. Jako *klient* bude chápán člen clusteru, který bude provádět výpočet obrazu nebo jeho části. *Cloud master*, pokud je přítomen, je pak jeden z členů clusteru, který má funkci sběru obrazových dat z klientů a jejich sloučení do výsledného obrazu.

4.2.1 Master - Cloud master

V tomto případě probíhá snížení datového toku na lince mezi clusterem a zobrazovacím serverem. *Master* zašle kameru, scénu a požadavek na vykreslení na *cloud mastera*, který rozdistribuuje scénu *klientům* a vytvoří požadavky na vykreslení dílčích částí scény (bloků), které následně rozdělí mezi klienty.

Vzhledem k tomu, že *cloud master* disponuje již vykreslenou scénou, je možné použít běžné kodeky pro kompresi videa, a tak výrazně snížit datový tok mezi *cloud masterem* a *masterem*. Veškerá datově náročná komunikace (výměna dlaždic a požadavků na vykreslení), která navíc musí mít co nejmenší odezvu, probíhá pouze v rámci clusteru.

Nevýhodou tohoto řešení je, že vzniká úzké hrdlo v podobě *cloud mastera*. Pokud nebude kapacitně schopen distribuovat dlaždice, přijímat a distribuovat události ve scéně, sestavovat obraz z vypočtených bloků a provádět jeho kompresi, tak interakce bude velmi pomalá a v nejhorším případě může přestat reagovat úplně.

Pokud by hardware počítače v roli *cloud mastera* nestačil, je ještě možné zařadit za sebe dva počítače, které se budou tvářit jako jeden jediný. Kdy první (vstupní bod do clusteru zvenčí), nazvěme jej kompresní, by všechny požadavky přeposílal na skutečného *cloud mastera* a jediná jeho funkce by byla komprimovat výsledný obraz. *Cloud master* by pak neměl aktivovánu kompresi obrazu a pouze by výsledný obraz přeposílal na kompresní počítač.

4.2.2 Master - Client

V tomto případě *Master* rozdělí výpočet na menší bloky a jednotlivé bloky přiřadí *klientům* v clusteru. Každý z klientů provede výpočet bloků, které mu byly přiřazeny, a zašle je zpět *Masteru*, který výsledný obraz vykreslí. Každý výpočetní klient je tedy spojen přímo s *masterem* a veškerá komunikace probíhá mezi nimi.

Komprimován je každý blok samostatně. Tedy *Master* musí dekomprimovat každý doručený blok a ze všech vypočtených bloků musí sestavit výsledný obraz.

Hlavní nevýhodou tohoto přístupu je, že kapacita sítě směrem ke clusteru je výrazně nižší než-li kapacita sítě v rámci clusteru (tabulka 1 na straně 3). Například samotná distribuce velké scény může v případě 160 uzlů clusteru trvat velmi dlouho, jelikož je nutno odeslat scénu pomocí 10 Gb/s linky každému uzlu.

Hlavní výhodou tohoto přístupu je, že distribuci dlaždic řídí samotný *master*, který je spojen s každým uzlem clusteru samostatně. Díky tomu může být výpočet prováděn v rámci dvou oddělených clusterů, kdy jediným společným bodem bude *master*. *Master* v takovém případě rozdělí dlaždice mezi uzly dvou clusterů nezávisle na jejich lokalitě. Benefitem pak je možnost použít několik velmi výkonných lokálních počítačů a připojit je do clusteru.

4.2.3 Kombinace předchozích dvou

Kombinace předchozích možností nabízí úsporu datového toku v rámci clusteru i při komunikaci s ním. *Master* bude komunikovat s *Cloud masterem* pomocí komprese podporované na obou uzlech a *Cloud master* bude komunikovat s *klienty*, kteří mu zašlou komprimované dlaždice.

Optimalizací tohoto přístupu lze předpokládat významné snížení datového toku. Při správném přístupu může *Cloud master* distribuovat výpočet na klienty tak, aby výsledné bloky pouhým spojením do většího celku převedl na výsledný (komprimovaný) obraz. K tomu ovšem musí být bloky kódovány takovým způsobem, aby snadno porovnal, zda došlo v bloku k výrazné změně.

4.2.4 Omezení přenášené informace

Kromě využití bezztrátové komprese lze využít i kompresе ztrátové, kdy je omezeno množství přenášené informace na nějakou její podmnožinu. Například omezením barevné hloubky je snížen počet bitů potřebných k přenesení takto omezené barvy.

Také lze použít znalost systému a upravit síťovou komunikaci tak, aby byla přenášena pouze data, která jsou v daný okamžik opravdu potřeba. Do této kategorie patří například snížení počtu přenášených pixelů v prvních několika vykresleních scény. Implementace vrhání paprsků použitá v aplikaci VRUT totiž nejprve vrhá (dle nastavení) omezené množství paprsků na dlaždici a postupně jejich počet zvyšuje. Tedy při prvním výpočtu

jsou pro každou dlaždici například vrženy jen čtyři paprsky, při druhém průchodu osm paprsků a tak dále.

Omezení přenášené informace vede ke ztrátě informace či přesnosti jako takové a nelze ji zrekonstruovat na straně příjemce. Kompresi a omezení přenášených informací se budou věnovat později v samostatných kapitolách.

4.3 Zvolená architektura

Jako optimální se jeví architektura *Kombinace lokálního a distribuovaného výpočtu* ze strany 17 s možností (avšak ne nutností) využití *výpočtu ve více vrstvách* pro některé a nebo všechny uzly.

Toto řešení umožňuje efektivně komunikovat s clusterem za použití co nejmenšího vytížení spojení mezi řídícím počítačem a clusterem. Síť v rámci clusteru je pak využita k distribuci všech potřebných informací mezi klienty. Po vypočtení obrazu může být použit algoritmus využívající změn ve scéně k přenosu pouze změněných bloků.

Řídící uzel (který zobrazuje vykreslenou scénu) by pak nemusel mít vůbec spuštěn modul *RayTracer* ani vědět o jeho existenci. Scénu by mohl vykreslovat pomocí nového modulu, který by přenášel komprimované video. Nyní se také nabízí možnost použít zcela jiného zobrazovacího klienta, který je schopen zasílat řídící události (načtení scény, otočení kamery, ...).

Zadavatel ŠKODA AUTO a.s. ovšem napřed chce implementovat a otestovat pouze dílčí část této architektury, kterou následně rozšíří. Tato dílčí část je ekvivalentní *kombinaci lokálního a distribuovaného výpočtu* s možností využití *výpočtu ve více vrstvách*. Prozatím tedy nebude dostupný *cluster master*, který by koordinoval jednotlivé uzly a komprimoval výsledný obraz. Bohužel tak v této fázi neočekávám významné snížení přenášených dat mezi řídícím počítačem a clusterem.

Kapitola 5
Komprese videa

V této části se zaměřím na porovnání kodeků pro přenos videa. Cílem je zjistit, zda kodeky, které jsou v současné době používané a rozšířené, vyhovují potřebám zadavatele a jejich nároky na výpočetní výkon neomezují jejich nasazení v interaktivních aplikacích.

Abych zajistil objektivní podmínky, byl pro všechny kodeky použit stejný úsek zdrojového videa a stejný program. Implementace kodeků často využívá hardwarové podpory, a proto jsem použil prostředí virtuálního stroje s povolenou přímou komunikací s hardwarem a VT-x/AMD-V. Výhodou virtuálního stroje bylo přiřazení zdrojů tak, aby všechny kodeky disponovaly stejnými prostředky v době porovnávání.

5.1 Testovací prostředí

K testování výkonnosti jednotlivých kodeků byl použit virtuální stroj provozovaný v prostředí Oracle Virtual Box 5.1.14[14] s operačním systémem Debian 8.7.1[15].

Operační systém	Debian 8.7.1
RAM	4096 MB
Procesor	2x s PAE/NX
VirtualBox akcelerace	VT-x/AMD-V, Přímá komunikace s HW
Video paměť	16 MB
Počet monitorů	1

[14] https://www.virtualbox.org
[15] http://www.debian.org

5.2 Porovnávané kodeky

V tabulce 21 jsou uvedeny kodeky, které jsem porovnával, a spolu s nimi argumenty aplikace `ffmpeg`, která zpracování videa prováděla.

H.264 (Bezztrátový)	ffmpeg -c:v libx264 -preset ultrafast -crf 0
H.264 (Ztrátový, CRF 18)	ffmpeg -c:v libx264 -preset ultrafast -crf 18
H.264 (Ztrátový, CRF 23)	ffmpeg -c:v libx264 -preset ultrafast -crf 23
H.264 (Ztrátový s maximálním BR)	ffmpeg -c:v libx264 -crf 20 -maxrate 400k
VP9 (Bezztrátový)	ffmpeg -c:v libvpx-vp9 -lossless 1
VP9 (Ztrátový)	ffmpeg -c:v libvpx-vp9 -crf 18 -b:v 0
MPEG-4 (Vysoká kvalita)	ffmpeg -c:v libxvid -qscale:v 1
MPEG-4 (Střední kvalita)	ffmpeg -c:v libxvid -qscale:v 15

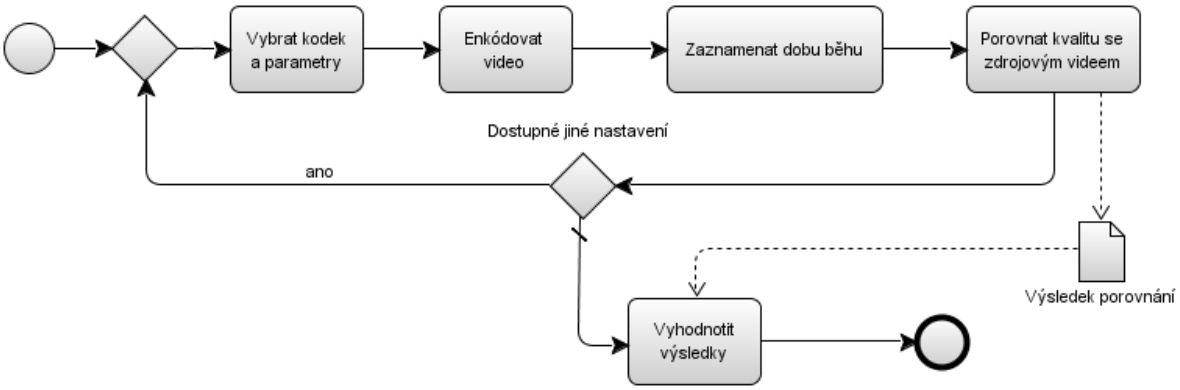
Tabulka 4: Porovnávané kodeky

5.3 Proces porovnávání

U kodeků byla měřena rychlost zpracování (kódování) videa a jeho výsledná kvalita, tedy rozdíl od originálního souboru. Pro porovnání kvality bude použit nástroj MSU Video Quality Measurement Tool[16], který nabízí různé metody a metriky (budou popsány v kapitole Porovnávané metriky na straně 22) pro porovnání kódovaného videa s originálním.

Proces porovnávání je znázorněn na obrázku 10. Nejprve je vybrán kodek a jeho parametry (jeden z tabulky 4). Na základě těchto údajů je spuštěn program `ffmpeg`, který provede kódování videa. Po skončení programu `ffmpeg` je zaznamenáno, jak dlouho kódování videa probíhalo, a pomocí programu MSU Video Quality Measurement Tool je provedeno porovnání výsledného videa se zdrojovým. Výsledky jsou zaznamenány a pokračuje se s jiným nastavením (kodek a parametry). Pokud již není dostupné jiné nastavení, jsou zaznamenané hodnoty vyhodnoceny.

[16] http://www.compression.ru/video/quality_measure/video_measurement_tool.html



Obrázek 10: Proces porovnávání

5.4 Originální soubor

Originálním zdrojem videa je patnáctiminutová pasáž filmu ve formátu RGBA (8 bitů na složku) s 25 snímky za sekundu v rozlišení 1920x1080 obrazových bodů.

Porovnání jsem nemohl uskutečnit na videu, které by mělo srovnatelné vlastnosti s požadovanými zadavatelem, neboť nemám zařízení, jenž by bylo schopno zaznamenat obraz o podobné kvalitě a obnovovací frekvenci, a nepodařilo se mi získat takové video ani z volně dostupných zdrojů.

Hodnotu pro originální video s hodnotami, které používá ŠKODA AUTO a.s., lze však snadno spočítat. Dáno (rozlišením) je 2073600 obrazových bodů, kdy na každý bod je potřeba 128 bitů. Při předpokladu zachování 25 snímků za sekundu je během 15 minut videa použito 22500 snímků. Jednoduchým vynásobením $2073600 \times 128 \times 22500$ je spočtena velikost souboru a to 695 GB (zaokrouhleno na celé GB).

5.5 Porovnávané metriky

U výsledných souborů (videa) byl sledován čas, ve kterém bylo zdrojové video převedeno na cílové, jeho výsledná velikost (v konkrétním nastavení), průměrné PSNR a SSIM.

PSNR (Peak signal-to-noise ratio) je metrika, která určuje odstup špičkového signálu od šumu. Tedy poměř mezi největší možnou silou signálu a silou šumu. Jedná se o nejčastěji používané měření kvality rekonstruovaného signálu ze ztrátové komprese. Jako signál je chápán zdroj původních dat a šum je chyba způsobená kompresí.[11]

SSIM (structural similarity) je index, který vyjadřuje podobnost dvou obrazů. Jedná se o metodu srovnání, jež bere v potaz skutečnost, že lidské vidění je vysoce přizpůsobeno k extrahování strukturální informace. Tento index nabývá hodnot -1 až 1, kde 1 vyjadřuje shodné obrazy.[12]

SSIM oproti běžným a široce používaným metodám (jako je PSNR) není založeno na

jednoduchém výpočtu a jasném významu, který dovoluje snadnou optimalizaci, avšak mnohem lépe vyjadřuje vnímání kvality obrazu.[13] Proto je velmi často vedle jiné metody. V této práci jsem vybral PSNR – tato kombinace je mimo jiné využívána také v článku *Cloud Gaming: Architecture and Performance*[6].

5.6 Výsledky porovnání

V tabulce 5 jsou uvedeny naměřené hodnoty. Ovšem je nutné poznamenat, že porovnávané kodeky používají ke zmenšení datového toku rozdílu snímků, což může být v případě použití vrhání paprsků problematické. Při vykreslení obrazu totiž může dojít (a dochází) k zobrazení velkého množství šumu (je vrženo málo paprsků – především prvních několik kol). To může vést k velkým skokům v množství přenášených dat. Zadavatel ŠKODA AUTO a.s. by upřednostňoval kompresi jednotlivých bloků (které se věnuji v kapitole Komprese dlaždic na straně 25), i když kompresi celkového obrazu nezavrhl.

Z tabulky 5 je patrné, že nejrychleji byl zpracován kodek H.264, který při velmi dobrém kompresním poměru zvládne zpracovat přibližně 36 snímků za sekundu na testovacím virtuálním stroji a zachovat velmi dobré obrazové vlastnosti. Na druhé straně spektra se pak nachází kodek VP9, který je vyvíjený firmou Google a používaný pro portál YouTube. Tento kodek měl nejpomalejší zpracování z porovnávaných kodeků (v testovacím prostředí), což bylo v rozporu s mým očekáváním, jelikož tento kodek bývá prezentován jako velmi rychlý.[15] Porovnání kodeků H.264 a MPEG-4 dopadlo dle očekávání. Kodek H.264 má také kromě vyšší rychlosti zpracování lepší obrazové vlastnosti.[14]

Při implementaci *cluster mastera* měla implementace využívat mezi prvními kodek H.264, který dopadl v porovnávání nejlépe a nejvíce vyhovuje potřebám zadavatele.

Označení	Čas	Velikost (MB)	Průměrné PSNR	Průměrné SSIM	Snímků za sekundu
Zdrojový (RGBA, 32 bit)		177979			
Zdrojový (RGBA, 128 bit)		711914			
H.264 (Ztrátový, CRF 23)	0:10:21	752	7,611819	0,005592	36,3929
H.264 (Ztrátový s maximálním BR)	0:10:58	1069	7,655911	0,005675	34,3465
H.264 (Ztrátový, CRF 18)	0:11:28	1348	7,659056	0,005729	32,8488
H.264 (Bezztrátový)	0:15:05	7776	7,348809	0,005199	24,9724
MPEG-4 (Střední kvalita)	0:20:34	313	7,423715	0,005626	18,3144
MPEG-4 (Vysoká kvalita)	0:24:05	3680	7,714738	0,005710	15,6401
VP9 (Bezztrátový)	2:47:44	9159	7,729741	0,005751	2,2456
VP9 (Ztrátový, CRF 18)	7:48:31	426	7,723102	0,005650	0,8040

Tabulka 5: Porovnání kodeků

Kapitola 6
Komprese dlaždic

V této kapitole se budu zabývat kompresí jednotlivých dlaždic. Ve zkoumaném případě uvažuji dlaždice o velikosti 32×32 bodů. Dlaždice (používané modulem *RayTracer*) uchovávají obrazové body pomocí RGBA, tedy každý bod se skládá z červené, zelené, modré a průhlednosti. Tyto barvy jsou reprezentovány pomocí datového typu `float` (32 bitů). Alfa kanál ve skutečnosti obsahuje dvě dvoubytové hodnoty. Jelikož modul *RayTracer* pracuje s těmito hodnotami při dalším výpočtu, není možné tyto hodnoty více omezit bez patřičného testování a konzultace s autory modulu *RayTracer*.

Následující postupy využívají běžných kompresních algoritmů pro obrázky. Také je uvažováno snížení barevné hloubky, díky čemuž může být výrazně snížena velikosti dlaždice. Za předpokladu, že je dlaždice v podobě (jak byla popsána výše), získáme $32 \cdot 32 = 1024$ bodů. Každý z těchto bodů je reprezentován čtyřmi bajty (32 bitů) na každou ze složek RGBA. $4 \cdot 32 = 128$ bitů. Každá taková dlaždice před kompresí obsahuje tedy 16 kB obrazových dat. Snížením barevné hloubky na 16 bitů jsem získal $3 \cdot 16 + 32 = 80$ bitů na obrazový bod, tedy 10 kB dat na dlaždici. Tato hodnota je dále snížena vhodným kompresním algoritmem.

Rec. 2020[17] uvažuje 10/12 bitů na vzorek[16]. Běžný je také RGB s osmi bity na vzorek. Porovnání vykreslované oblasti barev je znázorněno na obrázku 11.

Komprese byla testována na 40 dlaždicích o rozměrech 32×32 obrazových bodů.

6.0.1 JPEG

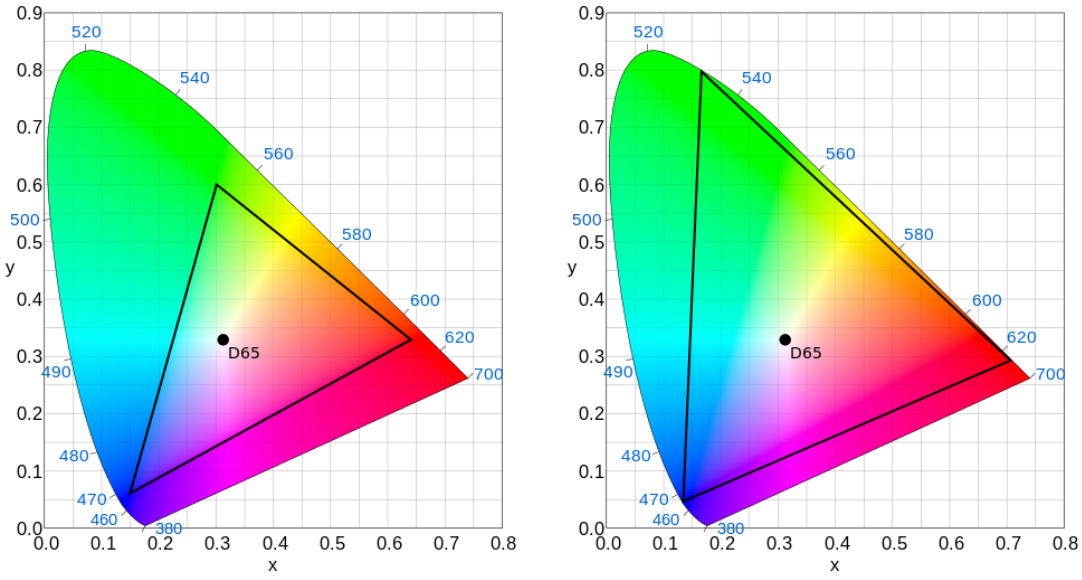
JPEG je metoda *ztrátové komprese* užívaná pro ukládání obrázků ve fotorealistické kvalitě. Zpracovávají se jednotlivé složky YCbCr (do kterých je RGB převedeno). JPEG bohužel nepodporuje průhlednost. Při zpracování dochází zejména k úbytku vzorků v barevných složkách, na které není lidské oko tak citlivé jako na jas[17, 18, 19]. Následně se složky obrázku rozdělí na bloky o velikosti 8×8 obrazových bodů. A pro každý takový blok se provede 2D DCT[18]. Díky tomu lze každý blok vyjádřit jako složení signálů o různých frekvencích. Následně se nad bloky provede kvantizace (umožňuje zanedbat některé změny, na které není oko tak citlivé). Dále se provede komprese pomocí RLE[19] a pak Huffmanovým kódováním nebo aritmetickým kódováním.[17]

Dlaždice v analýze (strana 3) pracovala se 32 bity na bod, tedy měla velikost 16 kB.

[17] ITU-R Recommendation BT.2020. Jedná se o doporučení vydané Mezinárodní telekomunikační unií (agentura OSN), které udává parametry pro televizní systémy v produkčním prostředí a mezinárodní programové výměně.

[18] Dvourozměrná diskrétní cosinová transformace

[19] Run-length encoding



(a) sRGB

(b) Rec.2020

Obrázek 11: Barevný prostor

Po omezení barevné složky na 8 bitů na barvu, jelikož JPEG používá 24 bitů na pixel, získáme před kompresí dlaždici o velikosti 7 kB ($(8 \cdot 3 + 32) \cdot 56$). Na obrázku 12b a 12c je vidět dlaždice, která je komprimovaná pomocí programu GIMP. Tato dlaždice má v podobě bez komprese 7 kB (takže má každý bit reprezentován pomocí RGB s hloubkou 8 bitů bez průhlednosti), po kompresi má 1,05 kB (JPEG s kvalitou 50) nebo 2,35 kB (JPEG s kvalitou 100) podle parametru komprese (bez průhlednosti). Aby byl funkční modul *RayTracer*, musí se navíc přenést data průhlednosti o velikosti 4 kB na dlaždici.

Z uvedených údajů je patrné, že při použití bezztrátové komprese typu JPEG je snížena velikost dlaždice ze 7 kB (3 kB RGB a 4 kB průhlednost) na 6,35 kB a v případě použití ztrátové komprese bylo dosaženo velikosti 5,15 kB na dlaždici. Velikost dlaždice byla tedy snížena o 26,4 % v případě bezztrátové komprese a o 37,2 % v případě komprese ztrátové.

V kapitole analýza (strana 3) jsem počítal s 6600 dlaždicemi pro CAVE projekci, kdy každá dlaždice měla velikost 16 kB. Použitím bezztrátové JPEG komprese byla velikost dlaždic snížena přibližně na 5,2 kB, dosaženo bylo úspory přibližně 67,8 %. Pro 6600 bloků na jednu stěnu CAVE projekce s 24 snímky za vteřinu je tak potřeba přenést $5,2 \text{ kB} \cdot 6600 \cdot 4 \cdot 24 = 3294720$ kB za sekundu, tedy přibližně 4,14 GB/s což je 25 Gb/s.

6.0.2 JPEG2000

Jedná se o rozšíření předchozího formátu, které jej doplňuje o podporu průhlednosti a větší barevnou hloubku.

Průměrná velikost dlaždice při použití bezztrátové komprese včetně 16 bitové průhlednosti byla 1,8 kB. Pokud bude zbylá složka průhlednosti (2 kB) přenášena samostatně, celková velikost průměrné dlaždice bude rovna hodnotě 3,8 kB.

Při použití 6600 dlaždic pro CAVE projekci bude požadovaný datový tok roven 2 307 680 kB (2,3 GB), tedy 17,6 Gb/s.

6.0.3 PNG

PNG je metoda *bezztrátové komprese* užívaná pro rastrovou grafiku. Na rozdíl od JPEGu z předchozí kapitoly, tento formát podporuje průhlednost. Barevná hloubka může být 1–16 bitů na pixel a je použit Deflate[20] algoritmus s filtry.[21, 22] Deflate algoritmus je používán i knihovnou zlib pro kompresi souborů.[23] Tuto knihovnu již zadavatel ŠKODA AUTO a.s. zkoušel použít pro přenos scény, ale momentálně je podpora komprese vypnuta.

Dlaždice v analýze (strana 3) pracovala se 32 bity na bod, tedy měla velikost 16 kB. Na obrázku 12a je vidět dlaždice, která je komprimovaná pomocí programu ImageMagick. Dlaždice má v podobě bez komprese 16 kB, po kompresi má průměrná dlaždice 1,82 kB nebo 2,23 kB podle parametru komprese (výsledná dlaždice je vždy stejná – bezztrátové zpracování) s přenosem poloviny průhlednosti. K tomu ovšem musí být samostatně přenesena druhá polovina informace o průhlednosti, (jak bylo zmíněno výše, nemůže být hodnota průhlednosti omezena) tedy ještě 2 kB dat (bez využití jiné komprese). Celková velikost je tedy 3,82 kB (snížení o 76,0 %) a 4,23 kB (snížení o 73,5 %).

Opět jsem předpokládal 6600 dlaždic užitých pro CAVE projekci a stejným výpočtem jako v předchozí části jsem získal množství dat přenesených za jednu sekundu s obnovovací frekvencí 24 snímků za sekundu, které je rovno $3,82\ \text{kB} \cdot 6600 \cdot 4 \cdot 24 = 2420352$ kB, což je 2,3 GB. Kapacita sítě potřebná pro přenesení dat, kdy jsou dlaždice komprimovány pomocí PNG, je tedy 18,4 GB/s.

(a) PNG (b) JPEG s kvalitou 50 (c) JPEG s kvalitou 100

Obrázek 12: Porovnání JPEG a PNG

[20] Deflate je kombinací algoritmů LZ77 a Huffmanova kódu.

Metoda	Kvalita komprese	Vstupní velikost	Výstupní velikost
16 bitů na složku	1	8 kB	1,7 kB
16 bitů na složku	9	8 kB	1,7 kB
32 bitů na složku	1	16 kB	1,8 kB
32 bitů na složku	9	16 kB	1,8 kB

Obrázek 13: Velikost dlaždic po kompresi pomocí Deflate algoritmu (gzip)

6.0.4 GZIP

Prostá komprese, která nezapisuje informace o obrázku (jako jsou rozměry, paleta barev, vrstvy, ...) má tu výhodu, že tyto nadbytečné informace nemusí být přenášeny. Testovací dlaždice byla komprimována programem `gzip` s volbou kvality komprese 1 (nejhorší) a 9 (nejlepší). Při použití programu gzip nedochází ke ztrátě dat, protože používá stejně jako PNG ke kompresi dat Deflate algoritmus[24, 21, 22]. To také znamená, že by měl dosahovat podobných kompresních vlastností. Aby byly výsledky porovnatelné (s výsledky v předchozích kapitolách), provedl jsem kompresi 16 bitové varianty (16 bitů na každou ze složek RGBA). Kromě 16 bitové varianty jsem provedl také kompresi varianty 32 bitové, která by přenášela kompletní rozsah hodnot.

Výsledné velikosti dlaždic jsou uvedeny v tabulce 13. V úvahu všek musí být bráno, že paleta použitá pro vytvoření testovací dlaždice neobsahuje barvy přesahující barevnou hloubku 16 bitů, proto je výsledná velikost 32 bitové varianty velmi blízko variantě 16 bitové.

Dále je vidět, že dlaždice včetně průhlednosti má velikost obrazových dat 1,8 kB. Za předpokladu použití 16 bitů na barevnou složku a 32 bitů na průhlednost (dvě 16 bitové hodnoty) a také, že barevná složka bude mít při větším rozsahu (než 16 bitů) ostatní bity nulové, bude možné je efektivně komprimovat a dosáhnout velikosti blízké 1,7 kB na dlaždici.

Opět využiji k výpočtu potřebné kapacity 6600 bloků a čtyř stěn CUBE projekce. Po dosazení velikosti dlaždice tedy získám $1,7\ \text{kB} \cdot 6600 \cdot 4 \cdot 24 = 1077120$ kB (1,02 MB/s) při obnovovací frekvenci 24 snímků za sekundu. To odpovídá přenosové kapacitě 8,2 Gb/s.

6.0.5 Zhodnocení

Jelikož pro dlaždice využívané modulem *RayTracer* je potřeba průhlednost, musí být u JPEG komprese uvažován i kanál průhlednosti, který musí být přenášen samostatně. Také je vidět z obrázku 12, že obrazová kvalita JPEGu s klesající volbou kvality výrazně klesá. Na druhou stranu formát PNG poskytuje podporu průhlednosti a až 16 bitů na pixel pro uchování barev i průhlednosti bez ztráty kvality, podobné možnosti nabízí

i JPEG2000, a při prosté kompresi obrazových dat pomocí programu `gzip` (využití knihovny zlib) můžeme zachovat dokonce 32 bitovou hloubku barev.

I při samostatném nekomprimovaném přenosu 16 bitů (druhé poloviny průhlednosti) v případě použití PNG je dosaženo velmi dobrého kompresního poměru, který by měl vést až k teoretické úspoře 76 % během přenosu. Avšak při kompresi pomocí nástroje `gzip` využívajícího zlib knihovnu může být dosaženo úspory až 89 %. Kapacita linky směrem ke clusteru je schopna přenést 10 Gb/s (viz. tabulka 1 na straně 3) a s využitím komprese je možno přenos realizovat na lince o kapacitě 9 Gb/s. Použití komprese by teoreticky mělo umožnit přenášet obraz požadovaný zadavatelem po existující infrastruktuře.

Kapitola 7
Vyvažování zátěže

V této kapitole se zaměřím na vyvažování zátěže mezi jednotlivé klienty výpočetního clusteru. Komponenta starající se o vyvažování zátěže je zároveň zodpovědná za řízení datového toku v rámci clusteru. Kromě rozdělování dlaždic, dle nějakého vzoru, by měla být schopna odeslat data všem klientům clusteru.

První část této kapitoly bude věnována známým metodám vyvažování zátěže, které jsou používány například pro vyvažování HTTP požadavků na webové servery. U každé z těchto metod jsem uvedl princip její funkce a shrnul výhody a nevýhody této metody při použití v aplikaci *VRUT*.

Ve zbylých dvou kapitolách popíši metodiku použitou při porovnávání těchto vyvažovacích metod, výsledky jejich porovnání a popis implementace vyvažovací komponenty.

7.1 Známé vyvažovací metody

Vyvažování zátěže se používá například při přístupu k webovým stránkám nebo webovým službám. Pro velké množství klientů, kteří požadují zpracování svého požadavku, je nemožné obsloužit je pomocí jednoho serveru, obzvláště pokud jsou požadované služby výpočetně náročné.

Při velmi časově náročných operacích se na výsledek zpracování většinou nečeká. Místo toho se použije nějaký robustní systém front a klient je informován (nebo se opakovaně dotazuje) o průběhu zpracování jeho požadavku.[25] Podobným způsobem je v aplikaci *VRUT* řešeno zpracování dlaždic. Dlaždice je totiž po přijetí zařazena do fronty, kde čeká na obsloužení (v našem případě výpočet) a po obsloužení odeslána zpět jako jiná zpráva. Jedná se o asynchronní komunikaci s využitím front. (Podrobněji bylo popsáno fungování aplikace VRUT v kapitole VRUT na straně 5.)

V současné době je k dispozici velké množství různých load balancerů využívajících celou škálu vyvažovacích algoritmů, ale žádný z těchto load balancerů není schopný efektivně pracovat s dlaždicemi aplikace VRUT (jelikož VRUT má specifickou komunikaci). Tyto load balancery využívají různých metod vyvažování zátěže; ty nejznámější metody představím v této kapitole a zároveň u každé uvedu možnosti použití v rámci projektu VRUT.

7.1.1 Round Robin

Jedna z nejjednodušších metod vyvažování zátěže. Tato metoda nezohledňuje rozdílný výpočetní výkon jednotlivých serverů, počet otevřených spojení a ani jejich zatížení. [26]

Požadavek je předán vždy dalšímu výpočetnímu uzlu v pořadí. Poté, co přiřadí výpočet všem uzlům, začíná opět od prvního uzlu. Při N uzlech jsou tedy vždy vybrány uzly v pořadí:

$$1, 2, ..., N, 1, 2, ...$$

Použitím tohoto vyvažovacího algoritmu by bylo zajištěno, že dlaždice budou rovnoměrně rozloženy na všechny výpočetní uzly bez ohledu na jejich aktuální zatížení a nebo výpočetní výkon.

Takové řešení se zdá být optimální v případě, že všechny výpočetní uzly jsou podobně výkonné a každá dlaždice vyžaduje přibližně stejný výpočetní výkon. Ovšem je velmi nevýhodné v případě, kdy jednotlivé uzly mají jiný výpočetní výkon nebo rozdílné připojení k síti. Výpočet dlaždic pomocí vrhání paprsků také nezaručuje, že jednotlivé dlaždice budou stejně výpočetně náročné. V nejhorším případě může load balancer přiřadit nejobtížnější dlaždice vždy stejné podmnožině serverů a tím výrazně zvýšit jejich zátěž a celkové zpoždění vypočtení obrazu scény.

7.1.2 Weighted Round Robin

Jedná se o metodu, která rozšiřuje předchozí o možnost přiřazovat serverům žádosti v určitém poměru, který je dán „váhou“. Výkonnější servery tak mohou zpracovávat například dvojnásobek žádostí než servery méně výkonné. [26, 27]

Například pokud existují výpočetní uzly A, B a C s váhami 4, 3 a 2, tak budou servery vybrány v pořadí:

$$A, A, B, A, B, C, A, B, C$$

Použitím této metody může být zohledněn výkon výpočetních uzlů. Tuto informaci je možno znát předem (například konfigurační soubor), a nebo ji zjistit od jednotlivých výpočetních uzlů za běhu programu.

Ve standardní implementaci se ovšem tato metoda potýká s obdobným problémem jako *Round Robin* a to s tím, že pokud jsou požadavky různě náročné, může dojít k tomu, že náročnější požadavky jsou vždy přiřazeny na stejnou podmnožinu serverů.

7.1.3 Random

Jedná se o jednu z nejjednodušších metod. Tato metoda přiřazuje požadavky serverům zcela náhodně.

Chování této metody je velmi obtížně predikovatelné. Podle pořadí uzlů v konfiguračním souboru a zdroje náhodných hodnot se může chovat při různých konfiguracích

a na různých systémech zcela odlišně. Ovšem pokud jsou všechny uzly stejně výkonné, jsou stejně přístupné a generátor generuje hodnoty v rovnoměrném rozdělení, může být předpokládána podobná účinnost jako u metody *Round Robin*.

Mohlo by se ovšem stát, že jednomu serveru bude přiřazeno 7 dlaždic, zatímco ostatním pouhé dvě a nebo dokonce žádná. Pak by vykreslení obrazu (vzhledem k synchronizaci vykreslení) proběhlo až poté, co by výpočetní uzel spočetl všech sedm dlaždic. Při předpokladu čtyř výpočetních vláken na jeden server, pak bude jeden server zahlcen a ostatní nevytíženi.

Možným řešením by pak bylo zvolit distribuční funkci tak, aby vybírala s větší pravděpodobností uzly, které mají méně aktivních výpočtů (nebo na ně bylo odesláno menší množství požadavků). Takováto modifikace by nejspíše způsobila i to, že na klienty s nižším výpočetním výkonem by bylo zasíláno méně požadavků (výpočet by trval delší dobu, a tak by přispěl k tomu, aby serveru bylo po delší dobu přiřazováno méně požadavků).

7.1.4 Source IP

Požadavek je distribuován serverům na základě své IP adresy. Pokud výpočetní uzel selže, je rozdělení IP adres na servery přepočítáno. Dokud výpočetní klient neselže, obsluhuje stále stejné žadatele.[26]

Tuto metodu bez modifikace, která bude zmíněna dále, nemá smysl uvažovat, neboť alespoň prozatím, je ke klientu připojen jeden zobrazovací server, jehož modul *cluster* zároveň poskytuje podporu pro vyvažování zátěže.

7.1.5 Tile ID

Jedná se o modifikaci *Source IP*, která místo IP adresy využívá k rozdělení dlaždic identifikaci dlaždice v celkové scéně. Na počátku komunikace se rozvrhnou dlaždice na jednotlivé servery (výpočetní klienty) a poté jsou posílány vždy na stejný server, pokud nedojde k jeho selhání. Pokud dojde k selhání některého z uzlů, je nutné opětovně rozvrhnout mapování dlaždic na servery, a to buď pro celý rozsah identifikátorů a nebo pouze pro dlaždice, jenž byly mapovány na server, který selhal.

Pokud bude dbáno na to, aby se nedistribuovaly na stejné uzly dlaždice, které se nacházejí v těsné blízkosti a zároveň, aby byl adresní prostor, pokud možno, co nejlépe rozvržen na servery (což je ve zkoumaném případě poměrně jednoduché, jelikož se jedná o posloupnost hodnot 1, 2, 3, ..., N[21]), mohou být očekávány u serverů s podobným výpočetním výkonem stejné vlastnosti jako u *Round Robin*, navíc však přibude možnost přenášet pouze rozdíl oproti předchozímu stavu (jelikož na daném serveru počítáme vždy stejnou množinu dlaždic).

[21] N je identifikátor poslední dlaždice

Pokud by servery neměly podobný výpočetní výkon, může být algoritmus, jenž bloky přiřazuje jednotlivým serverům, modifikován tak, aby uvažoval výpočetní výkon serveru.

7.1.6 URL

Metoda vyvažování zátěže na základě URL je podobná *Source IP*, jen s tím rozdílem, že výpočetní uzel obsluhuje určitou množinu URL.

Zde by místo URL muselo sloužit identifikační číslo pozice dlaždice v obrazu. Takovou změnou je ovšem problém převeden na variantu *Tile ID*.

7.1.7 Least connections

Je sledován počet otevřených spojení/požadavků na výpočetního klienta a nové požadavky jsou distribuovány serveru s nejmenším množstvím přidělených požadavků. [26]

Podobně jako u varianty *Weighted Round Robin* i zde mohou být použity váhy pro jednotlivé servery. Server s vyšší váhou pak bude schopen obsloužit více otevřených spojení.

Tato metoda zaručí, že na každý server je přiřazeno přibližně stejné množství požadavků a to proto, že odesláním prvního požadavku se zvýší počet požadavků, které server zpracovává. Následující požadavek již není distribuován na stejný server neboť je počet otevřených požadavků vyšší než u ostatních serverů. Servery se stejnou váhou pak mohou být vybrány různými metodami (náhodně, první v pořadí).

7.1.8 Least traffic

Je sledován datový tok od každého výpočetního klienta a nové požadavky jsou zasílány výpočetním klientům s nejmenším zatížením.

Jelikož jsou dlaždice pro požadovaný rámec vždy stejně velké, je tato metoda ekvivalentní metodě *Least connections*.

7.1.9 Least latency

Výpočetním klientům se odešle požadavek, na který mohou rychle odpovědět, aby se zjistilo, jak jsou zatížení (doba odezvy na požadavek). Požadavek je poslán prvnímu výpočetnímu uzlu, který odpoví.[26, 27]

Odpověď na požadavek by ve zkoumaném případě mohla zahrnovat počet dlaždic, které klient aktuálně zpracovává, zatížení procesoru (load) a průměrnou dobu výpočtu jedné dlaždice.

Nevýhodou je, že na každého klienta musí být odeslán požadavek, klient jej musí zpracovat a poslat nám odpověď. I když se jedná o jednoduchý požadavek a přenesení dat nevyžaduje velkou přenosovou kapacitu, není ve zkoumaném případě vhodné zvyšovat datový tok ani takto malými objekty.

Výhodou je, že informaci o svém vytížení poskytuje sám server (výpočetní klient), a tak může poskytovat daleko širší a přesnější soubor informací, než mohou být odvozeny ze samotné komunikace, nebo předem definovat.

7.2 Porovnání

V této kapitole popíši, jakým způsobem jsem porovnával výše uvedené metody, jaké nástroje jsem k tomu použil a jaký byl proces porovnávání.

Abych metody mohl porovnat, musel jsem si napřed stanovit, jaké hodnoty budu měřit a zajistit, pokud možno, ekvivalentní podmínky pro všechny algoritmy. Jednotlivé algoritmy jsou porovnávány pomocí *počtu kroků nutných k odeslání, výpočtu a přijetí všech dlaždic ve výpočetním clusteru.*

Rozhodl jsem se testovat distribuci jednoho tisíce dlaždic mezi osm výpočetních klientů. Tito klienti mají tři výkonové varianty, kde nejvýkonnější skupina je tvořena pěti uzly, následuje jeden méně výkonný stroj a dva nejméně výkonné.

Dále bylo potřeba uvažovat zpoždění na síti. Jelikož mají všechny uzly v rámci clusteru stejné síťové prostředky, rozhodl jsem se, že i všechny mnou testované uzly budou mít ekvivalentní přístup k síti.

Dlaždice, které jsou v rámci clusteru počítány, mohou mít různou složitost. (To je dáno množstvím dopadajícího/odraženého světla a materiály.) Proto jsem musel zvolit postup takový, který pokaždé produkuje a rozděluje stejně složité dlaždice.

7.2.1 Testovací nástroj

Z důvodu splnění všech uvedených podmínek jsem se rozhodl porovnání těchto metod provést pomocí simulované sítě i výpočetních uzlů. Napsal jsem tedy program v programovacím jazyce Java, který mi umožnil pozorovat chování uvedených metod na více výpočetních klientech za stejných podmínek. Program přijímá textové příkazy, které slouží k vybudování „sítě“ skládající se z *uzlů* a *spojení*.

Uzly a spojení implementují rozhraní `Tickable`, které definuje metodu `tick()`. Uzly implementující toto rozhraní jsou pak v každém časovém skoku (tiku) informovány o tom, že mají přejít do následujícího časového okamžiku. Toho je využito jak v průběhu simulace doby nutné k výpočtu, tak pro simulaci odezvy na síti.

Příkazy mohou být také čteny ze souboru, kdy jeden řádek odpovídá jednomu příkazu. Pokus soubor `example.txt` obsahuje:

```
addServer master 1 weighted-round
#addServer master 1 round
tileCount master 500
addClient node1 1
addClient node2 5
addClient node3 10
addClient node4 10
addClient node5 1
addClient node6 1
addClient node7 1
addClient node8 1
# Connection on both direction
addConnection master node1 5
addConnection node1 master 5
addConnection master node2 5
```

```
addConnection node2 master 5
addConnection master node3 5
addConnection node3 master 5
addConnection master node4 5
addConnection node4 master 5
addConnection master node5 5
addConnection node5 master 5
addConnection master node6 5
addConnection node6 master 5
addConnection master node7 5
addConnection node7 master 5
addConnection master node8 5
addConnection node8 master 5
render master
tickInterval 50
s
```

a program je spuštěn s parametrem `example.txt`, pak program vytvoří jeden řídící uzel, který požaduje provedení výpočtu s názvem *master* a osm výpočetních uzlů (node1, node2, . . . , node8):

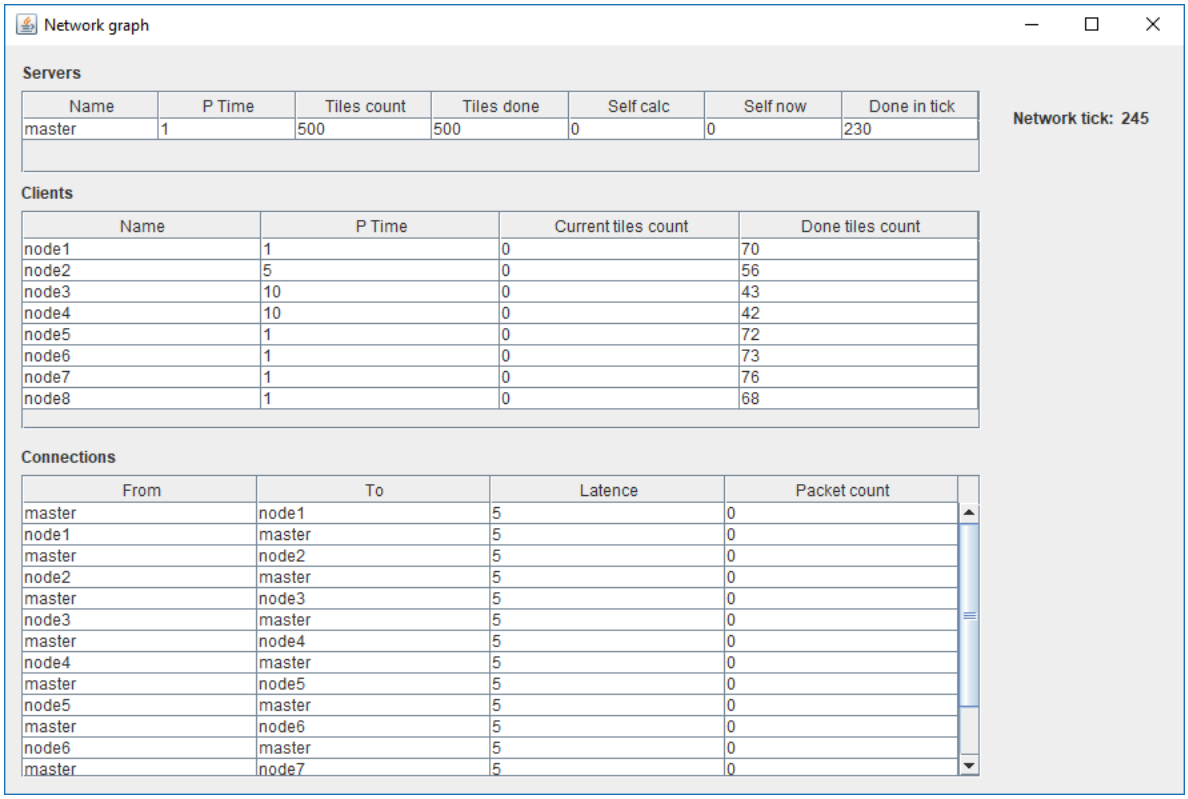
- node1 (doba zpracování průměrné dlaždice: 1)
- node2 (doba zpracování průměrné dlaždice: 5)
- node3 (doba zpracování průměrné dlaždice: 10)
- node4 (doba zpracování průměrné dlaždice: 10)
- node5 (doba zpracování průměrné dlaždice: 1)
- node6 (doba zpracování průměrné dlaždice: 1)
- node7 (doba zpracování průměrné dlaždice: 1)
- node8 (doba zpracování průměrné dlaždice: 1)

Řádky addConnection poté vytváří síťové spojení mezi jednotlivými uzly. Spojení je navázáno obousměrně mezi uzlem a *master*em.

Výstup je znázorněn graficky pomocí tabulky. Pro předchozí soubor příkazů pak výstup může vypadat jako na obrázku 14.

7.2.2 Porovnávané metody

Pro porovnání jsem vybral *Random*, *RoundRobin* a *Weighted Round Robin*. Tyto metody vyhovují nejvíce rozdělování zátěže v aktuálním stavu. Ke konci této kapitoly se ještě budu krátce zabývat ostatními metodami a jejich možným budoucím využitím. *Weighted Round Robin* navíc nepracuje se statickými váhami jednotlivých uzlů, a tak je jeho chování mírně odlišné od chování se statickými váhami. V mé testovací implementaci upravuji váhy na základě množství odeslaných požadavků a také na rychlosti, s jakou jednotlivé požadavky uzel zpracoval, čímž na sebe částečně převzal funkci jiných uvedených metod pro vyvažování.



Obrázek 14: Grafický výstup aplikace pro testování rozdělování zátěže

Například při každém odeslání nového požadavku na vykreslení dlaždice je odebráno určité skóre (dejme tomu 5) výpočetnímu uzlu, který odeslání provedl. Při přijetí odpovědi se provede porovnání času odeslání a přijetí dané žádosti a na základě statistických dat z minulých přenosů (na tentýž nebo jiný výpočetní uzel) se upraví váha tak, že je do její úpravy zahrnuta rychlost výpočtu uzlu. Uzlu může tedy být přičteno skóre $5 + 2$, protože jeho výpočet byl rychlejší než průměrný. Na druhou stranu může být uzlu přičteno skóre $5 - 3$, protože jeho výpočet trval déle než u jiných uzlů. Vidíme, že výsledná bilance za odeslání žádosti a přijetí odpovědi u průměrného uzlu je nulová (jeho skóre je nezměněno).

Některé z uvedených metod (Známé vyvažovací metody, str. 30) nebyly zahrnuty do porovnání, protože se jim bude nejspíše věnovat pozornost později. K těmto metodám patří například *Tile ID*, která bude vhodná, pokud se zadavatel rozhodne implementovat kompresi obrazu na základě zasílání pouze změněných dlaždic.

Metoda *Least connections* a její příbuzné varianty jsou pak částečně zahrnuty v mé implementaci *Weighted Round Robin*, pokud ovšem bude ze strany zadavatele zájem, nemělo by být těžké tyto metody implementovat.

7.2.3 Výsledky porovnání

Výsledky porovnávaných metod (respektive konkrétní implementace v testovacím prostředí) jsou zaneseny v tabulce 15. Z tabulky je patrné, že nejlepších výsledků dosahovala metoda *weighted round robin*, která v deseti testovacích kolech (jiný seed[22]) pokaždé vedla k nejrychlejšímu výpočtu celé scény. Průměrná doba dokončení výpočtu scény byla na tisíci dlaždicích o 47 kroků nižší než druhá nejlepší metoda.

Metoda	Průměrný poč. kroků	Průměrná odchylka
Random	682	20,77
Round Robin	635	0,00
Weighted Round Robin	445	1,00

Obrázek 15: Výsledek testování metod

Zároveň je v tabulce 15 vidět, že metoda *Weighted Round Robin* má v testovaném případě zanedbatelnou průměrnou odchylku (jeden krok).

Nejhůře dopadla metoda *Random*, která pracovala v průměru o 237 kroků pomaleji než nejrychlejší metoda. Zároveň měla největší průměrnou odchylku, což by při nasazení vedlo k tomu, že každý rámec by byl vykreslen jinou rychlostí. V některých případech by mohlo dojít k tomu, že proběhne několik pomalejších zpracování po sobě, což by vedlo k „trhání“ obrazu[23].

Pro implementaci jsem na základě výsledků vybral metody *Round Robin* a *Weighted Round Robin*.

[22]Seed je nějaká hodnota, která je použita jako vstupní parametr generátoru náhodných čísel. V mém případě byl využit generátor Random, který je součástí standardního API programovacího jazyka Java.

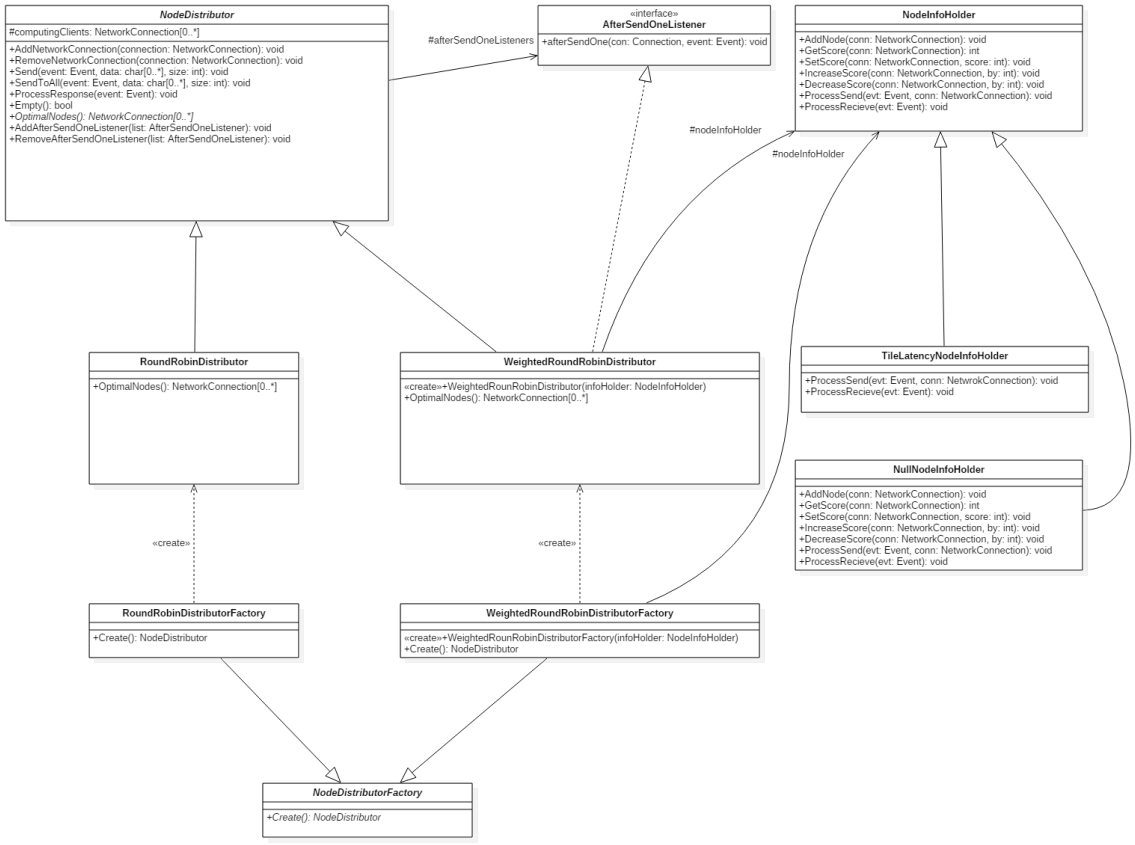
[23]V herním průmyslu se takovémuto snížení zobrazovací frekvence říká frame drop.

7.3 Implementace

Vybrané vyvažovací metody jsem musel implementovat ve dvou programovacích jazycích (testovací program je psán v programovacím jazyce Java, zatímco VRUT je psán v programovacím jazyce C++) a v případě implementace pro VRUT brát v úvahu i okolní komponenty a komunikaci jako takovou. Při implementaci bylo nutné algoritmy pozměnit tak, aby lépe odpovídali svému prostředí a rozhraní okolních komponent.

V této části se budu zabývat pouze implementací na straně VRUTu a to popisem jednotlivých komponent a změn, které byly nutné k fungování těchto metod.

Na obrázku 16 je zobrazen diagram tříd, na kterém vidíme, že metody *Round Robin* a *Weighted Round Robin* jsou reprezentovány pomocí tříd `RoundRobinDistributor` a `WeightedRoundRobinDistributor`, které mají společného předka.



Obrázek 16: Třídy vyvažovací komponenty

Třída `NodeDistributor`, jenž je jejich společným předkem, je abstraktní třídou, jenž poskytuje svým potomkům implementaci základních metod jako je přidání nebo odebrání spojení či odeslání události klientům.

`NodeDistributor` vystupuje ve zbytku modulu *Cluster2* jako interface poskytující distribuci dlaždic. Aby mohly implementace tohoto rozhraní (potomci této třídy) správně

rozdělovat dlaždice a jiné události, musí být kromě dat poskytnuta i událost, jenž je daty reprezentována[24]. Standardní volání metody `Send` způsobí, že po odeslání dat jsou informovány instance implementující rozhraní `AfterSendOneListener` a registrované pomocí metody `AddAfterSendOneListener`.

`AddAfterSendOneListener` může být použit pro vykonání akce po odeslání dat. Tuto možnost využívá i implementace metody *Weighted Round Robin* která po odeslání dat upraví skóre uzlu, na který byla data odeslána.

`NodeInfoHolder` představuje rozhraní udržující informace o konkrétních uzlech[25] clusteru. Tento interface dovoluje uživateli získat a modifikovat číselnou hodnotu, jenž je reprezentací výkonu daného uzlu. Konkrétní implementace může rozsah hodnot omezit, ale předpokládá se rozsah 1 až 100, kde vyšší číslo obvykle znamená výkonnější uzel.

`NodeDistributorFactory` je rozhraní, které slouží k vytváření instancí třídy `NodeDistributor`. Toto rozhraní vzniklo zejména proto, aby při vytváření nových instancí distributorů nebylo potřeba znát parametry požadované konstruktorem. Díky odstínění od nutnosti znát parametry konstruktoru a použití jednotného rozhraní pro vytváření instancí distributorů je možno dle nastavení modulu vytvořit instanci této tovární třídy[29] a odstranit tak nutnost kontrolovat nastavení při vytváření nového síťového spojení.

Původně implementace modulu *Cluster2* udržovala dvě oddělené mapy (tabulky klíč–hodnota) pro události vyžadující transformaci a události, které ji nevyžadují. Klíčem v těchto mapách byl pak identifikátor události a hodnotou seznam spojení, které tento typ události používají. Aby chování zůstalo pokud, možno, co nejvíce podobné původnímu, rozhodl jsem se zachovat toto rozdělení, a tak má každý typ události vlastní vyvažovací komponentu. Nově jsou tedy k dispozici dvě mapy, pro události s transformací a bez transformace, jejichž klíčem je typ události a hodnotou vyvažovací komponenta (instance třídy `NodeDistributor`).

Předchozí úprava si vyžádala i změnu metod `RegisterConnectionEventListener` a `UnregisterConnectionEventListener`, které jsou volány při vytváření spojení ze serveru na výpočetní klienty. Úpravy v obou metodách jsou velmi podobné, proto popíši jen úpravu v metodě `RegisterConnectionEventListener` (výpis 1), kde bylo nutno pozměnit přiřazování spojení k události. Po úpravě map (předchozí odstavec) vznikla nutnost vytvořit distribuční komponentu pro každý typ události, který ještě nebyl použit. Oproti prostému přidání do listu, musí být nejprve vytvořen objekt distribuční komponenty, kterému je spojení přidáno (`AddNetworkConnection`). Pokud tento typ události již používá jiné spojení, musí být objekt distribuční komponenty pouze vyzvednut a poté musí být přidáno spojení, které je dané události registrováno. Také

[24] Ve skutečnosti by tomu tak být nemuselo, ale původní implementace nepočítala s vložením dalšího členu do komunikačního řetězce, a tak probíhá serializace dat (respektive volání metod na objektu třídy `VRUTSerializer`) přímo v těle metody obsluhující příchozí události. Aby uživatelé mohli nadále využívat tohoto způsobu serializace, rozhodl jsem se poskytovat vyvažovací komponentě jak již připravená data (která si může uživatel upravit), tak událost, jenž je daty reprezentována.

[25] Za předpokladu, že každý uzel je v konfiguraci uveden jen jednou, představuje spojení jednoho výpočetního klienta.

musí být dbáno na to, kterého typu událost je, tedy zda byla označena jako vyžadující transformaci či nikoli.

```
void Cluster2::RegisterConnectionEventListener(NetworkConnection *networkHandler, const
    Event::EVENT_TYPE eventType, const bool transformed)
{
        if (eventType >= Event::EVT_HIGHEST)
                LOGERROR(wxT("<Cluster>Error registering listener, invalid event type")
                    );
        else
        {
                NodeDistributor * distributor;
                if (transformed) {
                        if (!networkConectionEventListenersTransformed[eventType]) {
                                networkConectionEventListenersTransformed[eventType] =
                                    nodeDistributorFactory->Create();
                        }
                        distributor = networkConectionEventListenersTransformed[
                            eventType];
                }
                else {
                        if (!networkConectionEventListeners[eventType]) {
                                networkConectionEventListeners[eventType] =
                                    nodeDistributorFactory->Create();
                        }
                        distributor = networkConectionEventListeners[eventType];
                }
                distributor->AddNetworkConnection(networkHandler);
                nodeInfoHolder->AddNode(networkHandler);
                REGISTER_LISTENER(eventType);
        }
}
```

Výpis 1: Zdrojový kód metody `RegisterConnectionEventListener`

Dále bylo potřeba pozměnit odesílání dat v metodě `processEvent` modulu *Cluster2* tak, aby využívalo distribuční komponenty. Jelikož výběr komponenty na základě události zůstal téměř stejný, bylo potřeba upravit jen samotné odeslání.

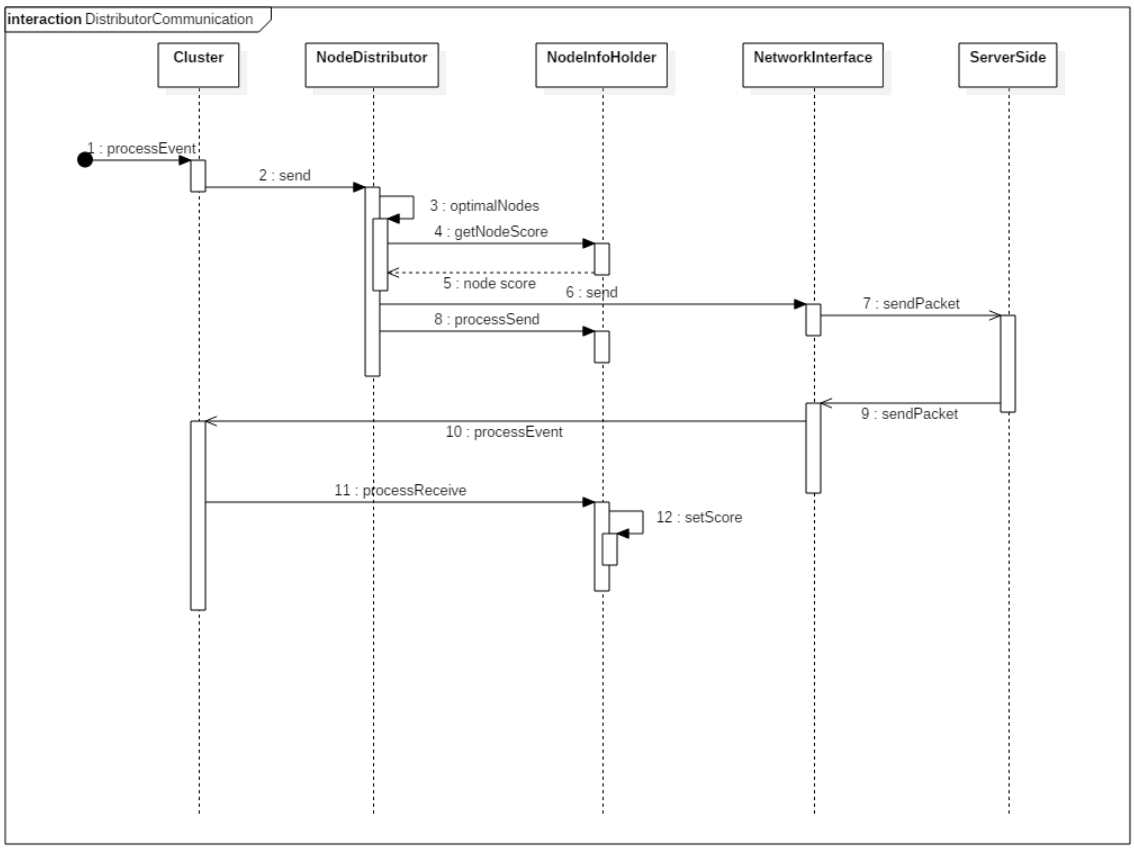
```
if (networkSerializer.GetSize() != 0)
{
        switch (evt->type) {
        case Event::EVT_RENDER_TILE:
                midList->Send(evt, networkSerializer.GetBuffer(), networkSerializer.
                    GetSize());
                break;
        default:
                midList->SendToAll(evt, networkSerializer.GetBuffer(),
                    networkSerializer.GetSize());
        }

        networkSerializer.Clear();
}
```

Výpis 2: Odesílání dat z metody `processEvent`

Výpis 2 je úryvek kódu metody `processEvent`, kde je volána distribuční komponenta. Ve výpisu je vidět, že distribuční komponenta se sice používá pro všechny typy událostí, ale vyvažování je momentálně dostupné jen a pouze pro události typu **Event::EVT_RENDER_TILE**. Za povšimnutí také stojí, že distribuční komponentě je předána jak událost, tak samotná serializovaná data a jejich velikost; o této skutečnosti jsem se již dříve zmiňoval.

Na obrázku 17 je sekvenční diagram zachycující využití distribuční komponenty v rámci modulu *Cluster2*. Po přijetí události je zavolána metoda `processEvent`, která v případě přijetí události **Event::RENDER_TILE** volá metodu `Send` objektu `NodeDistributor`. `NodeDistributor` následně vybere nejvhodnější výpočetní uzly[26], pozmění jejich skóre (pokud je to třeba) a provede samotné odeslání dat (k odeslání dat jsem využil existujících tříd, které byly popsány v kapitole VRUT na straně 5). Po přijetí a deserializaci události je zavolán hook[27], který provede případné úpravy skóre uzlu, který událost poskytl.



Obrázek 17: Komunikace s využitím `NodeDistributor`u

Na obrázku 17 a z předchozího odstavce si nejspíš všimnete, že úpravy skóre (volání metod na `NodeInfoHolder`) jsou někdy prováděny z distribuční komponenty a někdy přímo z modulu *Cluster2*. Jedná se o implementační rozhodnutí, které má zajistit rychlejší úpravu skóre uzlu po přijetí odpovědi. Zůstává zde také možnost předat příchozí událost distribuční komponentě, ale jelikož je využita pro každý typ události jiná distribuční komponenta, je výhodnější událost zpracovat přímo v modulu *Cluster2*.

[26] V případě mojí implementace *Weighted Round Robin* a *Round Robin* vybere vždy maximálně jeden uzel.

[27] Hook je rozhraní poskytované k rozšíření funkcionality vlastním kódem.[30].

Implementována byla také podpora pro oznamování výkonu vzdáleného výpočtu. Aktivací této funkce se aktivuje vlákno, jenž každých (v současné době) 15 sekund informuje jádro aplikace o výkonu vzdáleného výpočtu a doporučeném poměru lokálních a vzdálených dlaždic.

K získání informací o rychlosti lokálního výkonu jsem přidal dvě události jádra, které jsou shodné s **Event::RENDER_TILE** a **Event::RENDER_TILE_DONE** jen s tím rozdílem, že informují o tom, že byl započat lokální výpočet a nebo dokončen lokální výpočet. Tyto zprávy přijímá pak modul *Cluster2* a nechává je zpracovat `NodeInfoHolder`. Nový poměr lokálních a vzdálených dlaždic je navrhován na základě váženého průměru, kdy průměr minulých kol má větší váhu než průměrná rychlost aktuálního kola. Tento přístup minimalizuje skoky poměru.

Pan A. Míšek ze společnosti ŠKODA AUTO a.s. ovšem oprávněně namítl, že postup, kdy jsou přidány dvě nové události a měření času v každém výpočtu lokální dlaždice, vede ke zbytečnému zatěžování sběrnice aplikace a ztráta výkonu při vytváření těchto událostí může být v některých případech, kdy je potřeba velmi nízká odezva, kritická. Proto pracuji na modifikaci upravování poměru lokálních a vzdálených dlaždic a přesunu větší části odpovědnosti na modul *RayTracer*. Použití průměrného času vypočtené dlaždice (po vykreslení celého rámce) se v naivní implementaci nezdá efektivní.

7.3.1 RoundRobinDistributor

`RoundRobinDistributor` je implementací metody *Round Robin* a v tuto chvíli nejjednodušší implementovanou metodou vyvažování zátěže. Tato implementace nevyžaduje žádné vstupní parametry a pracuje pouze s počtem klientů. Z diagramu tříd (obrázek 16) pak vidíme, že o vytváření instancí se stará třída `RoundRobinDistributorFactory`.

```
NetworkConnectionVector * RoundRobinDistributor::OptimalNodes()
{
        NetworkConnectionVector * selected = new NetworkConnectionVector();
        wxCriticalSectionLocker lock(clientLock);
        int size = computingClients.size();
        if (size == 0) {
                return selected;
        }
        lastSelected = lastSelected % size;
        selected->push_back(computingClients[lastSelected]);
        lastSelected++;

        return selected;
}
```

Výpis 3: Metoda `OptimalNodes` (`RoundRobinDistributor`)

Jelikož je třída odvozena od svého předka, který definuje základní metody, implementuje pouze metodu `OptimalNodes`. Ze zdrojového kódu metody (výpis 3) je patrné, že implementace vrací vektor (seznam) spojení, kterým mají být data odeslána. Hned na začátku metody se tedy vytvoří vektor neobsahující žádné prvky. Poté se pomocí vytvoření objektu `wxCriticalSectionLocker`[28] uzamkne sdílená proměnná a vybere

[28] `wxCriticalSectionLocker` je pomocná třída pro použití objektů třídy `wxCriticalSection`. Tato

následující dostupné spojení. Pokud by měl být vybrán neexistující klient (s pořadovým číslem větším, než je počet klientů), tak bude vybrán klient první.

7.3.2 WeightedRoundRobinDistributor

Implementace metody *Weighted Round Robin*. Z výpisu 4 je patrné, že implementace metody `OptimalNodes` je složitější než implementace u předchozí varianty.

Opět se nejprve vytvoří prázdný vektor, který je použit jako výstup z metody. Poté se pomocí `while` cyklu provádí výběr uzlu na základě jeho váhy. Pokud by měly všechny uzly stejnou váhu (například 1), jedná se o metodu *Round Robin*.

Princip algoritmu je takový, že se nejprve zvýší číslo uzlu, který má být vybrán, a pokud se jedná o první uzel (pozice 0), je proveden výpočet při kterém se upraví váha tak, aby odpovídala rozdílu předchozí vybrané váhy a největšího společného dělitele[29] množiny vah všech uzlů. Největší společný dělitel v tomto případě slouží k určení optimálního kroku cyklu. Jsou předpokládány váhy 8, 16, 200, které mají největší společný dělitel s hodnotou 8, pak bude minimální požadovaná váha (ve výpisu `currentWeight`) snižována vždy o 8. Pokud je po odečtení skoku váhy získána záporná hodnota minimální váhy, je nastavena hodnota váhy na maximální hodnotu v množině vah. V případě, že jsou všechny váhy nulové a jsou k dispozici nějaké spojení, je vráceno první spojení v seznamu.

Ať se již jedná o identifikátor prvního uzlu nebo nikoli, tak pokud není hodnota všech vah nulová, provede se kontrola, zda spojení může být přidáno do výstupní množiny. Nejprve je získáno spojení, jenž je reprezentováno identifikátorem `lastSelected` a na základě jeho váhy (skóre) je rozhodnuto, zda může být použito. Pokud může být použito, tak je přidáno do výstupní množiny a ta vrácena jako výsledek volání metody. Pokud nemůže být použito (jeho váha je menší než minimální požadovaná), tak je celý proces opakován se zvýšeným identifikátorem `lastSelected`.

Pokud by byl požadován důkaz, že algoritmus je funkční, musel by být popsán invariant a variant. Variant v tomto algoritmu je reprezentován snižující se hodnotou proměnné `currentWeight`. Pokud totiž existuje nějaký uzel, který má hodnotu váhy nenulovou, musí být vždy vybrán. Buďto je jeho váha rovna maximální hodnotě všech vah a pak je vybrán po prvním průchodu, a nebo je jeho hodnota nižší a je vybrán v jednom z dalších kol. Invariantem je pak tvrzení, že `lastSelected` (tedy pořadové číslo spojení) je vždy v rozmezí nula až počet spojení a zároveň je váha vybraného spojení větší nebo rovna váze kola (uzel s maximální hodnotou váhy tedy bude vybrán ve všech kolech).

---

pomocná třída vstoupí do kritické sekce ve svém konstruktoru a opustí ji ve svém destruktoru. Objekt `wxCriticalSectionLocker` bude, vzhledem k deklaraci, automaticky zničen na konci metody, a tím uvolní zámek sdílené proměnné.[31]

[29] Největší společný dělitel, často značený jako *gcd*, dvou čísel je největší číslo takové, že beze zbytku dělí obě čísla. V případě množiny čísel se jedná o takové číslo, které beze zbytku dělí všechna čísla v množině.[32]

```
NetworkConnectionVector * WeightedRoundRobinDistributor::OptimalNodes()
{
        NetworkConnectionVector * selected = new NetworkConnectionVector();
        while (true) {
                // move to the next node
                lastSelected = (lastSelected + 1) % computingClients.size();
                // IV: lastSelected is between 0 and number of computingClients
                // if is it first node then recalculate sequence
                if (lastSelected == 0) {
                        currentWeight = currentWeight - gcd();
                        if (currentWeight <= 0) {
                                // current weight can not be negative
                                currentWeight = maxWeight();
                                if (currentWeight == 0) {
                                        // all weight are 0
                                        if (!computingClients.empty()) {
                                                selected->push_back(computingClients
                                                    [0]);
                                        }
                                        return selected;
                                }
                        }
                }
                NetworkConnection * c = computingClients[lastSelected];
                if (nodeInfoHolder->GetScore(c) >= currentWeight) {
                        selected->push_back(c);
                        return selected;
                }
        }
}
```

Výpis 4: Metoda `OptimalNodes` (`WeightedRoundRobinDistributor`)

Pro uchovávání skóre jednotlivých spojení s používá rozhraní `NodeInfoHolder`. Konkrétně je v aktuálním sestavení aplikace VRUT vždy používána implementace `TileLatencyNodeInfoHolder`, která vyhodnocuje skóre spojení na základě doby zpracování a zpoždění komunikace využitím informací o odeslání **Event::RENDER_TILE** a přijetí **Event::RENDER_TILE_DONE**.

7.3.3 NullNodeInfoHolder

`NullNodeInfoHolder` je implementací rozhraní `NodeInfoHolder` a je možné jej použít v případě, že není potřeba uchovávat informace o jednotlivých spojeních. Implementace této třídy vychází z návrhového vzoru *Null Object*.[33]

7.3.4 TileLatencyNodeInfoHolder

`TileLatencyNodeInfoHolder` je rovněž implementací rozhraní `NodeInfoHolder`, avšak oproti předchozí variantě uchovává informace o spojení.

Implementace využívá informací z odesílaných událostí typu **Event::RENDER_TILE** a příchozích událostí typu **Event::RENDER_TILE_DONE** k tomu, aby vypočítala čas zpracování jednotlivých dlaždic. Z události **Event::RENDER_TILE** extrahuje informaci o dlaždici, scéně a rámci a na základě této informace vytvoří záznam v tabulce, ke

kterému přiřadí čas odeslání. Při přijetí **Event::RENDER_TILE_DONE** provede stejnou extrakci informací, aby vyzvedl dříve uložený záznam. Následně odečte čas odeslání od aktuálního času, a tak získá dobu zpracování společně se zpožděním komunikace.

Kromě úpravy skóre uzle na základě doby výpočtu a zpoždění se mění skóre uzlu pokaždé, když je událost **Event::RENDER_TILE** odeslána nebo přijata událost **Event::RENDER_TILE_DONE**. Skóre uzlu je při odeslání dat vždy sníženo a při přijetí zvýšeno. Pokud se jedná o průměrně rychlý uzel, mělo by skóre po odeslání a přijetí dlaždice zůstat beze změny.

Kapitola 8
Optimalizace přenášených dat

Kromě komprese a řízení datového toku muže být také využita znalost systému a přenášet pouze ta data, která jsou v danou chvíli opravdu potřebná. Také může být informace určitým způsobem omezena bez toho, aniž byl použit nějaký kompresní algoritmus a bylo ovlivněno vnímání vykreslené scény.

Pokud je vynechán první byte, který je nutný a pro všechny události VRUTu přenášené pomocí sítě potřebný, je získán datový blok události **Event::RENDER_TILE_DONE** popsaný na obrázku 18. Datový blok této události se skládá z identifikace rámce a dlaždice. Také obsahuje vlastní obrazová data dlaždice a informace potřebné k jejímu zpracování (počet pixelů kupříkladu).

Vynecháním některých informací (například `workerId`) nebo omezením jejich rozsahu může být výrazně zmenšeno množství přenášených informací. Například již zmíněná položka `workerId` by mohla být vynechána (nepodařilo se mi dohledat její použití), nebo alespoň redukována na menší rozsah (4 294 967 296 možných pracovních vláken by i v případě použití 32 vláken na klienta znamenalo možných 134 217 728 výpočetních klientů a to jen v případě, že by identifikátor pracovního vlákna byl jedinečný).

Podobně víme, že podle vzorkovací strategie a kola se přenáší různé množství obrazových dat, kde ne vždy tato data tvoří celou dlaždici. Respektive nejprve bývá vrženo menší množství paprsků (například čtyři) na dlaždici a postupným přidáváním paprsků v dalších kolech se výpočet zpřesňuje.

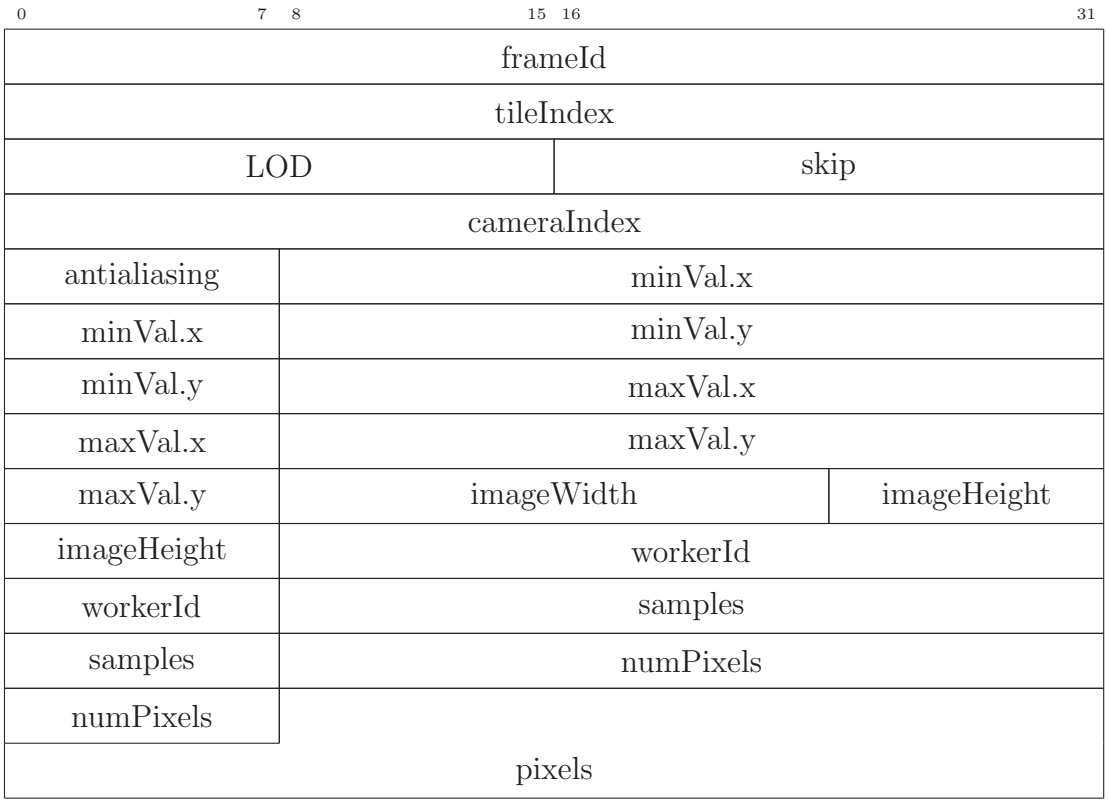
8.0.1 Přenos pouze vypočtených bodů

V úvodní části kapitoly jsem zmínil, že podle vzorkovací strategie a výpočetního kola se liší počet skutečně vypočtených bodů. Optimalizace spočívá v tom, že jak server, tak klient ví, jaká strategie je použita, a tak mohou správně vypočíst a zobrazit dané body. S využitím této znalosti mohou být modifikována serializovaná data tak, aby v případě vybrání čtyř pixelů, byly opravdu přenášeny jen tyto čtyři pixely.

Vzhledem k tomu, že pan A. Míšek v jednom z rozhovorů zmínil, že na této optimalizaci již pracuje jeden z jeho kolegů, nevěnoval jsem se analýze tohoto problému a ani řešení.

8.0.2 Konverze 32 na 16

Pro přenášení obrazových dat se používá struktura `Vector4`. Tato struktura se sestává ze čtyř hodnot datového typu float. Jazyk C má datový typ float definovaný jako nejméně

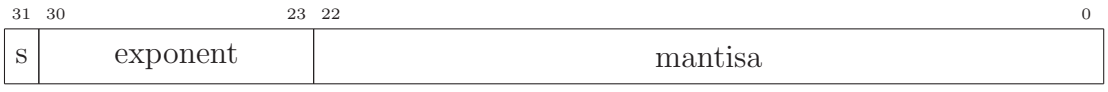


Obrázek 18: Velikost složek **Event::RENDER_TILE_DONE**

32 bitové číslo dle normy IEEE-754. [20] Norma uvádí konkrétní reprezentaci desetinného čísla s plovoucí řádovou čárkou a to pomocí znaménkového bitu (s), mantisy (m) a exponentu (e) (obrázek 19). Tato reprezentace využívá aditivního kódování.[35]

Výsledná hodnota čísla (X) je pak definována jako $X = (-1)^s \times 2^{E-127} \times (1 + Q)$. Hodnoty E a Q jsou pak dány binární reprezentací exponentu a mantisy. Zde je zajímavý fakt, že exponent je vyjádřen všemi svými bity (tedy $E = 2^7 \times e_7 + 2^6 \times e_6 + \cdots + 2^1 \times e_1 + e_0$), zatímco mantisa využívá jednoho „skrytého“ bitu (m_0), který je k ní ve formuli přičten vždy jako 1 ($\ldots (1 + Q)$).

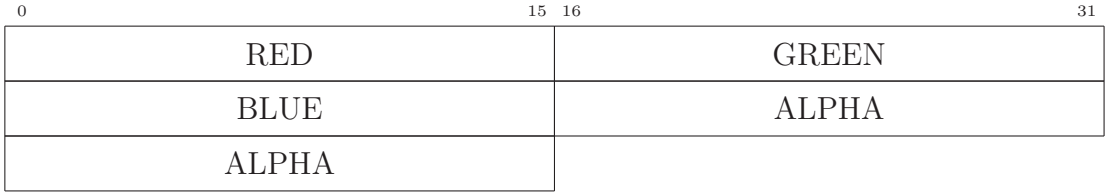
32 bitový float, používaný ve struktuře `Vector4`, je znázorněn na obrázku 19. Složky ve struktuře `Vector4` reprezentují *červenou*, *zelenou* a *modrou* barvu. Poslední složka pak reprezentuje průhlednost (obrázek 20).



Obrázek 19: Float 32

Z předchozích částí víme, že nemůže být omezen obsah průhlednosti, neboť složka ve skutečnosti obsahuje dvě dvoubytové hodnoty. Pokud by byly omezeny barevné složky na 16 bitů, tak bude dosaženo úspory 48 bitů na pixel (obrázek 21).

Obrázek 20: Velikost složek jednoho pixelu



Obrázek 21: Velikost složek jednoho optimalizovaného pixelu

Jelikož jazyk C, potažmo C++, nepodporuje čísla s plovoucí řádovou čárkou o rozsahu nižším 32 bitů, musel jsem implementovat pomocné typy a metody, které mi tento převod umožnili.

Prvním takovým typem je `union` reprezentující 16 bitový float. Tento typ vidíte ve výpisu 5. Definován je jako datový typ `union` a to proto, abychom mohli v případě 32 bitové varianty volně přecházet mezi reprezentací pomocí standardního datového typu float. Složka *bits* je definována jako struktura, jenž obsahuje tři členy označené jako m, e a s. Člen m reprezentuje mantisu a má k dispozici 10 bitů. Člen e obsahuje datovou reprezentaci exponentu, k dispozici má 5 bitů. Poslední ze členů, tedy s, je nositelem informace o znaménku.

V implementaci (výpis 5) je využito vlastnosti jazyka C, která umožňuje u datového typu `struct` definovat přesný počet bitů, jenž má být na složku použit.

```
typedef union
{
        struct
        {
                unsigned short m : 10;
                unsigned short e : 5;
                unsigned short s : 1;
        } bits;
} float16_s;
```

Výpis 5: Struktura reprezentující 16 bitový float

K převodu z 32 bitové (f32) varianty na 16 bitovou (f16) je využit tento postup: nejprve je zkopírován znaménkový bit, tedy f16.s = f32.s. Následně je provedena úprava exponentu; exponent je posunut na nulovou pozici (aditivní kódování) $l = \lfloor \text{f32.e} - 127 \rfloor$. Hodnoty exponentu jsou omezeny a je k nim přičteno číslo 15 (bias) $\text{f16.e} = \max(-15, \min(16, l)) + 15$. Poslední, co zbývá, je dosadit hodnotu pro mantisu

f16.m = f32.m << 13.

Uvedený postup využívá metoda `float32to16` (Výpis 6), která je použita v serializační třídě `VRUTSerializer`. Nutno podotknout, že tímto převodem je ztracena přesnost desetinného čísla.

```
inline void float32to16(float x, float16_s * f16)
{
        float32_s f32 = { x };
        // sign bit
        f16->bits.s = f32.bits.s;
        // exponent
        f16->bits.e = std::max(-15, std::min(16, (int)(f32.bits.e - 127))) + 15;
        // mantisa
        f16->bits.m = f32.bits.m >> 13;
}
```

Výpis 6: Zdrojový kód metody `float32to16`

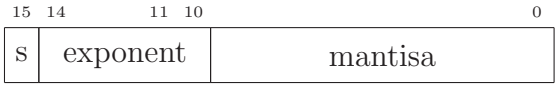
Při deserializaci dat bude postup podobný, jen opačným směrem. Zdrojový kód metody `float16to32` je uveden ve výpisu 7. Přesnost byla samozřejmě ztracena, a tak je výsledný rozsah hodnot stejný, jako kdyby byl použit 16 bitový float.

```
inline float float16to32(float16_s f16)
{
        float32_s f32;
        // sign bit
        f32.bits.s = f16.bits.s;
        // exponent
        f32.bits.e = (f16.bits.e - 15) + 127;
        // mantisa
        f32.bits.m = ((uint32_t)f16.bits.m) << 13;

        return f32.v;
}
```

Výpis 7: Zdrojový kód metody `float16to32`

16 bitový float je téměř identický se svou 32 bitovou variantou. Jediným rozdílem je počet bitů dostupných pro exponent (e) a mantisu (m) (obrázek 22).



Obrázek 22: Float 16

Kapitola 9
Komplikace

Tato práce se, jak již bylo zmíněno, zakládá na projektu VRUT, který je již delší dobu ve vývoji a obsahuje mnoho modulů. Projekt je v jazyce C++ a využívá pokročilé grafické algoritmy, které pracují ve více vláknech. Před započetím práce na optimalizaci síťového provozu bylo nutné nejprve zprovoznit komunikaci po síti s využitím modulů *Cluster2* a *RayTracer*, která nebyla implementována v době započetí této práce.

Na počátku práce jsem musel řešit konflikty vláken při přijímání zpráv po síti, opravu přenášených dat (kdy některé údaje chyběly), synchronizaci zobrazované scény a počítaných dlaždic. Tato práce pro mne byla o to těžší, protože nemám zkušenosti v jazyce C/C++. Žádnou z použitých knihoven (mimo standardní knihovny C++) jsem neznal a musel jsem se s nimi tedy seznamovat. Tento problém byl nakonec vyřešen jak mnou, tak ve společnosti ŠKODA AUTO a.s.. Zvoleno bylo řešení ŠKODA AUTO a.s..

Také byl problém s modulem *RayTracer*, který na žádném z mých počítačů nezobrazoval scénu. Tento problém byl nakonec vyřešen na katedře počítačové grafiky ČVUT. Konkrétně šlo o přesnost čísel ve formátu `double`, kdy výsledná čísla byla označena jako „nepřesná“ a shader tedy vykreslil černou plochu místo obrazových dat dlaždice.

Kapitola 10
Závěr

Na základě požadavků zadavatele a porovnávaných možností se implementoval model, při kterém server komunikuje s každým výpočetním klientem samostatně. Komunikace s těmito klienty je řízena navrženou a implementovanou vyvažovací komponentou s aktuální podporou dvou scénářů a to *Round Robin*, kdy jsou o výpočet žádány postupně všichni výpočetní klienti, a *Weighted Round Robin* s úpravou vah za běhu na základě výkonu výpočetního klienta.

Snížení datového toku bylo momentálně dosaženo jen použitím 16 bitové varianty čísel s plovoucí řádovou čárkou. Komprese, která z porovnání vyšla nejlépe (tedy pomocí Deflate algoritmu), není momentálně implementována. Podpora komprese dlaždic v rámci aplikace VRUT by měla být implementována, alespoň dle zjištěného průběhu komunikace, ve třídě `VRUTSerializer`.

Momentální stav tedy dovoluje distribuci dlaždic na výpočetní klienty a povolení automatické úpravy poměru lokálních a vzdálených dlaždic. Pomocí omezení barevné hloubky bylo dosaženo teoretické úspory 37,5 % na plné (všechny barevné hodnoty) dlaždici o velikosti 32×32 obrazových bodů a tak celkovému snížení datového toku o 37 %[30] (z 78 Gb/s na 49 Gb/s). Přenosová kapacita 49 Gb/s by nám stačila uvnitř clusteru, kde jsou jednotlivé uzly schopné komunikovat až rychlostí 56 Gb/s.

FPS	čas přenosu	1 Gb/s	10 Gb/s
15	67 ms	5717	57173
24	42 ms	3584	35840
30	34 ms	2901	29013
60	17 ms	1450	14506
100	10 ms	853	8533

Tabulka 6: Počet přenesených 1,7 kB bloků (gzip) za snímek

FPS	čas přenosu	1 Gb/s	10 Gb/s
15	67 ms	857	8576
24	42 ms	537	5376
30	34 ms	435	4352
60	17 ms	217	2176
100	10 ms	128	1280

Tabulka 7: Počet přenesených 10 kB bloků (16 bitů na barvu) za snímek

Aby byly naplněny požadavky zadavatele ŠKODA AUTO a.s. na přenos obrazu mimo cluster, je ještě nutno implementovat kompresi dlaždic nebo celkového obrazu a snížit tak datový tok pod 10 Gb/s. Zjistili jsme, že pomocí komprese dlaždic jsme teoreticky schopni tuto úlohu realizovat, protože bylo dosaženo teoretické úspory až

[30] všechny hodnoty datového toku jsou zaokrouhleny.

90 % (7,25 Gb/s dle kapitoly 6.0.4 GZIP).

Z tabulek 6 a 7 je patrné, že, aby zadavatel mohl použít zařízení HTC Vive s 5000 bloky o rozměrech 32×32 obrazových bodů, musel by se buď spokojit s 15 snímky za sekundu, a nebo obraz počítat v rámci clusteru a výsledný obraz přenášet pomocí rozdílové komprese některého z video kodeků.

Literatura

[1] Frame rate. In: *Wikipedia: the free encyclopedia* [online]. San Francisco (CA): Wikimedia Foundation, 2001- [cit. 2017-12-26]. Dostupné z: https://en.wikipedia.org/wiki/Frame_rate

[2] HTC Vive and Oculus Rift Hardware Requirements Explained. *Logical Increments* [online]. 2016 [cit. 2017-12-26]. Dostupné z: http://blog.logicalincrements.com/2016/08/htc-vive-oculus-rift-hardware-requirements-explained/

[3] Distributed Architecture. *Tutorialspoint.com* [online]. [cit. 2017-12-26]. Dostupné z: https://www.tutorialspoint.com/software_architecture_design/distributed_architecture.htm

[4] *Internet architecture for application service providers* [online]. 2002 [cit. 2017-12-26]. Dostupné z: http://ieeexplore.ieee.org/document/926872/

[5] Cloud Gaming: Gaming as a Service (GaaS). *NVIDIA* [online]. [cit. 2017-12-27]. Dostupné z: http://www.nvidia.com/object/cloud-gaming.html

[6] Foveated video streaming for cloud gaming. *IEEE Explore* [online]. 2017 [cit. 2017-12-27]. Dostupné z: http://ieeexplore.ieee.org/document/8122235/

[7] RYAN, Shea, Liu JIANGCHUAN, Ngai EDITH C.-H. a Cui YONG CUI. Cloud gaming: architecture and performance. *IEEE Network: the magazine of global information exchange* [online]. 2013 [cit. 2017-12-27]. DOI: 0890-8044. Dostupné z: http://ieeexplore.ieee.org/document/6574660/

[8] CRUZ-NEIRA, Carolina, Daniel J. SANDIN, Thomas A. DEFANTI, Robert V. KENYON a John C. HART. The CAVE: audio visual experience automatic virtual environment. *Communications of the ACM* [online]. New York, NY, USA, 1992, 1992, 35(6) [cit. 2017-12-28]. Dostupné z: https://dl.acm.org/citation.cfm?doid=129888.129892

[9] HTC Vive. In: *Wikipedia: the free encyclopedia* [online]. San Francisco (CA): Wikimedia Foundation, 2001- [cit. 2017-12-28]. Dostupné z: https://en.wikipedia.org/wiki/HTC_Vive

[10] Czech Republic. *Speedtest* [online]. 2017-10 [cit. 2017-12-28]. Dostupné z: https://www.speedtest.net/global-index/czech-republic

[11] Peak signal-to-noise ratio. In: *Wikipedia: the free encyclopedia* [online]. San Francisco (CA): Wikimedia Foundation, 2001- [cit. 2017-12-30]. Dostupné z: https://en.wikipedia.org/wiki/Peak_signal-to-noise_ratio

[12] SSIM. In: *Wikipedia: the free encyclopedia* [online]. San Francisco (CA): Wikimedia Foundation, 2001- [cit. 2017-12-30]. Dostupné z: https://cs.wikipedia.org/wiki/SSIM

[13] WANG, Z., A.C. BOVIK, H.R. SHEIKH a E.P. SIMONCELLI. *Image Quality Assessment: From Error Visibility to Structural Similarity. IEEE Transactions on Image Processing* [online]. 2004, 13(4), 600-612 [cit. 2017-12-30]. DOI: 10.1109/TIP.2003.819861. ISSN 1057-7149. Dostupné z: http://ieeexplore.ieee.org/document/1284395/

[14] Performance. *H.264 and MPEG-4 video compression: video coding for next generation multimedia*. Hoboken, NJ: Wiley, c2003, s. 246-255. ISBN 0-470-84837-5.

[15] Rok video formátu VP9: zaplavil YouTube, dekódování je 7x a enkódování 40x rychlejší. Cnews.cz [online]. Mladá fronta, 2014-06-30 [cit. 2017-12-30]. Dostupné z: https://www.cnews.cz/rok-video-formatu-vp9-zaplavil-youtube-dekodovani-je-7x-a-enkodovani-40x-rychlejsi/

[16] *Recommendation ITU-R BT.2020-2: Parameter values for ultra-high definition television systems for production and international programme exchange*. 2015. Dostupné také z: https://www.itu.int/dms_pubrec/itu-r/rec/bt/R-REC-BT.2020-2-201510-I!!PDF-E.pdf

[17] HUDSON, Graham, Alain LEGER, Birger NISS a Istvan SEBESTYEN. *JPEG at 25: Still Going Strong. IEEE MultiMedia* [online]. 2017, 24(2), 96-103 [cit. 2017-12-30]. DOI: 10.1109/MMUL.2017.38. ISSN 1070-986x. Dostupné z: http://ieeexplore.ieee.org/document/7924246/

[18] *Human Vision Light, Color, Eyes, etc*. Dostupné také z: https://people.cs.umass.edu/ elm/Teaching/ppt/691a/CV%20UNIT%20Light/691A_UNIT_Light_1.ppt.pdf

[19] UW CSE VISION FACULTY. *Color*. Dostupné také z: https://courses.cs.washington.edu/courses/cse455/09wi/Lects/lect11.pdf

[20] Fundamental types. *C++ reference* [online]. [cit. 2017-12-31]. Dostupné z: http://en.cppreference.com/w/cpp/language/types

[21] BOUTELL.COM, INC. *PNG (Portable Network Graphics) Specification*. 1997. Dostupné také z: https://tools.ietf.org/html/rfc2083#page-7

[22] W3C. *Portable Network Graphics (PNG) Specification (Second Edition)*. 2003. Dostupné také z: https://www.w3.org/TR/2003/REC-PNG-20031110/#11IDAT

[23] *Zlib: A Massively Spiffy Yet Delicately Unobtrusive Compression Library* [online]. [cit. 2017-12-31]. Dostupné z: https://zlib.net/

[24] Gzip. In: *Wikipedia: the free encyclopedia* [online]. San Francisco (CA): Wikimedia Foundation, 2001- [cit. 2017-12-31]. Dostupné z: https://cs.wikipedia.org/wiki/Gzip

[25] Task Queue Overview. *Google Cloud Platform Documentation* [online]. [cit. 2017-12-31]. Dostupné z: https://cloud.google.com/appengine/docs/standard/java/taskqueue/

[26] Load Balancing Algorithms. *KEMP: APPLICATION DELIVERY* [online]. [cit. 2017-12-31]. Dostupné z: https://kemptechnologies.com/emea/load-balancer/load-balancing-algorithms-techniques/

[27] Load Balancing Methods for Every Application. *Peplink* [online]. [cit. 2017-12-31]. Dostupné z: https://www.peplink.com/technology/load-balancing-algorithms/

[28] Weighted Round-Robin Scheduling. *LVS Knowledge Base* [online]. [cit. 2017-12-31]. Dostupné z: http://kb.linuxvirtualserver.org/wiki/Weighted_Round-Robin_Scheduling

[29] ARLOW, Jim a Ila NEUSTADT. UML 2 a unifikovaný proces vývoje aplikací: objektově orientovaná analýza a návrh prakticky. 2., aktualiz. a dopl. vyd. Brno: Computer Press, 2007. ISBN 978-80-251-1503-9.

[30] Hook. *WhatIs.com* [online]. [cit. 2018-01-01]. Dostupné z: http://whatis.techtarget.com/definition/hook

[31] WxCriticalSectionLocker Class Reference. *WxWidgets: Cross-Platform GUI Library* [online]. [cit. 2018-01-02]. Dostupné z: http://docs.wxwidgets.org/3.0/classwx_critical_section_locker.html

[32] Největší společný dělitel. In: *Wikipedia: the free encyclopedia* [online]. San Francisco (CA): Wikimedia Foundation, 2001- [cit. 2018-01-02]. Dostupné z: https://cs.wikipedia.org/wiki/Nejv%C4%9Bt%C5%A1%C3%AD_spole%C4%8Dn%C3%BD_d%C4%9Blitel

[33] Null Object Design Pattern. *SourceMaking* [online]. [cit. 2018-01-02]. Dostupné z: https://sourcemaking.com/design_patterns/null_object

[34] MÍŠEK, Antonín et al. *VURT: Dokumentace aplikace*. 2017.

[35] SPONSOR a MICROPROCESSOR STANDARDS COMMITTEE OF THE IEEE COMPUTER SOCIETY. *IEEE standard for floating-point arithmetic* [online]. New York, NY: Institute of Electrical and Electronics Engineers, 2008 [cit. 2018-01-03]. ISBN 978-073-8157-528.

Obsah CZ

/
- `changes_in_vrut.txt` ..... Změny provedené v rámci aplikace VRUT
- `vorislu1_diploma_thesis.pdf` ..... Text práce elektronicky
- `vorislu1_diploma_thesis_src` ..... Zdrojové soubory textu
- `tile_base_image_1.jpg` .... Obrázek ze kterého byly tvořeny testovací dlaždice
- `tile_base_image_2.jpg` .... Obrázek ze kterého byly tvořeny testovací dlaždice
- `NetworkSimulator` ..... Simulátor síťového provozu