VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
FACULTY OF INFORMATION TECHNOLOGY

ÚSTAV POČÍTAČOVÉ GRAFIKY A MULTIMÉDIÍ
DEPARTMENT OF COMPUTER GRAPHICS AND MULTIMEDIA

WEBGL2 RENDERER VE WEBASSEMBLY

WEBGL2 RENDERER IN WEBASSEMBLY

SEMESTRÁLNÍ PROJEKT
TERM PROJECT

AUTOR PRÁCE — Bc. PAVEL REŽŇÁK
AUTHOR

VEDOUCÍ PRÁCE — Ing. TOMÁŠ STARKA
SUPERVISOR

BRNO 2018

Vysoké učení technické v Brně - Fakulta informačních technologií

Ústav počítačové grafiky a multimédií Akademický rok 2017/2018

Zadání diplomové práce

Řešitel: **Režňák Pavel, Bc.**
Obor: Inteligentní systémy
Téma: **WebGL2 renderer ve WebAssembly**
WebGL2 Renderer in WebAssembly
Kategorie: Počítačová grafika

Pokyny:

1. Nastudujte standard Webgl2 (OpenGL ES 3.0), WebAssembly, Typescript a související technologie a vhodné nástroje, např. Emscripten.
2. Navrhněte spolu s vedoucím strukturu rendereru.
3. Implementujte renderer do již existujícího frameworku.
4. Vytvořte plakát nebo video k práci.

Literatura:

- Po dohode s vedoucim

Při obhajobě semestrální části projektu je požadováno:

- Body 1 a část 2.

Podrobné závazné pokyny pro vypracování diplomové práce naleznete na adrese http://www.fit.vutbr.cz/info/szz/

Technická zpráva diplomové práce musí obsahovat formulaci cíle, charakteristiku současného stavu, teoretická a odborná východiska řešených problémů a specifikaci etap, které byly vyřešeny v rámci dřívějších projektů (30 až 40% celkového rozsahu technické zprávy).

Student odevzdá v jednom výtisku technickou zprávu a v elektronické podobě zdrojový text technické zprávy, úplnou programovou dokumentaci a zdrojové texty programů. Informace v elektronické podobě budou uloženy na standardním nepřepisovatelném paměťovém médiu (CD-R, DVD-R, apod.), které bude vloženo do písemné zprávy tak, aby nemohlo dojít k jeho ztrátě při běžné manipulaci.

Vedoucí: **Starka Tomáš, Ing.**, UPGM FIT VUT
Datum zadání: 1. listopadu 2017
Datum odevzdání: 23. května 2018

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
Fakulta informačních technologií
Ústav počítačové grafiky a multimédií
612 66 Brno, Božetěchova 2

L.S.

doc. Dr. Ing. Jan Černocký
vedoucí ústavu

Abstrakt

Diplomová práce se zaměřuje na rychlé vykreslení 3D scény ve webovém prohlížeči s použitím moderních technologií, jako je WebGL a WebAssembly. V textu je popsán proces překladu aplikace psané v jazyce C++ do WebAssembly pomocí překladače Emscripten a její začlenění do webové stránky. Dále je rozebrán obousměrný způsob komunikace mezi jazykem C++ a JavaScriptem, jako je volání funkcí, vytváření tříd či sdílení paměti. Během návrhu vykreslovacího jádra jsou vzpomenuty některé způsoby a metody optimalizace vykreslování. Na závěr jsou jednotlivé technologie porovnány z hlediska jejich výkonu.

Abstract

This thesis is focused on fast rendering of the 3D scene in a web browser with usage of modern technologies, for instance WebGL and WebAssembly. In this thesis you will find out how to compile an application which was written in C++ language into WebAssembly via Emscripten compilator and how to insert this code into a web page. Futhermore, you will find out how to communicate between C++ language and JavaScript, how to call functions, create instances and how to share memory between them. During design of a rendering core you will learn a few methods how to improve rendering performance. In the end the performance of this technologies is compared.

Klíčová slova

Renderování na webu, WebGL, asm.js, WebAssembly, Emscripten

Keywords

Web rendering, WebGL, asm.js, WebAssembly, Emscripten

Citace

REŽŇÁK, Pavel. *WebGL2 renderer ve WebAssembly.* Brno, 2018. Semestrální projekt. Vysoké učení technické v Brně, Fakulta informačních technologií. Vedoucí práce Ing. Tomáš Starka

WebGL2 renderer ve WebAssembly

Prohlášení

Prohlašuji, že jsem tuto diplomovou práci vypracoval samostatně pod vedením pana Ing. Tomáše Starky. Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

. . . . . .
Pavel Režňák
2. května 2018

Obsah

1 Úvod **2**

2 Technologie použitelné v moderních webových prohlížečích **4**
2.1 Javascript 4
2.2 WebGL 5
2.3 WebGL 2 5
2.4 WebAssembly 8
2.5 Emscripten 9

3 Návrh rendereru a jeho začlenění do webové aplikace **18**
3.1 Analýza existující aplikace 18
3.2 Návrh nového vykreslovacího modulu 20
3.3 Návrh obecné části modulu 24
3.4 Návrh vykreslovací části modulu 26

4 Implementace vykreslovacího modulu **29**
4.1 Wrapper – WasmRenderGraph 29
4.2 Implementace obecné části modulu 32
4.3 Přehled použitých manažerů 33
4.4 Implementace vykreslovací části modulu 36

5 Porovnání existujícího řešení s novým vykreslovacím modulem **42**
5.1 Měření výkonnosti 43

6 Závěr **45**

Literatura **46**

A Obsah přiloženého paměťového média **49**

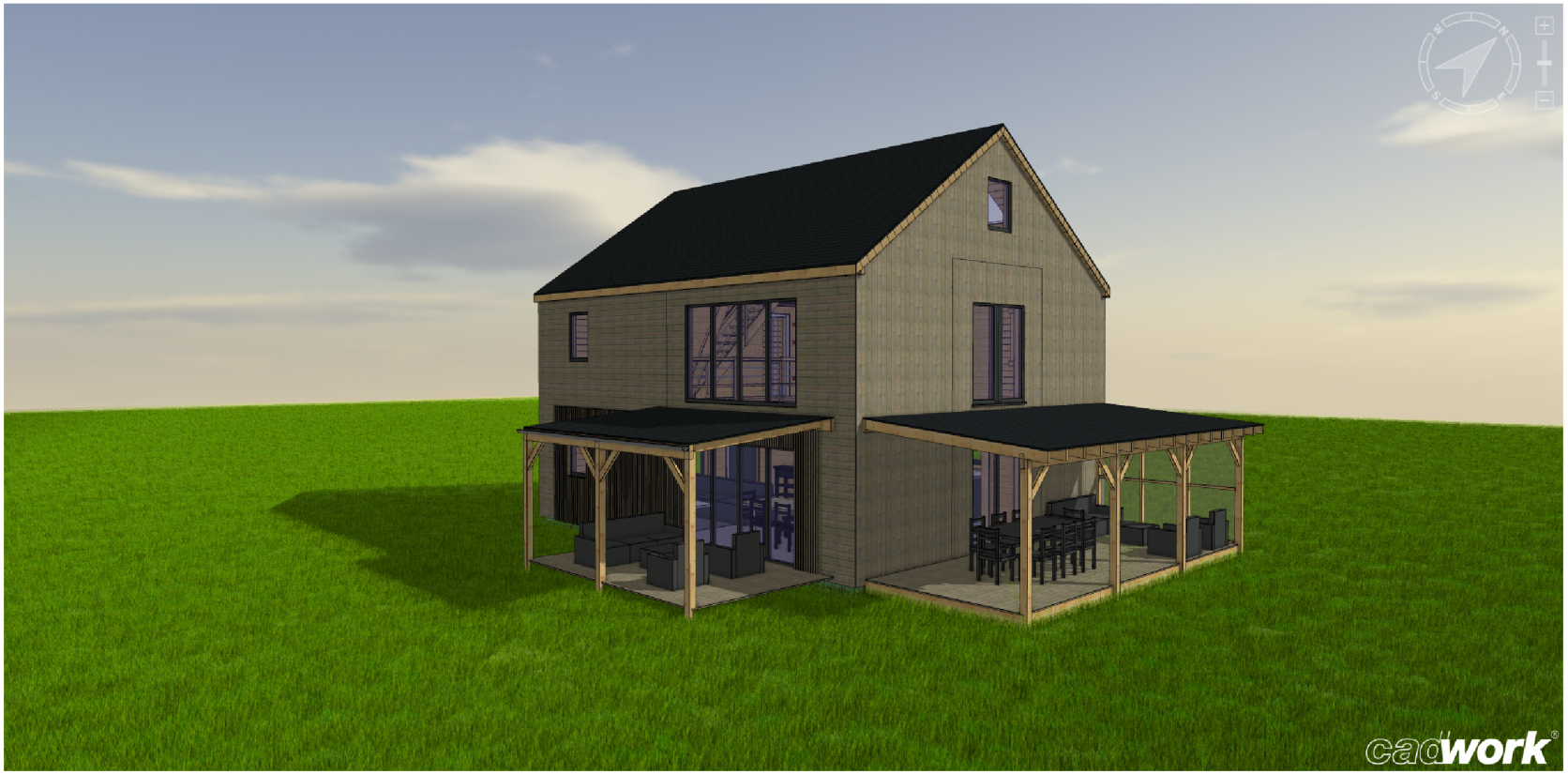
Kapitola 1

Úvod

Webový prohlížeč je jedním z často používaných programů na stolních počítačích, notebooocích a i na "chytrých" mobilních zařízeních s některým z dostupných operačních systémů. Možnosti těchto prohlížečů se stále rozšiřují, vznikají nové standardy a technologie, se kterými můžou programátoři a web designeři pracovat. Před několika lety byla do prohlížečů přidána vlastnost umožňující komunikaci s grafickou kartou, která vychází z OpenGL ES, tento nový standard se jmenuje WebGL a aktuálně se začíná rozšiřovat jeho druhá verze. V momentě, kdy bylo možné vytvářet graficky náročné aplikace, začaly vznikat i požadavky na zrychlení aplikací psaných pro prohlížeče, protože běžně používaný jazyk (JavaScript) nenabízí rychlost srovnatelnou s nativním kódem. Z tohoto důvodu se vyvinul i nový webový formát – WebAssembly, do kterého lze překládat existující aplikace psané v různých jazycích, jako jsou například C, C++ a Rust (lze portovat existující aplikace při splnění určitých podmínek). Ještě před vznikem tohoto formátu byl k dispozici překlad do jazyka asm.js (podmnožina JavaScriptu), který nabízí vysokou optimalizovatelnost v prohlížečích. Dalším důležitým milníkem pro vývoj kvalitních webových aplikací bylo vydání nového standardu značkovacího jazyka – HTML5.

V této diplomové práci jsou rozebrány technologie, které jsou dostupné v moderních prohlížečích, z hlediska rychlého vykreslování 3D scény. Při vývoji byl pro grafické operace použit standard WebGL 2 a z důvodu rychlosti byla aplikace přeložena do WebAssembly. Výsledný program se stal součástí již existující aplikace a v budoucnu bude nahrazovat aktuální vykreslovací část (v prohlížečích podporujících tyto technologie). Na závěr je přeložena nově implementovaná část i do asm.js a je porovnána rychlost těchto dvou vygenerovaných kódů (asm.js může sloužit pro zpětnou kompatibilitu s prohlížeči nepodporující WebAssembly). Na závěr jsou porovnány rychlosti původní a nové části.

Nový renderer vychází z existující aplikace, jejíž výstup je znázorněn na obrázku 1.1. Z obrázku je patrné, že aplikace zvládá vykreslit model obsahující objekty s texturami, s průhlednými částmi a i s ohraničením. Dále je k modelu vykreslena scéna s oblohou, měkkými stíny (shadow maps a ambient occlusion) a trávou (instancing). V práci jsou popsány principy, jak některé z těchto věcí vykreslit, a způsob jejich implementace.



Obrázek 1.1: Výstup z existující aplikace. Aplikace podporuje mnoho různých typů geometrie, formátů textur i efektů.

Kapitola 2

Technologie použitelné v moderních webových prohlížečích

Mezi moderní webové prohlížeče, které jsou podle statistik jedny z nejvíce používaných na stolních počítačích a noteboocích [10], lze zařadit například Google Chrome [1], Mozilla Firefox [13], Safari [15] a Microsoft Edge [16]. Tyto desktopové prohlížeče dosahují velice dobré podpory pro aplikace psané pomocí moderních standardů a technologií, jako je například HTML5 a WebGL. Webové aplikace využívající WebGL je možné spustit i v mnoha mobilních zařízeních s operačním systémem Android a iOS, kde je nutné vzít v potaz menší množství výpočetního výkonu a paměti.

V této kapitole budou rozebrány dílčí technologie, které budou použity při vývoji vykreslovacího modulu. Mezi ně patří například interpretovaný jazyk JavaScript, který se stal prakticky standardem při vývoji webových aplikací, WebGL pro rychlé vykreslování 3D grafických objektů a WebAssembly jako nový a efektivní způsob, jak nahradit výkonnostně kritické části JavaScriptu.

2.1 Javascript

JavaScript [9] je interpretovaný a nebo tzv. *JIT-compiled* (Just In Time) jazyk používaný často ve webových prohlížečích, ale i v jiných prostředích, jako je Node.js, Apache CouchDB a Adobe Acrobat. Jedná se o prototypový, multi-paradigmatický, dynamický jazyk s podporou pro objektově orientované, imperativní i funkcionální programování. V prohlížečích běží na klientské straně a je často použit k dynamickému chování webové stránky.

Dynamičnost tohoto jazyka mu propůjčuje neuvěřitelnou flexibilitu a sílu, avšak programátor může ztrácet přehled o vlastním kódu. Jednou z náročných činností u tohoto jazyka je například refaktoring. Protože se jazyk nekompiluje, není možné před spuštěním zjistit, zda všechny funkce byly přejmenovány správně, ani zda byly všem upraveným funkcím předány správné parametry. Dynamičnost jazyka nese ještě další problémy – pokud programátor použije proměnnou, u které se omylem přepíše v názvu, aplikace neoznámí neexistenci, pouze vytvoří novou, pak je možné, že se chyba projeví až o několik kroků později. Z těchto a ještě mnoha dalších důvodů vzniklo nad JavaScriptem několik jazyků, které jsou do něj překládány. Jedním z nich je TypeScript od firmy Microsoft [18], který nabízí velice podobný styl psaní programů, ale je možné specifikovat (a tedy i hlídat) datové typy proměnných a jejich existenci. Dalším dostupným řešením je překladač Emscripten, který je popsaný níže.

Propojení WebGL a existujících i nových JavaScriptových aplikací je poměrně jednoduché. Minimalistický příklad vytvoření WebGL aplikace a vykreslení HTML objektu canvas černou barvou je uveden v JavaScript 2.1. Kód provede získání elementu, do kterého se bude kreslit, nastaví kontext a barvu pro vyčištění framebufferu. Před spuštěním je nutné mít existující canvas na webové stránce. Vytvoření WebGL kontextu v jazyce C++ a následný překlad do webového formátu je možné překladačem Emscripten, který je popsaný v sekci 2.5.

```
// in HTML <canvas id="glCanvas" width="..." height="..."></canvas>
var canvas = document.getElementById("glCanvas");
var gl = canvas.getContext("webgl");
gl.clearColor(0.0, 0.0, 0.0, 1.0);
gl.clear(gl.COLOR_BUFFER_BIT);
```

JavaScript 2.1: Jednoduché vytvoření WebGL aplikace v JavaScriptu

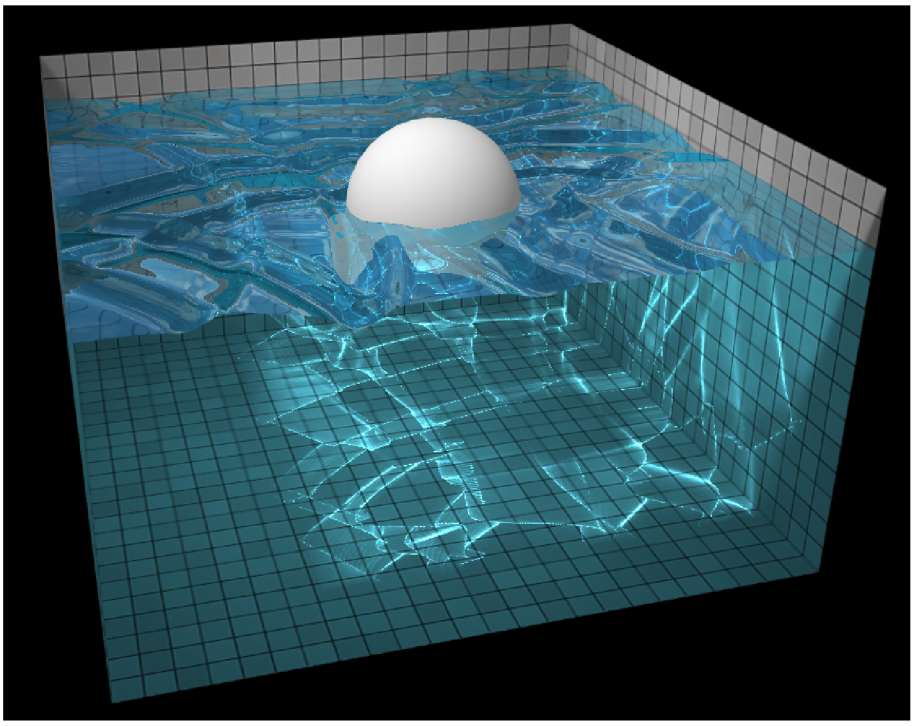
2.2 WebGL

WebGL umožňuje webové aplikaci použít API založené na OpenGL ES 2.0 pro kreslení 2D a 3D objektů na HTML canvas bez použití jakýchkoli zásuvných modulů. Tato WebGL aplikace se skládá z řídícího kódu, napsaném v jazyce JavaScript (nebo jazykem přeložitelným do kompatibilní verze – TypeScript přeložený do JavaScriptu, C++ přeložené překladačem Emscripten do JavaScriptu nebo WebAssembly), a kódu pro shader, napsaném v jazyce GLSL, kde se shader vykonává přímo na grafické kartě (GPU). WebGL objekty mohou být zároveň kombinovány s ostatními HTML objekty na stránce. [11]

První verze WebGL specifikace byla vydána v únoru 2011 a mnoho programátorů začalo vytvářet nádherné vizuální efekty (příkladem může být simulace vody s pokročilým osvětlením a detekcí kolizí zobrazeném na obrázku 2.1), které mohli jednoduše sdílet pomocí webových stránek. Programátor, který napíše aplikaci ve WebGL, může očekávat, že jeho dílo poběží na mnoha různých architekturách (jak stolních, tak mobilních) s akcelerací pomocí grafické karty. Aby toho bylo možno dosáhnout, webový prohlížeč interpretuje jednotlivé příkazy pomocí nativního grafického API v zařízení. Na většině mobilních zařízeních se používá OpenGL ES, a protože je na něm WebGL založeno, jsou si tyto příkazy velice podobné. Větší problém nastává na stolních zařízeních, které neobsahují ovladače pro OpenGL ES. Na systémech Linux a OS X se standardně používá OpenGL, ale například pro Windows se běžně programuje v Direct3D. Jelikož použitím OpenGL na systémech Windows mohou vznikat problémy, firma Google založila nový projekt – ANGLE. Tento projekt začal jako implementace OpenGL ES 2.0 pomocí Direct3D 9 a nyní ho používají například webové prohlížeče Google Chrome a Mozilla Firefox. V roce 2013 bylo do projektu ANGLE přidáno vykreslovací jádro založené na Direct3D 11. [22]

2.3 WebGL 2

Vývoj nové verze začal v roce 2013 a byl dokončen v lednu 2017. WebGL 2 je založeno na OpenGL ES 3.0 a poprvé se objevilo v prohlížečích Firefox 51, Chrome 56 a Opera 43. [17] S novou verzí přišla i nová vylepšení, z nichž se většina zaměřuje na vyšší rychlost a vizuální kvalitu [20]. Původní rozšíření ve WebGL se objevují v základní nabídce WebGL 2. Mezi důležitá vylepšení nové verze WebGL lze vzpomenout aktualizaci jazyka GLSL, podporu



Obrázek 2.1: Pokročilá aplikace napsaná ve WebGL simulující pohyb vodní hladiny, odlesky světel, měkké stíny a další. Aplikace funguje v reálném čase. Převzato z WebGL Experiments [30]

více cílů pro vykreslení (multiple render target), kreslení mnoha objektů se stejnou geometrií jedním vykreslovacím příkazem (instancing) a větší podporu textur (zavedení polí textur – array texture) a jejich formátů.

Nová verze jazyka pro shadery

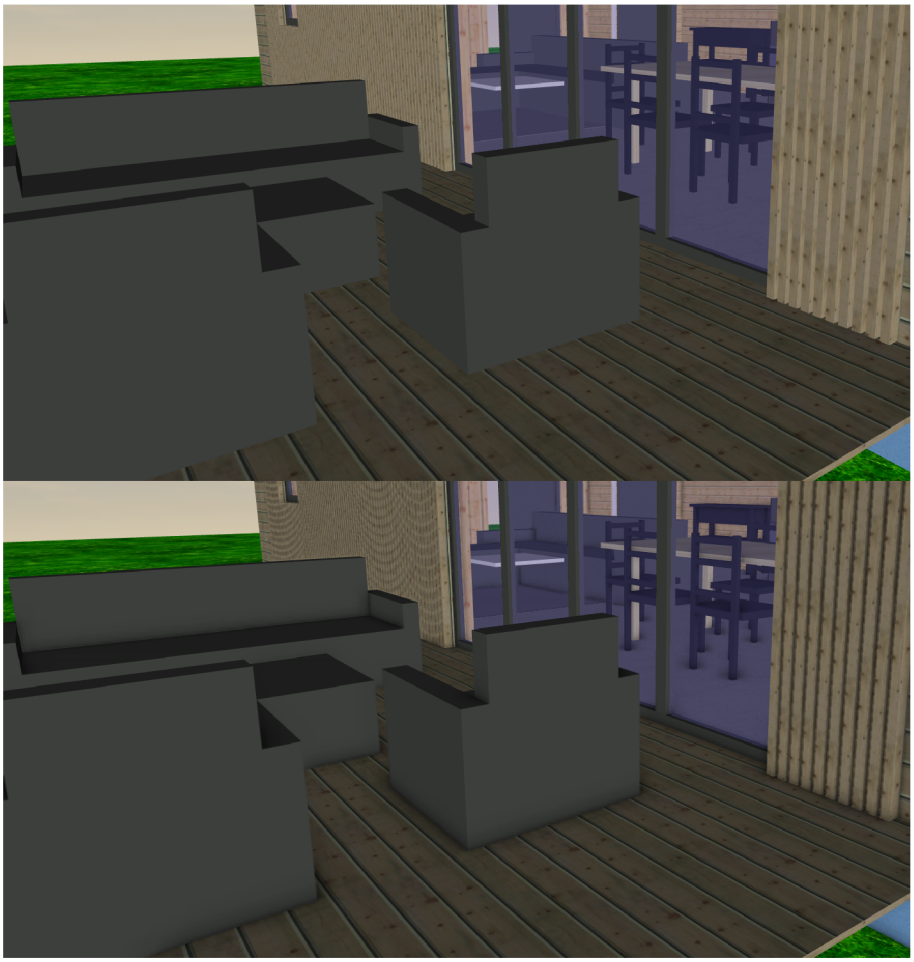
WebGL 2 nyní podporuje **OpenGL ES Shading Language 3.0**, který rozšiřuje předchozí verzi o tyto vlastnosti:

- Práce s celočíselnými typy
- Uniformní bloky
- Vertex Array Objects
- Bindování pozic pro vstupy a výstupy z shaderu
- Dynamické cykly
- Sofistikované vzorkování textur (texture samplers)
- Mnoho původních rozšíření je nyní v základu

Více cílů pro vykreslení (MRT - Multiple Render Target)

Tato funkcionalita se dala získat už ve WebGL při použití rozšíření *WEBGL_draw_buffers*, ale v nové verzi je podporována přímo. Jedná se o výstup hodnot z fragment shaderu do více

bufferů pouze pomocí jednoho kreslícího průchodu a jde o velice užitečný postup například při použití metody Deferred Shading, která umožňuje výrazně snížit náročnost počítání osvětlení scény s více světelnými zdroji, nebo pro počítání Ambient Occlusion, které je znázorněno na obrázku 2.2 a přidává do scény stíny pro větší vnímání hloubky. Tuto metodu lze použít i tak, že dochází k zápisu pomocných hodnot do druhého bufferu a následné čtení z něj (například zapís identifikačního čísla vykresleného objektu).



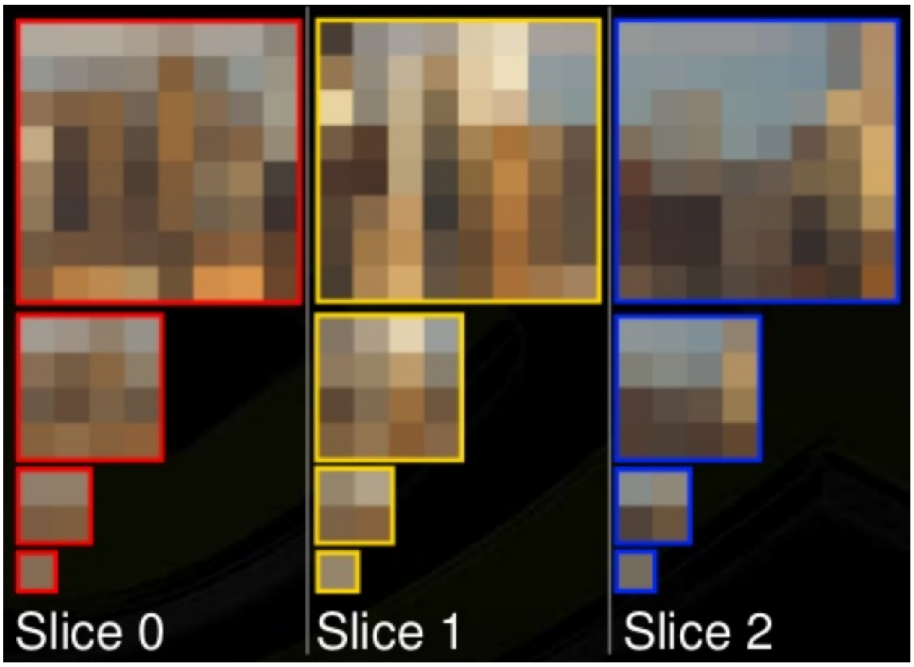
Obrázek 2.2: Příklad scény kreslené s efektem Screen Space Ambient Occlusion (dole) využívající MRT pro urychlení, ve srovnání se scénou bez tohoto stínování (nahoře). Scéna je vykreslena aplikací cadwork WebGL Viewer.

Nové vlastnosti pro textury

Byla přidána podpora 3D textur, polí 1D a 2D textur (*Array Texture*) a odstraněna nutnost používat pouze textury s rozlišením na všech stranách rovných mocnině dvou. 3D textura je textura, kde se k jednomu texelu přistupuje právě pomocí 3 souřadnic, na rozdíl od polí 2D textur, kde se přistupuje pomocí 2 souřadnic a jednoho indexu do pole. 3D textury je možné použít například pro mapování tónů a 2D pole místo tzv. *texture atlases* (vložení více menších textur do jedné veliké).

Použitím array texture je možné snížit nároky potřebné k vykreslení scény, která obsahuje stovky nebo i tisíce různých textur. Uložení více textur do jednoho pole je znázorněno

na obrázku 2.3. Z tohoto obrázku je patrné, že textury musí mít stejnou velikost a musí obsahovat stejný počet úrovní pro mipmaping.



Obrázek 2.3: Uložení textur v Array Texture. Převzato z prezentace nVidia [25]

2.4 WebAssembly

WebAssembly [19] je nový typ kódu, který může běžet na moderních prohlížečích. Je podporován na stolních počítačích v Chrome 57, Edge 15, Firefox 52, Opera 44 a Safari 11 a také na některých mobilních platformách. Jde o nízkoúrovňový jazyk připomínající assembler s kompaktním binárním formátem, který běží téměř rychle jako nativní aplikace. Umožňuje aplikacím psaných v jazycích C/C++/Rust a dalších, aby mohly být přeloženy do tohoto formátu a spuštěny na webu. Jedná se dále o jazyk, který umožňuje propojení s JavaScriptem, tedy je možné využít jak vyššího výkonu WebAssembly, tak flexibility JavaScriptu. Tento jazyk je nyní vyvíjen jako webový standard organizací W3C WebAssembly Working Group [2].

Pro JavaScript je v podporovaných prohlížečích k dispozici globální objekt `WebAssembly`, který nabízí funkce pro načtení kódu a jeho překlad, vytvoření paměti a její asociaci ke kódu. Příklad, který načte *WebAssembly bytecode*, přeloží, vytvoří instanci a importuje/exportuje funkce pro vzájemnou komunikaci JavaScriptu a instance získané z WebAssembly je znázorněn v JavaScript 2.2.

```
var importObject = {
  imports: {
    imported_func: function(arg) {
      console.log(arg);
    }
  }
};

fetch('simple.wasm').then(response =>
  response.arrayBuffer()
).then(bytes =>
  WebAssembly.instantiate(bytes, importObject)
).then(result =>
  result.instance.exports.exported_func()
);
```

JavaScript 2.2: Příklad inicializace WebAssembly kódu [19]

2.5 Emscripten

Emscripten [3] je open-source LLVM překladač pro webové prohlížeče. Dokáže přeložit programy psané v jazycích C, C++ a dalších, které podporují překlad do LLVM bitového kódu, ten je následně přeložen do JavaScriptu. Není možné přeložit libovolný program, protože existuje spousta omezení, která musí být splněna. I přes tato omezení, byly úspěšně přeloženy i velké projekty jako je například CPython, Poppler, Bullet Physic Engine a z komerčních lze zmínit Unreal Engine 4 a Unity.

Vygenerovaný kód je velice rychlý a standardně používá formát **asm.js**, jedná se o podmnožinu vysoce optimalizovatelného JavaScript kódu (omezení funkcionality s ohledem na výkon, *ahead-of-time* optimalizace a další). Existuje však i druhý výstupní formát – **WebAssembly**. Proces generování kódu a jeho etapy jsou na obrázku 2.4. Zdrojový kód je pomocí překladače Clang převeden do LLVM bitcode, který je následně zpracován Emscriptenem do finální podoby.

Obrázek 2.4: Etapy překladu s překladačem Emscripten.

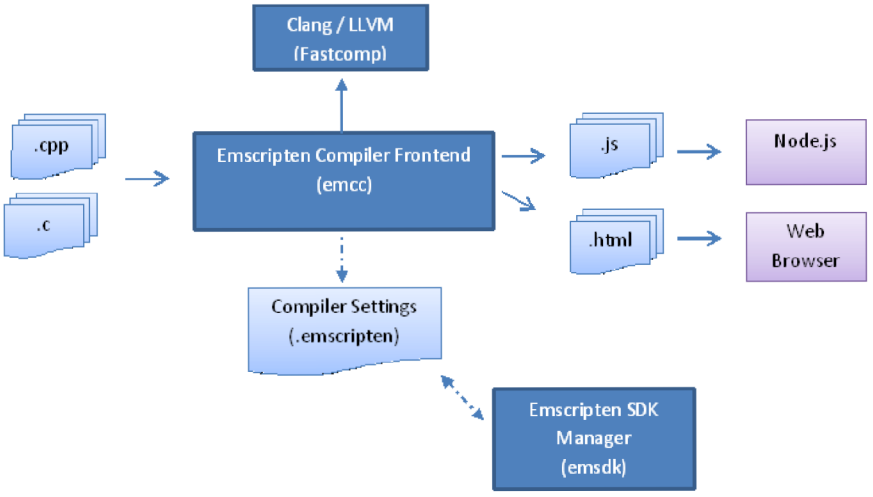
Emscripten Toolchain – sada nástrojů pro překlad

Překlad v Emscripten probíhá prakticky totožně jako s GCC, Clang a jimi podobnými nástroji. Obsahuje aplikace `emcc` a `em++`, které využívají Clang k překladu C/C++ souborů do LLVM bitového kódu a následně Fastcomp, který přeloží bitový kód do JavaScriptu. Tento JavaScriptový kód může být použit přímo v node.js nebo vložen do webové stránky.

Projekt nabízí *Emscripten SDK* (program `emsdk`), který spravuje více SDK a nástrojů a specifikuje, které jsou právě nastaveny jako aktivní. Dále umožňuje stažení, překlad a insta-

laci nejnovějších nástrojů ze serveru Github. Tento program používá Emscripten *Compiler Configuration File* (konfigurační soubor – `.emscripten`), ve kterém je uloženo aktuální prostředí, které bude použito pro překlad nástrojem `emcc`.

Překlad je možný na zařízeních s operačním systémem Linux, Windows i Mac OS X. Průběh překladu je znázorněn na obrázku 2.5.



Obrázek 2.5: Emscripten Toolchain. Převzato ze stránek Emscripten [3].

Projekty, které lze nakonfigurovat pomocí aplikace autoconf, je možné přeložit téměř standardně, stačí využít nástrojů emconfigure a emmake. Ukázka takového překladu je znázorněna ve skriptu 2.3. Příkaz pro překlad negeneruje přímo JavaScriptový kód, ale nejprve LLVM bitový kód a poté až cílový.

```
# Klasický přístup -- konfigurace a překlad do nativního kódu
./configure
make

# Překlad v Emscripten
# Konfigurace
./emconfigure ./configure

# Překlad do bitcode
./emmake make

# Finální fáze -- překlad bitcode do JavaScriptu
./emcc project.bc -o project.js
```

Bash 2.3: Konfigurace a překlad existujících aplikací a knihoven.

Optimalizace překladu

Optimalizace při překladu se dělí na dvě úrovně:

- Překlad zdrojového kódu do LLVM bitového kódu
- Překlad bitového kódu do JavaScriptu

Pro obě fáze je důležité dodržet stejné parametry (-01, -02, -Os, -Oz, ...), aby byl překlad optimální.

Emscripten Ports

Pod pojmem Emscripten Ports se skrývá řada knihoven, které byly úspěšně portovány do Emscripten a lze je bez větších problémů použít v aplikacích. Mezi tyto knihovny lze vzpomenout knihovnu SDL2, kterou lze přidat k překladu parametrem `-s USE_SDL=2`. Další knihovny lze najít na adrese https://github.com/emscripten-ports.

Emscripten API

Veřejné API (**A**pplication **P**rogramming **I**nterface) je k nahlédnutí na oficiálních stránkách Emscripten. Toto API nabízí řadu užitečných funkcí, z nichž budou některé detailněji rozebrány níže. Výčtově je lze shrnout následovně:

emscripten.h Propojení s prostředím prohlížeče.

html5.h Nízkoúrovňové propojení HTML5 s nativním kódem.

preamble.js Přístup k přeloženému kódu z JavaScriptu.

File System API Správa souborového sytému a synchronní práce se soubory.

Fetch API Přístup na internet pomocí XHR a IndexedDB.

Objekt Module Globální objekt v JavaScriptu, který slouží k načtení a překladu nativního kódu, k přístupu k jeho funkcím atd. Je možné vytvořit i objekt, který nebude globální (je tedy možné využít více přeložených modulů na jedné webové stránce).

Embind API Podpora JavaScript kódu v C++ a naopak.

trace.h Sledování využití paměti.

vr.h Přístup k funkcím WebVR v nativním kódu.

Objekt Module

Globální objekt **Module** [6] uchovává atributy, které Emscripten volá v průběhu svého běhu. Standardní chování lze upravit změnou atributů. Přes tento objekt lze přistupovat k exportovaným funkcím přeloženého kódu. Objekt je možné i schovat z globální viditelnosti a uložit do vlastní proměnné (tento princip je důležitý při použití více přeložených modulů na jedné stránce).

Užitečné atributy pro Module

`print` Voláno, pokud je něco odesláno na standardní výstup (stdout).

`printErr` Voláno, pokud je něco odesláno na chybový výstup (stderrr).

`arguments` Argumenty příkazové řádky. Hodnoty jsou v C++ na vstupu funkce `main` v `argv` a `argc`.

preInit Funkce, která je zavolána před globální inicializací, ale po základní inicializaci JavaScriptového prostředí.

preRun Funkce, která je zavolána těsně před spuštěním modulu, ale až po inicializaci všech globálních objektů.

noInitialRun Při nastavení na `true` se nezavolá funkce `main`.

noExitRuntime Pokud je nastaveno na `true`, modul není ukončen (a globální paměť není uvolněna) po skončení funkce `main`. Toto je automaticky nastaveno, pokud je například nastavena hlavní smyčka (*main loop*) příkazem `emscripten_set_main_loop`.

onRuntimeInitialized Funkce, která je zavolána, jakmile je kompletně připraveno prostředí (dokončení asynchronní přípravy – načtení WebAssembly, kompilace, přednačtení souborů atd.). Alternativně je možné počkat na spuštění funkce `main`.

onAbort Callback, který je zavolán při požadavku na abnormální ukončení (například funkcí `abort`). Po této funkci je program vždy ukončen, není tedy možné provést zotavení.

filePackagePrefixURL Jedná se o prefix k URL adrese dat, která mají být přednačtena.

locateFile Pokud je nastavena, pak je tato funkce volána při hledání souborů potřebných ke správné inicializaci modulu. Na vstupu dostane URL adresu a očekává se, že vrátí správnou URL adresu na soubory, jako jsou `.wasm` (WebAssembly), `.mem` (inicializační soubor paměti) a na soubor obsahující zabalená data. Jedná se o zobecnění `filePackagePrefixURL`.

logReadFiles Pokud je nastaveno, pak se zavolá `printErr` při každém čtení souboru.

canvas Zde se nastaví objekt, ve kterém se vytvoří WebGL kontext. Objekt lze definovat i přímo v C++.

preinitializedWebGLContext Při kompilaci s parametrem `-s GL_PREINITIALIZED_CONTEXT=1`, je nutné nastavit tento atribut na objekt s vytvořeným WebGL kontextem, který bude použit v C++ aplikaci.

instantiateWasm Slouží k přepsání defaultního inicializačního procesu k vytvoření WebAssembly instance. Je možné zde změnit způsob stažení, kompilace a vytvoření instance a upravit import / export funkcí.

Module je automaticky vytvořen, když se povolí při překladu generování `.html` souboru. Pokud se překládá do `.js` souboru, je nutné tento objekt vytvořit před načtením přeloženého `.js` souboru. Vytvoření jednoduchého objektu, který zobrazí výpis a chybové hlášky z přeloženého kódu v alert okně, je ukázán v JavaScriptu 2.4. Tento kód musí být vložen před načtením přeloženého kódu.

```
var Module = {
  'print': function(text) { alert('stdout: ' + text) }
  'printErr': function(text) { alert('stderr: ' + text) }
};
```

JavaScript 2.4: Definování chování jednoduchého objektu *Module*

Pro práci s více přeloženými zdrojovými soubory současně, lze při překladu zapnout možnost "Modularize" příkazy `-s MODULARIZE=1 -s EXPORT_NAME="Renderer"`. Při použití těchto parametrů nebude vytvořen globální objekt po načtení JavaScriptového souboru, ale jeho vytvoření a uložení vzniklé instance do proměnné obstará programátor. Ukázka takto vytvořeného objektu je v JavaScript 2.5. Nastavení tohoto modulárního objektu je předáno jako jeho první parametr (na rozdíl od globálního objektu Module).

```
var myModule = null;

window.onload = function() {
  // Vytvoření modulu, jako parametr je mu předán objekt, který nahradí
  // jeho standardní atributy programátorem definovanými
  myModule = Renderer({
    canvas: document.getElementById('canvas'),
    print: function(text) { alert('stdout: ' + text) }
  });
};
```

JavaScript 2.5: Vytvoření lokálního objektu *Module* a definice jeho chování

Hlavičkový soubor emscripten.h

Jedná se o hlavičkový soubor, který definuje naprosté základy při práci s Emscripten nebo věci, které jsou specifické pro JavaScript a webové prohlížeče. Obsahuje například:

- Vložení JavaScriptového kódu do nativního
- Volání JavaScriptu z C++
- Prostředí spuštěné aplikace v prohlížeči
- Asynchronní práci se souborovým systémem a IndexedDB
- WebWorkers, WebSockets
- Logování do konzole

Vložení JavaScriptového kódu do C++

JavaScript lze volat přímo z C++ pomocí bloku kódu označeného jako `EM_ASM`. Existují i varianty, které vrací hodnotu (`EM_ASM_INT` a `EM_ASM_DOUBLE`). Jednoduchým příkladem je otevření alert okna z C++.

```
// Zavolání alert okna se správou "Hello World"
// Řetězec v C++ ukončený nulovou hodnotou musí být převeden
// do vysokoúrovňové JavaScriptové reprezentace
EM_ASM({
  alert("Hello " + UTF8ToString($0));
}, "World");
```

C++ 2.6: JavaScriptový kód v C++

Volání JavaScriptového kódu z C++

Jedná se o druhý základní přístup volání JavaScriptu z C++, který je založený na funkci `emscripten_run_script()`. Tato funkce bere jako parametr řetězec, který bude v prohlížeči zpracován pomocí funkce `eval()`. Jedná se o lehce pomalejší přístup než přímé vložení kódu pomocí `EM_ASM`.

Main loop

Prohlížeče jsou založeny na *event* modelu, tedy každý požadavek má svůj prostor a musí do určitého časového intervalu vrátit řízení (skončit), narozdíl od klasických grafických aplikací, které mají nekonečnou smyčku (*main loop*). Tuto smyčku je možné emulovat pomocí `emscripten_set_main_loop()`, která se postará o volání určité funkce (*callback* funkce) v definovaném intervalu. Pokud volaná funkce má obdržet parametry, musí se použít pro nastavení hlavní smyčky funkce `emscripten_set_main_loop_arg()`. Ve funkcích se na druhém místě specifikuje, s jakým počtem snímků za sekundu (FPS) se bude volat callback funkce. Předá-li se hodnota `0`, použije se vestavěný mechanismus `requestAnimationFrame`, který je preferovaný (prohlížeč synchronizuje kreslení tak, aby bylo co možná nejplynulejší). Třetím parametrem se specifikuje, jestli se po nastavení callback funkce má simulovat "nekonečnost" hlavní smyčky. Pokud je nastavena nenulová hodnota, funkce vyhodí výjimku a program zde skončí. Nejsou smazány z paměti globální proměnné (nevolají se globální destruktory, atexit, atd.), ale v případě, že není simulována nekonečnost, pak je vyčištěn zásobník a hodnoty v něm. Volání hlavní smyčky je možné pozastavit, znovu spustit či ukončit pomocí `emscripten_{pause|resume|cancel}_main_loop()`. Vytvoření takovéto smyčky je uvedeno v C++ 2.7.

```
void callbackFunction() {
  /* ... */
}

int main(int argc, char* argv[]) {
  // 1. Callback funkce
  // 2. FPS nebo requestAnimationFrame
  // 3. Simulace nekonečnosti
  emscripten_set_main_loop(callbackFunction, 0, 0);
  return 0;
}
```

C++ 2.7: Nekonečná smyčka v C++ pro webový prohlížeč

Embind API

Embind [5] slouží k jednoduššímu propojení jazyka C++ a jazyka JavaScript, než bylo uvedeno výše. Z C++ lze exportovat jak funkce, tak třídy, a lze v něm volat i JavaScriptové objekty. Má podporu mnoha dalších konstrukcí z C++ (struct, enum, enum class, shared_ptr, vector, ...). Zapnout podporu pro Embind lze při překladu přepínačem `--bind`, pro export slouží v C++ kódu blok `EMSCRIPTEN_BINDINGS()` a všechny takto exportované funkce lze zavolat z objektu *Module*.

```
#include <emscripten/bind.h>
using namespace emscripten;

float foo(float a, float b, float c) {
    /* ... */
}

EMSCRIPTEN_BINDINGS(my_module) {
    function("foo", &foo);
}
```

C++ 2.8: Export C++ funkce s několika parametry pomocí Embind [5]

Aby bylo možné přenášet velká data mezi JavaScriptem a Emscriptenem přeloženým modulem, existuje v Embind možnost *Memory view*. Tento přístup je ukázán v příkladě C++ 2.9 a JavaScript 2.10. V C++ se alokuje kus paměti, který je zpřístupněn v JavaScriptu pomocí typovaných polí (Typed Arrays) [14]. JavaScript může udělat s tímto kusem paměti cokoli, například nahrát texturu pro WebGL, a následně oznámit modulu, že má tuto paměť využít.

```
#include <emscripten/bind.h>
#include <emscripten/val.h>
using namespace emscripten;

unsigned char *byteBuffer = /* ... */;
size_t bufferLength = /* ... */;
val getBytes() {
    return val(typed_memory_view(bufferLength, byteBuffer));
}

EMSCRIPTEN_BINDINGS(memory_view_example) {
    function("getBytes", &getBytes);
}
```

C++ 2.9: Export části paměti do JavaScriptu pomocí typovaného pole (Typed Array)

```
// Uint8Array
var bytes = Module.getBytes();

for (var i = 0, len = bytes.length; i < len; ++i) {
  bytes[i] = /* ... */;
}
```

JavaScript 2.10: Získání paměti a nahrání dat

HTML5

HTML5 [12] je momentálně poslední standard definující HTML. Jedná se jak o novou verzi jazyka HTML, rozšiřující jej o nové elementy, atributy a chování, tak o nové technologie, které je možné použít a které rozšiřují možnosti moderních webových stránek.

C++ API v souboru `html5.h` definuje nízkoúrovňový přístup k HTML5 z nativního kódu. API nabízí kontrolu nad:

- DOM 3 události – klávesnice, myš, změna velikosti okna, posun obsahu okna (scroll), aktivita okna (focus)
- Orientace zařízení – gyroskop a akcelerometr
- Orientace obrazovky – na výšku / na šířku
- Režim celé obrazovky
- Odchycení myši
- Ovládání vibrování
- Ovládání spotřeby energie
- Dotyky
- Gamepad
- WebGL kontext

WebGL kontext s html5.h

Vytvoření kontextu probíhá následovně: nastaví se požadované atributy, požádá se prohlížeč o vytvoření kontextu a nakonec se nastaví tento kontext jako aktivní. Atributy se nastavují pomocí předdefinované struktury `EmscriptenWebGLContextAttributes`, kterou je vhodné inicializovat funkcí `emscripten_webgl_init_context_attributes()`. Funkce je použita kvůli dopředné kompatibilitě a nastavení všech parametrů na defaultní hodnoty.

Parametry ovlivňující WebGL kontext

`alpha` default: `true`
: Požadavek na alpha kanál pro kontext. Pokud je `true`, pak lze míchat obsah elementu canvas a vrstvy, která se nachází pod ním.

`depth` default: `true`
: Požadavek na vytvoření hloubkového bufferu o alespoň 16 bitech.

`stencil` default: `false`
: Požadavek na stencil buffer o alespoň 8 bitech.

`antialias` default: `true`
: Požadavek na použití antialiasingu. Pokud je `true`, pak se použije prohlížečem nastavený algoritmus a úroveň kvality.

`premultipliedAlpha` default: `true`
Kontext se bude chovat k alpha kanálu jako kdyby byl premultiplied (přednásobený).

`preserveDrawingBuffer` default: `false`
Pokud je `true`, pak je uchován obsah kreslících bufferů mezi voláním hlavní smyčky. Při `false` je vyčištěn barevný, hloubkový a stencil buffer. Nastavení na `false` zvyšuje výkon.

`preferLowPowerToHighPerformance` default: `false`
Při `true` je prohlížeči oznámeno, ať vytvoří kontext méně náročný na spotřebu.

`failIfMajorPerformanceCaveat` default: `false`
Při `true` se nevydaří vytvořit kontext, který nenabízí dobrý hardwarově akcelerovaný výkon.

`majorVersion, minorVersion` default: `majorVersion=1`, `minorVersion=0`
Specifikace verze WebGL kontextu. Pro WebGL 2 je nutné změnit na `majorVersion=2`.

`enableExtensionsByDefault` default: `true`
Při `true` jsou automaticky povolena všechna rozšíření kompatibilní s GLES2, která neovlivňují výkon. Při `false` je nutné povolit tato rozšíření voláním funkce pro každé rozšíření, které má být povoleno – `emscripten_webgl_enable_extension()`.

Kapitola 3

Návrh rendereru a jeho začlenění do webové aplikace

Existující aplikace, pro kterou je tento modul vyvíjen, byla vytvořena a je nadále vyvíjena firmou **cadwork®**. Jedná se o webový prohlížeč 3D modelů, se kterými firma pracuje. Firma tak může prezentovat svým zákazníkům nový nebo upravený model pomocí URL adresy. Velikou výhodou je, že pro zákazníka odpadá nutnost stahování a instalace externího programu. Výstup z této aplikace je zobrazen na obrázku 1.1.

Existující webová aplikace se skládá z několika částí a je psána v čistém JavaScriptu. Tato práce se zaměřuje na vylepšení a zrychlení vykreslovací části použitím nových a modernějších technologií. Momentálně se k vykreslování používá první verze WebGL, která je poměrně omezená, a proto je použita spousta dostupných rozšíření (orientační podpora jednotlivých rozšíření pro různá zařízení je k dohledání na stránce `https://webglstats.com/`).

3.1 Analýza existující aplikace

Webovou aplikaci lze z pohledu programátora rozdělit na dvě samostatné části – engine a GUI. GUI část se stará o komunikaci s uživatelem, o zpracování těchto informací a jejich předání do části engine přes rozhraní Scene API. Scene API obsahuje funkce, které se starají o nastavení světel, stínů, pozadí, pozici modelu a mnoho dalšího.

Engine je zodpovědný za veškerou práci na pozadí, se kterou se uživatel nedostane přímo do kontaktu, například stažení, zpracování a zobrazení modelu, výpočet stínů, osvětlení a další. Po stažení a zpracování jednotlivých částí modelu se tyto části dostanou ke správnému objektu (v aplikaci pojmenovaném jako `RenderGraph`), který slouží k jejich vykreslení do scény. Momentálně existují dva objekty – CanvasRenderGraph (2D) a WebGLRenderGraph (3D). Nově implementovaný vykreslovací modul nahradí na podporovaných zařízeních starší a pomalejší WebGLRenderGraph.

Objekty typu RenderGraph

Objekt typu RenderGraph má na starost uložení objektů do vnitřní reprezentace a jejich následné vykreslení na scénu při určitém nastavení. Rozhraní tohoto typu objektu je uvedeno níže, z něj bude návrh modulu vycházet.

`initialize`
: Funkce, která je volána po vytvoření objektu. Slouží k přípravě shader programů, framebufferů a dalších pomocných objektů.

`release`
: Uvolnění prostředků při ukončení aplikace.

`resize`
: Odchycení události při změně velikosti elementu canvas. Tato funkce může sloužit ke změně velikostí framebufferů, nastavení kamery a dalších závislých parametrů.

`addShape`
: Přidání popisu renderovatelného objektu, který se bude vykreslovat. Objekt je asynchronně stažen a jeho data se naplní ve funkci `notifyMeshCreated`. Tato funkce slouží k přípravě paměti a jiných proměnných a je volána před stahováním dat modelu.

`removeShape`
: Odstranění objektu z vykreslování, objekt musí obsahovat existující unikátní ID.

`notifyMeshCreated`
: Oznámení modulu, že je stažena geometrie objektu. RenderGraph začlení tento objekt do své interní reprezentace a může začít vykreslovat.

`notifySetTextureDataForShapes`
: Přiřazení textury k určitým objektům, které jsou předány pomocí pole. Data textury jsou dekomprimována a nahrána na GPU.

`render`
: Funkce pro vykreslení objektů na scénu.

`renderDepth`
: Funkce, která vrací hloubkové a poziční informace o objektu na daných souřadnicích ve scéně. Využívá zapsaných dat do bufferu během vykreslování scény.

`renderGeometryObjectUID`
: Funkce vracející identifikační číslo objektu na daných souřadnicích scény. Využívá dodatečných dat zapsaných do bufferu během kreslení.

Analýza 3D modelů

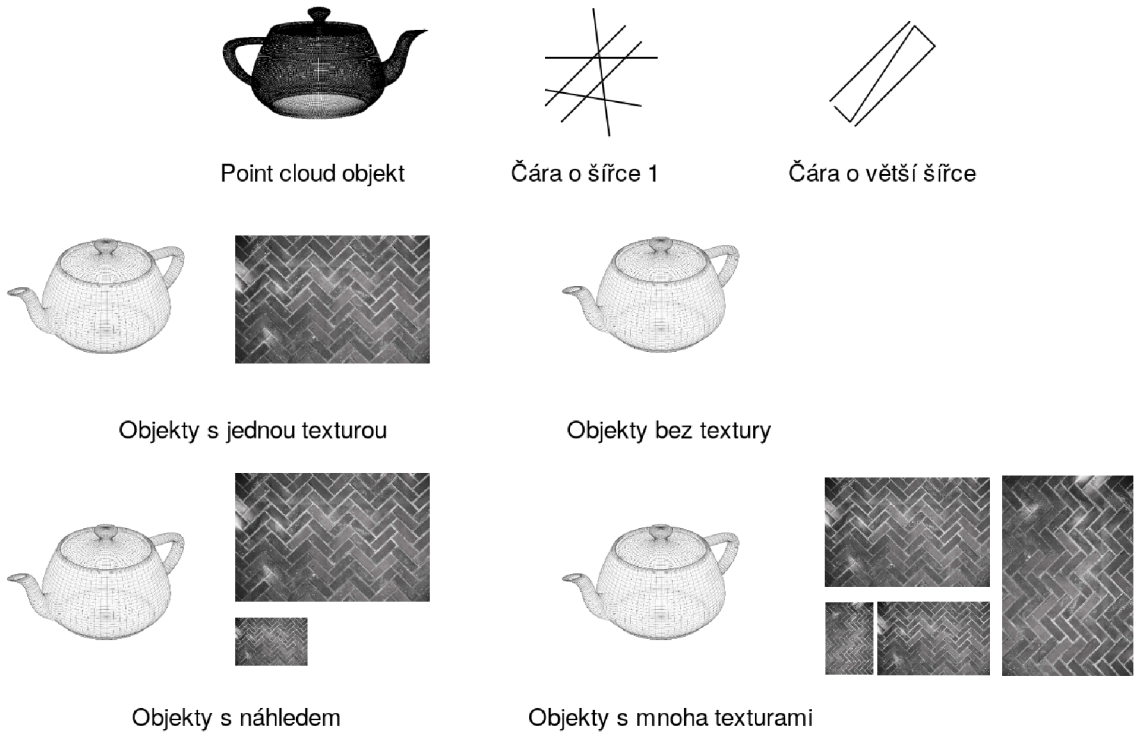
Formát modelu, který je načítán touto aplikací, obsahuje několik dílčích částí: geometrii (shapes), textury, informace o způsobu vykreslení a může obsahovat data pro point clouds. V modelu je uloženo i nastavení grafického rozhraní, aby pro různé typy modelů byla k dispozici různá nabídka pro uživatele. Geometrie objektů je zabalena do co nejméně binárních souborů, aby jejich stahování a zpracování trvalo co možná nejkratší dobu. Textury jsou zabaleny podle svých rozměrů do binárních souborů. Podporované formáty jsou PNG, JPEG, DDS (s podporou crunch komprese [24]), ETC1 a PVR. Dále model obsahuje popisné soubory ve formátu JSON, které určují propojení geometrie, textur, jejich umístění ve scéně, případně další parametry.

Existuje mnoho způsobů, jakým jsou modely pro tuto aplikaci uloženy. Nejjednodušším z nich je uložení maximálně jedné textury na geometrii. Další způsoby zahrnují texturu

s náhledem pro rychlé stažení (přibližné rozlišení je 32 pixelů na stranu) a texturu s maximálním rozlišení, a nebo existuje pro jednu geometrii celá škála textur, které se střídají podle své vzdálenosti od kamery (level of detail) – využito převážně u vykreslování terénu, aby bylo možné ušetřit paměť grafické karty. Momentálně je textura s určitým rozlišením právě jedna, ale počítá se s rozšířením minimálně o normálové textury.

Čáry se rozlišují podle své šířky, pokud je šířka čáry 1, pak je uložen vždy počáteční a koncový bod, pokud však má být čára širší, pak je z dat vygenerována sada trojúhelníků, které se pomocí své normály posouvají o danou šířku. Uvedené informace jsou znázorněny na obrázku 3.1.

Z těchto informací je jasné, že bude potřeba vytvořit několik možných způsobů interní reprezentace a jejich vykreslení. Pro každý způsob mohou být použity jiné metody optimalizace.



Obrázek 3.1: Části modelu, které může aplikace obsahovat – body (point cloud), čáry, emulace čar o větších šířkách generovaných jako objekt z trojúhelníků, objekty bez textur (obsahují pouze barevné informace pro výpočet osvětlení), objekty s jedinou texturou, s náhledem a texturou v plné velikosti a nebo objekty mající LOD.

3.2 Návrh nového vykreslovacího modulu

Nový renderer využívá technologie WebGL 2 a WebAssembly, které byly detailněji popsány v kapitole 2. WebGL 2 bylo zvoleno, protože nabízí více možností a i vyšší rychlost (při implementaci stejné aplikace ve WebGL a WebGL 2 je dosaženo zvýšení výkonu přibližně o 3 až 7% [7]). WebAssembly je moderní formát, který dosahuje menších velikostí a vyšších rychlostí než běžný JavaScript. Při srovnání s asm.js dosahuje stabilnějšího výkonu, pro-

tože asm.js obsahuje JavaScriptový kód, který jde velice kvalitně optimalizovat, ale každý prohlížeč může použít jiný způsob optimalizace a tedy na některých běží daný kód rychleji (pomaleji) než jinde.

Návrh nového rendereru je podřízený návrhu existující webové aplikace popsané výše, zároveň je celý modul psaný v odlišném jazyce (C++) a vyvíjen je bez neustálého používání prohlížeče s pomocí nativní (testovací) aplikace. Po odladění je tento modul přeložen do WebAssembly a otestován v původní aplikaci. V JavaScriptovém kódu je vytvořen objekt WasmRenderGraph, který slouží jako wrapper nad kódem z WebAssembly. Stará se o správné načtení modulu, jeho spuštění a o kódování dat, která budou přenášena mezi jazykem C++ a JavaScriptem (serializace objektu do formátu JSON, alokace paměti uvnitř C++ modulu, nakopírování obsahu z JavaScriptu, ...).

Pro předávání větších kusů paměti mezi C++ a JavaScriptem byl navržen systém, který umožňuje efektivně sdílet tuto paměť. Paměť je identifikována pomocí tzv. handleru (celočíselná hodnota), který je získán při její alokaci. Po alokování je možné získat v JavaScriptu typované pole a v C++ shared_ptr na data, a pracovat s nimi. Paměť počítá reference a je uvolněna automaticky. Pokud byla paměť vytvořena v JavaScriptu, je nutné zavolat funkci pro uvolnění (manuální snížení počtu referencí).

Rozhraní modulu

Aby bylo možné komunikovat mezi modulem a aplikací, je nutné definovat rozhraní které umožní přenos informací mezi jazykem C++ a JavaScriptem. Toto rozhraní bude zahrnovat následující možnosti:

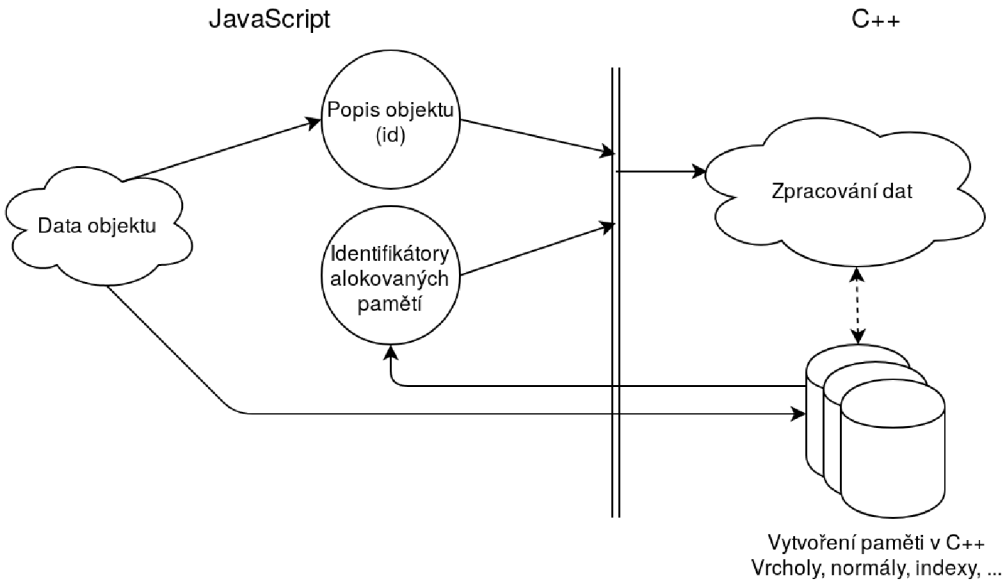
- Vložení informací (meta dat) o objektu k vykreslení
- Vyplnění grafického objektu daty
- Změnění některých vlastností objektu nebo jeho odstranění
- Nastavení textury k objektům
- Nastavení scény (kamera, světla, velikost vykreslovací oblasti)
- Informace, že objekty scény byly staženy a odeslány do modulu
- Správa paměti (alokace, získání, uvolnění)
- Získání informací o vykreslených objektech ve scéně
- Požadavek na aktualizaci stavu a kreslení

Vložení informací o objektu k vykreslení

Předání informací o renderovatelném objektu, který se bude vykreslovat. Odpovídá funkci `addShape` v RenderGraph. Vstupem bude řetězec ve formátu JSON (pro jednoduchý a dynamický přenos informací), který bude popisovat tento objekt (id, počet vrcholů, počet normál, ...). Modul připraví paměť (přesněji si poznačí, kolik paměti bude potřebovat až budou přicházet data), zařadí objekt do správné interní reprezentace a čeká na validní data.

Vyplnění grafického objektu daty

Funkce oznámí modulu, že je stažena geometrie pro daný objekt. Tato geometrie bude zpracována do interní reprezentace (propojuje `notifyMeshCreated` funkci z RenderGraph objektu). V JavaScriptu se vytvoří paměťové bloky obsahující data, tyto bloky jsou pomocí jejich identifikátorů předány do C++. Zároveň s daty vrcholů je předána i informace o objektu, která se kóduje do formátu JSON. Toto je znázorněno na obrázku 3.2.



Obrázek 3.2: Znázornění komunikace mezi modulem a webovou aplikací pro předání většího objemu dat.

Změnění některých vlastností objektu nebo jeho odstranění

Pro odstranění grafického objektu je v objektu RenderGraph vytvořena funkce removeShape. Tato funkce předá do modulu identifikaci objektu, který má být odstraněn. Pro změnu informací o objektu momentálně neexistuje žádná funkce v JavaScriptu, protože to nebylo potřeba, v tomto případě tato funkce přibude (jedná se například o zneviditelnění, změnu pozice v prostoru nebo i změnu barevných složek). Přidání funkce pro změnu umožní efektivnější práci s pamětí (není nutné objekt odstranit a znova vytvořit).

Nastavení textury k objektům

Proces předání textury probíhá podobně jako proces předání dat do modulu. Paměť je alokována na straně C++, v JavaScriptu je naplněna validními daty a následně je předán identifikátor této paměti. Spolu s identifikátorem paměti na data textury jsou předány i identifikátory objektů, ke kterým má být tato textura přiřazena. Jelikož textury mohou dosahovat i několika MB dat, je nutné omezit počet přenesených dat v rámci jednoho update cyklu, aby byla práce s modulem plynulá. Toto omezení bude detailněji probráno v implementaci wrapperu nad C++ modulem.

Nastavení scény

Parametry scény před samotným vykreslením zahrnují nastavení vykreslovací oblasti (velikost okna / canvasu), nastavení kamery a parametry světel (typ, pozice, barva, intenzita). Scéna může obsahovat i několik různých světel, kde momentálně aplikace obsahuje jedno osvětlení simulující slunce a druhé typu headlight (světlo z kamery). Informace se opět kódují do formátu JSON.

Informace, že objekty scény byly staženy a odeslány do modulu

Po spuštění je modul v režimu stahování a přípravy dat, je tedy možné použít lehce jiný přístup pro zpracování dat, kdy není nutné provádět změny okamžitě, ale například si je odložit, dokud se těchto změn nenahromadí více. Tento přístup umožní zvýšit rychlost přípravy datových struktur (buffer pro vrcholy je alokován v ideálním případě jen jednou, jinak by mohlo docházet k časté realokaci paměti). Další výhodou tohoto přístupu je informovat uživatele o stavu načítání, aby věděl, kdy už může s modelem pracovat.

Správa paměti

Modul umožňuje spravovat paměť, a to tak, že je k dispozici jak pro JavaScript, tak i pro C++ (JavaScript vidí do C++, nikoli však naopak). V JavaScriptu se pro paměť používají typované pole (Float32Array, Uint8Array, ...), které umožňují specifikovat přesný typ jednotlivých položek. Alokace paměti rezervuje místo a vrátí identifikátor, který poté slouží k získání přímého odkazu / objektu na paměť. Tyto identifikátory slouží pro rychlý přenos dat mezi modulem a aplikací. Každá alokovaná paměť obsahuje počítání referencí, které ji při dosažení nuly automaticky uvolní.

Získání informací o vykreslených objektech ve scéně

V aktuální aplikaci se pro získávání informací o objektech ve scéně využívá jejich vykreslení, kdy se do vykreslovacího bufferu neukládá výsledná barva, ale určitá informace – identifikace prvku, jeho souřadnice v prostoru a další. S využitím WebGL 2 je možné použít MRT (multi render target) a vykreslit tyto informace spolu s barevnou částí při využití více bufferů zároveň. Hodnoty jsou následně čteny aplikací. Tento způsob naráží na pomalé čtení hodnot z grafického bufferu. Jedním z možných řešení je využít metody raycastingu na CPU, která vrhá paprsky do předzpracované scény (scéna může být dělena například na AABB obalová tělesa nebo KD stromy pro optimalizaci procházení paprskem).

Požadavek na aktualizaci stavu a kreslení

Modul je řízen z aplikace, proto musí být informován, kdy může provést aktualizaci vnitřního stavu a kdy může vykreslit objekty, které obsahuje. Pro plynulost programu je aktualizační cyklus omezen časovým intervalem, kdy při překročení tohoto intervalu dojde k přerušení provádění cyklu a uložení zbývajících událostí do fronty, která bude provedena během příští aktualizace. Tato omezení plynou z toho, že v modulu je silně omezena možnost využití více vláken (podpora pouze HTML5 Workers, běžná vlákna nelze použít v aktuální verzi).

Aplikace pro vývoj a testování

S vývojem modulu je vyvinuta i desktopová aplikace, která emuluje potřebnou část webové aplikace. Modul je nejdříve programován klasicky jako nativní aplikace (s ohledem na omezení, které je nutné dodržet) a až poté v určitých fázích vývoje je portován do webového formátu překladačem Emscripten a otestován v prohlížeči. Vykreslování používá OpenGL ES 3.0, na kterém je WebGL 2 založeno.

O otevření okna a vytvoření správného kontextu se stará knihovna **SDL2** (Simple DirectMedia Layer) spolu s knihovnou **GLEW** (The OpenGL Extension Wrangler Library). O matematické operace nad maticemi se stará knihovna **GLM** (OpenGL Mathematics). Pro jednoduchou práci s formátem JSON je použita knihovna **JSON for Modern C++** [27]. Dekomprese PNG a JPEG obrázků do textur probíhá pomocí knihoven **jpeg_decoder** [23] a **lodepng** [29].

Data, která budou předávána do jednotlivých funkcí modulu jsou externě uložena ve formátu JSON, a odpovídají datům generovaným z webové aplikace. Ukázka jednoho shape objektu (renderovatelného objektu) je v JSON 3.1, ukázka přiřazení textury v JSON 3.2.

```
1 {
2     "description": "{\"UID\":2,\"type\":\"SHAPE\",\"appearance\":{\"lineProperties\":...}",
3     "vertices": [2.6067423820495605, 4,1.218102216720581, -1.39325749874115, 4,...],
4     "indices": [0,1,2,2,1,3,4,5,6,6,5,7,8,9,10,10,9,11,12,13,...],
5     "normals": [0,1,0,0,1,0,0,1,0,0,1,0,0,-1, 0,0,-1, 0,0,-1, 0,...],
6     "colors": null
7 }
```

JSON 3.1: Ukázka exportovaných dat jednoho shapu z modelu

```
1 {
2     "description": "{\"UIDs\":[2],\"width\":32,\"height\":32,\"type\":\"imageTexture\"}",
3     "data": [255,216,255,224,0,16,74,70,73,70,0,1,1,1,0,72,0,72,...]
4 }
```

JSON 3.2: Ukázka exportované textury pro model

3.3 Návrh obecné části modulu

Aby bylo možné zobrazit data pomocí grafické karty na obrazovku, je nutné připravit a udělat mnoho věcí dopředu. Tato sekce rozebere základní myšlenky návrhu modulu, toku dat a zpracování těchto dat. V následující části bude rozebráno vykreslování a jeho optimalizace.

Bohužel není možné momentálně používat více vláken ve WebAssembly [8], jedná se zatím pouze o experimentální možnost, která navíc na začátku roku 2018 obsahuje bezpečnostní díry. Proto byl i návrh rendereru upraven tomuto faktu.

Příprava a přenos dat během načítání modulu

Komunikace mezi JavaScriptovou a C++ částí probíhá převážně pomocí textového formátu JSON nebo pomocí bufferů a předání jejich identifikátorů. Většina komunikace probíhá jedním směrem, a to z aplikace do modulu. Buffery se používají při přenosu většího množství

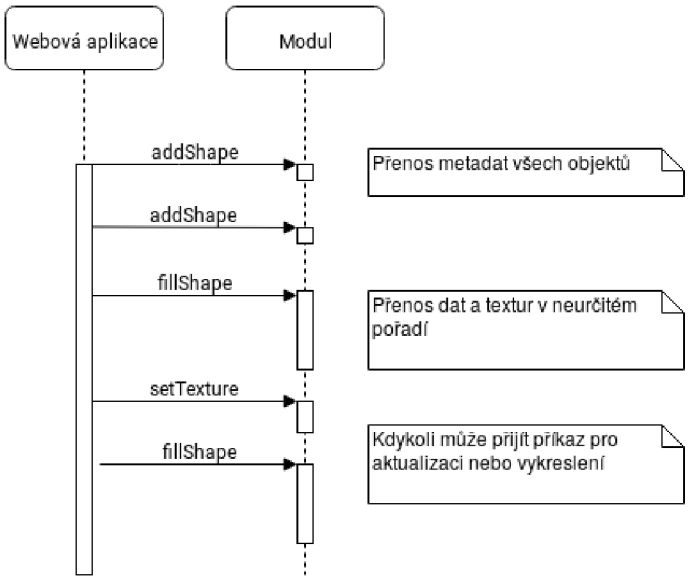
dat, jako jsou textury nebo vrcholy trojúhelníků. Následně se tyto objekty dekódují a uloží do struktur nebo objektů v C++ pro pozdější zpracování.

První data, která jsou přenášena mezi aplikací a modulem, jsou pouze "meta data" popisující objekty scény. Mezi těmito daty lze najít velikosti bufferů, počet vrcholů trojúhelníku, počet textur potřebných pro daný objekt, jejich velikost, ...

Následují data, která doplňují registrované objekty o jejich geometrii a textury. Data jsou stahována a zpracovávána v aplikaci paralelně, ale jejich přenos je serializován do hlavního vlákna. Zde nastává zpomalení, protože je nutné překopírovat relativně velké bloky paměti z JavaScriptu do WebAssembly (přeloženého C++ modulu).

Zároveň s přípravou dat probíhá vykreslování neúplné scény (`onRender`) a pravidelná aktualizace stavu modulu (`onUpdate`). Před kreslením se nastaví údaje o kameře a o světlech.

Celý tento proces je emulován v nativní aplikaci. Data jsou uložena v textových souborech, které obsahují stejné informace, jak kdyby byly přenášeny z webové aplikace. Jelikož je těchto souborů i několik desítek tisíc, pro rychlejší zpracování byla implementována podpora pro kompresi LZ4, model je tedy uložen jen v jednom souboru a poté dekomprimován.



Obrázek 3.3: Znázornění průběhu načítání modelu. Jako první příjdou do modulu meta data, po kterých následuje geometrie objektů a jejich textury. Kdykoli během těchto příkazů může přijít žádost o vykreslení nebo o aktualizaci vnitřního stavu.

Zapouzdření OpenGL / WebGL volání

Jelikož je OpenGL ES 3.0 pouze podobné s WebGL 2, je vhodné udělat nad těmito grafickými rozhraními vrstvu, která bude obsahovat pouze validní příkazy pro oba formáty, případně bude jeden z nich emulován pomocí jiných dostupných funkcí. Další výhodou této vrstvy je hlídání stavu vykreslovacího stroje, aby se znova nenastavoval stále stejný stav, případně pokud byla zjištěna nějaká informace, vrátit její hodnotu bez opětovného volání příkazů grafické karty (např. zjištění podporovaných rozšíření – "extensions").

Při implementaci budou zapouzdřeny ještě další objekty, se kterými se pravidelně pracuje. Jedná se o Vertex Array Object, Vertex Buffer Object, Element Buffer Object, Tex-

ture2D, Array Texture, Shader (Vertex / Fragment) a Shader Program. Všechny tyto objekty mají předlohu v jejich OpenGL reprezentaci (metody objektů se snaží zachovávat maximální podobnost) a nabízí přístup i k přímému OpenGL identifikátoru pro specifické práce. V ostatních částech modulu nedochází k přímému volání OpenGL API, ale používají se pouze tyto funkce a objekty.

Manažeři

Modul je složen z několika dílčích částí, kde každou jednotlivou část spravuje určitý "manažer". Jednou z oblastí, kterou je nutné spravovat, je sdílená paměť mezi jazykem C++ a JavaScriptem. Spolu s pamětí je nutné udržovat informace o vykreslovacích objektech a o jejich texturách. Je nutné umožnit vyhledání objektů i textur pomocí jejich identifikátoru a umožnit přidání, odebrání a změnu jejich stavu. Události jsou v modulu řízeny pomocí manažera událostí, který slouží k jejich sběru a následnému šíření těchto událostí všem zaregistrovaným posluchačům. Pro déle trvající procesy je navržen další manažer, do kterého jsou uloženy procesy, které jsou následně vykonávány pouze do maximálního časového intervalu, po uplynutí tohoto intervalu dojde k přerušení jejich provádění a procesy jsou přesunuty do dalšího cyklu (důležité je zachování pořadí). Pro ladění shader programů je navržen manažer, který kontroluje změny jejich zdrojových souborů a následně provede jejich přeložení a nahrazení za běhu aplikace. Změny se tak projeví bez nutnosti restartování aplikace.

3.4 Návrh vykreslovací části modulu

3D modely se mohou skládat z tolika drobných částí, že samotné volání funkce pro vykreslení, nastavení parametrů a obecně komunikace s ovladačem grafické karty, mohou výrazně snížit výkon při kreslení scény. Během kreslení ve WebGL dochází navíc k dodatečným bezpečnostním kontrolám, které způsobují další pokles výkonu oproti OpenGL [22]. Základní technika, která bude použita pro snížení počtu volání se jmenuje batchování a je popsána níže. Výměna textury je další častý případ, kdy dochází k výraznému snížení výkonu grafické karty, od verze WebGL 2 je možné použít pole textur – Array Texture [4], které tento problém snižuje. Dalším pravidlem bude udržovat aktuální stav grafické pipeline v paměti a znova nenastavovat již nastavené věci, jako je například použitý shader program, nabindovaný buffer atd.

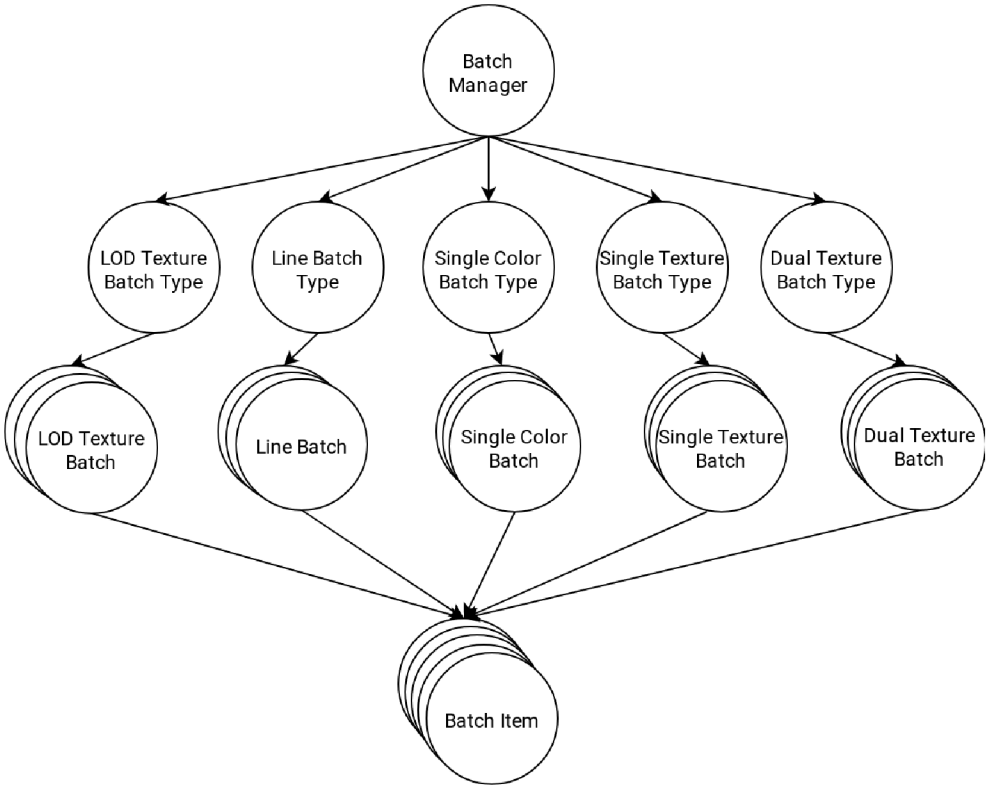
Batch rendering

Batchování, nebo-li batch rendering, je metoda, při níž se kompatibilní grafické objekty (geometrie a nastavení vykreslování) sdružují do skupin a následně se vykreslí celá skupina jako jeden celek (nastaví se jednou atributy pro vykreslení, zavolá se jen jeden vykreslovací příkaz – *draw call*). Této skupině objektů se říká batch (dávka). Batch rendering rozšiřuje možnosti metody *Instancing*, která umožňuje jedním vykreslovacím příkazem vykreslit objekty pouze se stejnou geometrií.

Jedna dávka obsahuje určitou maximální velikost paměti a další jiná omezení (například počet textur na dávku). Při překročení maximální velikosti paměti je nutné vytvořit nový batch a nebo paměť rozšířit (například využitím stejného principu jako std::vector ze standardní knihovny C++). Stejná pravidla platí i pro indexy v element buffer objektu, zde se však musí počítat s tím, že budou jednotlivé elementy "posunuty" o určitý offset. Při

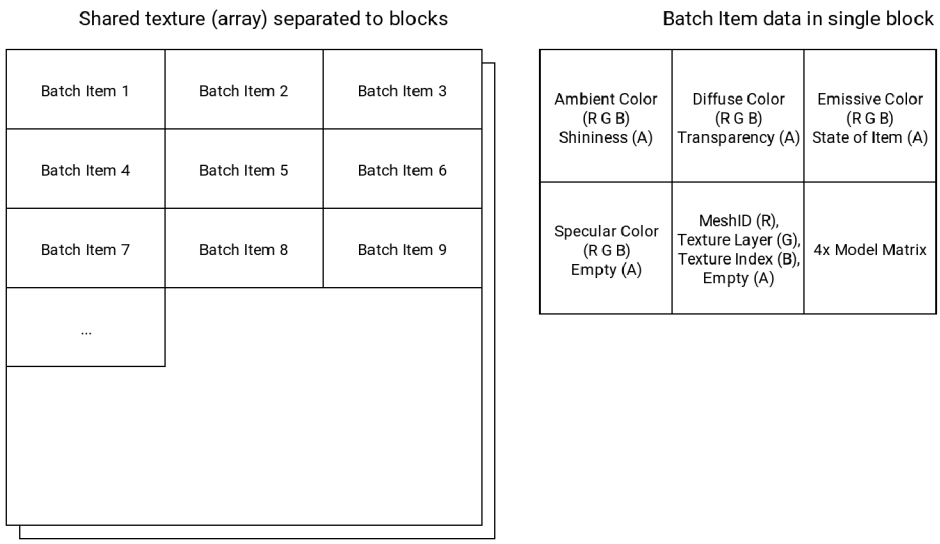
překročení jakéhokoli limitu (například při využití všech texturovacích jednotek) je nutné vytvořit nový batch.

V této práci jsou batche (dávky) specializované pro určitý typ renderovatelného objektu, nad nimiž vždy existuje BatchType objekt, který slouží ke správnému nastavení atributů a uniformů pro shader program. Nad BatchType objekty je správce, který třídí vykreslitelné objekty scény podle jejich typu. BatchType objekt je složen z dávek – Batch objektů, které jsou vykreslitelné jedním příkazem. Každá dávka je složena z několika BatchItem objektů, které jsou zde pro aktualizaci dat při přidávání a odebírání, dále propojují data s texturou. Velikosti jednotlivých Batch objektů jsou odvozeny z meta dat, která dostane modul jako první při spuštění. Hierarchie objektů je znázorněna na obrázku 3.4.



Obrázek 3.4: Hierarchie batchování použitá v modulu.

Data, která jsou specifická pro jednotlivé prvky z dávky, je nutné uložit pro každý prvek zvlášť do speciální paměti. Je možné využít Uniform Buffer Object, který slouží k nastavení většího bloku uniformních proměnných. Z důvodu omezené velikosti je v tomto případě použito pole textur, které je popsáno níže. Hodnoty z něj jsou načteny pro každý objekt pomocí vertex shaderu. Sdílená textura je dělena na bloky, které jsou přiřazeny právě jednomu Batch Item objektu. Vertex Buffer pro jednotlivé objekty obsahuje kromě základních informací (pozice vrcholu) i indexy do sdílené textury a ve vertex shaderu provede načtení požadovaných složek (barva, průhlednost, stav, modelová matice, ...). Základní koncept dělení paměti ve sdílené textuře je na obrázku 3.5.



Obrázek 3.5: Uložení dat pro jednotlivé objekty ve sdílené textuře. V levo je pohled na celkovou texturu a její dělení na sektory. V pravo je znázorněn jeden sektor, který je složený z mnoha dalších údajů.

Array Texture

Před tím, než bylo možné využít Array Texture, používalo se zabalení více textur do jedné velké textury (texture atlas), bohužel se v této starší metodě projevuje několik problémů (prolínání barev, generování mipmap). Při využití Array Texture se zvýší počet použitelných textur v rámci batche (přesněji v rámci jedné texturovací jednotky) a tím i sníží počet vykreslovacích příkazů, zároveň se odstraní neduhy dříve používané metody. Array Texture je textura, která na každé úrovni mipmapy obsahuje pole obrázků o stejných rozměrech. Jednotlivé obrázky na každé úrovni mipmapy se nazývají *layer* (vrstva).

Tento princip umožňuje zvýšit potenciál batchování, protože může být použito mnohem více textur v jedné dávce. Při použití více texturovacích jednotek se tento potenciál ještě více navyšuje.

Kapitola 4

Implementace vykreslovacího modulu

V této kapitole budou rozebrány implementační detaily návrhu probraném v předchozí kapitole. Nejprve bude rozebrána implementace wrapperu nad modulem, která poslouží pro transparentní propojení modulu se stávající aplikací, dále se bude práce zabývat implementací obecných úloh, které bylo nutné implementovat před samotným vykreslováním. Na závěr této kapitoly bude uvedena implementační část vykreslovací části rendereru.

4.1 Wrapper – WasmRenderGraph

Modul při překladu vytvoří konstruktor, který lze zavolat z JavaScriptu (jako parametr bere objekt s nastavením položek, které jsou popsány v sekci 2.5). Po provedení daného příkazu je modul asynchronně zpracován a po úspěšném nahrání kódu je vykonána vstupní funkce `main`. Tato funkce provede inicializaci (příprava paměti, manažerů, ...) a následně zavolá callback funkci do JavaScriptu, která informuje tento wrapper, že lze s modulem pracovat (posílat data k vykreslení). Než-li je modul k dispozici, je nutné ukládat do mezipaměti data, která v tomto intervalu přišla do wrapperu. Data jsou ukládána ve frontě (pro zachování pořadí) a provedena po inicializaci modulu.

Ukázka detekce podpory formátu WebAssembly a následné vytvoření modulu je v C++ 4.1. V budoucí verzi, pokud prohlížeč nebude podporovat WebAssembly, bude použit modul přeložený do asm.js.

```
if (!('WebAssembly' in window)) {
    alert("WebAssembly is not supported in this browser!");
    return;
}

this._renderer = CadworkRenderer({ /* ... */});
```

JavaScript 4.1: Detekce podpory WebAssembly a vytvoření instance z přeloženého modulu.

Vložení a odebrání geometrie (meshe)

Ke správě objektů, které budou vykreslovány, slouží celkem 4 funkce, první z nich předává informace o tomto objektu, aby se pro něj připravila paměť. Zde se provede konverze z JavaScript objektu do řetězce kódovaného ve formátu JSON. Pro převod jsou vybrány jen důležité parametry potřebné k identifikaci, počítání paměti (počet vertexů, indexů, barev, textur, ...) a k zařazení objektu do správné vykreslovací dávky.

Až po vložení meta dat je možné nahrát data. Tento systém je zaveden z důvodu, že se předpokládá velké množství dat hned při spuštění aplikace. Tyto objekty musí být vysoce optimalizovány, ale během běhu aplikace se mění jen jejich malá část (výjimkou jsou textury pro level of detail, ale ty mají vlastní batch). Způsoby optimalizace jsou uvedeny v sekci 4.4 popisující jednotlivé batche.

Pro vyplnění dat je nutné alokovat paměť, která je přístupná jak pro modul, tak i pro aplikaci. API modulu nabízí funkce pro alokaci kusu paměti, získání objektu nebo ukazatele na ni a uvolnění (uvolnění paměti je nutné pouze v JavaScriptu, protože zde nedochází k automatickému snížení počtu referencí). Po získání bloku paměti nahraje JavaScript data a předá identifikátor na tento blok paměti spolu s identifikací objektu. Po předání dat je nutné uvolnit blok paměti z JavaScriptu – dojde pouze ke snížení počtu referencí, modul má paměť k dispozici dokud ji bude potřebovat.

V modulu se provede vyhledání meta dat o objektu. Pokud byl nalezen, přepošle se do správného batche a následně proběhne nahrání dat na grafickou kartu (buffer na grafické kartě o velikosti získané ze všech meta dat již existuje, takže dojde k přidání dat na konec tohoto bufferu a ke změně informací, kolik dat se bude kreslit). Batch je možné v tento moment vykreslit, i když je neúplný – slouží to k zobrazení průběhu načítání uživateli, aby byl informovaný, že aplikace běží v pořádku.

Pro změnu některých vlastností objektu je vyhrazena vlastní funkce, která může být použita například ke změně modelové matice, barev a stavu (zvýraznění, viditelnost). Funkce v původním návrhu aplikace chybí, tyto změny se prováděly odebráním a následným přidáním stejného objektu s různými parametry, docházelo tedy ke zbytečné režii. O těchto změnách je informován modul pomocí řetězce ve formátu JSON, který obsahuje identifikátor objektu a změněný parametr. Tímto způsoben není možné měnit geometrii ani textury.

Na konci běhu aplikace nebo při požadavku na odstranění objektu je modulu předána informace obsahující identifikátor objektu a modul zajistí jeho následné uvolnění. Uvolnění probíhá se zpožděním, takže i při tisíci požadavcích na odebrání objektu dojde jen k poznačení informací a až při následném aktualizačním cyklu (update), dojde k uvolňování paměti. Detailněji je tento princip probrán níže.

Nastavení textury

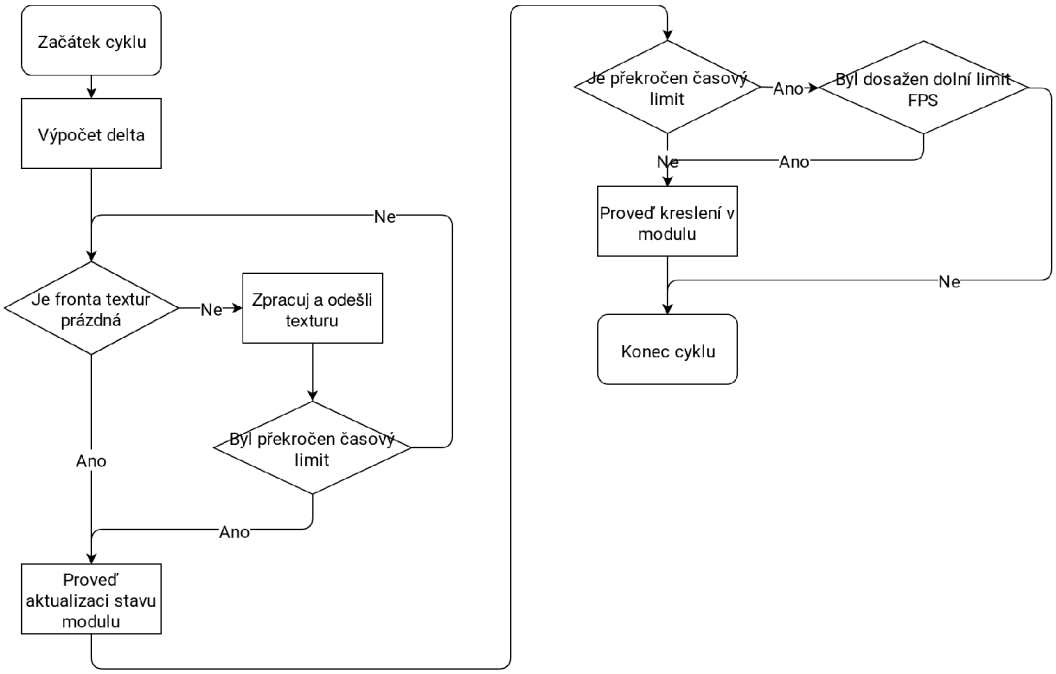
Textura musí být specifikována v meta datech (její rozměry, počet možných textur pro objekt, typ komprese), podle kterých je objekt tříděn a vložen do správných struktur – dávek. Následuje podobný postup jako při vyplnění dat objektu, nejprve je alokována společná paměť, poté jsou do ní nahrány komprimovaná data (dekomprimace probíhá na straně modulu, pokud jsou použity komprimační formáty podporované grafickou kartou, pak jsou data nahrána bez dekomprimace). V průběhu ladění bylo zjištěno, že textury obsahují velké množství dat, a tak kopie paměti trvala občas příliš dlouho. Tento problém byl vyřešen tak, že se na straně wrapperu požadavek na nastavení textury pozdrží až do aktualizačního cyklu, kde proběhne nahrání maximálně tolika textur, jejichž společný čas nepřekročí ur-

čitý limit, zbytek textur se odloží do příštího cyklu. Textury jsou ukládány do fronty, aby nedošlo k přeskládání pořadí požadavků.

Aktualizace stavu a jeho vykreslení

Nekonečná smyčka ve webových aplikacích je emulována funkcí `requestAnimationFrame`. Při tomto požadavku prohlížeč zavolá specifikovanou callback funkci v určitém intervalu (prohlížeč používá frekvenci monitoru, aby zajistil plynulost – běžně se jedná o 60Hz). Tento callback slouží k aktualizaci a vykreslení dat v rámci jednoho "cyklu". Každý cyklus je omezen určitým časovým intervalem, aby se aplikace nezačala trhat. Pokud dojde k výraznému výkonnostnímu výpadku, mohou se některé cykly přeskočit – převážně se přeskakuje vykreslovací část, tak aby došlo pouze ke snížení počtu snímků za sekundu (s minimálním limitem, jinak by mohla aplikace úplně přestat vykreslovat).

Během jednoho cyklu tedy dojde k zahájení počítání uplynulé doby od začátku cyklu, dále je spočítán delta čas mezi tímto a minulým cyklem. Pomocí času delta je možné řídit animaci a mnoho dalšího. Na začátku cyklu dojde k odesílání textur do modulu (pokud existují uložené požadavky), následně k zavolání update funkce modulu a po této funkci může a nemusí nastat vykreslování (podle uplynulé doby). Pro měření času se osvědčila funkce `performance.now`, která funguje s přesností 5 mikrosekund. Na obrázku 4.1 je nastíněn tento princip.



Obrázek 4.1: Princip jednoho cyklu aktualizace a vykreslení dat.

Ostatní funkce

Mezi funkce, které zde nebyly zmíněny, patří událost na změnu velikosti okna, která odešle nové rozměry do modulu, a získání informací o objektech ve scéně (hloubka, pozice, ID).

4.2 Implementace obecné části modulu

Modul obsahuje jeden hlavní objekt, jedná se o singleton `Application`, který udržuje kontext a obsahuje všechny další důležité objekty, které je vhodné mít jednoduše přístupné (například manažery). Podle typu modulu se vytvoří správná instance, protože modul může běžet jako desktopová C++ aplikace, existuje objekt DesktopApp, pro webový modul je implementován EmApp. Detekce se provádí pomocí preprocesoru podle existence makra `__EMSCRIPTEN__`.

Objekt DesktopApp slouží pro emulaci webové aplikace, takže řídí aktualizace a vykreslování, načítá model z uložených dat, která jsou exportována z webové aplikace a získává a zpracovává události z operačního sytému a od uživatele. Na rozdíl od něj, EmApp slouží pro vytvoření WebGL 2 kontextu a přeposílání žádostí z webové aplikace ke správným komponentám.

Interface

Ke komunikaci s JavaScripten byl použit Embind (součást projektu Emscripten). Jedná se o flexibilní a zároveň jednoduše použitelný systém popsaný v kapitole 2.5. Ukázka takového kódu je v C++ 4.2, kde je možné vidět propojení funkcí, které mají různé parametry i odlišné návratové hodnoty. Embind umožňuje export i celých tříd (to ale nebylo potřebné v tomto případě). Funkce jsou pak vztaženy k vytvořenému objektu v JavaScriptu a lze je volat bez jakýchkoli speciálních úprav (podle ukázky JavaScript 4.1 by volání funkce vypadalo `this._renderer.onResize(x,y)`). Je dobré upozornit na fakt, že Embind generuje názvy funkcí podle jejich typů parametrů, a proto je nutné přetypovat JavaScriptové objekty do správného typu (nedochází k automatickému přetypování float na int a podobně).

```
1 void RendererAPI::onResize(int width, int height) {/* ... */}
2 bool RendererAPI::addShape(const std::string& description) {/* ... */}
3 int RendererAPI::allocateFloat32(int count) {/* ... */}
4
5 EMSCRIPTEN_BINDINGS(renderer) {
6   emscripten::function("onResize", &RendererAPI::onResize);
7   emscripten::function("addShape", &RendererAPI::addShape);
8   emscripten::function("allocateFloat32", &RendererAPI::allocateFloat32);
9 }
```

C++ 4.2: Ukázka kódu využívající Embind API pro export několika funkcí do JavaScriptu.

Funkce využívají globální singleton `Application`, kterému jsou přeposílány zpracovaná data. Tato data jsou vytvořena z příchozích parametrů a případně z předaných identifikátorů sdílené paměti. Pro příchozí geometrii platí, že musí být nejprve odeslány informace o ní, kdy dojde k registraci identifikátoru se strukturou popisující tento objekt a začlenění geometrie do vykreslovací části. Pokud je geometrie registrována a dojdou její data, jsou dohledány informace o objektu v manažeru spravujícím geometrické objekty (MeshManager) a pokud existují, jsou data předána dále, kde se zpracují (nahrají na grafickou kartu). Textury například využívají TextureManager pro svoji registraci a přidělování sdílené paměti využívá MemoryManager.

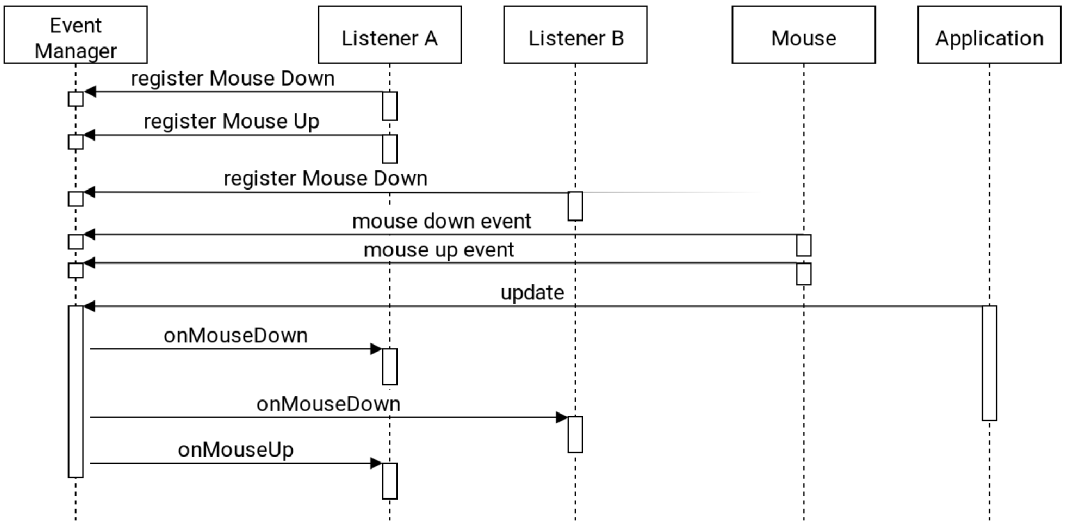
4.3 Přehled použitých manažerů

V této sekci budou popsáni manažeři, kteří se starají o určitý typ objektů. Modul obsahuje několik typů manažerů, z nichž ti důležití budou rozebráni níže.

Event manager – manažer událostí

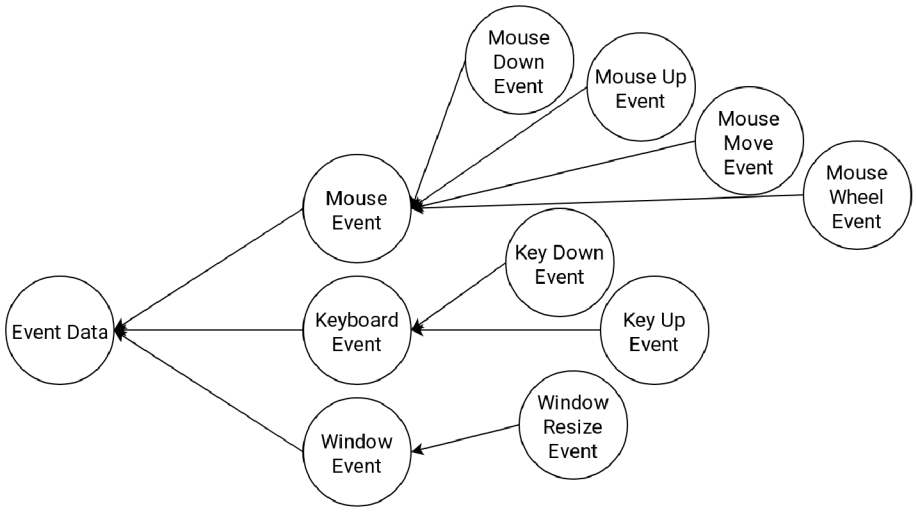
První manažer, který zde bude zmíněn, slouží ke zpracování událostí a jejich rozšíření registrovaným posluchačům. Stará se jak o události generované uživatelem (stisk tlačítka myši, klávesnice, změna velikosti okna, ...), tak i o události generované modulem. Pro registraci callback funkcí je použita knihovna SRDelegate uživatele Benjamin ze serveru GitHub [21], jedná se o modifikaci kódu ze článku The Impossibly Fast C++ Delegates [28].

`EventManager` nabízí funkce pro registraci nebo zrušení registrace callback funkcí pro specifickou událost, dále vložení události do fronty (fronta se zpracovává během aktualizace stavu a může být omezena časovým limitem) nebo lze provést událost okamžitě. Funkce `update` slouží ke zpracování událostí uložených ve frontě a při dosažení časového limitu odloží nezpracované události do dalšího aktualizačního cyklu. Implementace manažera je inspirována z knihy Game Coding Complete [26]. Pro lepší představu je na obrázku 4.2 znázorněn průběh registrace funkce pro poslouchání na události, vytvoření nové události a její šíření.



Obrázek 4.2: Registrace funkcí na odposlouchávání, vytvoření události a její šíření.

Události, které jsou podporovány tímto manažerem musí být zděděny ze třídy `EventData`, která požaduje přepsání funkcí pro získání typu (identifikátoru) a jména události. Jméno slouží pouze pro ladící účely, ale typ je důležitý a musí být unikátní (v modulu je reprezentován číselnou hodnotou). Pro identifikaci událostí nebyl použit výčtový typ, který by zajišťoval unikátnost identifikátorů, ale je použita statická konstanta definovaná pro každou třídu zvlášť, a to z důvodu, aby při přidání nové události nebylo nutné znova překládat veškerý kód využívající systém událostí. Znázornění hierarchie mezi událostmi je na obrázku 4.3, mezi tyto události lze vytvořit libovolnou další (události lze využít i k informování o příchozí geometrii, textuře a podobně).



Obrázek 4.3: Hierarchie mezi událostmi. Jakákoli nová událost musí dědit ze základní události Event Data a přepsat potřebné funkce.

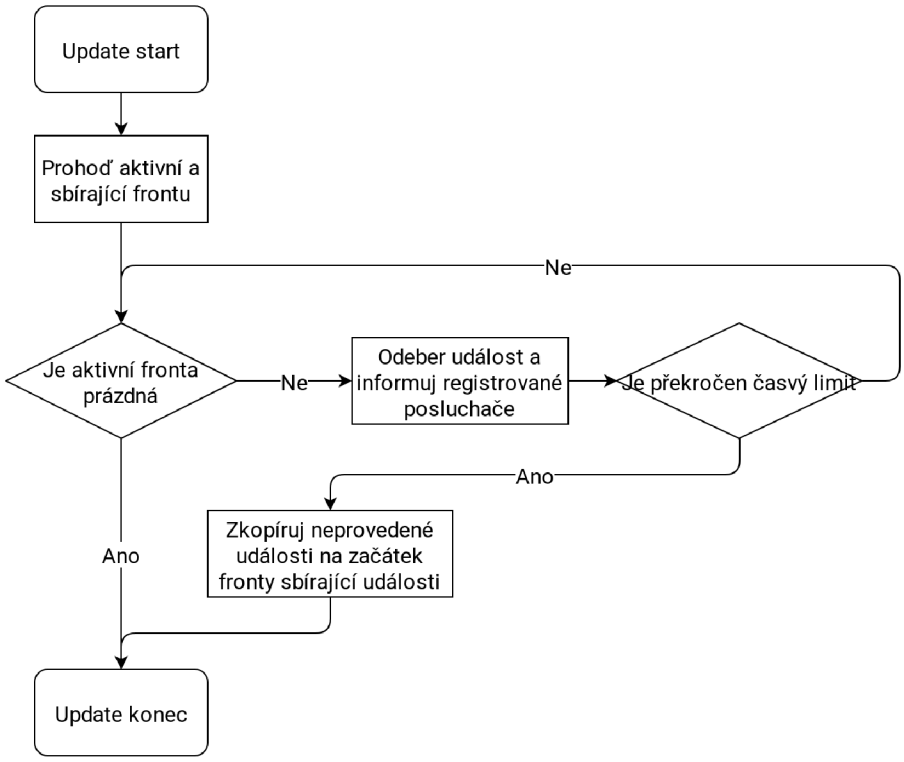
K registraci callback funkce nebo k jejímu odregistrování je nutné zjistit, zda funkce existuje / neexistuje, a podle toho ji vložit / odstranit z manažera. Vložení události provede její uložení do fronty sbírající události. Manažer pracuje se dvěma frontami, kde první fronta sbírá události a druhá je zpracovávána (aktivní) – zamezení zacyklení, kdy jedna událost generuje jinou a ta opět první. Okamžité provedení události najde všechny registrované funkce k danému typu a všem oznámí existenci této nové události. Složitější funkcí je aktualizační funkce `update`, jejíž provádění je zobrazeno na obrázku 4.4. Tato funkce provede prohození aktivní fronty a provádí v cyklu všechny uložené události, zároveň kontroluje, zda nepřekročila během provádění časový limit. Pokud je časový limit porušen, provádění skončí a neprovedené události jsou uloženy ve správném pořadí na začátek druhé fronty pro příští cyklus.

Mesh manager – manažer objektů k vykreslení

Jedná se o jednoduchý manažer, jehož hlavním účelem je registrace vykreslitelných objektů a jejich snadné nalezení podle identifikátoru. Objekty jsou uloženy pomocí `std::weak_ptr`, aby nezůstávaly v paměti, pokud už v programu nejsou potřeba. Rychlé vyhledávání objektů podle jejich identifikačního čísla je realizováno pomocí hashovací třídy `std::unordered_map`. Sekundárním úkolem manažera je sběr informací pro ladění (celkový počet objektů, jejich velikost, ...).

Texture manager – manažer textur pro objekty

Manažer je velice podobný třídě `MeshManager`. Jeho hlavní funkcí je registrace textur, které jsou validní pouze tak dlouho, dokud jsou potřeba (zajišťuje `std::shared_ptr` a `std::weak_ptr`). Pokud se textura nachází v paměti a je z aplikace požadavek o její nastavení na jiný objekt, použije se existující textura. Není nutné alokovat nový paměťový prostor a dekomprimovat příchozí data (JPEG, PNG). Sekundárním úkolem manažera je opět sbírání statistik.



Obrázek 4.4: Algoritmus šíření událostí.

Memory manager – správce paměti

`MemoryManager` je třída zajišťující vytváření a správu paměti, která může být sdílena mezi C++ modulem a JavaScriptovou aplikací. Manažer využívá počítání referencí dostupné z objektu `std::shared_ptr`, který slouží pro automatické uvolnění paměti. Toto počítání je dostupné pouze v rámci C++ kódu, a tak pro JavaScript vznikla funkce uvolňující paměť (ke snížení počítadla referencí). Dále manažer emuluje typy pamětí, které jsou dostupné v JavaScriptu (Float32Array, Uint8Array, ...).

Task manager – manažer déle trvajících procesů

Během vývoje modulu bylo zjištěno, že některé operace trvají příliš dlouho na to, aby mohly být provedeny všechny v jednom cyklu, a tak bylo nutné tyto operace buď vložit do jiného vlákna, nebo je rozptýlit mezi více cyklů. Bohužel Emscripten momentálně nepodporuje vlákna, tak jak jsou známa z jazyka C++, ale pouze Workers, jejichž velkou nevýhodou je nemožnost sdílení paměti, kterou by bylo nutné neustále kopírovat tam i zpět. Vývoj vláken založených na `pthreads` je aktivní, a tak je jen otázkou času, kdy budou k dispozici a modul bude přepsán na vícevláknovou aplikaci.

`TaskManager` využívá ke své práci frontu tzv. lambda funkcí, které lze vkládat za sebe. Tyto funkce si zachovávají pořadí a jejich volání probíhá podobně, jako šíření událostí v manažeru událostí. Manažer obsahuje aktualizační funkci `update`, která provádí tolik úkolů, kolik se jich vleze do časového limitu. Pokud je časový limit překročen, provádění skončí. Tento manažer našel své uplatnění při vytváření bufferů, dekomprimaci textur a jejich nahrávání na grafickou kartu.

Shader manager – manažer shader programů

Pro ladění aplikace, která běží v desktopovém prostředí, byl implementován `ShaderManager`, který dokáže při spuštěné aplikaci detekovat změny v shader kódech (podle časového razítka) a dynamicky je rekompiluje. Aplikaci je tak možné ladit a vylepšovat přímo za běhu, je tedy ušetřeno mnohdy hodně času, zvláště pokud se jedná o aplikaci, která se dlouho načítá (při větším modelu se i tento renderer načítá poměrně dlouho).

Manažer obsahuje funkce, kterými je informován o vytvoření nového shader programu (tento program je pak zaregistrován do interní reprezentace) a o jeho smazání. Dále obsahuje aktualizační funkci `update`, která v rámci cyklu ověří, zda došlo k nějaké změně a následně pokud došlo, dojde k rekompilaci změněné části. Jelikož je shader program zapouzdřen do objektové C++ reprezentace, je možné bez problému prohodit jeho interní identifikátor na OpenGL shader program za nový. Funkce update je volána v delších časových intervalech než u ostatních objektů (přibližně jednou za sekundu).

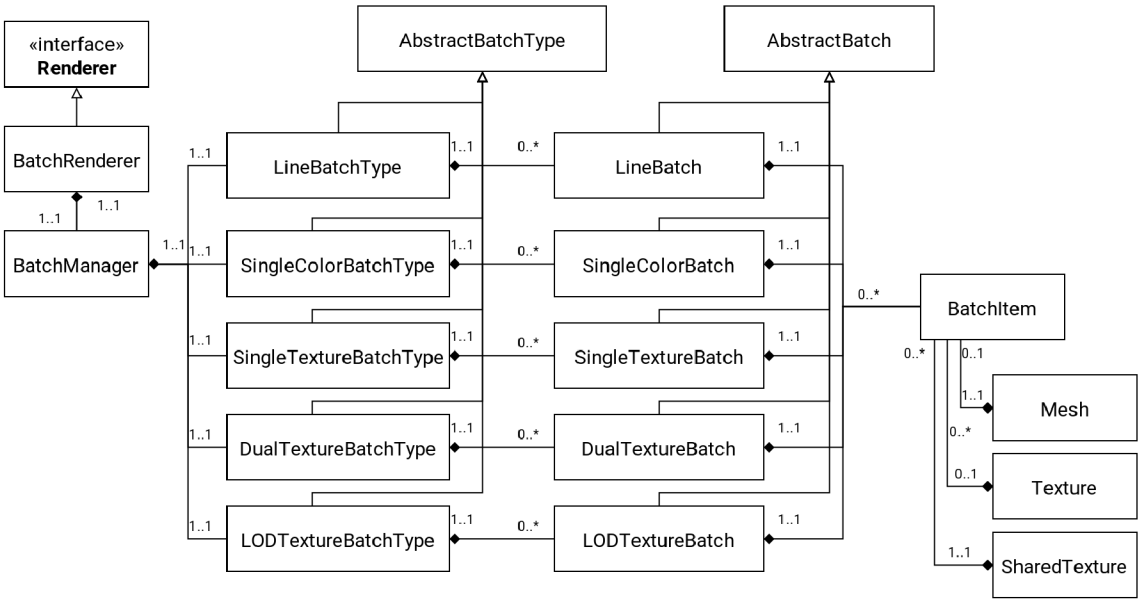
4.4 Implementace vykreslovací části modulu

I když se modul zabývá vykreslováním, z předchozích kapitol je jasné, že obsahuje mnohem více věcí, než jen vykreslování. Vykreslovací část je interně oddělena pomocí abstraktního objektu obsahujícího pouze interface (rozhraní) – `Renderer`, ze kterého může být odvozeno několik různých kreslících objektů. Modul používá dva typy, a to `BatchRenderer` a `SimpleRenderer`. Celá práce se zabývá optimalizovaným rendererem používajícím batchování, ale během počátku vývoje byl vytvořen jednoduchý renderer – to co dostane na vstup, to vykreslí, bez jakýchkoli optimalizací. Výkonnostní rozdíl mezi těmito renderery je poznat už při menším počtu objektů, ale při několika desetitisících objektů je nemožné používat SimpleRenderer i na výkonných herních počítačích (testováno na grafické kartě NVIDIA GeForce GTX 1060 6GB).

Batch Renderer

Návrh tohoto rendereru je probrán v sekci 3.4, zde budou probrány implementační detaily celku i specializovaných podčástí. Objekt třídy BatchRenderer implementuje rozhraní ze třídy Renderer. Tato třída se stará o správnou sekvenci vykreslovacích příkazů (z-prepass, vykreslení neprůhledných prvků, čar, transparentních prvků, ...) a obsahuje BatchManager, který spravuje rozdílné typy Batch objektů (každý typ se specializuje na určitou geometrii – obrázek 3.1). BatchType objekty dědí část kódu z abstraktní třídy AbstractBatchType, která eliminuje duplicitu kódu. Tyto objekty slouží ke správnému nastavení základních atributů pro shader program. Každý BatchType objekt obsahuje několik Batch objektů, které lze konečně vykreslit – jedná se o ucelený blok dat, které lze poslat na grafickou kartu ke zpracování. Batch objekty jsou složeny z BatchItem objektů, tyto objekty reprezentují jednotlivé geometrie, které byly sloučeny do většího bloku dat. Propojují geometrii s Batch objektem a texturou. Diagram znázorňující předchozí text je na obrázku 4.5.

BatchRenderer obsahuje frame buffer object, který se skládá z více textur (MRT), které slouží pro zobrazení barev a zároveň pro ukládání dodatečných informací, jako je identifikátor objektu a jeho pozice ve scéně. S podporou nových formátů textur je možné vytvořit přesnou reprezentaci dat, která je potřeba (R32UI – 32 bitový unsigned int v červené složce textury, RGBA32F – 32 bitový float ve 4 složách textury atd.).



Obrázek 4.5: Diagram zobrazující interní reprezentaci třídy BatchRenderer.

Nastavení textury probíhá tak, že je její obsah nejdříve dekomprimován (pomocí TaskManager objektu) a po dokončení je nastavena všem požadovaným objektům. Momentálně jsou podporovány pouze formáty PNG a JPEG, které se pro jednoduchost rozbalují do formátu RGBA8 (zde je možná optimalizace paměti, kdy JPEG a PNG bez průhlednosti je možné rozbalit pouze do formátu RGB8 a šedé textury pouze do formátu R8). Více formátů by znamenalo větší dělení polí textur uvnitř Batch objektů, a proto bylo momentálně vynecháno.

Vykreslování je zde jako sekvence příkazů popisující jednotlivé etapy. Před každou etapou je nutné správně nastavit požadované parametry (správný zapisovací buffer, práce s hloubkovou maskou, blending, ...) a na závěr je obsah barevné textury z MRT frame bufferu nahrán do screen bufferu (defaultního bufferu vytvořeného kontextu). Etapy vykreslování v modulu jsou: vyčištění obsahu bufferu, z-prepass, neprůhledné objekty, průhledné objekty a čáry.

Batch Manager

`BatchManager` slouží ke správě různých typů objektů, které se liší jak způsobem reprezentace geometrie, tak třeba i počtem textur. Jedná se o vrstvu, která má zajistit transparentnost přidání "neurčitého" grafického objektu do scény. Obsahuje všechny specializované BatchType objekty popsané níže. Přeposílá události, které mu přicházejí buď do správného BatchType objektu (přidání geometrie, nastavení textury, ...), nebo je rozšíří všem (inicializace, aktualizace stavu, vykreslení).

Abstract Batch Type

Společná třída pro všechny BatchType objekty, která slouží ke správě Batch objektů a shader programů. Každý BatchType objekt má seznam podporovaných etap vykreslení a ke každé podporované etapě má definovaný shader program. Pokud je vykreslovací etapa

podporována, je nastaven shader program, dále jsou nastaveny všechny společné parametry a atributy a dochází ke kreslení všech uložených Batch objektů (detailnější popis je níže).

Jelikož se jedná pouze o abstraktní třídu, je potřeba přepsat nebo zcela implementovat některé funkce. Je nutné přepsat inicializaci objektu, protože v ní jsou vytvářeny shader program objekty, dále kreslení a práce s objekty typu Batch (vytvoření, vyhledání). Třída nabízí aktualizační funkci update, která se stará o aktualizaci interního stavu a stavu uložených objektů (pokud dojde k jejich změně, mohou být označeny k odstranění, a dojde k setřídění podle jejich požadavků na stav vykreslování – face culling, polygon offset).

Abstract Batch

Jak již bylo zmíněno, každý Batch Type objekt obsahuje několik vykreslitelných objektů typu Batch. Aby nedocházelo k duplikaci kódu, existuje abstraktní objekt, ze kterého jsou další objekty odvozeny a rozšiřují jeho funkcionalitu (například o práci s texturami). Tento objekt je složen z mnoha Batch Item objektů a slouží k seskupení jejich dat. Abstraktní objekt obsahuje metody pro zpracování geometrie jak s indexovanými vrcholy tak i bez indexů, pro správný počet atributů je parametrizovaný – při dědění si nastaví každý odvozený objekt určitý počet atributů na vrchol. Každý Batch objekt obsahuje funkci, která musí jednoznačně určit, zda je možné vložit vykreslovací objekt podle jeho meta dat. Další porovnávací funkce slouží k setřídění podle jejich parametrů pro vykreslování – dochází ke snížení počtu změn stavu grafické karty.

Po ověření informací dochází k registraci objektu (nastaví se počítadla – počet vrcholů, indexů, textur) a čeká se na příchozí data. Dojdou-li data v pořádku, jsou vložena do mezipaměti a čeká se na aktualizační signál (využívá se zde Task Manager), ve kterém se vytvoří nové buffery nebo se přidají změny na konec. Pokud existují v bufferech nějaká data, jsou průběžně vykreslována (není nutné čekat na všechny registrované objekty). Dojde-li objekt a Batch nemá dostatečně velký buffer pro jeho data, dojde k vytvoření nového většího bufferu a ke kopii existujících dat do něj, vše probíhá na grafické kartě bez nutnosti kopírování dat z a do paměti RAM. Nejedná se tedy o výrazný výkonnostní problém, který by mohl vzniknout při přesunu paměti mezi CPU a GPU.

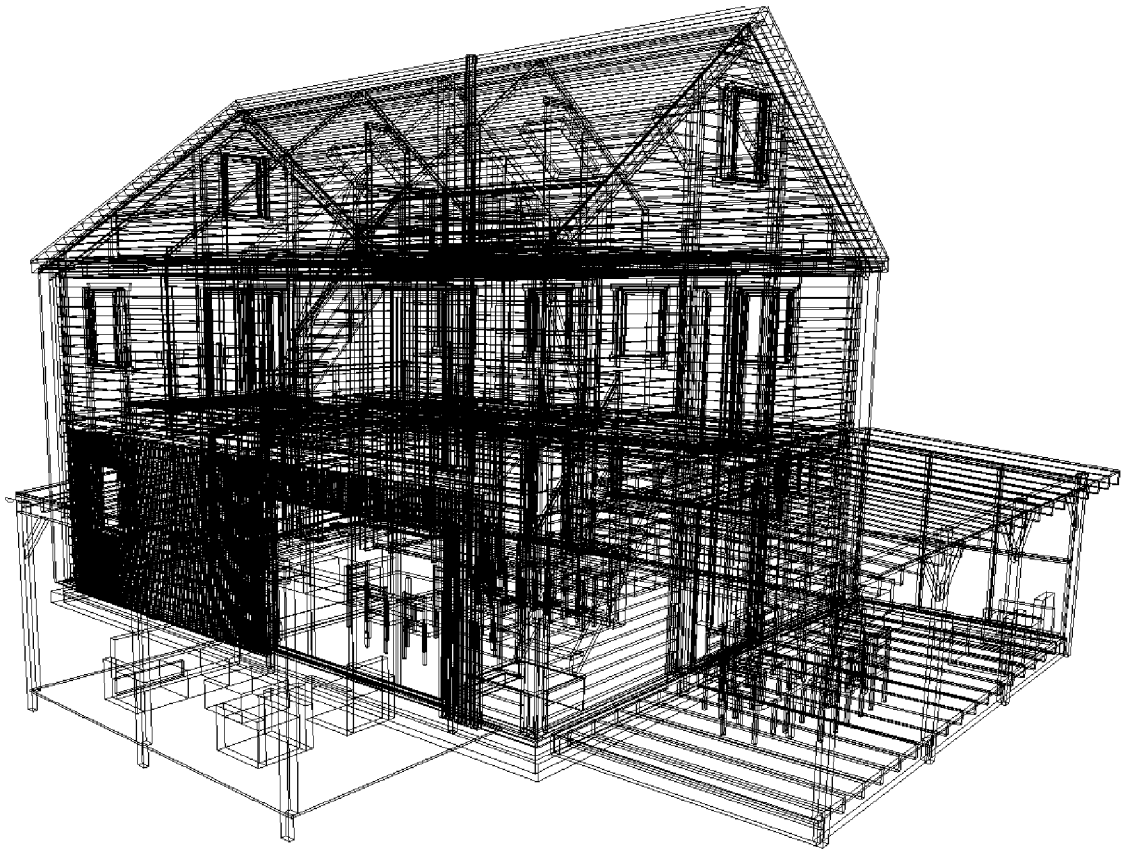
Odstranění objektu je složitější proces. Odstranění kreslitelného objektu hned po registraci je jednoduché, dochází pouze ke změně počítadel, pokud však už došla data objektu, jsou možné dvě následující možnosti. Data jsou zpracována a nahrána na grafické kartě, pak je nutné odebrat objekt z interní reprezentace a přepočítat celý buffer (tato operace je náročná na výpočetní výkon) nebo data ještě nejsou zpracována a dochází k odstranění objektu ze seznamu nově příchozích a snížení počítadel.

Změna dat objektu je poměrně jednoduchá (pokud se netýká geometrie či textur). Stačí změnit údaje ve sdílené textuře a změna se projeví okamžitě. Tuto operaci je vhodné optimalizovat tak, že se změny zapíšou pouze do paměti RAM, kde je duplikována textura a následně se nahrají změněné vrstvy této textury na grafickou kartu (při aktualizační funkci). Předpoklad vychází ze situace, kdy se tisíci objektům změní modelová matice (například je posunuta skupina objektů).

Práce s texturami je individuální pro děděné objekty. V aplikaci se objevují objekty, které mají žádnou, jednu, dvě (náhled + maximální) a nebo N textur (LOD). Pro tyto účely jsou Batch objekty specializované a každý má určitý typ optimalizace.

Line Batch – vykreslování čar

Jedná se o specializaci Batch objektu na vykreslování úseček. Každá úsečka je definována dvěma body (počáteční a koncový). Shader program je velice jednoduchý, vertex shader má na vstupu pouze informaci o pozici vrcholu a odkaz na sdílenou datovou texturu, ze které získá modelovou matici, barvu a stav. Fragment shader buď zahodí fragmenty (pokud není objekt viditelný) nebo nastaví barvu do výstupního bufferu. `LineBatch` je prakticky totožný se svým rodičem, jediným rozdílem je nastavení parametru šířky čáry před vykreslováním. Výsledek z tohoto bufferu je na obrázku 4.6.



Obrázek 4.6: Výstup z aplikace při vykreslování pouze čar.

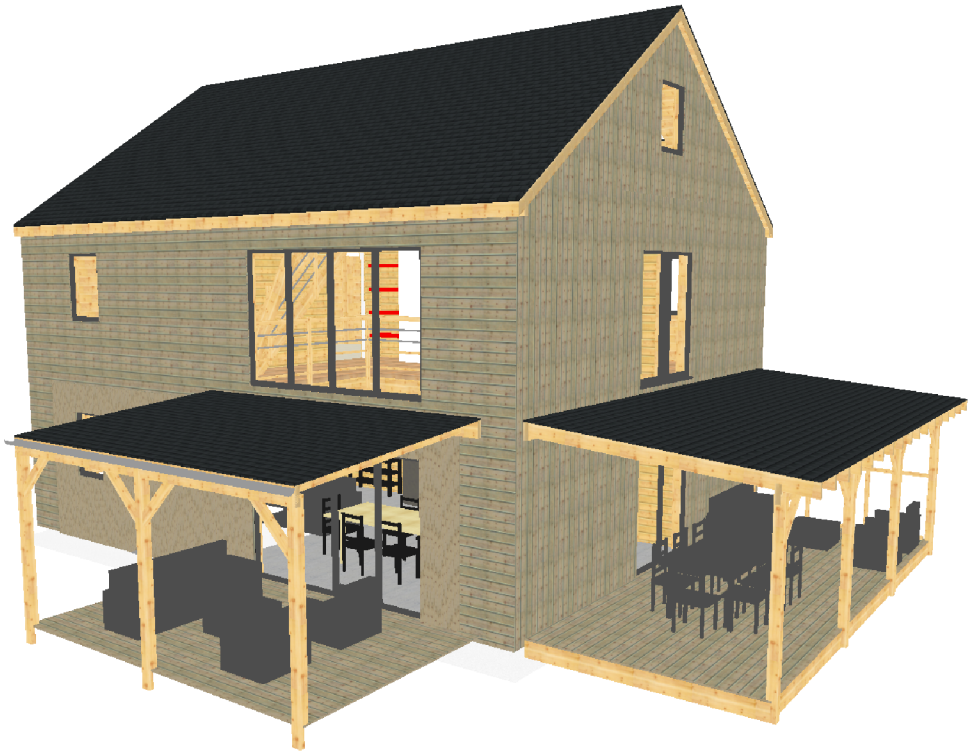
Single Color Batch – vykreslování geometrie s jednou barvou

Jednoduchými objekty k vykreslení jsou ty, které neobsahují texturu. Tyto objekty obsahují vykreslovací parametry, které je nutné nastavit před kreslením a barvu, kterou lze získat ze sdílené datové textury (spolu s modelovou maticí). Oproti svému rodiči nepřináší výrazné změny. Použitý vertex shader obsahuje 3 atributy – pozici, normálu vrcholu a odkaz do sdílené textury, z které se získá barva, modelová matice, identifikátor a další. Ve fragment shaderu se provede detekce viditelnosti, výpočet osvětlení, test zda je objekt vybrán (zvýraznění) a k výsledné barvě se do dalšího bufferu uloží identifikátor a pozice objektu (`gl_FragCoord`).

Single Texture Batch – vykreslování geometrie s jednou texturou

Do doby, než je objektu k vykreslení nastavena textura, chová se tento Batch téměř stejně jako SingleColorBatch (dochází k vykreslení geometrie s jednou barvou). Tento Batch momentálně využívá 8 texturovacích jednotek z běžně 16 dostupných na většině zařízení podporující WebGL 2, kde každá jednotka obsahuje pole textur o určité velikosti. Batch je rozšířen o práci s texturami, kdy se při registraci objektu provede test, zda textura odpovídá některé z už registrovaných velikostí textur (a není dosažen horní limit počtu těchto textur) nebo zda je některá texturovací jednotka volná. Po registraci se čeká na příchozí data, která jsou zpracována stejně jako u rodiče. Po vyplnění daty je objekt vykreslován pouze jednolitou barvou a při příchodu textury se přenastaví index reprezentující texturovací jednotku na některou z validních možností a objekt je vykreslován správně. Do takto optimalizovaného Batch objektu se vleze velké množství objektů a dochází k výraznému navýšení výkonu (jak je zobrazeno například na obrázku 5.2, kde model běží v reálném čase bez problémů, i když má přes tisíc textur). Tento Batch je vhodný pro scény, kde se objekty příliš často nemění.

Vertex a fragment shader obsahuje navíc oproti předchozím specializacím pouze texturovací souřadnice a celočíselný index, podle kterého se vybírá barva nebo správná texturovací jednotka. Pokud je index záporný, textura ještě není připravena a objekt je vykreslen stejně, jako by žádnou texturu neměl. Výsledek vykreslení scény tímto Batch objektem (bez průhledných objektů) je na obrázku 4.7.



Obrázek 4.7: Výstup z aplikace při vykreslování objektů s jednou texturou a bez průhlednosti.

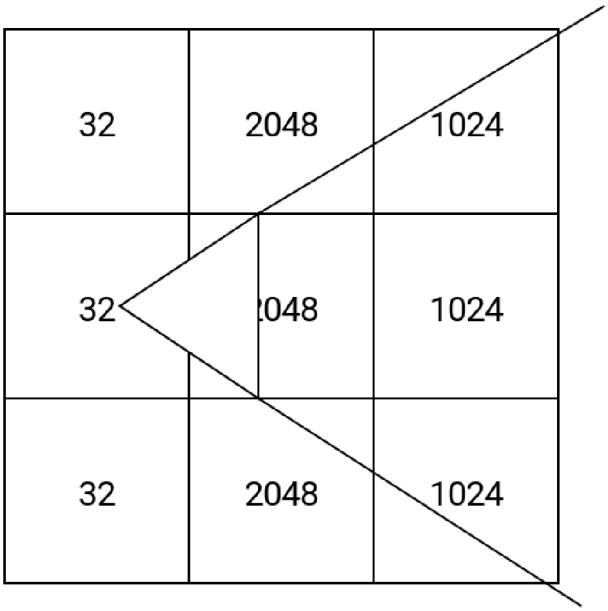
Dual Texture Batch – vykreslování geometrie obsahující náhled a texturu v plné velikosti

Částečně se jedná o metodu level of detail, kdy je nejprve stažena textura s malým rozlišením a následně se stáhne textura s maximálním rozlišením. Na rozdíl od níže uvedeného Batch objektu nedochází k přechodu zpět na nižší rozlišení (zde se nemluví o (ne)využití mipmap). Implementace vychází z předchozího Batch objektu, s tím rozdílem, že pro každou skupinu objektů existují vždy dvě pole textur. Zde je možné využít hned několik návrhů. Jedna z možností je použít jen jedno pole textur na náhledy, zbytek texturovacích jednotek by se choval stejně jako při objektech s jedinou texturou. Dalším možným řešením je vložení plně načtených objektů do předchozího Batch objektu a jejich odstranění odsud.

LOD Texture Batch – vykreslování geometrie s mnoha texturami

Level of detail, nebo-li metoda s různým stupněm detailu, slouží ke snížení náročnosti a zvýšení výkonu při kreslení objektů, které nepotřebují maximální detaily. V existující aplikaci je tato metoda stále ve vývoji, a proto byla zatím implementována jen metoda s dynamickou změnou textury (která stále není plně optimální) a geometrie. Její princip spočívá v tom, že pokud zůstane geometrie objektu stejná, dojde pouze ke změně textury (na větší nebo menší rozlišení). Dojde k nahrání dat na GPU a úpravě indexu. Pokud dojde těchto požadavků několik v jednom cyklu, pak mohou být podle doby provádění odloženy do dalších cyklů – snaha o plynulost. Až po nahrání textury na GPU je objekt informován o změně, upraví svůj index a odstraní starou texturu (dochází k počítání referencí, mnoho objektů může sdílet stejnou texturu). Pokud objekt obsahuje jinou geometrii, je nutné jej vložit do jiného Batch objektu (případně i do stejného) a počkat na nahrání nové textury. Poté je nutné odstranit objekt ze starého Batch objektu a jeho texturu. Pro různé rozlišení textur je (stejně jako u SingleTextureBatch objektu) využito několika texturovacích jednotek.

Změna rozlišení textur je znázorněna na obrázku 4.8. Při změně rozlišení textury může být změněna i geometrie.

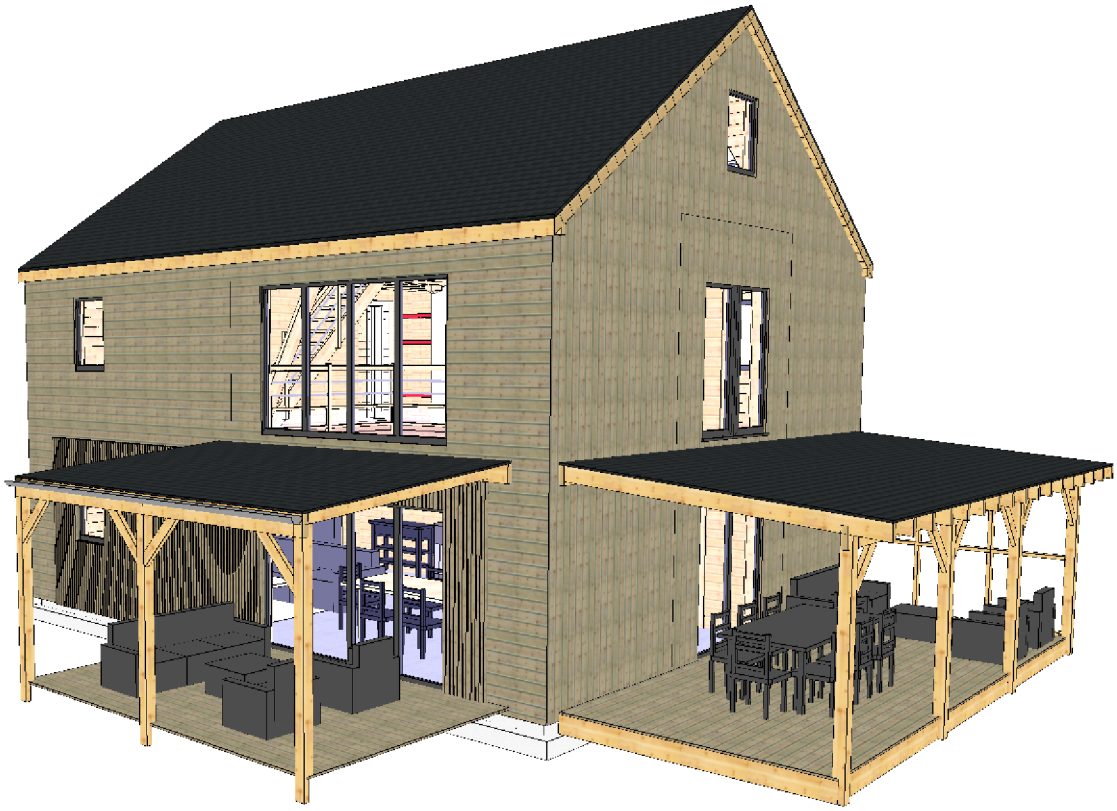


Obrázek 4.8: Princip změny rozlišení textury v závislosti na kameře.

Kapitola 5

Porovnání existujícího řešení s novým vykreslovacím modulem

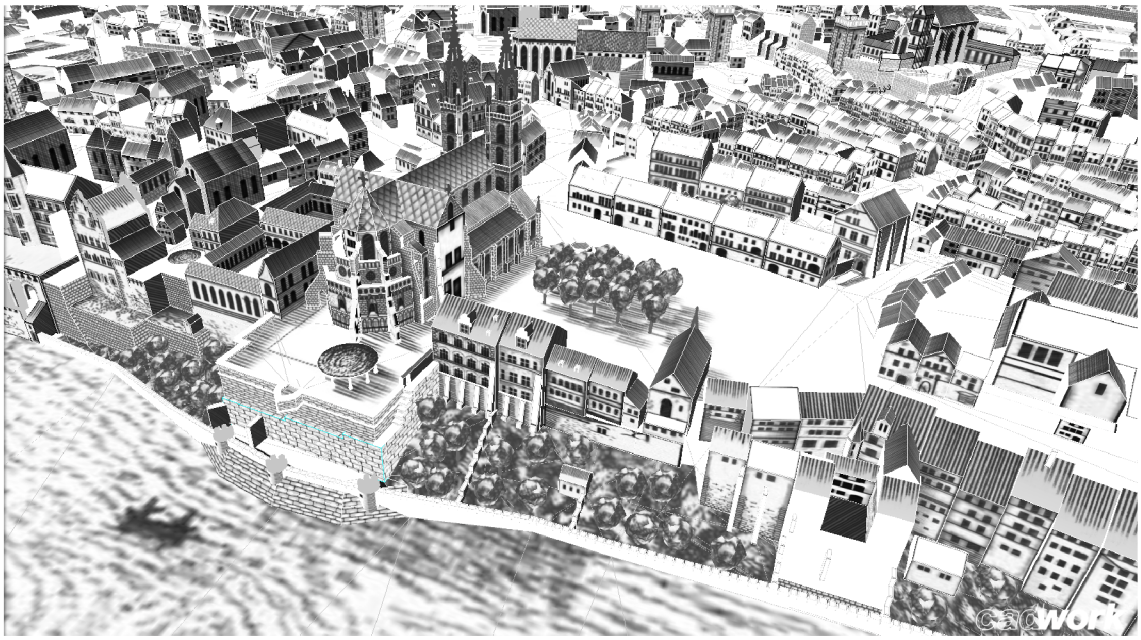
Při využití všech výše uvedených principů a postupů bylo docíleno vykreslování většiny existujících modelů. Věci, které nebyly implementovány, zahrnují například plynulé přepínání textur při metodě LOD (dochází k trhání obrazu způsobeným delší dobou zpracování a nahrávání textur o rozlišením větším jak 2048 pixelů). Dále nebyly implementovány efekty, jako jsou stíny a ambient occlusion. Vykreslený testovací model je na obrázku 5.1.



Obrázek 5.1: Výstup z nové aplikace. Aplikace podporuje širokou škálu objektů a jejich optimalizaci pomocí vykreslení v dávkách. Na obrázku je patrný aliasing a taky spousta chybějících efektů – stíny, řezy, ...

5.1 Měření výkonnosti

Pro porovnání vykreslujících modulů budou sloužit následující 4 modely, kde se každý model zaměřuje na jinou problematiku: maison, merian, andermatt, milano. Model maison je z nich nejmenší a obsahuje velice málo vrcholů a objektů a je vhodný pro vývoj (více údajů je uvedeno v tabulce 5.1). Maison je model, který doprovází tuto práci již od samého úvodu z obrázku 1.1 a jehož vykreslení novým rendererem je na obrázku 5.1. Model merian je ručně dělaný model vycházející z černobílé kreslené předlohy, tento model obsahuje výrazné množství různých textur a jeho vykreslení je na obrázku 5.2. Model, který sloužil pro testování metody LOD se jmenuje Andermatt (jedná se o terén kolem stejnojmenného města ve Švýcarsku) a modelem obsahujícím spoustu vrcholů je milano.



Obrázek 5.2: Výstup z webové aplikace, která obsahuje přes tisíc textur a přesto běží plynule na běžném PC.

Model \Počet	vrcholů	objektů	průhledných	čar	textur
Maison	113 995	160	12	170	29
Merian	659 278	15 111	0	14 439	1 411
Andermatt	15 286	64	0	0	65
Milano	4 634 590	22 685	0	22 685	6

Tabulka 5.1: Údaje o použitých modelech.

Srovnání probíhalo na počítačové sestavě s procesorem Intel©Core™i7-7700K, grafickou kartou NVIDIA GeForce GTX 1060 6GB a aplikace využívala plnou velikost FullHD monitoru (1920 x 1080) a aplikaci Google Chrome. Měření probíhalo celkem 5x a každý model byl testován vícekrát. Měřil se jeden cyklus aplikace (vždy se provedla aktualizace scény a následné vykreslení – bylo zakázáno přeskočit vykreslovaní). Průměrovalo se po sobě následujících 60 cyklů, kde se využívala funkce `performance.now` pro měření začátku

a konce cyklu a po skončení vykreslování (před měřením konce) se volala funkce `glFinish` pro počkání na dokončení vykreslování. Naměřené výsledky jsou v tabulce 5.2.

Model \Metoda [ms]	Originál	WebAssembly	asm.js
Maison	0.919	0.414	0.587
Merian	31.911	2.485	2.643
Andermatt	0.909	0.348	1.286
Milano	0.876	0.425	1.535

Tabulka 5.2: Průměrná doba vykreslení modelu v milisekundách na testovací sestavě v prohlížeči Google Chrome.

Z tabulky je patrné, že renderer využívající WebAssembly je přibližně 2x rychlejší. Výjimkou je model Merian, kde je dosaženo více jak deseti násobku rychlosti. To je způsobeno využitím polí textur, kdy se výrazně snížil počet vykreslovacích příkazů a nastavování různých textur před kreslením. Modul byl bezproblémově přeložen i do formátu asm.js, kdy stačilo změnit jeden parametr při překladu. Výsledné časy jsou zajímavé, protože tento formát byl rychlejší než originál pouze ve dvou případech, v modelu merian a v modelu maison. V ostatních případech zaostával. Výsledky měření mezi originálem a novým modulem nejsou absolutní, protože bylo využito jiných technik při implementaci, které mohou výrazně ovlivnit výkonnost. Zajímavé je porovnání dvou vyexportovaných formátů, protože i při použití stejných implementačních technik, jen se změnou parametru při překladu, je WebAssembly často mnohem rychlejší než asm.js a jindy jsou rozdíly zanedbatelné.

Měření proběhlo i na mobilním zařízení. Vzhledem k omezené výkonnosti byl použit pouze nejmenší model. Údaje jsou v tabulce 5.3. Zde je originální aplikace výrazně rychlejší (pravděpodobně z důvodu optimalizace shader programů, které mají co nejvíce výpočtů přesunutých do vertex shaderu a sníženou přesnost float čísel, dalším důvodem může být neexistence modelových matic v původním vykreslovacím modulu – dochází tedy k mnohem méně matematickým operacím).

Originál	WebAssembly	asm.js
68.131	88.426	90.033
66.066	88.262	90.721
72.033	88.524	92.918
62.540	87.869	90.787
64.066	87.606	88.557
66.5672	**88.1374**	**90.6032**

Tabulka 5.3: Rychlost vykreslení modelu maison v milisekundách na mobilním zařízení Samsung Galaxy S7 v prohlížeči Mozilla Firefox 59.0.2.

Kapitola 6

Závěr

Na začátku práce byly rozebrány technologie, které lze využít v moderních prohlížečích a které byly použity i v novém vykreslovacím modulu. Z těchto technologií byl zmíněn JavaScript, který se objevil roku 1996 (v prohlížeči Netscape 2) a stále je používán i v moderních prohlížečích a byly zmíněny technologie, které se snaží tento jazyk částečně nahradit (alespoň co se týče výkonnostně kritických sekcí). Dále bylo uvedeno, že moderní prohlížeče podporují urychlení vykreslování pomocí WebGL – webové grafické knihovny, která je nyní ve verzi WebGL 2, a která na podporovaných zařízeních obsahuje mnohem více možností než její předchozí verze. Dalším důležitým prvkem, který byl zmíněn, je open-source překladač Emscripten využívající LLVM. Emscripten umožňuje překlad aplikací psaných v některých jazycích (jako je například C, C++ a Rust) do webových formátů – WebAssembly, asm.js.

Původní aplikace, která momentálně využívá čistý JavaScript, byla modernizována o vygenerovaný kód ve formátu WebAssembly s tím, že tento kód může být vygenerován i do formátu asm.js pro podporu starších prohlížečů. Dalším krokem modernizace byl přechod k nové verzi grafické knihovny – WebGL 2. V práci byla popsána metoda optimalizace pomocí vykreslování skupin objektů najednou a využití polí textur (Array Texture) a několika texturovacích jednotek. Během návrhu a implementace došlo k jednomu výraznému problému, kdy bylo zjištěno, že WebAssembly momentálně nepodporuje vlákna, jak jsou známa z jazyka C++. Problém byl částečně eliminován tak, že déle trvající události jsou rozloženy do několika cyklů aplikace.

Vykreslovací modul obsahuje ještě spousty míst, na kterých by mohl být vylepšen. Tato vylepšení zahrnují paměťovou náročnost, protože i černobílé a neprůhledné textury momentálně zabírají plný rozsah paměti (4 barevné složky). Dalším vylepšením, co se týče textur, je chybějící podpora komprimovaných formátů. Modul není optimalizovaný pro mobilní zařízení, jak je patrné při srovnání rychlosti s originální aplikací, a chybí některé efekty, které existují v původní aplikaci (například stíny, řezy modelu, skybox). I přes tyto nedostatky už teď dosahuje kreslící modul zajímavých výsledků a přispěl ke zrychlení vykreslování modelů.

Literatura

[1] *Chrome for Desktop*. [Online; navštíveno 27.11.2017].
URL https://www.google.com/chrome/browser/desktop/index.html

[2] *WebAssembly Working Group*. [Online; navštíveno 5.12.2017].
URL https://www.w3.org/wasm/

[3] *About Emscripten*. 2015, [Online; navštíveno 5.12.2017].
URL http://kripken.github.io/emscripten-site/docs/introducing_emscripten/about_emscripten.html

[4] *Array Texture*. Říjen 2015, [Online; navštíveno 12.4.2018].
URL https://www.khronos.org/opengl/wiki/Array_Texture

[5] *Embind*. 2015, [Online; navštíveno 17.12.2017].
URL http://kripken.github.io/emscripten-site/docs/porting/connecting_cpp_and_javascript/embind.html?highlight=embind

[6] *Module Object*. 2015, [Online; navštíveno 17.12.2017].
URL http://kripken.github.io/emscripten-site/docs/api_reference/module.html?highlight=module

[7] *Optimizing WebGL*. 2015, [Online; navštíveno 29.12.2017].
URL https://kripken.github.io/emscripten-site/docs/optimizing/Optimizing-WebGL.html

[8] *Pthreads support*. 2015, [Online; navštíveno 18.3.2018].
URL https://kripken.github.io/emscripten-site/docs/porting/pthreads.html

[9] *About JavaScript*. 2017, [Online; navštíveno 27.11.2017].
URL https://developer.mozilla.org/en-US/docs/Web/JavaScript/About_JavaScript

[10] *Desktop Browser Market Share Worldwide*. 2017, [Online; navštíveno 27.11.2017].
URL http://gs.statcounter.com/browser-market-share/desktop/worldwide

[11] *Getting started with WebGL*. 2017, [Online; navštíveno 27.11.2017].
URL https://developer.mozilla.org/en-US/docs/Web/API/WebGL_API/Tutorial/Getting_started_with_WebGL

[12] *HTML5*. 2017, [Online; navštíveno 18.12.2017].
URL https://developer.mozilla.org/en-US/docs/Web/JavaScript/Typed_arrays

[13] *Internet pro lidi, ne pro peníze*. 2017, [Online; navštíveno 27.11.2017].
URL http://webassembly.org/

[14] *JavaScript typed arrays*. 2017, [Online; navštíveno 17.12.2017].
URL
https://developer.mozilla.org/en-US/docs/Web/JavaScript/Typed_arrays

[15] *macOS - Safari*. 2017, [Online; navštíveno 27.11.2017].
URL https://www.apple.com/safari/

[16] *Microsoft Edge | Oficiální web prohlížeče pro Windows 10*. 2017, [Online; navštíveno 27.11.2017].
URL https://www.microsoft.com/cs-cz/windows/microsoft-edge

[17] *The WebGL API: 2D and 3D graphics for the web*. 2017, [Online; navštíveno 4.12.2017].
URL https://developer.mozilla.org/en-US/docs/Web/API/WebGL_API

[18] *TypeScript - JavaScript that scales*. 2017, [Online; navštíveno 21.3.2018].
URL http://www.typescriptlang.org/

[19] *WebAssembly*. Listopad 2017, [Online; navštíveno 5.12.2017].
URL https://developer.mozilla.org/en-US/docs/WebAssembly

[20] *WebGL 2 lands in Firefox*. 2017, [Online; navštíveno 4.12.2017].
URL https://developer.mozilla.org/en-US/docs/Web/API/WebGL_API

[21] yxbh (Benjamin): *C++ Delegate Library Collection*. 2018, [Online; navštíveno 20.3.2018].
URL https://github.com/yxbh/Cpp-Delegate-Library-Collection

[22] Cozzi, P.: *WebGL Insights*. CRC Press, July 2015, ISBN 978-1498716079, http://www.webglinsights.com/.

[23] Graham, S.: *Mini Jpeg Decoder*. Prosinec 2009, [Online; navštíveno 22.3.2018].
URL http://h4ck3r.net/2009/12/02/mini-jpeg-decoder/

[24] McAnlis, C.; Lubbers, P.; Jones, B.; aj.: *HTML5 Game Development Insights*. Apress, 2014, ISBN 978-1430266976.

[25] McDonald, J.: *Eliminating Texture Waste: Borderless Ptex*. Březen 2014, [Online; navštíveno 21.3.2018].
URL https:
//www.slideshare.net/basisspace/borderless-per-face-texture-mapping/

[26] McShaffry, M.; Graham, D.: *Game coding complete*. Course Technology, 2013, ISBN 978-1-133-77657-4.

[27] nlohmann: *JSON for Modern C++*. [Online; navštíveno 22.3.2018].
URL https://github.com/nlohmann/json

[28] Ryazanov, S.: *The Impossibly Fast C++ Delegates*. Červenec 2005, [Online; navštíveno 20.3.2018].
URL https:
//www.codeproject.com/Articles/11015/The-Impossibly-Fast-C-Delegates

[29] Vandevenne, L.: *LodePNG for C (ISO C90) and C++*. 2013, [Online; navštíveno 22.3.2018].
URL http://lodev.org/lodepng/

[30] Wallace, E.: *WebGL Water*. [Online; navštíveno 21.3.2018].
URL http://madebyevan.com/webgl-water/

Příloha A

Obsah přiloženého paměťového média

/3rdParty Adresář se závislostmi na produkty třetích stran, obsahuje všechny hlavičkové soubory a knihovny potřebné k překladu. Uvnitř adresáře jsou složky společné pro oba projekty (include) a složky specifické pro překlad aplikace v prostředí Windows nástrojem Visual Studio (vsInclude, vsLib) a pro překlad webového modulu překladačem Emscripten (emInclude).

/assets Adresář, ve kterém se nachází soubory potřebné při spuštění, jako jsou shadery, pod Windows jsou zde dynamické knihovny (.dll) a exportované modely z webové aplikace, které jsou načteny při spuštění nativní aplikace.

/em Skripty ke generování překladových informací pro Emscripten. Překlad probíhá do složky build s využitím dočasné složky tmp. Skripty jsou generovány pro nástroj ninja. K vygenerování slouží soubor generate.py, který lze spustit v Python 2 a pro dodatečné úpravy vygenerovaných souborů, případně pro jejich automatické zkopírování do webového projektu, existuje soubor postbuild.py. Pro zjednodušení překladu zde existují dávkové soubory pro systém Windows, které provedou překlad nebo odstranění již přeložených souborů.

/msvc Soubory pro Visual Studio 2017, které definují způsoby překladu, cesty k souborům atd.

/src Zdrojové soubory aplikace / modulu. Soubory jsou členěny do adresářů podle jejich zaměření. Více se lze dočíst v návrhu a implementaci modulu. Zdrojové kódy obsahují přes 14 tisíc řádků kódu.

/thesis Soubory potřebné k vytvoření textové části práce ve formátu LaTeX.

/video Adresář obsahující ukázky výstupu z modulu.