
**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**


# WEBOVÝ PROHLÍŽEČ PANORAMATICKÝCH SNÍMKŮ

**WEB-BASED PANORAMA IMAGE VIEWER**

**BAKALÁŘSKÁ PRÁCE**
**BACHELOR'S THESIS**


**AUTOR PRÁCE** **TOMÁŠ SLUNSKÝ**
**AUTHOR**

**VEDOUCÍ PRÁCE** **Doc. Ing. MARTIN ČADÍK, Ph.D.**
**SUPERVISOR**



**BRNO 2017**

*Zadání bakalářské práce/19133/2016/xsluns01*

**Vysoké učení technické v Brně - Fakulta informačních technologií**

Ústav počítačové grafiky a multimédií Akademický rok 2016/2017

# Zadání bakalářské práce

Řešitel: **Slunský Tomáš**
Obor: Informační technologie
Téma: **Webový prohlížeč panoramatických snímků**
**Web-Based Panorama Image Viewer**
Kategorie: Web

Pokyny:

1. Proveďte rešerši existujících projektů pro prezentaci sférických panoramat a videí ve webovém prohlížeči.
2. Seznamte se s technologiemi, které budete potřebovat pro vývoj, zejména s HTML5.
3. Navrhněte architekturu a uživatelské rozhraní prohlížeče sférických panoramat a videí. Prohlížeč bude zobrazovat informace o orientaci a zorném poli kamery a další metadata.
4. Implementujte prohlížeč s využitím technologie HTML5. Soustřeďte se na efektivní zobrazení sférických videí jak v ekvirektangulárním módu, tak v módu rybího oka.
5. Proveďte uživatelské testování.
6. Dosažené výsledky prezentujte formou videa, plakátu, článku, apod.

Literatura:

- http://cadik.posvete.cz/

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

Technická zpráva bakalářské práce musí obsahovat formulaci cíle, charakteristiku současného stavu, teoretická a odborná východiska řešených problémů a specifikaci etap (20 až 30% 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: **Čadík Martin, doc. Ing., Ph.D.**, UPGM FIT VUT
Datum zadání: 1. listopadu 2016
Datum odevzdání: 17. května 2017

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

Účelem práce je navrhnout a realizovat prohlížeč sférických videí a panoramatických fotek. Důraz je kladen na zpracování videí v režimu rybího oka. Mezi další aspekty práce patří vylepšení prohlížeče grafické prvky, jako je kompas, nebo informace i zorném úhlu pozorovatele.



# Abstract

The purpose of this bachelors thesis is the implementation of panoramic viewer of videos and images. Thesis is focussing on processing videos, which are in fisheye mode. Between other aspects of this thesis belongs for instance the improvemenst of panoramic viewer with compass, or with informations about viewers angle.



# Klíčová slova

WebGL, HTML5, javascript, panoráma, sférické video, prohlížeč



# Keywords

WebGL, HTML5, javascript, panorama, spherical video, viewer



# Citace

SLUNSKÝ, Tomáš. *Webový prohlížeč panoramatických snímků*. Brno, 2017. Bakalářská práce. Vysoké učení technické v Brně, Fakulta informačních technologií. Vedoucí práce Čadík Martin.

# Webový prohlížeč panoramatických snímků

## Prohlášení

Prohlašuji, že jsem tuto bakalářskou práci vypracoval samostatně pod vedením pana X... Další informace mi poskytli... Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

.....
Tomáš Slunský
10. května 2017

## Poděkování

Rád bych poděkoval Martinu Čadíkovi za jeho cenné rady, vedení při tvorbě bakalářské práce a jeho celkový přístup. Dále bych rád poděkoval své rodině a blízkým za jejich trpělivost, toleranci a pochopení.

# Obsah

# Seznam obrázků

# Kapitola 1

# Úvod

V současné době se technologie ve všech odvětvích stále posouvá a není tomu jinak ani u prohlížení fotografií a videí. V současné době i takto nahrané informace pomocí digitálních zařízení je trend dále posouvat a zvyšovat tak zážitek ze zaznamenané události. Díky tomuto trendu si už nevystačíme s "klasickými" přehrávači, popř. prohlížeči videí, nebo fotografií. Díky sférickým 360 stupňovým kamerám, je dnes možné zaznamenat video o srovnatelné velikosti jako u dnes běžného telefonu, ale s mnohem větším objemem informací, které se ale nedají srozumitelně přehrát v klasických přehrávačích. Ruku v ruce se sdílením takových dát, je velice výhodné takový prohlížeč nabídnout jako webovou verzi, aby i jiní uživatelé mohli svá takto natočená videa přehrávat popř. sdílet. Důkazem toho je rozmach multimediálního obsahu na internetu, kdy samotný jazyk HTML pro prezentaci webových stránek dříve nenabízel přímou podporu videí, což se změnilo příchodem nové verze HTML5. Některé portály jako např. Youtube již dnes začíná nasazovat do svého prohlížení videí jeho rozšířené sférické verze, což tedy potvrzuje tento trend. Tato práce má za cíl takové prohlížení posunout ještě dál.

Následující práce se věnuje implementaci webového prohlížeče panoramatických snímku a videí v různých módech, ve kterých je video interpretováno. Cílem je navrhnout a zrealizovat řešení, aby uživatel po natočení videa dále nepotřeboval k přehrání specifický program. Zvláště v případě sférického videa v režimu rybího oka, kdy uživatel nejprve musí video překonvertovat do ekvidistantního zobrazení, aby jej mohl sféricky prohlížet. Díky řešení v následujících kapitolách tato nutnost odpadá.

Další inovací v prohlížení videí je přidání některých dat, které v běžném prohlížení nejsou k dispozici. Např. informace o světových stranách ve vztahu k obrázku, či videu popřípadě údaje o velikosti zorného pole v daném kontextu prohlížení až po dodatečná metadata v panoramatických obrázcích, které jsou vhodné např. k tomu, aby autor obrázku mohl popsat jeho zajímavé částí popřípadě blíže popsat zachycenou scénu.

Samotná práce je členěna do šesti částí. V kapitole Webové technologie prohlížeče se budu věnovat nastínění všech použitých technologií nutných pro pochopení problematiky, se kterými se bude pracovat v následujících kapitolách. O návrhu řešení pojednává kapitola Návrh klíčových částí prohlížeče, na kterou navazuje implementační část Implementace panoramatického prohlížeče. Otestování správného návrhu a implementace se zabývá kapitola Testování. V poslední části práce Závěr jsou prezentovány dosažené výsledky a možnosti dalšího rozšíření.

# Kapitola 2

# Webové technologie prohlížeče

Následující kapitola pojednává o technologiích, které samotný prohlížeč používá a jsou tedy nezbytné v následujících kapitolách, zejména v části Implementace panoramatického prohlížeče, kde se s nimi přímo pracuje.

## 2.1 HTML5

Jazyk HTML (HyperText Markup Language) je značkovací jazyk určený pro popis webových stránek. Vychází z univerzálního značkovacího jazyka SGML (Standard Generalized Markup Language). V současné době aktuální verzi jazyka HTML je jeho již pátá verze - HTML5. Nová verze HTML přináší zásadní vylepšení, nové funkce a možnosti, které jsou nezbytné pro návrh a implementaci webového prohlížeče.

Pro návrh samotného programu je nezbytná podpora multimédií – tedy audio a video, v neposlední řádě také plátno canvas sloužící pro práci s grafikou.

### 2.1.1 Video

Dříve nebylo možné do webových stránek vložit video tak jako dnes. K tomuto účelu bylo využíváno různých zásuvných modulů třetích stran a do webových stránek se video vkládalo např. jako objekt. Nejvíce rozšířeným přehrávačem videí a tedy jakási náhrada za podporu videí, kterou tehdy HTML nemělo, se v širším spektru stal Adobe Flash, který funkci přehrávače plní v menší míře až doposud, avšak je již zastíněn efektivnějším řešením, a to právě HTML5.

Nový prvek v HTML5 vytvořený k tomuto účelu je `<video>`. Byl navržen tak, aby mohl být použit bez detekčních skriptů na stránce. V elementu videa je možné nastavit více souborů s videem a dle podpory si daný prohlížeč vybere jim podporované video. V případě, že by prohlížeč prvek videa nepodporoval, bude jej ignorovat. Nastavení více zdrojů videa s odlišnými kodeky lze pomocí elementu `<source>` uvnitř páru elementů `<video>...</video>`. Jakmile prohlížeč narazí na `<video>`, podívá se, zdali je přítomen atribut `src`, v opačném případě začne procházet jeden element `<source>` po druhém a bude hledat právě takový, který umí přehrát.

Aby bylo možné ovládat video dynamicky pomocí kláves nebo myší, bude nutné využít DOM (Document Object Model). Jedná se o stromovou strukturu dokumentu, kterou si prohlížeč sestavuje po načtení webové stránky. Všechny elementy webové stránky jsou interpretovány v DOM jako objekt. Některé značky se v DOM vytvoří, aniž by byly ve

zdrojovém kódu zapsány. Obecně platí, že pomocí objektového modelu je díky javascriptu možné tuto stromovou strukturu dále upravovat a rozšiřovat.

#### 2.1.1.1 Stavy načítání a přehrávání videa

Další velice důležitou částí je načítáni videa a jeho ověření, zdali je již video připraveno k přehrání. K ověření dostupnosti videa lze použít jeden ze síťových stavů elementu pomocí stavového atributu `networkState`, popřípadě přímo zjišťovat připravenost videa pomocí atributu `readyState` [6], vracející jeden z následujících stavů, podle kterých se můžeme dále při přehrávání řídit a přizpůsobit tomu běh programu. Jednotlivé konstanty nabývají hodnot od `0` do `4` a dle hodnot rozlišujeme:

- `HAVE_NOTHING` (ekvivalentní hodnotě 0)
  - nastane v situaci, kdy zdrojové video není dostupné, nebo nejsou dostupná žádná data pro aktuální přehrávanou pozici
  - koresponduje také s návratovou hodnotou síťové metody `networkState()` v případě, když její návratová hodnota je rovna `0`, což odpovídá konstantě `NETWORK_EMPTY`.
- `HAVE_METADATA`
  - základní data o videu se podařilo získat a zdroj je tedy považován jako dostupný. V tomto stavu, ale ještě není dostatek dat, aby bylo možné začít s přehráváním. V aktuálním stavu jsou k dispozici jen data popisující přehrávány subjekt. Jsou načteny údaje videa o jeho délce, šířce, dekódovaní apod. Dochází také k vyvolání události `loadedmetadata`.
- `HAVE_CURRENT_DATA`
  - data jsou pro bezprostřední začátek přehrávání již připravena, ale video ještě není načteno dál za tuto pozici. Přehrávání se může dostat do stavu `HAVE_METADATA`, nebo se také může stát, že následující data videa již nejsou k dispozici.
- `HAVE_FUTURE_DATA`
  - data pro přehrávání jsou připravena pro současnou i nadcházející pozici. V případě, že by bylo video dosáhlo takového stavu poprvé, bude vyvolána událost `canplay`.
- `HAVE_ENOUGH_DATA`
  - data připravena pro aktuální a nadcházející pozice pro plynulé přehrání. Dochází k vyvolání události `canplaythrough`.

Mimo stavy načtení videa jsou tu i události indikující, co přesně se právě děje s videem po jeho načtení. Tyto stavy se hodí k dotazování elementu `<video>` pomocí DOMu např. pro tvorbu vlastního specifického ovládání videa, nebo pro zahrnutí interakce s myší.

- `playing` - video se aktuálně přehrává, atribut `paused` je nastaven na `false`
- `waiting` - přehrávání videa je pozastaveno, protože následující data videa nejsou k dispozici, webový prohlížeč ale očekává, že v vzápětí dostane
- `ended` - přehrávání videa doběhlo nakonec
- `canplaythrough` - tato událost říká, že je možné video přehrát až do konce bez nutnosti video zastavit k dalšímu načítání. Tohoto stavu může dosáhnout jen v případě, že stavový atribut `readyState` je roven `HAVE_ENOUGH_DATA`.

#### 2.1.1.2 Atributy videa

Důležité atributy pro manipulaci s videem a nastavením.

**src** - jedná se o atribut, do kterého se definuje zdrojové adresa videa jako URL, které se má zobrazit. Používá se zejména v situaci, kdy je k dispozici právě jedna verze videa.

**autoplay** - jedná se atribut typu `bool`. V případě, že je `true`, spustí se přehrávání média automaticky jakmile je vše připraveno. Uvedení názvu atributu do elementu samotného je již dostačující informace o tom, co se bude dít s videem. Tedy bude tato hodnota typu `bool` považována jako `true`, jinak `false`.

**poster** - Slouží k vystihnutí obsahu daného videa. Ještě než začne samotné přehrávání, je možné jako první snímek videa zvolit obrázek zadaný cestou. V případe, že tento atribut nebude vyplněn, HTML automaticky zvolí jeden z prvních neprázdných rámců.

**loop** - jedná se atribut typu `bool`, který začne po dosažení konce videa s přehráváním videa opět od začátku. Tato operace se provádí v nekonečné smyčce.

**width** - šířka přehrávače v pixelech.

**height** - výška přehrávače v pixelech.

**paused** - atribut videa typu `bool`. Indikuje, zdali se video přehrává či nikoliv.

**controls** - jedná se opět o atribut typu `bool`, který říká, aby prohlížeč použil vlastní ovládací prvky pro video. Ovládací prvky se dají také vytvořit a přizpůsobit velice snadno díky DOMu.

**preload** - tímto říkáme, jakou část videa by měl webový prohlížeč načíst ihned po načtení stránky. Tento atribut nabývá jednou ze tří hodnot:

- `none` - nebude načítat nic, výhodné v případě, kdy je potřeba minimalizovat vytížení pásma
- `metadata` - načte pouze metadata daného videa - základní údaje o jeho délce apod.
- `auto` - sám zvolí, co přesně udělá

#### 2.1.1.3 Metody videa

Jelikož pomocí DOM je možné upravovat vlastnosti elementu `<video>`, tak jsou tedy nutné metody k ovlivnění jeho atributů, aby bylo možné s přehráváním videa manipulovat.

`play()` - metoda sloužící k nastavení atributu `paused` na `false`, pokud video již skončilo, začne je přehrávat od začátku.

`pause()` - metoda sloužící k nastavení atributu `paused` na `true`, tím dojde k pozastavení videa

`canplaytype()` - metoda sloužící k ověření, zdali webový prohlížeč dokáže video daného typu přehrát. Metoda vrací jednu z následujících hodnot:

- `""` - pokud by metoda vrátila prázdný řetězec, video prohlížeč nedokáže přehrát.
- `maybe` - prohlížeč si není zcela jist, zdali dokáže formát přehrát.
- `probably` - daný formát videa dokáže prohlížeč přehrát s vysokou pravděpodobností.

`load()` - všechny data videa se načtou znovu, čímž dojde ke zrušení všech akcí a současných dat, které byly doposud staženy a načteny. Poté se vše zavolá a načte znovu následujícím způsobem:

1. Nejprve dojde k inicializaci, kdy `readyState` je nastaven na `HAVE_NOTHING` a nastaví i příslušný síťový atribut `networkState` na `0`, `seeking` je nastaven na `false`, `paused` je nastaven na `true`, vše ostatní je prázdné, nebo nula
2. vybere se zdroj z atributu `src` nebo `<source>`, dále je vyvolána událost `loadstart` a dochází ke stažení metadat videa
3. jakmile jsou metadata stažena, nastaví se zakladní vlastnosti videa - šířka, výška, délka a `readyState` je nastaven na `HAVE_METADATA` a spolu s tím je vyvolána událost `loadedmetadata`
4. Jakmile je `readyState` větší nebo roven `HAVE_FUTURE_DATA` je vyvolána událost `canplay` a `loadeddata`
5. dojde ke spuštění přehrávání. Je vyvolána událost `play` a `playing`, atribut `paused` nastaven na `false`

### 2.1.2 Plátno

Jak již bylo avizováno výše, plátno, neboli element `<canvas>` slouží v HTML5 pro vykreslení grafů, herní grafiky, obrazů bitmap apod. Díky elementu `<canvas>` lze vykreslovat i náročnější grafické objekty za pomocí WebGl. V HTML5 slouží tedy především k vykreslení 2D prvků pomocí Javascriptu. Plátno je párový element, přesto se mezi párové tágy zdánlivě nic nevykresluje, celý obsah je skryt. Takový obsah se nazývá tzv. *fallback content* a je zobrazen v případě chyby, nebo prohlížečům, které tento element HTML5 nepodporují.

Plátno je bezpochyby nejdůležitější částí pro realizaci celé práce. Bude pro vyobrazení používat především `<video>`, z kterého bude číst data a využije tedy element video jak zdroj. Díky DOM pak dokážeme s objektem snadno manipulovat. Díky elementu `<canvas>` jsme tedy schopni získat kontext Javascriptového API - WebGl a díky němu schopni pracovat na úrovni grafické karty.

#### 2.1.2.1 Atributy plátna

Canvas má pouze dva atributy a těmi jsou `width` a `height` pro nastaveni šířky a výšky plátna. Je možné nastavit výšku a šířku pomocí kaskádových stylů, avšak tato možnost není doporučována, protože by mohlo dojít k nežádoucí deformaci obsahu. Při změně velikosti bitmapy pomocí atributů dojde pouze ke změně velikosti dané bitmapy, změna kaskádovými styly ale změní velikost obsahu celé bitmapy a tím tedy dojde ke zkreslení.

#### 2.1.2.2 Použití a práce s plátnem

Použit plátno lze pomocí rozhraní `HTMLCanvasElement`, které slouží pro přístup k jeho vlastnostem a metodám. Přístup k objektu `HTMLCanvasElement` lze pomoci standardní javascriptové metody `getElementById()` v případě, že je v tágu `<canvas>` nastaven atribut `id`, nebo lze objekt získat pomocí metody `querySelector("canvas")`.

Po získání objektu plátna lze dále s elementem manipulovat pomocí dvou metod (počet volání dané metody vrátí pokaždé ten stejný objekt):

- `toDataURL([type[,quality]])` [6]
  - vrací URL adresu aktuálního obrázku v elementu `<canvas>`. Metoda může a nemusí mít nějaké argumenty. První argument, v případě že je zadán, rozhoduje jaký bude navrácen výsledný typ obrázku. Pokud nezadáme žádný argument, tak jako výchozí formát obrázku bude automaticky zvolen *image/png*. Výsledná adresa URL bude vrácena jako řetězec.

- `getContext(contextId[, ... ])`
  - metoda vrací referenci na daný objekt aplikačního rozhraní, díky kterému bude možné kreslit na plátno. Pokud by kontext pro daný argument nebyl podporován prohlížečem, návratová hodnota bude `null`. Typ aplikačního rozhraní je vybrán na základě argumentu metody:
    - `"2d"` - vrací objektové rozhraní `CanvasRenderingContext2D`, kterí slouží pro kreslení grafických primitiv.
    - `"webgl"` - pokud tuto funkci prohlížeč podporuje, bude navrácen objekt `WebGLRenderingContext`.

### 2.1.3 Vektory

SVG (Scalable Vector Graphics) - škálovatelná vektorová grafika je již zažitý standard. Doposud ale nebylo možné kód `svg` vkládat přímo do HTML. Změna přišla až s HTML5. SVG vychází z jazyka XML (e**X**tensible **M**arkup **L**anguage)[1]. Mezi jeho nesporné výhody patří velikost, přenositelnost a nezávislost na rozlišení aj. Do HTML se vkládá pomocí elementu `<svg>`, který má zpravidla dále zanořené elementy, které reflektují jeho obsah. Mezi takové vnořené elementy patří take tág `<path>`.

#### 2.1.3.1 Element path

Generický element `<path>` slouží ve vektorové grafice k vytvoření křivky, která může být dále vyplněna barvou, nebo může sloužit k vytvoření ořezové cesty apod. Dále může mít nastaveny atributy specifikující vykreslování obrysů, nastavení stylů a tříd, nebo nastavení kalkulace délky cest. Mezi takové atributy patří:

**d** - tento atribut nese informace o cestě, resp. data k vyrekslení obrysu, který poté můžeme vyplnit konkrétní barvou, popř. jej využít k dalším podobným účelům víz výše. Data atributu se skládají z následujících částí:

- `M`/`m` (*moveto*) její syntaxe: `M x y` nebo `m x y` - realizuje posun na danou dvojice souřadnic `(x, y)` bez kreslení. Velké písmeno `M` značí, že souřadnice budou absolutní. Malé písmeno `m` indikuje, že budou souřadnice relativní[2]. Tuto vlastnost je velice výhodné používat pro všechny cesty, jinak by vždy následující čára začínala na konci té předchozí.

- `Z`/`z` (*closepath*) - tento příkaz je bez parametrů a slouží k uzavření cest, tedy spojení posledního bodu s prvním bodem vykreslovaných cest.

- `L`/`l` (*lineto*) - tento příkaz již kreslí čáru, jeho syntaxe je stejná jako u příkazu *moveto* tj `L/l x,y`

- `H`/`h` (*horizontal lineto*) - kreslí horizontální čáru z bodu (cp**x**, cp**y**) to (**x**, cp**y**), argument příkazu je pouze souřadnice `x`

- `V`/`v` (*vertical lineto*) - kreslí vertikální čáru z bodu (cp**x**, cp**y**) to (cp**x**, **y**), argument příkazu je pouze souřadnice `y`

- `A`/`a` (*elliptical arc*) - parametry: *(rx ry x-axis-rotation large-arc-flag sweep-flag x y)* kde `rx` značí první poloměr elipsy, `ry` druhý poloměr, `x-axis-rotation` rotace kolem osy x, příznak `large-arc-flag` značí, zdali bude oblouk krátký (hodnota `0`) popř. dlouhý (hodnota `1`), dále argument `sweep-flag` příznak značící oblouk proti směru/po směru (`0`/`1`), `x` a `y` souřadnice koncového bodu

**fill** - vyplní křivku zadanou argumentem `d` libovolnou barvou

[1] obecný značkovací jazyk pro serializaci dat.
[2] Velikost písmen má stejný význam u všech příkazů.

## 2.2 GLSL

**GLSL** (Open**GL** **S**hading **L**anguage) - jedná se o jazyk pro psaní shaderů [3]. Jazyk vychází z jazyka C a také proto je mu svojí syntaxí velice podobný. Základní konstrukce každého programu je v zásadě úplně stejná jako v jazyku C, každý program musí obsahovat hlavní funkcí `main()` bez jakýkoliv parametrů a návratových hodnot. V hlavní funkci programu jsou vyhrazeny proměnné se speciálním významem, jako např. `gl_Position`, kterou nastavujeme pozici vrcholu ve vertex shaderu, nebo `gl_FragColor` pro nastavení barvy ve fragment shaderu.

### 2.2.1 Vertext shader

Jedná se o program v jazyce GLSL, který je pro prováděn pro každý vstupní vrchol zadané geometrie.

Každá souřadnice vrcholu je dále přepočítána do tzv. *clipspace*, což je souřadný systém pro vykreslování do plátna. V případě, že by některé souřadnice byly mimo hranice souřadného systému, budou oříznuty a vykreslovat se nebudou. Hranice *clipspace* jsou definovány pro každou ze souřadnic `x`,`y`, `z` intervalem `<-1,1>`. Vertex shader může sám ovlivnit souřadnice vrcholů, textur popř. normálových vektorů a barev. Při zpracování vrcholů není známa topologie zpracovávaného objektu, protože nejsou dostupné informace o sousedících vrcholech.

Poslední operací vertex shaderu před samotným vykreslením je převod do NDC[4] [5]. Rozdílné operační systémy mají rozdílné způsoby reprezentace grafických objektů. Zabránit těmto rozdílným intepretacím v praxi znamená, že se souřadnice `x`, `y` a `z` vydělí homogenní souřadnici.

### 2.2.2 Fragment shader

Je program - označovaný taktéž jako *pixel shader*, který je volán pro každý pixel vykreslované scény. Jeho úkolem je výpočet barvy každého pixelu a díky tomu lze s takto získanou barvou dále pracovat. Barva je získána jako čtyřsložkový vektor, první tří složky tvoří paletu RGB, poslední čtvrtá složka tvoří $\alpha$ kanál [5]. V případě vykreslování složitějších objektů umožňuje GLSL automaticky body geometrie interpolovat, tedy automaticky získat pixel dané geometrie, aniž by musel být každý samostatně definován. Ukázka interpolace je demonstrována na obrázku 2.1.

[3] Jedná se o programy řídící vykreslování na grafické kartě.
[4] Normalized Device Coordinates.
[5] Složka pixelu udávající jeho průhlednost.

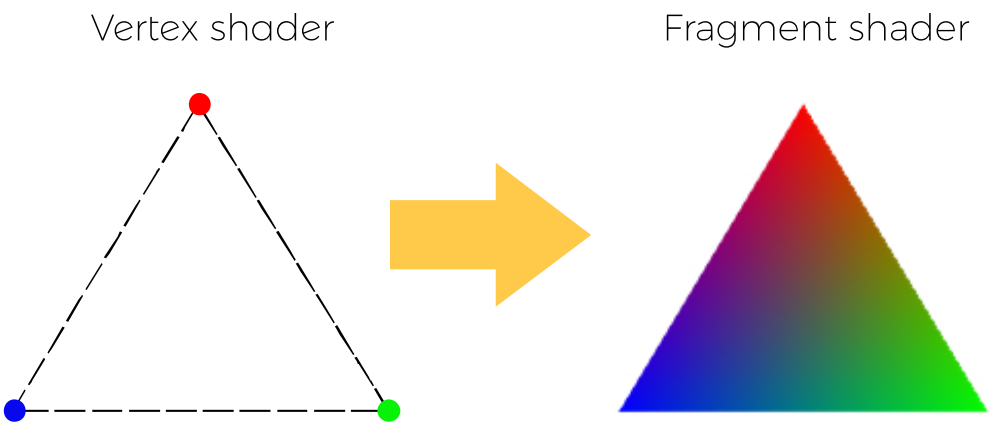
### 2.2.3 Typy proměnných

Předávání hodnot v globálních proměnných může probíhat za pomocí třech datových typu, těmi jsou:

**`uniform`** - uchovává hodnoty, které jsou konstantní po celou dobu běhu programu. Využívá se např. pro data transformačních matic apod.

**`attribute`** - jedná se o vstupní data vertex shaderu, která mění svoje hodnoty pro každý vrchol.

**`varying`** - tento datový typ slouží k předávání hodnot z vertex shaderu do fragment shaderu. Pomocí něj se provádí výše avizovaná interpolace, viz. obr. 2.1




Obrázek 2.1: Ukázka interpolace hodnot mezi vertex a fragment shaderem.

## 2.3 WebGL

WebGL (**Web G**raphics **L**ibrary) je javascriptové aplikační rozhraní, které slouží pro nativní vykreslování 2D a 3D interaktivní grafiky přímo z webového prohlížeče [2]. K samotnému běhu WebGL není potřeba žádných dodatečných knihoven, nebo zásuvných modulů, protože je v tomto ohledu využívána přímo grafická karta. Program WebGL se v zásadě skládá z více jazyků, přičemž každý mů svůj vlastní účel a je nutné je používat dohromady. Jedním z nich je javascript, druhý je jazyk GLSL. V jazyce GLSL jsou napsány dva programy - vertex shader a fragment shader. Oba programy kompiluje ovladač grafické karty do kódu, který je poté vykonáván přímo na grafické kartě. Vertex shader i fragment shader mohou být předávány jako řetězce, takže manipulace s nimi je díky tomu značně zjednodušena. Velkou výhodou WebGL je integrace s DOM tudíž od automatické správy pamětí přes zpracování událostí až po jednoduché načítání bitmapy přímo v prostředí prohlížeče apod.. Zpracování všech dat i samotný běh programu je z hlediska implementace velice podobný konečnému automatu.

K samotnému vykreslení je použit již výše zmíněný element HTML5 `<canvas>`, ze kterého se získá WebGL kontext.

V současné době je oficiálně vydána verze WebGL 1.0, která je založeno na OpenGL ES 2.0. Doposud se jedná o jedinou stabilní verzi. OpenGL ES[6] je zjednodušená verze OpenGL, která eliminuje většinu vykreslovacích pipeline. Tato verze je především určena pro menší vestavěné systémy. Dalším rozšířením WebGL bude verze 2.0, která se stále nachází ve vývoji. Bude založena na OpenGL ES 3.0 přinášející velká vylepšení v zobrazování 3D grafiky - zejména textur, akceleraci vizuálních efektů, novou verzí jazyka GLSL s plnou podporu desetinných a celočíselných operací apod.

### 2.3.1 Vykreslování primitiv

K vykreslení nějakého grafického prvku jsou potřeba v zásadě jen dvě věci - vrcholy přepočítané do tzv. *clipspace* a jemu přiřazené odpovídající barvy. Jak pro vrcholy tak pro barvy slouží jiný GLSL shader. K vyobrazení objektu používá WebGL tří základní primitiva - vykreslení samotných bodů, čar nebo trojúhelníku. Vše funguje tak, že zadaný bod, kterému budeme chtít přiřadit barvu, se načte do odpovídající speciální globální proměnné ve vertex shaderu, tam se daná hodnota přepočítá do intervalu `<-1,1>` pro všechny jeho osy a vzniklá hodnota se uloží na grafické kartě. Počet bodu záleží na primitivu, které vykreslujeme. Pokud se tedy jedná o bod, stačí pouze jeden bod ve vertex shaderu, pokud se jedna o čáru tak dvě popř. trojúhelník, tak trojice bodů. GPU[7] poté zjistí, které pixely korespondují s vykreslovaným primitivem a pro každý takový pixel zavolá fragment shader.

### 2.3.2 Vyrovnávací paměť

Buffery resp. vyrovnávací paměti jsou rychle přístupné paměti na grafické kartě. V bufferech jsou uloženy potřebné data k vykreslování grafických objektů jako např. body geometrie, normály, mapování textur apod. Data jsou poté snadno dále předávány GLSL shaderům, které s nimi pracují.

[6] OpenGL for Embedded Systems.
[7] Graphic Processing Unit - grafický procesor

### 2.3.3 Projekce

Některé frameworky[8] založené na WebGL zaměňují projekci za kameru, ta bohužel jako taková ve WebGL neexistuje. Jedná se pouze o způsob vykreslení grafické informace, ke které jsou využívány právě projekční zobrazení. WebGL podporuje dva základní typy projekce - ortografickou a perspektivní.

Ortografická projekce využívá skutečné velikosti objektu. Scéna je reprezentována jako pravoúhlý hranol.

Perspektivní projekce zkresluje velikost objektů způsobem, jak je tomu v reálném světě. Čím je objekt dál od pozorovatele, tím se bude jevit jako menší, protože jeho pozice je středem této projekce. Dalším příkladem bychom mohli uvést koleje, které se v dálce jeví, jakoby se sbíhaly v jednu jedinou apod.

Pokud bychom chtěl vykreslit nějaký 3D objekt, bude potřeba mít v bufferech informace o vrcholech, normálách, textuře apod. Tyto data bychom zároveň museli neustále aktualizovat a nahrávat zpět do paměti, aby bylo možné s daným 3D objektem jakkoli manipulovat, posouvat apod., aby nedocházelo k neustálému přepisování hodnot v paměti na GPU. Veškeré změny zrealizujeme transformačními maticemi, kterými se bude geometrie objektu v paměti násobit, aniž by data v paměti byla jakkoli změněna.

### 2.3.4 Homogenní souřadnice

Homogenní souřadnice je tvořena čtyřmi prvky - `x`, `y`, `z` a `w`. První tři složky tvoří souřadnice euklidovského prostoru, poslední prvek `w` je perspektivní složka a spolu se souřadnicemi `x`, `y`, `z` tvoří projekční prostor.[1]

Perspektivním dělením - vydělením souřadnic `(x, y, z)` perspektivní složkou `w`, lze získat normalizované hodnoty bodu zpět do trojrozměrného souřadného systému. Platí zde ale, že perspektivní složka nesmí být nulová.

Samotná perspektivní složka funguje v homogenních souřadnicích jako změna měřítka daného bodu geometrie.

Jednou z dalších vlastností homogenní souřadnice je ta, že umožňuje definovat body do nekonečna, resp. nekonečnou délku vektoru, což ve standardním trojrozměrném prostoru možné není. Tato situace nastává v případě, kdy je složka rovna nule.

### 2.3.5 Model matrix

Jedná se o matici, kterou držíme pro každý vykreslovaný objekt. Stará se o úpravu tvaru, natočení a posuvu objektu.

### 2.3.6 View matrix

Díky teto transformační matici realizujeme již avizovanou “roli kamery” , kdy simulujeme vzájemná vztah scény a pozorovatele.

### 2.3.7 Perspective matrix

Stará se o perspektivní zkreslení scény. Tato transformace rozhoduje o tom, jak velké zorné pole bude vykresleno a mapováno na obrazovku monitoru.

[8] Rámcová softwarová struktura usnadňující řešení dané problematiky

### 2.3.8 Vytvoření a běh programu

Po vytvoření kontextu `webgl` je již možné využívat funkce pro práci s WebGL. Funkce by se daly rozdělit na několik částí, nejprve ty, které pracují s shadery, dále ty které vytváří program, v neposlední řadě funkce pro prací s buffery a vykreslením dat. Všechny funkce WebGL pracují nad vytvořeným kontextem plátna[9]. Práce s funkcemi WebGL je velice podobné konečnému automatu, v zásadě jde pouze o volání funkcí s danými parametry, které mění kontext pro specifické účely. WebGL disponuje celou řadou funkcí, které využijeme dále v implementaci, v této částí se ale zaměříme na vykreslování, které je třeba více rozebrat, než se vrhneme na samotnou implementaci.

Po nahrání dat do bufferů se vykreslení uložených data realizuje funkcemi:

**`gl.drawArrays(mode, first, count)`** - vykresluje vstupní data bufferu. Parametr `mode` určuje, jaké grafické primitivum budem použito pro vykreslení, `first` určuje odkud se data zažnou upracovávat, `count` specifikuje počet vykreslovaných hodnot

**`gl.drawElements(mode, count, type, offset)`** - stejně jako u `drawArrays()` určuje vykreslované grafické primitivum, to stejné se týká i parametru `count`, `type` definuje typ dat indexů a `offset` odkud začínáme zpracovávat indexy.

Hlavní rozdíl mezi funkcemi je ten, že `drawArrays()` vykresluje přímo data geometrie tak, jak byla spočítána a vložena do vertex bufferu. Samotné body prostoru, tak jak jsou definovány, jsou uvedeny vícekrát pro každé sousedící primitivum.

Funkce `drawElements()` pracuje s takovými daty odlišným způsobem. Vedle bufferů potřebných k vykreslení geometrie a mapování textury, je zaveden nový buffer tzv. index buffer, který disponuje odkazy na jednotlivé body do bufferu geometrie a zavádí sdílení bodů, čímž data geometrie zmenší.

[9] Vytvořený WebGL kontext budeme označovat jako `gl`

# Kapitola 3

# Návrh klíčových částí prohlížeče

Kapitola nastiňuje, jakým způsobem jsou řešeny nejzásadnější problémy implementace prohlížeče. Snaží se navrhnout ty nejefektivnější způsoby řešení daných problémů a v případě, že tomu tak v konkrétním případě není, snaží se tyto kroky návrhu jasně odůvodnit, proč jsou daným způsobem takto řešeny.

## 3.1 Geometrie

Geometrie je potřebná k mapování textury, a to tak, aby daná textura reflektovala scénu co nejvěrněji. Kdybychom pro texturu nevytvářeli geometrii, na kterou se bude mapovat a jednoduše data namapovali do libovolné 2D roviny, dojde k významnému zkreslení pořizované scény.

K uložení dat geometrie budeme využívat klasické pole, kde každý bod bude tvořen třemi rozměry v kartézské soustavě souřadnic, tedy formátu `[x1,y1,z1, x2,y2,z2, ...]`, které utváří jednotlivé body v 3D prostoru. Body pak mezi sebou tvoří prostor, na který budou texturovací data mapovaná. Ve WebGL jsme schopni vykreslit jednotlivé body, čáry – tedy spojnice jednotlivých bodů, nebo trojúhelníky. V našem případě bude potřeba vykreslit texturu jako plochu, na kterou se bude vše mapovat. Pro tento případ se ve WebGL používá vykreslení pomocí trojúhelníku, jako základní vykreslované primitivum plochy.

Geometrickým tělesem pro mapování bude zvolena koule, protože scéna resp. pořízená data, která budeme zpracovávat jsou zachycena ve sférickém modu a mapováním na kouli je dosažena nejvěrnější reprezentace scény.

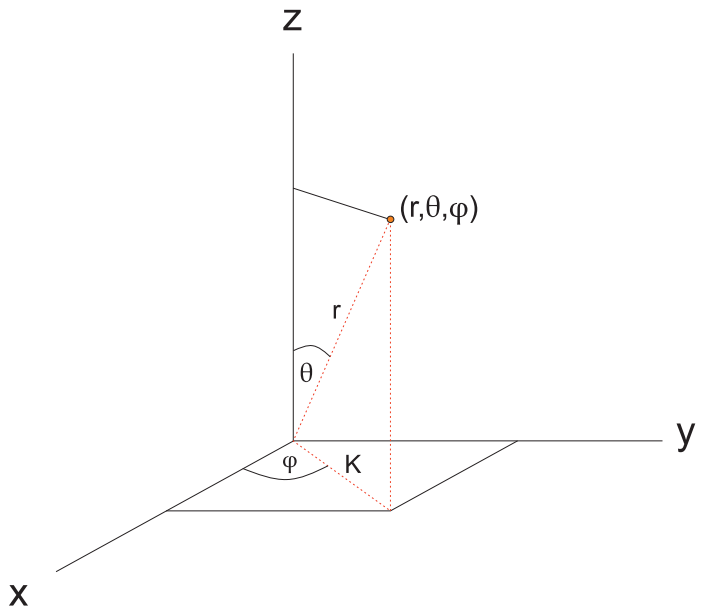
Kouli si rozdělíme na poledníky a rovnoběžky. Poledníkem je v tomto případě myšlena spojnice severního a jižního pólu, kde úhel z počátku kartézské soustavy souřadnic mezi jednotlivými poledníky nabývá hodnot od 0 - 360°. Rovnoběžkou je myšlena kružnice rovnoběžná se vzájemně sousedícími a svírající úhel s počátkem souřadnic od 0 - 180°.

Je třeba tedy nejprve stanovit počet poledníků a rovnoběžek a z takových hodnot bude dále vypočítán úhel. Na základě úhlu budeme schopni přesně definovat bod v trojrozměrné rovině. Čím větší počet rovnoběžek a poledníků, tím bude geometrie věrněji aproximovat kouli - zvýší se hustota bodů v bufferech a tím bude i samotné vykreslení náročnější. Proto je ideální kompromis mezi jemností bodů a rychlosti načítání s téměř stejným výsledkem. Tato vlastnost by se nám hodila hlavně tedy v případech, kdy bychom měli vykreslit rozmanitý geometrický objekt se spoustou detailních sekvencí. V našem případě tento rozdíl tedy nepůjde okem rozeznat, takže skrze efektivitu je výhodné spíše snížit počet rovnoběžek a poledníků, čímž dosáhneme pozitivního efektu na samotnou rychlost programu.

### 3.1.1 Sférický souřadný systém

Každá rovnoběžka má po svém obvodu průsečíky s jednotlivými poledníky, tyto průsečíky jsou naše body geometrie, které budou nahrány do bufferů. Jelikož každá rovnoběžka nabývá úhlu od 0 do 180°resp. $0-\pi$, tak tento úhel můžeme podělit počtem rovnoběžek a vynásobit aktuální rovnoběžkou, čímž získáme svíraný úhel s počátkem souřadnic. Na obrázku 3.1 označujeme tento úhel jako $\theta$. Provedením této operaci pro všechny horizontální obvodové kružnice, vypočítáme všechny svírané úhly s počátkem souřadnic.

Dále je nutné vypočítat svíraný úhel všech poledníků - vertikálních čar svírajících jej s počátkem souřadnic. Na obrázku 3.1 je vyznačen jako $\varphi$. Jak jsem již nastínil výše, $\varphi$ nabývá hodnot od 0 do 360°resp. $0-2\pi$. Vytyčený úhel mezi jednotlivými čarami jsme schopni spočítat tak, že jej podělíme počtem vertikálních čar a vynásobíme aktuálním poledníkem. Takto získáme rovnoměrné rozložení bodů na celé rovnoběžce. Provedením operace pro všechny rovnoběžky již bude geometrie koule z hlediska úhlů kompletní.




Obrázek 3.1: Výpočet jednotlivých bodů geometrie

Na základě vypočtených úhlů, je možné již spočítat jednotlivé složky kartézské soustavy souřadnic `[x,y,z]` každému bodu, čímž dojde ke kompletní aproximaci tělesa.

K výpočtu využijeme goniometrické funkce a vztahy v pravoúhlém trojúhelníku. Jelikož $\cos(\varphi)$ je dána poměrem délky přilehlé odvěsny a přeponou [4], tak souřadnici `x` definujeme vtahem:

$$\cos(\varphi) = \frac{x}{K} \Rightarrow x = \cos(\varphi) \cdot K \tag{3.1}$$

Ze vzniklého vztahu jasně vidíme, že je třeba dopočítat přeponu $K$ v trojúhelníku o stranách $|xKy|$, kterou ale odvodíme vztahem odvěsny $K$ v trojúhelníku $|zrK|$. Mezi odvěsnou `K`[1] a protilehlou přeponou $r$ platí, že délka odvěsny protilehlé k úhlu $\varphi$ a délka přepony [4] mají vztah:

$$\sin(\theta) = \frac{K}{r} \Rightarrow K = \sin(\theta) \cdot r \tag{3.2}$$

Všechny neznáme máme již dopočítané a můžeme dosadit do první rovnice, čímž dostáváme výsledný vztah převedené souřadnice z kartézské soustavy souřadnic do sférického souřadného systému:

$$x = \sin(\theta) \cdot \cos(\varphi) \cdot r \tag{3.3}$$

Následující souřadnici `y` vyjádříme obdobným vztahem jako u předchozí souřadnice, ale s tím rozdílem, že protilehlá odvěsna bude k úhlu $\varphi$ a přeponou bude strana `K`:

$$\sin(\varphi) = \frac{y}{K} \Rightarrow y = \sin(\varphi) \cdot K$$

Opět máme v rovnici pro výpočet souřadnice `y` stranu `K`, kterou ale máme již vypočtenou ve vztahu 3.2, tudíž už jen tento vztah dosadíme a vznikne výsledná rovnice hledané souřadnice:

$$y = \sin(\varphi) \cdot \sin(\theta) \cdot r \tag{3.4}$$

Poslední souřadnice `z` je značně jednodušší, protože nám odpadá proměnná `K`. Vyjádření souřadnice `z` je principiálně úplně stejné, jako tomu bylo u souřadnice `x` - přilehlá odvěsna k úhlu $\varphi$ ku přeponě `r`. Výsledný vztah je tedy možné vyjádřit následovně:

$$\cos(\theta) = \frac{z}{r} \Rightarrow z = \cos(\theta) \cdot r \tag{3.5}$$

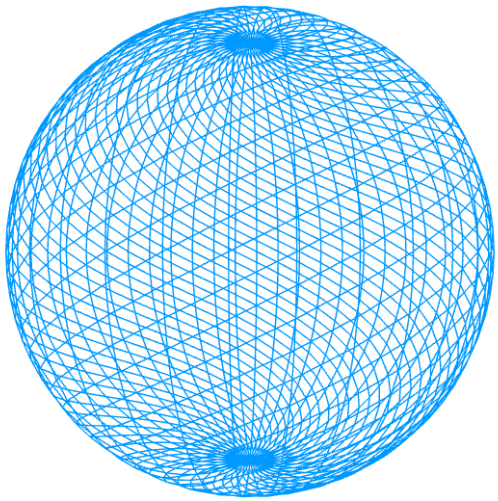
[1] Z obrázku 3.1 je patrné, že je přeponou v $\triangle xKy$, ale odvěsnou v $\triangle zrK$

### 3.1.2 Vrcholy

Jakmile máme vypočítány konkrétní složky trojrozměrného prostoru - tedy kompletně spočtený sférický souřadný systém, můžeme přistoupit ke kompletaci dat určených k nahrání do grafické karty. Těmito daty jsou vrcholy samotné geometrie, texturovací data a indexy.

Získání samotného vrcholu lze vynásobením složek `x`, `y` a `z` průměrem tělesa `r`. Tato operace musí být provedena u všech bodů koule.

Vzniklé těleso vyobrazené pomocí čar[2] je demonstrováno na obr. 3.2.



Obrázek 3.2: Koule vyobrazena pomocí čar

### 3.1.3 Textura

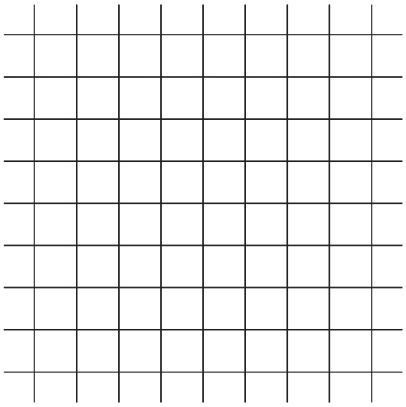
Data textury se počítají oproti vrcholům zcela odlišně. Všechny body textur jsou v současné stabilní verzi WebGL pouze dvojrozměrné, tudíž souřadnice `z` nebudeme potřebovat. Počet bodů textury bude přímo záviset na počtu rovnoběžek a poledníků.

Od zvoleného počtu rovnoběžek a poledníků se bude dále odvíjet počet bodů textury tak, že pro každou rovnoběžku se spočítají všechny poledníky. Celkový počet bodů textury bude tedy: *počet_rovnoběžek* · *počet_poledníků*.

[2]Použito grafické primitivum `gl.LINES`.

Pro lepší představu mapovaní textury můžeme jak poledníky, tak rovnoběžky znázornit ve dvourozměrném prostoru, kde jednotlivé průsečíky tvoří body textury, přičemž horizontální čáry prezentují souřadnici s označením `u`. Vertikální čáry zase souřadnici `v`.

Znázornění demonstruje obrázek 3.3. Souřadnici `u` spočteme vydělením $k$-tého poledníku celkovým počtem poledníků. Souřadnici `v` vydělením $k$-té rovnoběžky celkovým počtem rovnoběžek.



Obrázek 3.3: Mapování textury

### Výpočet konečné barvy ve fragment shaderu

Výsledná interpolovaná barva textury se ve fragment shaderu vypočítá vztahem:

$$gl\_FragColor = t(s, v_t)$$

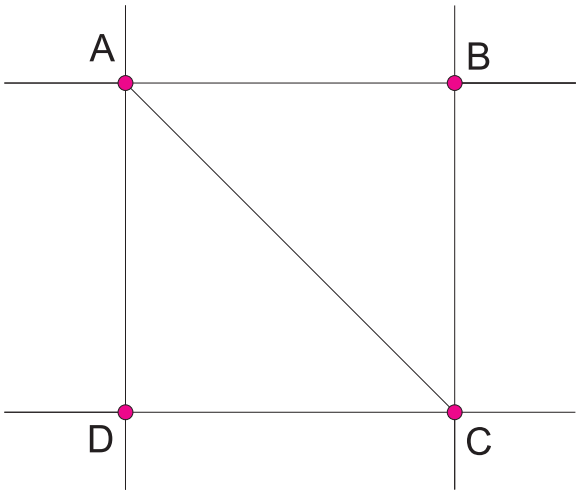
kde `t` je texturovací funkce, která načte data fragmentu, `s` označuje vzorkovací proměnnou a $v_t$ interpolovaný bod z vertex shaderu.

## 3.1.4 Indexy

Zavádění indexů se vyplatí tehdy, pokud chceme vykreslování objektu zefektivnit z hlediska vykreslování jednotlivých bodů. V případě, že se při vykreslování nepoužívá index buffer, tak se pro každé vykreslované primitivum načítá adekvátní počet vrcholů. Potíž nastává v situaci, kdy je vrcholy společný. Dojde k tomu, že souřadnice vrcholu jednoho bodu bude v paměti vícekrát a bude volán pro každé primitivum zvlášť. Díky indexům je pro danou geometrii zmíněn sdílený bod právě jednou, v index bufferu je tento bod zmíněn vícekrát, ale odpovídá mu právě jeden bod geometrie. Díky tomuto odkazování se nám sníží počet vykreslovaných bodů. Indexování se používá spolu s funkcí `drawElements()` a pro účely implementace bude využita pro zrychlení načítání ekvidistantního zobrazení textur.

#### 3.1.4.1 Návrh index bufferu

Rovnoběžky a poledníky využijeme i v této části. Jak lze z obrázku 3.3 pozorovat, horizontální a vertikální čáry tvoří mezi sebou čtvercovou síť, kdy čtverec můžeme snadno interpretovat pomocí trojúhelníku jako grafického vykreslujícího primitiva víz obr. 3.4.



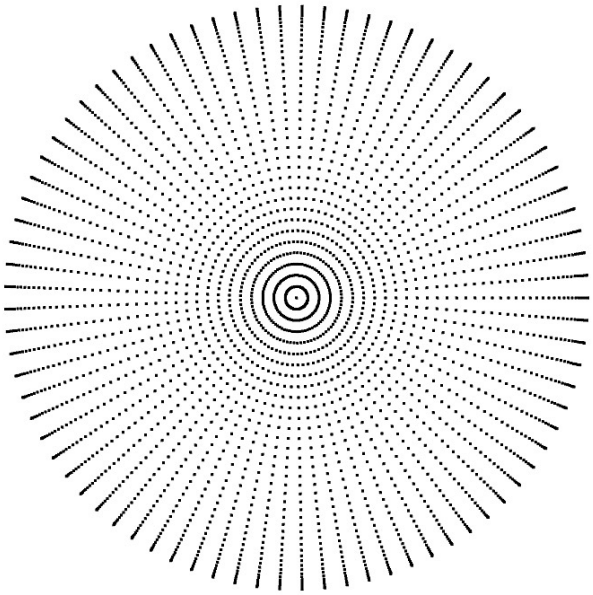

Obrázek 3.4: Implementace indexů

Je tedy jasně patrné, že čtverec budeme vykreslovat trojúhelníky $ABC$ a $ADC$. Horizontální přímky protínající body $A$, $B$ a $D$, $C$ jsou rovnoběžky, po kterých se budeme při výpočtu bodů posouvat. Naproti tomu body $A$,$D$ a $B$, $C$ prochází poledníky, díky kterým budeme při výpočtu realizovat vertikální posuv.

Samotné zpracování indexů tedy probíhá tak, že pro každou rovnoběžku procházíme její průsečíky s danými poledníky, do index bufferu uložíme pro každé grafické primitivum trojici bodů, což tedy v praxi znamená šestici bodů pro každý čtverec sítě, kde se budou vždy dva body opakovat.

## 3.2 Korekce textur

Základní mapování textury lze pozorovat na obr. 3.3, takové mapování lze využít u equirectangulárního prohlížení, kde samotná vstupní textura má jíž vše potřebné a není tedy již potřeba nic upravovat - samotná ekvidistantní projekce je převod tělesa trojrozměrného zobrazení do 2D, proto přímo koreluje s daty textury. Pro mód rybího oka je situace složitější, protože vstupní textura není v equirectangulárním modu a naším úkolem je udělat korekci textury tak, aby se textura namapovala na kouli co nejpřesněji.



Obrázek 3.5: Textura vycházející z předpočítaných dat. Bez korekce.

## 3.3 Korekce pro rybí oko

Korekci využijeme především k tomu, aby se nám textura mapovala přesněji v oblasti švu, tedy přesně v polovině tělesa. Důvodem je ten, že pokud bychom korekci textury neudělali, okraje by byly rozostřené a v oblasti švu data hůře čitelná. Samotná korekce nemá na funkci prohlížení žádný vliv. S korekcí se mapuje textura daleko přesněji, resp. okraje hemisféry. Na obrázku 3.5 si lze povšimnout ukázky dat textury ještě před provedením korekce krajních bodů.

Pro výpočet korekce můžeme využít jako počáteční hodnoty souřadnic již ty spočítané pro normály[3] popř. vrcholy geometrie.

Již před-počítané hodnoty je ale nutné ještě upravit, protože jsou počítány pro trojrozměrný prostor a my pro texturu potřebujeme pouze dva rozměry. Úpravou je v tomto smyslu myšleno to, že velikost poloměru kružnice se bude měnit rovnoměrně, tedy hlavně u bodů, které odpovídají bodům v oblasti švu, v tomto místě dochází k největšímu zhuštění bodů, což má za následek již avizované neúměrné roztažení a deformaci textury.

[3] V našem případě hodnota souřadnice vrcholu podělená průměrem tělesa.

Je-li souřadnice ***y*** definována rovnicí 3.4 a souřadnice ***z*** vztahem 3.5, tak korekce textury bude realizována vynásobením vertikální a horizontální souřadnice funkcí ***arccos(y)***. Každá hemisféra koule bude mít odlišnou modifikaci souřadnice `x`, resp. výpočet každé hemisféry probíhá v jiném kvadrantu kartézské soustavy souřadnic. V opačném případě bychom nemohli mapovat obě sféry, protože by se nám spojily do jedné tj. body by se překrývaly - důvod je zřejmý, data z kterých korekce vychází jsou počítány pro celou kouli a nerespektováním změny souřadnice, by se bod zobrazil až "za" vyobrazeným bodem na obrazovce[4].

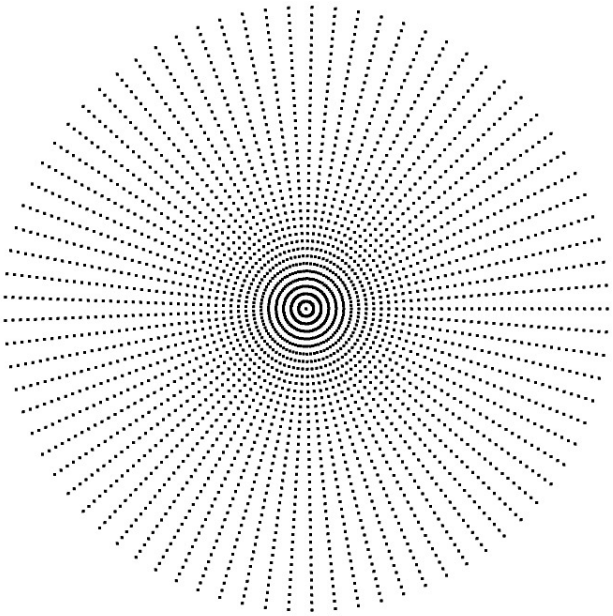
Horizontální a vertikální souřadnici `u` a `v` spočítáme následujícím způsobem:

- Pro levou hemisféru bude platit následující vztah:

$$u = x \cdot \frac{\arccos(y)}{r} \wedge v = z \cdot \frac{\arccos(y)}{r}$$

- Pravá hemisféra je difinována vztahem:

$$u = -x \cdot \frac{\arccos(y)}{r} \wedge v = z \cdot \frac{\arccos(y)}{r}$$



Obrázek 3.6: Textura s korekcí krajních bodů.

Proměnná `r` označuje poloměr koule, resp. kružnice ve 2D. Obrázek 3.6 demonstruje již aplikovanou korekci vstupních dat textury z obr. 3.5 avizovanou výše.

[4] Z pohledu na těleso ve 3D.

## 3.4 Návrh projekce

Součástí již avizované projekce v předchozí kapitole, jsou matice, které modelovou, zobrazovací a projekční matici utváří. Mezi ně patří např. rotační matice po osách `x`,`y` a `z`, matice posuvu, změny měřítka apod. Díky těmto maticím jsme schopni projekci zrealizovat - kdy v perspektivním pojetí vyvářímе frustrum[5], resp. komolý jehlan, kdy stěny tvoří lichoběžník a obě podstavy čtverec popř. obdélník. Veškeré vykreslovaná data, která neleží v komolém jehlanu jsou ořezány a nebudou vykreslena.

### 3.4.1 Modelová matice

Sem patři rotační matice všech os souřadného systému, dále matice pro změnu měřítka souřadnic a posuvná matice objektu.

Cílem modelové matice je tedy transformovat body modelu ze středu souřadného systému, vůči kterému byly body geometrie modelu napočítány, do zobrazitelného prostoru, se kterým budeme dále pracovat. Cílem modelové matice je posunout a pootočit zobrazovaným objektem do výchozí pozice promítání. Otáčení a další operace v reálném čase jsou doménou zobrazovací matice. Začneme s matici pro změnu měřítka modelu:

$$S = \begin{pmatrix} \mathbf{x} & 0 & 0 & 0 \\ 0 & \mathbf{y} & 0 & 0 \\ 0 & 0 & \mathbf{z} & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

Matice rotace kolem osy x:

$$R_x = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos(\alpha) & -\sin(\alpha) & 0 \\ 0 & \sin(\alpha) & \cos(\alpha) & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

Matice rotace kolem osy y:

$$R_y = \begin{pmatrix} \cos(\alpha) & 0 & \sin(\alpha) & 0 \\ 0 & 1 & 0 & 0 \\ -\sin(\alpha) & 0 & \cos(\alpha) & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

Matice rotace kolem osy z:

$$R_z = \begin{pmatrix} \cos(\alpha) & -\sin(\alpha) & 0 & 0 \\ \sin(\alpha) & \cos(\alpha) & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

[5] V ortografickém zobrazení by to byl kvádr/krychle.

Matice pro posun objektu po osách `x`,`y` a `z`. Přímo vychází z identické matice[6]:

$$T = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ \mathbf{x} & \mathbf{y} & \mathbf{z} & 1 \end{pmatrix}$$

Výsledná modelová matice `M` je tedy dána vztahem:

$$M = T \cdot R_x \cdot R_y \cdot R_z \cdot S$$

Při násobení původní matice rotační maticemi hrají velkou roli také znaménka. Jelikož v implementační části budeme používat interakce s myší a tím měnit náhled pozorovatele resp. kamery na náš model, bude třeba zvolit směr posuvu zorného pole kamery.

Vlastnosti prohození znamének využijeme právě u rotace $R_x$ a $R_y$, které reprezentují pohyb myši.

### 3.4.2 Zobrazovací matice

Jedná se o matici, která kontroluje způsob, jakým se na objekt díváme. V našem konkrétním případě budeme view matrix používat k napodobeni pohybu kamery snímající objekt, jak je tomu v reálném světě.

Prostor objektu, který jsme pomocí modelové matice transformovali do zobrazitelného prostoru musíme nyní převést do prostoru, který bude relativní vzhledem k pohledu pozorovatele.

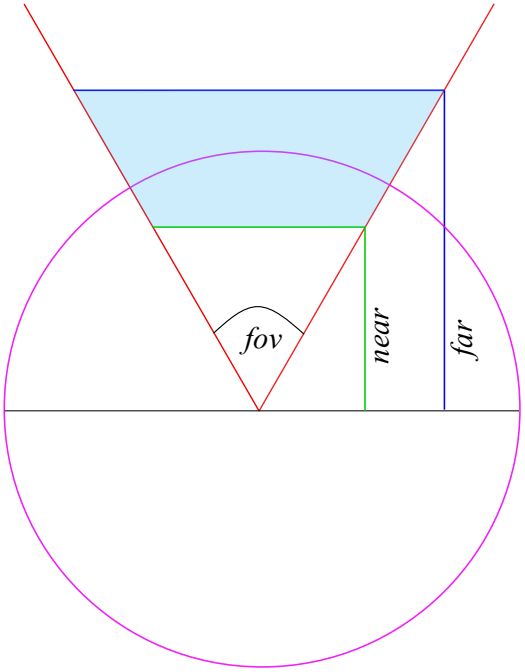
Takového efektu docílíme tak, že složku o kterou se chceme posunout, vynásobíme opačnou hodnotou modelu ve scéně. Výsledný vztah pro výpočet zobrazovací matice tedy je:

$$V = T \cdot R_x \cdot R_y \cdot R_z$$

[6] Na hlavní diagonále jsou jedničky, všude jinde nuly.

### 3.4.3 Perspektivní matice

Poslední částí projekce je perspektivní matice, díky které budeme realizovat zkreslení scény. Mezi vstupní parametry patří úhel, neboli zorné pole. Čím by bylo zorné pole blíže hodnotě $\pi$ resp. `180°`, tím by se předmět v zorném poli jevil dál od pozorovatele. Hodnota `180°` je krajní hodnota zorného pole. Pokud bychom se jí příliš přibližovali, tak by se nám projekční prostor více a více zmenšoval, až by se objekt jevil nekonečně daleko a tedy se vůbec nezobrazil, protože v přímém úhlu je projekční plocha nulová. Naopak zmenšováním hodnoty, by se objekt jevil zase blíže. Zorné pole budeme označovat jako ***`fov`***. Bližší hranici projekčního prostoru jako ***`near`***, vzdálenější hranici jako ***`far`***.




Obrázek 3.7: Demonstrace perspektivního zobrazení.

Perspektivu demonstruje obrázek 3.7. Modře vyznačená plocha reprezentuje již avizovaný projekční prostor, který je definován zorným úhlem a hranicemi. Fialová kružnice značí data textury, která se mapují na vypočítanou geometrii.

Jednotlivé vstupní argumenty potřebné pro výpočet perspektivní matice zobrazení:

**fov** - pole, které definuje zorný úhel

**aspect** - poměr šířky a výšky scény

**near** - reprezentuje rovinu, která ořeže tu část modelu, která je příliš blízko pozorovateli

**far** - reprezentuje rovinu, která ořeže tu část modelu, která je příliš daleko pozorovateli

$$P = \begin{pmatrix} \dfrac{1}{aspect \cdot \tan\left(\dfrac{fov}{2}\right)} & 0 & 0 & 0 \\ 0 & \dfrac{1}{\tan\left(\dfrac{fov}{2}\right)} & 0 & 0 \\ 0 & 0 & \left(\dfrac{near + far}{near - far}\right) & -1 \\ 0 & 0 & \left(\dfrac{near \cdot far \cdot 2}{near - far}\right) & 1 \end{pmatrix}$$

## 3.5 Výpočet projekce

Konečný vztah pro výpočet projekce scény $P_s$ bude dán pouhým vynásobením spočtených matic v následujícím pořadí:

$$P_s = P \cdot V \cdot M$$

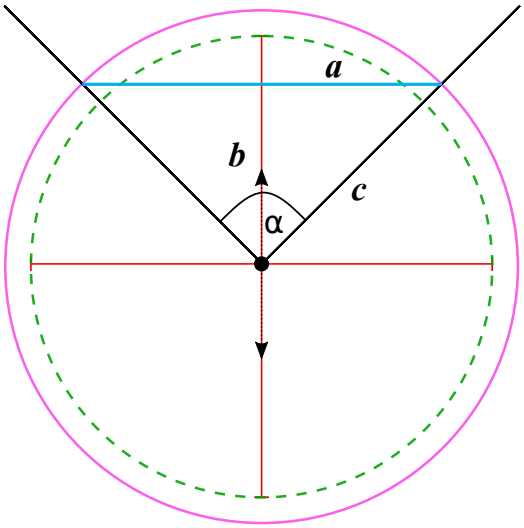
vypočítaná projekce se poté násobí bodem geometrie $G$. Výsledná pozice bodu ve vertex shaderu má tedy vztah:

$$gl\_Position = P_s \cdot G$$

## 3.6 Zorný úhel

Zorný úhel je grafický prvek, který udává informace o aktuálním zorném poli zobrazovací matice. Tento úhel se mění na základě přibližování, nebi oddalování scény od pozorovatele.

### 3.6.1 Vztah perspektivy a pozorovatele




Obrázek 3.8: Textura s korekcí krajních bodů.

### 3.6.2 Návrh prvku

K vyobrazení prvku budeme potřebovat vektor svg zobrazující přímý úhel. Reprezentován bude tvarem polokoule, u které se využije vlastnosti tahu. Ve vektorové grafice se jedná o nastavení atributu `stroke-width`, který bude reprezentovat zorný úhel. Důvodem, přoč pro vyobrazení samotné není využita samotná křivka vyplněná barvou je ten, že pokud se přenastaví úhel zobrazení, tak se křivka zdeformuje tak, že z ní není zjevný zorný úhel.

Vstupními argumenty je pozice prvku pomoci ve dvojrozměrném prostoru `(x,y)` a průměr kruhu. Dále je potřeba přepočítat tyto souřadnice dvojrozměrného prostoru na sférické souřadnice. K samotnému výpočtu dostačuje Pythagorova věta, kdy pomocí bodu `(x,y)` jsou známy hodnoty odvěsen a je možní spočítat přeponu. Takto odvodíme vztahy pro `x` a `y`.

$$x = r \cdot \cos(\theta)$$

$$y = r \cdot \sin(\theta)$$

## 3.7 Kompas

Dalším rozšířením prohlížení jsou údaje o světových stranách díky kompasu. Střelka kompasu se bude pohybovat k severu díky vstupním argumentům, které budou předány přímo do programu. Ukazatel severu se bude měnit na základě rozdílu vůči vstupní pozici, která definuje pozici severu ve scéně. Samotna realizace střelky bude díky obrázku `svg` a natáčení pomocí změně kaskádových stylů. Změna zobrazení bude udávána ve stupních vůči původní pozici. Samotná transformace zobrazení, resp. otočení střelky budeme realizovat pomocí vlastnosti kaskádových stylů `transform: rotate(X deg)` kde ***X*** je posuv ve stupních.

### 3.7.1 Odchylka střelky

Korekce kompasu bude daná úhlem, který svírá střed zorného pole a vstupní statickou pozici zadanou argumentem programu, která po celou dobu běhu programu nad danými daty zůstane neměnná. Tedy samotná korekce bude reagovat pouze na změnu horizontální osy.

### 3.7.2 Výpočet pozice kompasu v plátně

Výsledné zobrazení kompasu resp. jeho pozice je přímo závislá na rozměrech elementu `<canvas>`, je to z toho důvodu, že prvek plátno překrývá. Samotný výpočet a následné korekce změnou šířky plátna, popř. využití režimu celé obraziocky vyžaduje přepočítání pozice, která je definována vztahem:

$$(Canvas_{width} - Compass_{width}) \cdot x$$

$$(Canvas_{height} - Compass_{height}) \cdot y$$

kde $Canvas_{width}$ a $Canvas_{height}$ jsou vstupní argumenty kompasu udávající velikost prvku. Naproti tomu $x$ a $y$ souřadnice nabývající hodnot od $0 - 1$, která definuje pozici v plátně nezávisle na rozlišení.

# Kapitola 4

# Implementace panoramatického prohlížeče

V této kapitole rozeberu implementaci programu spolu s problémy, které bylo nutné v průběhu implementace řešit. Program se skládá z jazyka HTML5, ve kterém je nastíněna výchozí kostra, nad kterou pracuje veškerá funkcionalita. Jádro je napsáno v jazyce Javascript bez použití frameworků a shadery, které jsou implementovány v jazyce GLSL.

## 4.1 Stavy a data programu

Jelikož samotná implementace ve WebGL vyžaduje k realizaci problematiky více jazyků, vzniká v programu velké množství proměnných, které je velice vhodné určitým způsobem pro vyšší přehlednost v kódu sdružovat. Program jako takový je napsán procedurálním stylem, ale sám využívá některé objektové vlastnosti. Mezi ně patří právě sdružování proměnných, které jsou hierarchicky uspořádány do objektu.

Každá důležitá část implementace má svůj vlastní objekt proměnných, které se k dané problematice vztahují, včetně všech nutných interních stavů.

### 4.1.1 Myš a klávesnice

Veškeré interakce s myší se z hlediska datové částí promítají do objektové proměnné `MOUSE`, která udržuje data o pohybu pozorovatele ve scéně po vertikální a horizontální ose otáčení, včetně hloubky posuvu - tedy přibližování scény pomocí kolečka. Většina udržovaných dat v proměnné `MOUSE` pochází z objektu `MouseEvent` popř. `WheelEvent`. Objekty poté díky Javascriptové metodě `addEventListener()` uchovávají aktuální hodnoty vstupu odkud se čtou. Dalším objektem je klávesnice realizována pomocí objektové proměnné `KEYBOARD`, která udržuje pouze základní informace, zdali je aktivní popř. jaká citlivost pohybu pozorovatele ve scéně pomocí klávesnice má být. Veškeré interakce s klávesnici jsou dále realizovány pomocí čtení z objektu `KeyboardEvent`, do které se načítají údaje o stisku kláves, tudíž na rozdíl od myši není třeba tyto data uchovávat, protože klávesnice je závislá pouze na aktuálním stavu.

### 4.1.2 Data programu a nastavení

Obdobným způsobem, jak je již nastíněno výše, jsou uchovávány proměnné a nastavení samotného programu. Programová objektová proměnná `PROG` uchovává data jako jsou buffery programu, transformační matice, nad kterými probíhá projekce a v neposlední řadě atributy shaderů, díky kterým jsou data matic, bufferů apod. nahrávány na grafickou kartu.

Odděleně od programu je vytvořen objekt geometrie, který počítá a vytváří všechna data spojených s geometrií a mapováním textury pro nahrání na GPU. Objekt vytváří funkce `createSphereGeometry()`.

Objektová proměnná `SETTINGS` uchovává informace o nastavení módu programu resp. formát dat, které bude program vyobrazovat. Dále informace pro nastavení a inicializaci grafických prvků.

## 4.2 Běh programu

Ihned po načtení HTML kostry programu je zavolána hlavní funkce programu `main()`, která je spuštěna na základě vyvolání události `onLoad` [1] .

Před samotným spuštěním programu je nutné provést některé inicializační operace - nastavení potřebných proměnných pro běh, ověření a ošetření.

Tuto roli obstarává funkce `initProgram()`. Je zde nutné ověřit, zdali se podařilo získat WebGL kontext z plátna `<canvas>`. Pokud by došlo k situaci, že prohlížeč nebude podporovat kontext WebGL, program se ukončí. Mezi další funkce inicializace patří, že na základě vstupu nastaví, jaký mód prohlížení bude zvolen, resp. na základě vstupních proměnných objektu `SETTINGS` se nakonfiguruje funkce `createSphereGeometry()`.

V případě, že inicializování proběhlo v pořádku, zavolá se funkce `setupProgram()`, kde dochází k sestavení všech částí programu a nahrání nejdůležitějších dat na grafickou kartu. Dochází zde také ke kompilaci obou shaderů a provázání s programem. Mezi další operace, patří naslouchání vstupních informací z myši, kolečka klávesnice aj. pomocí metody `addEventListener()`.

Pokud `setupProgram()` proběhne v pořádku a data jsou připravena, je možné nahrát také texturu - pro panoramata jsou textury nahrávány také z funkce `setupProgram()`, v případě videa je tato operace volána z funkce `render()`, důvodem je ten, že video potřebuje reflektovat změny v čase.

### 4.2.1 Shadery

Veškeré operace se shadery v zásadě probíhají ve funkci `setupProgram()`. Samotný kód vertex a fragment shaderu se předává jako textová řetězec, tudíž jej můžeme uložit do proměnné, externího souboru popř. využít html značky. V našem konkrétním případě se pro umístění shaderu využívá kód mezi HTML značkami. Důvodem je ten, že na zpracování programu to nemá vůbec žádný vliv a kód je daleko lépe editovatelný. Každý shader mÁ svoje specifické označení, pro fragment shader je nastaven v tágu `<script>` atribut `type` na `x-shader/x-fragment`, ve vertex shaderu potom `x-shader/x-vertex`.

Výpočty transformačních matic se počítají v hlavním programu, ale přímé operace s nimi až v shaderech, konkrétně ve vertex shaderu. U projekce se všechny matice vynásobí aktuálním bodem geometrie, která je do vertex shaderu předána z bufferu, tato operace se opakuje pro každý bod geometrie modelu. Jelikož vertex shader slouží pro propojení

[1] Událost `onLoad` nastane ihned po načtení webové stránky.

webgl programu a obou shaderů, musí se předat do fragment shaderu textura, jejíž body se automaticky interpolují a předají fragment shaderu. K předání hodnoty slouží proměnná `varyingTexture`.

Fragment shader se provádí nad každým pixelem textury, tudíž nejprve načte předanou hodnotu `varyingTexture` z vertex shaderu a podle interpolace načte aktuální pixel vykreslující grafické primitivum, čímž dojde k vykreslení celého trojúhelníku. Jakmile je fragment shader hotov s daným primitivem, zavolá se vertex shader pro další body u bufferu a operace fragment shaderu se opakuje.

Kódy shaderů se nakonec musí propojit s WebGL, kde se jako první použije funkce `createShader()`, která shader vytvoří, poté funkcí `shaderSource()` načteme zdrojový kod daného shaderu a zkompilujeme pomocí `compileShader()`.

Načtené a zkompilované shadery nejsou v globální objektové proměnné uloženy, protože se jako takové v programu využívají jen ve funkci `setupProgram()`.

### 4.2.2 Program

Ze zkompilovaných shaderů se vytvoří program resp.se oba shadery provážou do jednoho WebGL programu, aby se mohlo nad nimi dále pracovat. Vytvoření programu se realizuje funkcí `createProgram()` a následně se přiloží oba shadery k programu pomocí funkce `attachShader()`. V další fázi se program slinkuje s `linkProgram()` a funkcí `useProgram()` aktivuje. Výsledný program spolu s přiloženými shadery reprezentuje globální objektová proměnná `PROG.program`.

### 4.2.3 Transformační matice

Modelová matice je z hlediska své implementace po celý běh programu stejná, je jí ale potřeba přenastavit pro každý mód prohlížení, protože jak ekvidistantní projekce, tak režim rybího oka mapuje texturu odlišným způsobem a je tedy nutné tomu přizpůsobit i modelovou matici.

V případě equirectangulárního prohlížení je vstup programu přímo mapován na kouli bez jakýkoliv úprav dat textury, protože vstupu odpovídá navrženy model. V módu pro rybí oko je nutné nad mapovanými daty sice provést korekci, ta samotná ale na úpravu modelové matice nemá vliv, hlavním důvodem je posun vstupní projekce rybího oka vůči té equirectangulární o 90°

U perspektivní matice je situace obdobná jako u modelové matice. V `setupProgram()` se pouze vytvoří a nahraje do shaderu, dále se v programu již nemění. Naproti tomu zobrazovací matice, která plni roli pozorovatele ve scéně resp. roli kamery, se pokaždé interakci se vstupním zařízením data mění a je proto nutné její data nahrát do shaderu znova.

V shaderu již je připravena uniformní proměnná určená pro data matic, tudíž nyní se program `PROG.program` prováže s těmito proměnnými v shaderech webgl funkcí `getUniformLocation()` a její reference se načte do objektové globální proměnné `PROG.matrices.<`*`model/view/projection`*`>.link`, kde ***model/view/projection*** reprezentují tří reference na modelovou, zobrazovací a projekční (perspektivní) matici.

### 4.2.4 Buffery a geometrie

Vytvoření bufferů ve webgl jak již bylo avizováno provedeme funkcí `createBuffer()`, poté se funkci `bindBuffer()` nastaví buffer jako aktivní a nahraje do něj data funkcí `bufferData()`. Data geometrie jsou načteny v objektu `GEOMETRY`, a ze kterého budeme předávat data bufferům. V naší implementaci jsou využívány tři buffery, a to vertex buffer pro data geometrie modelu, dále texturovací buffer a nakonec index buffer s indexy do pole vertex bufferu, ukazující na body geometrie. V případě módu rybího oka budeme využívat pouze první dva zmíněné. V případě equirectangulárního prohlížení využijeme data všech bufferů. Všechny data se mimo indexů nahrávají na grafickou kartu jako `Float32Array`.

Objekt `GEOMETRY` vrací funkce `createSphereGeometry()`. Vstupními argumenty geometrické funkce jsou počet poledníku a počet rovnoběžek, poloměr koule a příznak použití indexů. První tři se využijí k výpočtu souřadnic, poslední zdali zvolit indexování, či nikoliv. Samotný výpočet bodů probíhá ve dvojici cyklů `for`. První cyklus prochází rovnoběžky a počítá úhel $\theta$, vnitřní cyklus prochází poledníky a počítá úhel $\varphi$ nutný spolu s úhlem $\theta$ pro výpočet sférického souřadného sytému. Ve vnitřním cyklu vedle bodů geometrie dále počítají normály, textura a indexy.

Výpočet indexů je oddělen od hlavní výpočetní smyčky, protože musí mít již v poli spočítaná data bodů tj. musí vědět, na která data má odkazovat. Indexy se opět počítají dvojicí cyklů `for`, kdy vnější smyčka pouze prochází rovnoběžky.Vnitřní smyčka prochází nejen poledníky, ale počítá indexy na pozice bodů.

### 4.2.5 Atributy

Stejně jako data matic se musí i data bufferů propojit se shadery. To se provede funkcí `getAttribLocation()`, která vrací referenci na atribut. Reference na atribut bodů geometrie se načte do globální proměnné `PROG.attributes.vertex`, pro mapování textury do obdobné proměnné `PROG.attributes.texture`.

### 4.2.6 Textury

Implementace textur je zasazena jak do funkce `setupProgram()` tak do vykreslovací funkce `render()`, každá implementace v dané funkci má trochu odlišný charakter, který více rozeberu až v následující kapitole. Textura oproti programu, vyžaduje v našem konkrétním případě širší definici. Je třeba definovat mj. i chování textury při interpolaci.

Samotná textura se vytvoří funkcí `createTexture()`, která vrací referenci na objekt WebGL textury, s kterým se dále pracuje. Tento objekt si uložíme opět do globální objektové proměnné `PROG.texture`. Po zisku reference se textura, stejně u bufferů popř. programu, ještě aktivuje obdobnou funkcí `bindTexture()`.

Chování textury při zmenšení a zvětšení je konkretizováno funkcí `texParameteri()`, která je nastavena na `LINEAR`, protože v našem případě implementace dosahuje lepších výsledků než-li `NEAREST`.

## 4.3 Vykreslování a módy prohlížení

Nutnost uvádět texturu resp. její nastavení ve více funkcích pramení z toho důvodu, že je rozdíl ve vykreslované frekvenci mezi panoramatem a videem. Textura panoramatu je pro každý snímek stejná, potřebuje pouze aktualizovat pozici pozorovatele vůči textuře. V případě videa je situace komplikovanější, textura jako taková se pro každý snímek mění, zároveň ale musí být ještě promítnuta aktualizace pozice pozorovatele vůči měnící se textuře.

Situaci řeší funkce `updateTexture()`, která je aktivní pouze v režimu videa[2]. Funkce je volána již před začátkem vykreslování ve funkci `render()`. Nejprve zjišťuje, zdali je video již načteno pomocí ověření `video.readyState >= video.HAVE_CURRENT_DATA` a pokud ano, tak načte texturu webgl funkcí `texImage2D()`. Pokud by data načtena neměla a `readyState` byl menší než `video.HAVE_CURRENT_DATA`, tak vrátí `false` a zobrazí v plátně načítání.

Samotné vykreslování se provádí až ve funkci `render()`. Mimo vykreslení zde ještě dochází k aktualizaci pozice kamery pomocí webgl funkce `uniformMatrix4fv()` a aktivaci textury, se kterou se bude pracovat. Nejdůležitější částí funkce `render()` jsou funkce pro vykreslení `drawArrays()` a `drawElements()`. Na základě dat, které se budou vykreslovat, se zvolí, jakou funkcí se bude vykreslovat. V případě ekvidistantního módu jak pro panoramata, tak pro videa se použije funkce `drawElements()`. Důvodem je ten, že pomocí indexování je vykreslení ekvidistantních dat efektivnější. V případě rybího oka, kdyby se použilo `drawElements()`, tak by sice bylo možné data věrné interpretovat, ale došlo by k problému se sdílenými body na hranách při mapování textury, tudíž by nedošlo k tak přesnému vykreslení, jako s funkcí `drawArrays()`.

Animování scény zajišťuje volání `window.requestAnimationFrame()`. Jedná se o efektivnější řešení, než-li je tomu u funkce `setInterval`. Dokáže lepé přizpůsobit výkon při vykreslování, jako např. v situaci, kdy uživatel překlikne na jiné okno, funkce to automaticky rozporná a sníží frekvenci snímkovaní.

## 4.4 Ovládání myší

Myš je implementována trojicí funkcí, které fungují nad generovanými javascriptovými událostmi vstupu. Po kliknutí do plátna se myš aktivuje tím, že dojde k události `mouseDown`, kterou v programu zpracovává funkce `onDocumentMouseDown()`. Po stisku levého tlačítka myši nad plátnem, se snímání vstupu myši aktivuje z hlediska programu proměnnou `MOUSE.active` a dále uloží pozice kurzoru myši do globálních proměnných `MOUSE.down.x` a `MOUSE.down.y`. Po celou dobu tahu myši v plátně zůstává myš aktivní, dokud není vygenerována událost `mouseUp`, kterou obsluhuje programová funkce `onDocumentMouseUp()`. Událost `mouseUp` funkci myší přeruší tak, že nastaví `MOUSE.active` na `false`. Mezi událostí `mouseDown` a `mouseUp` se vykonává hlavní funkce pro práci s myši `onDocumentMouseMove()`. Je spouštěna událostí `mouseMove`, tedy pohybem myši po plátně. Funkce vychází z uložených hodnot pozice kurzoru již ve funkci `onDocumentMouseDown()`, aby zjistila odkud a kam se kurzor myši pohnul.

Pohyb kurzoru po stisku levého tlačítka myši lze tedy tahem měnit horizontální a vertikální pozici pozorovatele scény. Problém s tahem nastává nad elementy, které plátno překrývají. Proto se tato funkce ošetřuje již ve funkci `initProgram()`, kde se pomocí `addEventListener()` vyberou ty grafické elementy, které může myš překrývat, zbytek je

[2] Jak v módu rybího oka tak v ekvidistantním módu.

automaticky považován za nepřekryvný - myš nad těmito prvky nebude fungovat, protože `MOUSE.active` se nastaví na `false` a myš tím deaktivuje.

Hloubku zobrazení, neboli přiblížení scény a pozorovatele se provádí kolečkem. Událost `mouseWheel` zastřešuje funkce `onDocumentMouseWheel()`, která ji zpracovává. U kolečka nastává problém s interpretací prohlížečů, protože každý prohlížeč ukládá interakci s kolečkem do objektu dané události odlišně a také do jiných proměnných.

Na základě hodnot se dále určuje směr rotace kolečka a intenzita s jakou hodnoty přibližování a oddalování narůstají. Tento problém je ale vyřešen tak, že intenzitu nárůstu si řídí program sám dle vstupu, pouze se rozlišuje, zdali je hodnota kladná nebo záporná. Dále funkce ošetřuje krajní meze pro přibližování a oddalování scény, aby nedošlo k problému při prohlížení.

Ovládání myši dále rozšiřuje podpora pro dotyková zařízení. Události generované dotykem jsou velice podobné těm, které se generují pro myš. Díky tomu je možné stávající funkce pro zpracování vstupních dat myši rozšířit o novou vlastnost. Při dotyku obrazovky se vyvolá událost `touchstart`, kterou obsluhuje funkce `onDocumentMouseDown()`, tudíž navazuje přímo na její funkcionalitu. Obdobné je to u pohybu, kdy je generovaná událost `touchmove`, která rozšiřuje funkci `onDocumentMouseMove()`. Dokončení tahu na plátně zajišťuje událost `touchend`, která je obdobou uvolnění levého tlačítka myši, obě události obsluhuje funkce `onDocumentMouseUp()`.

## 4.5 Ovládání klávesnicí

Zpracování všech stisků kláves zajišťuje funkce `onDocumentKeyPress(event)`. K ověřování stisku programových kláves se využívá vlastnost `event.keyCode`, do kterého se snímá `ASCII`[3] kód stisknutého tlačítka.

Klávesnice ovládá jak pozorovatele scény, tak i přehrávání videa. Pro ovládání videa jsou vyhrazeny klávesy q pro spuštění videa, e pro zastavení a space kombinující funkci obou předešlých kláves.

Pohyb kamery lze ovládat tlačítky w, s, a a d. Stisk tlačítek w a s mění pozici kamery po vertikální ose, naproti tomu tlačítka a a d posun v horizontálních směrech. Všechny operace s klávesnicí lze i spolu se zapnutým režimem capslock.

Přiblížení a oddálení scény se provádí klávesami + a -.

[3] American Standard Code for Information Interchange.

## 4.6 Grafické rozhraní

Veškeré prvky, které nejsou vykreslovány skrze grafickou kartu, se v elementu jak již bylo nastíněno v předcházejících kapitolách, `<canvas>` nezobrazí z důvodu, že je vše skryto. Jen v případě, že by prohlížeč nepodporoval plátno, zobrazí co je v něm. Realizaci samotných prvků jako je kompas, nebo zorný úhel až po ovládací prvky videa, bude třeba realizovat za pomocí kaskádových stylů, kdy pomocí dynamického pozicování budeme všechny prvky zobrazovat překryvem plátna.

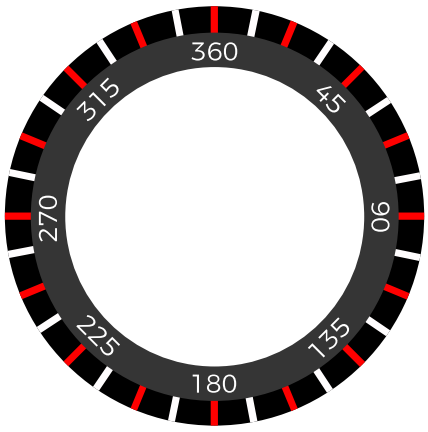
### Pozicování

Pozice všech grafikách prvků na plátně je třeba přizpůsobovat velikosti plátna, např. když dojde k přepnutí na režim celé obrazovky. Pozice jednotlivých prvků je zadána argumenty `position_x` a `position_y` nabývající hodnot od `0` do `1`. Takto zadané hodnoty jsou nezávislé na režimu zobrazení. Stačí tedy nastavit jen hodnoty kaskádových stylů `left` a `top` tak, že `div.style.left` bude roven rozdílu šířky plátna a šířky grafického prvku vynásobená argumentem `position_x`, pro vertikální umístění pak nastavení vlastnosti `div.style.top` jako rozdíl výšky plátna a grafického prvku, vynásobená hodnotu argumentu `position_y`. Při použití odskoku elementu od okraje obrazovky je pozice prvku nastavena na `absolute`, aby takto vypočítaná pozice měla smysl.

### 4.6.1 Kompas

Prvek překrývající plátno `<canvas>`. Vytvoření samotného kompasu realizuje funkce: `createCompass(width, height, position_x, position_y, north_position, angle)` . Argument `width` udává šířku kompasu, argument `height` zase jeho výšku. Vstupními metadaty je zadán kotevní bod severu. Díky interakci s myší - tedy násobení zobrazovací transformační maticí realizujeme pohyb pozorovatele, čímž se mění prostorový úhel, který ovlivňuje odchylku severu od původní zadané pozice.

Samotná konstrukce kompasu je zasazena do tágu `<div id="compass-box">`. Funkce, která nahraje data do kontejneru kompasu se nazývá `createCompass()`. Kompas se skládá ze dvou částí, první je statická část, kterou utváří element div s identifikátorem `compass`. Statická část, ukázaná na obrázku 4.1, nese informace o natočení ve stupních, na které bude ukazovat šipka střelky.




Obrázek 4.1: Kostra kompasu

Druhá část kompasu je střelka, která je vložena jako `<div id="compass_arrow">` do statické části, její podoba je demonstrována na obrázku 4.2. Střelka se otáčí pomoci transformace kaskádových stylů.


Obrázek 4.2: Střelka kompasu

### 4.6.2 Ovládací panel videa

Ovládací prvky videa jsou vytvářeny pomocí DOM a přidávány do kostry HTML. Celá kostra s atributem `id=viewer` obsahuje i plátno, které překrývají `<canvas>`. Při změně velikosti plátna musíme pozice prvků vždy přepočítat. Veškeré operace nad ovládacími prvky zastřešuje funkce `createVideoControlls()`, kde probíhá i přepočet pozice ovládacích prvků. Jedním ze vstupních argumentů funkce je objekt videa, který je již před-načten do objektové proměnné `SETTINGS.input.video`. Důvodem jen ten, že nad objektem funguje celá řada vyvolávaných událostí. Pomocí funkce `addEventListener()` zde dochází, podobně jako v inicializační části, k vykonávání všech základních operací s videem na základě událostí. Veškeré typy jsou vyvolány interakcí uživatele, řízeni časové osy a dotazování se na množství načtených dat videa, je generovány automaticky. Tlačítko `fullscreen` pro zobrazení videa na celou šířku obrazovky, potřebuje přizpůsobit plátno tak, aby nedošlo k deformaci dat, čehož dosáhneme zavoláním funkce `gl.viewport()`. Ještě je třeba ale uchovávat informace o původní velikosti plátna `<canvas>`, abychom se mohli z režimu celé obrazovky vrátit. Tyto data jsou uchovány rovněž v objektové proměnné.

Vedle přímých akcí kontrolovaných uživatelem, jsou tu ještě automatické operace prováděné nad videem. Jedná se o detekcí aktivního okna při přehrávání, kdy si program sám ověřuje pomocí funkce `addEventListener()` pracující nad objektem `window`, zdali aktuální okno prohlížeče je aktivní. Vše zprostředkovávají události `focus` a `blur`. Událost `blur` se spustí ve chvíli, kdy uživatel nemá zobrazené okno s panoramatickým prohlížečem, tato událost bude mít vliv na video jen v případě, kdy se bude video přehrávat a v takové situaci je i pozastaví, dokud se okno opět nezaktivní událostí `focus`.

### 4.6.3 Zorný úhel

Zorný úhel implementuje funkce `createFieldVision()`. Výpočet pozice včetně její změny pobíhá stejným způsobem jako u kompasu. HTML element div je v kostře umístěn pod `id` s názvem `field_vision`. Velikost zorného úhlu je propojen s kolečkem myši tak, že změny v zorném úhlu závisí především na přiblíženi popř. oddálení scény v proměnné `MOUSE.wheel.delta`.

<part:2:doplnit>

# Kapitola 5

# Testování

Výsledný program bylo třeba otestovat z hlediska kompatibilita webových prohlížečů s WebGL, nebo s HTML5 až po rozdílnou interpretaci některých vstupních událostí pro periferie počítače.

Pro testování byl využit jednoduchý nástroj frameworku `three.js`, který reflektuje aktuální rychlost překreslování scény grafickou kartou. Jedná se konkrétně o třídu `Stats()`, kterou volá funkce `DebugThreeStat()` uvedené v knihovně pro externí zdrojové kódy. Vedle měření snímkové frekvence dále disponuje měřením v milisekundách, kolik je zapotřebí času k vykreslení jednoho snímku. Dále velikost alokované paměti v řádech megabajtů.

## Vývojové rostředí a test výkonu

Program byl vyvíjen na platformě operačního systému Windows 8 s procesorem Intel Core i7 2.4 GHz a GPU Nvidia GT 630m 2GB. Rozlišení LCD displeje $1920 \cdot 1080$px.

Hlavním prohlížečem pro vývoj sloužil Google Chrome. V průběhu vývoje byl program také testován v prohlížečích Opera a Firefox. Ověření pak v konečné fázi vývoje programu ještě testováno na operačnímu systému Ubuntu.

Vstupní data, s kterými probíhaly veškeré testy, byly pořízeny sférickou kamerou Ricoh Theta S.

Průměrné naměřené hodnoty při přehrávání dvou minutového videa v režimu celé obrazovky[1] v různých prohlížečích:

| | **Firefox** | **Opera** | **Chrome** |
|---|---|---|---|
| Frekvence snímků videa | 28,5 fps | 13,1 fps | 40,3 fps |
| Doba načtení jednoho snímku | 35 ms | 76 ms | 24 ms |
| Využití paměti | 12 MB | 10 MB | 12 MB |

Testování vykreslování panoramat o rozlišení $1920 \cdot 1080$px v režimu celé obrazovky po dobu jedné minuty:

| | **Firefox** | **Opera** | **Chrome** |
|---|---|---|---|
| Frekvence snímků obrázku | 41,2 fps | 41.5 fps | 44 fps |
| Doba načtení jednoho snímku | 24,2 ms | 24,09 ms | 22,72 ms |
| Využití paměti | 10 MB | 10 MB | 10 MB |

[1] V režimu rybího oka.

Stejný test pro panorama a video, ale po vypnutí režimu celé obrazovky[2]:

| | **Firefox** | **Opera** | **Chrome** |
|---|---|---|---|
| Frekvence snímků videa | 49,6 fps | 17,3 fps | 59,9 fps |
| Frekvence snímků obrázku | 59,1 fps | 58,5 fps | 59,6 fps |

## Uživatelské testování

`<uziv:test>`

## Shrnutí testování

Jak lze vidět, ve všech testech Opera velice ztrácí, zejména v přehrávání videa, kdy video nebylo při přehrávaní vůbec plynulé, a to ani po vypnutí režimu celé obrazovky. Naproti tomu prohlížení sférického videa v prohlížeči Firefox bylo znatelně plynulejší a ve všech módech prohlížení si vedl obstojně. Nejlépe ale vše fungovalo v prohlížeči Chrome, kde bylo přehrávání a prohlížení ve všech módech vždy plynulé.

`<uziv:test:sourhn>`

[2]Zmenšené plátno mělo při testování velikost $1100 \cdot 700$px.

# Kapitola 6

# Závěr

## 6.1 zhodnocení dosažených výsledků s vyznačeným vlastním přínosem studenta

Účelem práce bylo implementovat prohlížeč ve WebGL, který je schopen přehrát sférická videa v různých režimech. Hlavně tedy v režimu, který by v jiných prohlížečích byl velice zkreslen, protože by nepodporoval režim rybího oka. Implementovaný prohlížeč je schopen tento režim zvládnout bez pomocí externích konvertorů popř. konvertorů, které dává k dispozici přímo výrobce 360 stupňové kamery. Díky programu tedy již externí konvertor není potřeba a výstup z kamery lze prohlížet okamžitě po natočení.

## 6.2 náměty(zdroje inspirace) vycházející ze zkušeností s řešeným projektem

## 6.3 návaznosti na právě dokončené projekty

## 6.4 zhodnocení z pohledu dalšího vývoje projektu

Vylepšení programu by mohlo přijít spolu s novou verzí WebGL, která v době implementace této práce ještě nebyla oficiálně vydána. Nová verze disponuje širší škálou možností a otevírá nové možnosti, jak program vylepšit a optimalizovat. Projekt jako takový má velký potenciál, jak již bylo uvedeno v úvodu práce, v současné době je trendem rozšiřování možnosti prohlížení. Dalším vylepšením by mohla být přidána širší podpora zařízení, jako např. ty, které disponují více kamerami, aby pokryly lépe celou sféru, zde by byla velice zajímaví implementace, kdy by si musela poradit s více záběry a mapovat je na jeden sférický model. Popřípadě přidat automatické rozeznávání objektu na kaměře, apod.

Závěrečná kapitola obsahuje zhodnocení dosažených výsledků se zvlášť vyznačeným vlastním přínosem studenta. Povinně se zde objeví i zhodnocení z pohledu dalšího vývoje projektu, student uvede náměty vycházející ze zkušeností s řešeným projektem a uvede rovněž návaznosti na právě dokončené projekty.

Pravidla [3]. Pravidla [4]. Goniometrie

# Literatura


[1] Diego Cantor, B. J.: *WebGL Beginner's Guide*. Packt Publishing, 2012, ISBN 9781849691734.

[2] Foundation, M.: *WebGL - Web APIs | MDN*. 2016-2017, [Online].
URL https://developer.mozilla.org/en-US/docs/Web/API/WebGL_API

[3] Katka Maria Delventhal, M. K., Alfred Kissner: *Kompendium matematiky: vzorce a pravidla : četné příklady včetně řešení : od základních operací po vyšší matematiku.* Euromedia Group k.s - Knižní klub v edici Universum, 2004, ISBN 80-242-1227-7.

[4] ODVÁRKO, O.: *Matematika pro gymnázia: Goniometrie*. Prometheus, 2009, ISBN 978-80-7196-359-2.

[5] OpenGL: *WebGL - Web APIs | MDN*. 2016-2017, [Online].
URL http://open.gl

[6] W3C: *HTML5*. 2016-2017, [Online].
URL https://dev.w3.org/html5/spec-author-view/

# Přílohy

# Příloha A

# Příloha 1

# Příloha B

# Obrázky z programu