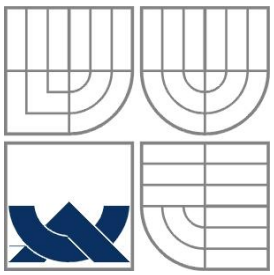
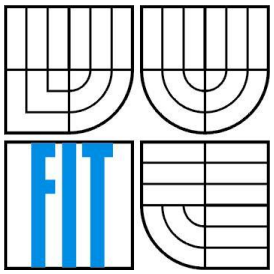


VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY



FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV POČÍTAČOVÉ GRAFIKY A MULTIMÉDIÍ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF COMPUTER GRAPHICS AND MULTIMEDIA

# GENEROVÁNÍ PROCEDURÁLNÍCH TEXTUR V SHADERU

GENERATING PROCEDURAL TEXTURES IN SHADER

BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE — Pavel Veverka
AUTHOR

VEDOUCÍ PRÁCE — Ing. Lukáš Polok
SUPERVISOR

BRNO 2012

*Zadání bakalářské práce/14005/2011/xvever04*

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

Ústav počítačové grafiky a multimédií Akademický rok 2011/2012

# Zadání bakalářské práce

Řešitel: **Veverka Pavel**
Obor: Informační technologie
Téma: **Generování procedurálních textur v shaderu**
**Generating Procedural Textures in Shader**
Kategorie: Počítačová grafika

Pokyny:

1. Prostudujte grafické rozhraní OpenGL, zejména jeho rozšíření.
2. Prostudujte metody generování procedurálních textur
3. Navrhněte shadery, pracující v reálném čase, umožňující výpočet textur, potřebných pro zobrazení areálu Božetěchova 1.
4. Implementujte jednoduché grafické demo s využitím navržených algoritmů.
5. Zhodnoťte dosažené výsledky, identifikujte omezení dané implementací v OpenGL shaderech.
6. Vytvořte stručný plakátek, prezentující výsledky Vaší práce, s důrazem na výtvarné zpracování.

Literatura:

- Dle pokynů vedoucího.

Při obhajobě semestrální části projektu je požadováno:

- Body 1 - 3

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: **Polok Lukáš, Ing.**, UPGM FIT VUT
Datum zadání: 1. listopadu 2011
Datum odevzdání: 16. května 2012

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

doc. Dr. Ing. Jan Černocký
*vedoucí ústavu*

# Abstrakt

Práce se zabývá problematikou generování procedurálních textur a jejich použití v OpenGL. Jsou zde popsány teoretické základy OpenGL a shaderů. Jádrem je rozebrání principů metod generování procedurálních textur pomocí různých algoritmů a šumů, zvláště pak Perlinův šum. Příklady shaderů procedurálních textur jsou demonstrovány v aplikaci, napsané v C++, aplikované na modelu areálu Božetěchova FIT VUT Brno.



# Abstract

This work deals with procedural texture generation using programmable graphics pipeline in OpenGL. It describes basics of OpenGL operation and programmable shading. The main contribution is the analysis of methods used for generating procedural textures using various algorithms and noise functions, with focus on Perlin noise. The examples of procedural texture shaders are demonstrated by a C++ application, displaying 3D model of Božetěchova 2 building.



# Klíčová slova

OpenGL, procedurální textura, shader, GLSL, Perlinův šum, Cellularní šum, alias, MIP mapping, anti-aliasing

# Keywords

OpenGL, procedural texture, shader, GLSL, Perlin noise, Cellular noise, alias, MIP mapping, anti-aliasing


# Citace

Veverka Pavel: Generování procedurálních textur v shaderu, bakalářská práce, Brno, FIT VUT v Brně, 2012

# Generování procedurálních textur v shaderu

## Prohlášení

Prohlašuji, že jsem tuto bakalářskou práci vypracoval samostatně pod vedením Ing. Lukáše Poloka. Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

……

Pavel Veverka

10. 5. 2012

## Poděkování

Chtěl bych rád poděkovat svému vedoucímu práce Ing. Lukášovi Polokovi za odbornou pomoc, rady a vedení při vypracování této práce.

# Obsah

# 1 Úvod

Procedurální textury jsou dnes hojně využívaným prostředkem pro ztvárnění povrchu grafického modelu. Jak už jejich název napovídá, jedná se o textury reprezentované programovým kódem (procedurou). Jejich velikost v řádu kilobytu umožňuje oproti bitmapovým texturám využití více druhů textur, při zachování celkové malé velikosti. Tato vlastnost se využívá například ve 64kB intru. Jelikož jsou textury počítány přímo v místě použití, dá se velice snadno měnit jejich rozlišení, můžeme je parametrizovat a při dobrém generátoru je každá část textury originální. Moderní grafické karty zvyšují výkonnost procesorů zavedením superpipeline (hluboká pipeline). Superpipeline má zvětšený počet stupňů pipeline, tím se zrychlí hodiny a roste počet cyklů hodin, které jsou potřeba pro přenos dat. Zde se mohou procedurální textury výborně uplatnit, protože grafická karta načte pouze kód textury a pak ho provede. Výpočty jsou daleko rychlejší než komunikace a tím se celé zobrazování rapidně zrychlí. Nevýhodou je pak komplikované ladění a některé materiály pomocí rovnic vyrobit nejdou vůbec.

První část věnuje texturám, jejich rozdělením, výhodám a nevýhodám. Také jsou popsány metody a možné problémy při aplikaci textur jako je mapování textur a antialiasing procedurálních, ale i rastrových textur.

V následující kapitole jsou rozebrány základní metody generování procedurálních textur. Uveden je příklad tvorby procedurální textury. Podrobněji je rozebrán Perlinův šum, jeho modifikace a Cellularní šum.

Další část pojednává o grafické knihovně OpenGL , která je v dnešní době hojně užívána pro tvorbu 3D aplikací. Popsány zde jsou základní principy OpenGL, tvorba obrazu a reprezentace modelu. Závěrem této části je vysvětlen princip skládání obrazu pomocí shaderu a programovací jazyk shaderu GLSL

Dále je umístěna praktická ukázka generování materiálu areálu Božetěchova. Postupně podle druhů jsou popsány použité metody geneování, je uvedená ukázka textury a zdrojový kód fragment shaderu.

Předposlední kapitola je věnována vytvoření modelu areálu Božetěchova. Tento model je zásadní částí pro demonstraci příkladu použití procedurálních textur. Pro tento účel byl vytvořen program v C++, do kterého byl model nahrán, a následně na něj byly naneseny textury. V aplikaci se dá po prostorách modelu pohybovat a pro ukázku bylo vytvořené malé demo procházky po nádvoří.

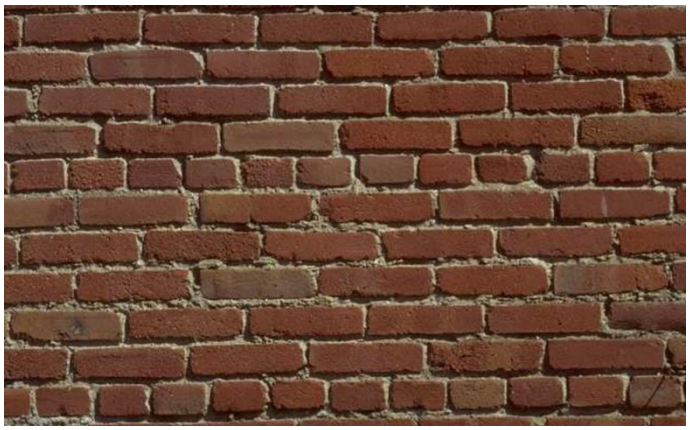
V závěru jsou zhodnoceny výsledky práce a navrhnuty další možné vylepšení demonstračního modelu a jeho textur.

# 2 Textury

Abychom mohli počítačový model realisticky zobrazit, potřebujeme jednotlivým zobrazeným bodům přiřadit barvy, které budou dohromady tvořit strukturu nějakého povrchu. Prvek textury se neoznačuje pixel (prvek zobrazení), nýbrž textel. Samotné barvy ale nestačí. Kdybychom všechny objekty modelu vytvářeli pouze z obarvených geometrických primitiv, tak než bychom věrohodně ztvárnili i nejobyčejnější materiály, museli bychom definovat velmi mnoho objektů. Pro tento účel jsou vytvářeny textury, které definují vzhled povrchů. Poté nám stačí vytvořit jednoduchý objekt a na něj texturu nanést. Nejrozšířenější je příklad cihlové zdi. V reálném světě je zeď složená z cihel a ty jsou pospojované maltou. Samozřejmě že takto se dá zeď vymodelovat i v počítačovém prostředí. Cihlám se nastaví barva červená a malta by měla barvu bílou. Problém nastává, kdybychom takto chtěli modelovat celý dům. Každá cihla by musela být reprezentována zvlášť a počet objektů by nám rapidně narostl. Celá aplikace by poté byla mnohem náročnější na prostředky. V případě použití textury, která ztvárňuje cihlovou zeď, stačí vymodelovat plochu zdí domu a nanést na ně textury. Aplikace je pak značně zjednodušena. Příklad nám dokázal, že se vyplatí modelovat jednodušší geometrii a nanést na ni složitější texturu, zvláště když se jedná o vzdálený objekt. Textury se rozdělují na rastrové a procedurální. První typ se skládá z předpřipraveného rastrového obrázku, který se vždy před zobrazením musí celý načíst, což je mnohdy velmi pomalé a obrázek musí být dostatečně detailní. Druhým typem jsou již výše zmíněné procedurální textury.

## 2.1 Rastrové textury

Základním rysem těchto textur je to, že informace o vzhledu mají uloženou v rastrovém obrázku. Vzniknout mohou vyfotografováním, v grafickém softwaru, nebo mohou být i procedurálně vygenerované. Tímto způsobem můžeme dosáhnout digitalizace jakéhokoliv materiálu. Velice snadno se s nimi pracuje a téměř hned máme představu o jejich podobě. Rastrové textury s sebou nesou také značné nevýhody. První je jejich pevné rozlišení. Jestliže zvolíme malé rozlišení, urychlíme zobrazení modelu, ale mohou se projevit chyby textury. Když naopak máme moc velké rozlišení, tak textura zabere velké množství paměti a zobrazování se zpomalí. Proto je nutné volit kompromis. Další vadou je již zmíněná velikost textury. Jestliže nám dojde paměť v grafickém akcelerátoru, textury se musejí načítat z hlavní paměti počítače a tím zpomalují vykreslování. Poslední nevýhoda částečně souvisí s tou první. Pokud se textury roztahují, nebo zmenšují dochází k *aliasu,* který zobrazenou scénu znehodnocuje. Tento problém je podrobněji vysvětlen v kapitole 2.3.



***Obrázek 2-1*** *Příklad rastrové textury vytvořené fotografií*

## 2.2 Procedurální textury

Přídavné jméno *procedurální* představuje v informačních technologiích entitu, která není vyjádřena datovou strukturou, ale programovým kódem. Procedurální textury využívají pro generování tvarů a barev matematické funkce. Nejčastěji se objevují zapsané v shaderu pomocí speciálního programovacího jazyku. V kapitole 4.1. jsou tyto shadery popsané a v její podkapitole je uveden příklad shaderovacího jazyku pro OpenGL.

Výhody a nevýhody jsme si už nastínili v úvodu, ale zde si je ještě trochu rozvedeme. Jestliže jsme v kapitole o rastrových texturách zmiňovali jejich jednoduché vytváření, tak zde je tomu naopak. Vyplívá to ze způsobu reprezentace. Vytváření procedurálních textur je mnohdy zdlouhavé a většinou od myšlenky k realizaci vede množství pokusů. Metodám generování se věnuje kapitola 3. Máme-li texturu zapsanou v kódu a s jejím vzhledem jsme spokojeni, zjistíme, že manipulace s ní je velice příjemná. Můžeme s ní pokrýt jakkoliv rozměrnou plochu a jelikož nemá pevné rozlišení, vždy pokryje celou plochu, aniž by se roztáhla, nebo zmenšila. Díky její malé velikosti se rychle nahraje do paměti grafického akcelerátoru a GPU ji pak zpracovává samostatně. Můžeme ji parametrizovat a tímto způsobem získáme lehce různé modifikace. Problém pak nastává při provádění antialiasingu, které je složitější.

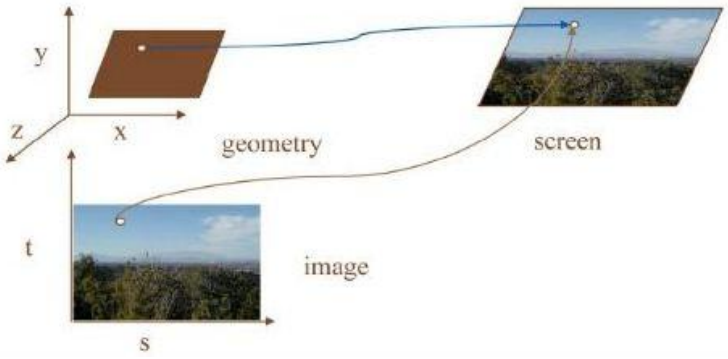
Procedurální textury jsou dále děleny na implicitni a explicitní. Explicitní pracují se všemi body povrchu, kdežto v implicitních se každý bod počítá zvlášť. Pro tuto práci byl zvolen přístup implicitní, kdy do shaderu vstupují textely po jednom a postupně se počítají podle zadaných funkcí. Výhoda výpočtu implicitních textur je například u ray-tracingu (realistické zobrazení odrazu světla), kdy můžeme počítat nezávisle každý pixel zvlášť a nemusíme celou texturu udržovat v paměti. Dále pak můžeme dělit textury na 2D a 3D (solid). V této práci se více setkáme s 2D texturami, které se nanášejí na povrch tělesa. 3D textura má definovanou i hodnotu v prostoru a proto při aplikaci na zakřivená tělesa nedochází k deformaci.

## 2.3 Aplikace textur

Aplikace textur se skládá ze dvou kroků. Prvním krokem je definice textury, tedy zvolení typu reprezentace dat (rastrové, procedurální). Druhým je nanášení textury na objekt, který se nazývá mapování textur (texture mapping). Metody mapování se liší podle toho, zda je textura 2D nebo 3D. Mapování 2D textur můžeme přirovnat k tapetování, kdy nanášíme rovinnou texturu na plochy objektů. 3D textura se spíše podobá vyřezávání z jednoho kusu materiálu. Také velice záleží na členitosti povrchu tělesa. Čím je povrch členitější, tím je složitější nalézt vhodnou projekční funkci, aby nedošlo k deformaci nanášené textury. Většinou se aplikuje více textur a kombinuje se více texturovacích technik (multitexturing). Tímto způsobem lze dosáhnout komplikovaných povrchů složených z více materiálů.

### 2.3.1 Mapování textur

Mapováním textur je označován proces spojování textur a geometrie tělesa. Tato práce je zaměřena na 2D textury a proto se zaměříme právě na ně. Dvourozměrné textury jsou určené souřadnicemi *(s,t)*, ale mapují se na trojrozměrné objekty v souřadnicovém systému *(x,y,z)*. Pro mapování textur na různé geometrické objekty musíme použít různé projekční metody. Jinak se textury mapují přímo na rovnou plochu a jinak na zaoblená, nebo jinak pokřivená tělesa. Každému textelu textury je spočítán dle mapovací funkce příslušný pixel na obrazovce.

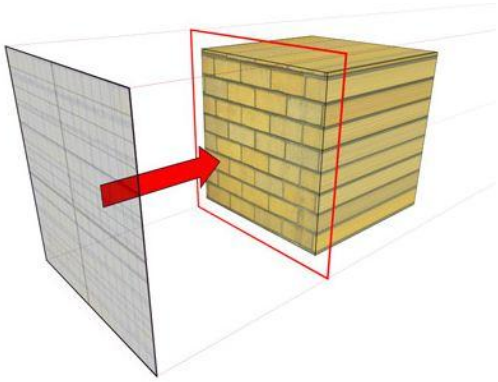



***Obrázek 2-2*** *Mapování textur převzato z [1]*

Vytváření a rozebírání mapovacích funkcí není jádrem této práce. Zde si ukážeme základní principy projekce textur na geometrická tělesa. Tyto projekce se mohou použít ve většině grafických softwarů.

**Planar (Rovinná)**

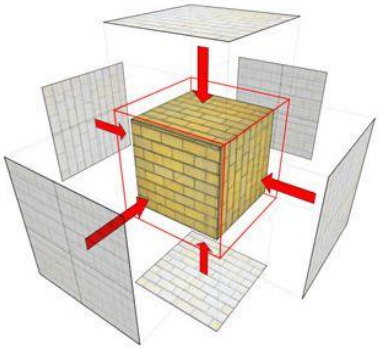
Tento způsob mapování je nejjednodušší. Textura se mapuje na plochu, která se pak promítne na objekt. Je velice užitečný při nanášení textur na objekt pouze z jedné strany.



***Obrázek 2-3*** *Rovinná projekce převzato z [7]*

**Box (Obdélníková)**

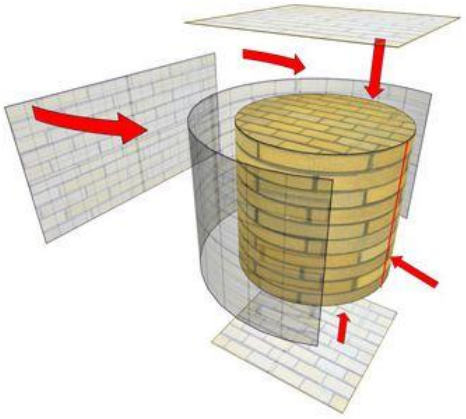
Základem je rovinná projekce, která je aplikována na jednotlivé povrchy cílového tělesa. Tímto způsobem nedojde k deformaci textury, vzniklé použitím pouze rovinnou projekcí. Využívá se u objektů, které jsou viditelné ze všech stran.



***Obrázek 2-4*** *Obdélníková projekce převzato z [7]*

**Cylindrial (Válcová)**

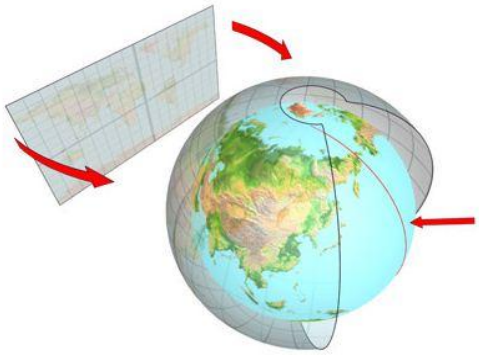
Válcová kombinuje dva způsoby mapování. Prvním je nanášení textury na plášť. Textura se přiloží a obtočí kolem válce. Druhým je mapování obou podstav, které je prováděno rovinným mapováním.



***Obrázek 2-5*** Válcová projekce *převzato z [7]*

**Spherical (Kulová)**

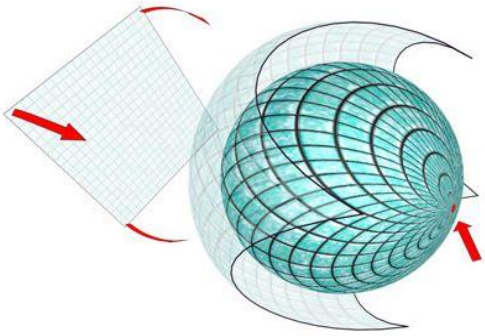
Způsob nanášení textur na kouli. Rovinná textura se z jedné strany přiloží a obtočí se okolo koule tak, že se vrcholy setkají na pólech koule.



***Obrázek 2-6*** *Kulová projekce převzato z [7]*

**Shrink Wrap**

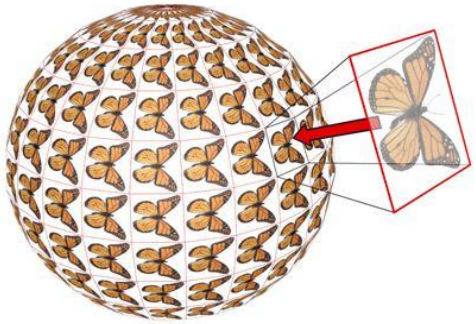
Doslova přeloženo „smršťující obal“ využívá kulový způsob nanášení. Rovinná textura se nanese na kouli tak, že se vrcholy setkají pouze v jednom místě. Tento způsob je velmi užitečný, když chceme ukrýt singularitu textury.



***Obrázek 2-7*** *Shrink Wrap projekce převzato z [7]*

**Face**

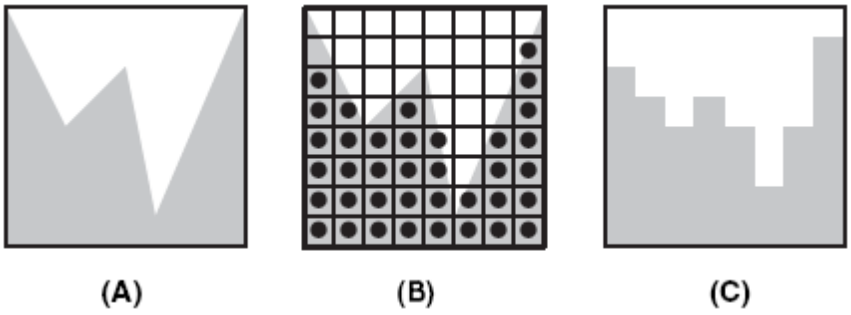
Neboli promítání na plochy. Nanáší kopii textury na každou plochu cílového objektu.



***Obrázek 2-8*** *Face projekce převzato z [7]*

### 2.3.2 Alias

Alias je nežádoucí jev v počítačové grafice, který vzniká nízkofrekvenčním vzorkování signálu s vysokou frekvencí. Tento jev se vyskytuje během zobrazování textur v podobě grafických vad. Lidské oko je extrémně vnímavé na přesnost hran, ale počítačové obrazovky jsou limitovány nejmenšími zobrazovacími jednotkami *Pixely*. Zkombinujeme-li tyto dva faktory, vzniká nám problém při zobrazování detailů menších, než jsou *pixely,* jako jsou hrany. Na obrázku 2-9 je problém ilustrován.

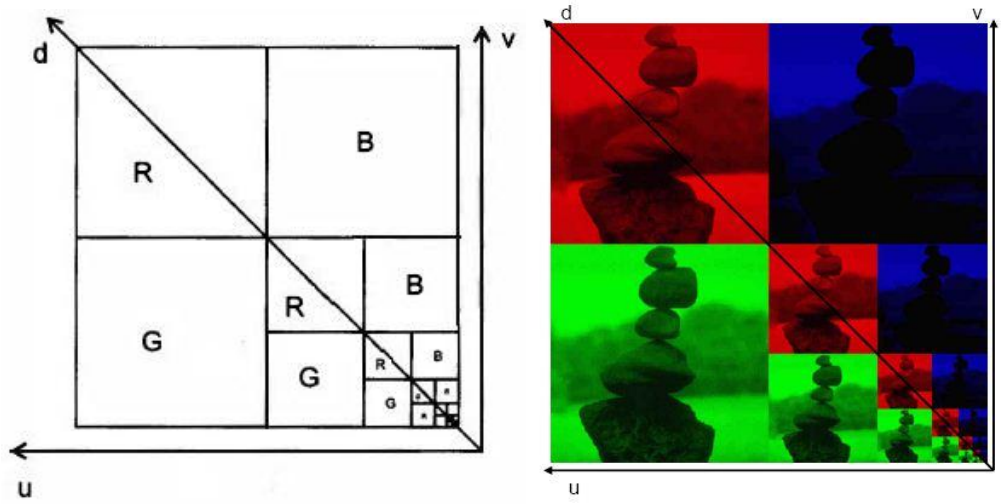



***Obrázek 2-9*** *Vzorkování obrázku převzato z* [3]

V první části obrázku 2-9 (A) je zobrazen vstupní obrázek. Ve druhé části (B) je naznačen způsob vzorkování vstupního obrázku. Třetí část (C) ukazuje nedokonalost zobrazení na obrazovce počítače.

### 2.3.3 MIP mapping

MIP (multum in pravo, neboli mnoho v malém) mapping je metoda pro zrychlení zobrazování texturovaných modelů a jako prostředek anti-aliasingu, tedy odstraňování vzniku aliasu při použití 2D textur. MIP mapping se skládá ze série zmenšených verzí jedné textury rozdělených do jednotlivých barevných složek RGB tak, že tvoří pyramidu od největšího po nejmenší. Tato pyramida se skládá vždy z obrázku textury pouze červené složky, modré složky a složky zelené. V celkovém obrázku zbývá jeden volný roh a v něm je uložena textura opět rozložena do všech tří barevných složek s polovičním rozlišením. Princip zmenšování se opakuje až do reprezentace textury jedním textelem. Uvedeme si příklad. Máme-li texturu s rozlišením 64x64, rozložíme ji na barevné složky a poté změníme rozlišení na 32x32, 16x16, 8x8, 4x4, 2x2 a 1x1. Vznikne nám tak 6 reprezentací textury ($2^6 = 64$), které se získávají filtrací. Možná filtrace je například průměrování čtyř textelů do jednoho nebo složitějšími algoritmy jako Fourierova transformace. Jelikož součet všech ploch konverguje k 1/3 je potřeba o třetinu více místa, než na původní texturu.




***Obrázek 2-10*** *MIP mapping*

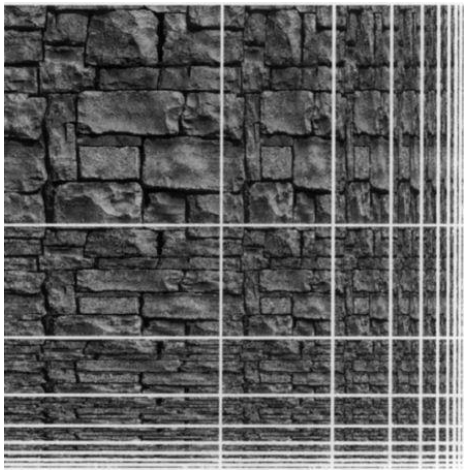
Při mapování je pak nutno zohlednit souřadnici *d*, která určuje vertikální souřadnici v pyramidě. Souřadnice *d* se odvozuje od vzdálenosti od texturovaného objektu dle vzorce (2.3.2).

$$D = \max\left(\sqrt{\left(\frac{\partial u}{\partial x}\right)^2 + \left(\frac{\partial v}{\partial x}\right)^2}, \sqrt{\left(\frac{\partial u}{\partial y}\right)^2 + \left(\frac{\partial v}{\partial y}\right)^2}\right)$$

kde (2.3.2)

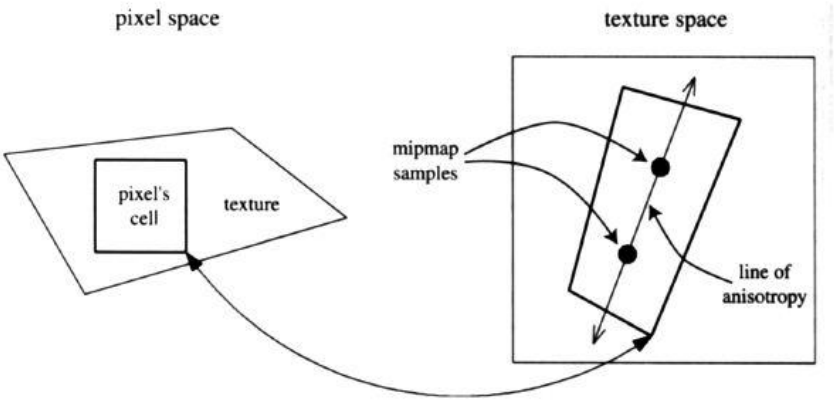
$$d = \log D$$

Souřadnice *u* a *v* pak udávají běžné souřadnice pro mapování textur. Při použití určíme nejbližší stupeň v pyramidě, nebo trilineární interpolaci dvou sousedních stupňů. Nevýhodou MIP mappingu je možné rozmazání textury při pohledu na polygon z velkého úhlu. Tento problém je řešitelný například použitím ripmappingu, který ukládá zmenšené kopie textury nejen v obou směrech zároveň, ale i každý směr zvlášť.



***Obrázek 2-11*** *Ukázka RIP mapping*

Další možností je využití anizotropního filtrování, kde se pro výpočet *d* bere „kratší“ strana promítnutého pixelu, směr „delší“ strany definuje osu anisotropie viz obrázek 2-13




***Obrázek 2-12*** *Anisotropní filtrování*

### 2.3.4 Antialiasing procedurálních textur

Procedurální textury přinášejí mnoho výhod, ale přinášejí i mnohá úskalí. Jednou s těchto nevýhod je složitost řešení aliasingu. Jelikož jsou procedurální textury generovány až na místě potřeby, nemůžeme použít klasické metody anti-aliasingu, jako je MIP mapping. Tento způsob vyžaduje uloženou předlohu textury, ale tím bychom přišli o většinu výhod procedurálního generování textur. Proto problém aliasingu by měl vyřešit už programátor při tvorbě textury. Metod pro řešení aliasingu je několik.

**Analytická předfiltrace**

Častým původem aliasu je používání „ostrých“ přechodů, tedy přechodů kde se změna z 0 na 1 děje skokově a není nic mezi tím. Zmírnit tento efekt můžeme tak, že texturu před použitím přefiltrujeme pomocí dolnopropustních filtrů (low-pass filter). Filtr potlačuje vysoké frekvence a ty nízké procházejí beze změny. V praxi se potlačení vysokých frekvencí projeví jako uhlazení a tedy i potlačení zubatých hran. V kapitole 4.3.1 se následně dozvíme, že v OpenGL lze této metody využít používáním funkce *smoothstep* na místo funkce *step*.

**Přizpůsobivá analytická předfiltrace**

Velikost vyrovnávacího filtru je definována v parametrickém prostoru, ale parametr se nemění při konstantním hodnocení v zobrazovaném prostoru. V tomto případě se souřadnice mění rychleji na pólech koule než na rovníku. Proto použití obyčejného filtru bude dobře fungovat na rovníku, ale hůře pak na pólech. Abychom mohli dostatečně zareagovat na změnu parametrů v prostoru, musíme monitorovat, jak rychle se funkce mění v závislosti na pozici v zobrazovaném prostoru. Tyto informace můžeme získat pomocí parciálních derivací souřadnic *x* (jak rychle se mění funkce ve směru x) a *y* (jak rychle se mění funkce ve směru y). Z těchto informací lze spočítat

„Gradient vektor“ . Máme-li funkci *f(x,y)* tak gradient vektor na pozici *(x,y)* je definován dle vzorce (2.3.3.A)

$$G[f(x, y)] = \begin{bmatrix} \frac{\partial f}{\partial x} \\ \frac{\partial f}{\partial y} \end{bmatrix} \qquad (2.3.3.A)$$

Další důležitou vlastností Gradient vektoru je to, že ukazuje směr a maximální míru zvýšení funkce, nazývaný „Gradient funkce“(vzorec 2.3.3.B)

$$[G[f(x, y)]] = \mathbf{sqrt}((\partial f/\partial x)^2 + (\partial f/\partial y)^2) \qquad (2.3.3.B)$$

V praxi se nemusí počítat gradient tak náročně. Proto je zaveden filtr width, který počítá sumu absolutních hodnot.(vzorec 2.3.3.C)

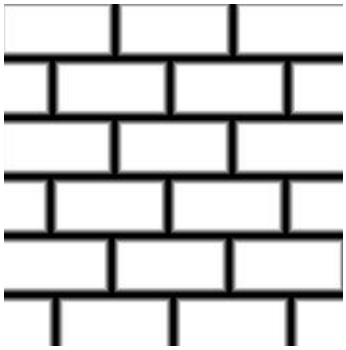
$$[G[f(x, y)]] \cong \mathbf{abs}(f(x, y) - f(x + 1, y)) + \mathbf{abs}(f(x, y) - f(x, y + 1)) \qquad (2.3.3.C)$$

Tato hodnota je poté horní mez pro odstranění frekvencí potřebných k potlačení aliasingu.

# 3 Metody generování

Nejčastějším způsobem tvorby procedurálních textur je postupná kombinace metod generování, které budou popsány dále. Jako příklad si můžeme uvést postup vytvoření cihlové zdi.

1. Vytvoření cihel – využijeme funkci pro tvorbu přechodů, které umístíme na správné pozice, tím dosáhneme tvaru cihel

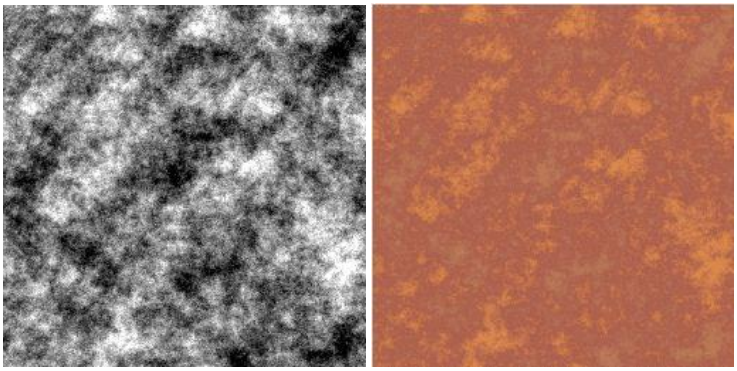


***Obrázek 3-1*** *Tvar cihel*

2. Deformace spár – aby spáry nevypadaly pravidelně, ale více realisticky, sečteme mapu cihel s Perlinovým šumem

***Obrázek 3-2*** *Deformace tvaru cihel*

3. Obarvení – pro spáry použijeme kombinaci černé a bíle, kterou zkombinujeme pomocí dalšího Perlinova šumu. Tímto vytvoříme zrnitý efekt spár tvořených maltou. Na cihly poté naneseme barvu složenou s několika různě upravenými Perlinovými šumy, které zajistí nepravidelné rýhy v cihlách



***Obrázek 3-3*** *Kolorizace Perlinova šumu*

4. Kompletace - posledním krokem je zkompletování všech částí do jedné. Výsledek příkladu je na obrázku 3-4.

***Obrázek 3-4*** *Výsledná textura cihel*

# 3.1 Šumy

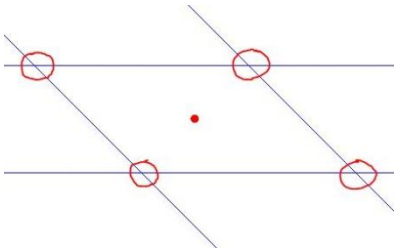
Šumy jsou velmi využívanou metodou v procedurálním generování nejen textur. Díky nim je velmi snadné vytvářet struktury materiálů. Šumy vytváří pocit náhodnosti (pseudo-náhodné), ale jsou kontrolovatelné. Samotné šumy vytvářejí pouze jednoduché vzory, jejich uplatnění přichází až s kombinací s jinými matematickými funkcemi. Tímto způsobem lze vytvořit celou řadu různých materiálů.

## 3.1.1 Perlinův šum [6]

Název nese podle svého tvůrce, kterým byl Ken Perlin. Definován může být pro jakýkoliv počet dimenzí, ale většinou je jako funkce implementován jen pro dva, tři, nebo čtyři rozměry. Podle počtu dimenzí narůstá i složitost algoritmu(vzorec 3.1.A).

$$O(2^n) \tag{3.1.A}$$

Algoritmus tvorby Perlinova šumu postupuje následovně. Rozdělí prostor na čtvercovou síť. Poté co se příjme vstupní bod *P*, zaznamenají se všechny jeho sousední body ze sítě označované *Q*. Pro 2D jsou takové body čtyři, pro 3D jich pak je osm. Pro *n* dimenzí zaznamenáme $2^n$ bodů a proto nám vzniká složitost uvedená vzorci 3.1.A



***Obrázek 3-5*** *Sousední body dvourozměrného vstupu*

Pro všechny získané body *Q* určíme pseudonáhodný gradient vektor *G*. Nyní můžeme spočítat lineární funkci označovanou jako „dot product“ (vzorec 3.1.B).

$$G * (P - Q) \tag{3.1.B}$$

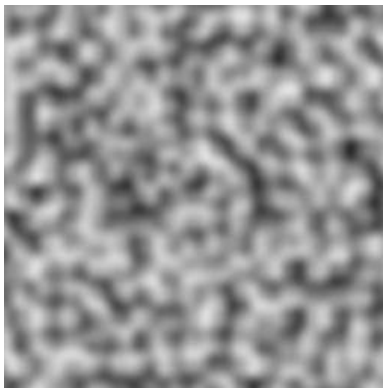
Rovnici „dot product“ vypočteme pro všechny body *Q*, tím získáme $2^n$ hodnot, které mezi sebou interpolujeme ke vstupnímu bodu. Pomocí křivek ve tvaru S určené vzorcem (3.1.C) určíme váhu interpolace pro každou dimenzi.

$$3t^2 - 2t^3 \tag{3.1.C}$$

Počítání pseudonáhodného gradientu musí být velice rychlé, proto je nutné si připravit tabulku permutací *P[n]* a následně tabulku gradientů *G [n]*. (vzorec 3.1.D)

$$G = G[i + P[j + P[k]]] \tag{3.1.D}$$

Kde ve vzorci (3.1.C) jsou *i,j,k* celočíselné hodnoty *Q* bodu.



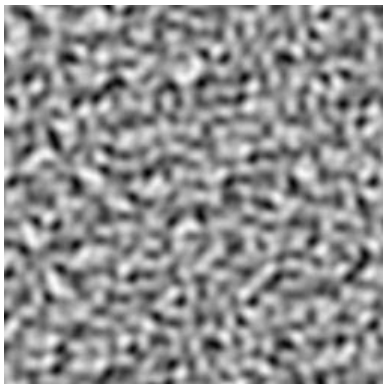
***Obrázek 3-6*** *Klasický Perlinův šum*

## 3.1.2 Simplex šum

Ken Perlin v roce 2001 vyvinul vylepšení klasického Perlinova šumu, aby byl méně výpočetně náročný. Zatímco klasický šum rozděluje prostor čtvercovou sítí, tak Simplex šum rozděluje prostor na síť bodů. Pro výpočet se využívá pouze nejbližší bod, a proto pro každý bod potřebujeme pouze *n+1* výpočtů. Složitost je tedy definována vzorcem (3.1.E).

$$O(n^2) \tag{3.1.E}$$

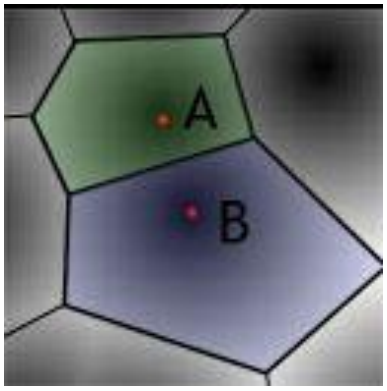
V této práci je využit právě tento „vylepšený“ Perlinův šum.



***Obrázek 3-7*** *Simplex šum*

### 3.1.3 Cellulární šum [8]

Hlavní principem je vytvoření bodů náhodně rozptýlených po celé textuře a poté pro každý textel nalézt nejbližší vygenerovaný bod a spočítat jeho vzdálenost. Vzdálenost můžeme následně využít k obarvení textelu. Například v obrázku 3-8 je obarvení nastaveno úměrné k druhé mocnině vzdálenosti k bodu, tím dosáhneme ztmavení textelů blíže u bodu.




***Obrázek 3-8*** *Cellularní šum převzato z [8]*

Při zjišťování nejbližšího bodu je dosti nevhodné a prostředkově náročné procházet a porovnávat všechny body. Pro odstranění tohoto problému vznikají různé modifikace. Například lze použít BSP-strom, do kterého jsou body ukládány již při generování.

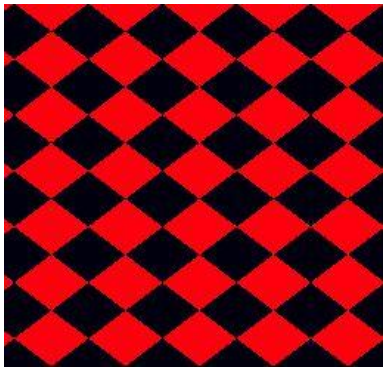
## 3.2 Vymezení prostoru

V zásadě se jedná o určování oblastí pro použití definovaných barev, nebo použití celých barevných map. Jelikož pojednáváme o texturách procedurálních a barvy jednotlivých textelů se počítají dle rovnice reprezentované kódem, můžeme k definování oblastí využít kombinaci většiny

matematických funkcí. Kódová representace nám umožní využít rozhodovacích konstrukcí, nebo cyklů. Jedním typickým příkladem je rozdělení obrazu na dvě zóny. (vzorec 3.2.A)

$$F(x, y) > 0 \tag{3.2.A}$$

Dle výsledku pravdivosti výrazu lze nastavit příslušnou barvu. Některé programovací jazyky shaderů mají pro tento případ implementované speciální funkce. Například v GLSL to je funkce *step()*. Poté se použije jako třetí parametr funkce *mix()*. Tyto funkce jsou popsané v kapitole 4.3.1.



***Obrázek 3-9*** *Šachovnice*

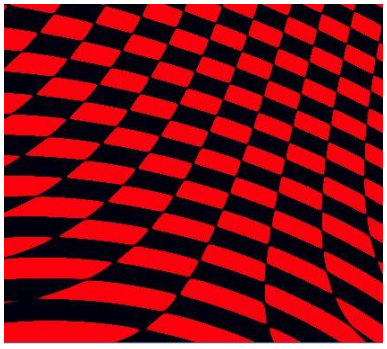
Pro vytvoření obrázku šachovnice (obrázek 3.2) je použita funkce pro rozdělení prostoru (vzorec 3.2.B)

$$\sin(\mathrm{x}) - \cos(y) > 0 \tag{3.2.B}$$

# 3.3 Deformace souřadnicového systému

Další velmi užitečnou metodou, která je také dále využita v této práci, je deformace souřadnicového systému. Jedná se o metodu, kdy se textura generuje normálním způsobem, ale na začátku kódu se zdeformuje celý souřadnicový systém. Metodu můžeme přirovnat k potisku na textilii, kterou různě zohýbáme a poté položíme na zem. Při pohledu shora nám vznikne textura nová. Jako příklad jsme použili texturu použitou na obrázek 3-9 a té jsme upravili souřadnice vzorcem (3.3) a výslednou texturu můžeme vidět na obrázku 3-10.

$$x' = x * \sin(x - y) \tag{3.3}$$
$$y' = y * \sin(x - y)$$



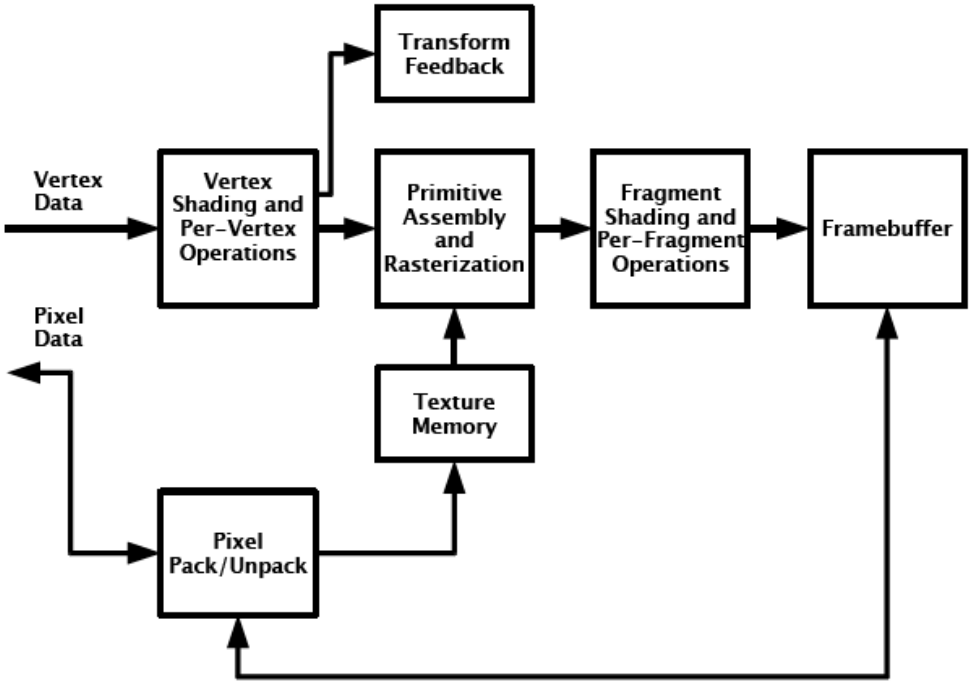
***Obrázek 3-10*** *Šachovnice s deformovanými souřadnicemi*

# 4 OpenGL

Open Graphics Library, neboli OpenGL [2], je knihovna původně vytvořená jako otevřená alternativa k Iris GL. Je to API (aplikační programové rozhraní) pro komunikaci s grafickým subsystémem. Při tvorbě byl kladen veliký důraz na použitelnost na různých typech operačních systémů a grafických akcelerátorů. Komunikace je založena na způsobu klient/server, kdy mohou být obě části na jednom počítači. Knihovna je vytvořena tak, aby se dala použít v co nejvíce druzích programovacích jazyků, ale jako základní je používán jazyk C a C++. Při vykreslování se chová jako stavový automat. OpenGL přímá příkazy pro zobrazení grafických primitiv (bod, úsečka, trojúhelník, …), ale také informace, jak mají být tato primitiva vykreslena (barva, průhlednost, transformace, …). Tyto informace se uloží do OpenGL state machine (stavový stroj) a toto nastavení zůstane do té doby, než jej změníme. Takto mohou mít vykreslovací funkce méně parametrů a jedním příkazem se dá změnit způsob vykreslení celé scény.

## 4.1 Rendering pipeline

Vykreslení probíhá tak, že se ve Framebufferu vytvoří rastrový obrázek pomocí rendering pipeline (vykreslovací řetezec), který je podrobněji zobrazen na obrázku 4-1.




***Obrázek 4-1*** *Rendering pipeline OpenGL převzato z [2]*

Do rendering pipeline vstupují Vertex data, která nesou informace o vrcholech grafických primitiv (souřadnice, barvu, normálu,…). V první fázi se řeší geometrie primitiv popsaných vrcholů. Data se mohou transformovat a osvětlit. Poté se poskládají do geometrických primitiv, aby byla připravena na další fázi. Geometrie stínování generuje nová primitiva. Výsledek je omezen na zobrazování rozsahu pro rasterizaci. Rasterizace vytváří sérii adres framebufferu a hodnot používajících dvojdimenzionální popis bodu, úsečky nebo polygonu. Každý fragment je přiveden do další fáze, která provádí operace jednotlivě na fragmenty předtím, než jsou umístěny do framebufferu. Tyto operace zahrnují podmíněné aktualizace ve framebufferu, založených na dříve uložených hlubokých hodnotách, které jsou efektem hlubokých bufferů, míchání barev příchozích fragmentů, uložených barev, maskování a jiných logických operací hodnot fragmentů. Nakonec mohou být hodnoty kopírovány do jiných framebufferů, nebo vyčteny z framebufferu s využitím různých technik kódování.

## 4.2 Reprezentace modelu

Geometrické modely jsou reprezentovány pomocí vrcholů, které jsou uloženy pomocí vertexů. Vertexy jsou kolekcí generických atributu. Obsahují informace o pozici souřadnic vrcholu, barvě, souřadnici textury a některá další, které s bodem mohou souviset. Všechny tyto vertexy jsou uložené ve *vertex buffer objects* označovaným také jako VBOs. Pro přiřazení VBOs k jednotlivým objektům

je využíván další buffer a to *vertex array objects* zkráceně VAOs. To znamená, že VAOs je kolekcí VBOs, které se tak dají lehce spravovat. Tento přístup nám značně zjednoduší implementaci vykreslování modelů. Bez těchto bufferu bychom museli pro každé z primitiv volat sérii funkcí pro každý jeho parametr a velice by narostla komunikace mezi procesorem počítače (CPU) a grafickým procesorem (GPU). Můžeme tak jedním příkazem vykreslit geometrii po větších kusech, aniž bychom museli komunikovat mezi CPU a GPU, protože data potřebná pro vykreslování se přenesou najednou a uloží se přímo do GPU.

***Obrázek 4-2*** *Model zobrazen sítí trojúhelníků*

# 4.3 Shader

Je program usnadňující programátorům kontrolovat části grafické pipeline. Shadery se rozdělují na několik typů, podle toho pro kterou část jsou určeny. Nejčastěji se používají vertex, fragment a geometry shader. S vydáním OpenGL 3.2 byly do vykreslovacího řetězce přidány moduly pro teselaci, která umožňuje generování detailů geometrie těles s podporou hardwaru. Teselace se v OpenGL provádí ve třech fázích. První je konfiguratelná a druhě dvě programovatelné. Tyto programovatelné části jsou Tessellation control shader a Tessellation evaluation shader.

**Vertex shader** – provádí operace na všechny vstupující vrcholy grafických primitiv. Zejména maticové transformace, díky kterým můžeme simulovat různé grafické efekty, jako pohyb vodní hladiny.

**Fragment shader** – označovaný v knihovne Direct3D jako Pixel shader. Počítá barvu každého pixelu a právě pomocí tohoto shaderu můžeme vytvářet procedurální textury.

**Geometry shader –** do shaderu vstupují celá grafická primitiva a může ovlivňovat celou geometrii. Body mohou být přidávány i odebírány a tímto způsobem se dají v reálném čase generovat dodatečné detaily. Například můžeme vygenerovat ostny růže na její stonek.

**Tessellation control shader** – zpracovává vrcholy vystupující z vertex shaderu a generuje z nich výstupní patche a definuje teselační parametry.

**Tessellation evaluation shader** – vstupem jsou teselační patche a atributy. Z nich počítá finální vertexy generovaných primitiv.

Pro psaní těchto programů můžeme využít speciální programovací shader jazyky. Rozdělujeme je dle typu použitého grafického rozhraní. Pro psaní shaderů v OpenGL se používá OpenGL Shading Language.

### 4.3.1 GLSL

GLSL je zkratkou výše zmíněného programovacího jazyka OpenGL Shading Language [3]. Syntax, operátory, skalární datové typy i řídící struktury a funkce přebírá z jazyka C. Výjimku tvoří operátory ukazatelů, které GLSL nepodporuje. Dále poskytuje specifické prostředky pro práci s počítačovou grafikou. V prvé řadě to je datový typ vektor. Jsou podporovány jednosložkové až čtyřsložkové vektory. Název tvoří prefix (vec), který označuje datový typ složek a sufix určující jejich počet. Vektor může nabývat různých datových typů, jako jsou celá i desetinná čísla, ale i pravdivostní hodnoty. Příklad vytvoření vektoru: *vec4 color = vec4(1.0, 0.0, 0.0, 0.0);* Další novým datovým typem je matice. Ta se může vytvářet ve velikosti 2x2 až 4x4. Můžeme použít všechny kombinace rozměrů, tudíž matice nemusí být čtvercová. Název je tvořen podobně jako u vektorů, kde je prefix (mat) a sufix (2x4, nebo u čtvercové matice pouze 2). Oproti vektorům lze vytvořit matice obsahující pouze datový typ s plovoucí desetinnou čárkou. Zde je opět příklad vytvoření *mat3 matrix;* Oproti jazyku C disponuje GLSL také novým operátorem swizzle. Tento operátor umožňuje permutaci jednotlivých složek vektoru. Mějme například vektor A = {1, 2, 3, 4}, jehož složky jsou x, y, z a w v daném pořadí. Potom můžeme vypočíst vektor B = A.wwxy, přičemž výsledný vektor bude roven B = {4, 4, 1, 2}.

GLSL poskytuje implementaci řady funkcí, zejména pak matematických. Funkce jsou rozdělené do několika kategorií. Úhlové funkce slouží pro převod jednotek úhlů mezi stupni a radiány. Trigonometrické obsahují celou sadu trigonometrických funkcí od základního sinusu až po hyperbolický tangens. Exponenciální obsahují funkce pro mocnění, odmocňování a logaritmování. Geometrické umožňují operace s vektory, maticové s maticemi. Obsáhlou skupinou jsou funkce pro práci s texturami. Některé funkce jsou použitelné pouze ve fragment shaderu, využívající fragment procesor. Fragment shader má omezené možnosti získání informací o svých sousedech a pro využití metod anti-aliasingu musíme vypočítat hodnoty derivací. Počítání derivací může být výpočetně náročné, nebo numericky nestabilní. OpenGL využívá rychlé, ale méně přesné výpočty s využitím dopředné diference (vzorec 4.3.1.A), která je numerickou derivací.

$$F(x+dx)-F(x) \sim dFdx(x)\cdot dx$$

$$dFdx(x) \sim \frac{F(x+dx)-F(x)}{dx}$$

(4.3.1.A)

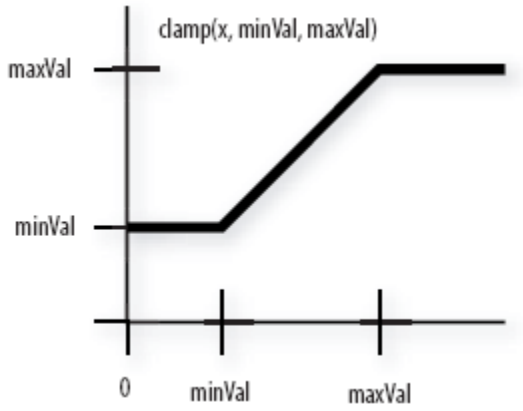
dFdy je aproximováno stejně, jen je nahrazeno x za y. Po výpočtu *dFdx* a *dFdy* můžeme aplikovat filtr width, který je na těchto funkcí založen. Pro aplikaci filtru width je implementována funkce *fwidth,* která vrací sumu absolutních hodnot derivaci podle *x* a *y* s argumentem *p* podle vzorce (4.3.1.B) .

```
abs(dFdx(p)) + abs(dFdy(y))
```
(4.3.1.B)

Zvláštní kategorií jsou pak funkce obecné, do které zahrnujeme všechny ostatní funkce. Do této kategorie patří například funkce *mix*, do které vstupují parametry *x, y, a*. Funkce vrací lineární smíchání x a y, použitím vzorce (4.3.1.C).

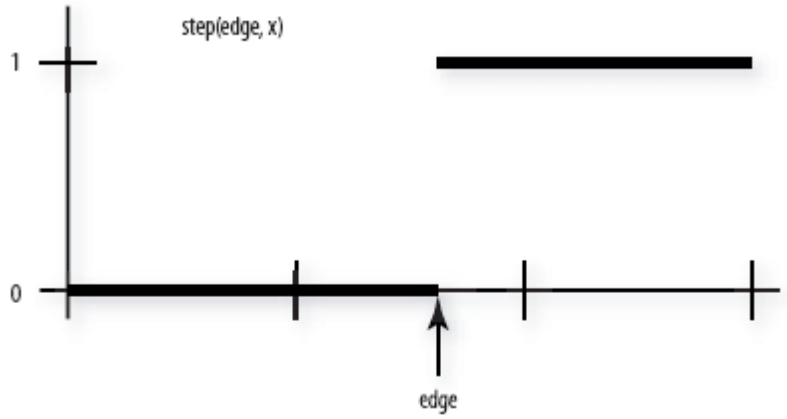
$$x*(1-a)+y*a \tag{4.3.1.C}$$

Pro tyto vlastnosti je využívána například k smíchání dvou barev dle parametru *a*. Další zajímavou funkcí je *clamp.* Vstupem jí jsou parametry *x, minVal, maxVal* a vrací hodnotu určenou vzorcem *min(max(x,minVal),maxVal)*. Funkce *min* a *max*, jak už název napovídá vracejí maximální, nebo minimální hodnoty vstupů a proto má *clamp* průběh, který je zobrazen na grafu 4.3.1.



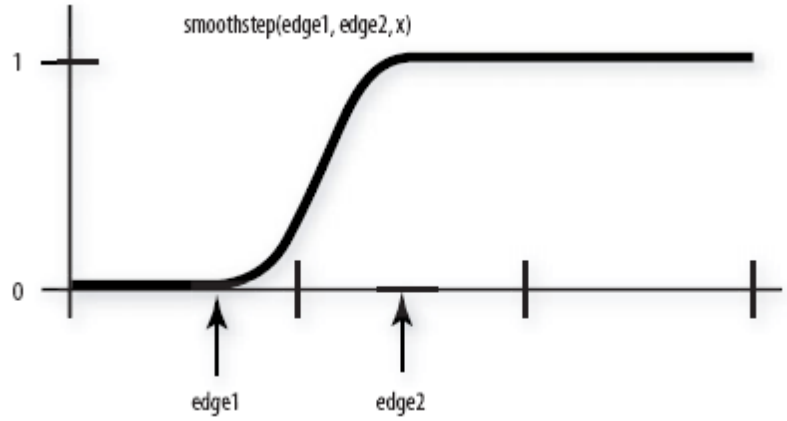

***Graf 4-1*** *Průběh funkce clamp převzato z [3]*

Hojně využívanou funkcí, která umožňuje vytvářet přechody, je *step*. Funkce vytváří kontinuální skoky na zadané hraně.




***Graf 4-2*** *Průběh funkce step převzato z [3]*

Pro vytváření plynulých přechodů GLSL má implementovanou funkci *smoothstep*. Vstupem jsou hrany *edge1*, *edge2* a hodnota *x* . Funkce vrací 0.0 když je `x <= edge0` a 1.0 když `x>=edge1`, jestliže `edge0 < x < edge1` vrací Hermitovu interpolaci mezi 0 a 1.




***Graf 4-3*** *Průběh funkce smoothstep převzato z [3]*

Funkce *smoothstep* je pro své vlastnosti programátory shaderů využívána pro potlačení alisau způsobem uvedeným v kapitole 2.3.4. Na obrázku4-3 vlevo je vytvořen přechod pomocí funkce *step* s hranou v hodnotě 0.01. Vpravo je pak použita funkce *smoothstep* s první hranou na hodnotě 0.01 a druhou hranou na hodnotě 0.015.

***Obrázek 4-3*** *Porovnání funkce step a smoothstep*

Z uvedeného příkladu je vidět vyhlazení přechodových hran funkcí *smoothstep*. Textura vytváří lehce rozmazaný efekt, ale má zcela odstraněný alias tedy zubaté hrany.

# 5 Materiály budovy Božetěchova

Všechny výše popsané metody generovaní byly použity na výrobu materiálů použitých na vnitřním nádvoří areálu kláštera na ulici Božetěchova v Brně. Nejdříve bylo nutné všechny druhy materiálů detailně nafotit a poté byly vyrobeny kombinováním metod popsaných v kapitole 3. Materiály jsou seřazeny od nejjednodušších po složitější. Vždy je uvedeno porovnání screenshotu vyrobené textury (umístěn **vlevo**) a fotografie skutečného materiálu (umístěna **vpravo**), popsané metody použité v dané textuře a zdrojový kód shaderu. Každý shader má jednu vstupní proměnnou deklarovanou `in vec2 v_texcoord;`, jednu výstupní proměnnou `out vec4 frag_color;`, implementované funkce šumu `snoise, cellular` a definovaný vektor `color`. Všechny textury jsou na modelu naškálovány na potřebnou míru.

## 5.1 Omítka

Omítka v areálu pokrývá největší plochu a zároveň vytvoření metodami pro generování procedurálních textur je nejjednodušší. Základem je funkce *mix*, která smíchá dvě barvy v závislosti na funkci šumu. Textura omítky byla vytvořena ve dvou barevných kombinacích v bílé a béžové.

```
vec4 wall = color(0xE6E6E6);
vec4 lumps = color(0xEFEFEF);
float sul = snoise(v_texcoord*200);
frag_color = mix(wall, lumps,  sul);
```

***Kód 5-1*** *Textura omítky*



***Obrázek 5-1*** *Textura omítky bílé*

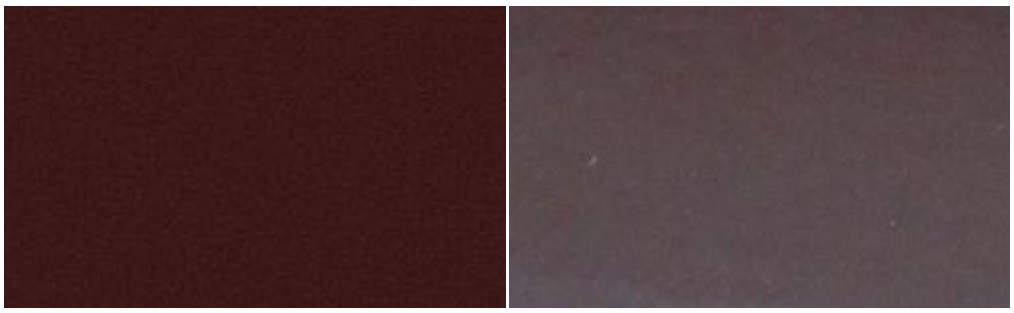
***Obrázek 5-2*** *Textura omítky béžové*

## 5.2 Železo

Železo je vytvořeno téměř totožně jako omítka. Rozdíl je v barevné kombinaci a vypočtený šum je posunut od počátku souřadnicového systému. Předloha byla převzata z podoby parapetu.

```
vec4 barva1 = color(0x4C1F1F);
vec4 barva2 = color(0x2E1313);
float sul = snoise(v_texcoord * 200.0) * .3 + .5;
frag_color = mix(barva2,barva1,sul);
```

***Kód 5-2*** *Textura železa*



***Obrázek 5-3*** *Textura železa*

## 5.3 Tráva

V textuře trávy je demonstrována možnost deformace vstupních souřadnic do fragment shaderu. Opět jsou míchány dvě barvy podle vygenerovaného šumu. Při výpočtu šumu je souřadnice *x* vynásobena konstantou a tím je dosaženo efektu členitosti trávy.

```
vec4 barva1 = color(0x009200);
vec4 barva2 = color(0x004E00);
float sul = snoise(vec2(x*10,y)*20) * .3 + .7;
grass = mix (barva2,barva1 , sul);
```

***Kód 5-3*** *Textura trávy*

***Obrázek 5-4*** *Textura trávy*

## 5.4 Umělé dřevo

Pro texturu dřeva, které se používá pro ztvárnění podoby dveří a rámu oken, je využit princip generování šumu jako u textury trávy jen s jinými parametry. Pro realistické ztvárnění odstínu barev je smícháno více barev s různými intenzitami.

```
vec4 barva1 = color(0xDC7610);
vec4 barva2 = color(0xFF9933);
vec4 basic = mix(mix(barva2, barva1,
abs(cnoise(vec2(x*80,y-2*80))*100)),barva1,.75);
vec4 barva3 = mix(barva1, color(0x00000),.05);
frag_color = mix(barva3,basic,0.02);
```

***Kód 5-4*** *Textura umělého dřeva*

***Obrázek 5-5*** *Textura umělého dřevo*

## 5.5 Dřevo na lavičky

Dřevo na lavičky vycházejí opět z principu posunutí jedné souřadnice při generování šumu. Nyní je poprvé použita funkce *step* pro vytvoření pruhů přes texturu. Šířku pruhů určuje hodnota určující hranu přechodu funkce *step*. Frekvence opakování je dána hodnotou vypočtenou funkcí *mod*. Barva je zvolena trochu výraznější pro lepší demonstraci ve výsledném modelu.

```
float sul = snoise(vec2(x,y*100)*10) * .3 + .7;
float pruh = step(.03, abs(mod(v_texcoord.y, .07) - .05));
vec4 barva1 = color(0x94704D);
vec4 barva2 = color(0x684E36) * color(0x958372);
frag_color = mix(barva1*sul, sul* barva2 , pruh);
```

***Kód 5-5*** *Textura dřeva na lavičky*

***Obrázek 5-6*** *Textura dřeva na lavičky*

## 5.6 Mřížky

Mřížky použité kolem kašny pro odvod vody tvoří znovu funkce *step*, avšak nyní je vstupem souřadnice *x*, čímž se zajistí svislá orientace pruhů. Pruhy jsou čistě černé a jednotlivé mřížky jsou tvořeny barvou s přidaným šumem.

```
float sul = snoise(v_texcoord * 300.0) * .3 + .7;
float pruh = step(.03, abs(mod(v_texcoord.x, .07) - .05));
vec4 black = vec4(0);
vec4 zelezo = color(0x78AEDF) * color(0x555555);
frag color = mix(black, sul* zelezo , pruh);
```

***Kód 5-6*** *Textura mřížek*

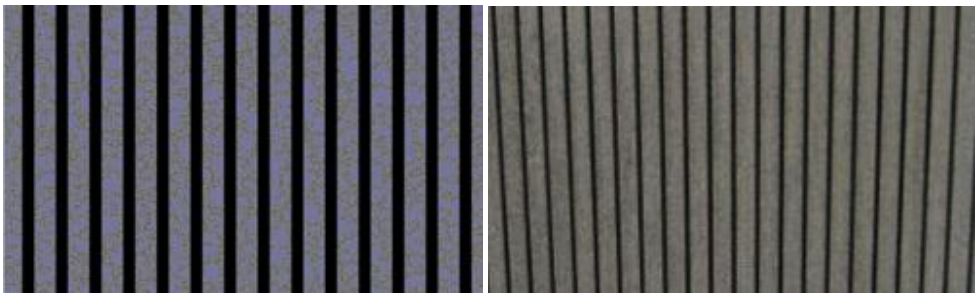
***Obrázek 5-7*** *Textura mřížek*

## 5.7 Kašna

Kamenná struktura kašny je kombinací dvou šumů. První je klasický šum použitý i v texturách výše. Ve druhém byla využita funkce *abs*, která vytvoří strukturu typickou pro kámen použitý na fontáně. Pruhy mezer jsou nyní vytvořené funkcí *smoothstep*, která zjemní hrany přechodů.

```
vec4 barva1 = color(0x767676);
vec4 barva2 = color(0x7272A5);
float sul = snoise(v_texcoord * 100.0) * .3 + .7;
float struktura = abs(snoise(v_texcoord*15)*sul) * .5 + .5;
barva1 = mix (vec4(1),barva1 , sul * .6 + .6);
vec4 kamen = mix(barva1,barva2,smoothstep(.6, .8, struktura));
float pruh = smoothstep(.01, .015, abs(mod(v_texcoord.x, .07) - .05));
vec4 mezera = vec4(0);
frag color = mix(mezera, kamen , pruh);
```

***Kód 5-7*** *Textura kamenů na kašně*



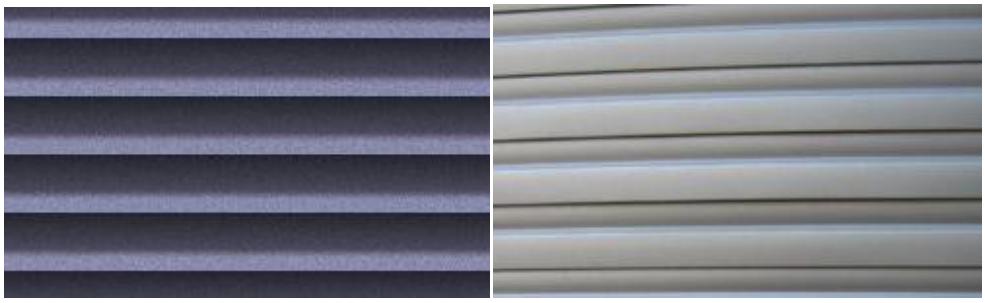
***Obrázek 5-8*** *Textura kamenů na kašně*

## 5.8 Žaluzie

Pro vytvoření textury žaluzií jsou použité dva pruhy, které jsou zkombinované do sebe. Nejprve je vytvořen první pruh, tvořící ostrý přechod jednotlivých žaluzií. Druhý pruh překrývá první a tvoří přechod mezi světlejší a tmavší barvou. Tím je vytvořen efekt stínu. Na celou texturu je pak přidán šum, který vytváří jemnou strukturu.

```
float sul = snoise(v_texcoord * 500.0) * .1 + .9;
float pruh = mod(v_texcoord.y, .05) / .05;
float pruh2 = smoothstep(.01, .02, abs(mod(v_texcoord.y, .05) - .05));
pruh = pruh * .8 + .5;
frag_color = mix(color(0x666680)*sul*1.5,color(0x525266)* sul * pruh,pruh2);
```

***Kód 5-8*** *Textura žaluzií*



***Obrázek 5-9*** *Textura žaluzií*

## 5.9 Železná podlaha

Vynásobením dvou pruhů tvořených funkcí *smoothstep*, jenž do jednoho vstupuje souřadnice *x* a do druhého souřadnice *y*, získáme čtvercovou síť. Materiál železa je vytvořen kombinací barev s přidaným šumem. Mezery mezi železnou sítí jsou vyplněny černou barvou.

```
float sul = snoise(v_texcoord * 50.0) * .3 + .7;
float pruh = smoothstep(.01, .02, abs(mod(v_texcoord.x, .13) - .05));
float pruh2 = smoothstep(.01, .02, abs(mod(v_texcoord.y, .13) - .05));
vec4 vnitrky = color(0x000014);
vec4 pruhy = mix(color(0xE0E0EB),color(0xC2C2D6),sul*1.5)*color(0xCCCCCC);
frag_color = mix(pruhy,  vnitrky, pruh * pruh2)*color(0xB2B2F0);
```

***Kód 5-9*** *Textura železné podlahy*

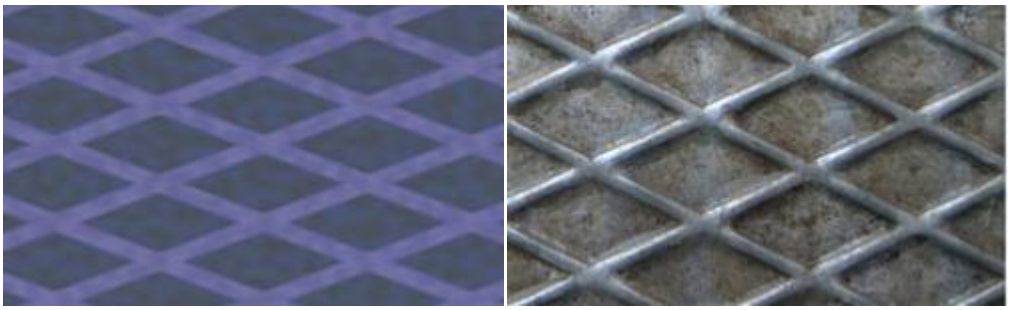
***Obrázek 5-10*** *Textura železné podlahy*

## 5.10 Tahokov

Zde je demonstrována další metoda generování textur a to přepočítání souřadnicového systému. Pro tahokov jsou použité dva pruhy jako u textury železné podlahy s tím rozdílem, že před vstupem souřadnic do funkce *smoothstep* přepočítány tak, aby se čtvercová síť zdeformovala do kosočtverců. Dále jsou použity již popsané techniky míchání barev a šumu.

```
float sul = snoise(v_texcoord * 50.0) * .3 + .7;
vec2 _v_texcoord = vec2(v_texcoord.x + v_texcoord.y*2, v_texcoord.x -
v_texcoord.y*2);
float pruh = smoothstep(.01, .02, abs(mod(_v_texcoord.x, .13) - .05));
float pruh2 = smoothstep(.01, .02, abs(mod(_v_texcoord.y, .13) - .05));
vec4 vnitrky = mix(color(0x999966),color(0x99B5BA),sul)*color(0xB2B2B2);
vec4 pruhy = mix(color(0xE0E0EB),color(0xC2C2D6),sul*1.5)*color(0xCCCCCC);
frag_color = mix(pruhy,  vnitrky, pruh * pruh2)*color(0xB2B2F0);
```

***Kód 5-10*** *Textura tahokovu*



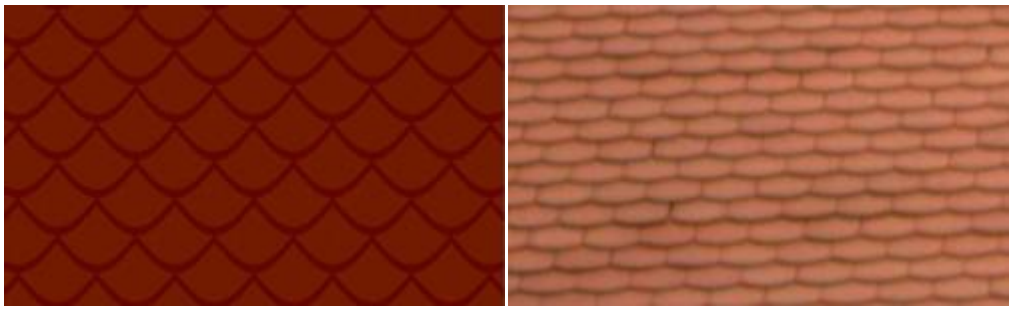
***Obrázek 5-11*** *Textura tahokovu*

## 5.11 Střešní tašky

Tašky jsou tvořeny vlnkami, které jsou navzájem o půl vlny posunuté. Posunutí bylo docíleno dvojím přepočítáním souřadnic, které jsou vstupem znovu do funkce *smoothstep*. Nejprve se použije sinus a poté cosinus, tyto dvě funkce jsou přesně o čtvrt periody posunuté. Aby byly vlnky jen na jednu stranu je na sinus i cosinus použita funkce *abs* pro vypočítání absolutní hodnoty. K jemnějším přechodům je využito filtru width. Aby tašky vypadaly přirozeně je do barvy přidán šum.

```
float sul = snoise(v_texcoord * 500.0) * .1 + .9;
float fun = v_texcoord.y + abs(.08 * sin(v_texcoord.x * 20.0 + 1.0));
float fun2 = v_texcoord.y + abs(.08 * cos(v_texcoord.x * 20.0 + 1.0));
float dd = fwidth(v_texcoord.y);
float waves = 1.0 - smoothstep(.01 - dd, .015, abs(mod(fun, .2) - .08)) +
              1.0 - smoothstep(.01 - dd, .015, abs(mod(fun2 + .1, .2) - .08));
vec4 tasky = color(0xD23300);
vec4 ohraniceni = color(0xA80900) + color(0x1A1A1A);
frag_color = mix(tasky*sul,ohraniceni , waves) * color(0x999999) ;
```

***Kód 5-11*** *Textura střešních tašek*



***Obrázek 5-12*** *Textura střešních tašek*

# 5.12 Dlažba

Kamenná dlažba se může ztvárnit více způsoby. V této práci jsou vybrány dva a nakonec použit pouze jen jeden. Nepoužitá dlažba je vytvořena pomocí deformace souřadnicového systému celulárním šumem. Různé barvy kamenů je docíleno tak, že hodnota pro míchání barev je vypočtena ze zaokrouhlené hodnoty aktuální souřadnice.

```
vec2 _v_texcoord =  v_texcoord + cellular(v_texcoord * 8.0) * .03 + .03;
float kachle = snoise(floor(_v_texcoord * 10.0)) * .5 + .5;
float sul = snoise(_v_texcoord * 300.0) * .3 + .7;
vec4 stone = color(0xA9A9DC);
vec2 ve_ctverci = mod(_v_texcoord * 10.0, 1.0) - .5;
float okraje = .95 - max(abs(ve_ctverci.x), abs(ve_ctverci.y));
vec4 spary = vec4(0.0);
spary = mix (vec4(1),spary , sul * .6 + .6);
frag_color = mix(spary, vec4(sul * kachle)*stone, smoothstep(.5, .6, okraje));
```

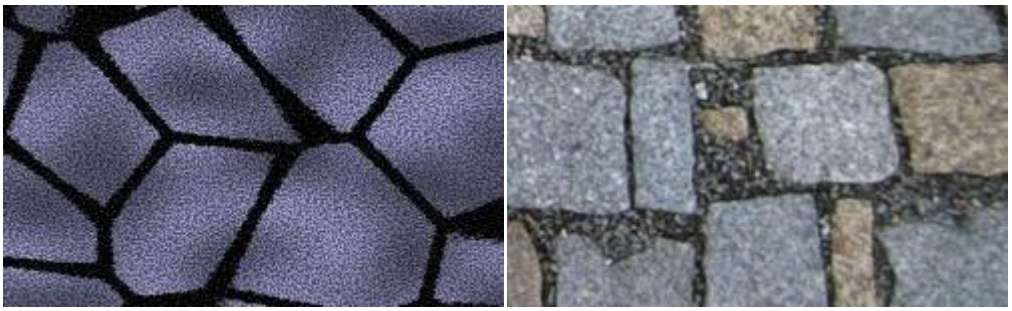
***Kód 5-12*** *Textura dlažby s různou barvou kamenů*

***Obrázek 5-13*** *Textura dlažby s různou barvou kamenů*

Druhá a použitá dlažba se skládá z nepravidelné sítě tvořené celulárním šumem doplněné Perlinovým šumem. Tímto jsou vygenerovány spáry složené z jemných kamínků. Tato síť je nanesena na strukturu kamenů. Textura je zvolena do modelu pro její větší členitost.

```
vec2 _v_texcoord =  v_texcoord + cellular(v_texcoord * 8.0) * .03 + .03;
float kachle = snoise(floor(_v_texcoord * 10.0)) * .5 + .5;
float sul = snoise(_v_texcoord * 300.0) * .3 + .7;
vec4 stone = color(0xA9A9DC);
vec2 ve_ctverci = mod(_v_texcoord * 10.0, 1.0) - .5;
float okraje = .95 - max(abs(ve_ctverci.x), abs(ve_ctverci.y));
vec4 spary = vec4(0.0);
spary = mix (vec4(1),spary , sul * .6 + .6);
frag_color = mix(spary, vec4(sul * kachle)*stone, smoothstep(.5, .6, okraje));
```

***Kód 5-13*** *Textura dlažby s větší členitostí*



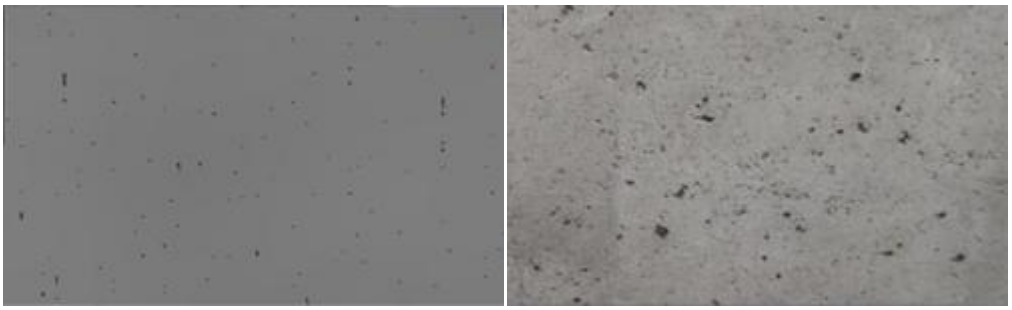
***Obrázek 5-14*** *Textura dlažby s větší členitostí*

# 5.13 Pohledový beton

Základem této textury je šedá barva s lehce se měnícím odstínem docíleným pomocí šumu. Do pozadí jsou vloženy nepravidelné oválné tvary (deformované elipsy), tvořené kombinací funkce *smoothstep* a celulárního šumu. Oválný tvar je vytvořen tak, že od vstupních parametrů je odečten celulární šum. Deformace se provede přidáním šumu k souřadnici y, která se poté používá pro generaci celulárního šumu. Dále je možné použít vypočtené hodnoty do funkce *mix* pro smíchání děr a podkladu do sebe.

```
float ruch1 = snoise(vec2(y,x))/10 + snoise(vec2(y,x)*5)/16;
y += ruch1;
vec2 F = cellular(vec2(x,y)*10);
vec4 holes = color(0x383838);
vec4 wall = color(0xCCCCCC);
float sul = snoise(v_texcoord * 100.0) * .3 + .4;
float ruch = snoise(vec2(x,y))/100;
float s = fwidth(F.x);
float n1 = smoothstep(0.1-s-ruch, 0.1+s-ruch, F.x*4)+ruch;
float n2 = smoothstep(0.2-s, 0.2+s, F.x*2);
wall += snoise(vec2(x,y))/100;
vec4 mezi = mix(holes,wall+ruch,n1);
frag_color = mix(mezi,mezi,n2) * color(0x999999);
```

***Kód 5-14*** *Textura pohledového betonu*



***Obrázek 5-15*** *Textura pohledového betonu*

# 5.14 Zeď na budově B

Nejsložitější textura v této práci. Kombinuje do této doby všechny použité techniky. Textura se skládá z podkladu, který obsahuje dva typy pruhů tvořenými funkcí *smoothstep*. Vstupními hodnotami jsou hrany přechodů a absolutní hodnota souřadnic. Tímto způsobem vytvoříme nepravidelné vlny. Dále jsou spočítány umístění a velikosti kostek. Přičtením šumu k souřadnici *y* dosáhneme proměnlivé výšky řádků kostek. Do mezer mezi kostkami je přidán podobně vytvořený pruh, jako v prvních dvou případech, jen orientovaný horizontálně. Nakonec jsou všechny části spojené v jednu.

```
vec2 _v_texcoord = v_texcoord * 5.0;
_v_texcoord.y += .5 * snoise(vec2(.0, _v_texcoord.y));
vec2 F = cellular(vec2(x,y)*10);
vec2  position, useBrick;
vec4  color;
float sul = snoise(_v_texcoord * 300.0) * .3 + .7;
float sul2 = snoise(_v_texcoord )* .2 + .9;
float pruh = smoothstep(.01, .015, abs(mod(_v_texcoord.y+sul2, .08) - .08))+sul;
float pruh2 = smoothstep(.001, .02, abs(mod(_v_texcoord.y*1.5+sul2, .08) -
.08))+sul;
float pruh3 = smoothstep(.001, .02, abs(mod(_v_texcoord.x*6+sul2, .08) - .08))+sul;
vec4 prvni_barva = color(0x8D6277) * color(0xE6E6E6);
vec4 druha_barva = mix(vec4(.45), prvni_barva, pruh2);
vec4 treti_barva = color(0x33291F)*color(0x5C1F3D);
vec4 BrickColor = druha_barva+treti_barva/pruh ;
vec2 BrickPct = vec2(0.75, 0.9);
vec2 BrickSize = vec2(0.5, 0.3);
vec3 LightPosition = vec3(0, 0, 4);
vec4 MortarColor = mix(vec4(.45), prvni_barva, pruh3) * color(0x999999);
position = _v_texcoord / BrickSize;
 if (fract(position.y * 0.5) > 0.5)
       position.x +=0.5;
position = fract(position);
useBrick = step(position, BrickPct);
frag_color = mix(MortarColor, BrickColor, useBrick.x * useBrick.y) *
color(0x999999);
```

***Kód 5-15*** *Textura zdi na budově B*

***Obrázek 5-16*** *Textura zdi na budově B*

# 5.15 Nebe

Jediná textura, pro kterou byl využit 3D šum. Pro vytvoření mraků se jednotlivé šumy sčítají a tím je docíleno tmavších částí mraků uprostřed při větší oblačnosti. Intenzita oblačnosti se nastavuje do proměnné přímo v kódu shaderu.

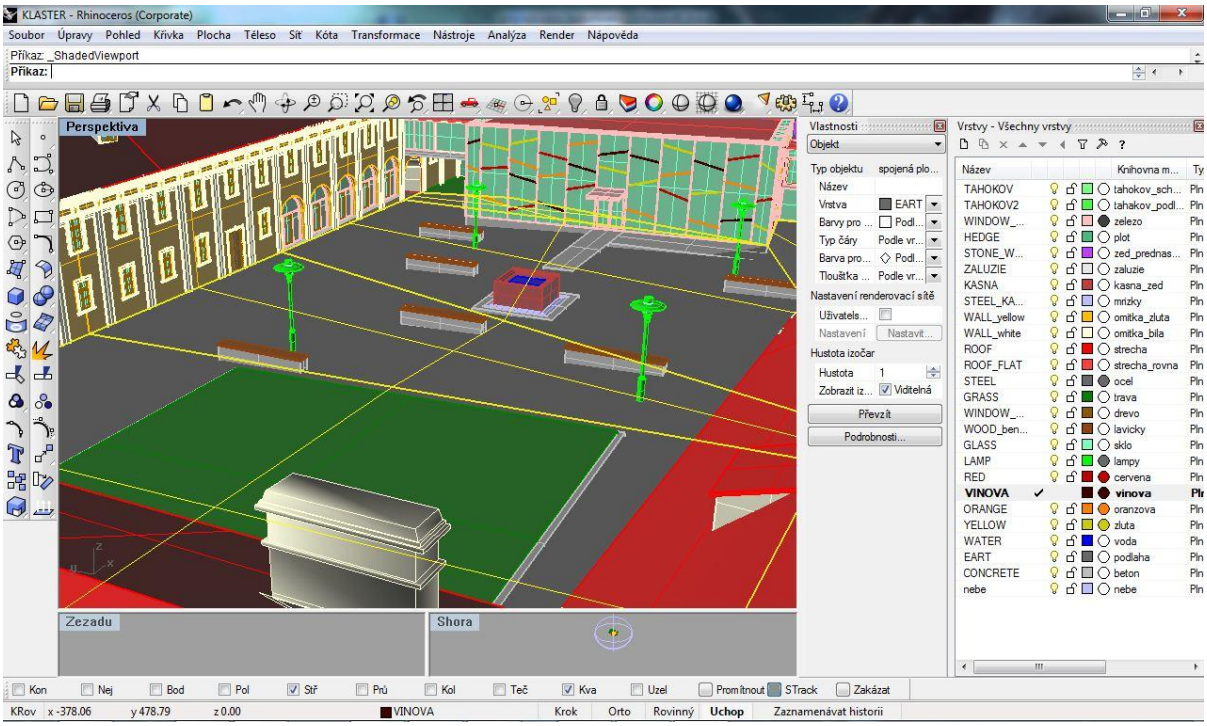
```
const vec3 SkyColor = vec3(0.1, 0.2, 0.8);
const vec3 CloudColor = vec3(0.8, 0.8, 0.8);
const vec3 CloudColorDark = vec3(0.6, 0.6, 0.6);
const float base_freq = 0.5;
float noise = snoise(v_texcoord3D * base_freq * 2.0) * 4.0 + snoise(v_texcoord3D *
base_freq*4.0) * 2.0 + snoise(v_texcoord3D * base_freq * 8.0);
noise = (noise / 8.0 + 1.0) / 2.0;
noise -= 0.5;
noise *= 1.8;
float intensity = noise;
vec3 color = vec3(0.0, 0.0, 0.0);
intensity = clamp(intensity, 0.0, 1.0);
   if (intensity < 0.5)
        color = mix(SkyColor, CloudColor, intensity*2.0);
   else
        color = mix(CloudColor, CloudColorDark, (intensity - 0.5)*2.0);
frag_color = vec4(color, 1.0);
```

***Kód 5-16*** *Textura nebe*

***Obrázek 5-17*** *Textura nebe*

# 6 Demonstrační model

Pro vytvoření modelu areálu Božetěchova v Brně, byl vybrán grafický software Rhinoceros. Zvolený byl pro jednoduchou práci při tvorbě 3D modelů budov. Základem je model areálu získaný se souhlasem autora ze služby Google Earth, vytvořený v programu Google SketchUp. Model byl konvertován a načten do prostředí programu Rhinoceros, aby mohl být doplněn o další části. Původní model obsahoval pouze hrubou stavbu bez oken, dveří, říms a s prázdným nádvořím. Všechny tyto části byly domodelovány. V grafickém softwaru byly na jednotlivé plochy nastavené typy materiálu. Poslední využitou funkcí grafického softwaru bylo nastavení orientace a škálování jednotlivých textur.

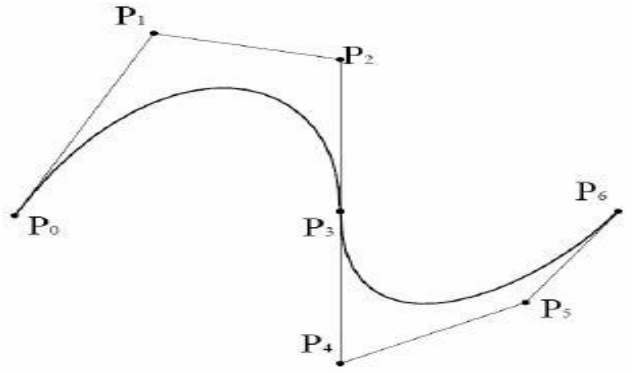



***Obrázek 6-1*** *Ukázka modelu v programu Rhinoceros*

Takto vytvořený model byl uložen ve formátu 3ds, který je typický pro software 3D Max. Výhodou tohoto formátu je možnost rozložení polygonů na trojúhelníky a uložení informací o jednotlivých materiálech a k nim přiřazených skupin objektů. Následně ve vytvořeném programu jsou textury vždy aplikované na objekty dle daného materiálu.

Pro zobrazení modelu je vytvořen program napsaný v jazyce C++, který je součástí přílohy této práce. Program využívá grafickou knihovnu OpenGL. Aplikace umožňuje uživateli pohyb po modelu pomocí kláves *W,S,A,D* a pro ukázku je vytvořeno malé demo pohybu po modelu spouštěné klávesou *P*. Demo využívá pro pohyb ve scéně křivek, které tvoří trasu kamery a její natočení. Výhodou použití křivek k vytvoření dráhy kamery je jejich malá paměťová náročnost, jelikož nám stačí uložit

pro jednu křivku pouze její řídící body. S tímto souvisejí další výhody jako je jednoduchá práce s křivkami. Oproti pohybu po přímkách nemusíme řešit zaoblení hran a kamera se pohybuje plynule. V našem případě jsou využity Beziérovy kubiky, které jsou reprezentovány čtyřmi řídícími body.




***Obrázek 6-2*** *Beziérové kubiky*

# 7 Závěr

Zadáním této práce bylo prostudovat postup a technologie potřebné pro tvorbu procedurálních textur v shaderu. Následně bylo zadané vytvořit sadu shaderů materiálu vyskytujících v areálu Božetěchova Fakulty informačních technologií VUT v Brně. Výsledky se měly demonstrovat na modelu tohoto areálu jednoduchým grafickým demem a stručným plakátkem.

Základem pro vytvoření této práce bylo nastudování základních principu texturování. V této části byly vysvětleny charakteristiky různých druhů textur. Důležitými částmi pak byly kapitoly pojednávající o mapovaní textur a vzniku i potlačení aliasu.

Následně bylo nutné nastudovat grafické rozhraní OpenGL a jeho rozšíření. Zvláště pak byla věnována pozornost pro OpenGL 3.0 u které bylo nutné pochopení principu vykreslování a reprezentace modelů. Tato knihovna byla nakonec využita k tvorbě shaderu pomocí shaderovacího jazyku GLSL.

Poslední teoretickou částí bylo nastudování a popsání metod pro generování procedurálních textur. Základem těchto metod je tvorba šumů, zvláště pak Perlinova šumu. Perlinův šum je v počítačové grafice hojně využíván nejen v texturování, ale i v generování terénu. Proto bylo velmi užitečné pochopení jeho principu i pro další práce.

Při vytváření modelu a reálných shaderů byly získány zkušenosti nejen z oblasti programování, ale i s grafickou úpravou obrazu a s grafickým softwarem pro tvorbu modelů Rhinoceros. Tento software mi byl doporučen díky jeho intuitivnímu ovládání a rozsáhlými modelovacími prostředky.

Pro demonstraci bylo vymodelováno pouze nádvoří areálu s okolními budovami. Pro rozšíření vytvořené aplikace by bylo možné vymodelovat i okolní budovy a jejich interiér. Tím bychom mohli vytvořit detailní 3D model areálu, například pro jeho virtuální prohlídku. Shadery použité v aplikaci mají velkou míru realičnosti, ale mohli by být v budoucnu vylepšené osvětlením a využitím technik pro simulaci hrubosti povrchů například použitím Bum-mapingu. Dalším vylepšením by mohlo být využití ostatních druhů shaderů, jako vertex shaderu pro vytvoření animace vlnění vody ve fontáně.

Využívání procedurálních textur je velice užitečný nástroj. Přináší s sebou spoustu výhod, zmiňovaných v této práci, jako malou velikost, možnost parametrizace nebo neomezené rozlišení. Největším problémem zůstává řešení aliasingu a poměrně náročný postup vytváření kódu pro generování procedurálních textur.

# Literatura


[1] ANGEL, Ed a Dave SHREINER. *An Introduction to Modern OpenGL Programming* [online]. SIGGRAPH, 2011 [cit. 2012-05-07].
Dostupné z: http://www.scribd.com/doc/75639730/Modern-OpenGL

[2] SEGAL, Mark a Kurt AKELEY. A Specification Version 4.2. *The OpenGL R Graphics System* [online]. August 8, 2011 [cit. 2012-05-07].
Dostupné z: http://www.opengl.org/registry/doc/glspec42.core.20110808.pdf

[3] ROST, Randi J. *OpenGL shading language*. 3rd ed. Upper Saddle River: Addison-Wesley, c2010, 743 s. [cit. 2012-05-07]. ISBN 978-0-321-63763-5.

[4] ŽÁRA, Jiří, Bedřich BENEŠ, Jiří SOCHOR a Petr FELKEL. *Moderní počítačová grafika* Vyd 2. Brno: Computer Press, 2004, 609 s. [cit. 2012-05-07]. ISBN 80-251-0454-0.

[5] EBERT, David S., F. Kenton MUSGRAVE, Darwyn PEACHEY, Ken PERLIN a Steven WORLEY. *Texturing and modeling*: *A Procedural approach*. Vyd. 3. San Francisco: Morgan Kaufmann Publishers, 2003, 687 s. [cit. 2012-05-07]. ISBN 15-586-0848-6.

[6] PERLIN, Ken. MAKING NOISE. [online]. [cit. 2012-05-07].
Dostupné z: http://www.noisemachine.com/talk1/index.html

[7] Autodesk 3ds Max Help: UVW Map Modifier. [online]. [cit. 2012-05-07].
Dostupné z:
http://docs.autodesk.com/3DSMAX/13/ENU/Autodesk%203ds%20Max%202011%20Help/index.html?url=./files/WSf742dab04106313366400bf6112a1cea097-7ea6.htm,topicNumber=d0e94531

[8] Making Cellular Textures. [online]. [cit. 2012-05-07].
Dostupné z: http://www.blackpawn.com/texts/cellular/default.html

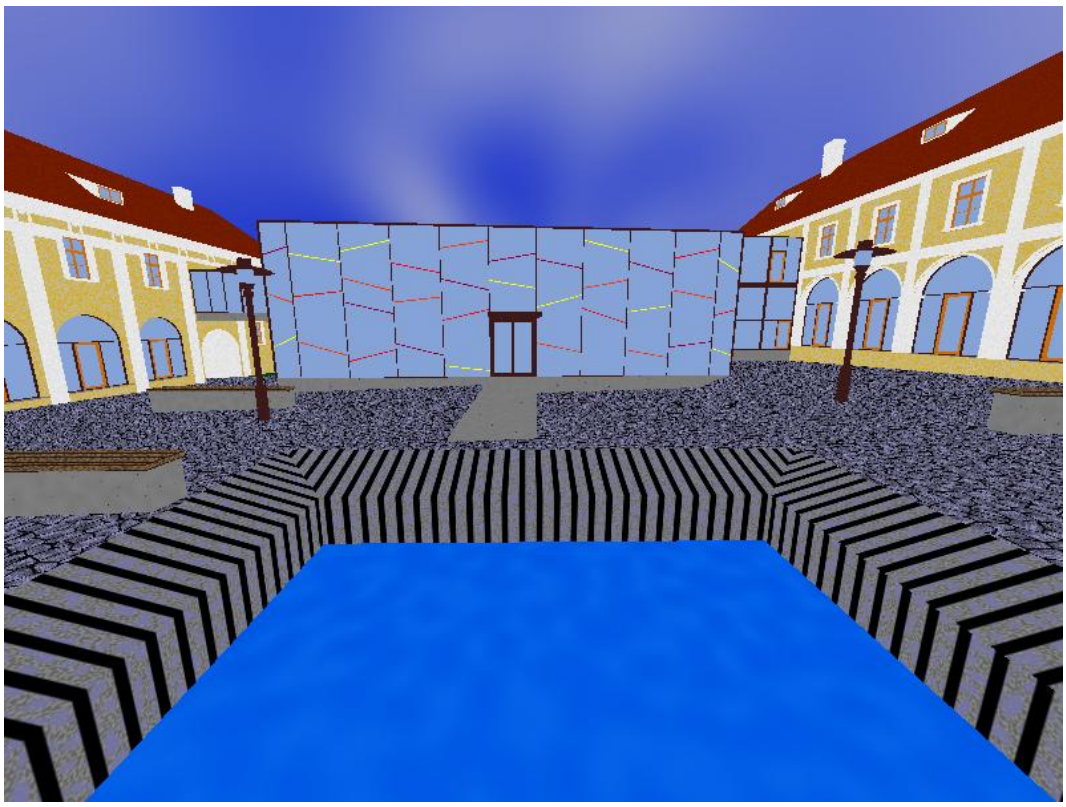
[9] KARCZMARCZUK, Jerzy. *Functional Approach to Texture Generation* [online]. University of Caen, France [cit. 2012-05-07].
Dostupné z: https://users.info.unicaen.fr/~karczma/arpap/texturf.pdf

[10] *Rhinoceros Level 1 Training Manual*: *NURBS modeling for Windows* [online]. Robert McNeel & Associates, 2008 [cit. 2012-05-07]. v4.0.
Dostupné z: http://www.andrew.cmu.edu/course/48-125/IDM2/HANDOUTS_files/Rhino%20Level%201%20v4.pdf

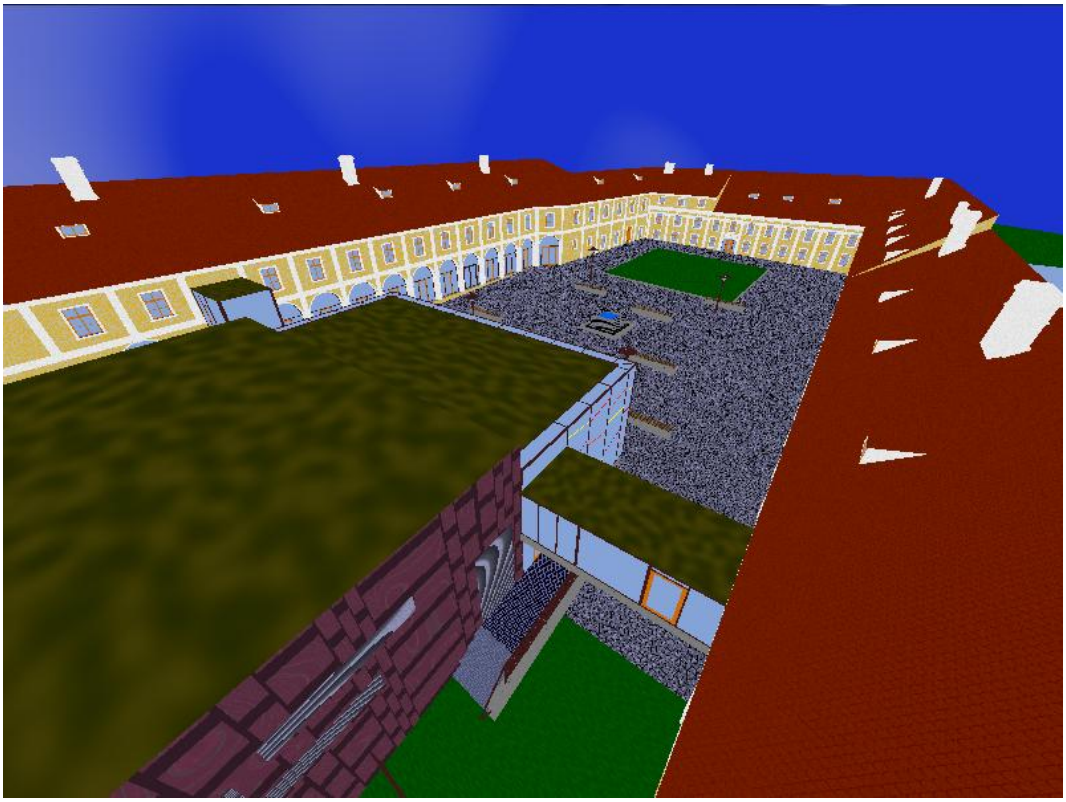
# Seznam obrázků

# Seznam příloh

# Příloha 1

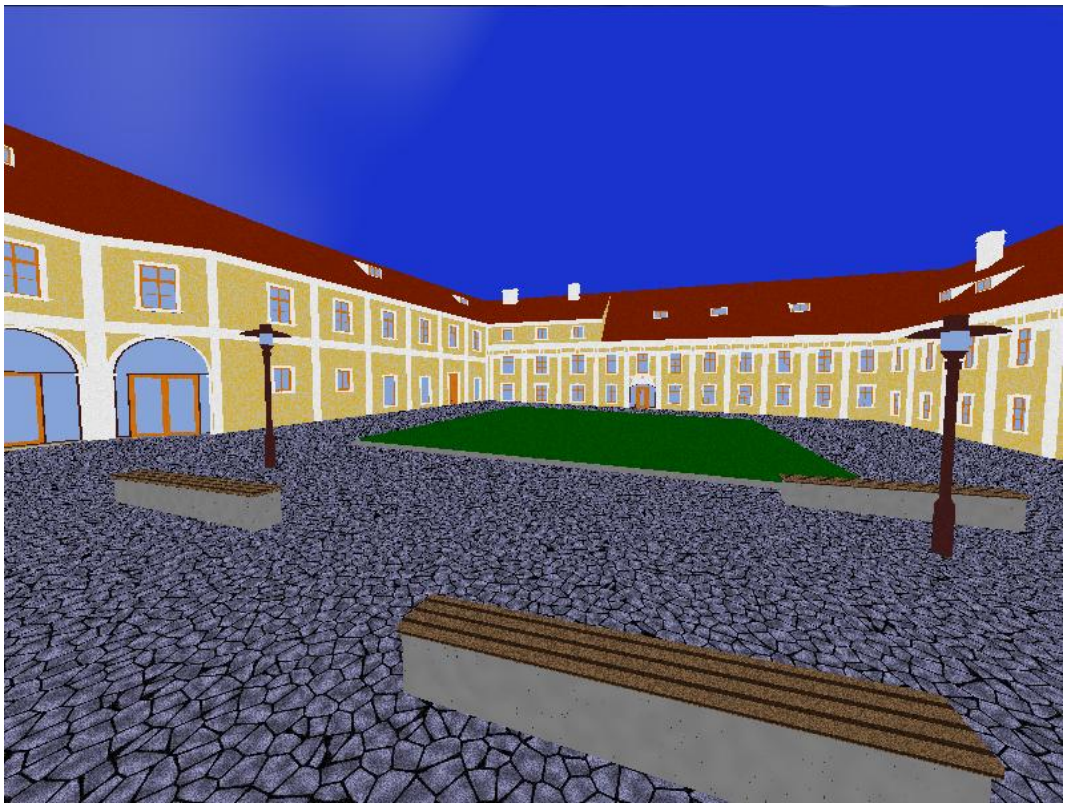


*Pohled na budovu B*



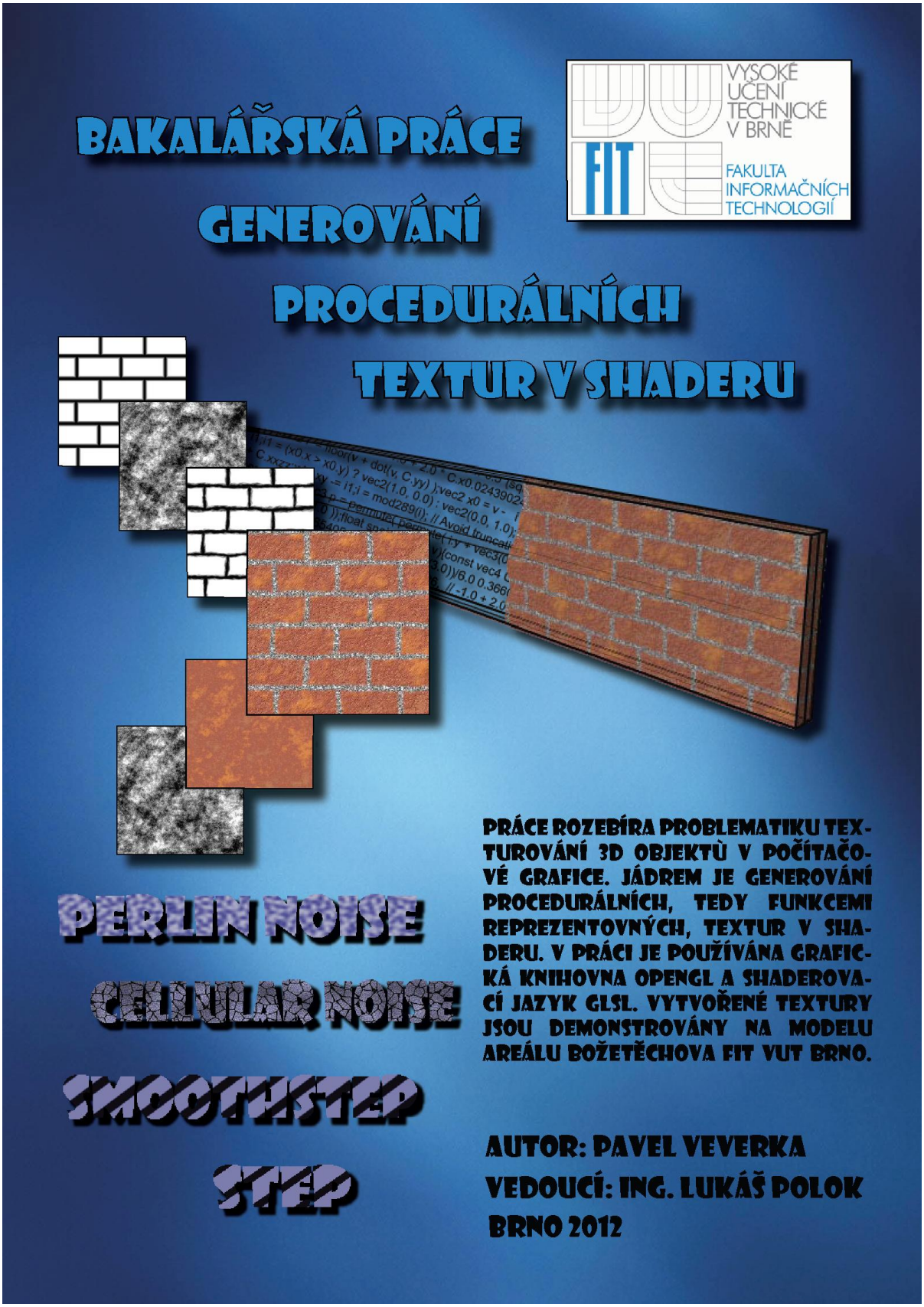
*Pohled na celý areál Božetěchova*

*Průchod u budovy B*



*Pohled směrem od budovy B*

# Příloha 2


BAKALÁŘSKÁ PRÁCE
GENEROVÁNÍ
PROCEDURÁLNÍCH
TEXTUR V SHADERU
VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
FIT
FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
PERLIN NOISE
CELLULAR NOISE
SMOOTHSTEP
STEP
PRÁCE ROZEBÍRA PROBLEMATIKU TEXTUROVÁNÍ 3D OBJEKTŮ V POČÍTAČOVÉ GRAFICE. JÁDREM JE GENEROVÁNÍ PROCEDURÁLNÍCH, TEDY FUNKCEMI REPREZENTOVNÝCH, TEXTUR V SHADERU. V PRÁCI JE POUŽÍVÁNA GRAFICKÁ KNIHOVNA OPENGL A SHADEROVACÍ JAZYK GLSL. VYTVOŘENÉ TEXTURY JSOU DEMONSTROVÁNY NA MODELU AREÁLU BOŽETĚCHOVA FIT VUT BRNO.
AUTOR: PAVEL VEVERKA
VEDOUCÍ: ING. LUKÁŠ POLOK
BRNO 2012