Bakalářská práce

České vysoké
učení technické
v Praze

F3

Fakulta elektrotechnická
Katedra počítačové grafiky a interakce

Knihovna pro vykreslování textu v OpenGL

Karel Tomanec

Vedoucí práce: Ing. Jaroslav Sloup
Obor: Otevřená informatika
Studijní program: Počítačové hry a grafika
Květen 2020

ZADÁNÍ BAKALÁŘSKÉ PRÁCE

ČVUT
ČESKÉ VYSOKÉ UČENÍ TECHNICKÉ V PRAZE

I. OSOBNÍ A STUDIJNÍ ÚDAJE

Příjmení: **Tomanec** Jméno: **Karel** Osobní číslo: **478155**

Fakulta/ústav: **Fakulta elektrotechnická**

Zadávající katedra/ústav: **Katedra počítačové grafiky a interakce**

Studijní program: **Otevřená informatika**

Studijní obor: **Počítačové hry a grafika**

II. ÚDAJE K BAKALÁŘSKÉ PRÁCI

Název bakalářské práce:

Knihovna pro vykreslování textu v OpenGL

Název bakalářské práce anglicky:

Font Rendering in OpenGL

Pokyny pro vypracování:

Seznamte se s existujícími formáty pro popis fontů (TTF,OTF) a metodami jejich vykreslování [1-5]. Proveďte srovnání těchto metod (princip, výhody, nevýhody) a na jeho základě vyberte alespoň dvě metody vhodné pro implementaci pomocí grafické knihovny OpenGL. Metody implementujte jako knihovnu, která bude nejenom vykreslovat požadované znaky, ale zajistí i správné vzájemné umístění znaků při vykreslování řetězců. Knihovna bude součástí tzv. PGR-frameworku používaného v předmětu Programování grafiky pro tvorbu semestrálních projektů. Vytvořte ukázkovou aplikaci, která demonstruje funkčnost a použití implementované knihovny. Implementované metody porovnejte z hlediska rychlosti, pamětové složitosti a kvality/přesnosti vykreslení.
Implementaci proveďte v C/C++ a OpenGL.

Seznam doporučené literatury:

[1] Eric Lengyel, GPU-Centered Font Rendering Directly from Glyph Outlines, Journal of Computer Graphics Techniques (JCGT), Vol. 6, No. 2, 31-47, 2017.
[2] M.J. Kilgard, J. Bolz : GPU-accelerated Path Rendering. ACM Transactions on Graphics, Vol.31, No.6, 172:1-172:10, November 2012.
[3] C. Loop, J. Blinn: Resolution Independent Curve Rendering using Programmable Graphics Hardware. ACM Transactions on Graphics (TOG), Vol.24, 1000-1009, 2005.
[4] C. Loop, J. Blinn: Rendering Vector Art on the GPU. GPU Gems 3, 543-562, 2007.
[5] B. Esfahbod: GLyphy - high-quality glyph rendering using OpenGL ES2 shading language. https://glyphy.org, 2014.

Jméno a pracoviště vedoucí(ho) bakalářské práce:

Ing. Jaroslav Sloup, Katedra počítačové grafiky a interakce

Jméno a pracoviště druhé(ho) vedoucí(ho) nebo konzultanta(ky) bakalářské práce:

Datum zadání bakalářské práce: **12.02.2020** Termín odevzdání bakalářské práce: ______________

Platnost zadání bakalářské práce: **30.09.2021**

Ing. Jaroslav Sloup podpis vedoucí(ho) práce	podpis vedoucí(ho) ústavu/katedry	prof. Mgr. Petr Páta, Ph.D. podpis děkana(ky)

© ČVUT v Praze, Design: ČVUT v Praze, VIC

III. PŘEVZETÍ ZADÁNÍ

Student bere na vědomí, že je povinen vypracovat bakalářskou práci samostatně, bez cizí pomoci, s výjimkou poskytnutých konzultací.
Seznam použité literatury, jiných pramenů a jmen konzultantů je třeba uvést v bakalářské práci.

.	
Datum převzetí zadání	Podpis studenta

© ČVUT v Praze, Design: ČVUT v Praze, VIC

Poděkování

Rád bych poděkoval vedoucímu své práce Ing. Jaroslavu Sloupovi za vedení, odbornou konzultaci a rady při zpracování této práce. Zároveň chci také poděkovat své rodině a přátelům za jejich podporu v průběhu studia.

Prohlášení

Prohlašuji, že jsem předloženou práci vypracoval samostatně, a že jsem uvedl veškerou použitou literaturu.

V Praze, 20. května 2020

Abstrakt

Tato bakalářská práce se zabývá problematikou vykreslování textu a vývojem knihovny pro jeho vykreslování založené na OpenGL. Představuje základní typografické koncepty, existující formáty pro popis fontů a metody jejich vykreslování. Součástí práce byla podrobnější analýza dvou vybraných metod a jejich implementace v podobě knihovny. Výsledky implementace obou metod byly v závěru porovnány z hlediska rychlosti, paměťové složitosti a přesnosti vykreslování.

Klíčová slova: vykreslování textu, knihovna, Bézierovy křivky, nezávislost na rozlišení, OpenGL

Vedoucí práce: Ing. Jaroslav Sloup

Abstract

This thesis deals with the topic of text rendering and the development of the text rendering library based on OpenGL. The thesis introduces basic typographic concepts, existing font formats, and text rendering methods. Part of the thesis was a detailed analysis of two selected methods and their implementation in the form of a library. The results of the implementation were compared in terms of speed, space complexity, and accuracy of rendering.

Keywords: text rendering, library, Bézier curves, resolution independence, OpenGL

Title translation: Font Rendering in OpenGL

Obsah

1 Úvod **1**

2 Typografie **3**
2.1 Font . . . 3
2.2 Glyf . . . 3
2.3 Metrika . . . 4
2.4 Standardy pro popis fontů . . . 5
2.4.1 TrueType . . . 6
2.4.2 OpenType . . . 6

3 Křivky **7**
3.1 Parametrické křivky . . . 7
3.2 Bézierovy křivky . . . 7

4 Metody vykreslování textu **11**
4.1 Metody využívající předrenderované glyfy . . . 11
4.2 Metody využívající vzdálenosti . . . 12
4.3 Geometricky založené metody . . . 13
4.4 Knihovny pro vykreslování textu . . . 14
4.4.1 FreeType . . . 15
4.4.2 Slug . . . 15

5 Analýza vybraných metod **17**
5.1 Metoda vykreslování kvadratických a kubických křivek nezávisle na rozlišení (Loop, Blinn) . . . 17
5.1.1 Vykreslování kvadratických křivek . . . 18
5.1.2 Vykreslování kubických křivek 19
5.1.3 Triangulace . . . 23
5.1.4 Antialiasing . . . 25
5.1.5 Paměťová složitost a rychlost 26
5.2 Metoda přímého vykreslování textu (Lengyel) . . . 27
5.2.1 Výpočet navíjecího čísla . . . 27
5.2.2 Antialiasing . . . 30
5.2.3 Optimalizace . . . 32
5.2.4 Paměťová složitost a rychlost 33

6 Návrh řešení a implementace **35**
6.1 Návrh řešení . . . 35
6.2 Implementace . . . 36
6.2.1 Metoda vykreslování kvadratických a kubických křivek nezávisle na rozlišení (Loop, Blinn) 36
6.2.2 Metoda přímého vykreslování textu (Lengyel) . . . 42
6.2.3 Knihovna . . . 46

7 Výsledky **49**
7.1 Metoda vykreslování kvadratických a kubických křivek nezávisle na rozlišení (Loop, Blinn) . . . 49
7.2 Metoda přímého vykreslování textu (Lengyel) . . . 52
7.3 Porovnání výsledků obou metod 55
7.4 Ukázková aplikace . . . 56

8 Závěr **59**
8.1 Budoucí práce a rozšíření . . . 59

Bibliografie **61**

A Seznam zkratek **65**

B Obsah přiložených souborů **67**

Obrázky

1.1 Ukázka vykreslení písma s aliasem, bez aliasu a v perspektivě. 1

2.1 Anatomie glyfu. 4
2.2 Metrický systém používaný v typografii. Překresleno z [12]. 5

3.1 Bézierova křivka prvního a druhého stupně. 8
3.2 Bézierovy křivky třetího stupně. . 8

4.1 Glyfy uložené v atlasu textur vygenerované nástrojem Font Texture Generator [21]. 12
4.2 Glyfy se vzdáleností v alfa kanálu uložené v atlasu textur vygenerované nástrojem Font Texture Generator [21]. 13
4.3 Porovnání glyfu vykresleného z textury o vysokém rozlišení (nahoře) a glyfu vykresleného z textury uchovávající pole vzdáleností o menším rozlišení pomocí bilineárního filtrování (dole). Převzato z [18]. . . 13
4.4 Text vykreslený metodou Lengyel [11]. 14

5.1 Tři typy klasifikace kubické Bézierovy křivky. Překresleno z [15] 19
5.2 Řídící polygon kubické Bézierovy křivky s texturovacími souřadnicemi. 20
5.3 Možný artefakt u křivky typu loop. 23
5.4 Odstranění překrývajících se trojúhelníků. 24
5.5 Tři možné triangulace řídícího polygonu kubické Bézierovy křivky. 25
5.6 Vizualizace výpočtu navíjecího čísla (*winding number*). Překresleno z [12]. 29
5.7 Výpočet částečného pokrytí pixelu glyfem. Překresleno z [12] 32
5.8 Glyf rozdělený na horizontální a vertikální pásy. Překresleno z [11]. 33

6.1 Glyf vykreslený metodou Loop/Blinn bez použití antialiasingu s barevně odlišenými komponentami. 41
6.2 Struktura textury uchovávající seznam křivek. Převzato z [11] 42
6.3 Struktura textury uchovávající data pro pásy. Převzato z [11] 43

7.1 Ukázka různých fontů vykreslených metodou Blinn/Loop. 49
7.2 Detail glyfu vykreslený metodou Loop/Blinn. 50
7.3 Porovnání diskrétního výstupu s aliasem a výstupu bez aliasu dosaženého kombinací použití MSAA a výpočtu vzdálenosti od křivky. . . 50
7.4 Ukázka různých fontů vykreslených metodou Lengyel. 52
7.5 Detail glyfu vykreslený metodou Lengyel. 53
7.6 Text v perspektivě vykreslený metodou Lengyel. 54
7.7 Ukázka aplikace demonstrující možné použití knihovny. 56
7.8 Odstavec textu vykreslený v perspektivním zobrazení v ukázkové aplikaci. 57
7.9 Text při použití kerningu. 57

Tabulky

5.1 Tabulka tříd ekvivalence pro klasifikaci kvadratických Bézierových křivek. Převzato z [11]. 31

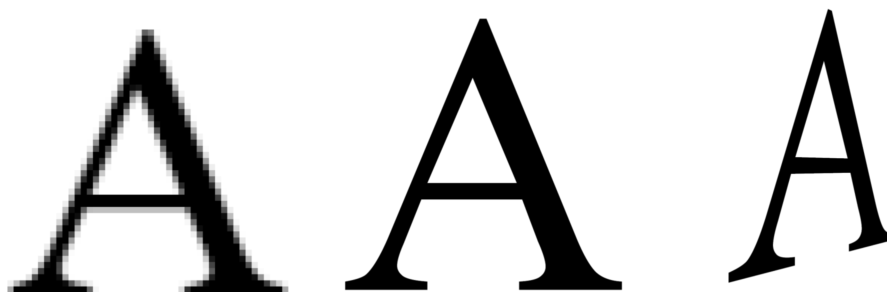
7.1 Měření doby vykreslování textu metodou Loop/Blinn. 51
7.2 Měření prostorové složitosti textu metody Loop/Blinn. 52
7.3 Měření doby vykreslování textu metody Lengyel. 54
7.4 Měření prostorové složitosti textu metodou Lengyel. 55

Kapitola 1

Úvod

Denně přicházíme do kontaktu s textovým vyjádřením informací, ať už psanou, tištěnou nebo digitálně zobrazenou formou. Vykreslování textu je v počítačové grafice velmi rozsáhlým tématem a za dobu existence počítačů lidé vyvinuli několik metod a technik jak text ukládat a zobrazovat. Při statickém renderování lze text vykreslit s vysokou přesností. Jakmile ale na text aplikujeme transformace nebo neustále se měnící záběr perspektivní kamerou, musí se použít specializované metody, které si poradí s vykreslováním v reálném čase.

Cílem této práce je nejprve prostudování existujících formátů pro popis fontů a metod jejich vykreslování. Dále provést srovnání těchto metod, na jehož výsledku se vyberou dvě metody nejvhodnější pro implementaci pomocí grafické knihovny OpenGL. Tyto metody budou implementované jako knihovna, která bude nejenom vykreslovat požadované znaky, ale také zajistí správné vzájemné umístění znaků při vykreslování řetězců. Tato knihovna bude také součástí PGR-frameworku používaného v předmětu Programování grafiky pro tvorbu semestrálních projektů. Dalším cílem je vytvoření ukázkové aplikace, která bude demonstrovat funkčnost a použití implementované knihovny. Implementované metody budou porovnané z hlediska rychlosti, paměťové složitosti a kvality vykreslení. Cílem je vybrat a implementovat metody, které vykreslují text bez aliasu a nezávisle na rozlišení ve 3D scéně při záběru perspektivní kamery 1.1.



Obrázek 1.1: Ukázka vykreslení písma s aliasem, bez aliasu a v perspektivě.

Tato práce je strukturována celkem do osmi kapitol. V kapitole 2 jsou

představeny základní typografické pojmy, metrika v typografii a dva nejpoužívanější standardy pro popis fontů. Kapitola 3 popisuje základní vlastnosti parametrických a Bézierových křivek, které jsou potřebné k analýze vybraných metod vykreslování textu. V kapitole 4 jsou popsány a porovnány metody vykreslování textu, které jsou rozděleny do tří kategorií, podle principu dle kterého fungují. Jsou zde také porovnány existující knihovny pro vykreslování textu. V kapitole 5 je provedena teoretická analýza principů fungování dvou metod, které byly vybrány na základě porovnání v předchozí kapitole. Kapitola 6 popisuje implementační detaily dvou vybraných metod a architekturu knihovny, do které jsou zkompletovány. V kapitole 7 jsou představeny výsledky implementací obou metod a výsledky měření rychlosti vykreslování. Obě metody jsou zde také vzájemně porovnány. Kapitola 8 shrnuje výsledky práce, zda se podařilo dosáhnout stanovených cílů a jakým způsobem může být práce rozšířena v budoucnosti.

Kapitola 2

Typografie

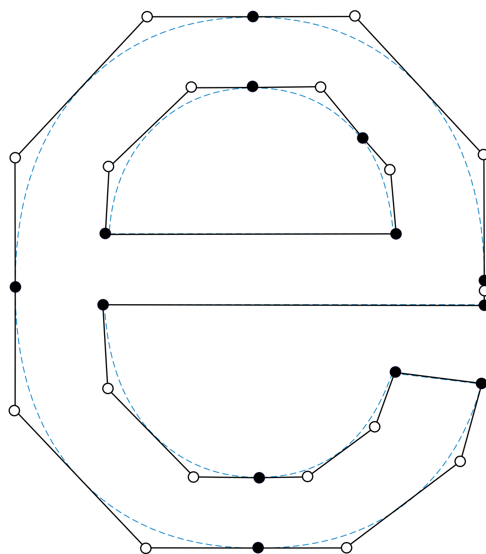
Typografie je umělecko-technický obor, který se zabývá písmem. Umělecká část této disciplíny se zabývá především tvorbou písma, jejím výběrem, správným umístěním na stránku, použitím a sazbou. Jejím cílem je zajistit čtenáři jednodušší čtení a efektivní vnímaní čteného textu. Pro technickou část je typografie proces, ve kterém je řetězec znaků transformován na glyfy, které jsou poté zobrazeny grafickým médiem. Obecně tento proces zahrnuje aplikování fontů, velikostí, pozicování a dalších vlastností, které jsou specifikovány uživatelem. Na nižší úrovni ovšem řeší spoustu dalších úprav jako jsou jemné úpravy pozic glyfů, složení ligatur, umístění akcentů ke znakům nebo substituce za stylistické alternativy.

2.1 Font

Font neboli počítačové písmo je ucelená sada znaků abecedy jednotného stylu. Znaky stejného fontu sdílejí stejné vlastnosti jako vzhled, tloušťku nebo serif. Obsahuje také údaje o metrice, glyfech a o dalších informacích, které jsou potřeba ke správnému zobrazení písma. Jé důležité uvést také související pojem *font face* neboli řez písma. Font se může skládat z několika řezů, jako například Regular, Plain nebo Roman, které vychází ze stejného vzoru. Tento vzor se označuje jako *font family*. Často se oba tyto pojmy označují slovem font v závislosti na kontextu [17].

2.2 Glyf

Glyph neboli glyf je grafická realizace podoby znaku. Může reprezentovat jeden znak (písmeno, číslo) nebo více znaků v případě ligatur (slití více písmen). Také může reprezentovat diakritické znaménko, které je přidáno k jinému znaku. Každý font se skládá z množiny glyfů, kterým jsou přiřazena jednotlivá čísla podle typu kódování (ASCII, Unicode) [17].



Obrázek 2.1: Anatomie glyfu.

Glyf se skládá z množiny jedné nebo více uzavřených kontur. Každá z kontur se dále skládá ze sekvence bodů definující úsečky nebo křivky jak lze vidět na obrázku 2.1. Černé body reprezentují krajní body těchto úseček nebo krajní řídící body křivek a leží vždy na kontuře. Bílé body jsou řídící body křivek, kterými křivka nemusí nutně procházet a určují její tvar.

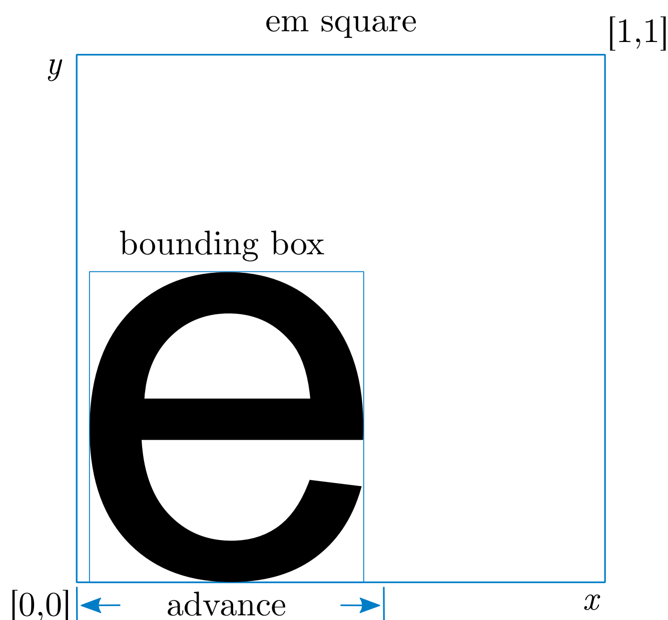
2.3 Metrika

Pozice každého řídícího bodu glyfu je vyjádřena relativně vzhledem k boxu nazývaném *em square*. Jak je vidět na obrázku 2.2, em square využívá pravoúhlý souřadnicový systém s osou x směřující doprava a s osou y směřující nahoru a jeho hranice nabývá hodnot 0 do 1 v obou souřadnicích. Počátek tohoto systému reprezentuje začátek psaného textu. Jestliže je text psán velikostí S, tak všechny body v jsou uniformě zvětšeny podle S.

Uvnitř em square je obsažena většina glyfů, ale existují glyfy, u kterých některé řídící body křivek mohou ležet mimo něj. Řídící body některých glyfů mají často negativní souřadnice y (například písmena „j“, „g“ nebo „p“ ve fontu arial) a existují také glyfy, které jsou širší nebo vyšší než em square [17].

Nejmenší obdélník, který obsahuje každý bod glyfu se nazývá *bounding box*. Je běžné, že bounding box začíná na souřadnici x s mírnou vzdáleností od počátku em square.

Pro každý glyf definuje font hodnotu *advance*. Tato hodnota určuje jak daleko má být vykreslený následující glyf v řetězci. Advance je obvykle o něco širší, než je šířka bounding boxu, ale není to pravidlem. Standardně je advance



Obrázek 2.2: Metrický systém používaný v typografii. Překresleno z [12].

definován pro horizontální zápis řetězce, může být ovšem definován také pro vertikální zápis řetězce [17].

2.4 Standardy pro popis fontů

Existují dva základní principy tvorby fontů a to bitmapové a vektorové. Záznam tvaru znaku může být tedy ukládán v těchto dvou podobách.

Bitmapové fonty jsou pevně vykresleny v dané mřížce a pro každou další požadovanou velikost musí existovat další varianta této mřížky. Jejich výhodou je jednoduchost a rychlost vykreslování. Tyto fonty jsou vhodné pro displeje s nízkou kvalitou obrazu nebo menších velikostí a stále se používají například v příkazovém řádku, v displejích vestavěných systémů nebo u jehličkových tiskáren.

Vektorové fonty mají tvar znaku definovaný pomocí křivek. Tyto fonty jsou díky matematickému popisu nezávislé na rozlišení a mohou být libovolně zvětšeny nebo zmenšeny beze změny kvality. Vektorové písmo ovšem musí být na konci převedeno do pixelové formy procesem zvaným rasterizace, protože všechny displeje, na kterých je písmo dále zobrazeno, pracují s konečným počtem zobrazovacích bodů.

V této chvíli existují dva nejpoužívanější standardy pro popis fontů a to

TrueType a OpenType.

2.4.1 TrueType

Tento standard pro popis vektorových písem byl vyvinutý koncem 80. let společnostmi Microsoft a Apple jako konkurence k fontům Adobe Type 1. Primárním cílem TrueType původně bylo nabídnout tvůrcům písma velkou kontrolu a přesnost nad tím, jak je písmo zobrazeno při různých velikostech [6].

Glyf je v tomto standardu definovaný konturami, které tvoří seznamy úseček a kvadratických Bézierových křivek. Tyto křivky jsou z matematického hlediska jednodušší a pro výpočty v algoritmech pro vykreslování textu vhodnější než kubické Bézierovy křivky, které jsou používané v jiných standardech. Toto zjednodušení ale znamená, že některé tvary potřebují pro popis kontury více bodů, což má za následek větší velikost souboru. TrueType může také obsahovat speciální tabulky zajišťující podporu některých typografických funkcionalit jako například *kerning*, což je optimalizace mezer mezi dvěma znaky pro lepší vizuální vzhled. Obsahuje na TrueType fonty mohou dokonce obsahovat i speciální bitmapové bloky, které nahrazují vektorový popis v případě potřeby vykreslení velmi malých znaků [6].

2.4.2 OpenType

Tento standard pro popis vektorových písem byl vyvinutý společnostmi Microsoft a AdobeSystems jako nástupce standardu TrueType a dá se označit za sjednocení dvou formátů fontů PostScript a TrueType. Základní výhody OpenType formátu jsou nezávislost na platformě, podpora mezinárodních znakových sad, zmenšení velikost souboru a lepší typografická podpora. Kontury mohou být tvořeny jak kvadratickými tak i kubickými Bézierovými křivkami.

Jednou z výhod v lepší typografické podpoře je možnost začlenit do jednoho fontu nejen obrovské množství znaků (OpenType podporuje Unicode stejně jako TrueType), ale dovoluje také zavést jejich různé varianty. Díky tomuto může písmo obsahovat alternativní sadu číslic nebo speciálních znaků. Další výhodou je podpora automatické záměny vzhledu některých znaků, aniž se tím naruší původní text. To se týče například *ligatur* neboli slitků, které dovolují spojit více znaků do jednoho glyfu. Tyto automatické náhrady se daleko více uplatní v jiných jazycích, jako je například arabština, kde má každý znak jiný tvar v závislosti na tom, stojí-li na začátku, uprostřed nebo na konci slova či samostatně. OpenType také podporuje *small caps* neboli kapitálky, což jsou písmena velké abecedy zobrazena velikostí malých písmen [3].

Díky tomuto formátu je možné vytvořit font, který obsahuje podporu pro mnoho jazyků a poskytuje velkou typografickou podporu. Tento standard díky plné podpoře Unicode umožňuje ukládat asi 65 000 znaků [3].

Kapitola 3

Křivky

V této kapitole jsou popsány základní koncepty geometrie křivek sloužící k pochopení principu metod rozebíraných v dalších kapitolách.

3.1 Parametrické křivky

Parametrický tvar vyjádření křivky je jedním z nejpoužívanější zápisů křivek v počítačové grafice. Výhodou tohoto vyjádření je závislost na jediném parametru.

Souřadnice na křivce můžeme zjistit volbou parametru t, který je obvykle volen v rozsahu $[0, 1]$. Racionální parametrickou křivku stupně n zapíšeme jako:

$$C(t) = \mathbf{t} \cdot \mathbf{C} \tag{3.1}$$

$$\mathbf{t} = \begin{bmatrix} 1 & t & \cdots & t^n \end{bmatrix} \quad \mathbf{C} = \begin{bmatrix} x_0 & y_0 & w_0 \\ \vdots & \vdots & \vdots \\ x_n & y_n & w_n \end{bmatrix}$$

Vektor $\mathbf{t}$ zde určuje monomiální bázi a matice $\mathbf{C}$ určuje její tvar.

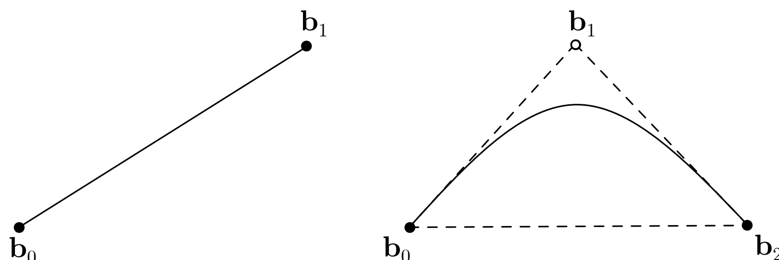
3.2 Bézierovy křivky

Bézierovy křivky jsou jedním ze zástupců parametrických aproximačních křivek. Glyfy jsou ve většině používaných formátů definovány svou konturou, která se skládá ze sekvence úseček a Bézierových křivek. Různé formáty ale využívají pro popis kontur glyfů jiné stupně těchto křivek.

Úsečka je ve své podstatě Bézierova křivka prvního stupně, kterou lze parametricky vyjádřit jako:

$$C(t) = (1-t)\,\mathbf{b}_0 + t\,\mathbf{b}_1 \tag{3.2}$$

Jedná se ve své podstatě o vážený průměr dvou bodů. Úsečka je množina bodů C pro $t \in \langle 0, 1 \rangle$ a lze ji vidět na obrázku 3.1 vlevo.

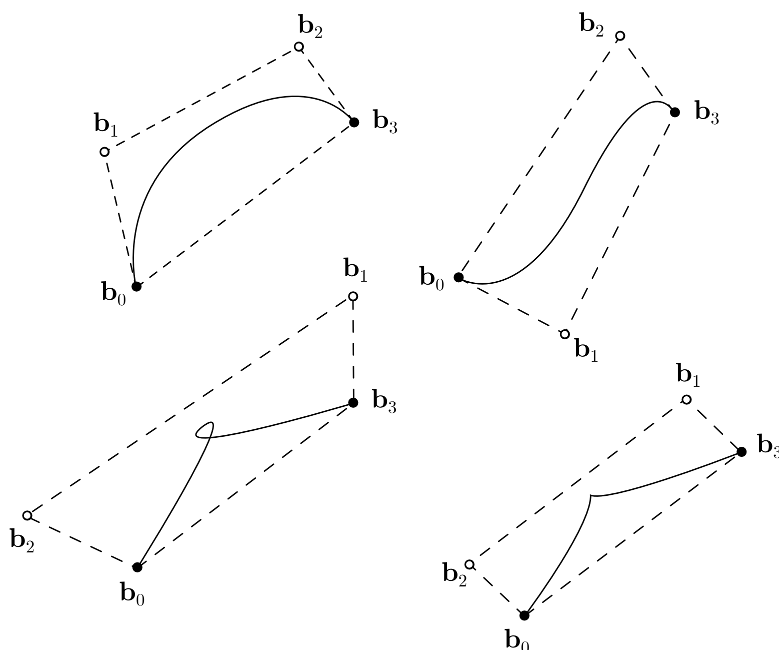


Obrázek 3.1: Bézierova křivka prvního a druhého stupně.

Bézierovu křivku stupně dva určují tři řídící body dohromady tvořící její řídící polygon. Dva koncové body určují její umístění a třetí prostřední bod určuje vychýlení dráhy spojující koncové body. Kvadratická Bézierova křivka s řídícími body $\mathbf{b}_0$, $\mathbf{b}_1$ a $\mathbf{b}_2$ je specifikována jako:

$$C(t) = (1-t)^2\,\mathbf{b}_0 + 2\,(1-t)\,t\,\mathbf{b}_1 + t^2\,\mathbf{b}_2 \tag{3.3}$$

Příklad Bézierovy křivky druhého stupně lze vidět na obrázku 3.1 vpravo.



Obrázek 3.2: Bézierovy křivky třetího stupně.

Kubickou Bézierovu křivku určují čtyři řídící body $\mathbf{b}_0$, $\mathbf{b}_1$, $\mathbf{b}_2$ a $\mathbf{b}_3$:

$$C(t) = (1-t)^3\,\mathbf{b}_0 + 3\,(1-t)^2\,t\,\mathbf{b}_1 + 3\,(1-t)\,t^2\,\mathbf{b}_2 + t^3\,\mathbf{b}_3 \tag{3.4}$$

Několik příkladů Bézierových křivek třetího stupně lze vidět na obrázku 3.2.

Obecně můžeme Bézierovu křivku definovat jako:

$$C(t) = \begin{bmatrix} B_0^n(t) & B_1^n(t) & \cdots & B_n^n(t) \end{bmatrix} \cdot \mathbf{B} \tag{3.5}$$

$$B_i^n(t) = \binom{n}{i} (1-t)^{n-i}\, t^i \qquad \mathbf{B} = \begin{bmatrix} \mathbf{b}_0 & \cdots & \mathbf{b}_n \end{bmatrix}^T$$

$B_i^n(t)$ je prvek Bernsteinovy báze a $\mathbf{B}$ je matice $3 \times n$ řídících bodů Bézierovy křivky.

Převod z Bernsteinovy do monomiální je invertibilní zobrazení $\mathbf{M}_i$ a lze jej vyjádřit součinem $\mathbf{M}_i\, \mathbf{B}$.

Matice změny báze pro kvadriky a kubiky:

$$\mathbf{M}_2 = \begin{bmatrix} 1 & 0 & 0 \\ -2 & 2 & 0 \\ 1 & -2 & 1 \end{bmatrix} \quad \mathbf{M}_3 = \begin{bmatrix} 1 & 0 & 0 & 0 \\ -3 & 3 & 0 & 0 \\ 3 & -6 & 3 & 0 \\ -1 & 3 & -3 & 1 \end{bmatrix} \tag{3.6}$$

Kapitola 4

Metody vykreslování textu

Od doby vzniku počítačů a potřeby vykreslování textu na obrazovku vzniklo mnoho metod, které využívají různé přístupy. Jejich vývoj popohání stále náročnější požadavky na rychlost vykreslování, vysokou kvalitu nebo také zobrazování textu ve 3D scénách pod různým úhlem kamery.

Neexistuje univerzální metoda, každá z nich má svoje výhody a nevýhody, které se musí zvážit podle potřeb aplikace, ve které mají být implementovány. Při výběru metody je důležité určit, jestli bude mít využití ve 2D nebo 3D aplikaci nebo zda text bude dynamický nebo statický. Důležitým faktorem je také trojice paměťová náročnost, přesnost vykreslení a rychlost.

Metody zobrazování se dají rozdělit do tří kategorií podle přístupu, dle kterého fungují [19]. Tyto kategorie jsou: metody využívající předrenderované glyfy, metody využívající vzdálenosti a geometricky založené metody.

4.1 Metody využívající předrenderované glyfy

Tyto metody v podstatě využívají předrenderované glyfy uložené v atlasu textur, které jsou ve 3D scéně dále zobrazeny na povrch objektů.

V roce 1997 Mark Kilgard představil jednoduché OpenGL API [8] pro mapování textu z textur na objekty ve 3D scéně. API má uložené několik rasterizovaných glyfů v jediné textuře a používá look-up table pro zjištění souřadnic jednotlivých obdélníků obsahujících glyfy. Tyto obdélníky jsou dále namapovány na požadovaný objekt ve scéně.

Tato technika je extrémně rychlá a jednoduchá na implementaci, ale trpí artefakty v podobě „pixelového“ vzhledu, jakmile velikost glyfů překročí rozlišení, ve kterém jsou uloženy v atlasu textur, i když je použita bilineární interpolace.

Pro zajištění vysoké kvality vykreslování musí být tyto glyfy uloženy ve vysokém rozlišení. Vysoké rozlišení atlasu textur může zabírat velké množství paměti, což může být problémové například u her.

Pro optimalizaci velikosti atlasu textur bylo vyvinuto mnoho tzv. *bin*

Obrázek 4.1: Glyfy uložené v atlasu textur vygenerované nástrojem Font Texture Generator [21].

packing algoritmů [7]. Tyto algoritmy rozmisťují jednotlivé glyfy do atlasu textur tak, aby se minimalizoval prázdný prostor mezi nimi a výsledná velikost atlasu se může výrazně snížit oproti naivnímu přístupu rozmístění glyfů.

Použití těchto metod může být efektivní pokud zaručíme, že jeden pixel odpovídá jednomu texelu (např. UI nebo HUD ve hrách). Pokud ale aplikujeme transformaci zvětšení nebo rotaci, tento předpoklad se poruší a lze zpozorovat artefakty.

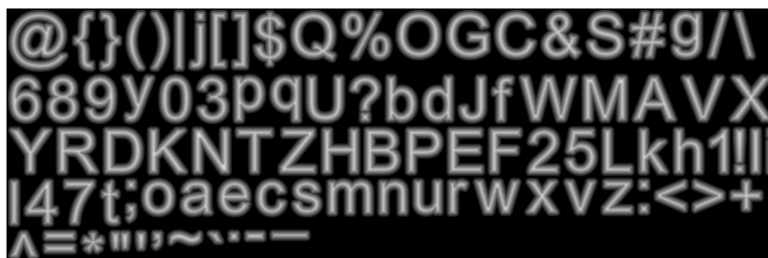
4.2 Metody využívající vzdálenosti

S použitím předchozích metod ve 3D interaktivních aplikacích (v počítačových hrách) může uživatel vidět text namapovaný na objekt s velkým zvětšením a proto je potřeba mít texturu uloženou ve vysoké kvalitě. Textury v menší kvalitě lze zvětšit procesem zvaným bilineární filtrování alfa kanálu. Pokud na výstup tohoto filtru aplikujeme *alpha testing*, můžeme dostat ostrý obraz bez rozmazání, který ovšem může obsahovat artefakty v podobě „zvlněných" hran, protože funkce pokrytí reprezentující alfa kanál není lineární.

Chris Green v roce 2007 představil metodu jednokanálového pole vzdáleností (*distance field*) [4]. Pole vzdáleností je vypočítáno z obrazu glyfu o vysokém rozlišení a poté uloženo do alfa kanálu textury menšího rozlišení. Každý alfa kanál udává nejbližší vzdálenost k pixelu obsahující glyf, kde hodnota 0 je největší negativní vzdálenost a hodnota 1 největší pozitivní vzdálenost. Hodnota 0.5 udává pozici kontury. Bilineárním filtrováním alfa kanálu a následným alfa testováním lze z pole vzdáleností o malém rozlišení vykreslit glyf v mnohokrát větším rozlišení. Atlas textur ukládající pole vzdáleností lze vidět na obrázku 4.2.

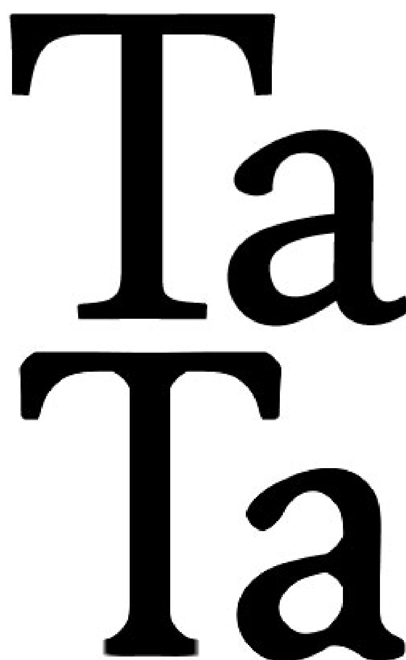
Jelikož je známá pozice kontury, tak není problém implementovat antialiasing mapováním určitého okolí kolem hodnoty 0.5 do intervalu $[0, 1]$ pro hodnotu alfa výsledného pixelu. Podobným způsobem lze implementovat mnoho různých efektů jako je zvětšení nebo zmenšení tloušťky glyfu, záři, stín nebo ohraničení glyfu.

Výstup této metody lze vidět na obrázku 4.3. Tato metoda má ovšem tendenci zaoblovat ostré rohy a proto nezachovává správný obrys kontury



Obrázek 4.2: Glyfy se vzdáleností v alfa kanálu uložené v atlasu textur vygenerované nástrojem Font Texture Generator [21].

glyfů, pokud se nezvětší rozlišení textury uchovávající pole vzdáleností.



Obrázek 4.3: Porovnání glyfu vykresleného z textury o vysokém rozlišení (nahoře) a glyfu vykresleného z textury uchovávající pole vzdáleností o menším rozlišení pomocí bilineárního filtrování (dole). Převzato z [18].

Problémy se zaoblením rohů vyřešil v roce 2015 Viktor Chlumský ve své diplomové práci [5]. Vhodnou dekompozicí glyfu vytvoří vícekanálové pole vzdáleností a pomocí operací AND a OR mezi nimi odstraní toto zaoblení. Algoritmus ovšem vyžaduje komplikovanější předzpracování atlasu textur.

4.3 Geometricky založené metody

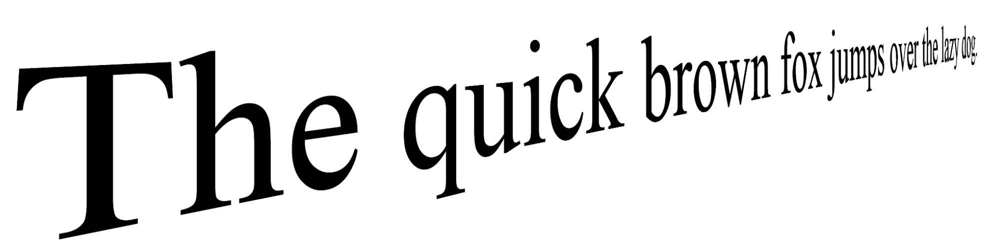
Předchozí metody ukládají data do atlasu textur a pracují tedy s diskrétní reprezentací uložených glyfů. Tento přístup produkuje artefakty, které lze vyřešit zvětšením rozlišení textury, což ovšem nemusí být vždy vhodným

řešením, pokud je aplikace paměťově omezená. Těmto limitacím se lze vyhnout vykreslováním glyfů přímo z matematicky popsaných křivek, které definují jejich tvar.

Charles Loop a Jim Blinn představili v roce 2005 nový přístup vykreslování kvadratických a kubických křivek nezávisle na rozlišení na GPU [14]. Jejich metoda vhodným způsobem provádí triangulaci glyfu a krátký výpočet ve fragment shaderu rozhoduje zda daný pixel leží uvnitř nebo vně dané komponenty. Využili tak nových možností programovatelných částí grafického zobrazovacího řetězce. Tento přístup nabízí vysokou kvalitu renderování a vykresluje glyfy libovolné velikosti ve 3D scéně bez artefaktů při záběru perspektivní kamerou.

Tato metoda vytvořila zcela nový pohled na problém vykreslování textu a v budoucích letech po jejím představení byla mnohokrát vylepšena a rozšířena jako například v [10]. U této metody není potřeba složitý proces triangulace jako v [14] a využitím stencil bufferu pro výpočet vnitřní části glyfu je dosaženo až 10x většího výkonu. Tato metoda je také vhodná pro vektorové objekty, které se dynamicky mění, ale bohužel plně nepodporuje vykreslování kubických Bézierových křivek. Na základě ní je postavena metoda [9], která pomocí OpenGL extenze `NV_path_rendering` umožňuje vykreslovat křivky a křivkami ohraničené oblasti. Tato metoda rozšířila předchozí metodu o možnost vykreslování všech segmentů ohraničených kubickými Bézierovými křivkami, díky vhodnému rozdělení těchto křivek a také zlepšila antialiasing.

V roce 2017 představil Eric Lengyel metodu [11], která taktéž vykresluje text nezávisle na rozlišení, bez aliasu, přímo z dat popisující jednotlivé kontury glyfu. Tyto data jsou po minimálním preprocessingu poslána na GPU, kde probíhá celý výpočet bez použití předpočítaných textur nebo polí vzdáleností. Metoda je přímo určená pro využití ve 3D scénách s velmi vysokým a efektivním využitím GPU. Tato metoda jako jediná existující vykresluje text bez aliasu a artefaktů na GPU. Text vykreslený touto metodou lze vidět na obrázku 4.4



Obrázek 4.4: Text vykreslený metodou Lengyel [11].

4.4 Knihovny pro vykreslování textu

Součástí této práce je kompletace vybraných metod do knihovny, která bude součástí PGR-frameworku. Před návrhem této knihovny je proto důležité

prostudovat již existující knihovny, aby mohla být vhodně navržena její struktura.

4.4.1 FreeType

FreeType je zdarma dostupná knihovna pro vykreslování textu implementovaná v programovacím jazyku C. Tvůrci této knihovny se soustředili na to, aby byla knihovna relativně malá, výkonná, lehce přizpůsobitelná aplikaci, ve které je použita a byla multiplatformní [16].

Tato knihovna umožňuje načítat fonty z několika vektorových i bitmapových formátů včetně formátů TrueType a OpenType s plnou podporou Unicode. V této knihovně je implementovaný resterizovací engine, který je schopný vyprodukovat kvalitní bitmapu z glyfu, kterou lze dále využít k vykreslení na obrazovku. Knihovna také poskytuje přístup k informacím o metrikách, kerningu a dovoluje také přístup k datům až na úroveň jednotlivých kontur glyfu. Knihovna FreeType neposkytuje API pro zajištění pokročilého rozvržení textu, vykreslování barevného textu nebo speciálních efektů (stín, záře).

Jakmile chceme tuto knihovnu použít v aplikaci společně s OpenGL, musíme vytvořit obdélníky z trojúhelníků pro jednotlivé znaky, vygenerovat bitmapu požadovaných znaků o dané velikosti a trojúhelníkům přiřadit texturovací souřadnice určitého znaku.

Tato knihovna tedy využívá předrenderované glyfy a je ji vhodné použít, pokud je předem známá velikost textu, který chceme vykreslit a během vykreslování se dále nemění. Tuto knihovnu také není vhodné použít ve 3D scéně s perspektivní projekci, pokud text není vždy otočený směrem ke kameře.

4.4.2 Slug

Slug je knihovna napsána v jazyce C++, která poskytuje vysokou kvalitu vykreslování textu a vektorové grafiky nezávisle na rozlišení s vysokým využitím GPU [13].

Tato knihovna je postavena na metodě [11] a proto je vysoce efektivní a využitelná pro vykreslování ve 3D scéně i při záběru perspektivní kamerou.

Dovoluje načítat fonty TrueType a OpenType, která dále převádí na vlastní formát, ze kterého lze font načíst. Poskytuje také komplexní rozvržení textu pro řetězec znaků, kerning, ligatury, kombinování diakritických znamének, vykreslování barevného textu (emoji), alternativní substituci jednotlivých znaků a mnoho dalších funkcionalit. Je zde plná podpora Unicode a dovoluje také například správné skládání arabských znaků, které používá velmi neobvyklý a složitý systém rozvržení.

Při použití knihovny v aplikaci se nejprve načtou data z určeného fontu, vygeneruje se vertex buffer uchovávající data o vrcholech trojúhelníků, na které jsou glyfy vykresleny a vytvoří se dvě textury potřebné pro výpočet. První textura uchovává data o jednotlivých křivkách a druhá je potřebná k efektivnímu

vykreslování (k optimalizaci). Při vykreslení textu knihovna pošle data z vertex bufferu na GPU společně s texturami pro daný glyf. Ty jsou následně vykresleny vhodným vertex a fragment shaderem.

Knihovna Slug může být využita v široké škále aplikací od vykreslování GUI až po umístění textu do 3D scény.

Kapitola 5

Analýza vybraných metod

Metody popsané v této kapitole jsou zástupci geometricky založených metod. Obě metody umožňují zobrazit text bez aliasu, nezávisle na rozlišení a také v perspektivě s vysokým využitím GPU bez použití předpočítaných textur nebo polí vzdáleností. Díky těmto vlastnostem jsou vhodné pro vykreslování textu ve 3D scéně, kde dochází k neustálé změně pozice kamery s perspektivní projekcí a velikost glyfů se tedy téměř vždy liší snímek po snímku.

Tyto metody byly k analýze a následné implementaci do výsledné knihovny vybrány kvůli zmíněným vlastnostem, efektivitě, vhodnosti implementace pomocí grafické knihovny OpenGL a přiměřené složitosti jejich principu. Obě metody umožňují splnit požadavky na vlastnosti implementovaných metod specifikované v kapitole 1.

5.1 Metoda vykreslování kvadratických a kubických křivek nezávisle na rozlišení (Loop, Blinn)

Tato metoda je jedním z prvních zástupců geometricky založených metod a je popsána v článku [2] a částečně také v knize [15]. Poskytuje zcela nový pohled na problematiku vykreslování textu vůči předešlým metodám. Dovoluje nezávisle na rozlišení vykreslovat křivky a křivkami ohraničené útvary, které jsou definované pomocí úseček a kvadratických nebo kubických Bézierových křivek. Moderní GPU dovolují částečné přeprogramování grafického zobrazovacího řetězce, umožňují rychlé vykreslování trojúhelníků a interpolaci hodnot mezi jejich jednotlivými vrcholy, čehož tento přístup využívá. Tato metoda dovoluje také křivky zobrazit ve 3D scéně v záběru perspektivní kamerou.

Hlavní myšlenkou této metody je vytvoření implicitní funkce, kdy transformací parametrické funkce $C(t) = \begin{bmatrix} x(t) & y(t) \end{bmatrix}$ získáme její implicitní zápis $f(x, y) = 0$. Algoritmus poté vyrendruje řídící polygon Bézierových křivek jako trojúhelníky a krátký výpočet ve fragment shaderu podle implicitního zápisu vyhodnotí, zda se daný pixel nachází uvnitř křivky nebo mimo ni. Fragment shader lze také doplnit o výpočty umožňující antialiasing.

Díky tomu, že je algoritmus popsaný pro kvadratické i kubické Bézierovy křivky, tak je možné načítat fonty z formátu TrueType i OpenType. Algoritmy pro oba stupně jsou dále popsány pro obecné Bézierovy křivky (ne pro racionální), jelikož racionální křivky se pro popis fontů nepoužívají. Verze pro racionální křivky je popsána v článku [2] a může mít využití ve vykreslování jiné vektorové grafiky.

5.1.1 Vykreslování kvadratických křivek

Jak již bylo zmíněno, je potřeba vytvořit implicitní zápis křivky z parametrického zápisu. Výhodou implicitního zápisu je, že lze jednoduše rozhodovat, zdali se bod nachází uvnitř útvaru, který křivka definuje.

Parametrickou křivku lze výpočtem implicitovat, ale projekce do *screen space* se může s každým snímkem lišit a tím se liší i implicitní zápis křivky a může být výpočetně náročné tento zápis při každém novém snímku hledat.

Uvažujme racionální kvadratickou parametrickou křivku 3.1. Dále uvažujme křivku:

$$F(t) = \mathbf{t} \cdot \mathbf{F} = \begin{bmatrix} 1 & t & t^2 \end{bmatrix} \begin{bmatrix} 0 & 0 & 1 \\ 1 & 0 & 0 \\ 0 & 1 & 0 \end{bmatrix} = \begin{bmatrix} t & t^2 & 1 \end{bmatrix} \tag{5.1}$$

Pro křivku F označme $u(t) = t$ a $v(t) = t^2$. Můžeme ji tedy zapsat implicitně jako:

$$f(u, v) = u^2 - v \tag{5.2}$$

Dále hledáme zobrazení Φ takové, že $\mathbf{C} = \mathbf{F} \cdot \mathbf{\Phi}^{-1}$. Snadno vidíme, že $\mathbf{\Phi}^{-1} = \mathbf{F}^{-1} \cdot \mathbf{C}$, což znamená, že křivka $C(t)$ je transformací křivky $f(u, v)$.

Převedením $\mathbf{F}$ do Bernesteinovy báze 3.6 získáme souřadnice řídících bodů Bézierovy křivky odpovídající křivce $\mathbf{F}$.

$$\mathbf{M}_2^{-1} \cdot \mathbf{F} = \begin{bmatrix} 0 & 0 & 1 \\ \frac{1}{2} & 0 & 1 \\ 1 & 1 & 1 \end{bmatrix} \tag{5.3}$$

Můžeme tedy přiřadit souřadnice řídících bodů $[0, 0]$, $[\frac{1}{2}, 0]$ a $[1, 1]$ jako texturovací souřadnice $[u, v]$ řídícím bodům kvadratické Bézierovy křivky.

Texturovací souřadnice se z vertex shaderu do fragment shaderu interpolují, takže dostaneme korektní hodnotu texturovacích souřadnic pro každý pixel. Výpočet ve fragment shaderu tedy bude vyhodnocovat 5.2 pro texturovací souřadnice u a v.

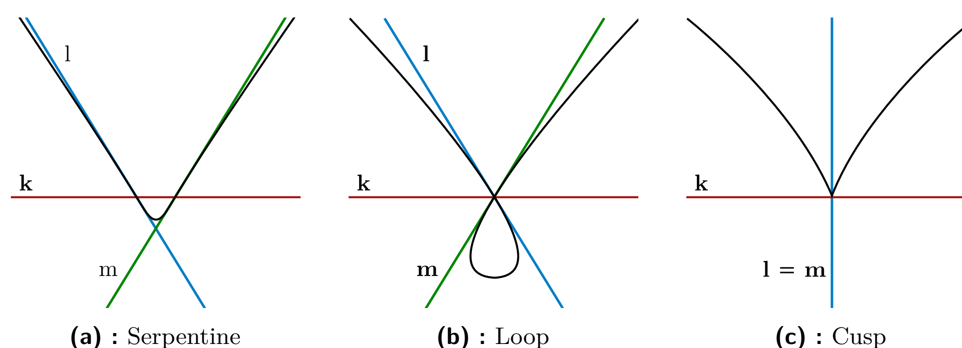
5.2 odpovídá konvexní křivce. Pro naše potřeby je nutné vyhodnocovat také konkávní křivky. Stačí pouze 5.2 změnit na $f(u, v) = v - u^2$.

5.1.2 Vykreslování kubických křivek

Přístup pro vykreslování kubických Bézierových křivek je velmi podobný kvadratickým, co se týče vyhodnocování ve fragment shaderu. Preprocessing dat pro získání texturovacích souřadnic a dekompozice glyfu je ovšem složitější.

Pro kvadratické Bézierovy křivky platilo, že každá racionální parametrická křivka je transformací křivky implicitního zápisu 5.2. Pro kubické Bézierovy křivky platí, že každá racionální parametrická křivka C je transformací jedné ze tří typů křivek. K těmto křivkám lze ještě zahrnout speciální případy, pro které je kubická Bézierova křivka křivkou kvadratickou, úsečkou nebo bodem. Tyto případy by se ovšem ve standardech pro popis fontů neměly vyskytovat, jelikož je lze vyjádřit jednodušším způsobem.

Každou kubickou Bézierovu křivku můžeme tedy klasifikovat jako *serpentine*, *cusp* nebo *loop* 5.1.



(a) : Serpentine **(b)** : Loop **(c)** : Cusp

Obrázek 5.1: Tři typy klasifikace kubické Bézierovy křivky. Překresleno z [15]

Všechny tři typy těchto křivek lze implicitně zapsat jako:

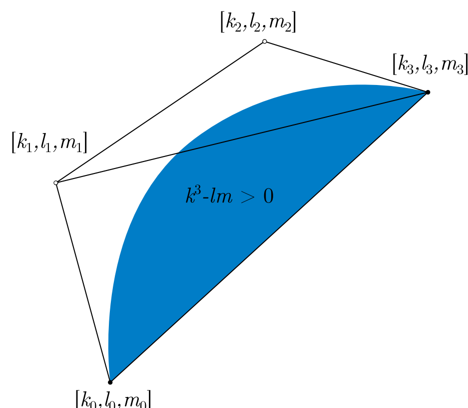
$$c(x, y) = k^3 - lm \tag{5.4}$$

kde k, l a m jsou homogenní rovnice přímek $\mathbf{k}$, $\mathbf{l}$ a $\mathbf{m}$. Například pro k je rovnice $k = au + bv + cw$ kde $\begin{bmatrix} u & v & w \end{bmatrix}$ jsou homogenní souřadnice bodu v rovině a $\mathbf{k} = \begin{bmatrix} a & b & c \end{bmatrix}^T$

Cílem tohoto algoritmu je vypočítat hodnoty k, l a m po každý řídící bod kubické Bézierovy křivky a přiřadit je jako texturovací souřadnice, které dále bude fragment shader vyhodnocovat podle 5.4. Vizualizace této myšlenky je pro jasnější představu na obrázku 5.2.

Klasifikace

Pro zjištění hodnot texturovacích souřadnic musíme křivku nejprve klasifikovat jedním z typů 5.1. To lze zjistit počtem inflexních bodů dané křivky, což jsou body, ve kterých druhá derivace C'' mění znaménko.



Obrázek 5.2: Řídící polygon kubické Bézierovy křivky s texturovacími souřadnicemi.

Vztah mezi přímkami **k**, **l**, **m** a křivkou C je následující. Křivka C typu serpentine má inflexní body v $\mathbf{k} \cap \mathbf{l}$ a $\mathbf{k} \cap \mathbf{m}$. Křivka C typu loop má dvojitý inflexní bod v $\mathbf{k} \cap \mathbf{l}$ a $\mathbf{k} \cap \mathbf{m}$. Křivka C typu cusp má hrot v bodě, kde **k** má průnik se shodnými přímkami **l** a **m**.

Kubickou Bézierovu křivku s řídícími body $\mathbf{b}_0$,...,$\mathbf{b}_3$ můžeme parametricky zapsat jako:

$$C(s,t) = \begin{bmatrix}(s-t)^3 & 3(s-t)^2 s & 3(s-t)s^2 & s^3\end{bmatrix} \begin{bmatrix}\mathbf{b}_0 \\ \mathbf{b}_1 \\ \mathbf{b}_2 \\ \mathbf{b}_3\end{bmatrix} \tag{5.5}$$

Prvním krokem je převedení křivky z Bernsteinovy do monomiální báze 3.6 $\mathbf{C} = \mathbf{M}_3 \cdot \mathbf{B}$ a dále výpočet koeficientů funkce $I(s,t)$, jejíž kořeny jsou inflexní body křivky $C(s,t)$ jak je popsáno v [2].

$$I(s,t) = s(-3\,d_1 t^2 + 3\,d_2 ts - d_3 s^2) \tag{5.6}$$

Funkce 5.7 je kubická se třemi kořeny - nemusí být všechny reálné. Počet reálných kořenů této funkce určuje typ kubické křivky. Pro obecné kubické Bézierovy křivky (ne racionální, tj. $w = 1$) je $\begin{bmatrix}s & t\end{bmatrix} = \begin{bmatrix}1 & 0\end{bmatrix}$ vždy kořenem I. Klasifikace se tedy omezuje na znaménko diskriminantu I:

$$discr(I) = d_1^2(3\,d_2^2 - 4\,d_3 d_1) \tag{5.7}$$

$$d_1 = -\det \begin{bmatrix} x_3 & y_3 \\ x_2 & y_2 \end{bmatrix}$$
$$d_2 = \det \begin{bmatrix} x_3 & y_3 \\ x_1 & y_1 \end{bmatrix}$$
$$d_3 = -\det \begin{bmatrix} x_2 & y_2 \\ x_1 & y_1 \end{bmatrix}$$

Pro klasifikaci podle znaménka diskriminatu platí:

Serpentine	$discr(I) > 0$
Cusp	$discr(I) = 0$
Loop	$discr(I) < 0$

Křivka typu serpentine má podle znaménka diskriminantu tři reálné kořeny. Křivka typu cusp má jeden reálný kořen a jeden dvojitý reálný kořen. Křivka typu loop má jeden reálný kořen a dva komplexní kořeny.

Po klasifikaci již zbývá pouze najít texturovací souřadnice $\begin{bmatrix} k & l & m \end{bmatrix}$ pro každý řídící bod. Tyto souřadnice jsou dány funkcemi:

$$\begin{aligned} k(s,t) &= \mathbf{k} \cdot \mathbf{C}(s,t) \\ l(s,t) &= \mathbf{l} \cdot \mathbf{C}(s,t) \\ m(s,t) &= \mathbf{m} \cdot \mathbf{C}(s,t) \end{aligned} \tag{5.8}$$

Tyto funkce lze zkonstruovat součinem jejich lineárních faktorů, které lze zjistit nalezením kořenů $I(s,t)$ a hessiánu $I(s,t)$. Jejich převedením do Bézierova tvaru dostaneme matici ve tvaru:

$$\mathbf{M} = \begin{bmatrix} k_0 & l_0 & m_0 \\ k_1 & l_1 & m_1 \\ k_2 & l_2 & m_2 \\ k_3 & l_3 & m_3 \end{bmatrix} \tag{5.9}$$

jejíž řádky odpovídají texturovacím souřadnicím pro jednotlivé řídící body.

V některých případech může mít křivka špatnou orientaci, tu lze obrátit změnou znaménka implicitní funkce 5.4. To lze provést změnou znaménka texturovacích souřadnic k a l.

Serpentine

Pro křivku typu serpentine bude matice $\mathbf{M}$ vypadat následovně:

$$\mathbf{M} = \begin{bmatrix} l_s m_s & l_s^3 & m_s^3 \\ \frac{1}{3}(3l_s m_s - l_s m t - l_t m_s) & l_s^2(l_s - l_t) & m_s^2(m_s - m_t) \\ \frac{1}{3}(l_t(m_t - 2m_s) + l_s(3m_s - 2m_t)) & (l_t - l_s)^2 l_s & (m_t - m_s)^2 m_s \\ (l_t - l_s)(m_t - m_s) & -(l_t - l_s)^3 & -(m_t - m_s)^3 \end{bmatrix} \tag{5.10}$$

$$\begin{bmatrix} l_s & l_t \end{bmatrix} = \begin{bmatrix} 3d_2 + \sqrt{9d_2^2 - 12d_1d_3} & 6d_1 \end{bmatrix}$$
$$\begin{bmatrix} m_s & m_t \end{bmatrix} = \begin{bmatrix} 3d_2 - \sqrt{9d_2^2 - 12d_1d_3} & 6d_1 \end{bmatrix}$$

Pokud $d_1 < 0$ musí se obrátit orientace 5.4.

Cusp

Křivka typu cusp je hranicí mezi typy serpentine a loop. Pro křivku typu cusp bude matice **M** vypadat následovně:

$$\mathbf{M} = \begin{bmatrix} l_s & l_s^3 & 1 \\ l_s - \frac{1}{3}l_t & l_s^2(l_s - l_t) & 1 \\ l_s - \frac{2}{3}l_t & l_s(l_s - l_t)^2 & 1 \\ l_s - l_t & (l_s - l_t)^3 & 1 \end{bmatrix} \tag{5.11}$$

$$\begin{bmatrix} l_s & l_t \end{bmatrix} = \begin{bmatrix} d_3 & 3d_2 \end{bmatrix}$$

Je zde ovšem výjimka. pokud $d1 \neq 0$ a $3d_2^2 - 4d_1d_3 = 0$, pak sloučíme tento případ s křivkou typu serpentine. V případě křivky typu cusp se nemusí měnit orientace 5.4.

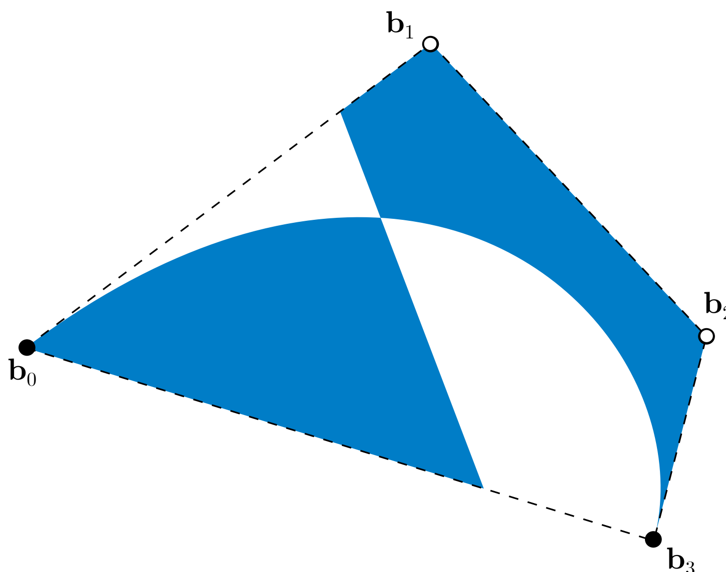
Loop

Pro křivku typu loop budou jednotlivé sloupce matice **M** vypadat následovně:

$$\begin{aligned} \mathbf{M}_{0,} &= \begin{bmatrix} l_s m_s \\ \frac{1}{3}(3l_s m_s - l_s mt - l_t m_s) \\ \frac{1}{3}(l_t(m_t - 2m_s) + ls(3m_s - 2m_t)) \\ (l_t - l_s)(m_t - ms) \end{bmatrix} \\ \mathbf{M}_{1,} &= \begin{bmatrix} l_s^2 m_s \\ -\frac{1}{3}l_s(ls_s(m_t - 3m_s) + 2l_t m_s) \\ \frac{1}{3}(l_t - l_s)(l_s(2m_t - 3m_s) + l_t m_s) \\ -(l_t - l_s)^2(m_t - ms) \end{bmatrix} \\ \mathbf{M}_{2,} &= \begin{bmatrix} l_s m_s^2 \\ -\frac{1}{3}m_s(ls_s(2m_t - 3m_s) + l_t m_s) \\ \frac{1}{3}(m_t - m_s)(l_s(m_t - 3m_s) + 2l_t m_s) \\ -(l_t - l_s)(m_t - ms)^2 \end{bmatrix} \end{aligned} \tag{5.12}$$

$$\begin{bmatrix} l_s & l_t \end{bmatrix} = \begin{bmatrix} d_2 - \sqrt{4d_1d_3 - 3d_2^2} & 2d_1 \end{bmatrix}$$
$$\begin{bmatrix} m_s & m_t \end{bmatrix} = \begin{bmatrix} d_2 + \sqrt{4d_1d_3 - 3d_2^2} & 2d_1 \end{bmatrix}$$

V tomto případě se může objevit artefakt, který lze vidět na obrázku 5.3, pokud $\frac{l_s}{l_t} \in [0,1]$ nebo $\frac{m_s}{m_t} \in [0,1]$. Toto zde vyřešit rozdělením křivky algoritmem de Casteljau v hodnotě $\frac{l_s}{l_t}$ nebo $\frac{m_s}{m_t}$ na dvě samostatné křivky a změnou orientace jedné z křivek.



Obrázek 5.3: Možný artefakt u křivky typu loop.

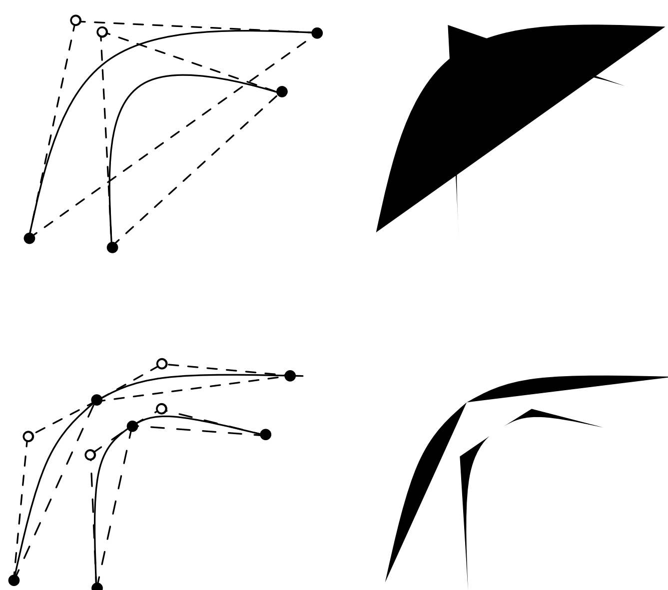
Změna orientace 5.4 je nutná pokud $d_1 > 0$ a zároveň $k_1 < 0$ nebo $d_1 < 0$ a zároveň $k_1 > 0$.

5.1.3 Triangulace

V této chvíli je možné korektně vykreslit kvadratickou i kubickou Bézierovu křivku pomocí jednoduchého výpočtu ve fragment shaderu. Zbývá tedy provést vhodnou dekompozici kontur TrueType a OpenType fontu.

Před dekompozicí je ovšem nutné zbavit se překrývajících trojúhelníků, které se mohou objevit u některých glyfů. Pokud je překrytí detekováno, tak je potřeba rozdělit trojúhelník s větším obsahem na dva menší trojúhelníky 5.4. Toto rozdělení se provede algoritmem de Casteljau.

Dekompozicí TrueType fontu získáme tři druhy trojúhelníku - konvexní, konkávní a vnitřní. Konvexní trojúhelníky odpovídají konvexním Bézierovým křivkám a konkávní trojúhelníky konkávním Bézierovým křivkám. Vnitřní trojúhelníky odpovídají vnitřní části glyfu a jsou vyplněné v celém svém



Obrázek 5.4: Odstranění překrývajících se trojúhelníků.

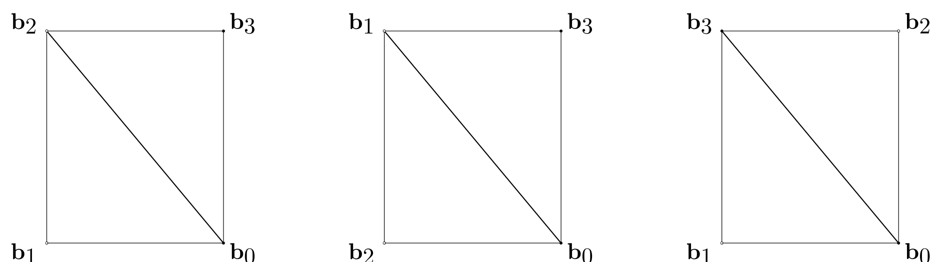
obsahu.

Iterací přes body kontury postupně vytváříme konvexní a konkávní trojúhelníky pro každé tři řídící body $\mathbf{b}_0$, $\mathbf{b}_1$, $\mathbf{b}_2$. Ty lze rozlišit podle orientace vektorů $\mathbf{u} = \mathbf{b}_1 - \mathbf{b}_0$ a $\mathbf{v} = \mathbf{b}_2 - \mathbf{b}_0$. Determinant $\det(\mathbf{u}, \mathbf{v})$ bude záporný pro konvexní křivky a kladný pro konkávní křivky v orientaci TrueType.

Při iteraci je také potřeba vytvářet seznamy bodů, které ohraničují vnitřek glyfu a to pro každou konturu zvlášť. Z tohoto seznamu budou dále vytvořeny vnitřní trojúhelníky glyfu. Pro konvexní trojúhelníky tento seznam musí obsahovat pouze krajní řídící body $\mathbf{b}_0$ a $\mathbf{b}_2$. Pro konkávní trojúhelníky tento seznam musí obsahovat všechny řídící body.

Dekompozicí OpenType fontu získáváme dva druhy trojúhelníků a to trojúhelníky odpovídající části řídícího polygonu kubické Bézierovy křivky a trojúhelníky vnitřní.

Řídící polygon kubické Bézierovy křivky se skládá ze čtyř bodu, takže musí být provedena jeho triangulace. Existuje 24 permutací řídících bodů a pokud ignorujeme rotace a reflexe, které jsou irelevantní, zůstanou nám pouze 3 možné uspořádání řídícího polygonu 5.5. Pro ně ovšem můžeme zvolit, kterou diagonálou tyto čtyřúhelníky rozdělíme.



Obrázek 5.5: Tři možné triangulace řídícího polygonu kubické Bézierovy křivky.

Z této triangulace dále určujeme, které body budou ohraničovat vnitřní část glyfu podobně jako u kvadratických Bézierových křivek.

Vnitřní trojúhelníky glyfů lze vytvořit vázanou Delaunyho triangulací, aby se zachovaly úsečky spojující jednotlivé body v těchto seznamech. Tyto trojúhelníky lze vykreslit buď samostatným fragment shaderem nebo jim přiřadit texturovací souřadnice, které odpovídají vnitřku křivky a využít stávající shader, čímž zrychlíme celý proces vykreslování, jelikož přepínání mezi shadery může snížit výkon.

5.1.4 Antialiasing

V tuto chvíli je možné glyf korektně vykreslit, ale vyskytují se u něj artefakty v podobě aliasu. Tento alias je možné odstranit uvnitř řídícího polygonu Bézierových křivek pomocí výpočtu vzdálenosti od křivky ve *screen-space*. Výpočet přesné vzdálenosti křivky od pixelu by byl výpočetně neefektivní, ale dobrou aproximací je použití výpočtu vzdálenosti pomocí gradientu funkce $f(x, y) = 0$. Gradient je kolmý ke křivce pro body $[x, y]$ ležící na křivce a téměř kolmý pro body $[x, y]$, které jsou blízko křivce a lze jej vypočítat následovně:

$$\nabla f(x, y) = \begin{bmatrix} \frac{\partial f}{\partial x}(x, y) & \frac{\partial f}{\partial y}(x, y) \end{bmatrix} \tag{5.13}$$

Aproximace vzdálenosti ke křivce je tedy:

$$sd(x, y) = \frac{f(x, y)}{||\nabla f(x, y)||} \tag{5.14}$$

Pro výpočet gradientu je výhodné využít hardwarový výpočet parciálních derivací, které jsou implementovány na nových GPU. Při rasterizaci trojúhelníků GPU spouští mnoho instancí fragment shaderu paralelně, které jsou organizovány do bloků velikosti 2×2 pixely. Parciální derivace jsou vypočítány jako rozdíl hodnot mezi těmito pixely v daném směru. Díky tomu, že počítají pouze rozdíl hodnot, je lze využít pouze k aproximaci derivací funkce vyšších řádů, což může vést k artefaktům. Přesnějšího výpočtu dosáhneme použitím řetízkového pravidla [15].

Pro shader kvadratických křivek vyhodnocující implicitní funkci $f(x, y) = u^2 - v = 0$ použitím řetízkového pravidla dostaneme:

$$\begin{bmatrix} \frac{\partial q}{\partial x} \\ \frac{\partial q}{\partial y} \end{bmatrix} = \begin{bmatrix} \frac{\partial u}{\partial x} & \frac{\partial v}{\partial x} \\ \frac{\partial u}{\partial y} & \frac{\partial v}{\partial y} \end{bmatrix} \cdot \begin{bmatrix} 2u \\ -1 \end{bmatrix} \tag{5.15}$$

Pro shader kubických křivek vyhodnocující implicitní funkci $f(x, y) = k^3 - lm = 0$ použitím řetízkového pravidla dostaneme:

$$\begin{bmatrix} \frac{\partial c}{\partial x} \\ \frac{\partial c}{\partial y} \end{bmatrix} = \begin{bmatrix} \frac{\partial k}{\partial x} & \frac{\partial l}{\partial x} & \frac{\partial m}{\partial x} \\ \frac{\partial k}{\partial y} & \frac{\partial l}{\partial y} & \frac{\partial m}{\partial y} \end{bmatrix} \cdot \begin{bmatrix} 3k^2 \\ -m \\ -l \end{bmatrix} \tag{5.16}$$

Finální podoba funkce určující vzdálenost ke křivce je:

$$sd(x, y) = \frac{f(x, y)}{\sqrt{(\frac{\partial f}{\partial x})^2 + (\frac{\partial f}{\partial y})^2}} \tag{5.17}$$

Tuto vzdálenost použijeme pro alfa kanál výsledného pixelu jako $alpha = \frac{1}{2} - sd(x, y)$, který zprůhlední pixely podle jejich vzdálenosti od křivky.

5.1.5 Paměťová složitost a rychlost

Tato metoda vyžaduje složitější předzpracování na CPU pro přípravu dat pro GPU. Tento proces je ovšem potřeba pouze při spuštění aplikace, kdy se musí provést potřebná triangulace glyfu a klasifikace jednotlivých křivek.

Data potřebná pro GPU se skládají pouze z vrcholů jednotlivých trojúhelníků, které obsahují prostorové souřadnice x, y a texturovací souřadnice u, v nebo k, l, m. Pro vnitřní trojúhelníky glyfu je potřeba ještě uložit indexy jednotlivých vrcholů pro konstrukci trojúhelníků.

Tyto data však mohou být jednou předzpracovány, uloženy disk a přímo poslány na GPU bez potřeby znovu provádět předzpracování.

Počet trojúhelníků výsledného glyfu je úměrný jeho složitosti. Rychlost vykreslování závisí tedy na složitosti daného fontu a u složitějších fontů, které potřebují více trojúhelníků je pomalejší. Evaluace ve fragment shaderu ovšem provádí u kvadratických i kubických křivek jednoduchou operaci a je tedy poměrně rychlá.

V porovnání s ostatními metodami využívající předrenderované glyfy nebo pole vzdáleností, které pouze samplují z textury mapované na obdélník, je zřejmé, že tato metoda bude pomalejší.

5.2 Metoda přímého vykreslování textu (Lengyel)

Tato metoda umožňuje zobrazovat text přímo z dat popisující jednotlivé glyfy a je popsána v článku [11]. Na rozdíl od předešlé metody vyžaduje minimální předzpracování dat pro GPU. Metoda je navržena tak, aby podporovala implementaci s vysokým využitím GPU a při výpočtech ve fragment shaderu se vyhýbala dynamickému programovému větvení vedoucímu k problému známém jako *thread divergence*. Svým přístupem předchází problémům s numerickou přesností, které měly za následek vytváření artefaktů a objevují se u dříve publikovaných technik vykreslování textu. Metoda je omezena pouze na kvadratické Bézierovy křivky používané standardem TrueType, aby byly výpočty ve fragment shaderu co nejjednodušší a tím bylo dosaženo vysokého výkonu při vykreslování. Základem této metody je výpočet navíjecího čísla, dle kterého lze rozhodnout zda daný pixel leží uvnitř glyfu nebo mimo něj.

5.2.1 Výpočet navíjecího čísla

Navíjecí číslo neboli *winding number* uzavřené křivky v rovině kolem bodu je celočíselná hodnota odpovídající počtu smyček, které křivka opíše okolo bodu.

Pozitivní navíjecí číslo je přiřazováno směru obtočení po směru hodinových ručiček (CW) nebo proti směru hodinových ručiček (CCW) a negativní opačnému směru oproti přiřazenému pozitivnímu směru. Pro bod ležící uvnitř glyfu tedy platí, že součet jeho navíjecích čísel je nenulový. Pokud je součet navíjecích čísel nula, pak bod leží v prázdném prostoru mimo glyf.

Navíjecí číslo se vypočítá z průsečíku polopřímky, která má počátek ve vykreslovaném bodě s jednotlivými konturami glyfu. Na začátku je navíjecí číslo inicializované na hodnotu nula. Pozitivní nebo negativní přírůstek k navíjecímu číslu se poté určuje podle směru, jakým kontura protíná polopřímku. Tato skutečnost dovoluje určení směru otáčení kontur.

Protože při výpočtu navíjecího čísla na směru polopřímky nezáleží, jsou pro jednoduchost výpočtů vybrány směry rovnoběžné se souřadnicovými osami.

Část kontury je definovaná kvadratickou Bézierovou křivkou 3.3. Tato funkce reprezentuje Bézierovu křivku se třemi 2D řídícími body $\mathbf{b}_0$, $\mathbf{b}_1$ a $\mathbf{b}_2$ s parametrem $t \in [0, 1]$. Pro řídící body platí $\mathbf{b}_i = (x_i, y_i)$. Pro polopřímku v pozitivním směru osy x, pro kterou je vykreslovaný bod posunut do počátku soustavy souřadnic tak, aby počátek polopřímky měl souřadnice $(0, 0)$, lze vypočítat hodnoty parametru t, ve kterých $C_y(t) = 0$ nalezením kořenů polynomu:

$$C_y(t) = at^2 - 2bt + c \tag{5.18}$$

$$a = y_0 - 2y_1 + y_2$$
$$b = y_0 - y_1$$
$$c = y_0$$

Kořeny polynomu jsou:

$$t_1 = \frac{b - \sqrt{b^2 - ac}}{a}$$
$$t_2 = \frac{b + \sqrt{b^2 - ac}}{a} \tag{5.19}$$

Pokud a je blízké nule, vypočítáme kořeny lineárního polynomu $-2bt + c$ následovně:

$$t_{1,2} = \frac{c}{2b} \tag{5.20}$$

Pro hodnoty parametru $t \in [0, 1)$, pro které $C_x(t_i) \geq 0$ lze tímto způsobem najít průnik polopřímky s konturou. Jelikož jsou jednotlivé kontury glyfů uzavřené křivky, tak musíme vynechat $t = 1$, protože odpovídá parametru $t = 0$ křivce, která je další v pořadí, jinak by bylo možné průsečík započítat dvakrát, což by mohlo vést k nesprávné výsledné hodnotě navíjecího čísla.

Negativní nebo pozitivní přírůstek k navíjecímu číslu se dále vyhodnotí podle hodnoty $C_y(t)$ pro $t \in [0, t_i)$ nebo $t \in (t_i, 1)$.

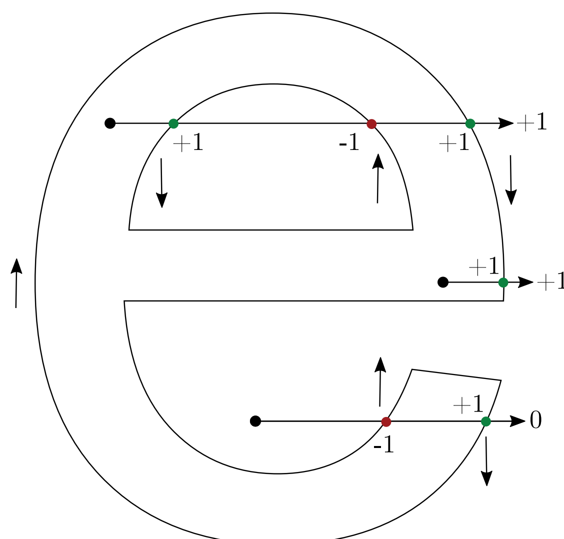
$C_y(t) > 0$ pro $t \in [0, t_i)$ nebo $C_y(t) < 0$ pro $t \in (t_i, 1)$	inkrementace
$C_y(t) < 0$ pro $t \in [0, t_i)$ nebo $C_y(t) > 0$ pro $t \in (t_i, 1)$	dekrementace

Vizualizaci výpočtu navíjecího čísla lze vidět na obrázku 5.6.

Tato čistě matematický popsaná metoda ovšem trpí v jakékoliv implementaci chybami v numerické přesnosti kdykoliv, kdy jsou ypsilonové hodnoty koncových řídících bodů $\mathbf{b}_0, \mathbf{b}_2$ blízké nule. Tyto chyby vznikají kvůli konečnému počtu bitů ve floating-point hodnotě a tudíž nelze poskytnout přesnost potřebnou pro výpočet kořenů t_1 a t_2. Tyto chyby způsobují artefakty v podobě pruhů nebo šumu, kvůli nesprávné výsledné hodnotě navíjecího čísla pro jednotlivé pixely.

Artefakty také vznikají pokud je křivka tečnou k polopřímce v prvním nebo v posledním řídícím bodě. Zaokrouhlovací triky nebo mírné posunutí vykreslovaných bodů, což se používalo u starších technik může tyto problémy v některých případech odstranit, avšak nevede k plně robustnímu řešení pro všechny vstupy.

Následující metoda vychází z doteď popsaného přístupu, ale nabízí přístup, který dosahuje robustnosti v konečné množině validních vstupů. Tato metoda nepotřebuje výpočet kořenů polynomu pro zjištění hodnoty navíjecího čísla a místo toho zjišťuje tuto hodnotu na základě binární klasifikace hodnot



Obrázek 5.6: Vizualizace výpočtu navíjecího čísla (*winding number*). Překresleno z [12].

y_0, y_1 a y_2. Podstatou je vytvořit třídy ekvivalence, které redukují problém do konečného počtu výstupů.

Hodnoty y_0, y_1 a y_2 mohou být kladné, záporné nebo nula, což odpovídá celkem 27 třídám ekvivalence. Tento počet lze ještě zredukovat budou-li se klasifikovat hodnoty y řídících bodů podle toho, zda jsou záporné nebo nezáporné.

Každou kvadratickou Bézierovu křivku tedy reprezentuje tří-bitová hodnota, která redukuje problém do 8 tříd ekvivalence podle hodnot jednotlivých bitů.

Zbývá tedy rozhodnout jak se bude měnit navíjecí číslo podle kořenů polynomu. Uvažujme derivaci polynomu 5.18:

$$C'_y(t) = 2at - 2b \tag{5.21}$$

$$a = y_0 - 2y_1 + y_2$$
$$b = y_0 - y_1$$

Po dosazení kořenů polynomu 5.19 do derivace:

$$\begin{aligned} C'_y(t_1) &= -2\sqrt{b^2 - ac} \\ C'_y(t_2) &= 2\sqrt{b^2 - ac} \end{aligned} \tag{5.22}$$

Pro kořeny polynomu s nenulovým diskriminantem $D = b^2 - ac$ podle 5.22 platí:

v t_1 křivka vždy prochází polopřímkou zleva doprava	inkrementace
v t_2 křivka vždy prochází polopřímkou zprava doleva	dekrementace

U každé třídy ekvivalence lze vytvořit pro tří-bitový vstup dvou-bitový výstup reprezentující změnu navíjecího čísla kořeny t_1 a t_2. Třídy ekvivalence jsou zobrazeny v tabulce 5.1 zobrazující všech 27 případů ve kterých $y_i < 0$, $y_i = 0$ a $y_i > 0$. Vstupní kód reprezentovaný bity y_i, určuje zda jsou křivky v dané třídě ekvivalence záporné či nezáporné. Výstupní kód reprezentovaný bity t_i určuje zdali nastal průsečík s osou x. Zelené body zaznamenávají inkrementaci navíjecího čísla a červené body dekrementaci navíjecího čísla.

Třída ekvivalence A zahrnuje případy, ve kterých jsou souřadnice y řídících bodů nezáporné a tedy nemají vliv na změnu navíjecího čísla. Opačným případem je třída ekvivalence H, ve které jsou všechny souřadnice y řídících bodů záporné a proto taktéž nemá vliv na změnu navíjecího čísla.

Ve třídách ekvivalence B a D křivky protínají osu v jednom bodě zprava doleva a dochází k dekrementaci navíjecího čísla. Třídy ekvivalence E a G jsou opačným případem, kdy křivky protínají osu v jednom bodě zprava doleva a dochází k inkrementaci navíjecího čísla.

Ve třídách ekvivalence C a F dochází ke dvěma průsečíkům díky kterým se inkrementace a dekrementace vzájemně vyruší. V těchto třídách ovšem nemusí dojít k průsečíku s osou a C_y nebude mít reálné kořeny. To lze vyřešit tak, že při výpočtu diskriminantu zařídíme, aby $D = max(0, b^2 - ac)$. To má za následek $t_1 = t_2 = b/c$ a oba kořeny tímto mají vliv na změnu navíjecího čísla a opět se vzájemně vyruší. Tento případ lze vidět v tabulce 5.1 u třídy ekvivalence F označený žlutým bodem.

Hodnoty v 5.1 ve sloupcích označených t_i tvoří 16-bitovou vyhledávací tabulku kterou lze hexadecimálně vyjádřit číslem `0x2E74`, ve kterém třída ekvivalence A odpovídá dvěma nejméně významným bitů a třída ekvivalence H dvěma nejvíce významným bitům. Dvojnásobek hodnot ve sloupcích označených y_i poslouží k pravé bitové rotaci hexadecimálního čísla `0x2E74`, ze kterého pak zjistíme třídu ekvivalence z nejmenších dvou bitů. Jestliže je nejmenší bit nastaven a $C_x(t_1) \geq 0$, inkrementujeme navíjecí číslo. jestliže je druhý nejmenší bit nastaven a $C_x(t_2) \geq 0$, dekrementujeme navíjecí číslo.

Tímto přístupem jsou eliminovány všechny problémy s numerickou přesností a výsledná hodnota navíjecího čísla je vždy správná.

5.2.2 Antialiasing

V této fázi algoritmus produkuje pouze diskrétní výstup - bod se nachází buď uvnitř glyfu nebo v prázdnem prostoru mimo něj. U glyfů jdou tudíž rozpoznat jednotlivé pixely a jejich šikmé části mají „zubatý“ vzhled.

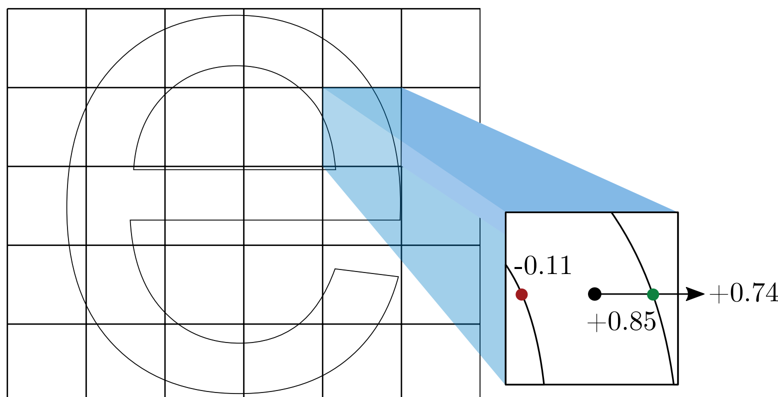
Třída	y_3	y_2	y_1	t_2	t_1	Reprezentativní křivky
A	0	0	0	0	0	
B	0	0	1	0	1	
C	0	1	0	1	1	
D	0	1	1	0	1	
E	1	0	0	1	0	
F	1	0	1	1	1	
G	1	1	0	1	0	
H	1	1	1	0	0	

Tabulka 5.1: Tabulka tříd ekvivalence pro klasifikaci kvadratických Bézierových křivek. Převzato z [11].

Pro odstranění tohoto aliasu je potřeba vytvořit funkci, která určí celkové pokrytí pixelu glyfem. Uvažujme, že počátek polopřímky je ve středu pixelu. Pokrytí pixelu f je dáno následující funkcí:

$$f = \text{clamp}\left(m\, C_x(t_i) + \frac{1}{2}\right) \tag{5.23}$$

$C_x(t_i)$ je souřadnice průniku přímky s konturou, m je počet pixelů na jednotku em a clamp je funkce, která zaručí, že výsledná hodnota bude v intervalu $[0, 1]$. Inkrementaci a dekrementaci nahradíme přičítáním a odčítáním těchto pokrytí od navíjecího čísla, které má efekt antialiasingu ve směru polopřímky. Vizualizace tohoto řešení je vidět na obrázku 5.7.



Obrázek 5.7: Výpočet částečného pokrytí pixelu glyfem. Překresleno z [12]

Pokud bychom chtěli antialiasing rozšířit do více směrů, což vede k lepšímu antialiasingu, stačí výsledné hodnoty pokrytí zprůměrovat. Využití více směrů vede ke snížení rychlosti algoritmu, proto je vhodným kompromisem uvažovat pouze směry rovnoběžné se souřadnicovými osami.

5.2.3 Optimalizace

Podle předchozího postupu je glyf korektně vykreslený a s použitím antialiasingu je dosaženo i správného vzhledu na obrazovce. Při implementaci tohoto algoritmu nedochází v shaderu k programovému větvení a je tímto dosaženo vysokého využití koherence vláken na GPU.

Většina testovaných křivek nemusí mít vliv na změnu navíjecího čísla, protože neprotnou horizontální nebo vertikální přímku procházející středem pixelu. Vysokého výkonu lze dosáhnout minimalizací počtu těchto testovaných křivek.

Myšlenkou této optimalizace je rozdělení bounding boxu glyfu na několik stejně širokých horizontálních a vertikálních pásů. Počet pásů je úměrný složitosti glyfu udávané v počtu křivek, ze kterých se glyf skládá. Pro každý pás je vytvořený seznam křivek, které jej protínají, seřazený sestupně podle maximální souřadnice řídících bodů x pro horizontální a y pro vertikální pásy. Pro vykreslovaný pixel se nejprve zjistí, do kterého pásu spadá. Dále algoritmus iteruje přes všechny křivky v tomto pásu a provádí výpočet navíjecího čísla. Iterace se provádí dokud nenarazí na křivku, pro kterou platí:

$$\max\{x_0, x_1, x_2\}\; m < -\frac{1}{2} \tag{5.24}$$

V této chvíli se může iterace zastavit, jelikož funkce 5.23 by při dalších iteracích vracela nulové hodnoty. Tento proces je obdobný pro iteraci přes vertikální pásy s rozdílem, že ve 5.24 vybíráme z maxima souřadnice y.

Pro pixely, které se nachází v levé nebo dolní části glyfu musí být testováno více křivek, než v pravé nebo horní části. Pásy je proto vhodné ještě rozdělit podle mediánu souřadnic řídících bodů křivek a testovat průniky i v opačném směru než pouze v pozitivním. Tímto je proces testování průniků více symetrický a pro jednotlivé pixely je počet testů lépe rozložený.

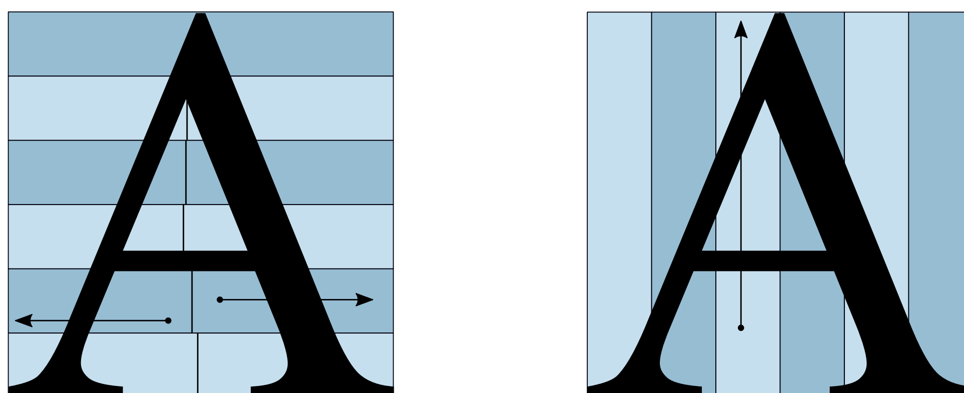
V negativním směru je pouze potřeba zaměnit funkci pokrytí pixelu 5.23 za:

$$f = \mathrm{clamp}\left(\frac{1}{2} - m\, C_x(t_i)\right) \tag{5.25}$$

V negativním směru je také potřeba seřadit křivky pro jednotlivé souřadnice vzestupně podle minimální souřadnice řídících bodů x pro horizontální a y pro vertikální. Iterace se pro negativní směr zastaví dokud nenarazí na křivku, pro kterou platí:

$$\min\{x_0, x_1, x_2\}\; m > \frac{1}{2} \tag{5.26}$$

Rozdělení glyfu na horizontální a vertikální pásy je zobrazeno ve 5.8. Pro horizontální pásy je také znázorněno rozdělení těchto pásů podle mediánu řídících bodů.



Obrázek 5.8: Glyf rozdělený na horizontální a vertikální pásy. Překresleno z [11].

5.2.4 Paměťová složitost a rychlost

Tato metoda vyžaduje jednodušší předzpracování na CPU, než metoda Loop/Blinn ovšem výpočet ve fragment shaderu je zde složitější. Předzpraco-

vání se skládá z načtení křivek z formátu TrueType do textury a vytvoření textury pro optimalizaci pomocí pásů. Pro každý glyf je potřeba poslat na GPU celkem šest vrcholů trojúhelníků uchovávající prostorové souřadnice x, y a texturovací souřadnice u, v. Tyto trojúhelníky tvoří obdélník, na který je glyf vykreslen a je jeho bounding boxem.

Takto předzpracovaná data mohou být ovšem uloženy na disk ve vhodném formátu a poté poslány rovnou na GPU. V porovnání s ostatními metodami využívající předrenderované glyfy nebo pole vzdáleností, které pouze samplují z textury, je opět zřejmé, že tato metoda bude pomalejší, kvůli složitosti výpočtu ve fragment shaderu.

Kapitola 6

Návrh řešení a implementace

Tato část popisuje návrh řešení pro architekturu knihovny, následnou implementaci obou metod a popis možnosti použití knihovny.

6.1 Návrh řešení

Požadavkem pro implementovanou knihovnu, co se týče funkcionality, bylo načtení fontu v daném formátu a vykreslení znaků se správným umístěním v textovém řetězci. Návrh této knihovny je inspirovaný knihovnou Slug [13], avšak s pokrytím pouze základních funkcionalit. Knihovna nejprve načte data z určeného fontu, vytvoří vertex buffer uchovávající data o vrcholech trojúhelníků a u metody Lengyel také textury, které jak je dále zmíněno, budou sloužit jako datové struktury potřebné pro výpočet. Tyto struktury budou následně odeslány na GPU, kde budou připraveny k použití. Při vykreslování pak aplikace nabinduje skrze API knihovny určitý buffer a texturu odpovídající danému glyfu, knihovna zvolí shader program a vykreslí trojúhelníky, na kterých se text zobrazuje nebo ze kterých se text skládá (dle použité metody). Cílem je aby knihovna podporovala ASCII.

Pro implementaci je zásadní načtení dat z fontu. Jelikož vytváření vlastního loaderu je časově náročné a není součástí požadavků, tak byla zvolena možnost využití knihovny FreeType, která standardně slouží pro vykreslování textu. Tato knihovna ovšem umožňuje také přístup k jednotlivým křivkám v konturách glyfu. Knihovna dovoluje načítat font v několika formátech včetně TrueType a OpenType, které v implementované knihovně budou podporovány.

Jednotlivé metody potřebují načtený font zpracovat jiným způsobem, jelikož každá vyžaduje rozdílné předzpracování a vytvoření datových struktur. Je tedy potřeba provést návrh knihovny tak, aby bylo možné určit pro jakou knihovnu bude daný font načtený.

Z tohoto důvodu je vhodné, aby každá z metod tvořila samostatnou třídu. Třídy by ovšem měly mít stejné základní funkcionality - načtení fontu a vykreslení textového řetězce. To lze zajistit vytvořením abstraktní třídy `TextRenderer` se dvěma virtuálními metodami pro načítání fontu a vykreslení

textového řetězce. Všechny třídy odvozené z této abstraktní třídy budou tedy muset implementovat tyto dvě funkce.

Z abstraktní třídy `TextRenderer` bude odvozena třída `GPUfriendlyRenderer` implementující metodu Loop/Blinn a `GPUonlyRenderer` implementující metodu Lengyel. Po vytvoření instancí těchto tříd si každá z metod zkompiluje své vlastní shadery, které používá pro vykreslování.

Každá ze tříd bude mít vlastní strukturu `Character`, která ukládá všechny potřebné data k vykreslení daného glyfu - VAO, VBO, textury, metriky. Tyto glyfy budou uložené ve struktuře `Character` budou poté uloženy do asociativního pole ve struktuře `FontFace`, která ukládá data pro daný font. Tato struktura bude navíc ukládat tabulku pro kerning. Každá instance tříd navíc bude uchovávat asociativní pole s uloženými `FontFace`, kde klíč je jméno fontu určené při jeho načítání. Knihovna tedy umožňuje načtení a vykreslení více různých fontů najednou.

Funkce pro vykreslení textového řetězce zaručí správné umístění jednotlivých znaků, vykreslí text na danou pozici ve 3D scéně o dané velikosti a rotaci kolem zvolené osy. Umožní také zvolit barvu textu, vykreslení textu vertikálně a použití kerningu.

Jestliže knihovna bude součástí PGR-frameworku, tak může být použita studenty při tvorbě semestrálních prací pro vykreslení textu přímo ve 3D scéně, k tvorbě UI, HUD nebo i výpisu aktuálních parametrů v aplikaci (FPS counter).

6.2 Implementace

Knihovna FreeType poskytuje funkci `FT_Outline_Decompose` pro dekompozici jednotlivých glyfů. Tato funkce má v argumentu pointer na obrys glyfu, který se skládá z více kontur a na strukturu `FT_Outline_Funcs`, která obsahuje funkce, které určují zpracování úsečky, kvadratické Bézierovy křivky, kubické Bézierovy křivky a přechodu na novou konturu. Tyto čtyři funkce si programátor implementuje podle potřeb dekompozice glyfu. Ukázky kódů fragment shaderu jsou napsány v jazyce GLSL. Pro kompilaci shaderů a operaci s nimi byla použita třída [20].

6.2.1 Metoda vykreslování kvadratických a kubických křivek nezávisle na rozlišení (Loop, Blinn)

Při dekompozici glyfu u této metody jsou vytvořeny celkem tři seznamy.

Prvním seznamem je vnitřní obrys glyfu. Tento vnitřní seznam obsahuje všechny body úseček definující konturu glyfu zpracované funkcí `lineToFunction` a také některé řídící body křivek, jak je popsáno v dalších odstavcích. Jelikož se glyf může skládat z více kontur, je nutné vytvořit také pole ukazatelů na počáteční body těchto podseznamů odpovídající konturám.

Pole ukazatelů je vytvářeno ve funkci `moveToFunction`, která zpracovává první bod v nové kontuře glyfu. Body každého podseznamu tvoří uzavřenou lomenou čáru.

Druhým seznamem je seznam kvadratických Bézierových křivek. Funkce `conicToFunction` zpracuje z kontury glyfu tři řídící body definující křivku a zavolá funkci, která tyto body klasifikuje. Při klasifikaci těchto bodů se určí, jestli se jedná o konvexní nebo konkávní křivku. Do prvního seznamu reprezentující vnitřní obrys glyfu je přidán první a poslední řídící bod, protože jimi křivka prochází. Pokud je navíc křivka konkávní, do prvního seznamu je přidán navíc i druhý řídící bod. Řídící body kvadratické Bézierovy křivky jsou tři, tudíž budou tyto body tvořit jeden trojúhelník.

Třetím seznamem je seznam kubických Bézierových křivek. Funkce `cubicToFunction` zpracuje z kontury glyfu čtyři řídící body definující tuto křivku. Do prvního seznamu reprezentující vnitřní obrys glyfu je přidán první a poslední řídící bod křivky, jelikož jimi křivka prochází. Dále je do tohoto seznamu přidán druhý a třetí řídící bod, podle toho jakou má orientaci vektor $\mathbf{b}_3 - \mathbf{b}_0$ vůči vektorům $\mathbf{b}_1 - \mathbf{b}_0$ a $\mathbf{b}_2 - \mathbf{b}_0$, což lze zjistit znaménkem determinantu dvojic vektorů. Tento výsledek je poté porovnán s orientací dané kontury (CW, CCW) a podle něj jsou přidány body do prvního seznamu. Důležité je také pořadí bodů $\mathbf{b}_1$, $\mathbf{b}_2$ přidaných do seznamu. Toto pořadí se určuje podle vzdálenosti od bodu $\mathbf{b}_0$ s tím, že je do seznamu přidán prioritně bližší bod.

Po vytvoření těchto tří seznamů je důležité odstranit překrývající se řídící polygony těchto křivek.

Každá kvadratická Bézierova křivka je testovaná na překrytí s ostatními kvadratickými křivkami a pokud je překrytí detekováno, tak je rozdělena křivka s větším obsahem na dvě menší křivky algoritmem De Casteljau v hodnotě parametru $t = 0.5$. Toto rozdělení ovšem může narušit seznam reprezentující vnitřní obrys glyfu. Pokud je takto rozdělena konvexní křivka, tak se na místo v seznamu mezi krajní řídící body rozdělované křivky musí přidat bod, ve kterém je tato křivka rozdělena. Pokud je takto rozdělena konkávní křivka, musí se odstranit ze seznamu druhý řídící bod. Na místo tohoto bodu musí být přidány další tři řídící body vzniklé rozdělením křivky ve vhodném pořadí. Překrytí je poté otestováno na dalších bodech definujících konturu glyfu, které netvoří kvadratickou křivku.

V tuto chvíli je seznam kvadratických křivek finální a může být zpracován do podoby vhodné pro výpočet v shaderech na GPU. Seznam je převeden do dvou polí (zvlášť pro konvexní a konkávní křivky) datového typu float a data jsou pro každý vrchol formátována jako prostorové souřadnice x, y a po nich následují texturovací souřadnice u, v. Tyto dvě pole poté představují dva samostatné vertex buffery. Souřadnice z je vždy nula, proto je přidána až ve vertex shaderu před aplikací transformací.

Vertex shader pro kvadratické křivky pouze umístí vrchol do prostoru a na svůj výstup pošle texturovací souřadnice u, v, které se dále perspektivně

interpolují.

Fragment shader pro kvadratické křivky nejprve vypočítá hardwarové gradienty texturovacích souřadnic pomocí GLSL funkcí `dFdx()` a `dFdy()` a aplikuje řetízkové pravidlo. Dále je vypočítaná vzdálenost ke křivce, která je namapována na alfa kanál výsledné barvy. Podle hodnoty alfa, se rozhodne, jaká je pozice pixelu vůči křivce - uvnitř, vně nebo na hranici. Fragment shader je stejný pro konvexní i konkávní kvadratické křivky. Podle toho, které křivky se zrovna vykreslují se nastaví hodnota `uniform float convex` na hodnotu 1 pro konvexní křivky a -1 pro konkávní. Pro konkávní křivky se tímto funkce $u^2 - v$ změní na $v - u^2$. Nastavení tohoto uniformu je rychlejší než přepnutí na jiný shader program. Použití tohoto uniformu pro změnu orientace je bohužel nutné, jelikož není způsob jak toto zařídit změnou znaménka texturovacích souřadnic (jako u metody vykreslující kubické křivky) kvůli druhé mocnině u souřadnice u. Kód fragment shaderu pro kvadratické křivky lze vidět ve zdrojovém kódu 6.1.

```
in vec2 p;
out vec4 fragColor;
uniform vec3 color;
uniform float convex;
void main()
{
  // gradients
  vec2 px = dFdx(p);
  vec2 py = dFdy(p);
  // chain rule
  float fx = (2.0f * p.x) * px.x - px.y;
  float fy = (2.0f * p.x) * py.x - py.y;
  // signed distance
  float sd = convex * (p.x * p.x - p.y) /
             sqrt(fx * fx + fy * fy);
  // mapping to opacity
  float alpha = 0.5f - sd;
  if(alpha >= 1.0f) {
    // inside
    fragColor = vec4(color, 1.0f);
  } else if (alpha < 0.0f) {
    // outside
    discard;
  } else {
    // boundary
    fragColor = vec4(color, alpha);
  }
}
```

Zdrojový kód 6.1: Fragment shader pro kvadratické Bézierovy křivky.

Překrytí pro kubické Bézierovy křivky je otestováno podobným způsobem jako u kvadratických křivek. Ty je ale potřeba dále klasifikovat a provést triangulaci, jelikož tento seznam netvoří trojúhelníky.

Klasifikace kubických křivek na tři typy je provedena, tak jak je popsáno v kapitole 5.

Uspořádání řídícího polygonu lze zjistit průsečíkem diagonál. Například pokud existuje průnik úsečky tvořené body $\mathbf{b}_0$, $\mathbf{b}_2$ s úsečkou tvořenou body $\mathbf{b}_1$, $\mathbf{b}_3$, tak jde o první uspořádání z obrázku 5.5. Po zjištění tohoto uspořádání je potřeba zvolit diagonálu, která rozdělí tento řídící polygon na dva trojúhelníky. Řídící polygon nemusí být vždy konvexní, proto je tedy potřeba zvolit kratší ze dvou diagonál konvexního obalu řídících bodů, protože ta je vždy stranou společnou pro dva trojúhelníky.

Po klasifikaci kubických křivek se vytvoří pole datového typu float a data jsou pro každý vrchol formátována jako prostorové souřadnice x, y a po nich následují texturovací souřadnice k, m a l. Toto pole poté představuje samostatný vertex buffer.

Princip vertex shaderu i fragment shaderu je podobný jako u kvadratických Bézierových křivek. zde je ovšem nutné změnit znaménko funkce vyjadřující vzdálenost od kontury, kvůli orientaci kontur ve formátu OpenType.

Kód fragment shaderu pro kubické křivky lze vidět ve zdrojovém kódu 6.2.

```
in vec3 p;
out vec4 fragColor;
uniform vec3 color;
void main()
{
  // gradients
  vec3 px = dFdx(p);
  vec3 py = dFdy(p);
  // chain rule
  float fx = (3.0f * p.x * p.x) * px.x
             - p.z * px.y - p.y * px.z;
  float fy = (3.0f * p.x * p.x) * py.x
             - p.z * py.y - p.y * py.z;
  // signed distance
  float sd = -(p.x * p.x * p.x - p.y * p.z)
             / sqrt(fx * fx + fy * fy);
  // mapping to opacity
  float alpha = 0.5f - sd;
  if(alpha >= 1.0f) {
    // inside
    fragColor = vec4(color, 1.0f);
  } else if (alpha < 0.0f) {
    // outside
    discard;
```

```
  } else {
    // boundary
    fragColor = vec4(color, alpha);
  }
}
```

Zdrojový kód 6.2: Fragment shader pro kubické Bézierovy křivky.

Zbývá tedy vykreslit vnitřek glyfu, který je potřeba převést na trojúhelníky. Tento převod lze provést vázanou Delaunyho triangulací.

Pro triangulaci jsem použil knihovnu CDT: Constrained Delaunay Triangulation [1], která poskytuje vázanou Delaunyho triangulaci. Na vstupu této triangulace je pouze potřeba definovat jednotlivé body a hrany, které je spojují.

Výstupem je tedy trojúhelníková síť tvořená body a trojicemi indexů, definující jednotlivé trojúhelníky. Vnitřní trojúhelníky jsou vykresleny v shaderu pro kvadratické křivky přiřazením stejných texturovacích souřadnic $(u, v) = (0, 1)$ pro každý vrchol trojúhelníku. Tyto souřadnice zaručí, že je celý trojúhelník vyplněný a není potřeba přepínat na samostatný shader. Je ovšem potřeba nastavit uniform pro konvexní křivky, aby souřadnice odpovídaly vnitřku křivky. Prostorové souřadnice vrcholů jsou uloženy do samostatného vertex bufferu a indexy vrcholů do element bufferu.

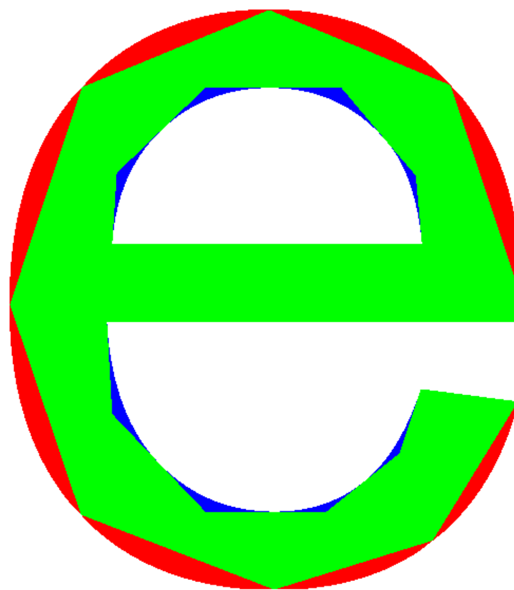
Glyf vykreslený shaderem pro kvadratické křivky lze vidět na obrázku 6.1. Konvexní trojúhelníky jsou zde označeny červeně, konkávní modře a vnitřní zeleně.

Při zpracování některých fontů ovšem může dojít k problémům, které je třeba vyřešit.

Prvním problémem je, že kontura může obsahovat smyčku tvořenou z více křivek. To vede k výsledku, že lomená čára definující vnitřek glyfu obsahuje smyčku a tu je potřeba odstranit. Delaunyho triangulace, by jinak nerozeznala vnitřek glyfu. Tato smyčka se odstraní tak, že se prochází seznam úseček lomené čáry a testuje se, zda aktuální úsečka neprotíná jinou úsečku ze seznamu, která již byla zpracována. Pokud je pokud je průnik detekován, tak se odstraní ze seznamu všechny body až do bodu, který je počátečním bodem pro úsečku, s kterou byl průnik detekován. Dále je přidán do seznamu průsečík těchto dvou úseček a pokračuje se v procházení seznamu.

Dalším problém je, že se jednotlivé kontury mohou překrývat. Tento problém je vyřešen tak, že se prochází seznam lomené čáry dané kontury a jakmile je detekováno překrytí s jinou, již zpracovanou konturou, tak se kontura odstraní. Tento problém se vyskytuje u některých velmi složitých fontů. Jelikož jsou kontury většinou definované od největších po nejmenší a toto překrytí nastává u nejmenších kontur, tak jsou odstraněny vždy menší kontury, což nemá tak velký vliv na vzhled daného glyfu.

Některé glyfy také obsahují duplicitní body. Pokud jsou tyto duplicitní



Obrázek 6.1: Glyf vykreslený metodou Loop/Blinn bez použití antialiasingu s barevně odlišenými komponentami.

body obsaženy v seznamu vnitřní kontury glyfu, taktéž nemůže být provedena Delaunyho triangulace. Pro přípravu dat pro triangulaci je tedy důležité duplicitní body detekovat, pro seznam hran využít stávající body a nepřidávat již body v seznamu obsažené.

Po odstranění těchto problémů lze glyf korektně vykreslit. Antialiasing ovšem funguje pouze pro trojúhelníky obsahující křivku, jelikož se vzdálenost od křivky počítá pouze v nich. Pro vnitřní trojúhelníky, jejichž strany jsou hranicí glyfu, je alias stále přítomný. Alias se také vyskytuje v některých rozích trojúhelníků obsahující křivku, pokud tento roh představuje první nebo poslední řídící bod křivky. V článku [14] autoři popisují několik způsobů, jak se tohoto aliasu zbavit.

Prvním možností je přidat dodatečnou geometrii kolem glyfu v podobě tenkého pruhu tvořeného trojúhelníky. V těchto trojúhelnících by se počítala vzdálenost od úseček a bodů, které jsou hranicí glyfu. Šířka tohoto pruhu ovšem závisí na projekci do screen space. Další možností je vytvoření bounding boxu glyfu, který je na každé straně o 10% větší, než je velikost glyfu. U obou těchto řešení se ale může stát, že některé pixely nebudou vykresleny korektně.

Nejjednodušším řešením jak se zbavit aliasu je využít techniky multisample antialiasng (MSAA). Pro tuto metodu se určí počet vzorků (*sample*) pro daný pixel. Color, depth a stencil buffer ukládá hodnotu pro každý z těchto vzorků, což znamená, že se velikost těchto bufferů zvětší $N\times$ pro N vzorků na pixel. Rasterizátor vyhodnotí kolik vzorků je v pixelu pokryto daným trojúhelníkem, spustí fragment shader a data jsou uložena pro každý z pokrytých vzorků zvlášť. Fragment shader se spustí pokud je alespoň jeden vzorek pokrytý daným trojúhelníkem a to pouze jednou, na rozdíl od jiných metod jako super-

sampling antialiasing (SSAA), čímž snižuje výpočetní náročnost antialiasingu. Pro MSAA o N vzorcích můžeme tedy vypočítat pokrytí jednoho pixelu až N trojúhelníky. Výsledná barva pixelu je na konci procesu zprůměrována ze všech vzorků. Pro potřebu antialisingu u této metody se ukázalo, že je dostačující použít 4 vzorky. To jaký je použitý vzor rozmístění těchto vzorků v pixelu je hardwarově závislé.

6.2.2 Metoda přímého vykreslování textu (Lengyel)

Při dekompozici glyfu u této metody je vytvářen pouze jeden seznam křivek. Tato metoda předpokládá, že je kontura glyfu definována pouze kvadratickými Bézierovými křivkami a proto může načítat pouze font formátu TrueType.

Funkce `moveToFunction` přidává počáteční bod dané kontury do seznamu.

Jelikož je poslední řídící bod dané křivky identický s prvním řídícím bodem křivky další v pořadí, tak funkce `conicToFunction`, zpracovávající kvadratické křivky, přidává pouze druhý a třetí řídící bod dané křivky do seznamu. Toto platí ovšem jen pro křivky ve stejné kontuře.

Funkce `lineToFunction`, zpracovávající úsečky, převádí úsečku na kvadratickou Bézierovu křivku vytvořením nového bodu v jejím středu.

Vytvořený seznam je potřeba uložit do textury, aby mohly být křivky zpracovány ve fragment shaderu. Jeden texel této textury se skládá ze čtyř 16-bitových kanálů datového typu `float`. Souřadnice prvního řídícího bodu křivky jsou uložené v komponentě (x, y) daného texelu a souřadnice druhého řídícího bodu dané křivky v komponentě (z, w). Souřadnice třetího řídícího bodu jsou uložené v komponentě (x, y) dalšího texelu. Pokud nastane přechod na novou konturu, tak je uložen poslední bod dané kontury v komponentě (x, y), komponenta (z, w) zůstává prázdná a zápis se provádí od následujícího texelu. Strukturu této textury lze vidět na obrázku 6.2.

R (16 bit)	G (16 bit)	B (16 bit)	A (16 bit)
$\mathbf{b}_0$.x (curve 1)	$\mathbf{b}_0$.y (curve 1)	$\mathbf{b}_1$.x (curve 1)	$\mathbf{b}_1$.y (curve 1)
$\mathbf{b}_2$.x (curve 1) $\mathbf{b}_0$.x (curve 2)	$\mathbf{b}_2$.y (curve 1) $\mathbf{b}_0$.y (curve 2)	$\mathbf{b}_1$.x (curve 2)	$\mathbf{b}_1$.y (curve 2)

⋮

Obrázek 6.2: Struktura textury uchovávající seznam křivek. Převzato z [11]

Je potřeba dále vytvořit datovou strukturu, která bude indexovat křivky z jednotlivých pásů určených k optimalizaci výpočtu. Tato struktura je taktéž uložena do textury tvořené čtyřmi 16-bitovými kanály, ale tentokrát datového typu `integer`. Tato textura má celkem tři části.

První část této textury uchovává data pro horizontální pásy s tím, že každý

texel odpovídá jednomu pásu. V prvním kanálu je uložený počet křivek, který se v daném pásu nachází. V druhém kanálu je uložený offset do třetí části textury udávající počátek seznamu křivek v daném pásu. Ve třetím a čtvrtém kanálu je uložena hodnota, ve které je pás rozdělen na dvě části.

Druhá část textury má stejnou strukturu jako první část, ale uchovává data pro vertikální pásy.

Třetí část textury uchovává ukazatele na křivky z první textury. V prvních dvou kanálech texelu je ukazatel na křivku pro vzestupně seřazený seznam a v posledních dvou ukazatel na křivku pro sestupně seřazený seznam. Strukturu této textury lze vidět na obrázku 6.3.

R (16 bit)	G (16 bit)	B (16 bit)	A (16 bit)
H band curve count	H band data offset	H band split value	
H band curve count	H band data offset	H band split value	
⋮			
V band curve count	V band data offset	V band split value	
V band curve count	V band data offset	V band split value	
⋮			
Curve location, ascending sort		Curve location, decending sort	
Curve location, ascending sort		Curve location, decending sort	

Obrázek 6.3: Struktura textury uchovávající data pro pásy. Převzato z [11]

Dalším krokem je vytvořit trojúhelníky, na kterých má být glyf vykreslen. Každý glyf je vykreslen na obdélníku složeném ze dvou trojúhelníků. Prostorové a texturovací souřadnice jsou zkonstruovány podle šířky a výšky glyfu a jeho odsazení od osy x a y v em space.

Vertex shader po aplikaci transformací umístí bod do prostoru a pošle texturovací souřadnice představující souřadnice v em space do fragment shaderu, které se perspektivně interpolují.

Ve fragment shaderu se nejprve určí, do kterého horizontálního pásu vykreslovaný bod patří podle vzdálenosti prvního pásu od osy x a jeho šířky. Tímto

získáme index do textury pro daný pás a můžeme z ní získat hodnotu počtu křivek v pásu, offset do seřazeného seznamu indexů a hodnotu rozdělující pás na dvě části.

Podle rozdělující hodnoty a souřadnice x daného pixelu se dále určí, jakým směrem bude testován průnik s polopřímkou. Poté následuje iterace přes všechny křivky podle směru polopřímky. V každé iteraci se nejprve získají souřadnice tří řídících bodů křivky do vektorů `vec4 b01` a `vec2 b2`. Poté se vyhodnotí, zda je potřeba dále procházet seznam křivek podle 5.24 a 5.26. Následuje výpočet kódu, ze kterého lze určit klasifikaci křivky. Tento výpočet je prováděn v samostatné funkci 6.3.

```
uint calcRootCode(float y0, float y1, float y2) {
  return (0x2E74U >> (
           ((y0 > 0.0) ? 2U : 0U)
         + ((y1 > 0.0) ? 4U : 0U)
         + ((y2 > 0.0) ? 8U : 0U)
         )) & 3U;
}
```

Zdrojový kód 6.3: Funkce fragment shaderu pro výpočet kódu klasifikující křivku.

Pokud tato funkce nevrátí hodnotu 0, tak může nastat změna při výpočtu pokrytí pixelu. Musí být tedy proveden výpočet průsečíků křivky s osou x. Tento výpočet je prováděn v samostatné funkci 6.4.

```
vec2 solvePolyHorizontal(vec4 b01, vec2 b2) {
  // solve values of t for polynomial
  // at^2 - 2bt + c
  // where curve crosses x axis
  float ax = b01.x - b01.z * 2.0 + b2.x;
  float ay = b01.y - b01.w * 2.0 + b2.y;
  float bx = b01.x - b01.z;
  float by = b01.y - b01.w;
  float da = 1.0 / ay;
  float d = sqrt(max(by * by - ay * b01.y, 0.0));
  float t1 = da * (by - d);
  float t2 = da * (by + d);
  if (abs(ay) < epsilon) {
    // polynomial is nearly linear
    // solve -2bt + c
    t1 = b01.y * 0.5 / by;
    t2 = b01.y * 0.5 / by;
  }
  return vec2(
    (ax * t1 - bx * 2.0) * t1 + b01.x,
    (ax * t2 - bx * 2.0) * t2 + b01.x);
}
```

Zdrojový kód 6.4: Funkce fragment shaderu pro výpočet průsečíku křivky s horizontální osou.

Tato funkce vrací souřadnice x dvou možných průsečíků. Následně je vypočítáno pokrytí funkcemi 5.23 pro pozitivní směr, nebo 5.25 pro negativní směr. Podle posledních dvou bitů výsledného kódu je pokrytí přičteno nebo odečteno.

Ukázka zjednodušené části tohoto výpočtu fragment shaderu pro pozitivní horizontální směr lze vidět ve zrojovém kódu 6.5.

```
for(all curves in band) {
  // fetch 2D control points for the current curve
  vec4 b01 = texelFetch(curveTex, ivec2(curveLoc, 0))
             - vec4(texCoord, texCoord);
  vec2 b2 = texelFetch(curveTex, ivec2((curveLoc + 1U), 0)).xy
            - texCoord;
  // check if all three control points falls left of
  // the current pixel
  if(max(max(b01.x, b01.z), b2.x) * pixelsInEm.x < -0.5) break;
  // get root code based on the signs
  // of the y coordinates
  uint code = calcRootCode(b01.y, b01.w, b2.y);
  if(code != 0U) {
    // at least one root makes a contribution
    // in winding number
    vec2 x = solvePolyHorizontal(b01, b2);
    // transform results to [0,1]
    x.x = clamp(x.x * pixelsInEm.x + 0.5, 0.0f, 1.0f);
    x.y = clamp(x.y * pixelsInEm.x + 0.5, 0.0f, 1.0f);
    // update winding number
    if ((code & 1U) != 0U) {
      coverage += x.x;
    }
    if (code > 1U) {
      coverage -= x.y;
    }
  }
}
```

Zdrojový kód 6.5: Zjednodušená část fragment shaderu pro výpočet pokrytí pixelu.

Tento výpočet je obdobný pro vertikální pásy a negativní směry.

Absolutní hodnoty pokrytí pro vertikální a horizontální směry jsou poté

zprůměrovány. Následně je hodnota odmocněna (aproximace gamma korekce) a je jí vynásobena hodnota alfa barvy výsledného pixelu.

6.2.3 Knihovna

Knihovna je implementována podle návrhu v podkapitole 6.1. Jejím jádrem jsou metody pro vytvoření datových struktur potřebných pro správnou reprezentace glyfů. Použití knihovny se omezuje na dvě metody.

První metodou je metoda `loadFont`, která slouží k načtení fontu, vytvoření datových struktur a vertex bufferů potřebných pro funkci dané metody vykreslování. Tato metoda má v argumentu název `string fontName`, který je dále použitý jako klíč ve struktuře ukládající data jednotlivých fontů a cestu k souboru fontu `const char *path`.

Druhou metodou je metoda `renderText`. Tato metoda slouží k vykreslení textového řetězce a jeho umístění do prostoru. Jejími argumenty jsou textový řetězec `string text`, který má být vykreslen, dále název fontu, který byl určen při jeho načítání. Dalšími argumenty jsou ukazatale na dvě matice `glm::mat4 *viewMatrix` a `glm::mat4 *`, které jsou potřebné ve vertex shaderu, aby mohla být provedena aplikace transformací v závislosti na pozici kamery a projekci. Následuje argument `glm::vec3 position` určující pozici x, y a z v prostoru, kde má počátek vykreslovaného řetězce začínat. Následujícím argumentem je velikost písma `GLfloat scale`. Dalším argumentem je úhel `GLfloat rotationAngle` o který se má text otočit okolo osy datového typu `glm::vec3 rotationAxis` určené v následujícím argumentu. Následující argument je určení barvy textu `glm::vec3 color` ve formátu RGB. Posledními dvěma argumenty `GLboolean vertical`, `GLboolean kerning` lze nastavit použití vertikálního vykreslování textu a kerningu (pokud je daný font podporuje).

Při použití knihovny stačí vytvořit instanci dané metody, načíst font a vykreslit text s danými parametry. Ukázka vykreslení textového řetězce je ve zdrojovém kódu 6.6.

```
#include "GPUonlyRenderer"
using namespace textrendering;
glm::mat4 projection = glm::perspective(...);
glm::mat4 view = glm::lookAt(...);
GPUonlyRenderer tr = GPUonlyRenderer();
tr.loadFont("arial", "fonts/arial.ttf");
tr.renderText("abcdefg",                    // text
              "arial",                      // font name
              &projection,                  // projection matrix
              &view,                        // view matrix
              glm::vec3(1.0f, 1.0f, 0.0f),// position
              1.0f,                         // scale
              0.0f,                         // rotation angle
```

```
glm::vec3(0.0f, 1.0f, 0.0f),// rotation axis
glm::vec3(0.0f, 0.0f, 0.0f),// RGB color
false,                      // vertical layout
true                        // kerning
);
```

Zdrojový kód 6.6: Příklad použití knihovny.

Pro metodu Loop/Blinn je ovšem potřeba pro kompletní antialiasing nastavit multisampling. Při použití GLFW toto lze provést dvěma příkazy `glfwWindowHint(GLFW_SAMPLES, 4)` a `glEnable(GL_MULTISAMPLE)` pro 4 samply na pixel.

Kapitola 7

Výsledky

V této kapitole jsou představeny výsledky implementací obou metod, měření rychlosti vykreslování a prostorové složitosti. Výsledky implementací jsou následně vzájemně porovnány a nakonec je představena ukázková aplikace demonstrující funkce knihovny.

7.1 Metoda vykreslování kvadratických a kubických křivek nezávisle na rozlišení (Loop, Blinn)

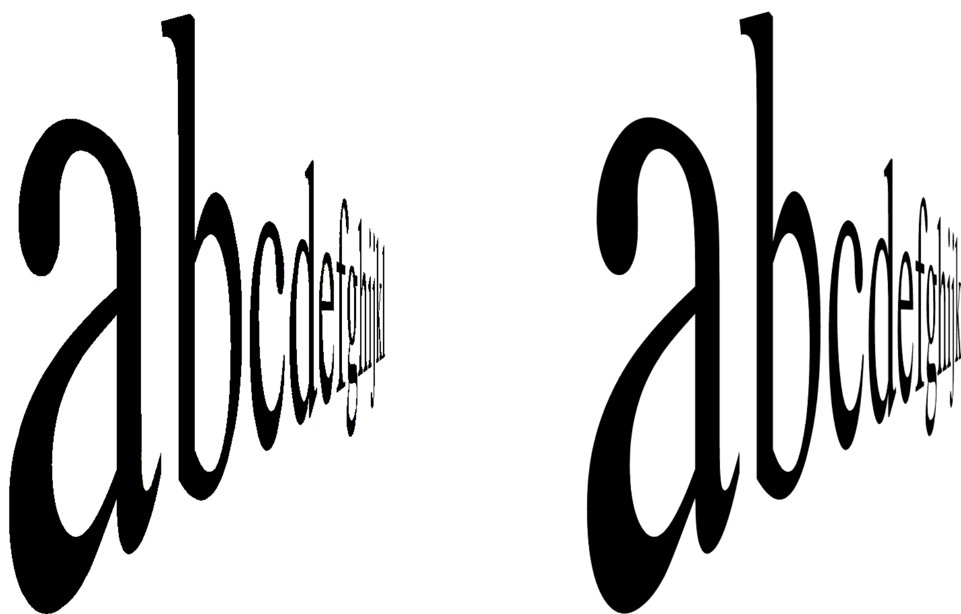
Tato metoda byla zdárně implementována a dovoluje korektně vykreslit většinu fontů formátu TrueType i OpenType. Glyfy jsou vykresleny bez aliasu a je možné je libovolně zvětšovat i zmenšovat. Vykreslení bylo otestováno na několika fontech různé složitosti i velikosti, jak je vidět na obrázku 7.1.

Obrázek 7.1: Ukázka různých fontů vykreslených metodou Blinn/Loop.

U těchto fontů nebyla zpozorována žádná chyba při vykreslování ani při detailnějším pohledu na jednotlivé glyfy 7.2.

Obrázek 7.2: Detail glyfu vykreslený metodou Loop/Blinn.

Pro antialiasing trojúhelníků neobsahujících křivku je u této metody použita technika MSAA. Rozdíly v jejím použití lze vidět na obrázku 7.3. Jak je vidět na tomto obrázku, tak je text s použitím této metody správně vykreslený i v záběru perspektivní kamerou.



Obrázek 7.3: Porovnání diskrétního výstupu s aliasem a výstupu bez aliasu dosaženého kombinací použití MSAA a výpočtu vzdálenosti od křivky.

Jak již bylo zmíněno v implementační části, metoda korektně nevykreslí

glyfy, ve kterých dochází k překrývání jednotlivých kontur. Tyto překrývající se kontury jsou po detekování odstraněny, aby mohla být korektně provedena Delaunyho triangulace a mohl být glyf alespoň částečně vykreslen.

Dále bylo provedeno měření rychlosti vykreslování fontů o různé složitosti. Měření byl provedno výpisem 50 řetězců „ABCDEFG“ na náhodné pozice při velikosti textu 32 pixelů na em, v okně velikosti 1920 × 1080 pixelů. Složitost představuje průměrnou hodnotu počtu trojúhelníků na glyf. Měření bylo provedeno na grafické kartě AMD Radeon™ 530. Výsledky měření jsou v tabulce 7.1.

Font	Text	Složitost	Čas
Centaur	ABCDEFG	93	3.9 ms
Jokerman	ABCDEFG	90	6.4 ms
Halloween	ABCDEFG	131	6.7 ms
Wildwood	ABCDEFG	799	24.2 ms
Spider	ABCDEFG	1734	30.1 ms

Tabulka 7.1: Měření doby vykreslování textu metodou Loop/Blinn.

V tabulce 7.1 je možné vidět, jak se doba vykreslování drasticky snižuje při větší složitostí fontu.

Bylo také provedeno měření prostorové složitosti. Jelikož se každý glyf skládá z komponent definující trojúhelníky, tak má na prostorovou složitost vliv vertex bufferu (VBO) uchovávající informace o vrcholech. Každý vrchol zde ukládá prostorové souřadnice x, y a texturovací souřadnice u, v nebo k, l, m, vše v datovém typu `float` (32 bit). Důležitá je také velikost element bufferu (EBO), který uchovává indexy vrcholů pro konstrukci trojúhelníků uvnitř glyfu v datovém typu `unsigned int`. Velikost těchto bufferů po načtení 26 velkých písmen anglické abecedy lze vidět v tabulce 7.2. Složitost textu je určena průměrnou hodnotou počtu trojúhelníku na glyf a velikosti v KB jsou zaokrouhlené na jednotky nahoru.

Font	Složitost	VBO (KB)	EBO (KB)
Centaur	93	128	37
Jokerman	90	98	30
Halloween	131	162	48
Wildwood	799	838	294
Spider	1734	1704	538

Tabulka 7.2: Měření prostorové složitosti textu metody Loop/Blinn.

7.2 Metoda přímého vykreslování textu (Lengyel)

Tato metoda byla zdárně implementována a dovoluje korektně vykreslit všechny fonty formátu TrueType. Glyfy jsou vykresleny bez aliasu a je možné je libovolně zvětšovat a zmenšovat. Vykreslování bylo otestováno, stejně jako u předchozí metody, na mnoha fontech o různé složitosti a velikosti. Ukázku vykreslení různých fontů touto metodou lze vidět na obrázku 7.4.

Obrázek 7.4: Ukázka různých fontů vykreslených metodou Lengyel.

U těchto fontů nebyla zpozorována žádná chyba při vykreslování ani při detailnějším pohledu na složitější glyfy jak je vidět na obrázku 7.5.

Glyfy jsou korektně vykresleny i v záběru perspektivní kamerou 7.6 Korektní vykreslování se ovšem podotýká s několika dalšími problémy.

Prvním problémem je, že jsou glyfy vykresleny do bounding boxu, jehož hrany se dotýkají kontury glyfu. Na těchto hranách probíhá špatný výpočet

Obrázek 7.5: Detail glyfu vykreslený metodou Lengyel.

antialiasingu a může se ztratit pokrytí až 50 % pixelu. Je tedy potřeba zvětšit tento bounding box o půl pixelu z každé strany, což lze provést podle dané velikosti textu. Velikost na obrazovce se ovšem může měnit při pohybu kamery, takže je potřeba tuto změnu velikosti udělat adaptivní vůči poloze kamery, což se mi zatím nepodařilo implementovat.

Druhým problémem je, že pokud je text zmenšený a pixel na obrazovce ve výsledku obsahuje hodně detailů daného glyfu, tak je potřeba zvětšit počet samplů v pixelu. Bylo vyzkoušeno, že 4 samply mohou toto dostatečně pokrýt. Více samplů ovšem snižuje výkon a bylo by potřeba jejich počet určovat adaptivně, což zatím nebylo implementováno. Tento problém se také vyskytuje pokud se vykreslí hodně detailní glyf.

Dále bylo provedeno měření rychlosti vykreslování fontů o různé složitosti. Měření byl provedno výpisem 50 řetězců „ABCDEFG“ na náhodné pozice při velikosti textu 32 pixelů na em, v okně velikosti 1920 × 1080 pixelů. Složitost představuje průměrnou hodnotu počtu kvadratických Bézierových křivek na glyf. Měření bylo provedeno na grafické kartě AMD Radeon™ 530. Výsledky měření jsou v tabulce 7.3.

Obrázek 7.6: Text v perspektivě vykreslený metodou Lengyel.

Font	Text	Složitost	Čas
Centaur	ABCDEFG	48	2.2 ms
Jokerman	ABCDEFG	60	3.0 ms
Halloween	ABCDEFG	68	3.1 ms
Wildwood	ABCDEFG	546	23.5 ms
Spider	ABCDEFG	666	29.6 ms

Tabulka 7.3: Měření doby vykreslování textu metody Lengyel.

Z tabulky 7.3 lze vypozorovat jak se doba vykreslování snižuje se složitostí fontu. Optimalizace, která byla provedena v této metodě pomocí rozdělení glyfu na pásy velmi pomohla snížit rychlost vykreslování. Počet pásů je úměrný složitosti glyfu a je zvolen od 1 do 16 a má velký vliv na rychlost vykreslování.

U metody byla také změřena paměťová složitost. Vertex buffer pro každý glyf má vždy stejnou velikost. Jde o dva trojúhelníky skládající se z šesti vrcholů. Každý vrchol ukládá prostorové souřadnice x, y a texturovací souřadnice u, v. Tyto souřadnice jsou datového typu `float` (32 bit) a pokud je načteno 26 písmen anglické abecedy, velikost vertex bufferu bude 2.5 KB pro každý font. Rozdíly u jednotlivých fontů jsou až u velikosti dvou textur uklá-

dající data o křivkách a pásech. Velikosti těchto textur po načtení 26 velkých písmen anglické abecedy lze vidět v tabulce 7.4. Složitost textu je určena průměrnou hodnotou počtu křivek na glyf a velikosti v KB jsou zaokrouhlené na jednotky nahoru.

Font	Složitost	Textura křivek (KB)	Textura pásů (KB)
Centaur	48	11	31
Jokerman	60	11	32
Halloween	68	17	48
Wildwood	546	129	308
Spider	666	150	356

Tabulka 7.4: Měření prostorové složitosti textu metodou Lengyel.

7.3 Porovnání výsledků obou metod

Obě metody dovolují vykreslit text nezávisle na rozlišení a v záběru perspektivní kamerou. Rozdíl mezi vykreslenými texty při standardní velikosti (32 pixel na em) téměř nelze rozpoznat. Metoda Lengyel ovšem v některých úhlech kamery produkuje jemný alias, který lze ovšem vyřešit zvýšením počtu samplů ve fragment shaderu.

Jak již bylo ale zmíněno, metoda Loop/Blinn odstraňuje u složitějších fontů překrývající se kontury, proto je vhodnější pro složitější fonty použít metodu Lengyel. Lengyelova metoda má ovšem problém s dilatací bounding boxu, v některých úhlech při pohybu kamery, což vede ke špatnému vykreslení krajních pixelů. Lengyelova metoda také hůře vykresluje velmi malý text, pokud není zvětšený počet samplů. Metoda Loop/Blinn má také výhodu, že dovoluje načítat větší množství fontů, jelikož na rozdíl od metody Lengyel dovoluje načítat font formátu OpenType. Co se týče doby předzpracování, tak je v tomto ohledu lepší metoda Lengyel, která vyžaduje minimální předzpracování v řádech sekund. U metody Loop/Blinn může předzpracování trvat i několik minut.

U obou metod bylo provedeno měření rychlosti vykreslování. Výsledky jsou v tabulkách 7.1 a 7.3. Při porovnání těchto výsledků je zřejmé, že aktuální implementace metody Lengyel je rychlejší než metoda Loop/Blinn. Metoda Loop/Blinn si také vede při rychlosti vykreslování hůře, pokud je text zmenšený. Možným důvodem je, že trojúhelníky, ze kterých se glyf skládá, se stanou velmi malými, což může vést k problému neefektivního využití skupin vláken na GPU.

U obou metod bylo také provedeno měření paměťové složitosti. Tyto výsledky jsou v tabulkách 7.2 a 7.4. Aktuální implementace metody Lengyel potřebuje menší množství paměti pro vykreslování než metoda Loop/Blinn. Důvodem je, že metoda Loop/Blinn rozděluje složitější glyfy na velké množství komponent. Pro velmi složité glyfy jich může být až tisíce.

7.4 Ukázková aplikace

Součástí zadání bylo vytvoření ukázkové aplikace demonstrující možné použití knihovny. Vytvořil jsem dvě ukázkové aplikace pro každou metodu zvlášť. Každá z ukázkových aplikací se skládá ze tří stránek.

Centaur	ABCDEFGHIJKLMNOPQRSTUVWXYZ
Jokerman	ABCDEFGHIJKLMNOPQRSTUVWXYZ
Halloween	ABCDEFGHIJKLMNOPQRSTUVWXYZ
Royal	ABCDEFGHIJKLMNOPQRSTUVWXYZ
Scratch	ABCDEFGHIJKLMNOPQRSTUVWXYZ
Spider	ABCDEFGHIJKLMNOPQRSTUVWXYZ
Wildwood	ABCDEFGHIJKLMNOPQRSTUVWXYZ

Obrázek 7.7: Ukázka aplikace demonstrující možné použití knihovny.

První stránka představuje ukázku vykreslování různých fontů, aby bylo vidět široké spektrum složitosti fontů, které je daná metoda schopná vykreslit 7.7.

Druhá stránka vykresluje odstavec menšího textu, aby bylo vidět jak si metoda poradí s vykreslováním odstavce s větším množstvím textu 7.8. Třetí stránka ukazuje dodatečnou funkcionalitu použití kerningu 7.9.

V ukázkových aplikacích lze pohybovat kamerou pomocí myši. Kamerou je možné posunovat do stran, otáčet kolem textu a provádět zoom. Díky tomuto ovládání je možné si text na každé stránce prohlédnou zblízka, čímž se demonstruje nezávislost metod na rozlišení. Otáčení demonstruje pohled na text perspektivní kamerou.

Obrázek 7.8: Odstavec textu vykreslený v perspektivním zobrazení v ukázkové aplikaci.

Obrázek 7.9: Text při použití kerningu.

Kapitola 8

Závěr

V úvodu této práce bylo stanoveno několik cílů, které byly v jednotlivých kapitolách zpracovány. Čtenář byl seznámen se základními pojmy typografie a formátů ukládajících font. Dále byly popsány základní vlastnosti parametrických a Bézierových křivek, bylo představeno a porovnáno několik metod vykreslování textu a také knihoven, které některé z těchto metod využívají. Byla provedena teoretická analýza dvou vybraných metod a jejich implementace v podobě knihovny pro vykreslování textu. Výsledky implementace byly porovnány z hlediska výpočetní a prostorové složitosti a kvality vykreslování.

Knihovna se nyní nachází ve funkčním použitelném stavu, poskytuje základní funkcionalitu načítání fontu, vykreslení textového řetězce a je ji tedy možné použít v PGR-frameworku. Při implementaci jednotlivých metod bylo objeveno mnoho problémů, které byly popsány v předchozích kapitolách a některé z nich nebyly zcela vyřešeny. Je tedy stále možné, že knihovna obsahuje chyby a nedokonalosti, které budou odhaleny časem při jejím dalším vývoji a testování.

8.1 Budoucí práce a rozšíření

Ačkoliv požadavky na funkcionalitu knihovny byly v této práci naplněny, existuje mnoho dalších možností, kterými by se dala implementace vylepšit.

Především by bylo možné optimalizovat předzpracování u obou metod a vytvoření možnosti exportu předzpracovaných dat, aby bylo možné přímé načítání dat do GPU bez potřeby opakovaného předzpracování.

U metody Loop/Blinn je potřeba v budoucnu dále vyřešit problém s překrývajícími konturami tak, aby nemusely být odstraněny. Bylo by také možné provádět efektivnější triangulaci glyfu u této metody, vyzkoušení jiných knihoven pro Delaunyho triangulaci nebo provést vylepšení této metody některou z metod [10] nebo [9]. Tato metoda dovoluje také rozšíření mapování textu na zaoblené 3D objekty.

U metody Lengyel je potřeba vyřešit adaptivní dilataci bounding boxu, aby mohly být jeho okraje vykresleny korektně při záběru kamery z jakéhokoliv

úhlu. Také by bylo vhodné vyřešit adaptivní počet samplů pro lepší zobrazení textu malé velikosti. Tato metoda nabízí také několik rozšíření v podobě vykreslování vícebarevných glyfů (emoji) nebo vytvoření různých efektů písma (stín, záře).

Do knihovny by bylo možné přidat mnoho dalších funkcionalit pro rozvržení textu, například vykreslování odstavců, jejich zarovnání apod. Bylo by možné také přidat možnosti nastavení několika dalších vlastností písma jako podtržený nebo přeškrtnutý text. Nabízejí se také možnosti v podobě typografických rozšíření jako podpora ligatur, kapitálek, kombinování diakritických znamének subscriptů apod. Bylo by také užitečné, aby knihovna měla plnou podporu Unicode.

Bibliografie

[1] Karl Åkerblom Artem Amirkhanov. *CDT: Constrained Delaunay Triangulation*. Knihovna na Delaunyho triangulaci. 2020. URL: https://github.com/artem-ogre/CDT.

[2] Jim Blinn. *Jim Blinn's Corner: Notation, Notation, Notation*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2002. ISBN: 1558608605.

[3] Microsoft Corporation. *OpenType® specification*. Technická dokumentace. 2020. URL: https://docs.microsoft.com/cs-cz/typography/opentype/spec/.

[4] Chris Green. „Improved Alpha-Tested Magnification for Vector Textures and Special Effects“. In: *ACM SIGGRAPH 2007 Courses*. SIGGRAPH '07. San Diego, California: Association for Computing Machinery, 2007, s. 9–18. ISBN: 9781450318235. DOI: 10.1145/1281500.1281665. URL: https://doi.org/10.1145/1281500.1281665.

[5] Viktor Chlumský. „Shape Decomposition for Multi-channel Distance Fields“. Diplomová práce. ČVUT FIT, květ. 2015. URL: https://dspace.cvut.cz/bitstream/handle/10467/62770/F8-DP-2015-Chlumsky-Viktor-thesis.pdf.

[6] Apple Inc. *TrueType Reference Manual*. Technická dokumentace. 2020. URL: https://developer.apple.com/fonts/TrueType-Reference-Manual/.

[7] Jukka Jylänki. *A thousand ways to pack the bin – a practical approach to two-dimensional rectangle bin packing*. Article. 2010. URL: http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.695.2918&rep=rep1&type=pdf.

[8] Mark Kilgard. *A Simple OpenGL-based API for Texture Mapped Text*. Popis API. 1997. URL: http://sgifiles.irixnet.org/sgi/opengl/contrib/mjk/tips/TexFont/TexFont.html.

[9] Mark J. Kilgard a Jeff Bolz. „GPU-Accelerated Path Rendering“. In: *ACM Trans. Graph.* 31.6 (lis. 2012). ISSN: 0730-0301. DOI: 10.1145/2366145.2366191. URL: https://doi.org/10.1145/2366145.2366191.

[10] Yoshiyuki Kokojima et al. „Resolution Independent Rendering of Deformable Vector Objects Using Graphics Hardware“. In: *ACM SIGGRAPH 2006 Sketches.* SIGGRAPH ’06. Boston, Massachusetts: Association for Computing Machinery, 2006, 118–es. ISBN: 1595933646. DOI: 10.1145/1179849.1179997. URL: https://doi.org/10.1145/1179849.1179997.

[11] Eric Lengyel. „GPU-Centered Font Rendering Directly from Glyph Outlines“. In: *Journal of Computer Graphics Techniques (JCGT)* 6.2 (čvc 2017), s. 31–47. ISSN: 2331-7418. URL: http://jcgt.org/published/0006/02/02/.

[12] Eric Lengyel. *GPU-Centered Font Rendering Directly from Glyph Outlines.* Presentation. 2018. URL: http://www.terathon.com/i3d2018_lengyel.pdf.

[13] Terathon Software LLC. *Slug Font Rendering Library.* Knihovna pro vykreslování textu. 2020. URL: https://sluglibrary.com/.

[14] Charles Loop a Jim Blinn. „Resolution Independent Curve Rendering Using Programmable Graphics Hardware“. In: *ACM Trans. Graph.* 24.3 (čvc 2005), s. 1000–1009. ISSN: 0730-0301. DOI: 10.1145/1073204.1073303. URL: https://doi.org/10.1145/1073204.1073303.

[15] Hubert Nguyen. *GPU Gems 3.* Addison-Wesley Professional, 2007. ISBN: 9780321515261. URL: https://developer.nvidia.com/gpugems/gpugems3/part-iv-image-effects/chapter-25-rendering-vector-art-gpu.

[16] The FreeType project. *FreeType features.* Funkcionality knihovny. 2018. URL: https://www.freetype.org/freetype2/docs/index.html.

[17] The FreeType project. *Glyph Conventions.* Informace o glyfech. 2018. URL: https://www.freetype.org/freetype2/docs/glyphs/index.html.

[18] David Rosen. *High-quality text rendering.* Obrázek distance field. 2013. URL: http://blog.wolfire.com/2013/03/High-quality-text-rendering#comment-833333936.

[19] Nicolas P. Rougier a Behdad Esfahbod. „Digital Typography: 25 Years of Text Rendering in Computer Graphics“. In: *ACM SIGGRAPH 2018 Courses.* SIGGRAPH ’18. Vancouver, British Columbia, Canada: Association for Computing Machinery, 2018. ISBN: 9781450358095. DOI: 10.1145/3214834.3214837. URL: https://doi.org/10.1145/3214834.3214837.

[20] Joey de Vries. *Learn OpenGL - Shader class*. Třída pro shader program. 2020. URL: https://learnopengl.com/code_viewer_gh.php?code=includes/learnopengl/shader_s.h.

[21] Evan Wallace. *Font Texture Generator*. Nástroj na generování texture atlas. 2016. URL: https://github.com/evanw/font-texture-generator.

Seznam zkratek

2D dvoudimenzionální

3D třídimenzionální

OpenGL Open Graphics Library

GLSL OpenGL Shading Language

ASCII American Standard Code for Information

HUD Head-Up Display

UI User Interface

API Application Programming Interface

GPU Graphics Processing Unit

CW Clockwise

CCW Counterclockwise

VBO Vertex Buffer Object

EBO Element Buffer Object

RGB Red Green Blue

GLFW Graphics Library Framework

FPS Frames Per Second

MSAA Multisample Anti-aliasing

SSAA Supersampling Anti-aliasing

Příloha B

Obsah přiložených souborů

```
/
├── lib
├── release
├── appsource
├── videoLoopBlinn
├── videoLengyel
└── imgs
```

- **lib** Zdrojový kód implementované knihovny.
- **realease** Ukázková aplikace.
- **appsource** Zdrojový kód aplikace.
- **videoLoopBlinn** Video z používání ukázkové aplikace metody LoopBlinn.
- **videoLengyel** Video z používání ukázkové aplikace metody Lengyel.
- **imgs** Snímky výstupů implementovaných metod.