PŘÍRODOVĚDECKÁ FAKULTA UNIVERZITY PALACKÉHO
KATEDRA INFORMATIKY

BAKALÁŘSKÁ PRÁCE

Renderování vektorové grafiky v XNA Frameworku

2013 Petr Kalandra

Anotace

Cílem práce je vytvořit dynamickou knihovnu umožňující renderování vektorové grafiky v XNA frameworku pouze pomocí jeho funkcionality a bez použití knihoven třetích stran. Knihovna musí zvládat základní operace vektorové grafiky, např. kreslení jednoduchých objektů (včetně Bézierových křivek), transformace, barevné a bitmapové výplně, základní barevné přechody. Nezbytnou součástí práce je ukázková aplikace a srovnání s konkurenčními knihovnami.

Děkuji vedoucímu práce Mgr. Eduardu Bartlovi, Ph.D. a také své rodině za ochotu, trpělivost a podporu během tvorby práce.

Obsah

Seznam obrázků

Seznam zdrojových kódů

1. Úvod

1.1. Motivace

Počátečním impulzem k sepsání této bakalářské práce byl jeden z momentů během tvorby vlastního projektu, kdy jsem zjistil, že k dispozici prakticky není žádná kompletní implementace hardwarově akcelerovaného renderování vektorové grafiky v žádném aplikačním rozhraní pro 3D grafiku. V oblasti interaktivních grafických aplikací, zejména pak her, se díky bitmapově orientované technologii grafických karet stále primárně používají bitmapy a to i pro tvorbu uživatelských rozhraní. Přitom právě zde se přímo nabízí použití grafiky vektorové, která umožní zobrazit uživatelské rozhraní beze ztráty kvality na rozličných zobrazovacích zařízeních a při různých rozlišeních. Rozhodl jsem se proto prozkoumat možnosti vykreslování vektorové grafiky s co největším využitím potenciálu moderních grafických procesorů, prověřit situaci na poli knihoven vektorové grafiky a dospět k závěru, zda je použití vektorové grafiky v popsaných podmínkách výhodné.

1.2. Cíle

Po dohodě s vedoucím práce byla jako primární cíl stanovena tvorba dynamické knihovny umožňující renderování vektorové grafiky pouze s využitím grafického aplikačního rozhraní bez použití knihoven třetích stran ve vykreslovacím jádře knihovny. Díky tomu by mělo být snadné portovat knihovnu do jakéhokoli jiného trojrozměrného aplikačního rozhraní. Knihovna musí zvládat vykreslení základních primitiv vektorové grafiky, jimiž jsou bezpochyby Bézierovy křivky a to jak kvadratické, tak kubické. Dále je pak nezbytností podpora afinních grafických transformací, barevných a bitmapových výplní včetně základních barevných přechodů. Jako vlastní cíl jsem si stanovil implementaci jednoduchého vykreslování textu jakožto elementárního prvku uživatelských rozhraní. Za nezbytnou součást práce je považována ukázková aplikace, za kterou je považován jednoduchý přehrávač souborů ve formátu Flash. Posledním z cílů je přinést objektivní srovnání funkcionality knihoven pro vektorovou grafiku a jednoduchý benchmark.

2. Volba frameworku

2.1. Volba API

Jako první krok, před samotným zahájením práce, bylo nezbytné zvolit vhodné aplikační rozhraní pro práci s trojrozměrnou grafikou. Zde jsem mohl volit buď přímo mezi nízkoúrovňovými rozhraními OpenGL a DirectX nebo některý z frameworků. Každá z těchto variant má svá pro a svá proti. Nízkoúrovňová rozhraní jsem zavrhl prakticky ihned z počátku. Jejich volbou bych sice nemusel řešit mnohá omezení, avšak přinesla by nutnost ošetření spousty základních problémů a odváděla tak od hlavní myšlenky práce. Rozhodl jsem se proto pro některý z dostupných herních frameworků. Mezi potenciální kandidáty připadaly knihovny Irrlicht engine, OGRE a XNA. Prvotní volba padla na Irrlicht engine. Bohužel však záhy následovalo zjištění, že nemá žádné přímé rozhraní pro manipulaci se stencil bufferem a bylo tedy nutné funkcionalitu dodělat. Její začlenění však autoři zamítli a bude-li nakonec začleněna, nestane se tak v dohledné době. Udržovat vlastní vývojovou větev knihovny kvůli dvěma upraveným souborům by bylo nevhodné a stavět na „nedostupné“ funkcionalitě nemá žádný praktický význam. Z těchto důvodů jsem se rozhodl prověřit konkurenční OGRE. Kvůli jeho těžkopádnosti jsem jej zavrhl po velmi krátké době a to i přes fakt, že knihovna obsahuje veškeré potřebné funkce. Nakonec tedy volba padla na XNA Framework od společnosti Microsoft. Ačkoli sebou tato volba nesla několik problémů, které zmíním v dalších částech práce, dá se považovat za cestu nejmenšího odporu. Díky využití .NET frameworku a jazyka C#jsem byl během tvorby odstíněn od většiny nízkoúrovňových problémů jakými jsou například správa paměti, inicializace grafického rozhraní nebo správa grafických prostředků (textur, modelů, apod.). Mohl jsem se tudíž přímo soustředit na tvorbu prototypu knihovny.

2.2. Seznámení s XNA frameworkem

XNA framework je aplikační rozhraní a sada nástrojů z dílny společnosti Microsoft. Sestává z několika samostatných knihoven, které společně poskytují veškerou nezbytnou funkcionalitu pro tvorbu her na platformách Windows, Xbox 360 a Windows Phone včetně integrace do ekosystému Windows Live. První verze frameworku byla představena již v roce 2006 a nyní se nachází ve své poslední verzi 4.0. Další vývoj byl zastaven 31. ledna 2013. Technologicky vychází z .NET Compact frameworku verze 2.0. Jedná se tedy o tzv. *managed kód*, který ke svému provozu vyžaduje běhové prostředí .NET frameworku. Na platformě Windows je obecně možné využívat funkcionalitu kterékoli verze .NET, ale taková aplikace nebude spustitelná na jiných zařízeních. Díky jednotnému aplikačnímu rozhraní jsou hry velice snadno přenositelné mezi jednotlivými platformami. Není vyžadován praktický žádný zásah uživatele. Součástí XNA Game Studia jsou i nástroje pro snadnou tvorbu obsahu mezi které se řadí například Cross-platform

Audio Creation Tool (XACT). Další důležitou součástí frameworku je takzvaná *content pipeline*, která slouží k převodu a předzpracování rozličných datových formátů do jednoduchých a jednotných formátů, kterým XNA framework rozumí. Tím je odstraněna nutnost implementovat všechny tyto formáty na cílové platformě a také se tím redukuje čas nutný ke spuštění aplikace, protože nedochází k žádnému dalšímu zpracování vstupních dat.

Z pohledu grafiky se fakticky jedná o zapouzdření rozhraní DirectX 9. Na rozdíl od něj se však framework sám stará o správu prostředků a umožňuje tak vývojářům zaměřit se přímo na herní logiku. Aplikace mohou být tvořeny v jednom ze dvou dostupných profilů. Profil *Reach* umožňuje pracovat pouze s omezenou částí grafické knihovny tak, aby byla výsledný kód spustitelný na všech zmíněných platformách. Naproti tomu profil *HiDef* zpřístupňuje celou grafickou knihovnu a takovéto aplikace jsou spustitelné jen na platformách Xbox 360 a Windows. Zde je však nutno podotknout, že na platformě Windows je vyžadována novější grafická karta, která poskytuje veškerou funkcionalitu frameworku (obecně se jedná o karty s podporou Shader Modelu 3 a vyšším).

Existuje i několik neoficiálních implementací (např. Mono.XNA), díky kterým je možné spustit aplikace i na dalších platformách.

3. Zkratky a pojmy použité v práci

.NET Framework
Platforma pro vývoj aplikací od společnosti Microsoft.

Aliasing
Jev, ke kterému dochází při nedostatečném vzorkování během převodu spojitého signálu na diskrétní. V případě počítačové grafiky se projevuje především zubatými hranami objektů.

Anti-aliasing
Techniky, jejichž cílem je více či méně potlačit projevy aliasingu.

Cesta
Pojem z terminologie vektorové grafiky. Jedná se o sled úseček a křivek dohromady tvořící obrys vykreslovaného objektu.

Depth buffer
Buffer grafické karty obsahující informace o vzdálenosti objektu od obrazovky.

Flash
Technologie společnosti Adobe zaměřená na tvorbu interaktivních animací zejména pro web.

Fragment
Označení pixelu s metadaty v průběhu jeho zpracování uvnitř grafické pipeline.

Geometrie
Pojem z oblasti počítačové grafiky označující pro počítač srozumitelnou reprezentaci grafického objektu.

GPU (*graphics processing unit*)
Hlavní výpočetní jednotka grafické karty.

Grafická pipeline
Posloupnost operací prováděných nad trojrozměrnou reprezentací scény za účelem jejího převodu na dvourozměrnou rastrovou reprezentaci.

HLSL (*High Level Shading Language*)
Programovací jazyk pro programování shader jednotek grafické karty vyvinutý společností Microsoft.

Index buffer
Jeden z bufferů používaných při vykreslování trojrozměrného modelu obsahující indexy vykreslovaných vrcholů definovaných ve vertex bufferu.

Pixel
: Označení reprezentace jednoho bodu rastrové grafiky.

Plátno
: Pojem z terminologie vektorové grafiky. Jedná se o rovinu, do které probíhá kreslení. Jedná se o analogii reálného malířského plátna.

Stencil buffer
: Buffer grafické karty sloužící primárně k maskování obrazu na úrovni jednotlivých pixelů.

SWF
: Formát, do něhož jsou ukládány klipy vytvořené s využitím technologie Flash.

Teselace
: Proces, během kterého je rovinný útvar rozložen na více jednodušších útvarů.

Textura
: Označení rastrového obrázku v terminologii trojrozměrné grafiky.

Texel
: Označení pixelu textury.

Vertex buffer
: Buffer obsahující vrcholy trojrozměrného modelu.

Vertex
: Označení jednoho vrcholu trojrozměrného modelu s případnými metadaty.

4. Systémové požadavky

Pro správný chod knihovny aplikace jsou vyžadováno splnění následujících bodů:

- Microsoft Windows Vista (Service Pack 2) nebo vyšší,
- Microsoft XNA Framework Redistributable 4.0,
- grafická karta s podporou profilu HiDef,
- doporučený procesor Intel nebo AMD 2 GHz nebo vyšší,
- doporučeno 2 GB RAM nebo více,

Knihovna byla navržena tak, aby ji bylo možné využít i na platformě Xbox 360, ale tato varianta nebyla nikdy testována a je možné, že v současném stavu nebude pracovat zcela korektně. S minimální námahou by mělo být možné vytvořit verzi pro profil Reach.

5. Použité techniky

5.1. Hardwarové technologie

5.1.1. Stencil buffer

Stencil buffer je jeden z bufferů, které se nacházejí v moderním grafickém hardwaru. V tomto bufferu jsou ukládány celočíselné hodnoty pro každý pixel obrazu a je možné nad ním provádět bitové a aritmetické operace pomocí standardního vykreslování grafických primitiv. Uložené hodnoty je pak možné využít během fáze renderovací pipeline nazývané stencil test. Pixely, které tímto testem projdou jsou vykresleny, ostatní jsou zahozeny a nebudou tak do výsledného obrazu zahrnuty.

Ke konfiguraci chování stencil bufferu slouží v XNA frameworku datová struktura *DepthStencilState*. Jak název napovídá, je s její pomocí možné ovlivnit i chování tzv. depth bufferu, který ovšem není v knihovně nijak využit a nebudu se o něm proto dále rozepisovat. Pro představu uvádím přehled nastavitelných hodnot této struktury souvisejících se stencil bufferem, který může sloužit k demonstraci všech jeho schopností:

StencilEnable
: Povoluje nebo zakazuje stencil test.

StencilFail
: Operace provedená, selže-li stencil test.

StencilDepthBufferFail
: Operace provedená, jestliže byl stencil test úspěšný, ale selhal tzv. depth test.

StencilPass
: Operace provedená v případě obou úspěšných testů.

StencilFunction
: Funkce porovnávající maskovanou hodnotou bufferu s maskovanou referenční hodnotou během stencil testu.

StencilMask
: Bitová maska aplikovaná na referenční hodnotu a hodnotu bufferu během testování.

StencilWriteMask
: Maska označující bity modifikované během zápisu do bufferu.

ReferenceStencil
: Referenční hodnota využívaná během testu k porovnávání.

TwoSidedStencilMode
: Povoluje nastavení rozdílných operace pro opačně orientovaná grafická primitiva. Ty jsou pak pro primitiva orientovaná proti směru hodinových ručiček definovány pomocí hodnot s prefixem CounterClockwise.

K dispozici jsou následující funkce srovnávající referenční hodnotu a hodnotu v bufferu během testování:

Always
: Test je vždy úspěšný.

Never
: Test vždy selže.

Equal
: Test je úspěšný, je-li referenční hodnota rovna hodnotě bufferu.

NotEqual
: Test je úspěšný, je-li referenční hodnota různá od hodnoty bufferu.

Greater
: Test je úspěšný, je-li referenční hodnota větší než hodnota bufferu.

GreaterEqual
: Test je úspěšný, je-li referenční hodnota větší nebo rovna hodnotě bufferu.

Less
: Test je úspěšný, je-li referenční hodnota menší než hodnota bufferu.

LessEqual
: Test je úspěšný, je-li referenční hodnota menší nebo rovna hodnotě bufferu.

Jako operaci provedenou nad hodnotou v bufferu v reakci na výsledek testování lze zvolit některou z těchto možností:

Decrement
: Sníží hodnotu v bufferu o jedničku. Byla-li v bufferu nula, dojde k zapsání maximální hodnoty.

DecrementSaturation
: Sníží hodnotu v bufferu o jedničku. Nulová hodnota zůstane nezměněna.

Increment
: Zvýší hodnotu v bufferu o jedničku. Byla-li v bufferu maximální hodnota, dojde k zapsání nuly.

IncrementSaturation
Zvýší hodnotu v bufferu o jedničku. Maximální hodnota zůstane nezměněna.

Invert
Provede bitovou negaci hodnoty v bufferu.

Keep
Zachová stávající hodnotu v bufferu.

Replace
Nahradí hodnotu v bufferu referenční hodnotou.

Zero
Nahradí hodnotu v bufferu nulovou hodnotou.

5.1.2. Programovatelná pipeline

Grafické procesory mají již celé desetiletí možnost změnit do jisté míry výchozí způsob zpracování grafických dat pomocí programovatelné pipeline. K tomu slouží takzvané shadery. Jedná se o malé programy běžící přímo na grafickém čipu. Program se dělí na několik částí podle toho, jaká data zpracovává. V knihovně využívám Shader Modely verze 2 a 3, které se skládají pouze z následujících dvou částí:

Vertex shader
Jedná se o první spuštěnou část shaderu, vykonávanou jednou pro každý vrchol renderované geometrie. S jeho pomocí lze provádět například grafické transformace či dynamické generování modelu. Jeho možnosti jsou však velmi omezené neboť neumožňuje měnit počet vrcholů geometrie. Výstupem je tedy vždy jen modifikovaný vrchol.

Pixel shader
Druhá spuštěná část shaderu, která se aplikuje na každý pixel výsledného obrazu. V této fázi již není známa původní struktura geometrie. Jediným vstupem jsou (zpravidla interpolované) výstupní hodnoty vertex shaderu v daném pixelu. Výstupem je barevná hodnota pixelu, případně jeho hloubka.

Existuje několik jazyků pro programování shaderů, ale v rámci prostředí XNA frameworku je možné použít pouze jazyk HLSL (High level shading language) určený pro rozhraní DirectX společnosti Microsoft. Proto veškeré úryvky zdrojového kódu shaderu budou uváděny právě v tomto jazyce.

5.2. Bézierovy křivky

Bézierovy křivky jsou bezesporu základem vektorové grafiky. Pro svoji jednoduchost a některé vlastnosti staly hlavním využívaným druhem křivek v této oblasti a jsou obsaženy v drtivé většině vektorových formátů. Jedná se o parametrické křivky definované pomocí řídících bodů. Jméno dostaly podle svého francouzského autora, inženýra, Pierra Béziera. Matematickým základem Bézierových křivek jsou Bernsteinovy polynomy. Podstatnou vlastností je, že se celá křivka vždy nachází uvnitř konvexního obalu svého řídícího mnohoúhelníka definovaného řídícími body. Křivka vždy prochází svými koncovými body, což je další zásadní fakt, který je činí vhodné pro použití ve vektorové grafice. Díky vlastnostem Bernsteinových polynomů je také možné křivky rekurentně dělit s využitím Casteljauova algoritmu. Jedná se vlastně o dělení úseček řídícího polygonu křivky v požadovaném poměru. Obecně je možné vytvořit křivku libovolného řádu, avšak běžně se používají pouze křivky prvního (úsečky), druhého (kvadratické křivky) a třetího řádu (kubické křivky).

Vykreslovací jádro knihovny pracuje, kromě úseček, pouze s křivkami kvadratickými, které jsou dány následujícím parametrickým předpisem:

$$\mathbf{B}(t) = (1-t)^2\mathbf{P}_0 + 2(1-t)t\mathbf{P}_1 + t^2\mathbf{P}_2, \text{ kde}$$

$t \in [0,1]$ je parametr, $\mathbf{P}_0$ počáteční bod, $\mathbf{P}_1$ řídící bod a $\mathbf{P}_2$ koncový bod křivky.

5.3. Technika Stencil, then cover

Vykreslovací jádro knihovny je postaveno na technice *stencil, then cover*. Ačkoli se jedná o dlouhodobě využívanou techniku, ve vztahu k vektorové grafice byla pojmenována a popsána společností nVidia [4]. Hlavní myšlenkou je rozdělení vykreslovacího procesu do dvou fází a to fází *stencil* a *cover*.

Během fáze stencil dochází k definování pixelů plochy pokryté obrazcem. K uchování této informace slouží stencil buffer, kde nenulová hodnota značí pokrytý pixel. Zápis barevných dat je během celé fáze vypnut, proto nedochází k přepsání současného obrazu a zároveň je tím dosaženo vyššího výkonu (zapisuje se méně dat).

Ve fázi cover jsou pak pixely označené ve stencil bufferu pokryty výplní a zapsány do výsledného obrazu. Příslušná oblast stencil bufferu je během vykreslení vynulována a tím se připraví k renderování dalšího obrazce. K pokrytí dochází vykreslením geometrie, která zahrnuje všechny označené pixely. Tuto podmínku splňuje například jakýkoli konvexní obal všech vrcholů původního obrazce. Zde je nutné zvolit vhodný kompromis mezi složitostí krycí geometrie a množstvím nadbytečných pokrytých pixelů. Dva krajní případy jsou obdélník a původní obrazec. Obdélník obsahující všechny vrcholy obrazce je jedním z nejtriviálnějších konvexních obalů. Výhodou je snadná manipulace a generování takového obdélníka, nevýhodou pak vysoké množství nadbytečných pixelů ve stencil

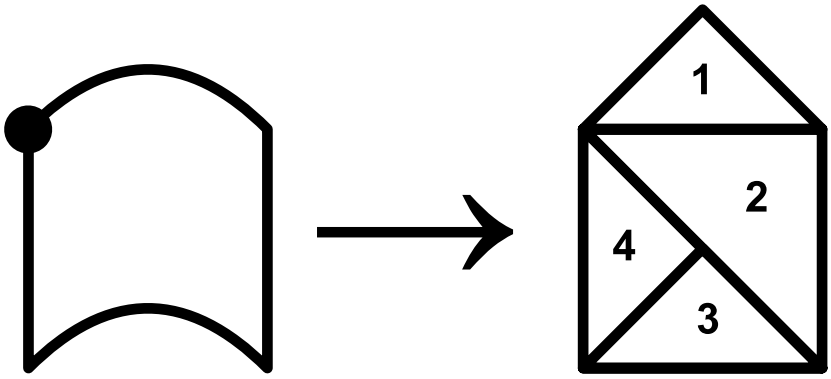
testu. Opačným případem je pak opakované vykreslení geometrie původního obrazce. Při použití této varianty je testováno minimální množství pixelů, avšak grafický procesor je zatížen větším množstvím vstupních dat geometrie. V knihovně je použit krycí obdélník především z důvodu snadné manipulace a tudíž nízkých nároků na CPU.

5.4. Generování vyplněných ploch

Vyplňování oblastí ohraničených uzavřenou cestou je častějším a jednodušším případem ve vykreslování vektorové grafiky. Při použití techniky *Stencil, then cover* je celá procedura provedena během fáze *stencil*. Nejprve je nutné vykreslovanou cestu převést na trojúhelníkovou reprezentaci srozumitelnou pro grafický procesor (tzv. teselace). Poté musí být na vrcholy aplikována uživatelem určená afinní transformace a projekce, které zajišťují umístění cesty v rámci výsledného obrazu. Poslední fází je označení vnitřních pixelů plochy.

5.4.1. Teselace

Aby bylo možné cestu vyplnit, je nutné vytvořit reprezentaci, které bude grafická karta rozumět. Aby byla zachována jednoduchost celého procesu a nebylo nutné využívat více shaderů, je cesta již během svého vzniku převáděna na kvadratické Bézierovy křivky případně úsečky. Kubické Bézierovy křivky jsou aproximovány pomocí čtyř kvadratických segmentů. Během teselace jsou pak veškeré segmenty postupně procházeny od začátku do konce a pro každý z nich jsou do výsledné geometrie přidávány příslušné trojúhelníky. V případě úsečky se jedná o trojúhelník tvořený krajními body úsečky a počátečním bodem aktuálně vyplňované cesty. Kvadratická Bézierova křivka pak přidává ještě jeden trojúhelník určený všemi řídícími body křivky. Situaci nejlépe vystihne obrázek 1. Kolečko značí počáteční bod cesty.



Obrázek 1. Teselace výplně jednoduché cesty

5.4.2. Transformace

Trojúhelníková reprezentace vytvořená v předchozím kroku je poté předána k vykreslení grafické kartě. Zde nejprve prochází zpracováním pomocí vertex shaderu, který na každý z vrcholů aplikuje požadovanou transformaci. Ta sestává z dvojího maticového násobení. Nejprve je k souřadnicím vrcholu přičten posun v souřadnicovém systému cesty. To je vhodné pro rychlé vykreslení posunutých objektů, jako jsou například znaky textu. Tyto souřadnice jsou poté násobeny transformační maticí, která je převádí do souřadnicového systému plátna. Ty jsou pak dále násobeny projekční maticí, která zajistí převod souřadnic do tzv. homogenního souřadnicového systému. V případě DirectX je pak oblast obrazovky v tomto systému definována (uvažujeme-li pouze dvourozměrné souřadnice) jako obdélník s levým horním rohem v bodě $[-1; 1]$ a pravým dolním rohem v bodě $[1; -1]$. Posledním krokem je přičtení posunu v homogenním souřadnicovém systému, který slouží primárně pro vyhlazování. Vertex shader také předává k interpolaci řídící texturové souřadnice pro rasterizaci křivky.

```
1  VSOut TransformVertex(VSIn i)
2  {
3    VSOut o;
4
5    o.Texcoord = i.Position.zw;
6    o.Position.xyz = float3(i.Position.xy + Offset.xy, 1);
7    o.Position.xyz = mul(o.Position.xyz, Transformation);
8    o.Position.xyz = mul(o.Position.xyz, Projection);
9    o.Position.xy += Offset.zw;
10   o.Position.w = 1;
11
12   return o;
13 }
```

Zdrojový kód 1 Vertex shader pro renderování vyplněných ploch

5.4.3. Rasterizace

Následující fází generování vyplněné plochy je rasterizace. Jedná se o proces, kdy je transformovaná trojúhelníková reprezentace z předchozího kroku převedena na množinu pixelů výsledného obrazu prokrytých touto plochou. O to se stará hardwarový rasterizér na grafické kartě, který označí všechny pixely pokryté alespoň jedním trojúhelníkem. Tím však není dosaženo vykreslení zakřivených okrajů. K tomuto účelu je využit pixel shader, který pro každý z pixelů ověří, zda se nachází uvnitř obrazce a na základě tohoto testu vnější pixely vylučuje.

Princip činnosti shaderu je jednoduchý a vychází z techniky popsané v publikaci [2]. Jednou ze vstupních hodnot jsou správně perspektivně interpolované řídící texturové souřadnice pro procedurální generování křivky. Ty jsou definovány jako $[0;0]$ pro počáteční bod, $[\frac{1}{2};\frac{1}{2}]$ pro řídící bod a $[1;0]$ pro koncový bod křivky. Po dosazení do vzorce kvadratické Bézierovy křivky dostáváme křivku odpovídající parametrickému předpisu $x(t) = t, y(t) = t - t^2$. Uvažujeme-li předpis rovinné křivky $f(x, y) = 0$, pak všechny body splňující rovnici $(1 - x)x - y = 0$ leží na křivce. Můžeme proto napsat shader, který řeší nerovnici $(1-x)x-y \geq 0$. Tato nerovnice není splněna pro vnější pixely a ty mohou být zahozeny pomocí intristické funkce jazyka HLSL *clip*. Ta pixel vylučuje z renderování, má-li její argument zápornou hodnotu čímž se k využití přímo nabízí.

```
1 float4 StencilCurve(VSOut i) : COLOR0
2 {
3   clip(i.Texcoord.x * (1 - i.Texcoord.x) - i.Texcoord.y);
4   return float4(1, 1, 1, 1);
5 }
```

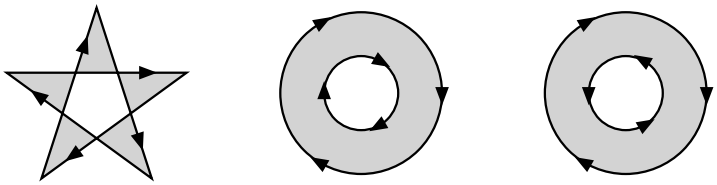
Zdrojový kód 2 Pixel shader pro renderování vyplněných ploch

5.4.4. Pravidla Even-Odd a NonZero

Pixely získané během rasterizace jsou nyní připraveny k zapsání do stencil bufferu. Před tím je však nutno ošetřit ještě jeden případ a tím jsou vyplněné plochy s výřezem. Zde je nejprve nutné pozastavit se nad definicí takového výřezu. V praxi se používají dvě takové definice (též pravidla) odborně označované jako *Fill rules* nebo *Winding rules*:

Pravidlo Even-Odd

Pro každý pixel obrazu se určí parita jeho výskytů. Lichý počet výskytů značí, že bod je vnitřním bodem obrazce Sudý počet výskytů pak označuje bod vnější.

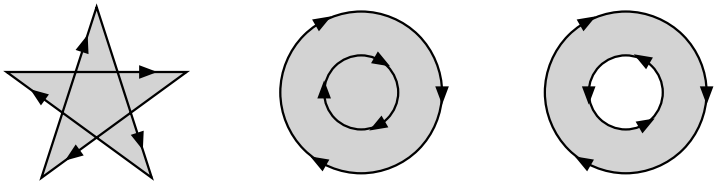


Obrázek 2. Demonstrace pravidla Even-Odd

Pravidlo NonZero

U tohoto pravidla záleží na směru jednotlivých segmentů cesty. Pro každý

pixel obrazce se uchovává celočíselná hodnota (na počátku nulová), která je pixely výplně segmentu definovaného v jednom směru inkrementována a ve směru opačném dekrementována. Všechny pixely s nenulovou hodnotou jsou považovány za vnitřní pixely obrazce.



Obrázek 3. Demonstrace pravidla NonZero

Obě tyto pravidla lze velice snadno implementovat právě s využitím stencil bufferu a proto se celý problém redukuje na jeho správné nastavení [3]. V případě pravidla Even-Odd stačí provádět bitovou negaci hodnoty ve stencil bufferu. U pravidla NonZero je třeba uvažovat i orientaci. Pro pixely generované trojúhelníkem definovaným ve směru hodinových ručiček se hodnota ve stencil bufferu nastaví na jedničku, v opačném směru na nulu. Tím je dosaženo korektního vygenerování masky obrazce ve stencil bufferu připravené k vyplnění během fáze cover.

5.5. Generování obrysů

Renderování obrysů cesty je druhým a složitějším případem. Při použití techniky *Stencil, then cover* je celá procedura stejně jako u vyplňování provedena během fáze *stencil*. Komplikace při renderování obrysu cesty dané tloušťky spočívá v hledání tzv. offsetové křivky, což je křivka taková, že vzdálenost všech jejich bodů od příslušného bodu původní křivky je konstantní. Matematicky lze tento problém zapsat jako $\|\mathbf{B}(t) - \mathbf{B}_o(t)\| = konst.$, tedy jako eukleidovskou normu vektoru mezi příslušným bodem původní ($\mathbf{B}$) a posunuté ($\mathbf{B}_o$) křivky. Algebraické řešení tohoto problému pro Bézierovy křivky však není známo. Je tedy nutné sáhnout po více či méně přesné aproximaci. Během tvorby knihovny jsem vyzkoušel několik hardwarově akcelerovaných metod, ale všechny se ukázaly jako neefektivní či nedokonalé nebo závisely na využití rozhraní DirectX 10, což není v rámci XNA Frameworku možné. Nakonec jsem tedy zvolil tradiční postup a to aproximaci lomenou čárou, která vznikne spojením krajních bodů dostatečně malých úseků vypočtených rekurentním dělením původní křivky. Jestliže není grafický výsledek z nějakého důvodu pro designera uspokojivý, může využít možnosti dnešních grafických editorů a převést problém vykreslování obrysů na vyplňování uzavřené plochy.

5.5.1. Teselace

Stejně jako v případě vyplněných ploch je nejprve potřeba převést cestu na trojúhelníkovou reprezentaci. Za tímto účelem je nejprve křivka aproximována lomenou čarou. K tomu jsem využil rekurentního algoritmu De Casteljau [6]. Ke každému uzlu takto vzniklé lomené čáry jsou vygenerovány body posunuté po a proti směru jednotkového tečného vektoru v tomto uzlu. Oba tyto body jsou zahrnuty do výsledné trojúhelníkové reprezentace, ale jsou uloženy jako čtveřice $[X; Y; X_o; Y_o]$, kde X a Y jsou souřadnice původního uzlu a X_o a Y_o jsou souřadnice vektoru posunutí. Tato reprezentace umožňuje měnit tloušťku obrysu za chodu bez nutnosti znovu provádět teselaci. V tomto kroku jsou také přidány spoje a zakončení segmentů cesty. Tuto problematiku však nebudu na tomto místě více rozebírat, protože je snadno dohledatelná v jakékoli literatuře zabývající se vektorovou grafikou.

5.5.2. Transformace

Transformace je principiálně shodná s transformací v případě vyplňování uzavřených ploch, avšak je zde několik drobných rozdílů. Vzhledem k reprezentaci vrcholů (bod a směr tečny), je nejprve nutné určit výslednou pozici vrcholu. Toho se dosáhne vynásobením směrového vektoru polovinou tloušťky obrysu a následným přičtením k souřadnicím bodu. Teprve v tomto okamžiku je možné provést samotnou transformaci.

Speciálním případem je pak vykreslování neškálovatelných obrysů. Ty si zachovávají stále stejnou tloušťku bez ohledu na aplikovanou transformaci. Toho je dosaženo záměnou pořadí operací. Nejprve je provedena transformace a následně přičítání vektoru. Aby měl tento vektor správný směr, je nutné jej také transformovat a poté normalizovat na jednotkovou délku. V případě neuniformních transformací to ovšem není dostatečné, protože dojde k znehodnocení informace o směru (horizontální a vertikální souřadnice jsou škálovány v opačném poměru). Tento problém je vyřešen převedením tečného vektoru na normálový, následnou transformací a převodem zpět na tečný. Převod mezi tečným a normálovým vektorem spočívá v záměně horizontální a vertikální složky a změnou znaménka jedné z nich.

5.5.3. Rasterizace

Jelikož jsou obrysy sestaveny pouze z cest neobsahujících křivkové segmenty, není potřeba provádět žádné úkony navíc a celý proces je tak pouze v režii standardní grafické pipeline. Pixel shader neobsahuje žádné další výpočty.

5.5.4. Zaoblené spoje a zakončení

Je-li spojování segmentů nebo zakončení cest nastaveno na zaoblenou va-

```
float4 TransformVertex_Scaling(float4 Position : POSITION0)
  : POSITION0
{
  float3 position = float3(Position.xy, 1);
  float3 offsetDir = float3(Position.zw, 0);

  position = position + Thickness * offsetDir;
  position = mul(position, Transformation);

  return float4(mul(position, Projection), 1.0) +
       float4(Offset, 0, 0);
}
```

Zdrojový kód 3 Vertex shader pro transformaci škálovatelných obrysů

```
float4 TransformVertex_Nonscaling(float4 Position : POSITION0)
 : POSITION0
{
  float3 position = float3(Position.xy, 1);
  float3 offsetDir = float3(Position.wz, 0);

  offsetDir *= float3(-1, 1, 0);
  offsetDir = normalize(mul(offsetDir, Transformation));
  position = mul(position, Transformation);
  offsetDir = offsetDir.yxz * float3(1, -1, 0);
  position = position + Thickness * offsetDir;

  return float4(mul(position, Projection), 1.0) +
       float4(Offset, 0, 0);
}
```

Zdrojový kód 4 Vertex shader pro transformaci neškálovatelných obrysů

riantu, je do vykreslování přidán druhý průchod. Ten opět sestává z teselace, transformace a rasterizace. Teselace a transformace zůstává stejná. Vygenerovaná geometrie sestává z čtverců, které mají středy v bodech spojů nebo zakončení. Rasterizace využívá speciální pixel shader, který podle vzdálenosti od středu aktuálního čtverce zahazuje pixely, které se nachází mimo kružnici s průměrem shodným s tloušťkou obrysu.

```
1 float4 StencilRadial(VSOut i) : COLOR0
2 {
3   clip(1.0 - length(i.Texcoord.xy));
4   return float4(1, 1, 1, 1);
5 }
```

Zdrojový kód 5 Pixel shader pro renderování zaoblení

5.6. Vybarvení pokrytých oblastí

5.6.1. Bitmapové výplně

Ne vždy si během vybarvování postačíme s jednolitou barevnou plochou, proto jsou jednou z implementovaných alternativ bitmapové výplně. Aby bylo možné plochu správně vyplnit, je třeba specifikovat bitmapu, kterou budeme plochu vyplňovat a transformační matici, která stanoví, jak bude bitmapa na plátno přenesena. Transformace výplně má z matematického hlediska za cíl převádět souřadnice v rovině plátna na souřadnice v rovině výplně. Ty jsou pro snadnou manipulaci stanoveny tak, aby levému hornímu rohu bitmapy odpovídal bod $[-1; -1]$ a pravému dolnímu rohu bod $[1; 1]$. O samotnou transformaci a interpolaci se poté postará grafická karta. Ta také umožňuje přímo zvolit, jak se má výplň opakovat pro souřadnice mimo výše zmíněný čtverec.

5.6.2. Barevné přechody

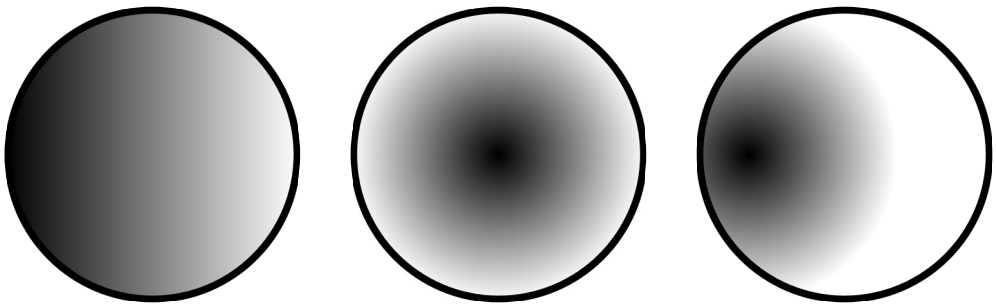
Další populární výplní jsou barevné přechody. Barevný přechod je zadaný pomocí několika takzvaných zarážek. Každá zarážka má přiřazenou barvu a pozici v rámci přechodu (v knihovně celočíselná hodnota v intervalu od 0 do 255). Z těchto zarážek je poté lineární interpolací barevných hodnot vygenerována bitmapa o rozměrech 256x1 px. Z pohledu grafické karty je pak situace obdobná jako u bitmapových výplní. Liší se pouze v tom, jak je volen příslušný bod z bitmapy. Tato volba záleží na požadovaném typu přechodu. V knihovně jsou k dispozici následující typy barevných přechodů:

Lineární
: Barvy plynule přechází od levého okraje (horizontální souřadnice rovna -1, zarážka na pozici 0) k pravému okraji (horizontální souřadnice rovna 1, zarážka na pozici 255). Na obrázku 4. se nachází vlevo.

Radiální
: Barvy plynule přechází od počátku systému souřadnic výplně (zarážka na pozici 0) směrem k vnějším okrajům výplně a tvoří soustředné kružnice. Bodu vzdálenému od středu jednu jednotku přísluší zarážka na pozici 255. Na obrázku 4. se nachází uprostřed.

Radiální s fokálním bodem

Varianta radiální výplně specifikovaná v rámci technologie Flash. Ta je rozšířena o definici souřadnic fokálního bodu, což je vlastně optický střed výplně. Ten je zadán pomocí vektoru, který určuje posunutí počátku souřadného systému výplně, neboť u standardní radiální výplně odpovídá fokální bod právě počátku souřadnicového systému. Tím je možné změnit umístění optického středu přechodu bez nutnosti přepočítávat transformační matici. Na obrázku 4. se nachází vpravo a má fokální bod umístěný blíže k levému okraji.



Obrázek 4. Ukázky barevných přechodů

5.6.3. Barevné transformace

Barevné transformace jsou ve vektorové i bitmapové grafice jednou z nezbytných operací. Díky nim je možné vytvářet různé barevné varianty obrazce aniž by bylo nutné mít každou z nich samostatně uloženou. Jelikož je barva z matematického hlediska reprezentována jako bod ve čtyřrozměrném prostoru, je možné (po doplnění bodu na pětirozměrný přidáním páté souřadnice s hodnotou jedna) takovou barevnou transformaci zapsat pomocí násobení maticí rozměru 5x5. V drtivé většině případů je však využita pouze multiplikativní a aditivní část transformace. V tomto případě je maticová reprezentace paměťově i výpočetně neefektivní a upřednostňuje se proto reprezentace pomocí dvou vektorů $\vec{t_m}$ (multiplikativní složka) a $\vec{t_a}$ (aditivní složka). Transformace pak odpovídá zápisu $C' = max(0, min(1, C \circ \vec{t_m} + \vec{t_a}))$, kde C je původní bod barevného prostoru, C' je bod barevného prostoru po provedení transformace a operace $\circ$ značí násobení bodu a vektoru po složkách.

V rámci knihovny jsou barevné transformace zajištěny v předposledním kroku fáze cover a jejich výpočet je kompletně prováděn na grafické kartě pomocí pixel shaderu.

```
1 float4 CxForm(float4 color)
2 {
3   return clamp(color * MulTerm + AddTerm, 0, 1);
4 }
```

Zdrojový kód 6 Funkce zajišťující barevné transformace

5.6.4. Stencil masky

Stencil maska se používá v případě, kdy chceme do výsledného obrazu zahrnout pouze specifické pixely. Jedná se o masku jednobitovou, pixel tedy do výsledného obrazu buď zahrnut je nebo není. K implementaci této techniky není zapotřebí žádného většího množství speciálního kódu, neboť přímo k tomuto účelu slouží stencil buffer. XNA podporuje pouze osmibitovou hloubku stencil bufferu, ale pro samotné vykreslování je postačující jeden bit. To umožňuje využít zbývajících sedm bitů jako sedm různých stencil masek. Vygenerování masky probíhá stejně jako samotné vykreslování, pouze je zvolena jiná bitová maska pro zápis do stencil bufferu a je vynechána fáze cover. Stejně tak aplikace masky na výsledný obraz znamená jen změnit nastavení bitové masky pro čtení ze stencil bufferu. Pro porovnání je zvolena rovnost s referenční hodnotou 255, čímž je dosaženo možnosti kombinovat více stencil masek pomocí konjunkce.

5.6.5. Alfa masky

Použití alfa masky se nabízí v případech, kdy je žádoucí specifikovat pro každý bod množství barvy, které má být do výsledného obrazu zahrnuté. Takováto maska je definovaná pomocí bitmapy s alfa kanálem (hodnota průhlednosti). Zvolená maska je během vykreslování na grafické kartě namapována přes celou oblast plátna a všechny složky barvy každého zapisovaného pixelu jsou vynásobeny hodnotou složky alfa v příslušném bodě masky. Knihovna umožňuje k maskování zvolit libovolnou složku barevné hodnoty bodu masky, což umožňuje úsporu místa v paměti na grafických kartách, které nepodporují jednosložkový formát textury, tím, že lze uložit do každého barevného kanálu jednu masku (tj. až čtyři masky v jedné bitmapě). Tato operace je posledním krokem uvnitř pixel shaderu pro fázi cover.

5.7. Vyhlazování

Vyhlazování (nebo též anti-aliasing) je jeden ze zásadních problémů při vykreslování jakékoli grafiky. Na tomto poli stále probíhá řada výzkumů a existuje mnoho metod s více či méně úspěšnými vizuálními výsledky. Jednou ze základních metod, navíc v současnosti přímo zabudovaných v grafickém čipu, je multisampling (multisample anti-aliasing, MSAA). Jedná se o speciální variantu nadvzor-

kování (tzv. supersamplingu), kde je každý výsledný pixel rozdělen na několik subpixelů, které jsou vyhodnoceny samostatně a výsledek je zprůměrován. Multisampling je pak optimalizací, kde jsou takto vyhodnocovány pouze pixely poblíž hran a šetří se tak výpočetní výkon.

V rámci knihovny je k vyhlazování využit multisampling, avšak není použita jeho hardwarová implementace přímo. Příčinou je způsob, jakým multisampling zajišťuje pipeline grafického rozhraní DirectX. Ta v případě zahození pixelu (funkcí **clip** nebo instrukcí **discard**) nevyhodnocuje jednotlivé subpixely a jsou tedy vyhlazeny pouze přímé hrany. XNA framework ovšem umožňuje manuálně stanovit bitovou maskou, do kterých vzorků se má zapisovat. Toho je využito právě při renderování zaoblených tvarů. Uvnitř knihovny jsou zahrnuty vzorkovací šablony, které definují, o kolik se má obraz renderovat do aktuálního vzorku posunutý (jedná se řádově o desetiny pixelu). Útvar je poté opakovaně renderován pro všechny posuny v šabloně a zapisován se správnou bitovou maskou. Tím je nasimulováno standardní chování multisamplingu. Geometrie je do grafické karty poslána pouze jednou a poté jsou pouze zasílány pokyny k vykreslení. Nedochází tedy ke zbytečným přesunům dat mezi pamětí RAM a VRAM. Otázkou zůstává, zda se opravdu jedná o multisampling nebo supersampling, protože veškeré optimalizace závisí na kódu ovladačů grafických karet, které jsou zpravidla uzavřené. S jistou mírou optimalizace se dá ovšem počítat, jelikož jsem během testů nepostřehl žádnou přímou závislost mezi rychlostí vykreslování a nastavenou mírou vyhlazování. Zajímavostí také je, že v případě použití OpenGL není nutné tento problém řešit. Pipeline tohoto rozhraní totiž pro zahozené pixely vyhodnocuje pixel shader samostatně pro každý subpixel a hardwarový multisampling díky tomu funguje korektně.

5.8. Renderování textu

Jelikož je vykreslování dynamického textu nezbytnou součástí každého uživatelského rozhraní, implementoval jsem za tímto účelem do knihovny jednoduchý systém.

Nejprve je nutné vytvořit font ve formátu, kterému knihovna rozumí. To je možné udělat buď za běhu tak, že se jednotlivým znakům přiřadí příslušná cesta představující vektorovou reprezentaci daného znaku. Druhou alternativou je využít jednoduché rozšíření *content pipeline*, který font importuje z formátu *OpenType* a přichystá pro rychlé a snadné použití. Nevýhodou je, že importér v současné době nepodporuje načítání informací pro kerning. Takto vytvořený font je možné použít k vykreslování textu.

K samotnému vykreslování je možné zvolit jednu ze dvou implementovaných technik. Jednodušší variantou je vykreslování pomocí procesoru počítače, ve které je pro každý znak spočítán příslušný offset. Ten je poté s teselovanou geometrií znaku poslán k vykreslení grafické kartě. Použití této techniky je vhodnější především pro krátké texty, protože je nutné postupně přepínat mezi vykreslova-

nou geometrií a pro každý znak je nutná další komunikace s grafickou kartou.

Druhou alternativou je použití techniky vykreslování za pomocí grafické karty. Tuto techniku jsem navrhl a implementoval jako náhradu standardního hardwarového instancování geometrie, která nebyla pro vykreslování textu použitelná. Font je v případě této techniky není reprezentován jako množina vrcholů geometrie, ale jako bitmapa obsahující jednotlivé vrcholy. Text je reprezentován pomocí bitmapy obsahující offsety znaků a množina indexů do obou těchto bitmap. Grafická karta poté v rámci vertex shaderu stanoví správnou polohu vrcholu nahlédnutím do obou bitmap. Tato technika je vhodná především pro rozsáhlé bloky textu, protože ke komunikaci s grafickým čipem dojde jen na začátku vykreslování. Nevýhodou pak mohou být jistá omezení vyplývající z použitých prostředků. Tím je omezení na fonty s maximálně čtyřmi miliony vrcholů celkem a text s maximální délkou čtyři miliony znaků (hardwarově limitována velikost bitmap). Z použití bitmap také vyplývá, že může docházet k mírnému plýtvání paměti, protože bitmapa musí mít obdélníkový tvar a poslední řádek je tudíž zarovnán na požadovanou šířku nulami. Dále je nutná podpora shader modelu 3, aby bylo možné používat textury v rámci vertex shaderu. I přes zmiňované nedostatky je technika výhodná, protože s rostoucí délkou textu převyšuje rychlostně standardní metodu mnohonásobně. I u „krátkých“ textů jsem dosáhl i dvoj až trojnásobně vyšší rychlosti vykreslení.

Rasterizace v obou případech probíhá stejně jako u vyplňování uzavřených ploch.

```
1  float4 Array(sampler data, float index, float2 size)
2  {
3    float2 pos = float2(fmod(index, size.x), trunc(index / size.x));
4    return tex2Dlod(data, float4((pos + 0.5) / size, 0, 0));
5  }
6
7  VSOut TransformVertex(VSIn i)
8  {
9    float4 vertex = Array(VerticesSampler, i.Instance.x, VertexCount);
10   float4 glyph = Array(OffsetSampler, i.Instance.y, GlyphCount);
11   float3 pos = mul(float3(vertex.xy + glyph, 1), Transformation);
12   pos = mul(pos, Projection);
13
14   VSOut o;
15   o.Texcoord = vertex.zw;
16   o.Position = float4(pos, 1);
17   o.Position.xy += Offset;
18   return o;
19 }
```

Zdrojový kód 7 Vertex shader pro renderování textu

6. Aplikační rozhraní knihovny

Aplikační rozhraní knihovny je navrženo tak, aby umožňovalo intuitivní a snadné provádění všech operací zahrnutých v knihovně. Drtivá většina datových typů knihovny je obsažena ve jmenném prostoru *XnaVG* a implementuje rozhraní *IDisposable*. Kvůli výkonu aplikace je třeba důsledně dbát na uvolňování prostředků voláním metody *Dispose*.

6.1. Základní datové typy

Základní datové typy slouží ke snadnému uchovávání a efektivní manipulaci s nejčastějšími daty vektorové grafiky, jako jsou například matice a barvy.

6.1.1. Matice

Matice jsou elementární třídou grafických transformací. Jelikož se jedná o afinní transformace ve dvourozměrném prostoru, je třeba použít matici o rozměru 3x3. K reprezentaci takovéto matice a manipulaci s ní slouží třída *VGMatrix*. Třída implementuje operátor + a − pro součet a rozdíl matic, operátor ∗ pro násobení matic nebo vektoru maticí, operátor / pro dělení matic složku po složce či skalárem, unární operátor pro tvorbu inverzní matice a operátory == a ! = pro porovnávání dvou matic. Dále jsou zde zahrnuty statické metody pro snadnou tvorbu základních maticových transformací jako je posun, rotace, škálování nebo zkosení.

6.1.2. Barva

Barvy jsou reprezentovány prostřednictvím třídy *VGColor*. Tato třída je prakticky shodná s vestavěnou třídou XNA frameworku *Color*, ale na rozdíl od ní obsahuje navíc funkcionalitu pro převod mezi barevnými prostory sRGB, lRGB, sL a lL (standardní a lineární RGB a světelnost). Další rozdíl je ve způsobu uchovávání barev. Zatímco XNA uchovává barvy v přednásobeném formátu (všechny barevné složky jsou násobeny alfa kanálem), knihovna je ponechává v nepřednásobeném formátu, čímž je docíleno větší přesnosti barev během operaci s nimi. Pro převod mezi oběma třídami existují implicitní operátory a je tak možno využívat libovolnou z nich, avšak je třeba důsledně dbát na správnou reprezentaci barvy (přednásobení).

6.1.3. Barevná transformace

K uchování barevných transformací slouží třída *VGCxForm*. Transformace je definována pomocí dvou čtyřrozměrných vektorů (takzvané aditivní a multiplikativní složky). Při transformaci je barva nejprve násobena multiplikativní složkou a poté je přičtena složka aditivní. K provedení této transformace slouží

operátor násobení $*$. Pomocí něj je také možné transformace kombinovat (ve své podstatě se jedná o zjednodušené násobení matic).

6.1.4. Transformační zásobníky

Aby bylo možné jednoduše manipulovat s transformacemi během vykreslování, implementuje knihovna transformační zásobníky. Ty jsou reprezentovány třídami *VGMatrixStack* pro afinní transformace a *VGCxFormStack* pro barevné transformace. Obě se nacházejí uvnitř jmenného prostoru *XnaVG.Utils*. Kromě standardních metod pro práci se zásobníkem obsahují metody *PushCombineLeft* a *PushCombineRight*, které na vrchol zásobníku přidají hodnotu vzniklou násobením původního vrcholu zásobníku argumentem metody zleva respektive zprava. Tím je možné snadno kombinovat transformace například během vykreslování hierarchicky uspořádaných grafických objektů. Dále je zde pak zahrnuta metoda *Push* bez argumentů, která zdvojí hodnotu na vrcholu zásobníku.

6.1.5. Stav vykreslování

Během vykreslování je třeba znát různá nastavení, která specifikují, jak bude výsledný objekt zobrazen na plátno. K uchování všech těchto hodnot slouží stavová třída *VGState*. Hodnoty je možné přímo měnit a jedná-li se o aktivní instanci stavové třídy, bude tato změna zohledněna ve všech následujících operacích. Význam jednotlivých hodnot je popsán v příslušných následujících podkapitolách.

Důležitou hodnotou je *Projection*, což je maticový zásobník, jehož vrchol určuje, jak mají být souřadnice plátna mapovány na souřadnice povrchu. Ke snadnému nastavení slouží metoda *SetProjection*, která zásobník vyprázdní a přidá matici transformující souřadnice plátna o specifikovaných rozměrech na celou plochu povrchu.

6.2. Inicializace knihovny

Aby bylo možné knihovnu používat, je nutné ji inicializovat. K tomu slouží statická metoda třídy *VGDevice* s názvem *Initialize*, která vytvoří instanci rozhraní knihovny *IVGDevice* a zaregistruje jej mezi služby XNA Frameworku. Tuto instanci je možné získat buď jako návratovou hodnotu inicializační metody nebo prostřednictvím služeb (`Services.GetService(typeof(IVGDevice))`). Inicializaci je vhodné provádět uvnitř metody *Initialize* třídy *Game* respektive jejího potomka.

6.3. Zařízení

Výsledkem inicializace je objekt zařízení implementující rozhraní *IVGDevice*. Tento objekt je označován jako zařízení neboť je pevně vázán na grafický hardware aktuálně používaný XNA Frameworkem. Prostřednictvím metod instance

zařízení je možné vytvářet jednotlivé grafické objekty jakými jsou cesty, výplně, obrázky, fonty nebo povrchy. Dále pak obsahuje metodu *BeginRendering*, která vytvoří nový kontext a zahájí vykreslování.

6.4. Kontext

Kontext je hlavním objektem během vykreslování, neboť jeho prostřednictvím veškeré vykreslování probíhá. Je reprezentován objektem implementujícím rozhraní *IVGRenderContext*. Kontext je charakterizován zařízením, kterému náleží, povrchem, do kterého vykreslování probíhá a stavem řídícím parametry vykreslování.

Během vytváření je možné specifikovat i vlastní dodatečný stavový objekt. Ten je možné zpětně získat prostřednictvím vlastnosti *UserState* kontextu.

Toto rozhraní dále implementuje rozhraní *IDisposable* a tím pádem i metodu *Dispose*. Ta zajistí korektní ukončení vykreslování a případně obnovení původních hodnot stavových struktur rozhraní grafického zařízení XNA (bylo-li o to při zahájení kreslení požádáno). Tuto metodu je nutné volat vždy na konci vykreslování. Pro snadnější ošetření všech možných stavů se doporučuje používat bloku *using* jazyka C#.

6.5. Povrch

Povrch je objektovou reprezentací cíle vykreslování a je implementován pomocí třídy *VGSurface*. Může se jednat přímo o hlavní okno aplikace, jehož povrch lze získat prostřednictvím vlastnosti *WindowSurface* rozhraní *IVGDevice*, nebo o texturu v grafické paměti (všechny ostatní povrchy).

Každý povrch je charakterizován rozměry v pixelech, formátem pixelů (způsob uložení barev v grafické paměti), objektem textury (*null* v případě povrchu hlavního okna) a zařízením, kterému náleží.

Nový povrch lze získat zavoláním metody *CreateSurface* zařízení.

6.6. Cesty

Cesta je základním stavebním prvkem vektorové grafiky. Jedná se o vektorový popis obrysů vykreslovaného objektu. Cesta je reprezentována instancí třídy *VGPath*.

6.6.1. Tvorba cesty

Nejprve je nutné vytvořit instanci třídy *VGPath*. To je možné jak pomocí operátoru *new*, tak voláním metody CreatePath objektu zařízení.

Takto vytvořenou instanci je třeba naplnit daty. Za tímto účelem implementuje třída následující operace:

MoveTo – přesune pozici vykreslování na specifikované souřadnice,

ClosePath – uzavře aktuální úsek cesty zahájený operací MoveTo,

LineTo – vytvoří úsečku z aktuálních souřadnic do specifikovaných,

QuadraticTo – vytvoří kvadratickou Bézierovu křivku z aktuálních souřadnic do specifikovaných se zadaným řídícím bodem,

CubicTo – vytvoří kubickou Bézierovu křivku z aktuálních souřadnic do specifikovaných se zadanými řídícími body,

QuadraticSmoothTo – vytvoří kvadratickou křivku, ale jako řídící bod použije zrcadlový obraz předchozího řídícího bodu přes aktuální bod vykreslování,

CubicSmoothTo – podobné jako v případě kvadratické křivky; zrcadlí se pouze druhý řídící bod předchozího úseku,

Rectangle – vytvoří uzavřený obdélník.

Dále jsou zde zahrnuty metody pro posun a škálování cesty, vytváření identické kopie a skládání cest. Důležitou metodou je pak *Flatten*, která cestu převede do reprezentace tvořené pouze úsečkami. To je nezbytné například pro vykreslování obrysů.

6.6.2. Vykreslování cesty

Vykreslení cesty probíhá prostřednictvím volání metody *DrawPath* kontextu. Cestu je možno vykreslit ve dvou režimech a to *Fill* (vyplnění oblasti) nebo *Stroke* (vykreslení obrysů). Požadované režimy jsou předány jako argument metody a jedná se o prvky výčtového typu *VGPaintMode*, případně jejich logický součet.

Během vykreslování jsou zohledněny následující stavové hodnoty:

PathToSurface
: Transformační zásobník, jehož aktuální vrchol určuje transformaci souřadnic cesty na souřadnice plátna.

FillRule
: Pravidlo pro vyplňování cesty (viz. sekce 5.4.4.).

NonScalingStroke
: Logická hodnota určující, zda si mají obrysy zachovat stejnou tloušťku i po aplikování transformací.

StrokeThickness
: Tloušťka obrysu cesty před transformací.

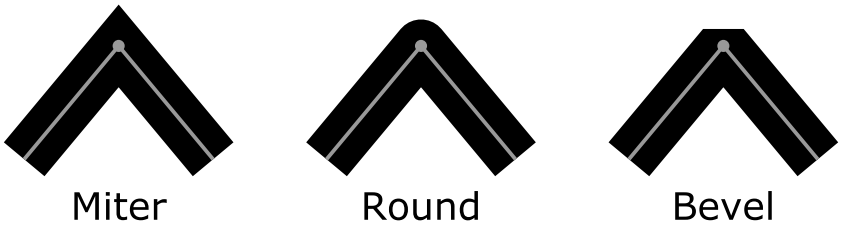
StrokeStartCap
Určuje, jak má být vykreslen začátek neuzavřeného obrysu.

StrokeEndCap
Určuje, jak má být vykreslen konec neuzavřeného obrysu.

StrokeJoin
Určuje, jak mají být vykresleny spoje mezi jednotlivými segmenty obrysu.

StrokeMiterLimit
V případě spojování segmentů obrysu typem spoje *Miter* určuje limit velikosti spoje po jehož překročení je nahrazen spojem typu *Bevel*.

Pro vykreslení spojů jsou k dispozici varianty *Bevel*, *Miter*, *Round* a *None*. Poslední zmíněná nevykreslí žádné spoje, význam ostatních hodnot je znázorněn na obrázku 5..



Obrázek 5. Spoje segmentů obrysu cesty

Pro vykreslení začátku a konce neuzavřeného obrysu (tzv. Caps) jsou k dispozici varianty *Butt*, *Triangle*, *Square* a *Round*. Význam hodnot je znázorněn na obrázku 6..

Obrázek 6. Zakončení obrysu cesty

6.6.3. Předteselace

Teselace cesty probíhá na procesoru počítače, což může mít v případě velkých cest neblahý dopad na výkon aplikace. Z tohoto důvodu umožňuje knihovna provést teselaci jednou a uložit takto připravenou cestu k opakovanému použití. Takováto cesta je pak reprezentována instancí třídy *VGPreparedPath* a je možné ji získat voláním metody *PrepareePath* objektu zařízení. Vykreslení se provádí stejně jako u normální cesty, avšak lze použít pouze režimy, pro které byla cesta připravena. Dále nejsou brány v potaz nastavení týkající se spojů a zakončení obrysů, protože ty jsou již pevně zahrnuté ve vygenerovaném objektu.

6.7. Bitmapy

V některých situacích je vhodné kombinovat bitmapovou grafiku s vektorovou. K tomu je možné využít buď přímo funkcionality XNA Frameworku nebo pro jednoduché operace třídu knihovny nazvanou *VGImage*, která reprezentuje bitmapový obrázek. Pro vytvoření instance je k dispozici metoda objektu zařízení nazvaná *CreateImage*. Té je nutné předat tři argumenty a to texturu reprezentující data obrázku, logickou hodnotu určující barevný prostor obrázku (sRGB nebo lRGB) a logickou hodnotu informující o přednásobení barevných složek složkou alfa. Takto vzniklé bitmapě je pak možné dále nastavit metodu filtrace (vlastnost *ImageFilter*) a způsob adresace (vlastnost *AddressMode*).

K vykreslení bitmapy slouží metoda kontextu *DrawImage*. Ta kromě obrázku umožňuje zvolit zdrojovou oblast, která má být vykreslena. Pro vykreslení celého obrázku je možné zadat hodnotu *null*. Jak bude obrázek umístěn na ploše plátna určuje maticový zásobník *ImageToSurface*. Matice na jeho vrcholu transformuje souřadnice bitmapy (pixely podobně jako v bitmapových editorech) na souřadnice plátna.

6.8. Výplně

Každý objekt musí být během vykreslování pokryt výplní. Všechny výplně implementované v knihovně jsou potomky abstraktní třídy *VGPaint* a nachází se ve jmenném prostoru *XnaVG.Paints*. K dispozici je hned několik různých druhů výplní.

6.8.1. Barevná výplň

První výplní je jednobarevná výplň. Ta je reprezentována třídou *VGColorPaint* a lze ji vytvořit voláním metody *CreateColorPaint* objektu zařízení. Jediným argumentem je požadovaná barva výplně.

6.8.2. Bitmapová výplň

Dalším druhem výplně je bitmapa. Tato výplň je reprezentována třídou *VGPatternPaint* a k jejímu vytvoření slouží metoda *CreatePatternPaint* zařízení. Jediným argumentem je instance požadované bitmapy.

6.8.3. Barevný přechod

Posledním druhem výplní jsou barevné přechody. K dispozici je lineární a radiální varianta. Jsou reprezentovány potomky třídy *VGGradientPaint* a to *VGLinearPaint* a *VGRadialPaint*. Barevné přechody je možné vytvořit pomocí metod *CreateLinearPaint* a *CreateRadialPaint*. Obě tyto metody berou stejné argumenty a to výčet zarážek a metodu interpolace. Každá zarážka je definována jako

dvojice hodnot typu *byte* a *VGColor* reprezentující její pozici a barvu. Interpolaci je možné provádět v lineárním nebo standardním barevném prostoru. Tato volba je reprezentována logickou hodnotou. Vytvořenému barevnému přechodu je pak možné nastavit další vlastnosti a to způsob filtrace bitmapy přechodu (vlastnost *GradientFilter*) a opakování přechodu (vlastnost *AddressMode*). V případě radiální výplně je pak možné posouvati fokální bod přechodu zleva doprava vlastností *FocalPoint*. Více informací o významu těchto hodnot se nachází v sekci 5.6.2..

6.8.4. Transformace a volba výplně

Aby bylo možné objekt správně zobrazit, je nutné informovat knihovnu o tom, jakou výplň má použít a jak jí na objekt nanést. To vše se provádí v rámci stavového objektu. Volbu výplně lze provést jednou z metod *SetFillPaint*, *SetStrokePaint* a *SetGlyphPaint* podle toho, na jaký druh objektu má být aplikována (výplň, obrys nebo text). K správnému namapování výplně slouží maticové zásobníky *PathToFillPaint*, *PathToStrokePaint* a *PathToTextPaint*. Tyto matice transformují souřadnice v rámci objektu na souřadnice výplně. V případě textu se za počátek soustavy souřadnic považuje nulový bod prvního znaku (zpravidla jeho levý okraj na úrovni řádku). Detailní popis souřadného systému výplně je taktéž k nalezení v sekci 5.6.2..

6.9. Text

Knihovna podporuje i základní vykreslování textu. K tomu je nejprve nezbytné definovat font a s jeho pomocí pak vykreslit požadovaný text.

6.9.1. Font

Font je základním stavebním kamenem textu. V rámci knihovny je reprezentován třídou *VGFont*. Skládá se z takzvaných glyphů (třída *VGGlyphInfo*), které jsou grafickou reprezentací jednotlivých znaků a obsahují informaci o velikosti znaku. Pro snadnou manipulaci je každému znaku (standardní dvoubajtový typ *char*) přiřazen jeden nebo žádný glyph. Znak který není definován je nahrazen znakem s číselnou reprezentací nula. Pokud ani ten neexistuje, není vykresleno nic. Každému fontu je možné přiřadit instanci podřezávací tabulky (*VGKerningTable*), která umožňuje upravit vzdálenost mezi dvojicí znaků a vylepšit tak vzhled textu.

Font může být načten ve dvou režimech. Prvním režimem je rozmísťování znaků na procesoru počítače a druhým je rozmísťování pomocí grafické karty. Vykreslování textu je detailně rozebráno v sekci 5.8..

6.9.2. Načtení fontu

Nejjednodušší metodou zahrnutí fontu do aplikace je využití funkcionality Content pipeline XNA Frameworku. Požadovaný font ve formátu OpenType (přípony *ttf* a *otf*) musí být přidán do projektu obsahu (content) aplikace a je při překladu v rámci pipeline přichystán pro rychlé načtení a vykreslování. Poté je v aplikaci standardní cestou pomocí metody *Content.Load* načten jako instance třídy *VGFontData*. Z těch je pak metodou zařízení *CreateFont* vytvořena příslušná reprezentace fontu. Takovýto font je statický a nelze za běhu měnit jeho grafickou reprezentaci, pouze podřezávací tabulku.

Aby bylo možné tento postup použít, je třeba přidat do referencí content projektu pomocnou knihovnu *XnaVGPipeline*, která obstarává načítání a zpracování formátu OpenType.

6.9.3. Generování fontu za běhu

Další metodou definice fontu je vygenerovat jej za chodu z cest. Nejprve je nutné zavolat metodu zařízení *CreateFont* a předat jí veškeré potřebné informace o fontu. Jedná se o pravidlo vyplňování, rozměry Em čtverce [7], výšku řádku, definici znaků a režim renderování textu. Takto vzniklý font je taktéž dále neměnitelný.

Je-li požadována možnost měnit vzhled jednotlivých znaků za chodu aplikace, je možné vytvořit font pomocí metody zařízení *CreateDynamicFont*. Rozmísťování znaků v tomto případě vždy probíhá na procesoru počítače.

6.9.4. Textové řetězce

Vykreslení textových řetězců má na starosti metoda kontextu *DrawText*. Font použitý k vykreslení řetězce je definován stavovou hodnotou *Font*. Na celý řetězec je poté aplikována transformace z vrcholu zásobníku *GlyphToSurface*.

Protože je rozmísťování znaků textu relativně náročná operace, je možné pro často používané řetězce nechat vygenerovat jejich výslednou podobu a tím ušetřit strojový čas nutný k opakovanému rozmísťování znaků. Pro manipulaci s touto reprezentací řetězce slouží třída *VGString*. Její instanci lze vytvořit voláním metody *CreateString* fontu, jímž má být řetězec vykreslen. Poté je třeba přiřadit mu jeho textovou hodnotu. K tomu slouží metoda *SetText*. Vykreslení se poté provádí stejně jako u standardního řetězce.

6.10. Doplňující informace

6.10.1. Manipulace s texturami

Při práci s texturou je často třeba vytvořit její kopii nebo přenést její data do jiné textury, případně na ně aplikovat nějaký efekt. Veškerá tato funkcionalita

je ve své nejjednodušší podobě v knihovně zahrnuta uvnitř třídy *VGTextureUtils* jejíž instanci lze získat prostřednictvím vlastnosti *TextureUtils* objektu zařízení.

6.10.2. Formát TGA

XNA Framework v sobě zahrnuje metody pro práci s některými bitmapovými formáty. Jediný z nich s podporou průhlednosti je formát PNG. Bohužel XNA obsahuje chybu, která při ukládání obrázku ve formátu PNG způsobuje únik paměti. Z tohoto důvodu jsem do knihovny zahrnul velmi jednoduchou statickou třídu pro nejzákladnější manipulaci s formátem Truevision Targa (TGA). Nese název *TgaImage* a nachází se ve jmenném prostoru *XnaVG.Loaders*. Její využití je výhodné zejména při ladění.

7. Srovnání s podobnými knihovnami

Jako součást práce bylo stanoveno i jednoduché ale objektivní srovnání knihovny s ostatními podobnými existujícími knihovnami.

7.1. Přehled srovnávaných knihoven

Volba srovnávaných knihoven nebyla zrovna jednoduchá, neboť knihovna implementovaná v rámci této práce je svým způsobem hodně specifická díky její přímé vazbě na trojrozměrnou grafickou pipeline. Nakonec jsem ke srovnání využil knihoven *Cairo* jakožto jedné z nejrozšířenějších otevřených knihoven s podporou hardwarové akcelerace a *OpenVG* jakožto knihovně snažící se o zavedení standardu pro vykreslování vektorové grafiky.

7.1.1. Cairo

Cairo je otevřená knihovna pro vykreslování vektorové grafiky s podporou hardwarové akcelerace. Je nezávislá na platformě a v současné době podporuje vykreslování prostřednictvím rozhraní GDI (Windows), OpenGL, X (Linux) a BeOS. Je šířena pod licencemi GNU Lesser General Public License 2.1 a Mozilla Public License 1.1. Knihovna je napsána v jazyce C, ale lze ji použít i v mnoha jiných jazycích.

7.1.2. OpenVG

OpenVG je aplikační rozhraní navržené s cílem sjednotit a standardizovat hardwarově akcelerované vykreslování dvourozměrné vektorové grafiky. Je vyvíjeno a spravováno konsorciem Khronos group. Dle specifikace je OpenVG navrženo tak, aby umožňovalo snadné vykreslení souborů ve formátech SVG a Flash (SWF), ale bylo přímočaře implementovatelné na specializovaných čipech. Primárně cílí do oblasti spotřební elektroniky. Na platformě PC je momentálně dostupná pouze jediná kompletní implementace a to referenční. Ta bohužel vůbec nevyužívá grafické karty a je proto velmi pomalá.

7.2. Srovnání funkcionality knihoven

Knihovna *XnaVG* vyvinutá v rámci této bakalářské práce umožňuje vykreslování cest sestavených z úseček a Bézierových křivek, podporuje všechny základní varianty spojování a zakončování segmentů cest. Jsou zahrnuty základní druhy výplní a to výplň jednou barvou, bitmapou, lineárním přechodem a radiálním přechod (včetně možnosti specifikovat fokální bod) i s podporou afinních i barevných transformací. Dále knihovna umožňuje maskování pomocí až sedmi

uživatelsky definovaných jednobitových masek (stencil masky) a čtyř osmibitových masek (alfa maskování) uložených v jednom obrázku. Jsou podporovány obě základní pravidla vyplňování (Even-Odd a NonZero) a škálovatelné i neškálovatelné vykreslování obrysů. Taktéž je zahrnuta základní podpora vykreslování textu včetně podpory podřezávání znaků. Umožňuje načítat formáty Truevision Targa a OpenType. Nespornou výhodou je možnost předzpracování statických dat pro zrychlení vykreslování scény.

Knihovna *Cairo* je po letech svého vývoje jednou z nejrozšířenějších a nejobsáhlejších knihoven v této oblasti. Co se týče samotného vykreslování vektorové grafiky, obsahuje veškerou funkcionalitu zahrnutou ve vytvářené knihovně. Na rozdíl od ní ale navíc podporuje přidávat do cest obloukové segmenty, má daleko rozsáhlejší a dokonalejší funkcionalitu pro vykreslování textu mimo jiné s využitím různých knihoven dostupných na aktuální platformě. Dále přímo umožňuje načítání a ukládání grafiky v mnoha vektorových formátech jakými jsou například PDF, PostScript, SVG a další. Také podporuje vytváření skriptů pro opakování častých operací. Ačkoli podporuje hardwarovou akceleraci tam, kde je dostupná, tak díky podpoře různorodých grafických rozhraní má jen omezenou možnost optimalizace vykreslování opakujících se dat.

Specifikaci rozhraní *OpenVG* jsem vedle specifikace SWF hojně využíval při návrhu aplikačního rozhraní vytvářené knihovny. Je zde tedy v mnoha ohledech patrná podobnost. Jelikož se jedná o rozhraní navržené pro přímou komunikaci s grafickým čipem, předpokládá se, že veškerá data jsou již ve formátu, kterému bude cílový hardware rozumět. Tím odpadá potřeba předzpracovávání dat. Na rozdíl od knihovny *XnaVG* umožňuje provádět některé další operace s cestami a přímo podporuje manipulaci s alfa maskami a zahrnuje základní bitmapové filtry. Naopak neumožňuje renderování neškálovatelných obrysů. Veškerá chybějící funkcionalita, která je v *OpenVG* zahrnuta může být „snadno“ implementována i v prostředí XNA Frameworku potažmo vytvořené knihovny, ale z důvodu optimalizace pro potřeby konkrétních aplikací (nároky na grafický hardware, zejména pak video paměť) je ponechána zcela v kompetenci vývojářů.

8. Ukázková aplikace

8.1. Popis aplikace

Jako ukázková aplikace byl vytvořen jednoduchý přehrávač formátu Flash, který umožňuje snímek po snímku procházet klipy v tomto formátu a nebo je nechat celé přehrávat. Podporovány jsou prakticky veškeré grafické možnosti formátu až do verze *SWF 7* včetně. Jedinou výjimkou jsou takzvané *Morph tweeny*, které jsou ve své podstatě transformací jednoho grafického objektu na druhý. Tato animace by vyžadovala další rozsáhlejší modifikace knihovny, na které již bohužel při vývoji nezbyl potřebný čas. Taktéž nejsou podporovány interaktivní funkce formátu, přehrávání zvuků a videa. Jedná se zde o velmi rozsáhlý soubor možností, který se nachází mimo rozsah i zaměření práce.

K vykreslení textu jsou využívány pouze fonty zahrnuté uvnitř souboru SWF, nikoli systémové fonty.

8.2. Ovládání aplikace

Po spuštění aplikace se objeví hlavní okno obsahující pouze hlavní nabídku. Hlavní nabídka obsahuje následující položky:

Soubor – Otevřít
: slouží k načtení klipu do aplikace,

Soubor – Zavřít
: uzavře přehrávání aktuálního klipu,

Soubor – Uložit snímek
: umožňuje uložit aktuální pohled do souboru,

Soubor – Konec
: ukončí aplikaci,

Pohled – Celá obrazovka
: slouží k přepnutí režimu celé obrazovky,

Pohled – Nízká kvalita
: vypne vyhlazování,

Pohled – Střední kvalita
: aktivuje čtyřnásobný multisampling (je-li podporován),

Pohled – Vysoká kvalita
: aktivuje osminásobný multisampling (je-li podporován),

Pohled – Nejvyšší kvalita
aktivuje šestnáctinásobný multisampling (je-li podporován),

Přehrávání – Pozastavit
umožňuje pozastavit nebo spustit přehrávání,

Přehrávání – Návrat
přesune přehrávání na první snímek klipu,

Přehrávání – Další snímek
přejde na další snímek klipu,

Přehrávání – Předchozí snímek
přejde na předchozí snímek klipu,

Přehrávání – Opakovat
zapne nebo vypne opakování klipu.

K dispozici jsou také následující klávesové zkratky pro jednotlivé položky nabídky:

Alt + F4 – ukončit aplikaci,

Ctrl + O – otevřít klip,

Ctrl + W – zavřít klip,

Ctrl + S – zachytit snímek obrazovky,

Ctrl + F – celá obrazovka,

Ctrl + F1 – nízká kvalita,

Ctrl + F2 – střední kvalita,

Ctrl + F3 – vysoká kvalita,

Ctrl + F4 – nejvyšší kvalita,

Ctrl + Enter – pozastavení klipu,

Ctrl + R – návrat na první snímek,

Ctrl + Vlevo – předchozí snímek,

Ctrl + Vpravo – další snímek,

Ctrl + L – opakování klipu,

Escape – alternativní klávesa pro opuštění režimu celé obrazovky.

8.3. Formát SWF

Formát SWF (čteme „swiff“) je primárně určen pro snadné šíření interaktivní vektorové grafiky prostřednictvím internetu. K tomu využívá hned několik různých metod uložení a komprese dat čímž umožňuje rychlé stažení souboru a následně plynulé vykreslování obsažené grafiky. Byl vyvinut FutureWave Software, poté přešel pod společnost Macromedia a od roku 2005 jej udržuje společnost Adobe. Na webových stránkách společnosti je k dispozici prozatím poslední verze specifikace v elektronické podobě [1].

8.3.1. Struktura souboru

První částí souboru je záhlaví. V něm jsou obsaženy informace o klipu, jako je metoda komprese, verze souboru, rozměry klipu nebo rychlost přehrávání. Následuje sled takzvaných tagů. Každý tag obsahuje informaci o tom, jaká data nese a kolik bytů jeho data zabírají. Tím je umožněno snadné přeskočení tagů, které nejsou přehrávačem podporovány. Tagy se dělí do dvou kategorií a to na řídící a definiční. Řídící tagy kontrolují průběh přehrávání a rozmísťování objektů na plátno, definiční tagy slouží k popisu multimediálních dat (grafika, zvuk, video, aj.) nazývaných symboly. Jejich pořadí musí být takové, aby definiční tagy vždy předcházely tagům řídícím, které s definovaným objektem pracují.

8.3.2. Komprese

Aby byl soubor co nejmenší, využívá formát SWF tři různé metody komprese dat.

První metodou je co největší odstranění redundance dat pomocí jejich vhodného definování a rozložení v souboru.

Druhou metodou je pak reprezentace hodnot s využitím bitové komprese. Jednotlivé položky struktur, které by s vysokou pravděpodobností obsahovaly mnoho nevyužitých bitů jsou reprezentovány jen s využitím nezbytně nutného počtu bitů případně ve speciálních formátech. Tím je dosaženo relativně velké úspory místa.

Poslední metodou je nepovinná komprese celého souboru (s výjimkou prvních osmi bytů záhlaví) s využitím otevřeného standardu ZLIB. O tomto faktu je přehrávač informován pomocí prvního bytu v souboru, který odpovídá číselné hodnotě znaku „C“ podle tabulky ASCII.

8.3.3. Slovník

Při zpracovávání souboru SWF hrají zásadní roli dvě speciální datové struktury. První z nich je takzvaný slovník. Jedná se o strukturu, ve které jsou uchovávány všechny symboly popsané definičními tagy. Každý tagový tag obsahuje kromě dat symbolu i jeho jednoznačný identifikátor reprezentovaný dvoubaj-

tovou neznaménkovou hodnotou. V okamžiku načtení definičního tagu je symbol ihned přidán do slovníku a tím připraven k použití řídícími tagy.

8.3.4. Display list

Druhou datovou strukturou je Display list. Jedná se o vzestupně setříděný seznam obsahující odkazy na symboly ze slovníku s informacemi o jejich umístění ve scéně. Pořadí symbolu ve scéně je určeno hloubkou, ve které se nachází. Čím nižší je hloubka, tím více je symbol vzadu. Opět se jedná o dvoubajtovou neznaménkovou hodnotu. Z toho vyplývá, že v jednom okamžiku může být na scéně maximálně 65536 symbolů. V prvních verzích formátu byl Display list obyčejný seznam, ale s příchodem takzvaných spritů bylo umožněno vnořování symbolů do sebe a tím se stal defakto hierarchickým stromem. O umísťování a odebírání symbolů ze scény se starají k tomu určené řídící tagy. Mezi jednotlivými snímky klipu jsou vždy uloženy pouze změny a proto je při přeskakování nutné vždy projít všechny přeskočené snímky. Výhodou je ovšem rychlé vykreslování v případě animací.

8.4. Zpracování a přehrávání klipu

K načítání a vykreslování formátu SWF jsem v rámci práce využil jeden ze svých nedokončených projektů z dřívější doby. Ten jsem navázal na knihovnu *XnaVG* vyvinutou v rámci práce a tím vznikla samostatná knihovna *XnaFlash*. Ta je rozdělena do tří jmenných prostorů *Swf* (načítání), *Content* (reprezentace symbolů pomocí objektů knihovny XnaVG) a *Movie* (instance klipů a jejich řízení a vykreslování). V době psaní práce obsahuje i další funkcionalitu, která však není v rámci ukázkové aplikace využita, neboť se nachází v nestabilním nebo nedokončeném stavu a je pouze nachystána na dokončení v případě potřeby.

8.4.1. Parsování

K parsování formátu SWF slouží třída *SwfStream*. Ta obsahuje veškeré metody potřebné k načítání bitových hodnot a dalších obsažených datových typů a struktur z předaného proudu dat. V případě komprese ZLIB se stará i o dekompresi. K tomu využívá otevřené [5] knihovny *SharpZipLib*. Výstupem je pak výčet tagů reprezentovaných objekty implementujícími rozhraní *ISwfTag*.

8.4.2. Generování grafických dat

K reprezentaci dat klipu je určena třída *FlashDocument*. Ta postupně prostřednictvím instance třídy *SwfStream* načítá a zpracovává jednotlivé tagy a na jejich základě generuje grafická data srozumitelná pro knihovnu XnaVG. Tyto data pak mohou být využita k vytvoření několika různých instancí klipu

(vhodné například k zobrazení dvou stejných ukazatelů zobrazujících odlišné hodnoty). Tím je ušetřena paměť a také čas potřebný k vytvoření další instance klipu.

8.4.3. Vykreslování

Posledním krokem je řízení a vykreslování přehrávání klipu. Základním stavebním kamenem v procesu vykreslování je struktura Display listu, která obsahuje jednotlivé instance symbolů. V případě symbolu typu sprite obsahuje každá jeho instance vlastní DisplayList. Těmto instancím se v terminologii programu Flash říká *MovieClip* a tento název nese i příslušná třída uvnitř knihovny. Z objektového hlediska je spritem i hlavní časová osa, jen zahrnuje několik málo rozšíření. Proto se přímo nabízí implementace poděděním třídy *MovieClip*. Tento potomek je nazván *RootMovieClip*.

Celý proces vykreslování je přímočarý, neboť knihovna přímo podporuje všechny grafické operace obsažené ve formátu SWF. Během vykreslování je DisplayList rekurzivně procházen vzestupně podle hloubky a symboly jsou takto postupně vykreslovány. Tento proces usnadňuje využití maticových zásobníků protože pro správné transformace jednotlivých (a to i vnořených) symbolů stačí před vykreslením přidat transformační matici na jeho vrchol násobením zleva a po vykreslení ji z vrcholu zásobníku odstranit.

Některé symboly mohou sloužit i k maskování obrazu. K tomu je využito stencil masek implementovaných v knihovně. Podporováno je až sedm vnořených masek. Každá takováto maska má specifikovanou hloubku, do které instance symbolů maskuje.

8.4.4. Řízení klipu

Ke snadné manipulaci s instancemi klipů byla vytvořena třída *Flash*. Ta implementuje abstraktní třídu XNA Frameworku *DrawableGameComponent* a umožňuje tak přímočaré začlenění do herní logiky vhodné například pro tvorbu menu (samozřejmě v případě doimplementování interaktivity). Každá instance této třídy obsahuje právě jednu instanci třídy *RootMovieClip* reprezentující hlavní časovou osu, jejímž prostřednictvím lze přistupovat k jednotlivým instancím symbolů a řídit tak průběh celého klipu.

Závěr

Hlavním cílem této bakalářské práce bylo ukázat, že je možné pomocí standardních aplikačních rozhraní pro vykreslování 3D grafiky efektivně zobrazovat i grafiku vektorovou a vytvořit adekvátní demonstraci. Za tímto účelem byly vytvořeny dvě knihovny a to knihovna *XnaVG* umožňující snadné renderování vektorové grafiky a dále knihovna *XnaFlash* pro základní zpracování klipů ve formátu SWF. Tím byl tento cíl v rámci možností naplněn. Veškeré technické požadavky ze zadání byly dodrženy a navíc byla zahrnuta i dodatečná funkcionalita, jakou je například zobrazování textu.

Vedlejším cílem pak bylo srovnat možnosti vzniklé knihovny s podobnými existujícími knihovnami. K tomu byly využity dvě podobné knihovny a srovnány po stránce integrované funkcionality.

Možným pokračováním práce by bylo rozšířit funkcionalitu obou knihoven o podporu všech možností poslední revize formátu SWF, avšak vzhledem k zastavení vývoje XNA Frameworku a jeho postupnému úpadku se jako přijatelnější alternativa jeví jejich kompletní přepracování do jiného aplikačního rozhraní nebo frameworku. Díky tomu by bylo možné využít veškeré funkcionality poskytované moderním grafickým hardwarem a přesunout tak celé vykreslování z procesoru počítače na grafickou kartu.

Conclusions

The main objective of this bachelor thesis was to show that it is possible to efficiently display vector graphics using the standard 3D rendering APIs, and to create an adequate demonstration. For this purpose two libraries have been created — *XnaVG* providing interface for simple vector graphics rendering and *XnaFlash* for processing and displaying movies represented by the SWF file format. All the technical requirements were fulfilled and some additional functionality, such as displaying text, is also included.

The secondary objective was to compare the functionality offered by this library with similar existing libraries. Two of them were used and compared in term of integrated functionality.

One of the possible extensions of this work would be to extend the functionality of both libraries by supporting all the possibilities mentioned in the last revision of the SWF file format, but due to the discontinued developement of the XNA Framework and its future decline I prefer to rewrite both libraries completely using any alternative 3D rendering API. In this case it would be possible to use all the functionality provided by modern graphics hardware to move the rendering completely away from CPU.

Reference

[1] Adobe Systems, Inc. SWF File Format Specification – Version 19, Elektronická publikace, dostupná ke dni 28.7.2013.

[2] Jim Blinn, Charles Loop. *Rendering Vector Art on the GPU*, Elektronická publikace, 2005.

[3] Mark J. Kilgard. *Vector Graphics & Path Rendering*, Elektronická prezentace, 2012.

[4] Mark J. Kilgard, Jeff Bolz. *GPU-accelerated Path Rendering*, Elektronická publikace, 2012.

[5] Kolektiv IC#Code. .NET Zip Library, Domovská stránka knihovny SharpZLib, dostupná ke dni 28.7.2013.

[6] Kolektiv Wikipedie. De Casteljau's algorithm, Elektronická publikace, verze ke dni 28.7.2013.

[7] Kolektiv Wikipedie. Em unit, Elektronická publikace, verze ke dni 28.7.2013.

[8] World Wide Web Consorcium. Scalable Vector Graphics (SVG) 1.1, Elektronická publikace, verze ke dni 28.7.2013.
Z tohoto zdroje jsou převzaty některé z obrázků použitých v práci.

A. Obsah přiloženého CD

Na tomto místě uvádím stručný soupis obsahu přiloženého CD.

adresář Bin
: Zde se nachází spustitelný soubor ukázkové aplikace a dll soubory využitých knihoven.

adresář Doc
: Tento adresář obsahuje zdrojový soubory a soubor PDF této práce.

adresář Media
: V tomto adresáři je umístěno několik jednoduchých klipů ve formátu SWF pro účely testovací aplikace

adresář Redist
: Tento adresář obsahuje software nezbytný k provozu aplikace.

adresář Src
: Na tomto místě se nachází veškeré zdrojové kódy obou knihoven a ukázkové aplikace.

readme.txt
: Soubor obsahující tento soupis a pokyny ke spuštění ukázkové aplikace.