VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
FACULTY OF INFORMATION TECHNOLOGY

ÚSTAV POČÍTAČOVÉ GRAFIKY A MULTIMÉDIÍ
DEPARTMENT OF COMPUTER GRAPHICS AND MULTIMEDIA

BLENDER PLUGIN PRO PŘEVOD MODELŮ NA VEKTOROVOU GRAFIKU

BLENDER PLUGIN FOR CONVERSION OF MODELS TO VECTOR GRAPHIC

BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE **JIŘÍ KOPÁČEK**
AUTHOR

VEDOUCÍ PRÁCE **Ing. TOMÁŠ MILET**
SUPERVISOR

BRNO 2021

Vysoké učení technické v Brně
Fakulta informačních technologií

Ústav počítačové grafiky a multimédií (UPGM) Akademický rok 2020/2021

Zadání bakalářské práce

Student: **Kopáček Jiří**
Program: Informační technologie
Název: **Blender plugin pro převod modelů na vektorovou grafiku**
Blender Plugin for Conversion of Models to Vector Graphic
Kategorie: Počítačová grafika

Zadání:

1. Naučte se základy práce s 3D modelovacím programem Blender (verze 2.8 a výše).
2. Seznamte se se skriptováním v Blender Python API.
3. Navrhněte plugin pro převod modelů na vektorovou grafiku (svg).
4. Implementujte navržený plugin a demonstrujte jeho použití na různých typech dat.
5. Zdokumentujte a zveřejněte výsledný plugin pro použití dalšími uživateli.
6. Vytvořte video prezentující výsledky vaší práce.

Literatura:

- Dle zadání vedoucího.

Pro udělení zápočtu za první semestr je požadováno:

- Body 1 až 3, experimenty vedoucí k plné implementaci.

Podrobné závazné pokyny pro vypracování práce viz https://www.fit.vut.cz/study/theses/

Vedoucí práce: **Milet Tomáš, Ing.**
Vedoucí ústavu: Černocký Jan, doc. Dr. Ing.
Datum zadání: 1. listopadu 2020
Datum odevzdání: 12. května 2021
Datum schválení: 30. října 2020

Abstrakt

Tato práce řeší návrh a implementaci rozšíření (pluginu) pro software Blender. Toto rozšíření přidává funkcionalitu převodu 3D modelů ve scéně Blenderu na soubor 2D vektorové grafiky ve formátu SVG tak, aby tento soubor reprezentoval původní scénu. Práce se zabývá tvorbou uživatelského rozhraní pluginu, převodem souřadnic vrcholů modelů ze 3D souřadnic scény na 2D souřadnice okna, výpočtem barvy výsledných polygonů na základě osvětlení a barvy materiálu, řezáním konfliktních (zaklíněných) objektů a určováním pořadí, v jakém je třeba výsledné polygony vykreslit. Výsledkem implementace je plugin, který umožňuje uživateli namísto ruční tvorby 2D vektorových obrázků vytvářet 3D modely, které lze následně ze scény Blenderu převádět („fotit“) do souborů vektorové grafiky ve formátu SVG.

Abstract

This thesis deals with the design and implementation of an extension (plugin) for Blender software. This extensions adds the feature of converting 3D models in a Blender scene to a 2D vector graphics file in the SVG format that represents the original scene. The thesis focuses on creating the plugin's user interface, conversion of vertex coordinates from 3D scene to 2D window coordinates, calculating the resulting polygon color based on lighting and material, cutting of conflicting (colliding) objects and determining the order in which the resulting polygons have to be drawn. The result of the implementation is a plugin that allows the user to create 3D models which can be converted (snapshot) to SVG formatted vector graphics files, instead of manually creating 2D vector images.

Klíčová slova

převod modelu, 3D na 2D, 3D na SVG, model na SVG, vektorová grafika, hloubkové řazení, detekce kolizí, řezání, oktálový strom, Blender, plugin, addon, Python

Keywords

model conversion, 3D to 2D, 3D to SVG, model to SVG, vector graphics, depth sorting, collision detection, cutting, octree, Blender, plugin, addon, Python

Citace

KOPÁČEK, Jiří. *Blender plugin pro převod modelů na vektorovou grafiku.* Brno, 2021. Bakalářská práce. Vysoké učení technické v Brně, Fakulta informačních technologií. Vedoucí práce Ing. Tomáš Milet

Blender plugin pro převod modelů na vektorovou grafiku

Prohlášení

Prohlašuji, že jsem tuto bakalářskou práci vypracoval samostatně pod vedením pana Ing. Tomáše Mileta. Uvedl jsem všechny literární prameny, publikace a další zdroje, ze kterých jsem čerpal.

.....
Jiří Kopáček
4. května 2021

Poděkování

Rád bych poděkoval vedoucímu práce Ing. Tomáši Miletovi za veškeré úsilí, které věnoval pravidelným konzultacím, a především za původní myšlenku, na které bylo toto zadání a celkově tato práce postavena. Dále bych chtěl také poděkovat své rodině a přátelům za veškerou podporu během studia.

Obsah

Kapitola 1

Úvod

Vektorová a rastrová grafika jsou v počítačové grafice dva základní způsoby reprezentace obrazu. Vektorová grafika je složena ze základních útvarů, které jsou přesně definovány a lze je tedy libovolně zvětšovat a zmenšovat bez ztráty kvality obrazu, na rozdíl od rastrové grafiky, která pouze definuje hodnoty barev jednotlivých pixelů v určitém rozlišení. Díky této vlastnosti se vektorová grafika používá například pro tvorbu diagramů nebo ilustrací v odborných textech.

Častou nevýhodou vektorové grafiky je tvorba samotného obrázku, která může být především u složitějších ilustrací poměrně pracná. Náročnou činností například může být tvorba vektorových obrázků reprezentujících určitou scénu/model v prostoru, například z více úhlů pohledu. Zjednodušení tvorby vektorových obrázků takových typů bylo hlavní motivací pro tuto práci.

Tento problém lze zjednodušit tak, že scéna nebo objekt, které je třeba ilustrovat, budou vymodelovány ve 3D a poté „vyfoceny“ z požadovaných úhlů. Toho je samozřejmě možné docílit modelováním a poté vykreslením v libovolném 3D modelovacím softwaru (v případě této práce je uvažován nástroj Blender), výsledkem tohoto procesu by však v takovém případě byl rastrový obrázek.

Zmíněný nástroj Blender umožňuje uživatelům vytvářet pluginy (rozšíření) pomocí rozhraní pro programování aplikace – Blender Python API (dále pouze API).

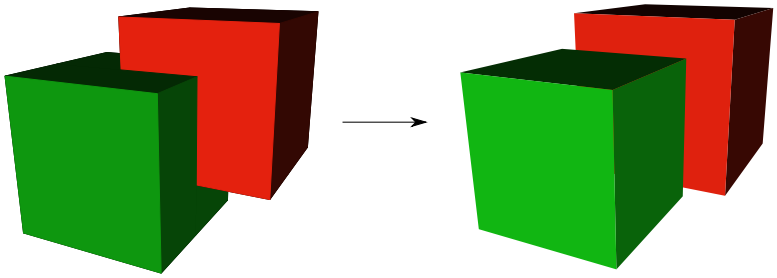
Cílem této práce bylo API prostudovat a vytvořit plugin v jazyce Python, který rozšíří funkcionalitu Blenderu o možnost převodu (resp. „vyfocení“) modelů ve scéně takovým způsobem, že výsledkem bude obrázek ve formě vektorové grafiky, přesněji soubor ve formátu SVG (Scalable Vector Graphics). Tento soubor by po vykreslení libovolným nástrojem, kterým může být například webový prohlížeč, editor vektorové grafiky apod., měl dát výsledný obraz co nejvíce podobný původním vyfoceným modelům ve scéně Blenderu.

Na úvod práce jsou zmíněny současné stavy existujících řešení stejné problematiky (kapitola 2). Dále se tato práce zabývá obecným formátem vstupních a výstupních dat převodu, základními kroky tohoto převodu a také algoritmy, metodami a modely, které jsou pro něj v případě této práce relevantní (kapitola 3), větší pozornost je věnována problematice hloubkového řazení jednotlivých objektů do souboru SVG (podkapitola 3.6), která se stala největší překážkou této práce. Následuje návrhová část zabývající se formátem dat v API, podrobnějším popisem převodu z praktičtějšího hlediska, návrhem uživatelského rozhraní a tvorbou pluginů v Blenderu (kapitola 4). Poté je probrána implementace, konkrétně jednotlivé třídy tohoto pluginu a jejich funkcionalita (kapitola 5). Pro lepší názornost následuje závěrečná kapitola představující výsledky, kterých lze dosáhnout s pomocí tohoto pluginu při práci s různými typy modelů (kapitola 6).

Kapitola 2

Stav existujících řešení

Přestože se jedná o poměrně specifické řešení převodu modelu na vektorovou grafiku v prostředí Blenderu, je nutno zmínit, že v současnosti už jedno podobné řešení existuje. Jedná se o skript „Viewport to SVG“ od uživatele Liero[1] pro verzi Blenderu 2.80. Tento skript nabízí velké množství nastavení a přizpůsobení převodu, které kromě převodu modelu zprostředkovává také převod křivek. Mimo to umožňuje náhodné obarvování modelů na základě palety barev, přizpůsobení tahů, různé speciální efekty pro jednotlivé vrcholy a plochy modelů a podobně. Mezi slabiny však patří například to, že uživatel nemůže určovat pozici světla ve scéně mimo primitivní nastavení předního a zadního světla a také především absence hloubkového řazení. Jednotlivé objekty scény jsou v rámci tohoto pluginu seřazeny dle vzdálenosti od kamery a poté vykresleny celé, což znamená, že dva objekty, které spolu kolidují, nebudou vykresleny korektně, neboť první bude překreslen tím druhým. Situaci lze vidět na obrázku 2.1.



Obrázek 2.1: Převod zaklíněných objektů už existujícím pluginem. Nalevo lze vidět objekty ve scéně Blenderu, které spolu kolidují. Napravo lze vidět výsledek převodu už existujícím pluginem. Je zřejmé, že správné řazení jednotlivých ploch modelů bylo zanedbáno a řazeny byly pouze objekty jakožto celky.

Protože výše zmíněný plugin už nabízí širokou škálu možností převodu, není cílem této práce jej co nejvíce napodobit a implementovat ještě více nastavení převodu. Naopak se tato práce zaměřuje především na nedostatky předchozího pluginu, tedy na ten menší, kterým je možnost určování polohy zdroje světla a celkově proces stínování a poté na ten podstatnější, kterým je hloubkové řazení v rámci jednotlivých ploch scény, ne pouze řazení samotných objektů jakožto celků.

[1] Vlákno fóra komunity Blenderu publikující plugin: https://blenderartists.org/t/svg-output-script/566412

Kapitola 3

Pojmy, metody a algoritmy potřebné pro převod

Ačkoliv cílem této práce bylo vytvořit kompletní plugin pro Blender, jádrem samotného řešení je z obecnějšího pohledu převod vstupních 3D dat (modelů) na výstupní 2D data (soubor ve formátu SVG). Následující kapitola slouží jako teoretický úvod do problematiky tohoto převodu, proto se nejdříve zabývá formou vstupních a výstupních dat a poté krátce popisuje, jakými kroky je převod mezi těmito daty zajištěn. Některé části tohoto procesu ovšem nejsou natolik triviální, aby bylo možno je kompletně shrnout v rámci popisu jednotlivých kroků, a zaslouží si bližší specifikaci. Proto jsou dále rozebírány tyto vybrané metody, modely a algoritmy, mezi které patří metoda pro vynechání odvrácených ploch (Backface culling), algoritmus pro ořez částí ploch ležících mimo okno (Clipping), model použitý pro výpočet stínování a na závěr je také podrobně popsána problematika hloubkového řazení (respektive hledání a řezání konfliktních ploch a jejich řazení), které se stalo hlavním problémem při snaze získat korektní výsledný obraz za všech situací.

Přestože je následující kapitola převážně teoretická, v některých pasážích se nachází zmínky o návrhové či implementační části, především kvůli opodstatnění výběru nebo zavrhnutí některých metod a vylepšení.

3.1 Vstupní a výstupní data převodu

Tato podkapitola je pouze krátkým úvodem vstupních a výstupních dat převodu potřebným pro lepší pochopení následujících sekcí. Další podrobnější popis formátu dat, který je potřebný pro správný návrh a následnou implementaci, lze nalézt v podkapitole návrhové části 4.1.

Protože je v rámci této práce převáděn model na vektorovou grafiku, je zřejmé, že vstupem budou 3D data popisující tělesa. Tělesa budou reprezentována jedním z nejběžnějších způsobů, a to popisem hranice modelu, což je v podstatě množina vrcholů, hran a ploch společně definujících tvar objektu. Spojení dvou vrcholů tvoří hrany, sekvence těchto hran poté tvoří hranici polygonu (mnohoúhelníku, dále bude používán pro stručnost termín polygon) reprezentujícího plochu objektu. Všechny tyto polygony dohromady tvoří polygonovou síť, která udává výsledný tvar tělesa. Tento typ popisu neuchovává žádné informace o vnitřních bodech tělesa [1, s. 240–243]. Dále lze reprezentovat i barvu této plochy tak, že danému polygonu je přiřazen materiál, jehož vlastnosti jsou použity při počítání výsledné barvy při zobrazení 3D objektu.

Jak již bylo zmíněno, výstupem bude obraz vektorové grafiky. Přesněji řečeno se bude jednat o soubor, který tento obraz vektorové grafiky reprezentuje ve formátu SVG. Samotná vektorová grafika je jeden ze dvou základních způsobů reprezentace dvojrozměrného obrazu, společně s grafikou rastrovou. Rastrová grafika reprezentuje obraz hodnotami barvy jednotlivých bodů obrazu v určitém rozlišení. Naopak vektorová grafika obraz reprezentuje jakožto soubor geometrických útvarů, které lze zvětšovat a zmenšovat dle potřeby bez újmy na kvalitě obrazu. Formát SVG poté specifikuje určitou syntaxi zápisu těchto geometrických útvarů do souboru, který lze následně použít jako vstup libovolného programu pro vykreslení vektorové grafiky (rendereru SVG) pro získání výsledného obrazu.

3.2 Obecný proces převodu dat z 3D na 2D

Tato podkapitola je krátkým přehledem některých základních kroků prováděných při převodu 3D dat na 2D data. Tento souhrn však není vyčerpávající a kroky nemusí být vždy prováděny přesně v uvedeném pořadí. Následující přibližný popis jednotlivých fází převodu (a problémů s nimi spojených) je ale potřebný pro lepší pochopení kontextu blíže popsaných metod v dalších podkapitolách. Podrobnější popis celého procesu převodu lze najít v podkapitole návrhové části 4.2, která uvádí seznam a pořadí kroků prováděných v rámci tohoto pluginu ze specifičtějšího a praktičtějšího hlediska.

Pro převod 3D dat na 2D, resp. pro zobrazení 3D dat, slouží sekvence kroků, které lze obecně shrnout konceptem zvaným vykreslovací nebo zobrazovací řetězec (graphics pipeline, rendering pipeline). V různých zdrojích lze najít různé definice jednotlivých fází a kroků tohoto procesu, které se často nemusí shodovat v rozdělení, pojmenování a někdy také pořadí některých kroků, především kvůli odlišným úhlům pohledu na problém a jeho zjednodušení. Následující text proto není přesnou definicí vykreslovacího řetězce, ale pouze zjednodušeným shrnutím kroků zobrazování podstatných pro tuto práci, které se v tomto řetězci objevují a jejichž sekvence se mu místy podobá. Tato zjednodušená sekvence byla sestavena na základě popisů zobrazovacího řetězce v literatuře [1, s. 302–304] a [4, s. 14–17].

1. Před začátkem samotného procesu jsou rozmístěny jednotlivé modely jakožto objekty ve 3D prostoru (scéně). Stejně tak má ve scéně danou pozici také kamera, skrze kterou je na scénu nahlíženo. Do obrazu této kamery se jednotlivé plochy modelů ve výsledku promítnou jako dvourozměrné polygony.

2. Jedním z potřebných kroků je převod souřadnic objektů. Souřadnice vrcholů a normál těchto objektů jsou totiž pouze lokální v rámci daného objektu (local space), tento objekt je ovšem umístěn na určitých souřadnicích v rámci prostoru celé scény (world space). Souřadnice je tedy potřeba transformovat (resp. provést maticový součin zobrazovacími maticemi) pro získání jejich skutečné pozice v prostoru.

3. Dále mohou být do procesu zahrnuty kroky, které slouží k eliminaci nebo ořezu takových ploch, které by na výsledném obraze neměly být viditelné. Může se jednat o odvrácené plochy objektů nebo o plochy, které leží mimo záběr kamery. Tyto kroky částečně řeší problém viditelnosti, jehož podstata je popsána na konci této sekce. Samotné kroky jsou blíže popsány v následujících podkapitolách 3.3 a 3.4.

4. Pro jednotlivé zbylé plochy je dále třeba určit jejich barvu, následuje tedy krok stínování. Výsledná barva může záviset na více faktorech, jako je barva materiálu plochy, barva světla, pozice světla, úhel natočení plochy apod. Princip určení výsledné barvy je blíže popsán v podkapitole 3.5.

5. Transformace souřadnic na souřadnice scény nebyl jediný převod koordinát procesu. Další transformace souřadnic, která v rámci převodu probíhá, je převod 3D souřadnic scény do souřadnicového systému výsledného obrazu. Samotná pozice objektů v rámci celé scény nevypovídá o jejich pozici ve výsledném obraze, ta závisí na pozici kamery.

6. Po získání seznamu výsledných polygonů 2D obrazu by většinou následovala část rasterizace, která převádí jednotlivé útvary na výsledné fragmenty rastrového obrazu. Tato část v rámci této práce však není podstatná, neboť výstupem není rastrový obraz ale vektorová grafika. O rasterizaci obrazu při vykreslení této vektorové grafiky se stará libovolný nástroj pro její vykreslení. Cílem této práce je pouze převod na takový soubor, který těmito nástroji může být vykreslen. Namísto fáze rasterizace tak tedy bude proces tohoto převodu obsahovat krok zápisu výsledných polygonů 2D obrazu do souboru. Je také dobré mít na paměti, že tyto polygony jsou následně libovolným nástrojem vykresleny v takovém pořadí, v jakém byly zapsány do souboru (tzn. polygony zapsány v souboru jako první budou překresleny polygony zapsanými později), tudíž je nedílnou součástí tohoto kroku také určení pořadí zápisu jednotlivých polygonů pro korektní popis obrazu. Tato část je podrobně popsána v podkapitole 3.6.

Častým problémem výše popsaného procesu a počítačové grafiky obecně je problém viditelnosti. Ten spočívá v rozhodnutí o tom, které plochy jednotlivých modelů lze na výsledném obraze skutečně vidět z pohledu kamery. S tímto problémem souvisí pojem „hidden-surface determination“ (také zvaný jako například „occlusion culling“ nebo „hidden-surface removal“, přibližně přeložitelný do češtiny jako „rozhodnutí/odstranění skrytých povrchů“), což je proces zabývající se rozpoznáním a popřípadě odstraněním těch ploch, které na výsledném obraze nelze z pohledu kamery vidět [4, s. 1023]. Tato problematika není zcela triviální a jejím řešením se alespoň částečně zabývají algoritmy popsané v podkapitolách 3.3, 3.4 a 3.6.

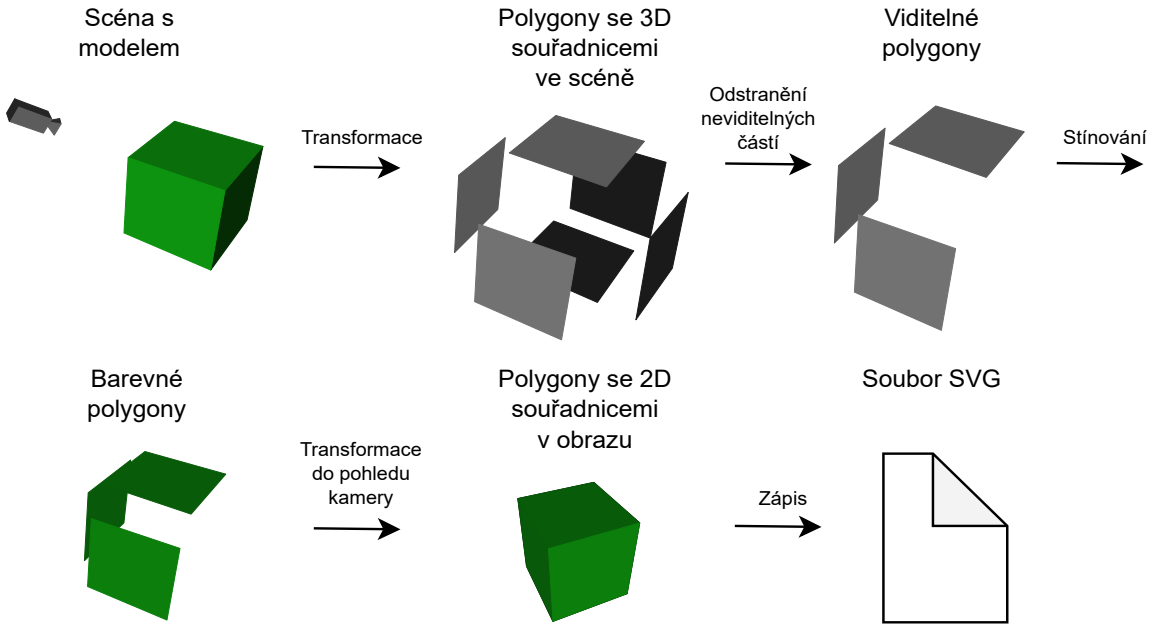
Výše uvedený zjednodušený proces převodu lze shrnout obrázkem 3.1.

3.3 Backface culling

Na úvod bude kvůli jednoduchosti svého principu stručně popsána metoda pro vyřazení odvrácených ploch modelů, Backface culling.

Důvodem použití této metody je odstranění těch ploch modelu, které by neměly být z pohledu kamery viditelné, protože se nachází na odvrácených stranách modelu. Odvrácené plochy samy o sobě nezpůsobují problémy u výsledného obrazu, protože polygony tyto plochy reprezentující budou vykresleny jako první a poté budou překryty polygony přední strany. Každý tímto způsobem „zbytečný“ polygon však bude zapsán v souboru a tím bude soubor zvětšovat. U většiny jednoduchých modelů, které budou mít přibližně stejné množství ploch přední i zadní strany, to znamená až dvojnásobnou velikost souboru oproti minimální potřebné velikosti pro korektní popis obrazu. Kromě velikosti souboru však větší množství ploch také značně zpomaluje převod. Jak bude vysvětleno v části o hloubkovém řazení, samotné řezání a řazení výsledných polygonů je výpočetně náročnější a je potřeba co nejvíce snížit počet polygonů, které je nutno kontrolovat a řadit. Proces je samozřejmě také zpomalen už jen převodem více ploch a zápisem většího množství polygonů do souboru.

Existují však i situace, kdy vyřazení odvrácených ploch není žádoucí, například při použití „dutých“ modelů, u kterých lze zadní strany modelu z pohledu kamery vidět, nebo při použití modelů, které mají průhledné plochy. Z tohoto důvodu bude použití této metody u procesu převodu volitelné.



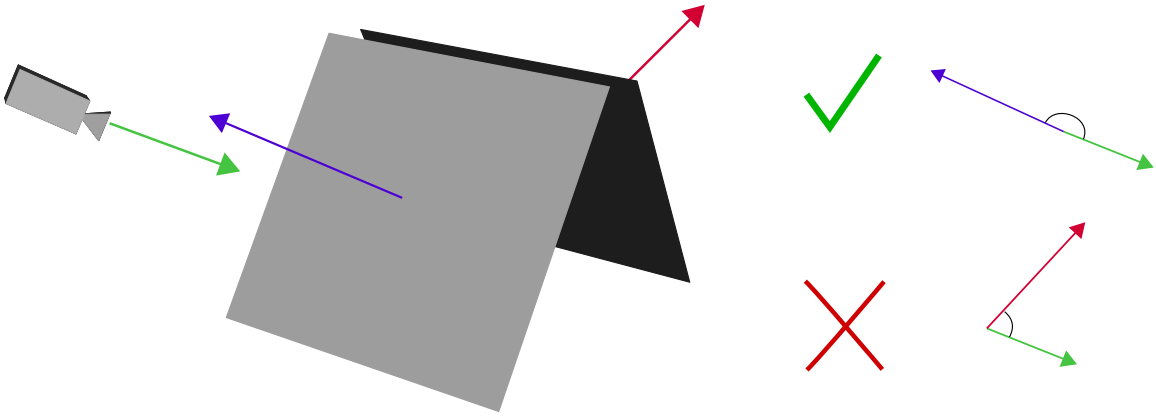
Obrázek 3.1: Shrnutí procesu převodu 3D modelu na 2D; mezery mezi stěnami modelu jsou pouze ilustrační. Na prvním obrázku lze vidět scénu i s pozicí kamery z třetího pohledu. Model této scény je popsán plochami tohoto modelu, které jsou transformovány z lokálních souřadnic na souřadnice scény (druhý obrázek). Poté jsou odstraněny části, které tvoří zadní plochy modelu z pohledu kamery (třetí obrázek). U zbylých ploch je vypočítána jejich barva (čtvrtý obrázek). Poté jsou převedeny do souřadnic pohledu kamery (pátý obrázek), jejíž umístění lze vidět na prvním obrázku. Výsledné polygony jsou zapsány do souboru vektorové grafiky (šestý obrázek).

Nyní už však k principu této metody. Samotné vyřazení odvrácených ploch je triviální – tyto plochy modelu jednoduše nejsou převedeny a zapsány do souboru popisujícího obraz. Problém spočívá v rozpoznání, které plochy lze vlastně za odvrácené považovat. K tomu je zapotřebí poznatků z oboru analytické geometrie.

Každá plocha, která je převáděna, má normálu, resp. normálový vektor v kontextu 3D modelu. To je přímka, která je na tuto plochu kolmá a lze tak říct, že určuje směr natočení plochy. Díky informaci o pozici kamery v prostoru lze určit směr od kamery k ploše nebo k jejím jednotlivým vrcholům jakožto vektor. Pokud vektor od pozice kamery k ploše a normálový vektor plochy jsou otočeny stejným směrem (resp. svírají úhel menší než 90°), znamená to, že plocha je otočena přední stranou od kamery a zadní stranou ke kameře.

Zbývá tedy zajistit výpočet svíraného úhlu. Toho lze docílit skalárním součinem dvojice vektorů, v tomto případě směru od kamery k ploše a normálového vektoru plochy. Pokud je skalární součin dvojice vektorů kladný, znamená to, že svírají úhel menší než 90°. Tím pádem lze plochu označit za odvrácenou a vyřadit ji z výsledného seznamu polygonů, který bude reprezentovat obraz vektorové grafiky [4, s. 1047–1048].

Princip metody Backface culling lze shrnout obrázkem 3.2.



Obrázek 3.2: Princip metody Backface culling. Zelený vektor označuje směr kamery, modré a červené vektory poté normálové vektory obou ploch. V pravé části obrázku je znázorněn úhel, který tyto normálové vektory svírají s vektorem směru kamery. Zelený a modrý vektor svírají úhel větší než 90°, proto plocha s modrým normálovým vektorem zůstane zachována. Naopak plocha s červeným vektorem svírá se směrem kamery úhel menší než 90°, a proto bude vynechána.

3.4 Culling/Clipping polygonů mimo okno

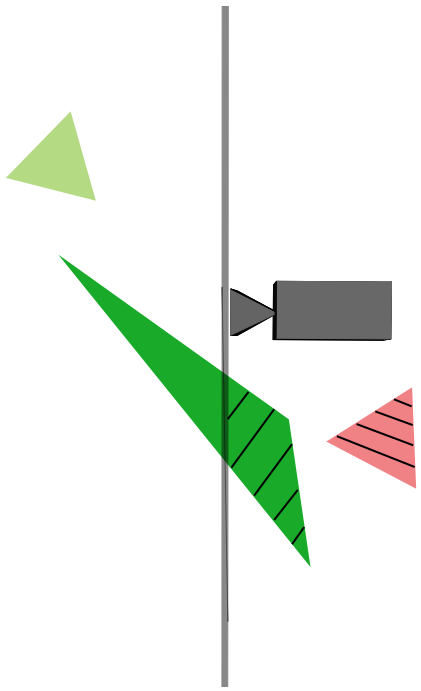
Další o trochu složitější metody, které také slouží k eliminaci těch částí obrazu, které by z pohledu kamery neměly být viditelné, jsou Clipping a Frustum culling. Tyto metody se na rozdíl od metody Backface culling starají o polygony (nebo jejich části), které leží mimo meze okna výsledného obrazu [4, s. 1044–1047].

Podobně jako u metody Backface culling není cílem těchto metod zlepšit kvalitu obrazu. Do souboru ve formátu SVG lze totiž také zapisovat polygony s vrcholy se souřadnicemi ležícími mimo meze okna definované souborem. Při vykreslení tohoto souboru jsou části ležící mimo okno automaticky ořezány a nemají tak vliv na výslednou kvalitu. Přesto však existují důvody, proč se chtít těchto částí zbavit a definovat výsledné polygony přesně (tyto důvody se spíše týkají návrhové části, je však lepší je zde zmínit jakožto podnět pro zavedení těchto metod do procesu převodu).

Prvním důvodem, který se týká ploch kompletně ležících mimo záběr kamery, je podobně jako u metody Backface culling zmenšení velikosti souboru tím, že bude snížen počet zapisovaných polygonů, které by ani ve výsledném obraze nebylo možno vidět. Snížení počtu zpracovávaných polygonů taktéž vede ke zlepšení rychlosti převodu tím, že bude třeba kontrolovat a řadit menší množství prvků. Dalším důvodem je usnadnění dalších úprav souboru ve formátu SVG. Přestože jsou polygony při vykreslení souboru automaticky ořezány na meze okna, jsou v souboru stále uloženy jejich přesné souřadnice. Pokud tedy uživatel chce tento soubor nadále upravovat v jakémkoliv editoru vektorové grafiky, například při přidávání dalších prvků, a bude zapotřebí zvětšit meze okna, budou tyto přečnívající části součástí obrazu, přestože nebyly vidět v původním záběru. To může být považováno za nežádoucí. Samotný dopad na velikost souboru při tomto ořezu nebude brán v potaz, neboť v případě některých polygonů (především trojúhelníků) se může soubor zvětšit a v případě jiných (především polygonů s více než třemi vrcholy) naopak zmenšit. Navíc oproti počtu všech polygonů výsledného obrazu bude počet ořezávaných polygonů (tzn. polygonů na pomezí okna) zanedbatelný.

Culling ploch za kamerou

Proces nejdříve začne vyřazením ploch ležících za kamerou. V tomto kroku ještě převod nepracuje s polygony výsledného 2D obrazu, ale s plochami modelu převedenými do souřadnic scény (world space). Plochy, jejichž vrcholy leží v poloprostoru za kamerou, nelze jednoduše převést do souřadnic výsledného 2D obrazu (nelze je promítnout), jsou proto buďto celé vynechány, pokud všechny jejich vrcholy leží za kamerou, nebo rozděleny rovinou definovanou směrem natočení a pozicí kamery na přední a zadní část, přičemž zadní část je vyřazena z procesu převodu. Lépe je situace vystižena na obrázku 3.3.



Obrázek 3.3: Ořez ploch ležících za kamerou. Světle zelený polygon nezasahuje do poloprostoru za kamerou a tudíž bude vykreslen celý. Červený polygon naopak leží za kamerou a proto bude celý vynechán. Tmavě zelený polygon, jehož pouze část leží za kamerou, bude rozdělen rovinou kamery na dvě části, kde přední (levá) zůstane zachována a zadní (pravá) bude vynechána.

Clipping polygonů ležících mimo okno

Zbývající plochy už je tedy možno všechny převést na výsledné polygony a promítnout do dvojrozměrného pohledu (tzv. viewportu) kamery. Z těchto polygonů je následně třeba eliminovat ty, jejichž všechny vrcholy leží mimo hranice okna a žádná jejich část se nepromítne do výsledného obrazu, a ořezat ty, jejichž pouze některé vrcholy (ale ne všechny) leží mimo viewport. V tomto bodě převodu už jsou známy 2D souřadnice polygonů a mezí okna; rozpoznání polygonů, které je třeba ořezávat, je tedy triviální. Jsou to právě ty, jejichž alespoň 1 vrchol neleží uvnitř mezí okna.

Pro ořezávání polygonů mezemi okna existuje mnoho algoritmů. V rámci této práce byl použit algoritmus Sutherland–Hodgman [7], resp. jeho zjednodušená verze pro ořezávání 2D polygonů mezemi obdélníkového okna. Algoritmus funguje na principu postupného řezání, kdy je polygon ořezáván postupně všemi jednotlivými mezemi. V případě obdélníkového

okna jde tedy o 4 meze, a to minimální hodnoty x, y a maximální hodnoty x, y. Pro každou mez je kontrolován každý vrchol polygonu. První vrchol je zkontrolován zvlášť. Pokud leží uvnitř okna, bude vrcholem výsledného polygonu, v opačném případě nikoliv. Pro každý další vrchol mohou nastat 4 různé situace [1, s. 108–111].

1. Tento vrchol leží v okně, předchozí vrchol také ležel v okně. V tomto případě bude tento vrchol také vrcholem ořezaného polygonu.

2. Tento vrchol neleží v okně, předchozí vrchol ale v okně ležel. V tomto případě bude dalším vrcholem výsledného polygonu průsečík úsečky mezi těmito dvěma vrcholy a mezí.

3. Tento vrchol neleží v okně, předchozí vrchol v něm také neležel. V tomto případě nebude přidán žádný vrchol do výsledného polygonu.

4. Tento vrchol leží v okně, předchozí vrchol v něm ale neležel. V tomto případě budou dalšími vrcholy výsledného polygonu průsečík úsečky mezi těmito dvěma vrcholy a mezí a také tento vrchol samotný.

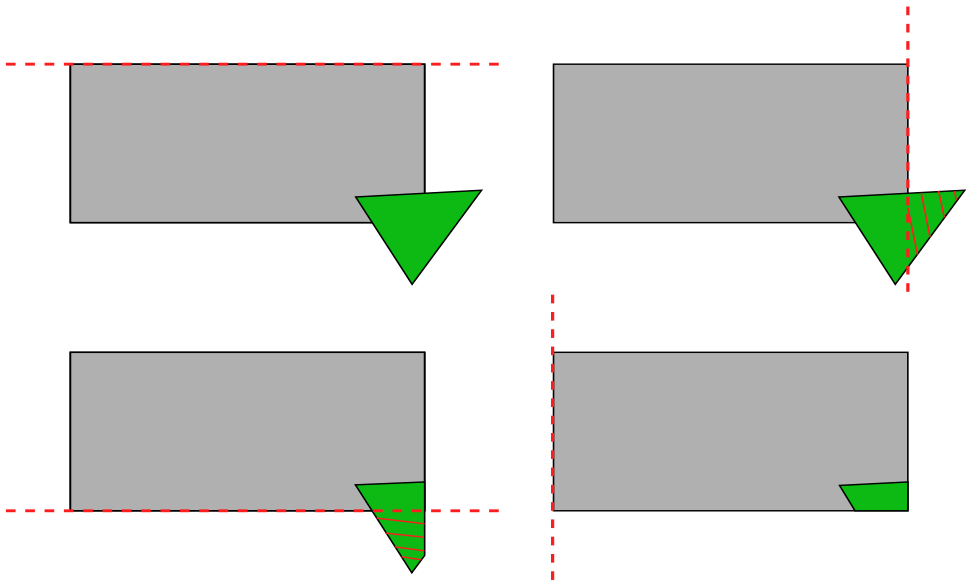
Po ořezání polygonu ze všech stran bude získán výsledný polygon. Nevýhodou tohoto algoritmu je zpracování nekonvexních polygonů, které můžou tímto způsobem ořezu vytvořit dva menší polygony, které jsou ale vzájemně propojeny v jeden. Tato slabina je ale v této práci zanedbána, neboť takové polygony se běžně ve 3D modelech nevyskytují kvůli mnoha problémům, které s jejich vykreslením souvisí. Navíc i v případě, že by některý model takový polygon obsahoval, musel by se nacházet právě na pomezí okna. Dále by musel být natočen specifickým způsobem, aby při jeho ořezu více polygonů vzniklo. I přes všechny tyto „náhody“ by však výsledný artefakt, který by po vykreslení tohoto nepřesného řezu vznikl, měl minimální dopad na celkovou kvalitu obrazu.

Blíže algoritmus specifikuje následující pseudokód 1 a obrázek 3.4 (nereprezentují však popis návrhu/implementace, jedná se pouze o ilustraci pro snadnější pochopení popsaného algoritmu).

Algoritmus 1: Algoritmus Sutherland-Hodgman pro ořez polygonů mezemi okna

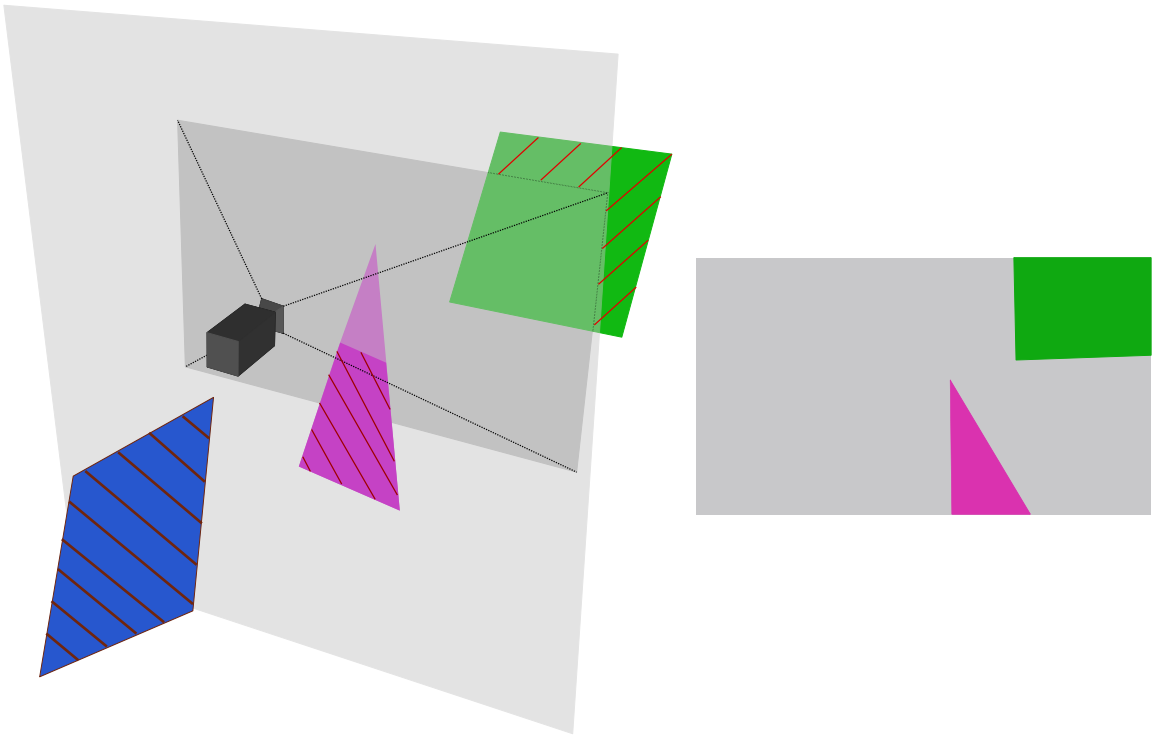
Vstup : Neořezaný polygon P
Výstup: Polygon P' ořezaný mezemi okna

```
1  Vytvoř kopii P' polygonu P;
2  Vytvoř prázdný seznam ořezaných vrcholů Clipped;
3  foreach mez okna do
4      foreach dvojice A, B po sobě jdoucích vrcholů v P' do
5          if A leží v okně then
6              if B také leží v okně then
7                  Přidej B do Clipped;
8              else
9                  Přidej průsečík AB a meze do Clipped;
10             end
11         else
12             if B leží v okně then
13                 Přidej průsečík AB a meze do Clipped;
14                 Přidej B do Clipped;
15             end
16         end
17     end
18     Nahraď seznam vrcholů polygonu P' vrcholy seznamu Clipped;
19     Vyprázdni seznam Clipped;
20 end
21 return ořezaný polygon P'
```



Obrázek 3.4: Sutherland–Hodgman algoritmus pro ořez. V první části je zelený trojúhelník ořezán horní mezí okna, za kterou však nezasahuje, proto se situace nezmění. V druhé části je řezán pravou mezí na menší čtyřúhelník. Ve třetí je tento čtyřúhelník dále ořezán spodní mezí okna. V poslední části polygon nepřesahuje levou mez a proto se situace opět nezmění.

Na závěr lze ještě pro přehlednost uvést obrázek 3.5 shrnující způsob vynechání a ořezu různě konfigurovaných ploch popsaný v této podkapitole.



Obrázek 3.5: Funkce metod pro vynechání ploch a ořezání polygonů. Na levém obrázku lze vidět scénu i s kamerou z třetího pohledu, na pravém poté scénu z pohledu kamery. Modrý polygon je celý vynechán, protože leží za rovinou kamery (znázorněnou průhledným světle šedým čtvercem). Růžový polygon je rozdělen na dvě části, protože rovinu protíná, zadní část je vynechána. Zelený polygon je poté řezán mezemi okna, které přesahuje (okno je znázorněno neprůhledným šedým obdélníkem). Na výsledném obraze tedy bude vidět pouze fragment růžového polygonu a ořezaný zelený polygon.

3.5 Stínování výsledných polygonů

Další důležitou součástí převodu, která už se tentokrát týká především kvality obrazu, je stínování. Stínování, resp. určení barvy jednotlivých polygonů výsledného obrazu, je nedílnou součástí procesu získání 2D obrazu reprezentujícího 3D model. Bez tohoto kroku by měly všechny výsledné polygony stejnou barvu, tudíž by výsledek vypadal pouze jako jednobarevná změť útvarů. Tento proces je prováděn ve fázi, kdy se plochy modelů ještě nachází ve 3D prostoru scény (world space) a kdy ještě nejsou převedeny na 2D polygony pohledu kamery.

Pokud není uvedeno jinak, všechny zmíněné hodnoty barev v této práci uvažují barevný model RGB, ve kterém jsou hodnoty jednotlivých barevných kanálů červené, zelené a modré popsány celými čísly v rozsahu 0 až 255. Jak bude později zmíněno, tento barevný model totiž slouží také k popisu materiálů v Blenderu a k popisu barev ve formátu SVG.

Primitivní určování odstínů šedi

Pro vysvětlení principu metody stínování nebudou v této části uvažovány barvy materiálu jednotlivých ploch a všechny výsledné polygony budou mít takové hodnoty barvy dle modelu RGB, ve kterých všechny 3 složky mají stejnou hodnotu (resp. budou bíle, šedé, nebo černé). Následující metoda stínování byla inspirována principem dříve zmíněné metody Backface culling (viz obrázek 3.3), jsou tedy použity vektory určující směr kamery k ploše, normálové vektory ploch a jejich skalární součiny.

Kamera je zde pro zjednodušení zároveň považována za zdroj světla, výsledný obraz se tedy bude jevit jakoby osvětlený z pozice kamery. Plochy, které jsou natočeny svou přední stranou nejvíce ve směru ke kameře, resp. jejich normálový vektor má opačný směr než vektor směru od kamery k ploše, jsou nejsvětlejší, naopak plochy odvrácenější jsou postupně tmavší a tmavší. Plochy úplně odvrácené, což jsou ty jejichž přední strana nejde z pohledu kamery vidět, jsou buďto vyřazeny volitelně prováděnou metodou Backface culling, nebo ponechány jakožto černé.

Je tedy zřejmé, že k určení stupně šedi je zapotřebí zjistit přesně úhel, resp. kosinus úhlu, který svírají vektor směru kamery a normálový vektor plochy. U metody Backface culling stačilo zjistit znaménko výsledku skalárního součinu těchto vektorů, u této metody je však potřeba využít vzorce platného pro skalární součin:

$$\cos(\alpha) = \frac{\mathbf{a} \cdot \mathbf{b}}{|\mathbf{a}||\mathbf{b}|} \tag{3.1}$$

K získání hledané hodnoty kosinu úhlu tedy stačí vydělit skalární součin dvou vektorů součinem jejich velikostí. Protože se jedná o kosinus úhlu, výsledky tohoto výpočtu leží v intervalu od -1 po 1. Hodnoty menší než 0 představují plochy odvrácené od kamery. Hodnoty vyšší představují osvětlené plochy. Čím vyšší hodnota kosinu, tím více světla by na plochu mělo dopadat. Hodnotu z intervalu 0 až 1 už poté stačí převést na hodnotu z intervalu 0 až 255 (pro všechny složky barvy dle RGB modelu) a tím určit stupeň šedi.

Tento způsob určování stupně šedi se však ve výsledku vyznačuje vysokým kontrastem, neboť jsou hodnoty kosinu převáděny na hodnoty z celého intervalu 0 až 255, tedy od úplně černé barvy až po úplně bílou. To může být nepřirozené a nežádoucí. Proto je tato metoda ještě vylepšena o možnost určit krajní hodnoty převáděného intervalu, tedy omezení extrémních stupňů šedi. Hodnota kosinu z intervalu 0 až 1 tedy není převáděna na interval 0 až 255, ale na nastavitelný interval minimální hodnoty až maximální hodnoty (v_{min} až v_{max}). Výsledný výpočet stupně šedi lze shrnout následovně:

$$brightness = MAX(\cos(\alpha), 0) \cdot (v_{max} - v_{min}) + v_{min} \tag{3.2}$$

Zakomponování barvy materiálu

Způsob výpočtu odstínu v předchozí sekci je použitelný pouze pro získávání černobílých obrázků. Tyto výsledky přesto mohou být v některých případech žádoucí a i přes primitivnost metody vypadají poměrně kvalitně. Proto bude možnost tohoto způsobu výpočtu ve výsledném pluginu nastavitelná. Je třeba však také představit vylepšenou metodu pro takový výpočet barvy, který zohlední i barvu modelů, popřípadě barvu světla.

Jako první vylepšení lze však zavrhnout zjednodušení předchozí metody týkající se kamery jakožto zdroje světla a zakomponovat nastavitelné světlo scény, a to jak jeho pozici, tak jeho barvu. To znamená, že jedním z vektorů pro výpočet úhlu natočení plochy už není

striktně směr od kamery k ploše, ale směr od bodového zdroje světla k ploše, způsob výše popsaného výpočtu to však příliš nemění.

Pro další vylepšení je potřeba shrnout některé základy stínování. Tím prvním je otázka, pro jak velké části modelu vlastně barvu počítat. Mezi základní typy stínování patří Flat shading (ploché stínování, kdy je barva počítána zvlášť pro každou plochu/polygon), Gouraud shading (Gouraudovo stínování, kdy je barva počítána pro každý vrchol a následně interpolována přes celou plochu) a Phong shading (Phongovo stínování, kde nejsou přes celou plochu interpolovány barvy, ale normály vrcholů a barva je následně počítána pro každý fragment obrazu) [1, s. 341–342][9].

Ačkoliv poslední dva typy vedou k mnohem kvalitnějším obrázkům s plynulými přechody barev, je potřeba v této teoretické části trochu myslet i na implementaci. Uložení těchto obrázků do vektorové grafiky ve formátu SVG je poměrně problematické, vzhledem k tomu, že tento formát neumožňuje definovat barvy vrcholů polygonu, které by následně byly interpolovány přes celou plochu, ale pouze barvy polygonů samotných. Určité interpolace lze ve formátu SVG docílit prvky typu gradient, ty však umožňují pouze plynulý přechod mezi 2 barvami. Pomocí jejich kombinací sice lze docílit popsání trojúhelníku s interpolací 3 různých barev z jeho vrcholů, jedná se však o náročnou a pracnou ruční činnost už jen pro 1 samotný trojúhelník, natož tak pro celou scénu tvořenou více mnohoúhelníky. Z toho důvodu bude dále popisovaná metoda uvažovat typ stínování Flat shading, který je pro vektorovou grafiku nejvhodnější.

Dalším problémem, který zbývá vyřešit, je způsob výpočtu výsledné barvy. Ten popisují osvětlovací modely. Vzhledem k tomu, že je potřeba počítat barvu pro celé výsledné polygony (Flat shading), je lepší zvolit primitivnější způsob výpočtu barvy. Složitější modely (jako například Phongův) zahrnují například odlesky světla, které lze jen těžko vykreslit na velkých plochách popsaných pouze jednou barvou. V kombinaci s metodou Flat shading se tedy jeví jako ideální využít Lambertův (kosinový) zákon.

Podle tohoto zákona je množství dopadajícího světla tím větší, čím je úhel mezi směrem dopadu a normálovým vektorem plochy menší. Barva povrchu tedy zjednodušeně závisí pouze na difuzní barvě materiálu ($matdif_X$), barvě světla ($light_X$) a úhlu, pod jakým světlo na povrch dopadá, nezáleží tedy například na pozici pozorovatele [1, s. 335][4, s. 125–126][5, s. 161–162]. Opět je nutno předem tentokrát z návrhové části zmínit, že ne každá plocha má definován materiál. Tyto plochy tedy používají zjednodušenou verzi výpočtu, která vlastnosti materiálu nezahrnuje. Princip určování intenzity světla je zde opět stejný, jako u předchozí sekce týkající se určování odstínů šedi, lze se tedy opět odkázat na obrázek 3.3 popisující využití normál a směru od kamery k ploše (nebo v tomto případě také i směru od světla k ploše) k určení odvrácenosti plochy a na rovnici 3.1 sloužící k výpočtu přesného úhlu svíraného znázorněnými vektory. Výpočet výsledné barvy plochy lze tedy popsat následujícími rovnicemi a obrázkem 3.6 [9]. V rovnicích jsou hodnoty složek modelu RGB uvažovány tentokrát jako reálná čísla z intervalu 0 až 1.

Pro jednotlivé barevné složky u ploch bez materiálu platí (barvy těchto ploch jsou počítány tak, jakoby materiál byl bílý, tedy všechny složky difuzní barvy měly hodnotu 1):

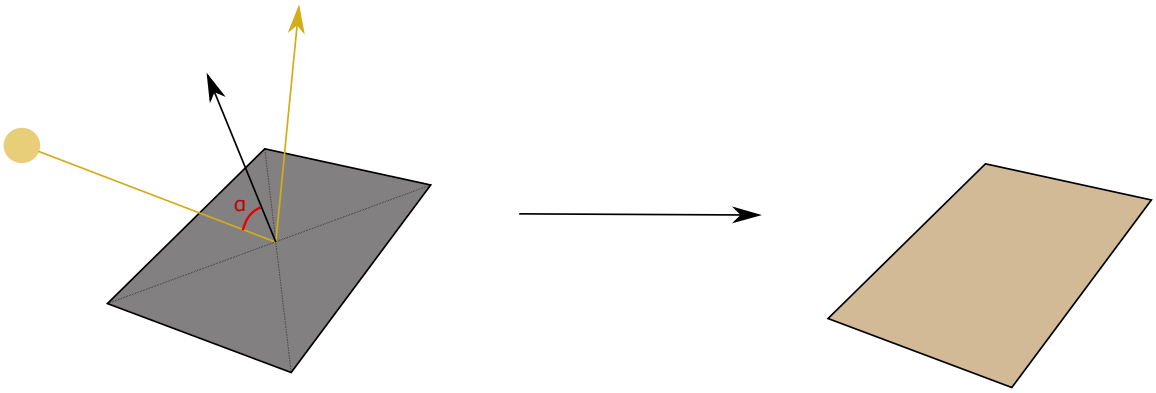
$$R = MAX(\cos(\alpha), 0) \cdot light_R \tag{3.3}$$

$$G = MAX(\cos(\alpha), 0) \cdot light_G \tag{3.4}$$

$$B = MAX(\cos(\alpha), 0) \cdot light_B \tag{3.5}$$

Pro jednotlivé barevné složky u ploch s materiálem platí:

$$R = MAX(\cos(\alpha), 0) \cdot matdif_R \cdot light_R \tag{3.6}$$
$$G = MAX(\cos(\alpha), 0) \cdot matdif_G \cdot light_G \tag{3.7}$$
$$B = MAX(\cos(\alpha), 0) \cdot matdif_B \cdot light_B \tag{3.8}$$



Obrázek 3.6: Princip Flat shading a Lambertova modelu. Oranžové světlo dopadá na šedou plochu z bodového zdroje světla. Množství světla, které na plochu dopadne (a odrazí se), je ovlivněno znázorněným červeným úhlem mezi směrem dopadu světla a normálovým vektorem plochy. Intenzita odraženého světla je v Lambertově modelu nezávislá na úhlu pohledu pozorovatele, k jejímu výpočtu stačí tedy pouze znázorněný úhel a jednotlivé dílčí barvy. Výsledná barva je počítána pouze jednou a je nastavena pro celou plochu dle principu Flat shading.

Okolní světlo

Podobně jako u určování odstínů šedi se výše zmíněný výpočet vyznačuje vysokým kontrastem odvrácených ploch a ploch, které jsou nejvíce osvětleny. To může být v některých případech nežádoucí, neboť ve skutečnosti nebývají strany objektů odvrácené od světla kompletně černé, protože určité (ačkoliv menší) množství světla na ně dopadá ze všech stran. Jako další rozšíření potlačující tento kontrast lze tedy uvažovat možnost zakomponování okolního světla (Ambient light).

Princip tohoto rozšíření je triviální, neboť se jedná o velmi hrubou aproximaci reálných vlastností světla. Pro scénu je nastavena barva okolního světla. K jednotlivým hodnotám barevných složek, získaných výše uvedenými výpočty (rovnice 3.3 a 3.6), jsou jednoduše připočteny hodnoty složek barvy okolního světla ($ambient_X$) vynásobené koeficientem materiálu, který definuje vliv okolního světla na materiál. Pro zjednodušení často bývá tento koeficient pro jednotlivé složky roven hodnotám složek difuzní barvy materiálu [1, s. 335]. Výpočet výsledné barvy lze opět tedy popsat následujícími rovnicemi [9] (v rovnicích jsou hodnoty složek modelu RGB uvažovány opět jako reálná čísla z intervalu 0 až 1).

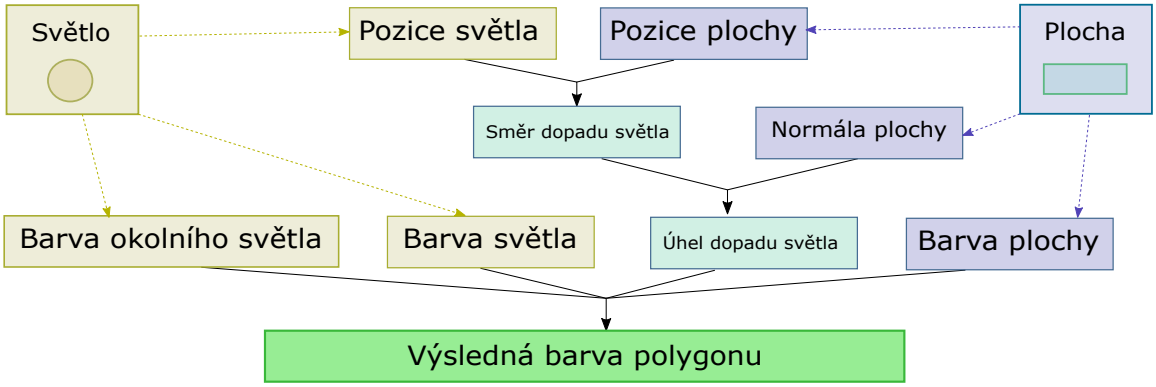
Pro jednotlivé barevné složky u ploch bez materiálu platí (barvy těchto ploch jsou opět počítány tak, jakoby jejich materiál byl bílý):

$$R = MAX(\cos(\alpha), 0) \cdot light_R + ambient_R \tag{3.9}$$
$$G = MAX(\cos(\alpha), 0) \cdot light_G + ambient_G \tag{3.10}$$
$$B = MAX(\cos(\alpha), 0) \cdot light_B + ambient_B \tag{3.11}$$

Pro jednotlivé barevné složky u ploch s materiálem platí:

$$R = MAX(\cos(\alpha), 0) \cdot matdif_R \cdot light_R + matdif_R \cdot ambient_R \tag{3.12}$$
$$G = MAX(\cos(\alpha), 0) \cdot matdif_G \cdot light_G + matdif_G \cdot ambient_G \tag{3.13}$$
$$B = MAX(\cos(\alpha), 0) \cdot matdif_B \cdot light_B + matdif_B \cdot ambient_B \tag{3.14}$$

Výše zmíněnými způsoby výpočtu je možno dosáhnout dostatečně kvalitního způsobu stínování pro vektorovou grafiku. Celý postup výpočtu barvy popsaný v této podkapitole (kromě primitivního určování odstínů šedi) shrnuje obrázek 3.7.



Obrázek 3.7: Postup výpočtu barvy výsledného polygonu. Z pozice světla a pozice plochy je vypočten směr dopadu, který společně s normálovým vektorem plochy určuje úhel dopadu světla. Ten společně s barvou zdroje světla, barvou materiálu plochy a barvou okolního světla určuje barvu celého výsledného polygonu.

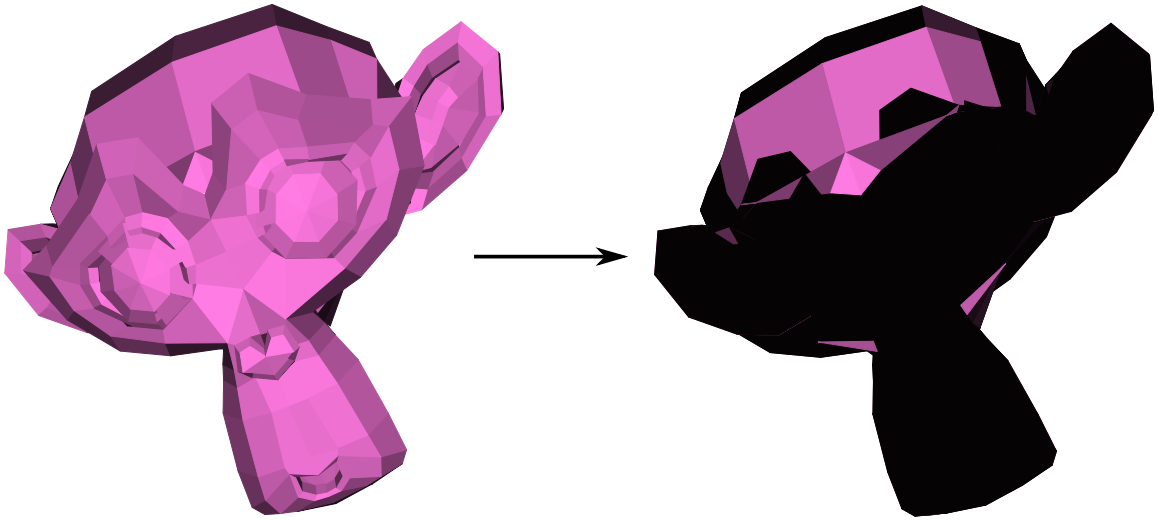
Jako poslední rozšíření procesu lze ještě zmínit možnost použití jiného zdroje světla než bodového, a to plošného. V případě bodového zdroje je směr dopadu světla počítán na základě pozice zdroje a pozice plochy, na kterou světlo dopadá. V případě plošného zdroje je však situace zjednodušena tak, že směr dopadu světla je konstantní pro celou scénu. Pro různé scény mohou být vhodné různé typy zdrojů, proto budou oba tyto typy ve výsledném pluginu podporovány.

3.6 Hloubkové řazení a určování pořadí vykreslení

Poslední a nejsložitější, ale zároveň nejdůležitější částí procesu převodu, probíhající těsně před konečným zápisem do souboru, je hloubkové řazení, které se stalo hlavní překážkou této práce. Před touto částí převodu už proběhly všechny předchozí kroky, jsou zde tedy uvažovány pouze výsledné polygony. Souřadnice vrcholů těchto polygonů už byly také převedeny do souřadnic okna kamery, je však také zachována informace o hloubce těchto vrcholů nutná pro řazení. Také byla už dříve určena jejich výsledná barva.

Zbývá tedy způsob určení takového pořadí, v jakém by měly být jednotlivé polygony zapsány do souboru, aby při jejich vykreslení jakýmkoliv nástrojem pro vykreslování souborů formátu SVG, který vykresluje a překresluje jednotlivé polygony od prvního až k poslednímu, byl získán korektní obraz. Je zřejmé, že při náhodném pořadí vykreslení vznikne špatný výsledný obraz, ve kterém například zadní strana objektu překryje přední, nebo se tyto strany budou náhodně míchat a ve výsledku bude obrazem mix polygonů různých barev, které spolu vůbec nesouvisí a od pohledu netvoří žádnou souvislou plochu. Příklady

takových situací lze vidět na obrázku 3.8 popisujícím nekorektní převod modelu zapříčiněný zanedbáním hloubkového řazení.



Obrázek 3.8: Příklad převodu modelu bez hloubkového řazení. Na levé části lze vidět původně převáděný model (tento obrázek byl pořízen po implementaci primitivního hloubkového řazení) a na pravé výsledek převodu modelu bez hloubkového řazení. Některé zadní části modelu, které jsou kvůli své odvrácenosti od zdroje světla černé, byly náhodně vykresleny později než přední části, což zapříčinilo jejich překrytí a vytvořilo dojem, že část modelu je celá černá.

Je zřejmé, že tyto útvary mají nízkou vypovídající hodnotu, co se týče znázornění původního modelu. Je tedy třeba vyřešit, jakým způsobem rozhodnout pořadí vykreslení.

Metoda č. 1: Primitivní seřazení podle hloubky

Způsob rozhodnutí pořadí vykreslení jednotlivých polygonů (resp. způsob zjištění, které části není třeba vykreslovat) je jedním z nejčastějších problémů počítačové grafiky, k jehož řešení bylo navrženo nemalé množství metod a algoritmů mnoha různých typů. Před podrobnějším rozborem těchto metod a problémů spojených s jejich aplikací při převodu na vektorovou grafiku by však bylo vhodnější nejdříve zmínit ten nejprimitivnější a nejintuitivnější způsob, jakým tento problém řešit, společně s jeho nedostatky. To by mělo poskytnout širší kontext pro lepší pochopení jednotlivých komplikací řazení.

Tato metoda řeší hloubkové řazení poměrně jednoduše, a to doslova řazením hloubky. Přesněji řečeno, jedná se o seřazení výsledných polygonů na základě jejich určité vlastnosti, kterou lze nazvat například hloubka. Tou je jediná hodnota udávající vzdálenost polygonu od kamery na ose Z, tedy ose směřující z obrazu/do obrazu. Metoda funguje na principu tzv. Malířova algoritmu, ve kterém jsou všechny objekty seřazeny podle hloubky a následně vykresleny od nejvzdálenějších po nejbližší. Tento způsob řazení je velice jednoduchý a snadno implementovatelný, zároveň je oproti ostatním metodám zdaleka nejrychlejší a nejméně výpočetně náročný. Tato jednoduchost se ovšem stává i slabinou této metody, neboť se jedná pouze o hrubou aproximaci pořadí vykreslení, která v některých situacích dále zmíněných selhává. Nelze totiž shrnout pozice všech vrcholů a konfiguraci polygonu v prostoru do jediné hodnoty zvané hloubka bez ztráty podstatných informací [1, s. 353–354].

Zbývá tedy vyřešit jediný problém této metody, kterým je samotné určení hloubky polygonu. Lze zvolit různé heuristiky pro odhadnutí hloubky polygonů takovým způsobem, aby výsledné seřazení bylo co nejpřesnější. Jako některé příklady lze uvést následující:

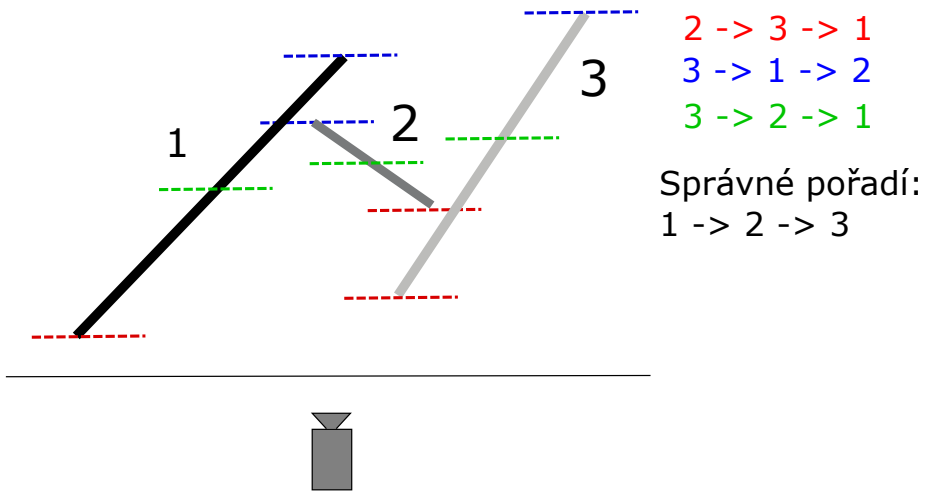
1. Hloubka polygonu je taková, jaká je hodnota souřadnice Z jeho nejbližšího vrcholu (tj. vrcholu nejméně vzdáleného od kamery na ose Z).

2. Hloubka polygonu je taková, jaká je hodnota souřadnice Z jeho nejvzdálenějšího vrcholu (tj. vrcholu nejvíce vzdáleného od kamery na ose Z).

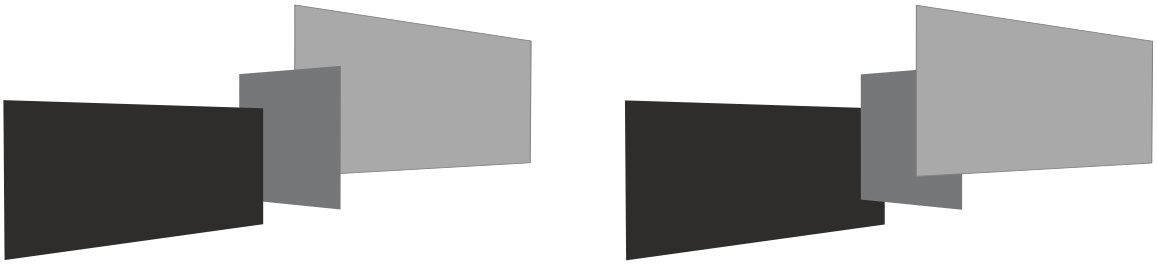
3. Hloubka polygonu je taková, jaká je hodnota souřadnice Z středu jeho bounding boxu (ohraničujícího kvádru, dále bude používán termín bounding box), což je nejmenší možný útvar s co nejmenším objemem v prostoru ve tvaru kvádru, ve kterém leží, resp. do kterého se vlezou, všechny vrcholy polygonu. Kvůli dále popsaným metodám jsou v každém polygonu uloženy rozměry jeho bounding boxu. Střed tohoto bounding boxu lze využít jako dobrý odhad hloubky samotného polygonu. Alternativně bez použití bounding boxu stačí určit minimální a maximální hodnotu souřadnic na ose Z v rámci všech vrcholů polygonu a vypočítat jejich střed.

4. Hloubka polygonu je taková, jaká je hodnota průměru hodnot souřadnic na ose Z všech jeho vrcholů. Opět je tedy určován střed polygonu, tentokrát se však jedná o vážený střed, což znamená, že více vrcholů na jedné straně posune střed směrem k sobě.

Přestože se myšlenka této metody může zdát na první pohled korektní, existuje nemalé množství poměrně častých konfigurací polygonů, při kterých metoda za použití jakékoliv výše zmíněné heuristiky provede nekorektní řazení. Jako primitivní ukázku lze použít obrázky 3.9 a 3.10.

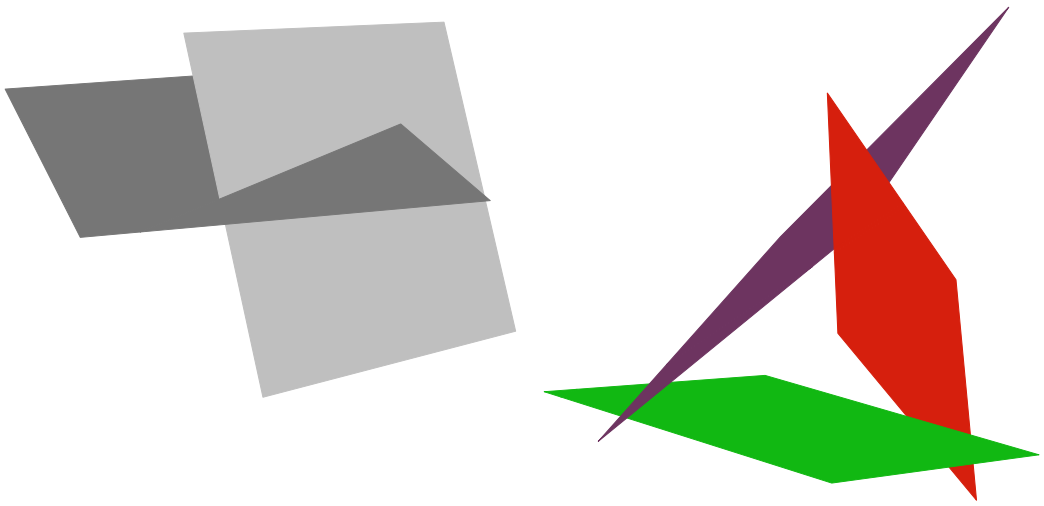


Obrázek 3.9: Primitivní řazení, značky specifikují vzdálenosti použité pro řazení při jednotlivých heuristikách. V případě heuristiky určování hloubky podle nejbližšího vrcholu ke kameře (červené značky) by byly polygony vykresleny v pořadí 2-3-1. V případě nejvzdálenějšího vrcholu (modré značky) v pořadí 3-1-2. V případě středu polygonů (i váženého středu, zelené značky) by výsledné pořadí bylo 3-2-1. Správně pořadí pro korektní obraz je však ve skutečnosti 1-2-3.



Obrázek 3.10: Nekorektní vykreslení scény předchozího obrázku (vlevo, pořadí 3-2-1) a korektní vykreslení scény (vpravo, pořadí 1-2-3).

Tento typ konfigurace však není jediný problém, který tato metoda nedokáže vyřešit. Tím dalším jsou konfigurace neseřaditelných polygonů. To jsou takové scény, jejichž polygony jsou vzájemně v takových pozicích, při kterých neexistuje žádné takové pořadí vykreslení jednotlivých polygonů, jehož výsledkem by byl korektní obraz. Jedná se o vzájemně zaklíněné polygony a o cyklicky se překrývající polygony viditelné na obrázku 3.11.



Obrázek 3.11: Vzájemně neseřaditelné polygony. V levé části lze vidět zaklíněné polygony, u kterých nelze říct, že jeden by měl být vykreslen před druhým. V pravé části lze vidět polygony, které se cyklicky překrývají a ačkoliv lze určit pořadí vykreslení jednotlivých dvojic, neexistuje pořadí korektního vykreslení celé trojice.

Na závěr popisu této metody lze říct, že existuje mnoho běžných konfigurací a poloh polygonů, které metoda nezvládne přesně seřadit. Výhodou této metody je však její jednoduchost a rychlost, zároveň je stále schopna korektně řadit polygony především u jednotlivých modelů s pravidelnou strukturou, ideálně tvořenou větším množstvím menších polygonů, kde méně často dochází k problematickým situacím. (Proto bude použita jako nastavitelný způsob řazení ve výsledném pluginu, společně s volbou jednotlivých heuristik.)

Zrekapitulování problematiky řazení

Předchozí sekce trochu lépe přiblížila problémy spojené s hloubkovým řazením, a tak ještě před popisem další metody bude vhodnější shrnout nejčastější způsoby řešení tohoto problému v počítačové grafice a zkusit dále postupovat jejich aplikací. Obecně tyto algoritmy

řeší problém zvaný Hidden Surface Determination (rozhodování o tom, které části scény nejsou vidět, lze však najít různé termíny pro tuto problematiku) [4, s. 1023].

Jedním z nejčastějších způsobů řešení je metoda Z-buffer. Tato metoda uchovává pro každý pixel obrazu kromě barvy také jeho hloubku. Dále není třeba s popisem pokračovat, neboť už z velmi obecného popisu metody je evidentní, že získaný obraz má přesnost pouze „na pixel“. Vzhledem k tomu, že výstupem tohoto převodu je vektorová grafika, není tato přesnost dostatečná.

Ze stejného důvodu lze zavrhnout metodu Ray Casting. Ta funguje na principu vyslání paprsku pro každý pixel obrazu a vybarvení tohoto pixelu barvou toho objektu, který protne nejdříve.

Tímto způsobem lze také vyřadit všechny ostatní algoritmy, které pracují s přesností odpovídající rozlišení obrazu, jako například Warnockův algoritmus. Obecně tyto algoritmy patří do tzv. třídy Image Space, protože pracují s přesností danou rozlišením samotného výsledného obrazu a jsou použitelné pro převod na rastrovou grafiku [1, s. 350]. Vyznačují se také tím, že jsou obvykle principiálně jednodušší a často v praxi používané.

Důležitější jsou pro tuto práci algoritmy tzv. třídy Object Space, které nepracují s jednotlivými pixely obrazu, ale s objekty jakožto geometrickými útvary v prostoru scény. Celková přesnost obrazu tedy není dána rozlišením výstupního zařízení, ale přesností počítače provádějícího výpočet. Tyto algoritmy jsou obvykle náročnější, a to jak výpočetně, tak implementačně, proto nejsou v praxi pro řazení příliš využívány oproti třídě Image Space. Mezi ty nejjednodušší lze zařadit například dříve zmíněný Backface culling, který byl použit pro vyřazení odvrácených ploch během převodu. Ačkoliv tato metoda neřeší hloubkové řazení, pracuje s polygony na úrovni geometrických útvarů.

Některé taxonomie algoritmů řešících viditelnost ale definují také hybridní třetí skupinu, a to tzv. třídu algoritmů List-Priority, která leží na pomezí dvou předchozích skupin. Ty pracují s geometrickými útvary při sestavování seznamu polygonů, čímž se podobají metodám třídy Object Space. Tento seznam řadí takovým způsobem, že polygony uložené dříve v seznamu nemohou být překryty polygony uloženými v seznamu za nimi. Výsledný obraz je však získán až postupným vykreslením (a rasterizací) tohoto seznamu v pořadí od posledního polygonu po první. Protože je korektní obraz získán až po rasterizaci, podobají se tyto metody také metodám třídy Image Space [2][8, s. 23–25].

Při tomto rozdělení na fáze řazení a vykreslení lze vidět jisté podobnosti s formátem SVG, ve kterém jsou vykreslovány polygony v pořadí od prvního zapsaného až po poslední. Fázi řazení tedy představuje zápis polygonů do souboru v daném pořadí a fázi vykreslení představuje vykreslení tohoto souboru libovolným nástrojem.

Všechny dále popsané metody této podkapitoly lze tedy zařadit právě do třídy algoritmů List-Priority, neboť jejich principem je seřazení polygonů jakožto geometrických útvarů a poté jejich rasterizace v tomto pořadí. O samotné vykreslení se však stará libovolný nástroj použitý pro vykreslení souboru ve formátu SVG, tato druhá fáze tedy není pro tuto práci relevantní. Pod pojmem „vykreslení polygonu“ v těchto částech textu tedy bude myšlen zápis polygonu do souboru.

Metoda č. 2: Binary Space Partition

Tato metoda ukládá jednotlivé řazené polygony do stromové struktury zvané Binary Space Partition Tree (přeložitelné jako strom binárního rozdělení prostoru, dále bude používán výraz BSP strom). Metoda pracuje na principu postupného rozdělování prostoru na poloviny a přiřazování polygonů k jednotlivým poloprostorům na základě toho, ve kterém

poloprostoru se nachází. Z tohoto popisu lze pochopit, že metoda nepracuje s pixely obrazu, ale přímo s polygony v prostoru, proto se pro řešení problému hloubkového řazení v této práci jeví jako ideální. Popis metody v této sekci čerpá z literatury [1, s. 356–358][3].

Nejdříve je důležité vyjasnit, co je vlastně BSP strom. To je takový strom, pro který platí, že každý rodičovský uzel má právě dva syny (levého a pravého, popřípadě přesněji předního a zadního), jedná se tedy už dle názvu o strom binární. Dále platí, že jednotlivé uzly tohoto stromu reprezentují samotné polygony v prostoru, a to 1 nebo i více pro každý uzel. Důležitější je však pravidlo, že v levém, resp. předním, podstromu každého rodičovského uzlu se nachází pouze uzly s takovými polygony, které leží z pohledu kamery před polygonem rodičovského uzlu a mají tedy být vykresleny později než polygon rodičovského uzlu. Stejně tak platí, že v pravém, resp. zadním, podstromu každého rodiče se nachází pouze uzly s polygony, které leží z pohledu kamery za polygonem rodiče a mají tedy být vykresleny dříve než polygon rodičovského uzlu. Samotný princip řazení pomocí této datové struktury lze shrnout do 2 částí.

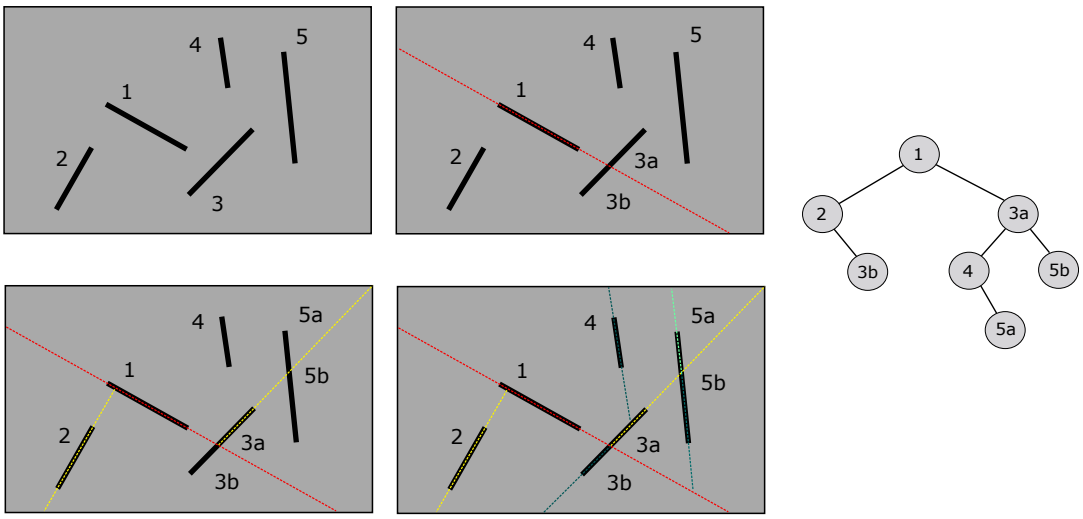
Prvním ze dvou kroků samotného procesu je sestrojení BSP stromu pro popis rozložení polygonů ve scéně, což je poměrně přímočaré. Jako první je nutno zvolit kořenový polygon, který scénu rozdělí na 2 poloviny, vyjmout ho ze seznamu a označit za kořen stromu. Volba tohoto polygonu může značně ovlivnit vyváženost výsledného stromu, proto existují různě složité heuristiky pro výběr počátečního polygonu. Pro jednoduchost lze například zvolit prostřední polygon seznamu všech polygonů. Rovina, ve které tento polygon leží, rozděluje celý prostor na 2 poloprostory, které budou reprezentovány synovskými uzly kořenového uzlu. O každém polygonu ze zbylého seznamu polygonů lze rozhodnout jeho relativní pozici vůči rovině rozdělujícího polygonu. Pokud leží před touto rovinou, je uložen do levého (resp. předního) syna, pokud leží za touto rovinou, je uložen tentokrát do pravého (resp. zadního) syna. Další dva případy, které mohou nastat, jsou, že polygon leží ve stejné rovině, nebo rovinu protíná. V prvním případě lze polygon uložit do stejného rodičovského uzlu jako původní kořenový polygon. V tom druhém je však třeba polygon rovinou „rozřezat“ na dva menší fragmenty, přední a zadní. Ty poté budou uloženy do příslušných synovských uzlů. Ve všech případech je testovaný polygon před uložením do stromu vyjmut z původního seznamu.

Po provedení tohoto kroku pro kořenový uzel je dále prostor rekurzivně dělen. Pro každý synovský uzel rekurzivně probíhá stejný proces určení rozdělujícího polygonu a následné rozdělení všech zbylých polygonů v seznamu tohoto syna na přední a zadní a jejich přidělení do dalších nových synovských uzlů hlouběji ve stromu. Toto rozdělování probíhá až do chvíle, kdy nové uzly mají v seznamu pouze 1 polygon a nelze už dále dělit. Lepší představu o této metodě konstrukce BSP stromu může poskytnout následující algoritmus 2 a obrázek 3.12 (nereprezentují však popis návrhu/implementace, jedná se pouze o ilustraci pro snadnější pochopení popsaného algoritmu).

Algoritmus 2: Postup tvorby BSP stromu

Vstup : Seznam polygonů L
Výstup: BSP strom

```
1  Vyber polygon P a odeber ho ze seznamu L;
2  Vytvoř uzel N BSP stromu a přidej do něj P;
3  foreach polygon Q v seznamu L do
4  |  if Q leží před P then
5  |  |  přidej Q do seznamu předních polygonů uzlu N;
6  |  else if Q leží za P then
7  |  |  přidej Q do seznamu zadních polygonů uzlu N;
8  |  else if Q protíná rovinu P then
9  |  |  rozděl Q podle roviny P na zadní a přední fragmenty a přidej do patřičných
   |  |   seznamů uzlu N;
10 |  else if Q leží v rovině P then
11 |  |  přidej Q do původního uzlu N
12 end
13 if seznam předních polygonů uzlu N není prázdný then
14 |  Rekurzivně opakuj proces pro seznam předních polygonů uzlu N (nový uzel
   |   bude syn uzlu N);
15 end
16 if seznam zadních polygonů uzlu N není prázdný then
17 |  Rekurzivně opakuj proces pro seznam zadních polygonů uzlu N (nový uzel bude
   |   syn uzlu N);
18 end
19 return kořenový uzel N BSP stromu
```



Obrázek 3.12: Princip tvorby BSP stromu (vlevo pohled na scénu shora, vpravo pohled na strukturu BSP stromu). Nejdříve je vybrán polygon 1 jakožto dělicí polygon (druhý obrázek) a tedy jako kořen BSP stromu. Ten rozdělí scénu na dvě poloviny, jedna obsahuje polygon 2 a část b polygonu 3, druhá obsahuje polygony 4 a 5 a část a polygonu 3. V každé polovině je vybrán další dělicí polygon. Tímto způsobem je scéna postupně dělena na poloviny, které jsou reprezentovány podstromy BSP stromu.

Druhým krokem je určení pořadí vykreslení tohoto stromu. Obecně mají BSP stromy tu výhodu, že jimi lze jednou popsat scénu a tu následně vykreslit z libovolného pohledu. Výsledný obraz je totiž vykreslován postupným průchodem stromu, který závisí na pozici samotné kamery vůči uzlům stromu. Proces vykreslení probíhá následovně. Pokud je současný uzel listový, jsou jeho polygony vykresleny. Pokud je nelistový a kamera se nachází v prostoru před jeho polygony, je rekurzivně vykreslen zadní podstrom, poté polygony současného uzlu a poté rekurzivně vykreslen přední podstrom. Pokud se naopak kamera nachází v prostoru za polygony tohoto uzlu, je rekurzivně vykreslen nejdříve přední podstrom, poté polygony současného uzlu a poté rekurzivně vykreslen zadní podstrom.

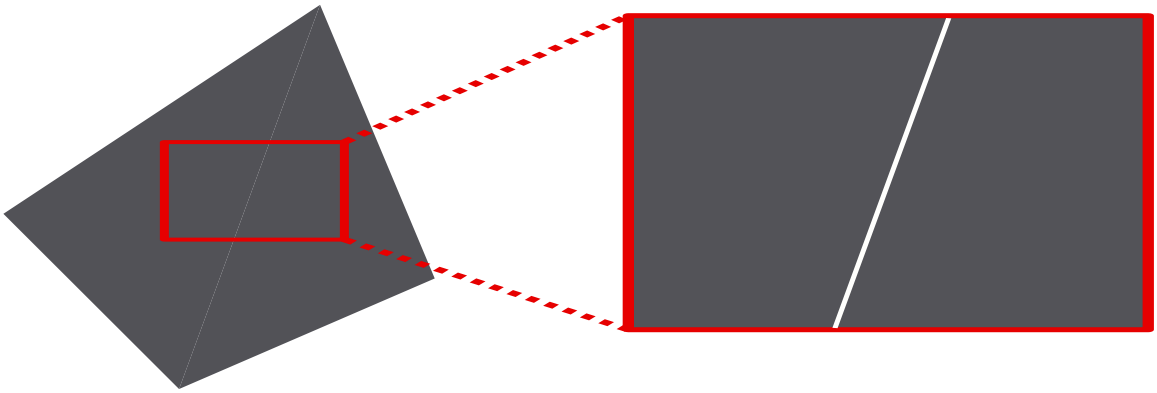
Zde je však potřeba dopředu zmínit některé nedostatky této metody zjištěné při implementaci. Celkově má metoda BSP několik nevýhod při její aplikaci pro tento plugin. Tím prvním je časová náročnost vytvoření samotného BSP stromu. Hlavní výhodou metody BSP je možnost předem pro scénu vygenerovat BSP strom (což je výpočetně náročná část) a poté už jen strom z různých pohledů kamery vykreslovat. V případě tohoto pluginu je však strom generován pro daný pohled z kamery a je tedy nutno ho generovat vždy, když se změní pozice kamery, tedy většinou při každém novém převodu na vektorovou grafiku. Existuje sice alternativa strom generovat pro Blender scénu před převodem do souřadnic pohledu kamery, čemuž by bylo třeba přizpůsobit návrh celého procesu převodu, to by však neměnilo nic na situaci, kdy uživatel mění pozice objektů při snaze získat požadovaný obraz.

Další nevýhodou je paměťová náročnost, neboť ve výsledném stromu je třeba vytvořit uzel nejen pro každý polygon scény, ale také pro každý fragment, který vznikne řezáním polygonů. To pro scény s tisíci polygonů znamená velmi rozsáhlý binární strom.

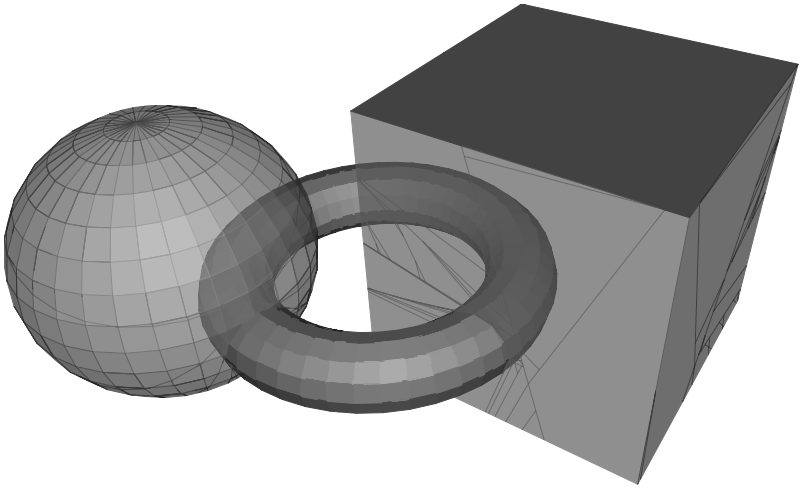
Obě výše zmíněné nevýhody jsou však zanedbatelné v porovnání s hlavním problémem použití metody BSP v tomto pluginu. Největší problém spočívá v kombinaci této metody se zápisem výsledného obrazu do souboru ve formátu SVG. Jak již plyne z popisu samotné metody, při konstrukci BSP stromu je provedeno nemalé množství řezů polygonů, neboť každý nově zvolený polygon rozděluje celou scénu na dva poloprostory a jeho rovina rozřezává všechny protínající polygony. To vede k mnoha zbytečným řezům, které způsobí, že souvislá viditelná plocha modelu není v souboru popsána jediným polygonem, ale například třiceti fragmenty tohoto polygonu. To nemusí vadit kupříkladu při převodu na rastrovou grafiku. Při vykreslování souboru ve formátu SVG se však sousední polygony, přestože sdílí souřadnice některých vrcholů, vykreslí s viditelnou mezerou mezi sebou. Příklad takové situace lze vidět na obrázku 3.13 reprezentujícím vykreslený soubor, v němž jsou zapsány dva sousední polygony se stejnými souřadnicemi dvou svých vrcholů.

Tento vizuální nedostatek lze do určité míry potlačit nastavením širších okrajů polygonů přímo v souboru. Takové řešení však vede ke vzniku artefaktů u menších polygonů výsledného obrazu a vrcholů jednotlivých modelů a především k velmi nekvalitnímu převodu průhledných modelů. Problém použití této metody lze vidět na vykreslení souboru SVG s průhlednými modely převedenými metodou řazení pomocí BSP stromu na obrázku 3.14.

I přes všechny nedostatky je však nutno dodat, že metoda řazení pomocí BSP stromu je schopna korektně seřadit výsledné polygony i u zaklíněných objektů. Přestože z hlediska počtu řezů není ideální a v mnoha scénách způsobuje vznik nemalého množství vizuálních artefaktů, může v některých specifických situacích přinášet kvalitní výsledky. Proto bude výsledný plugin BSP metodu zahrnovat jakožto nastavitelný způsob hloubkového řazení. Bude však také potřeba brát ohled na paměťovou a časovou náročnost vytváření BSP stromu, který může být ve scénách s mnoha polygony značně rozsáhlý, obzvlášť pokud řezáním vzniká mnoho velmi malých fragmentů polygonů. Toho lze docílit například nastavením maximálního počtu provedených dělení podprostorů.



Obrázek 3.13: Nepřesný způsob vykreslování souborů ve formátu SVG. Oba dva polygony v levé části mají ve zdrojovém souboru, ze kterého byly vykresleny, totožné souřadnice dvou sdílených vrcholů (a nulovou šířku okrajů). Lze i přesto zaznamenat viditelnou mezeru a obraz se nejeví jako ucelená plocha.



Obrázek 3.14: Příklad modelů převedených metodou BSP (s aktivním Backface culling) s nastavenou nenulovou šířkou okrajů. Na průhledné krychli a kouli lze vidět části, kde se jednotlivé průhledné fragmenty překrývají kvůli svým širším okrajům a plochy se tak nejeví jako ucelené.

Shrnutí dosavadních komplikací řazení

Po dalším pokusu lze shrnout předešlé a z implementace nově zjištěné problémy následovně. Je potřeba použít takovou metodu řazení, pro kterou platí následující podmínky.

1. Výstupem je soubor vektorové grafiky, metoda tedy neoperuje nad jednotlivými pixely obrazu, ale je schopna řadit polygony jakožto geometrické objekty v prostoru. Princip a implementace takových metod obvykle bývá složitější.

2. Metoda provádí minimální počet řezů při vypořádávání se s kolizemi a při řazení, neboť každý řez lze na výsledném obraze vykresleného souboru ve formátu SVG vidět. Tato podmínka vylučuje nejlepšího kandidáta – řazení pomocí BSP stromu.

3. Metoda je schopna řadit scénu s pár stovkami až tisíci polygonů v rozumném čase. Z dříve zmíněných metod už začínal být například problém u relativně dlouhého sestavování BSP stromu, které už pro pár tisíc polygonů mohlo trvat několik sekund.

Kombinace těchto podmínek vylučuje použití většiny algoritmů a metod řešících určování pořadí vykreslení polygonů, neboť se problém stává poměrně komplikovaným, a to obzvláště v situacích, kdy scéna obsahuje polygony v neseřaditelných konfiguracích (tedy cyklicky překrývajících se polygonů a kolizních polygonů).

Je dobré se tedy zaměřit na alespoň částečné zjednodušení této problematiky. Tímto krokem je myšleno rozdělení celé části hloubkového řazení na dva podproblémy (resp. na dvě fáze), řezání konfliktních polygonů a řazení bezkonfliktních polygonů. Jak už názvy napovídají, první fáze se postará o to, aby ve výsledném řazeném seznamu nebyly žádné polygony v konfliktu a tím pádem se jednalo o seřaditelný seznam. Toto rozdělení sice pro druhou fázi neeliminuje žádnou z výše zmíněných podmínek, může však pohled na problém zjednodušit. Kombinace první fáze řešení konfliktů společně s první metodou primitivního hloubkového řazení by navíc mohla vést k lepším výsledkům.

Detekce a eliminace konfliktů

Podproblém řezání konfliktních polygonů lze intuitivně rozložit na dvě části, detekce konfliktů a jejich odstranění. Algoritmus řezání tedy postupně prochází vybrané dvojice polygonů a ověřuje, zdali jsou vzájemně seřaditelné (resp. zdali je možno říct, že jeden leží před druhým nebo opačně). Pokud ne, je jeden polygon z této dvojice rozřezán na dva fragmenty rovinou polygonu druhého.

K ověření napomáhá Newellův algoritmus [6]. Tato metoda také řeší problém řazení jednotlivých polygonů na úrovni geometrických objektů, proto bude celá blíže popsána v pozdější sekci. Prozatím si tato sekce postačí s podmínkami, které Newellův algoritmus testuje pro rozpoznání, zdali je nutné ve dvojici polygonů provést řezání. Podle tohoto algoritmu není třeba polygony P a Q řezat a lze je seřadit, pokud platí následující podmínky.

1. Intervaly minimálních a maximálních hodnot souřadnic na ose Z polygonů P a Q se nepřekrývají (resp. tzv. z-extent polygonů P a Q se nepřekrývá).
2. Intervaly minimálních a maximálních hodnot souřadnic na ose X polygonů P a Q se nepřekrývají (resp. tzv. x-extent polygonů P a Q se nepřekrývá).
3. Intervaly minimálních a maximálních hodnot souřadnic na ose Y polygonů P a Q se nepřekrývají (resp. tzv. y-extent polygonů P a Q se nepřekrývá).
4. Všechny vrcholy P leží za rovinou, na které se nachází Q.
5. Všechny vrcholy Q leží před rovinou, na které se nachází P.
6. Zobrazení polygonů P a Q ve výsledném obraze se nikde nepřekrývají.

Tyto podmínky jsou testovány postupně od první po poslední. Jakmile kterákoliv z nich platí, je testování ukončeno a polygony jsou považovány za seřaditelné. Pokud neplatí žádná z nich, jsou považovány za konfliktní. Původní algoritmus využívá tyto podmínky také k tomu, aby zjistil, jaké je pořadí polygonů v této dvojici. Pro tuto situaci, kdy jsou pouze hledány konflikty, lze upravit podmínky 4 a 5 na „P neprotíná rovinu, na které se nachází Q“ a „Q neprotíná rovinu, na které se nachází P“, aby bylo zamezeno nutnosti provádět toto

testování dvakrát s opačným pořadím P a Q a celkově tak byla detekce konfliktů značně urychlena.

Jak už bylo výše zmíněno, po detekci konfliktu je konflikt vyřešen tak, že jeden polygon je řezán na dva fragmenty rovinou druhého polygonu. Tyto fragmenty jsou poté přidány zpět do seznamu polygonů, protože i ty samy o sobě mohou být v konfliktu s jiným dalším polygonem.

Metoda č. 3A: Detekce kolizí v seznamu

Detekce konfliktu dvou polygonů je však pouze samostatný nástroj, který je třeba uplatnit v širším kontextu. Hledání konfliktu v každé možné kombinaci všech polygonů ve scéně se už při stovkách nebo tisících polygonů jeví jako nepříliš ideální řešení z hlediska časové náročnosti.

Jako určitou optimalizaci lze použít seznam výsledných polygonů, který je seřazen podle maximálních hodnot souřadnic na ose Z jednotlivých polygonů (resp. hodnoty z_{max} jejich bounding boxu). Dále lze z testovaných podmínek vyřadit první test, který se týká testování intervalů na ose Z. Seznam je totiž procházen od konce a pro každý polygon P jsou uvažovány pouze takové polygony Q, jejichž maximální hodnota souřadnice na ose Z je vyšší než minimální hodnota u P. To je v podstatě náhrada prvního vynechaného testu, která má tu výhodu, že jakmile v seřazeném seznamu v průchodu od konce poprvé k překrytí na ose Z nedojde, není třeba testovat zbylé polygony, protože všechny jejich maximální hodnoty souřadnic na ose Z jsou menší. To zásadně sníží počet testovaných dvojic.

Algoritmus tedy funguje tak, že je na začátku zvolen poslední polygon seznamu jakožto polygon P. Následně jsou od konce procházeny a testovány všechny polygony Q, které překrývají P na ose Z. Při nalezení konfliktu je Q řezáno na dva fragmenty rovinou P, tyto fragmenty jsou zpět umístěny do seznamu na příslušná místa. Jakmile už není žádný polygon v konfliktu s P, je zvolen jako nový polygon P polygon sousedící s předchozím P v seznamu zleva (tedy polygon na indexu o 1 nižší) a proces je opakován, dokud není zvolen jako polygon P první, resp. nejlevější polygon seznamu.

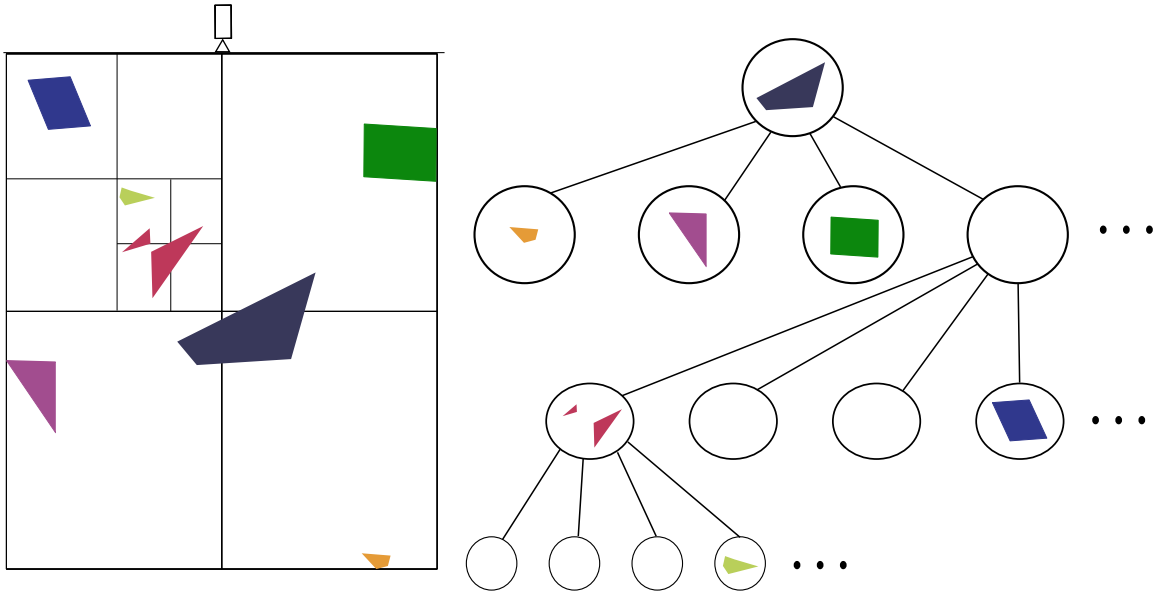
Opět je třeba dopředu zmínit výsledek implementace. Po implementaci tento algoritmus obsahoval některé chyby, které často způsobovaly nesprávné zacyklení u některých typů modelů. V případech, kdy proběhl, se však ukázala optimalizace seznamem jako nedostatečná, neboť už i při pár stovkách polygonů byl algoritmus velmi pomalý kvůli stále vysokému počtu kontrolovaných dvojic a samotný převod trval několik sekund. Kvůli tomuto zásadnímu nedostatku nebyly dále hledány a laděny některé specifické chyby a způsob tohoto řešení byl zavržen jako příliš neefektivní, proto nebude nadále rozebírán.

Metoda č. 3B: Detekce kolizí s použitím octree

Z předchozího pokusu je zřejmé, že je třeba nadále optimalizovat počet dvojic, ve kterých je konflikt hledán. K tomu lze použít datovou strukturu zvanou octree (oktálový strom, dále bude používán kratší anglicky výraz octree). Tento typ stromové datové struktury je často používán právě pro detekci konfliktů ve 3D prostoru. Obecně se jedná o stromovou strukturu, ve které každý uzel má právě 8 uzlů synovských.

Octree nejčastěji slouží k postupnému rozdělení 3D prostoru na 8 stejně velkých podprostorů. Jednotlivé uzly stromu tyto podprostory reprezentují. Ty mohou být dále děleny tím stejným způsobem na menší a menší podprostory. Při hledání sousedních nebo konfliktních objektů v octree tedy není třeba prohledávat celý prostor, ale pouze aktuální podprostor, což obecně značně urychluje různé typy algoritmů hledání v prostoru [5, s. 230–232]. Prin-

cip octree je poměrně obecný, proto bude dále popsán specifičtější způsob jeho využití pro detekci kolizí polygonů. Způsob reprezentace scény pomocí octree lze vidět na obrázku 3.15.



Obrázek 3.15: Vizualizace octree scény (vlevo, pohled shora, tedy podobný tzv. quadtree) a struktury octree (vpravo). Nejtmavší čtyřúhelník bude ležet v kořenovém uzlu octree, protože se nevleze do žádného podprostoru. Zelený, růžový a oranžový polygon budou ležet přímo v synovských uzlech kořene, přestože by se vlezly ještě do menších rozdělení, protože se v těchto částech octree nachází samy a není tedy třeba strom dále rozdělovat. Zbylé polygony jsou dle potřeby dále děleny hlouběji do octree, dokud se nenachází v uzlu samy (modrý a světle zelený polygon) nebo dokud se nevlezou do dalšího dělení podprostoru (červené polygony). Rozměry původního kořenového uzlu octree jsou určeny na jednotlivých osách nejextrémnějšími hodnotami koordinát vrcholů jednotlivých polygonů a pozicí kamery.

V tomto případě lze použít octree pro optimalizaci takovým způsobem, že všechny polygony jsou nejdříve rozděleny a uloženy do jednotlivých uzlů octree a následná detekce konfliktů poté probíhá pouze mezi dvojicemi v rámci stejného uzlu. Jsou uvažovány také tomuto uzlu nadřazené uzly, protože v octree nelze rozdělit všechny polygony striktně do listových uzlů, neboť to by vyžadovalo řezání větších polygonů, které by se do více zanořených uzlů nevešly. Jak už bylo předem zmíněno z implementace BSP stromu, tyto „zbytečné“ řezy by se promítly na výsledném obraze, například jakožto svislé a vodorovné mezery na větších plochách.

Samotná tvorba octree probíhá tedy postupným přidáváním polygonů ze seznamu do octree. Pro lepší představu o způsobu konstrukce octree v rámci této práce lze uvést následující pravidla přidávání polygonů do stromu (počínaje od kořenového uzlu).

1. Pokud aktuální uzel zatím neobsahuje žádný polygon a nebyl ještě dále dělen na 8 synovských uzlů (resp. tento podstrom zatím neobsahuje žádné polygony), přidej polygon do uzlu a konči.

2. Pokud aktuální uzel už obsahuje alespoň jeden polygon a nebyl ještě dělen na 8 synovských uzlů, rozděl uzel a přesuň všechny jeho polygony do těch synovských

uzlů, do kterých se vlezou celé; pokud se nevlezou do žádného, ponech je v aktuálním uzlu.

3. Pokud se přidávaný polygon vleze celý do některého synovského uzlu aktuálního uzlu, opakuj proces od začátku s tím, že novým aktuálním uzlem se stane právě tento synovský uzel; v opačném případě přidej polygon do aktuálního uzlu a konči.

Toto stromové uspořádání polygonů scény značně optimalizuje proces díky tomu, že podstatně sníží počet kontrolovaných dvojic polygonů. V takto vytvořeném octree může totiž každý polygon být v konfliktu pouze s polygony podstromů synovských uzlů, s polygony aktuálního uzlu, nebo s polygony rodičovských uzlů (od přímého rodiče až ke kořeni), ostatní dvojice tedy nebude nutno kontrolovat.

Dále lze snížit počet kontrol tak, že detekce konfliktů probíhá postupně od nejhlouběji zanořených uzlů směrem ke kořeni. V listových uzlech není potřeba kontrolovat synovské uzly, neboť žádné nemá, stačí se zaměřit pouze na aktuální uzel a uzly rodičovské, a to od přímého rodiče až ke kořeni. Protože kontrola probíhá postupně od listových uzlů, i pro všechny nelistové uzly platí, že není třeba kontrolovat synovské uzly (detekce takových konfliktů už totiž proběhla v synovských uzlech, pokud už dříve nebyly polygony synovského uzlu A v konfliktu s polygony rodičovského uzlu B, je zřejmé, že polygony rodičovského uzlu B nebudou naopak v konfliktu s polygony synovského uzlu A).

Podobnou myšlenkou lze optimalizovat i počet kontrol v rámci jednoho uzlu, kdy každý uzel obsahuje dva seznamy polygonů – resolved a unresolved („vyřešené“ a „nevyřešené“, dále budou používány pro odlišení od okolního textu anglické názvy). Zpočátku jsou všechny polygony každého uzlu v seznamu unresolved, seznam resolved je prázdný. Pokud kontrolovaný polygon daného uzlu už není v konfliktu s žádným jiným polygonem, je přesunut ze seznamu unresolved do seznamu resolved. Každý další polygon už poté při detekci konfliktů v rámci aktuálního uzlu nekontroluje všechny polygony uzlu, ale pouze ty, které ještě zbývají v seznamu unresolved – polygony v seznamu resolved už kontrolu jednou provedly, je zřejmé, že výsledek kontroly se nezmění při záměně pořadí kontrolovaných polygonů.

Následující část spadá spíše pod samotný návrh pluginu, pro lepší pochopení výše popsaných principů metody řezání je však dobré ji zmínit zde. S myšlenkou těchto optimalizací a sestaveným octree na základě výše zmíněných pravidel lze tedy shrnout metodu jako následující algoritmus 3, který slouží k vyřešení všech konfliktů polygonů ve scéně, a tedy i k vyřešení prvního podproblému hloubkového řazení, kterým je eliminace konfliktů (detekce konfliktů v následujícím algoritmu je prováděna kontrolou stejných 6 podmínek, které jsou popsány v části Detekce a eliminace konfliktů).

Algoritmus 3: Hledání a eliminace konfliktů uvnitř octree

Vstup : octree s konfliktními polygony
Výstup: octree bez konfliktních polygonů

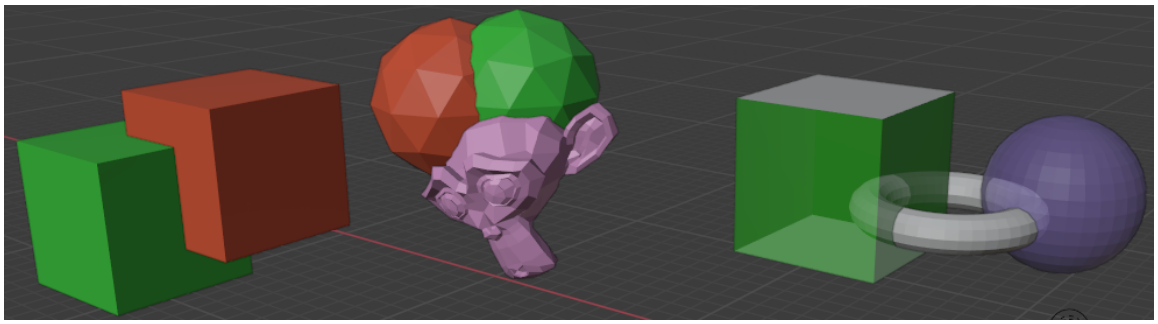
```
1  Vytvoř seznam všech uzlů octree;
2  Seřaď seznam uzlů od nejvíce zanořených (nejhlubších) uzlů až po kořen;
3  foreach uzel N v seznamu uzlů do
4      while v seznamu unresolved uzlu N je alespoň 1 polygon (polygon P) do
5          foreach polygon Q v seznamu unresolved uzlu N do
6              if polygony P a Q jsou v konfliktu then
7                  rozřež rovinou nalezeného konfliktního polygonu Q aktuální polygon
                    P na dva fragmenty;
8                  oba fragmenty vlož do seznamu unresolved uzlu N;
9                  go to 4;
10             end
11         end
12         foreach rodičovský uzel R (od přímého rodiče až ke kořeni) do
13             foreach polygon Q v seznamu unresolved uzlu R do
14                 if polygony P a Q jsou v konfliktu then
15                     rozřež rovinou nalezeného konfliktního polygonu Q aktuální
                        polygon P na dva fragmenty;
16                     oba fragmenty vlož do seznamu unresolved uzlu N;
17                     go to 4;
18                 end
19             end
20         end
           // Žádný konflikt nebyl nalezen
21         přesuň polygon P ze seznamu unresolved uzlu N do seznamu resolved uzlu
            N;
22     end
23 end
```

Všechny výsledné polygony lze ze stromu uložit zpět do výsledného seznamu, o kterém lze říct, že existuje takové pořadí vykreslení jeho jednotlivých polygonů, při kterém vznikne korektní obraz. Otázkou však zůstává, jak toto pořadí algoritmicky určit.

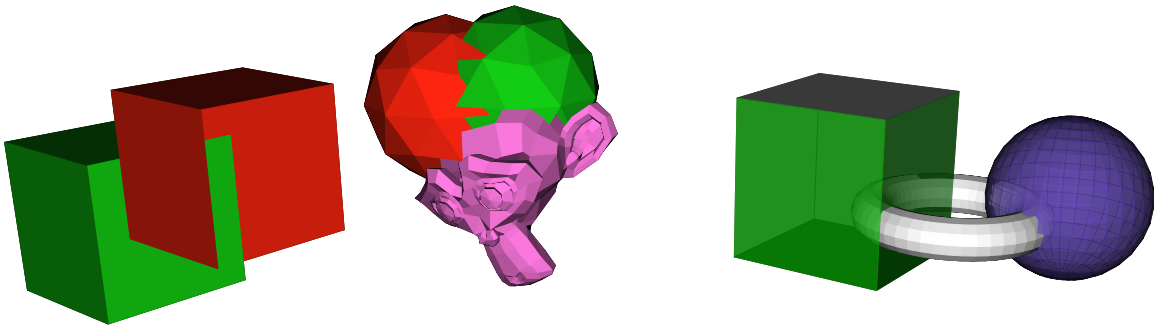
Tato metoda sice neřeší způsob určení pořadí zápisu, lze ji však zkombinovat s metodou primitivního seřazení podle hloubky, jejíž kvalita obrazu byla zhoršena zčásti právě konfliktními, neseřaditelnými polygony (viz obrázek 3.11), které lze pomocí octree metody odstranit. Avšak kvůli některým konfiguracím (jako například těm popsaným na obrázcích 3.9 a 3.10) výsledný obraz stále není v určitých situacích ideální.

Obecně však lze říct, že kombinace těchto metod by v situacích, kdy jednotlivé objekty mají větší množství menších polygonů, které i mohou být v konfliktu, měla přinášet poměrně kvalitní výsledky. Zároveň by měla být schopna pracovat podstatně rychleji než při kontrole všech dvojic v obyčejném seznamu.

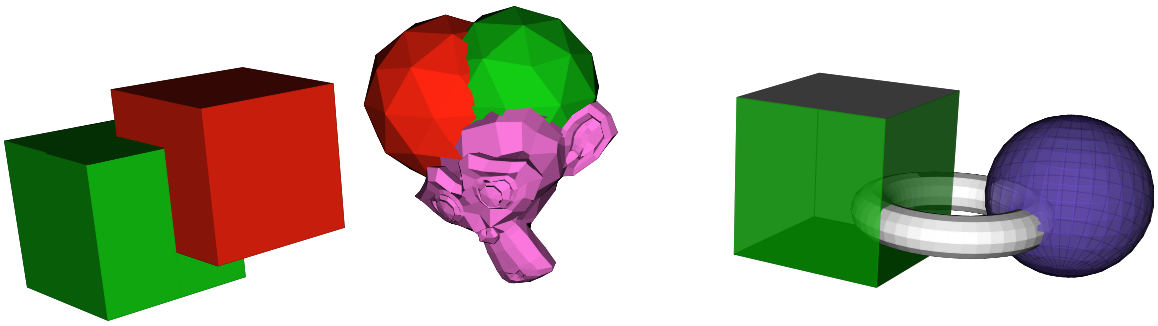
Pro zlepšení představy o kvalitě výsledných obrazů těchto jednotlivých metod následují obrázky 3.16, 3.17 a 3.18 porovnávající jednotlivé vstupy a vykreslené výsledné soubory, pořízené po implementaci.



Obrázek 3.16: Převáděné vstupní modely ve scéně v Blenderu, vlevo zaklíněné krychle, uprostřed zaklíněné složitější modely a vpravo zaklíněné průhledné modely.



Obrázek 3.17: Výsledný obraz po primitivním seřazení podle hloubky. Krychle jsou nekorektně vykresleny a vykreslení prostředních modelů je poměrně nepřesné.



Obrázek 3.18: Výsledný obraz po kombinaci eliminace konfliktů a primitivního seřazení. Krychle jsou vykresleny korektně a prostřední modely jsou vykresleny o něco přesněji. Přesnost vykreslení pravých modelů se příliš nezměnila.

Metoda č. 4: Newellův algoritmus

Přestože problematika byla předchozí metodou částečně zjednodušena, jádro problému, kterým je určení pořadí, stále zůstává.

Poslední pokus o určení správného pořadí v této práci představuje dříve zmíněný Newellův algoritmus, který je jedním z hlavních představitelů metod třídy List-Priority a jehož část už byla použita pro detekci konfliktů. Tento algoritmus celkově řeší problematiku řezání a řazení současně. Polygony jsou ukládány do seznamu. Detekce konfliktů, řezání a řazení probíhá mezi polygony na konci seznamu ve specifickém pořadí. Pomocí značkování polygonů a jejich řezání dochází k zaměňování jejich pořadí na základě jejich vzájemných pozic takovým způsobem, aby byl nalezen polygon, který už nemůže být s žádným jiným v konfliktu a může být tím pádem vykreslen. Podrobný popis a diagram tohoto algoritmu celkově lze nalézt v literatuře [6].

Je vhodné však dopředu zmínit výsledek implementace této metody. Ačkoliv se povedlo pomocí tohoto algoritmu provést korektně detekci a eliminaci konfliktů, nezdařilo se získat korektní pořadí zápisu jednotlivých polygonů. Výsledné obrazy proto měly v některých případech poměrně náhodné seřazení, které vedlo k nízké kvalitě výsledků. Přesto se však v kombinaci s metodou primitivního řazení polygonů povedlo tyto výsledky alespoň o trochu zlepšit, přestože seřazení stále nebylo korektní. V některých případech dokonce řezání podle Newellova algoritmu přináší o trochu přesnější výsledky než detekce a řezání pomocí octree, a proto bude ve výsledném pluginu zahrnuta možnost nastavit použití této metody pro řezání konfliktních polygonů.

Závěr hloubkového řazení

Na závěr lze říct, že hloubkové řazení se v kombinaci s vektorovou grafikou a formátem SVG nejeví jako triviální záležitost, a proto se v rámci této práce nepodařilo problém zcela vyřešit. V této podkapitole však bylo popsáno několik metod a způsobů, které alespoň zmírňují negativní dopady špatného hloubkového řazení na kvalitu výsledného obrazu.

Pro shrnutí lze uvést, že výsledný plugin bude podporovat 3 způsoby řešení konfliktních polygonů:

- řezání a řazení pomocí BSP stromu,
- detekce konfliktů a řezání pomocí octree v kombinaci s primitivním řazením podle hloubky,
- detekce konfliktů a řezání pomocí Newellova algoritmu v kombinaci s primitivním řazením podle hloubky.

Při nepoužití žádné metody pro odstranění konfliktů budou polygony řazeny pouze primitivním řazením podle hloubky, které ještě navíc bude podporovat více heuristik pro určování hloubky polygonu.

Kapitola 4

Návrh převodu, uživatelského rozhraní a struktury pluginu

Ačkoliv nejdůležitějším bodem této práce je samotný převod dat z modelů na vektorovou grafiku, je potřeba nad problémem uvažovat nejen z teoretického hlediska použitých metod a algoritmů předchozí kapitoly, ale také z praktického hlediska prostředí Blenderu. Blender je aplikace pro tvorbu 3D modelů, efektů, animací apod. Obsahuje širokou škálu nástrojů s versatilními funkcionalitami pro různá odvětví 3D grafiky. Samotnou funkcionalitu Blenderu lze však také libovolně rozšiřovat, neboť Blender umožňuje svým uživatelům tvořit pluginy pomocí Blender Python API. Následující návrhová kapitola proto nejprve vysvětluje formát uložení dat jednotlivých modelů v Blender API a kontextových proměnných v Blenderu, které společně tvoří vstupní data pro převod. Dále se zabývá formátem SVG pro zápis výstupních dat vektorové grafiky. Následně shrnuje celkový proces potřebný pro převod vstupů na výstup, a to jak z pohledu převodu typů dat, tak z pohledu jednotlivých kroků převodu, které bude muset plugin vykonávat. Dále rozebírá specifičtější část návrhu, a to návrh uživatelského rozhraní v Blenderu samotném a také návrh struktury pluginu.

Následující podkapitoly mohou obsahovat místy podrobnější popis metod a tříd připomínající implementaci, protože se ale jedná o kontext a funkcionalitu poskytovanou samotným API a ne o popis implementace, byly tyto části zahrnuty do návrhu. Většina těchto metod a tříd totiž popisuje formát dat na vstupu, jehož znalost je potřebná pro navržení převodu.

Veškeré části textu této kapitoly zmiňující specifikace Blender API čerpají z oficiální dokumentace Blender Python API pro verzi Blenderu 2.90[1].

4.1 Formát vstupních a výstupních dat (v Blenderu a SVG)

Aby byl Blender použitelný pro mnohá odvětví 3D grafiky, obsahuje různé typy pohledů na data s různými zaměřeními. Na úvod je proto dobré uvést, že tento plugin bude pracovat s běžným pohledem na hlavní scénu obsahující jednotlivé objekty, nejlépe přístupný na kartě „Layout“ (v kontextu API je tento pohled na 3D scénu označen jako „VIEW_3D“).

Následující části popisují vstupy a výstupy převodu z praktického hlediska, a to formát dat objektů v hlavní scéně, kontextové proměnné této scény důležité pro převod a na závěr specifikaci formátu SVG.

[1] Dokumentace Blender 2.90.1 Python API: https://docs.blender.org/api/2.90/

Vstupní 3D model v Blenderu

Všechny převáděné objekty scény jsou instance třídy `bpy.types.Object`. K těmto instancím poskytuje API přístup a lze z nich číst informace potřebné k převodu. Podstatnými vlastnostmi těchto instancí jsou:

- `type` – výčtový typ specifikující druh objektu jako například 'MESH', 'CURVE', 'LIGHT' nebo 'CAMERA', tuto vlastnost je třeba kontrolovat a převádět pouze objekty typu 'MESH',
- `data` – data objektu, u objektu typu 'MESH' obsahuje samotný polygon mesh (polygonová síť, dále jen mesh) objektu,
- `matrix_world` – transformační matice, která slouží k převodu objektu z lokálních souřadnic na souřadnice scény,
- `material_slots` – obsahuje seznam materiálů objektu, na tento seznam se poté odkazují jednotlivé plochy meshe, kterým jsou materiály přiřazeny právě z tohoto seznamu.

Samotná vlastnost data však není uzpůsobena pro čtení dat popisující mesh. K tomuto účelu slouží v API třída `bpy.types.BMesh`. Instance této třídy lze získat pomocí speciálních funkcí této třídy, které slouží k vytvoření kopie meshe z vlastnosti původních objektů `object.data`. S touto kopií už lze snadněji pracovat jako se vstupními daty, neboť už je instancí třídy `BMesh` a obsahuje následující podstatné vlastnosti a metody:

- `faces` – hlavní vlastnost meshe, jedná se o seznam všech ploch (faces) daného meshe se souřadnicemi všech vrcholů, každá plocha se promítne do výsledného obrazu jakožto mnohoúhelník,
- metoda `transform()` – pro transformaci souřadnic pomocí transformačních matic,
- metoda `new()` - pro vytvoření nové instance třídy `BMesh`,
- metoda `from_mesh()` - pro vytvoření kopie meshe z dat objektu,
- metoda `free()` - pro uvolnění paměti.

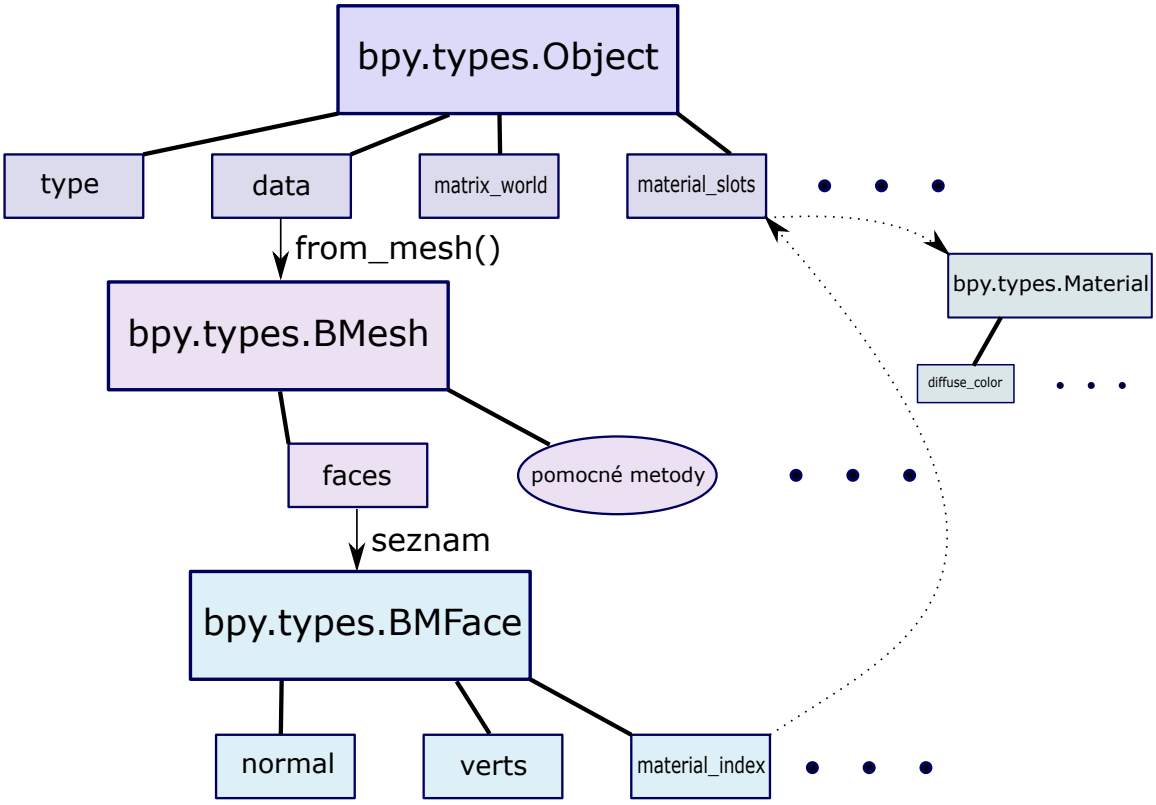
Nyní už zbývají pouze vlastnosti instancí třídy `bpy.types.BMFace` popisující jednotlivé plochy meshe:

- `normal` – normálový vektor plochy, trojice (x, y, z), který bude potřebný pro zjištění, jak je plocha v prostoru otočena,
- `verts` – hlavní vlastnost plochy, jedná se o seznam všech vrcholů definujících plochu meshe, jednotlivé vrcholy obsahují potom své souřadnice jako trojici (x, y, z),
- `material_index` – index prvku ve výše zmíněném seznamu materiálů, tento odkaz specifikuje, který materiál je přiřazen právě této ploše, čímž udává její barvu a průhlednost (plocha ale nemusí nutně mít nějaký materiál přiřazen).

Pro úplnost lze také zmínit vlastnost instancí třídy `bpy.types.Material`, která bude použita při výpočtu výsledné barvy plochy, pokud má plocha nějaký materiál přiřazen:

- `diffuse_color` – základní barva materiálu, čtveřice (R, G, B, A) popisující barvu a průhlednost.

Všechny výše zmíněné vlastnosti by dohromady měly postačovat k popisu převáděného modelu, neboť kompletně popisují jeho strukturu, pozici a materiál. Uložení modelu v Blenderu názorněji popisuje obrázek 4.1.



Obrázek 4.1: Diagram hierarchie uložení modelu v Blender API z pohledu tříd a jejich vlastností. Jednotlivé třídy obsahují mnohem více vlastností, byly však vybrány pouze ty relevantní pro převod. Vlastnost data neobsahuje instanci třídy pro uložení meshe přímo, lze ji však na tuto instanci převést zmíněnou metodou. Index materiálu plochy se odkazuje na materiály uložené v samotném objektu.

Kontextové proměnné v Blender API

V předchozí části byly blíže popsány třídy, jejichž instance uchovávají informace o modelu. Pro převedení 3D modelu na výstupní data však informace o samotném modelu nestačí, důležitý je také kontext. Ten lze rozdělit v případě tohoto převodu na 2 skupiny proměnných. Jednou skupinou jsou proměnné samotného pluginu, respektive nastavení pluginu, kam patří například způsob výpočtu stínování nebo volba zdroje světla. Počet a typy těchto proměnných lze později zvolit dle potřeby, jedná se totiž spíše už o implementační detail celého pluginu a tudíž v této kapitole nebudou brány v potaz. Druhou skupinou kontextových proměnných relevantních pro tuto kapitolu jsou kontextové proměnné samotné scény Blenderu, ke kterým poskytuje přístup API. Jako příklad lze uvést třeba rozměry okna scény v Blenderu.

Kontext však v API není pouze globální proměnná, která by byla vždy přístupná ze všech míst programu. Aby plugin v Blenderu dokázal provádět nějakou operaci (například po stisknutí určitého tlačítka v pluginu, v případě pluginu pro převod třeba tlačítko „EXPORT“), je potřeba definovat operátor. To je třída, která splňuje určité náležitosti požadované samotným Blenderem, mezi které patří dědění ze třídy `bpy.types.Operator` a definice několika metod. Tou hlavní je metoda `execute()`, která je volána při stisknutí tlačítka, které má operátor obsluhovat. Pro tuto sekci je však relevantní parametr této metody, `context` (instance třídy `bpy.context`), kterým je při stisknutí tlačítka kontext předán této metodě. Přístup k tomuto objektu zároveň poskytuje přístup ke všem potřebným kontextovým proměnným, které jsou jeho vlastnostmi. Mezi ty relevantní pro tento plugin patří:

- `selected_objects` – seznam instancí třídy `bpy.types.Object` specifikující, které objekty ve scéně byly označeny/vybrány uživatelem (pro lepší použitelnost bude plugin převádět pouze uživatelem vybrané objekty ve scéně namísto všech objektů),
- `region` – odkaz na okno v Blenderu s pohledem na scénu,
- `region.height` a `region.width` – společně udávají výšku a šířku okna scény v Blenderu,
- `space_data.region_3d` – odkaz na pohled na 3D prostor scény,
- `space_data.region_3d.view_location`– souřadnice ústředního bodu (pivotu) pohledu na scénu jako trojice (x, y, z),
- `scene.export_properties` – ačkoliv bylo řečeno, že tato kapitola nepopisuje proměnné samotného pluginu, pro úplnost je dobré zmínit, že případné nastavení pluginu bude přístupné touto vlastností kontextu (název může být libovolný, zde byl použit `export_properties`).

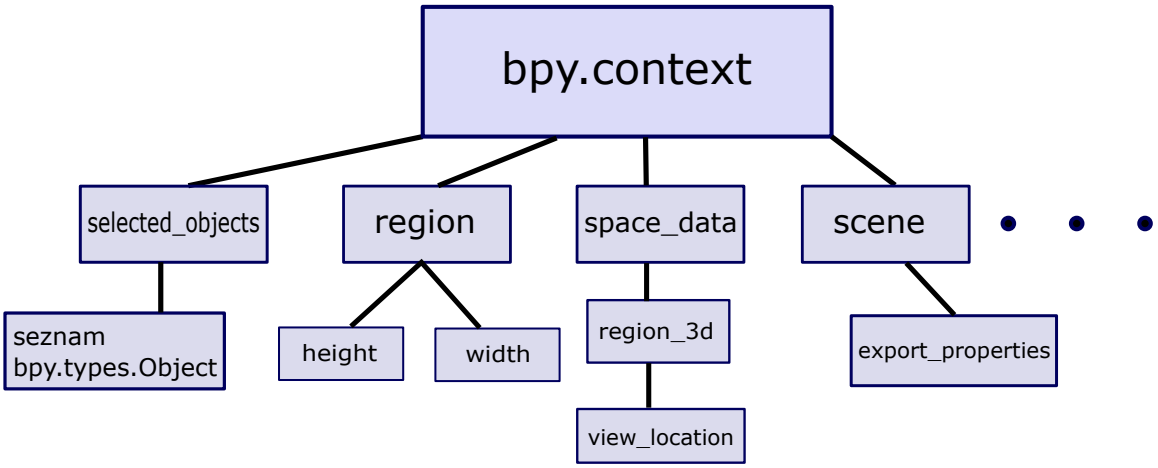
Výše zmíněné vlastnosti jsou postačující k popisu kontextu potřebnému pro převod dat. Některé zbylé vlastnosti (jako například pozice kamery) z nich lze dopočítat/odvodit, ať už výpočtem nebo speciálními metodami poskytovanými samotným API. Přehlednější souhrn poskytuje obrázek 4.2.

Výstupní 2D vektorová grafika

V předchozích částech byla popsána struktura veškerých vstupních dat (s výjimkou později definovaných vlastností pluginu). Nyní je potřeba se zaměřit na strukturu těch výstupních. Následující část textu čerpá ze specifikace formátu SVG verze 1.1[2].

Očekávaným výstupem by měl být soubor, ve kterém jsou obrazová data zapsána ve formátu SVG (Scalable Vector Graphics). Tento formát zápisu je založen na jazyce XML a slouží k definici 2D vektorové grafiky. Jednotlivé prvky jazyka XML slouží k popisu grafických útvarů obrázku. Tento formát a jeho vykreslení podporuje většina dnešních webových prohlížečů.

[2]Specifikace formátu SVG verze 1.1: https://www.w3.org/TR/SVG11/



Obrázek 4.2: Diagram hierarchie kontextových proměnných v Blender API. Třída pro uložení kontextu obsahuje mnohem větší množství vlastností, byly však vybrány pouze ty relevantní pro převod.

Pro úplnost lze uvést příklad syntaxe jazyka XML:

```
<prvek atribut1="hodnota1" atribut2="hodnota2">
    text/další prvky
</prvek>
```

Výstupem převodu v nejjednodušší formě tedy budou řetězce postupně zapisované do uživatelem zvoleného souboru. Je tedy třeba si ujasnit, jakou má takový soubor strukturu a jaké jsou typy a atributy jednotlivých prvků, kterými lze výsledný obraz popsat a kterými jsou tím pádem pro tento plugin relevantní. Podstatnými prvky jsou:

- `<svg>` – kořenový prvek souboru, jehož atributy ovlivní vykreslování všech prvků,
- `<g>` – seskupovací prvek; prvky do něj patřící budou členy stejné skupiny (všechny prvky generované tímto pluginem budou patřit do jedné společné skupiny, seskupeny budou z důvodu přehlednosti při případném ručním přidávání dalších prvků),
- `<polygon>` – hlavní prvek pro popis dat obrazu, ze kterého bude celý obraz složen, bude reprezentovat jednotlivé plochy modelů.

Pozornost je však také třeba věnovat atributům těchto prvků, nejdříve jsou tedy uvedeny atributy prvku `<svg>`:

- `width` a `height` – společně udávají rozměry okna, ve kterém budou jednotlivé prvky vykreslovány,
- `stroke-width` – šířka okrajů jednotlivých grafických prvků; na první pohled se může zdát méně podstatná, má však velký dopad na kvalitu výsledného obrazu,
- `stroke-linejoin` – udává tvar na rozhraní dvou po sobě jdoucích tahů; tento atribut bude však vždy nastaven na hodnotu `"round"`, která tvoří nejméně artefaktů v místech s ostrými úhly mezi jednotlivými tahy,

- `version` – verze SVG,
- `xmlns` – jmenný prostor xml dokumentu.

Jediným atributem seskupovacího prvku `<g>` je identifikátor skupiny `id`, který bude vždy nastaven na stejnou hodnotu, například `"model_to_svg_group"`.

Nejdůležitější jsou atributy prvku `<polygon>`:

- `points` – seznam vrcholů polygonu ve formátu `"x1,y1 x2,y2 x3,y3 ..."`, kde x a y jsou reálná čísla s libovolnou přesností udávající souřadnice vrcholu, výsledný polygon vzniká postupným spojením těchto vrcholů a vyplněním prostoru mezi nimi,
- `fill` – barva výplně polygonu ve formátu `"rgb(r,g,b)"`, kde r, g a b jsou čísla v rozmezí 0-255 udávající barvu dle modelu RGB,
- `stroke` – barva okraje polygonu ve formátu `"rgb(r,g,b)"`, kde r, g a b jsou čísla v rozmezí 0-255 udávající barvu dle modelu RGB,
- `fill-opacity` – průhlednost výplně polygonu ve formátu `"a"`, kde a je reálné číslo v rozmezí 0.0-1.0.
- `stroke-opacity` – průhlednost okraje polygonu ve formátu `"a"`, kde a je reálné číslo v rozmezí 0.0-1.0.

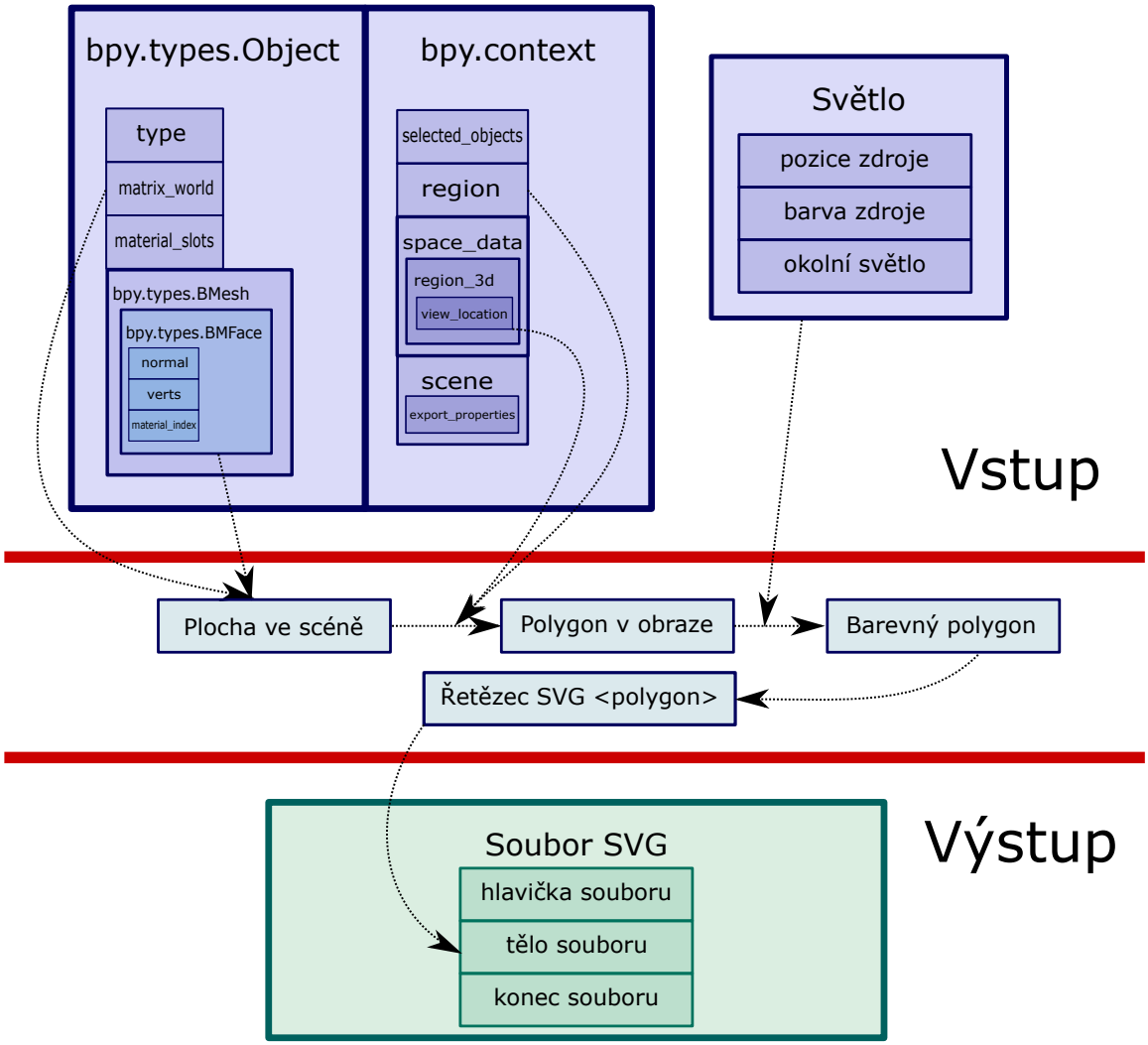
Pro lepší představu lze uvést příklad obsahu souboru ve formátu SVG a obrázek 4.3 znázorňující jeho vykreslení:

```
<svg width="1504" height="736" stroke-width="0.39" stroke-linejoin="round"
version="1.1" xmlns="http://www.w3.org/2000/svg">
    <g id="model_to_svg_group">
        <polygon
            points="884.0052,296.7056 773.222,365.2916
            782.2241,470.6645 905.4088,401.6855"
            fill="rgb(44,51,128)" fill-opacity="1.0"
            stroke="rgb(44,51,128)" stroke-opacity="1.0"
        />
        <polygon
            points="753.2341,187.4269 690.7902,420.4423
            952.827,460.4"
            fill="rgb(9,169,4)" fill-opacity="0.671"
            stroke="rgb(9,169,4)" stroke-opacity="0.671"
        /> </g> </svg>
```

Výše zmíněné prvky a jejich atributy jsou postačující k vytvoření výsledného souboru. Pomocí jediného typu prvku pro popis polygonů lze totiž vykreslit libovolný převáděný model, neboť každá jeho plocha se do výsledného 2D obrazu promítne jako mnohoúhelník. Důležité je také zmínit, že obraz souboru ve formátu SVG je vykreslován postupně od prvního prvku až k poslednímu. Pokud tedy jsou v souboru 2 polygony, jejichž některé části se překrývají, bude nejdříve vykreslen ten, který je v souboru zapsán jako první. Následující druhý polygon potom bude po vykreslení ten první ve výsledném obraze překrývat.

Na závěr podkapitoly lze ještě pro lepší představu shrnout formát vstupů a výstupu zjednodušeným diagramem 4.4.

Obrázek 4.3: Vykreslený příklad souboru ve formátu SVG. Zdrojový soubor pro tento obrázek má stejný obsah, jako je uveden v předchozím příkladu souboru SVG.



Obrázek 4.4: Výsledný diagram převodu. Horní část popisuje vstupní doménu, která je určena převáděnými objekty, kontextem scény a nastavením pluginu (světlo spadá pod nastavení pluginu). Prostřední část krátce zmiňuje sekvenci převodů prováděných při konverzi ploch modelu na výsledné polygony ve formátu SVG. Spodní část popisuje výstupní soubor.

4.2 Jednotlivé kroky procesu převodu

V předchozí podkapitole byla popsána vstupní data dostupná prostřednictvím API a očekávaný formát dat výstupních. Nyní už zbývá přiblížit, jak by měl vypadat samotný proces převodu mezi těmito daty. Protože na vstupu jsou 3D modely (respektive souřadnice vrcholů ploch a barvy jejich materiálů) a výstupem je soubor popisující obraz, bude samotný proces vypadat jako zjednodušený vykreslovací řetězec (graphics pipeline) uzpůsobený pro výstup na vektorovou grafiku. Následující kapitola tento proces blíže specifikuj, a to nejdříve z hlediska transformace dat a poté z hlediska jednotlivých kroků, které je potřeba pro tyto transformace vykonat.

Převod z pohledu datových typů

Ačkoliv by bylo možné celý proces převodu popsat pouze jednotlivými úkony, názornější bude nejdříve uvést sekvenci typů dat popisujících obraz v takovém pořadí, v jakém s nimi bude plugin pracovat. Tato část tedy příliš nepřibližuje samotný proces, slouží spíše pro názornější ukázku a lepší pochopení problematiky.

Snažit se převádět samotný model přímo na soubor ve formátu SVG by nebylo praktické, už jen z toho důvodu, že původní vstupní data nelze měnit a API k nim poskytuje pouze přístup pro čtení.

1. Jak již bylo zmíněno v předchozí podkapitole, vstupními daty, které obsahují informaci o modelu, jsou **objekty** ve scéně Blenderu (instance třídy `bpy.types.Object`).
2. Tyto objekty mají vlastnost data, která u objektů typu ‘MESH‘ nese informaci o meshi modelu. Přístup k němu lze získat vytvořením **kopie meshe**.
3. Samotný mesh umožňuje přístup k **seznamu ploch** modelu (`Face`).
4. Jednotlivé plochy modelu lze promítnout do výsledného obrazu jako polygony. Tyto plochy tedy budou převedeny z typu `Face` na datový typ, který obsahuje pouze nejnutnější informace pro popis polygonu výsledného obrazu. Tento typ bude později definován v implementační části, prozatím lze použít název například `ViewPolygon`. Výstupem této části tedy bude **seznam typu `ViewPolygon`**.
5. Protože typ `ViewPolygon` bude obsahovat všechny informace potřebné pro popis polygonu výsledného obrazu, lze je převést na **seznam řetězců ve formátu SVG** jakožto prvek `<polygon>`, který slouží ke stejnému účelu.
6. Samotné řetězce ve formátu SVG už pouze zbývá zapsat do **souboru**.

Názornější pohled na sekvenci transformace datových typů nastiňuje diagram 4.5.

Obrázek 4.5: Převod z hlediska datových typů od vstupního modelu až po výstupní soubor

Převod z pohledu jednotlivých kroků

Nyní zbývá blíže popsat sekvenci jednotlivých kroků, které výše zmíněné transformace a převod samotný budou zajišťovat. Pro úplnost jsou zde však uvedeny také kroky před počátkem převodu. Principy některých složitějších kroků byly blíže popsány v teoretické kapitole 3.

1. Kontrola volby objektů a nastavení pluginu. Před začátkem převodu je třeba zkontrolovat vstupní data převodu. Jedná se o seznam vybraných objektů k převodu, u kterého je třeba ověřit, zdali jsou vybrány objekty správného typu. Stejně bude potřeba ověřit vybraný zdroj světla a jeho typ. Také se jedná o kontextové proměnné převodu, které jej ovlivňují a které společně s převáděnými objekty slouží jako vstupní data.

2. Vytvoření a otevření výstupního souboru.

3. Generování hlavičky výstupního souboru ve formátu SVG. Hlavička v tomto případě bude obsahovat kořenový prvek SVG a seskupovací prvek. Následuje část generování těla souboru SVG.

4. Vytvoření seznamu převáděných objektů a seznamu pro uložení výsledných polygonů.

5. Vytvoření kopií meshů objektů. Z těchto kopií lze získat informace o jednotlivých plochách modelů, které budou převedeny na výstupní polygony následujícími kroky.

6. Převod souřadnic vrcholů a normál kopie meshe z lokálních souřadnic na souřadnice scény (local space na world space).

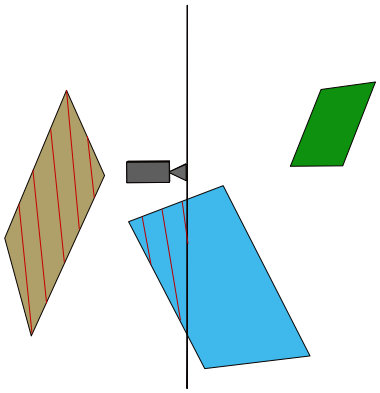
7. Backface culling. Tato metoda slouží k vyřazení takových ploch modelu, které by neměly být z pohledu kamery viditelné z toho důvodu, že jsou od kamery odvráceny svou přední stranou (typicky zadní strany modelu). Bližší popis této metody lze nalézt v podkapitole 3.3. Tato část převodu bude volitelná, například kvůli situaci, kdy uživatel chce zadní strany modelů zachovat z důvodu průhlednosti předních stran.

8. Frustum culling. Tato metoda běžně slouží k vyřazení objektů scény, které nejsou viditelné z pohledu kamery z toho důvodu, že leží mimo záběr kamery. Pro zjednodušení a zároveň kvůli metodám, které Blender API poskytuje, se v této části bude jednat pouze o vynechání nebo ořezání takových ploch, jejichž vrcholy leží v poloprostoru za kamerou. O vynechání a ořez těch ploch, které leží před kamerou, ale stále se celé nenachází v mezích okna, se bude později starat metoda Clipping. Princip obou těchto metod je blíže specifikován v podkapitole 3.4. Situace je pro rekapitulaci popsána na obrázku 4.6.

9. Převod zbylých ploch modelu, které nebyly vyřazeny v předchozích dvou krocích, na takový datový typ, který uchovává všechny podstatné informace pro popis polygonu ve výsledném obraze. Tento převod představují kroky 10-12.

10. Převod souřadnic vrcholů ze 3D souřadnic scény na 2D souřadnice výsledného obrazu.

11. Clipping. Tato metoda slouží k ořezání těch částí polygonů výsledného obrazu, které zasahují mimo meze okna (popřípadě k vynechání těch polygonů, jejichž všechny

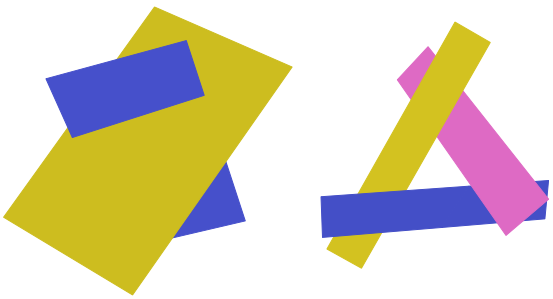


Obrázek 4.6: Frustum culling a Clipping, červenými pruhy jsou vyznačeny části, které jsou při převodu vynechány.

vrcholy leží mimo meze okna), ale stále reprezentují plochy, které se nacházely v poloprostoru před kamerou (ty v poloprostoru za ní už byly dříve vyřazeny v kroku Frustum culling). Bližší popis této metody lze opět nalézt v podkapitole 3.4.

12. Výpočet zbylých vlastností výsledného polygonu, tedy například barvy nebo průhlednosti. Princip výpočtu barvy byl podrobněji popsán v podkapitole 3.5.

13. Řezání konfliktních polygonů. Všechny z předchozích kroků získané polygony je třeba zkontrolovat pro ověření, zdali do sebe nejsou zaklíněné nebo zdali se cyklicky nepřekrývají (tyto situace lze vidět na obrázku 4.7). Kvůli dříve zmíněnému postupnému vykreslování souborů ve formátu SVG totiž nelze tyto polygony vykreslit po obyčejném převodu, protože neexistuje takové pořadí zápisu těchto polygonů, které by ve výsledku dalo korektní obraz. Je tedy nutné je řezat na menší. Tato problematika byla blíže popsána v podkapitole 3.6. Tato část převodu bude také volitelná, protože samotné řezání převod zpomalí a zvětší výsledný soubor, přičemž ne vždy je zapotřebí.



Obrázek 4.7: Neseraditelné polygony, vlevo zaklíněné, vpravo cyklicky překrývající se.

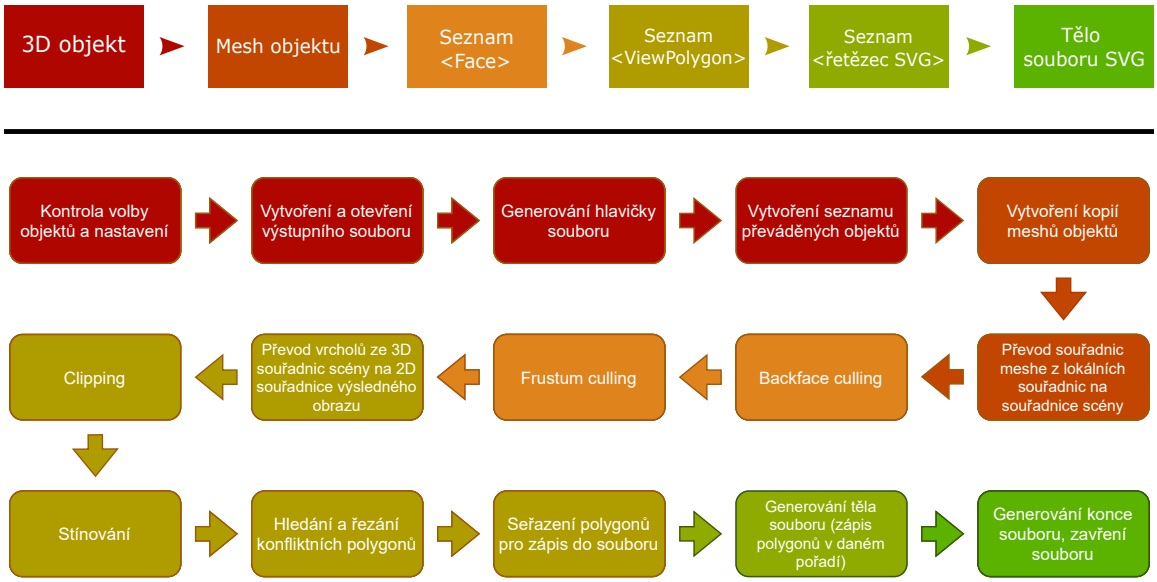
14. Seřazení seznamu polygonů pro zápis do výsledného souboru. Jak již bylo zmíněno, soubor ve formátu SVG je vykreslován od prvního prvku k poslednímu, je tedy nutné polygony do tohoto souboru zapsat v takovém pořadí, aby ve výsledku byl vykreslen korektní obraz. Tato problematika není zcela triviální, pokud je potřeba zajistit, aby byl obraz úplně přesný ve všech situacích. Blíže byla popsána společně s řezáním v podkapitole 3.6.

15. Zápis všech polygonů v daném pořadí do výstupního souboru.

16. Generování konce výstupního souboru.

17. Zavření souboru, výpis výsledné zprávy.

Po provedení výše zmíněných kroků budou vstupní data převedena na očekávaný výstup. Souhrn celého procesu převodu nastiňuje diagram 4.8.



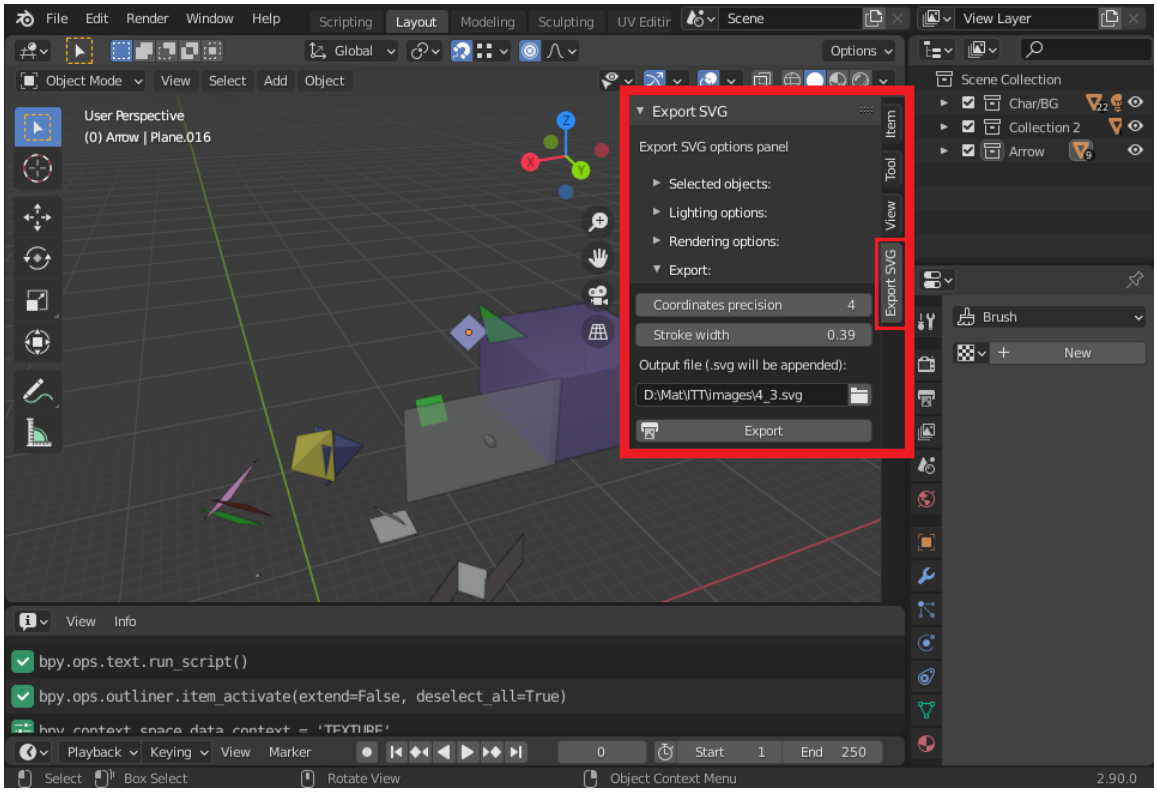
Obrázek 4.8: Digram procesu převodu v jednotlivých krocích. V horní části je znovu zmíněna sekvence převodu mezi jednotlivými typy dat, které jsou barevně odlišeny. Ve spodní části je popsána sekvence kroků převod zajišťujících, jednotlivé kroky jsou barevně odlišeny podle toho, s jakými datovými typy z horní sekvence v dané fázi převodu pracují.

4.3 Návrh uživatelského rozhraní

Až doposud se jednotlivé části textu zaměřovaly čistě na rozbor samotného procesu převodu, je však třeba mít na paměti, že cílem této práce je nejen tento převod navrhnout, ale také realizovat v rámci uživateli použitelného Blender pluginu. Následující části se tedy více zaměří na tvorbu pluginu samotného.

Nedílnou součástí jakéhokoliv rozšíření funkcionality je uživatelské rozhraní, které umožní jej využívat. Protože se jedná o plugin využívající specifické API, který musí pracovat už v existujícím prostředí, jsou možnosti návrhu uživatelského rozhraní poměrně omezeny, co se týče vizuálního stylu.

Základním prvkem uživatelského rozhraní Blenderu jsou panely, samotný plugin tedy bude jedním z takových panelů. Jak již bylo dříve zmíněno, Blender obsahuje pro různá odvětví počítačové grafiky různé pohledy na data s odlišnými funkcionalitami (např. okna pro modelování, tvorbu textur, tvorbu animací nebo skriptování všechna obsahují odlišné nástroje) a prvky uživatelského rozhraní se mění na základě aktuálního pohledu. Protože plugin bude v podstatě sloužit k „focení“ objektů ve scéně, bude vázán na okno pohledu na hlavní scénu. Nejvhodnější umístění pluginu bude tedy postranní panel tohoto okna. Lépe lze umístění těchto typů panelů vidět na obrázku 4.9.



Obrázek 4.9: Snímek obrazovky s umístěním panelu pluginu.

U samotného rozvržení obsahu okna není třeba dělat příliš mnoho složitých rozhodnutí, neboť obsah těchto panelů je tvořen prostým vertikálním skládáním jednotlivých elementů pod sebe. Výsledný plugin by tedy měl odpovídat rozložení znázorněném na obrázku 4.10.

Protože vizuální styl a rozvržení obsahu už jsou předem dány, zbývá pouze navrhnout samotné elementy panelu, resp. veškerá nastavení, která bude plugin podporovat a tlačítka pro jeho ovládání. Ty budou členěny do logických celků následovně:

- Vybrané objekty (Selected objects). Tato méně podstatná část panelu bude sloužit k zobrazení seznamu vybraných objektů pro převod. Aby ovšem nezabírala příliš mnoho prostoru, bude počet vypsaných objektů omezen. To však nevadí, protože hlavní podstatou této části je dát určitou míru zpětné vazby uživateli ještě před samotným převodem.

- Nastavení vykreslení (Rendering options). Tato část panelu bude sloužit k nastavení různých způsobů vykreslení, které se netýkají osvětlení. Zde bude patřit možnost nastavení vlastní barvy okrajů polygonů, vlastní barvy výplně polygonů, povolení metody Backface culling a také povolení řezání konfliktních polygonů. Dále se zde bude nacházet výběr metod pro řezání a řazení.

- Nastavení osvětlení (Lighting options). Tato část panelu bude sloužit k nastavení osvětlení scény. Mezi tato nastavení bude patřit možnost výběru typu zdroje světla, volba směru plošného světla nebo volba pozice bodového světla. Dále zde bude možno nastavit barvu zdroje světla a barvu okolního světla. Kromě zdrojů světla zde také

Obrázek 4.10: Příklad uspořádání prvků panelu v Blenderu. Kategorie lze vytvářet pomocí subpanelů (podpanelů). Jednotlivé panely a subpanely lze naplňovat dříve definovanými vlastnostmi a operátory pluginu, a to po řádcích. Na jednom řádku se může nacházet více prvků. Vzhled prvku a možnosti interakce s tímto prvkem jsou definovány typem a podtypem vytvořené vlastnosti. Například pro vlastnost podtypu barva je vykresleným prvkem výběr barvy, pro operátor je vykresleným prvkem tlačítko.

bude možno nastavit ignorování materiálů při výpočtu barvy polygonu a nebo povolení módu pro nebarevný převod na obraz odstínů šedi (tzv. grayscale).

- Export. Tato část panelu bude sloužit k nastavení vlastností výsledného souboru ve formátu SVG. Mezi ně patří nastavení přesnosti souřadnic vrcholů polygonů a nastavení šířky okrajů polygonů. Samozřejmě zde také patří výběr cesty k výslednému souboru. Na závěr tato sekce bude obsahovat dvě tlačítka, jedno pro provedení samotného převodu a druhé pro možnost resetování nastavení do výchozího stavu.

Výsledný vzhled uživatelského rozhraní lze vidět na obrázku z implementační části 5.1.

4.4 Struktura Blender pluginu

Následující část specifikuje, co vlastně samotný Blender plugin představuje, kde se v Blenderu nachází, jaké má náležitosti a jaká je přibližná struktura pluginu pro převod. Tato sekce byla záměrně umístěna na konec kapitoly návrhu a před začátek kapitoly implementace, protože se dotýká obou částí a zároveň je to první část textu řešící zdrojový kód. Přestože zmiňuje některé částečně implementační detaily, zabývá se především obecnými náležitostmi Blender pluginů a poté specifičtějším návrhem struktury pro výsledný převodový plugin.

Pro začátek je vhodné uvést, že seznam Blender pluginů lze nalézt v nastavení Blenderu na kartě Add-ons. Z této karty lze vyhledávat, povolovat a zakazovat instalovaná rozšíření. Rozšíření, která ještě nebyla nainstalována, je nutno manuálně instalovat pomocí zdrojových souborů.

Samotný plugin pro Blender má podobu souboru, popřípadě více souborů, s příponou „.py“. Jedná se tedy o modul v jazyce Python, který musí splňovat určité požadavky Blenderu. Jde sice o implementační detail, není jich ale mnoho a proto je lze pro úplnost stručně

vyjmenovat. Prvním požadavkem je definice proměnné ve formě datové struktury typu slovník, který obsahuje hlavičková data každého Blender pluginu, tedy například jméno pluginu, autora nebo kategorii rozšíření. Tím druhým jsou definice registračních a odregistračních funkcí, které jsou volány při povolování a zakazování rozšíření a starají se o načtení a uklizení prvků každého pluginu.

Výše zmíněné obecné požadavky stačí pro definici základního pluginu bez jakékoliv funkcionality. Nyní se lze zaměřit na návrh struktury samotného pluginu pro převod, a to z hlediska objektově orientovaného programování, resp. z hlediska potřebných tříd. Bližší detaily týkající se těchto tříd a jejich implementace jsou uvedeny v následující kapitole 5. V této podkapitole je uveden pouze jejich význam v rámci návrhu celkové struktury pluginu.

Pro celkové správné fungování bude plugin mimo třídy obsahovat:

- Slovník bl_info uchovávající informace o pluginu.
- Dvojici funkcí `register()` a `unregister()` starající se o povolování a zakazování pluginu.

Pro definici pluginu obecně jakožto prvku uživatelského rozhraní bude plugin obsahovat následující skupiny tříd, které společně tvoří jádro většiny Blender pluginů:

- Třídy pro uchovávání nastavení pluginu. Pokud je potřeba, aby plugin zaváděl nějaké své vlastní proměnné jako například nastavení, je nutno vytvořit a registrovat třídy, které reprezentují množiny těchto proměnných.
- Třídy pro definici operátorů pluginu. Pokud plugin obsahuje tlačítka nebo jiné prvky rozhraní, jejichž úlohou je spustit určitý proces, je potřeba vytvořit a registrovat třídy, jejichž metody jsou pro spuštění těchto procesů volány samotným Blenderem.
- Třídy pro definici uživatelského rozhraní pluginu. Pokud plugin vytváří své vlastní panely jakožto prvky uživatelského rozhraní, je potřeba vytvořit a registrovat třídy, které tyto panely reprezentují a jejichž metody se starají o jejich vykreslení. Tyto metody běžně vykreslují prvky dvou výše zmíněných skupin tříd jakožto prvky uživatelského rozhraní uvnitř panelů.

Po definování rozhraní pluginu zbývá tedy návrh tříd obstarávajících samotný proces převodu, který lze rozdělit následovně:

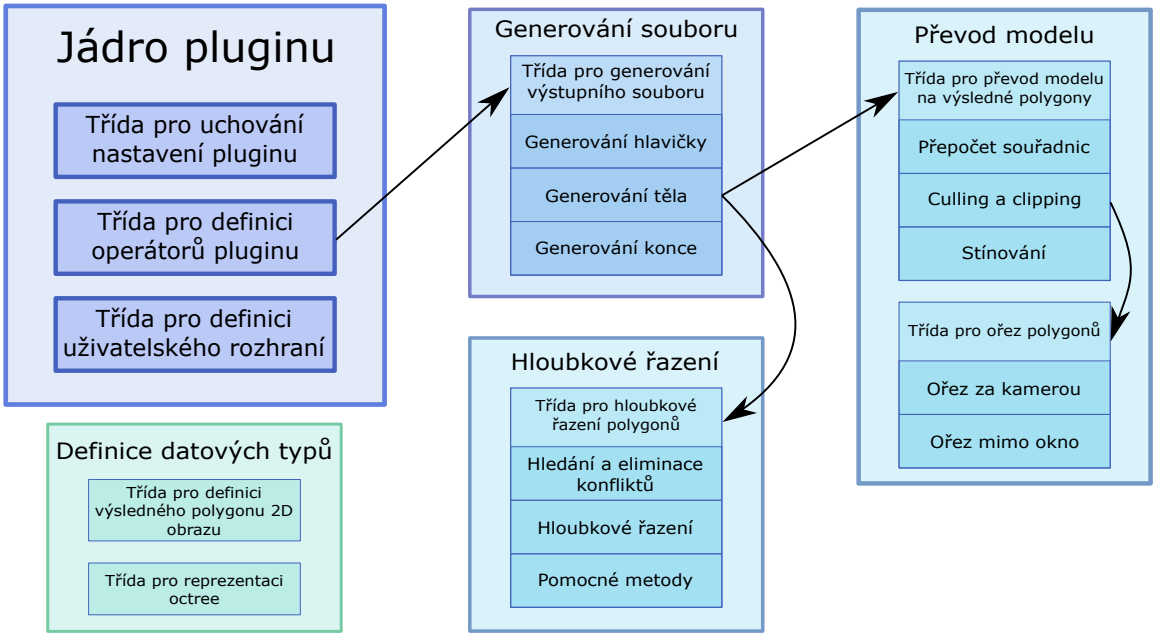
- Třída pro generování výsledného souboru ve formátu SVG. Tato třída bude základ řízení celého převodu a generování výstupu. Její úlohou bude generovat zvlášť hlavičku, tělo a konec výstupního souboru. Ke generování těla bude využívat ostatní třídy převodu.
- Třída pro převod modelu na výsledné polygony. Tato třída se bude starat o značnou většinu kroků převodu. V kontextu celkového procesu bude převádět jednotlivé objekty scény na seznam výsledných polygonů 2D obrazu. Tento převod bude zahrnovat přepočet souřadnic, Backface culling a výpočet barvy polygonů. Mezi těmito částmi bude probíhat také ořez polygonů ležících mimo okno, o ten se však bude starat třída následující.
- Třída pro ořez polygonu. Tato třída se bude starat o to, aby každý polygon byl ořezán takovým způsobem, aby žádná jeho část neležela mimo meze okna. Taktéž bude ořezávat ty polygony, jejichž některé vrcholy leží v poloprostoru za kamerou.

- Třída pro hloubkové řazení. Tato třída bude v podstatě soubor metod použitelných při hloubkovém řazení. Její metody budou sloužit pro určení takového výsledného pořadí zápisu polygonů do souboru, aby výsledný obraz po jeho vykreslení co nejvíce odpovídal situaci ve scéně Blenderu. Kromě řadicích metod bude tato třída také obsahovat metody sloužící k řezání a určování vzájemných pozic polygonů v prostoru.

Na závěr lze ještě navrhnout třídy, jejichž instance budou reprezentovat některé prvky scény a pomocné datové struktury:

- Třída pro reprezentaci výsledného polygonu 2D obrazu. Instance této třídy budou reprezentovat výsledný polygon společně se všemi jeho relevantními vlastnostmi a budou používány většinou zbylých tříd pluginu.
- Třídy pro reprezentaci octree. Jedna z těchto tříd a její instance budou reprezentovat samotný octree a budou sloužit jakožto jednotné rozhraní používané pro práci s octree jinými metodami. Druhá z těchto tříd a její instance pak budou reprezentovat jednotlivé uzly octree.
- Třída pro reprezentaci uzlu BSP stromu. S instancemi této třídy budou pracovat metody třídy pro hloubkové řazení při převodu pomocí metody BSP.

Lepší představu o celkové struktuře pluginu a sekvenci volání jednotlivých tříd může poskytnout diagram 4.11.



Obrázek 4.11: Diagram struktury pluginu z hlediska tříd. Jádro pluginu budou tvořit třídy definující plugin dědící z tříd definovaných API. Jádro pro zajištění procesu převodu bude využívat třídu pro generování výstupního souboru, která poté bude využívat pro samotný převod třídy pro převod modelu, ořez polygonů a hloubkové řazení. Posledními třídami budou třídy pro definice nových datových typů, jako je výsledný polygon obrazu nebo octree.

Kapitola 5

Implementace pluginu

Po shrnutí základních pojmů a algoritmů potřebných pro převod ze 3D do 2D a po návrhu jednotlivých kroků převodu, uživatelského rozhraní a potřebných tříd byl plugin implementován. Tato kapitola popisuje implementaci z hlediska jednotlivých tříd pluginu, jejich metod a vlastností a popřípadě také z hlediska jejich funkcionality v kontextu celého procesu převodu. Nejdříve je popsána implementace hlavních tříd Blender pluginu, které definují jeho vlastnosti/nastavení a uživatelské rozhraní. Následuje specifikace třídy starající se o generování výstupního souboru společně s definicí třídy pro popis výsledných polygonů. Poté je pozornost věnována třídám samotného procesu převodu, a to nejdříve třídám pro převod modelů na seznam výsledných polygonů a poté třídám pro jejich řazení. Kapitola je ukončena závěrečným shrnutím celkového procesu převodu modelu ve scéně na soubor ve formátu SVG z hlediska pořadí volaných metod.

Tento plugin byl implementován v jazyce Python pro verzi Blenderu 2.90. Pro přístup k Blender API slouží modul `bpy`, který bylo zapotřebí do pluginu zahrnout.

5.1 Definice vlastností a uživatelského rozhraní pluginu

Jádro samotného pluginu tvoří skupina tříd patřících do tří hlavních kategorií, které společně plugin definují, a proto byly implementovány jako první. Následující sekce popisuje jejich výslednou implementaci a je rozdělena na základě dříve zmíněných kategorií. Do této sekce lze také zařadit registrační a odregistrační funkce `register()` a `unregister()`, které nepatří do žádné třídy, musí být však v pluginu přítomny pro registraci a odregistraci všech tříd spadajících do některé z kategorií. Tyto třídy se vyznačují také tím, že dědí ze specifických tříd definovaných samotným API.

Vlastnosti pluginu (třída ExportSVGProperties)

Tato kategorie slouží k definici jednotlivých uživatelem nastavitelných vlastností pluginu. Vlastnosti ve třídě této kategorie definované lze později jednoduše vykreslit jakožto prvky uživatelského rozhraní na základě jejich typu. Aby Blender tuto třídu rozpoznal a správně ukládal nastavení pluginu, bylo zapotřebí, aby dědila ze třídy `bpy.types.PropertyGroup` a také aby byla registrována a rušena v rámci registračních funkcí jakožto skupina vlastností.

Samotná třída je tvořena pouze deklaracemi vlastností třídy a neobsahuje žádné metody. Jednotlivé vlastnosti reprezentují nastavení pluginu a všechny mají anotovány jejich datový typ. Tento datový typ vždy patří do třídy `bpy.props` (např. `bpy.props.StringProperty` nebo `bpy.props.BoolProperty`) a lze pomocí něj vlastnost blíže definovat, například uve-

dením jejího jména, popisku, výchozí hodnoty apod. Také umožňuje dále upřesňovat podtypy pro přesnější specifikaci vlastnosti. Touto bližší a přesnější specifikací lze do jisté míry ovlivnit, jak bude prvek vykreslen v uživatelském rozhraní pluginu. Tedy například po specifikaci podtypu `COLOR` u vlastnosti typu `bpy.props.FloatVectorProperty` je prvek vykreslen v uživatelském rozhraní jako výběr barvy namísto pole hodnot typu `float`.

Jednotlivé vlastnosti v této třídě deklarované zde nebudou podrobně popisovány, neboť by se jednalo pouze o výčet, který odpovídá dříve zmíněným možnostem nastavení v podkapitole 4.3 a který lze později vidět v praxi vykreslený na obrázku 5.1.

Operátory pluginu (třídy ExportSVGOperator a ExportSVGReset)

Třídy této kategorie slouží k definici jednotlivých proveditelných operací pluginu. V rámci uživatelského rozhraní jsou vykresleny jakožto tlačítka. Aby Blender tyto třídy operátorů rozpoznal, bylo zapotřebí, aby dědily ze třídy `bpy.types.Operator`, byly registrovány a rušeny v rámci registračních funkcí, definovaly vlastnosti specifikující jejich jméno a popisek a také především aby definovaly metodu `execute()`, která je volána při stisknutí tlačítka a které je předán při volání kontext Blenderu, který obsahuje například objekty scény (tato metoda vždy musí vracet hodnotu `{"FINISHED"}`).

Třída `ExportSVGReset` definuje operátor pro resetování nastavení pluginu. Její metoda `execute()` tak tedy pouze nastavuje vlastnosti pluginu na jejich výchozí hodnoty.

Třída `ExportSVGOperator` definuje hlavní operátor pro převod modelů na soubor ve formátu SVG. Definuje metodu `poll()`, která zajistí, aby proces nebyl spouštěn bez jakýchkoliv vybraných objektů. Dále definuje metodu `execute()`, která po kontrole některých nastavení ihned předává řízení a kontext poslední definované metodě třídy `main_export()`. Tato metoda slouží v podstatě jakožto hlavní metoda procesu převodu. V rámci této metody probíhá kontrola vybraného světla a převáděných objektů, která ověřuje, zdali byl vybrán plošný zdroj světla, nebo zdali byla jako světlo vybrána kamera nebo objekt typu `LIGHT` (s libovolným podtypem, světlo však bude vždy považováno za bodové) a zajišťuje, aby byly převáděny vybrané objekty pouze typu `MESH`. Dále metoda zprostředkovává otevření výstupního souboru, generování jeho obsahu pomocí třídy pro generování SVG, která je popsána v pozdější sekci 5.2, zavření souboru a vypsání výsledné zprávy.

Panely pluginu (ExportSVGPanel a potomci)

Třídy této kategorie slouží k definici uživatelského rozhraní jakožto panelů v Blenderu. Aby Blender tyto třídy panelů rozpoznal, bylo zapotřebí, aby dědily ze třídy `bpy.types.Panel`, byly registrovány a rušeny v rámci registračních funkcí, definovaly vlastnosti specifikující informace o panelu a také především aby definovaly metodu `draw()`, která je opakovaně volána pro vykreslení panelu.

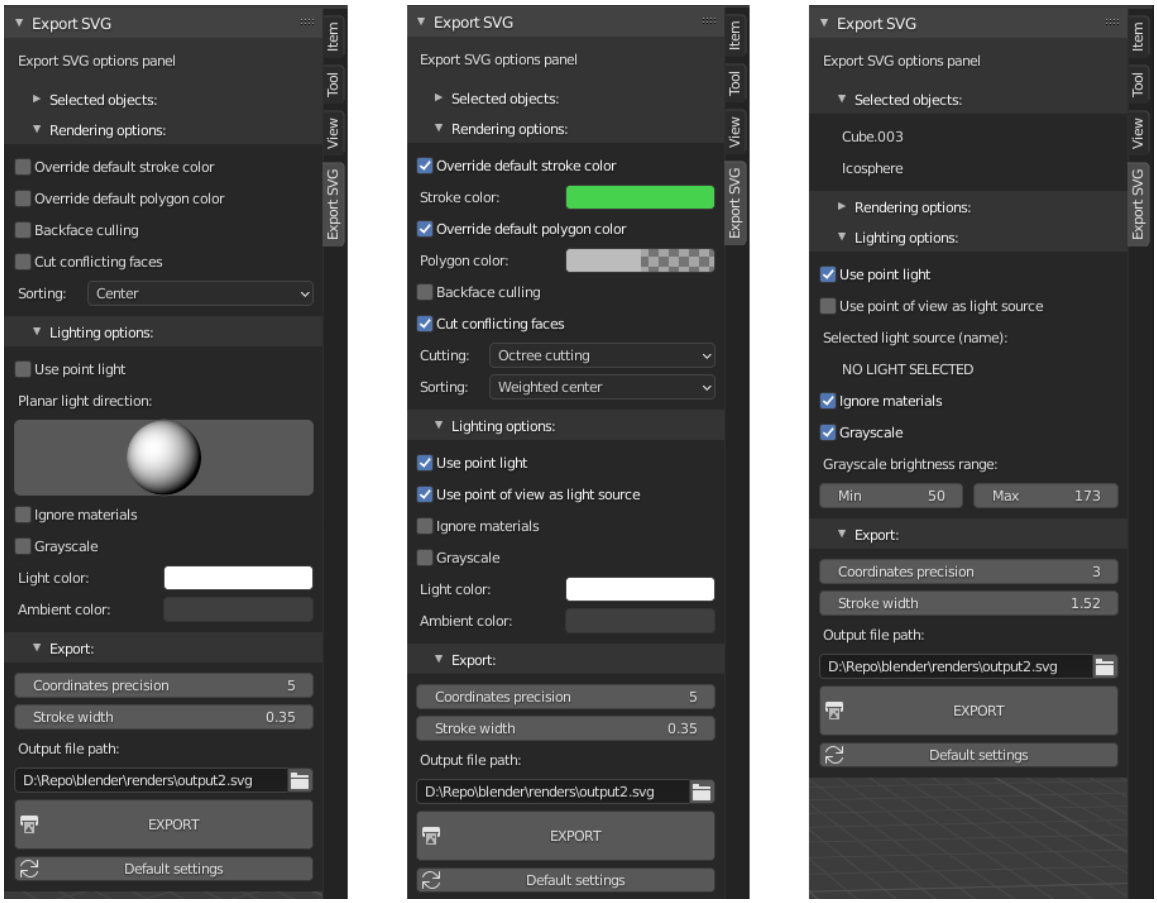
Byla navíc vytvořena třída `ExportSVGPanel`, která nereprezentuje panel Blenderu, ale pouze některé vlastnosti pro všechny ostatní panely společné, jako je například kategorie nebo pozice panelu. Všechny následující třídy dědí kromě vlastností třídy `bpy.types.Panel` také vlastnosti této třídy.

Třída `ExportSVGPanelMain` slouží pro definici hlavního panelu uživatelského rozhraní pluginu, její metoda `draw()` proto obsahuje pouze vykreslení hlavního popisku.

Jednotlivé třídy `ExportSVGPanelObj`, `ExportSVGPanelLight`, `ExportSVGPanelRender` a `ExportSVGPanelExport` poté slouží k definici vnitřních panelů a tím pádem jednotlivých kategorií hlavního panelu takovým způsobem, jakým bylo rozdělení naznačeno v sekci uživatelského rozhraní návrhové části 4.3. Jejich metody `draw()` ve většině případů vykreslují

jednotlivé popisky, vlastnosti pluginu deklarované ve třídě `ExportSVGProperties` a operátory definované ve třídách `ExportSVGReset` a `ExportSVGOperator`. S prvky těchto panelů může uživatel interagovat za účelem nastavení pluginu. Kromě obyčejného vykreslení všech prvků však kreslicí metody některých panelů obsahují také určitou logiku pro skrývání irelevantních kombinací nastavení. Například při výběru kamery jakožto zdroje světla nemá smysl vypisovat vybraný světelný objekt ve scéně.

Příklady výsledného stavu uživatelského rozhraní lze vidět na snímcích obrazovky 5.1.



Obrázek 5.1: Uživatelské rozhraní pluginu s různými kombinacemi nastavení

5.2 Generování souboru ve formátu SVG

Ačkoliv fáze zápisu výsledných polygonů do souboru je až posledním krokem celkového procesu převodu, je nutno zmínit, že tvorbou hlavičky tohoto souboru celý proces začíná a tvorbou konce souboru celý proces končí. Proto lze říct, že pokud třídy v předchozích sekcích tvořily jádro Blender pluginu, třída pro generování souboru potom tvoří jádro procesu převodu.

Generátor souboru (třída SVGFileGenerator)

Tato třída je tvořena souborem několika statických metod sloužících pro generování a zápis jednotlivých částí výsledného souboru, jehož deskriptor je těmto metodám předáván jako

parametr. Pro generování hlavičky, těla a konce souboru slouží metody `gen_svg_head()`, `gen_svg_body()` a `gen_svg_tail()`.

Metody pro generování hlavičky a konce pouze zapisují začátek a konec kořenového prvku `<svg>` a seskupovacího prvku `<g>`.

Jádro převodu však tvoří metoda pro generování těla souboru, která je trochu obsáhlejší. Této metodě jsou kromě deskriptoru souboru předávány také kontextové proměnné Blenderu, z nichž metoda určí pozici kamery v prostoru, směr jejího natočení a zdroj světla. Všechny tyto údaje předává společně s jednotlivými vybranými objekty třídě pro převod modelů (tato třída je blíže popsána v pozdější sekci 5.3), která této metodě vrací převedené modely v podobě seznamu výsledných polygonů. Postupným převodem všech objektů scény tedy vzniká jeden dlouhý seznam veškerých polygonů výsledného obrazu. Tento seznam je dále předáván třídě pro hloubkové řazení (tato třída je blíže popsána v pozdější sekci 5.4), která je použita podle nastavení uživatele buďto přímo pro řazení polygonů, nebo nejdříve pro eliminaci konfliktních polygonů a poté pro řazení. Seřazený výsledný seznam už je poté konvertován polygon po polygonu na řetězce ve formátu SVG, které jsou postupně zapisovány do výstupního souboru. Metoda končí zápisem posledního polygonu v seznamu.

Pro tuto konverzi polygonů na řetězce slouží poslední metoda této třídy s názvem `view_polygon_to_svg_string()`. Jejím úkolem je převést předanou instanci třídy reprezentující výsledný polygon na řetězec reprezentující prvek SVG `<polygon>`. Tento převod zároveň bere ohled na atributy této instance, které reprezentují výsledné atributy prvku ve formátu SVG, jako je například barva nebo seznam souřadnic vrcholů.

Reprezentace výsledných polygonů SVG (třída ViewPolygon)

Třída `ViewPolygon` slouží k definici datového typu popisujícího polygony výsledného obrazu. Na tento datový typ jsou všechny polygony scény převáděny v převaděči modelů, zároveň s tímto datovým typem pracuje hloubkový řadič a právě objekty tohoto typu jsou převáděny na výsledné řetězce formátu SVG. Bylo proto třeba tento typ definovat takovým způsobem, aby obsahoval všechny potřebné informace pro popis polygonů vektorové grafiky a také informace potřebné pro hloubkové řazení.

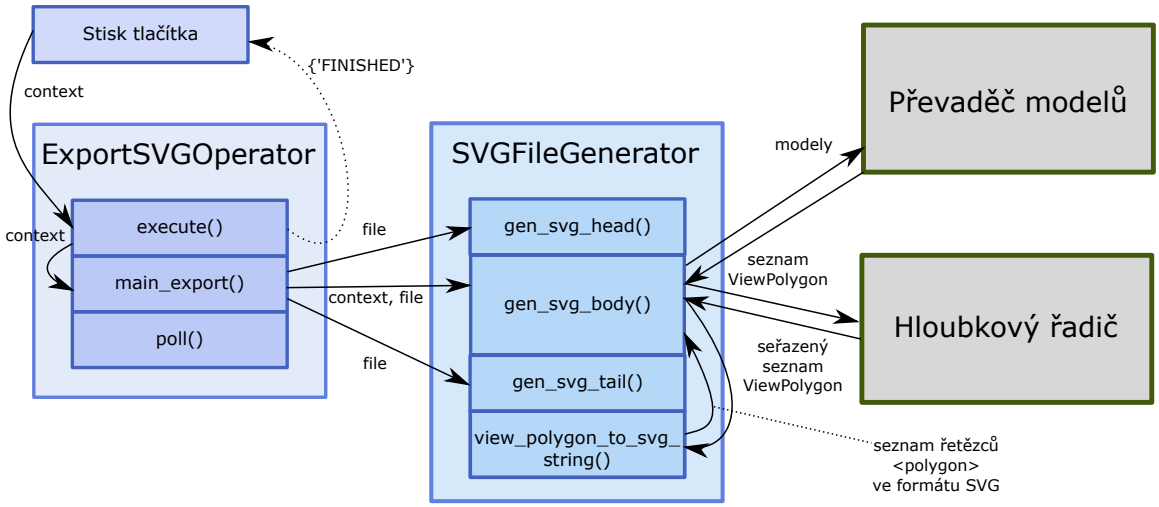
Tato třída tedy obsahuje pouze dvě metody. Tou první je konstruktor, který pro všechny vytvořené instance definuje následující atributy pro popis výsledných polygonů:

- Atribut `verts` pro uložení pozic vrcholů polygonu jakožto seznamu trojic (x, y, z) typu `float`.
- Atribut `depth` pro uložení hloubky polygonu jakožto hodnoty typu `float`, podle které je polygon řazen při primitivním hloubkovém řazení. Způsob výpočtu tohoto atributu se mění podle nastavené heuristiky.
- Atribut `rgb_color` pro uložení výsledné barvy výplně polygonu jakožto trojice (r, g, b) typu `int` s rozmezím hodnot 0-255 podle barevného modelu RGB. Tento atribut lze také při převodu nastavit na hodnotu None, která bude při konverzi na řetězec interpretována jakožto polygon bez výplně.
- Atribut `stroke_color` pro uložení výsledné barvy okraje polygonu jakožto trojice (r, g, b) typu `int` s rozmezím hodnot 0-255 podle barevného modelu RGB. Při běžném nastavení odpovídá jeho hodnota barvě výplně polygonu.
- Atribut `opacity` pro uložení průhlednosti výplně polygonu jakožto hodnoty typu `float` s rozmezím hodnot 0-1.

- Atribut `stroke_opacity` pro uložení průhlednosti okraje polygonu jakožto hodnoty typu `float` s rozmezím hodnot 0-1.
- Atribut `normal` pro uložení normálového vektoru polygonu jakožto trojice (x, y, z) typu `float`.
- Atribut `marked` pro uložení značky polygonu potřebné pro Newellův algoritmus.
- Atribut `bounds` pro uložení bounding boxu (ohraničujícího kvádru) polygonu jakožto šestice (xMin, xMax, yMin, yMax, zMin, zMax) typu `float` specifikující krajní hodnoty souřadnic polygonu na všech osách.

S posledním atributem souvisí druhá metoda této třídy `recalculate_bounds()`, která slouží pro přepočítání souřadnic bounding boxu z nových vrcholů polygonu, například po jeho řezání.

Po předchozím shrnutí základních tříd pluginu lze pro lepší představu ilustrovat roli jednotlivých tříd v procesu převodu diagramem 5.2.



Obrázek 5.2: Proces převodu z pohledu volání hlavních tříd. Po stisknutí tlačítka a volání metod operátoru je převod řízen především hlavní metodou generátoru souboru SVG, `gen_svg_body()`, která pro převod využívá převaděč modelů a hloubkový řadič a generuje poté z výsledných polygonů tělo souboru. Celý převod je ukončen navrácením hodnoty `{"FINISHED"}` z metody obsluhující stisknutí tlačítka.

5.3 Převaděč modelů a související třídy

Po představení základních tříd pluginu i procesu převodu zbývá zmínit třídy se specifičtějším zaměřením souvisejícím s jednotlivými kroky převodu. Tím prvním jsou třídy starající se o konverzi modelu na výsledné polygony.

Převaděč modelů (třída MeshConverter)

Tuto třídu tvoří několik statických metod, které slouží pro převod objektu ve scéně na seznam instancí třídy `ViewPolygon` pro další zpracování. Základní metodou tohoto procesu

je metoda `mesh_to_view_polygons()`, ve které je převod od objektu až po seznam polygonů prováděn.

Metoda nejdříve vytváří kopii meshe daného objektu, kterou následně transformuje z lokálních souřadnic do souřadnic scény. Pro každou plochu meshe poté transformuje i jeho normálu, zkontroluje natočení plochy pomocí metody `is_backface()` a případně plochu vynechá podle principu Backface culling. Plochy, které nejsou tímto způsobem eliminovány jsou dále předávány metodě `mesh_face_to_view_polygon()` pro konverzi na instance třídy `ViewPolygon`. Z těchto získaných instancí je vytvořen seznam, který je poté touto metodou vracen jakožto reprezentace objektu na výsledném obrázku.

Metoda `mesh_face_to_view_polygon()` převádí plochu tak, že nejdříve zkontroluje pozice všech vrcholů. Pokud některý z nich leží za kamerou, je plocha řezána rovinou kamery pomocí metody třídy `ViewPortClipping`, která je zmíněna v další sekci. Zbylé vrcholy jsou pomocnou funkcí Blender API `location_3d_to_region_2d()` a kontextovými proměnnými převedeny ze souřadnic scény na souřadnice pohledu kamery. Seznam těchto vrcholů je dále předáván opět metodě třídy `ViewPortClipping` pro ořez okraji okna, která zpět vrací nyní už výsledný seznam vrcholů polygonu. Následně probíhá určení barvy výsledného polygonu pomocí metody `get_face_color()`, které je předán také materiál plochy, pokud jí byl nějaký přidělen. Po určení všech těchto vlastností výsledného polygonu je vytvořena příslušná instance třídy `ViewPolygon`, která je zároveň návratovou hodnotou této metody.

Metoda `is_backface()` pracuje na principu popsaném v podkapitole teoretické části 3.3 rozebírající metodu Backface culling, jedná se tedy pouze o skalární součin dvou vektorů.

Metoda `get_face_color()` pracuje na principu popsaném v podkapitole teoretické části 3.5 rozebírající stínování. Tato metoda podle nastavení podporuje výpočet výsledné barvy jak pomocí primitivního určování odstínů šedi popsaného rovnicemi 3.1 a 3.2 zmíněné podkapitoly, tak pomocí Lambertova modelu a okolního světla popsaného rovnicemi 3.9 a 3.12 zmíněné podkapitoly.

Ořezávání polygonů (třída ViewPortClipping)

Tato třída je tvořena opět několika statickými metodami usnadňujícími ořezávání polygonů. Její hlavní metoda `clip_2d_polygon()` nejdříve zkontroluje, zdali alespoň nějaká část polygonu leží v okně. Pokud ne, je polygon vynechán. Pokud naopak všechny vrcholy polygonu leží v okně, není prováděno řezání. V případě, kdy nenastala ani jedna situace, je polygon řezán metodou `clip_to_boundary()`.

Tato metoda provádí řezání polygonu hranicemi okna podle algoritmu Sutherland-Hodgman, který byl popsán v teoretické sekci 3.4, viz obrázek 3.5.

Poslední z hlavních metod třídy je metoda pro ořez částí polygonů nacházejících se v prostoru za kamerou, `clip_to_front()`. Tato metoda jednoduše rozdělí polygon na dvě části pomocí roviny, na které leží kamera, kolmé ke směru kamery. Část v poloprostoru za kamerou je vynechána a je vrácena pouze část přední. Dělení je prováděno pomocnou metodou třídy `DepthSorter`, která se týká hloubkového řazení a je zmíněna v další sekci.

Kromě těchto metod obsahuje třída některé pomocné matematické metody, a to metodu `is_inside()` pro určení, zdali se bod nachází v mezích okna a metody `intersect_on_x()`, `intersect_on_y()` a `intersect_on_z()` pro určení průsečíku přímky definované dvěma vrcholy a roviny definované hodnotou souřadnice na ose x, y, nebo z v prostoru. Zjištění těchto průsečíků je podstatné pro řezání mezemi okna, neboť tyto průsečíky se stávají novými vrcholy ořezaných polygonů. Výpočet průsečíků využívá základní poznatky analytické geometrie týkající se směrnic přímky.

5.4 Hloubkový řadič a související třídy

Po bližším seznámení se s procesem převodu objektu na polygony už zbývá pouze část zabývající se jejich seřazením. Tu tvoří hlavní třída pro řazení společně s několika dalšími třídami sloužících pro reprezentaci pomocných datových struktur.

Hloubkový řadič (třída DepthSorter)

Tato třída je v podstatě knihovnou pro hloubkové řazení. Obsahuje větší množství pomocných metod, které slouží k řazení, určování vzájemné pozice rovin, polygonů a vrcholů v prostoru, hledání konfliktů a řezání konfliktních polygonů. Podstatná část této problematiky a principy jednotlivých metod byly do hloubky popsány v podkapitole 3.6. Tato sekce je tedy pouze stručný souhrn implementovaných metod.

Hlavními metodami této třídy jsou metody zprostředkovávající řazení podle jednotlivých zmíněných principů. Metoda `depth_sort()` byla tou původní a je určena k nejprimitivnějšímu způsobu řazení, tedy obyčejné seřazení seznamu polygonů na základě jejich vlastnosti `depth`. Podobně pracuje metoda `depth_sort_bb_depth()`, která je určena k primitivnímu způsobu řazení podle bounding boxů polygonů podporujícího dříve zmíněné heuristiky. Poslední z těchto metod jsou `depth_sort_Newell()`, která zajišťuje proces řezání podle Newellova algoritmu, a `depth_sort_bsp()`, která zajišťuje proces řezání a řazení pomocí BSP stromu.

Mezi další podstatné metody patří dvojice metod pro detekci a eliminaci konfliktů. První je metoda `in_conflict()`, která pro dva polygony v prostoru určí, zdali jsou konfliktní na základě podmínek uvedených v sekci o detekci a eliminaci konfliktů podkapitoly 3.6. Ověření poslední podmínky týkající se kolize 2D projekce polygonů na výsledném obrázku není až tak triviální, jak by se na první pohled mohlo zdát. Proto pro tento účel byla využita funkce knihovny Shapely[1]. Po nalezení konfliktu přichází na řadu metoda `cut_conflicting()` pro jeho eliminaci, která rozřeže rovinou prvního polygonu druhý polygon a vrátí jeho výsledné fragmenty.

U těchto metod nastal problém s přesností kontroly kolize a přesností řezu. Protože souřadnice polygonů jsou hodnoty typu `float` a ne celočíselné hodnoty, mohly občas nastat situace, kdy kvůli různé přesnosti metod pro kontrolu konfliktu a pro řezání byly nové fragmenty stále v konfliktu, i přestože konflikt měl být řezem odstraněn. To samozřejmě vedlo k nekonečnému zacyklení, kdy byl fragment stále řezán na menší a menší části, protože metoda pro detekci stále nacházela konflikt.

Tento problém byl vyřešen snížením přesnosti metody pro řezání, ve které byla zavedena určitá „rezerva“. Namísto toho, aby nové vrcholy fragmentů vzniklé řezem polygonu rovinou ležely přesně na této rovině, budou ležet těsně za rovinou v případě zadního fragmentu a těsně před rovinou v případě předního fragmentu.

Výše zmíněné metody často využívají pomocných metod pro určování vzájemných pozic geometrických primitiv v prostoru.

Mezi tyto metody patří `relative_pos()` a `relative_pos_bool()` pro zjištění vzájemné pozice polygonu a roviny (zjištění zdali je polygon před rovinou, v konfliktu s rovinou, za rovinou, v případě druhé metody zjištění zdali je nekonfliktní polygon před nebo za rovinou).

`Metody vert_relative_pos()` a `vert_relative_pos_bool()` slouží ke zjištění vzájemné pozice vrcholu a roviny (zdali je bod před nebo za rovinou, v případě první metody

[1] Python projekt Shapely: https://pypi.org/project/Shapely/

je implementována také určitá mez vzdálenosti od roviny, v rámci které je vrchol považován za bod ležící v rovině).

Metoda `is_fragment()` slouží pro primitivní určení, zdali výsledný polygon vzniklý po řezání polygonů není extrémně malý a tím pádem irelevantní pro výsledný obraz. Takové fragmenty jsou ve výsledném obraze vynechány.

Metoda `correct_normals()` je použita v rámci některých způsobů řazení pro otočení normálových vektorů všech odvrácených řazených ploch takovým způsobem, aby každá plocha byla přivrácená vzhledem ke kameře. Tento krok je prováděn z toho důvodu, aby vždy platilo, že pokud polygon B leží za polygonem A, znamená to, že polygon B leží za polygonem A také z pohledu kamery. Kdyby nebyla provedena korekce normálových vektorů a polygony A i B by byly odvráceny od kamery, polygon A by mohl ležet za rovinou polygonu B i přesto, že je celý blíže ke kameře a z pohledu kamery před ním.

Na závěr lze zmínit pomocné metody pro implementaci Newellova algoritmu a tvorby BSP stromu, kam patří `newell_half()` pro půlení polygonů, `newell_insert_fragments()` pro vkládání fragmentů polygonu zpět do seznamu řazených polygonů, `p_obscures_q()` a `p_behind_q()` pro určování vzájemných poloh polygonů. Mezi pomocné metody BSP patří `bsp_partition()` pro rozdělení uzlu stromu a `bsp_tree_to_view_polygons()` pro převod vytvořeného BSP stromu zpět na seznam polygonů.

BSP strom (třída BSPNode)

Tato třída slouží pouze k reprezentaci uzlu BSP stromu, její instance obsahují pouze odkazy na své synovské uzly, informaci o tom, zdali se jedná o listový uzel a také seznam polygonů tohoto uzlu. Třída neobsahuje žádné metody, o práci s BSP stromem se starají pouze metody třídy hloubkového řadiče.

Octree (třídy Octree a OctreeNode)

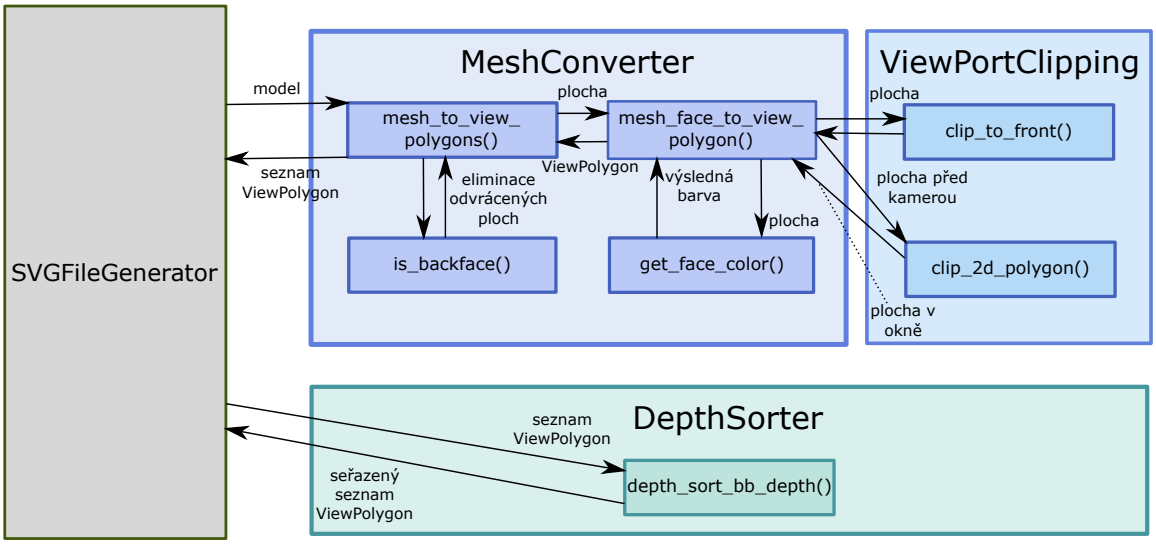
Tyto třídy slouží k reprezentaci datové struktury typu octree. Třída `Octree` a její instance slouží pro reprezentaci stromu jakožto celku a obsahují metody určené k práci se stromem. Mezi tyto metody patří konstruktor stromu a metoda `insert_polygon()` pro vložení polygonu do stromu. Dále metoda pro získání seznamu všech uzlů stromu `get_all_nodes()` využívaná při hledání konfliktů. Metoda `get_resolved_polygons()` poté slouží k získání všech polygonů stromu, které už nejsou v konfliktu s žádným jiným polygonem.

Samotné řešení konfliktů lze provést metodou `resolve_conflicts()`, která seřadí jednotlivé uzly od nejvzdálenějších po nejbližší ke kořeni v rámci stromu a pro všechny provede eliminaci konfliktů metodou třídy `OctreeNode` zmíněné dále. Před tímto procesem je však také volána metoda pro optimalizaci `compress_tree()`, která zjednoduší strom smazáním prázdných listových uzlů, které vznikly při přidávání polygonů.

Instance třídy `OctreeNode` tedy reprezentuje jednotlivé uzly stromu a obsahuje několik vlastních metod. Kromě konstruktoru obsahuje metodu `add_polygon()` pro přidávání polygonů do uzlu. Metoda `subdivide()` slouží pro rozdělení uzlu na 8 menších synovských uzlů a kontrole, zdali nějaké polygony tohoto uzlu nemohou být do těchto synovských přiřazeny. Tuto kontrolu zajišťuje metoda `contains_polygon()`, která ověří, zdali bounding box daného polygonu nepřesahuje hranice uzlu a tím pádem zdali se do uzlu vleze.

Pro samotné řešení konfliktů v rámci uzlu slouží metoda `resolve_node()`, která provede detekci a eliminaci konfliktů pro všechny polygony uzlu. Tato detekce a eliminace probíhá na principu algoritmu 3 popsaného v sekci týkající se řazení s pomocí octree.

Způsob celkového převodu objektů scény na seznam polygonů z pohledu použití jednotlivých tříd a hlavních metod lze znázornit diagramem 5.3 (za použití primitivního řazení bez Octree, BSP a podobně, v opačném případě jsou volány příslušné metody podle zvoleného typu řazení).



Obrázek 5.3: Popis implementace převodu objektu na seznam polygonů. Převod řízený generátorem souboru SVG nejdříve využívá hlavní metodu třídy pro převod modelů na výsledné polygony `mesh_to_view_polygons()`, která dále kromě metod stejné třídy k převodu využívá také třídu pro ořez polygonů. Poté generátor souboru SVG seznam výsledných polygonů řadí pomocí metody třídy pro hloubkové řazení.

5.5 Implementace z pohledu sekvence kroků převodu

Po bližším popisu všech tříd podílejících se na chodu pluginu byla na závěr implementační části této práce přidána následující podkapitola, která slouží k finálnímu stručnému shrnutí procesu převodu z pohledu posloupnosti jednotlivých kroků a volaných metod. Tato sekce neobsahuje oproti předchozím podkapitolám téměř žádné nové informace, slouží tak spíše pro spojení předchozích podkapitol do jednoho souvislého souhrnu a pro zřetelnější a koherentnější pohled na celkovou implementaci převodu. Ten je také reprezentován posledním obrázkem této kapitoly 5.4.

1. Před zahájením převodu uživatel vybere ve scéně objekty pro převod a upřesní nastavení pluginu.

2. Převod je zahájen stisknutím tlačítka pro export. Tato událost je ošetřena metodou `execute()` třídy `ExportSVGOperator`, která je po stisknutí volána a jako parametr je jí předán odkaz na veškeré kontextové proměnné Blenderu. Tento odkaz je předáván většině následujících metod. Samotná metoda `execute()` volá metodu `main_export()` ze stejné třídy.

3. Metoda `main_export()` zkontroluje nastavení, vybrané světlo a objekty pro převod. Poté otevře soubor a pomocí metody `gen_svg_head()` ze třídy `SVGFileGenerator`

generuje hlavičku výstupního souboru. Poté začne generovat tělo souboru voláním metody `gen_svg_body()` opět ze stejné třídy.

4. Metoda `gen_svg_body()` nejdříve spočítá pozici a směr kamery, pozici světla a poté pro všechny jednotlivé převáděné objekty volá zvlášť metodu pro převod na výsledné polygony `mesh_to_view_polygons()` třídy `MeshConverter`.

5. Metoda `mesh_to_view_polygons()` nejdříve vytvoří kopii meshe a transformuje ji do souřadnic scény. Dále podle nastavení pluginu může kontrolovat jednotlivé plochy meshe pro vynechání principem Backface culling pomocí metody `is_backface()` ze třídy `MeshConverter`. Zbylé plochy jsou předány metodě `mesh_face_to_view_polygon()` ze stejné třídy.

6. Metoda `mesh_face_to_view_polygon()` nejdříve ořezává plochu s pomocí metody `clip_to_front()` třídy `ViewPortClipping` tak, aby žádná její část neležela v poloprostoru za kamerou. Poté převede souřadnice plochy z prostoru scény na souřadnice výsledného obrazu. Tuto výslednou plochu ještě dále ořezává pomocí metody `clip_to_2d()` stejné třídy `ViewPortClipping` tak, aby její žádná část neležela mimo meze okna. Po těchto ořezech metoda dále kontroluje materiál plochy a vypočítává její výslednou barvu pomocí metody `get_face_color()` třídy `MeshConverter`. Ze zjištěných výsledných vlastností je vytvořena nová instance třídy `ViewPolygon` reprezentující výsledný polygon, která je vrácena zpět metodě `mesh_to_view_polygons()`, odkud byla tato metoda volána.

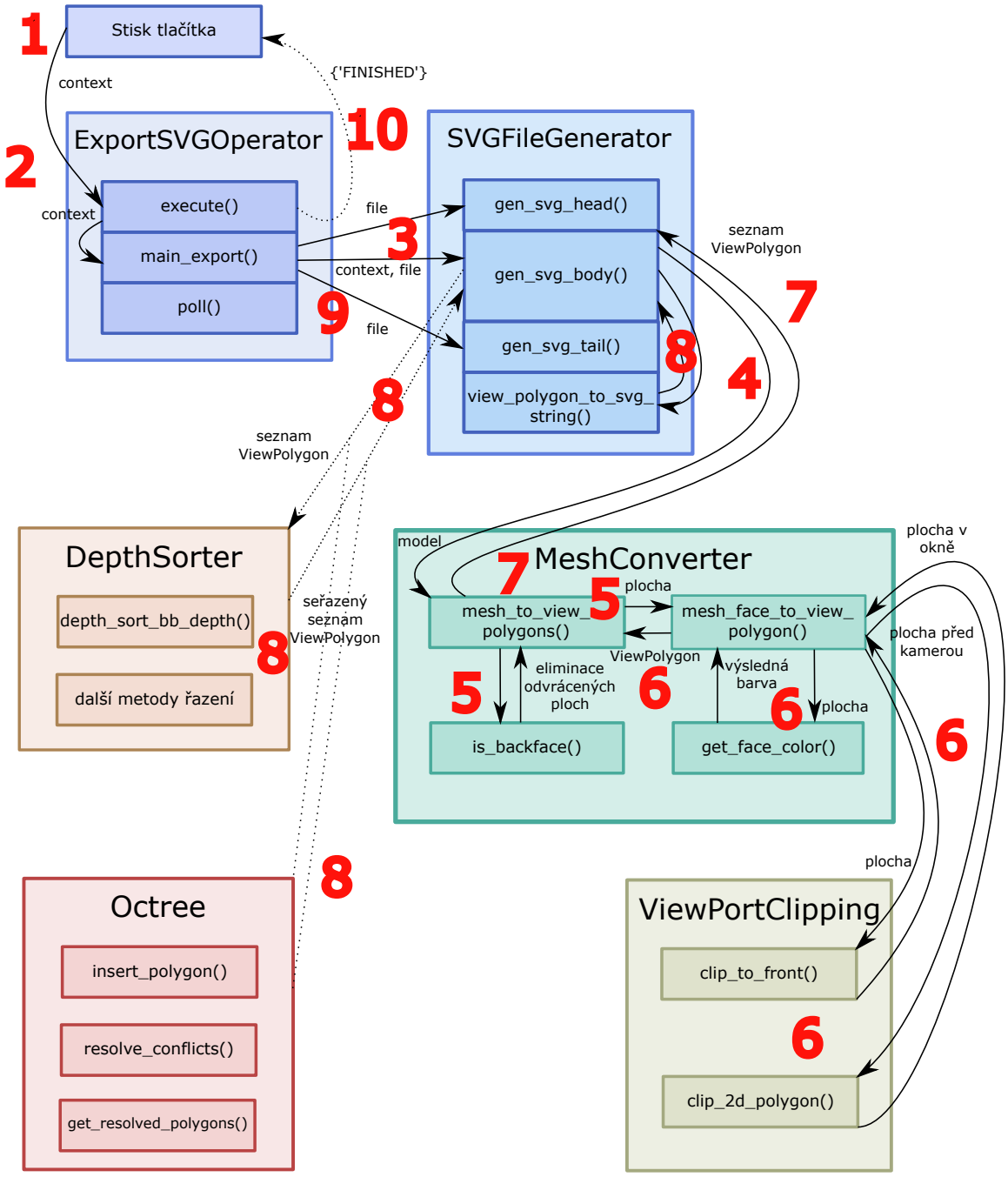
7. Metoda `mesh_to_view_polygons()` poté ze všech těchto vrácených polygonů vytváří seznam. Na závěr tato metoda uvolní kopii meshe z paměti a vrací seznam polygonů zpět metodě `gen_svg_body()`, odkud byla tato metoda volána.

8. Metoda `gen_svg_body()` poté ze všech těchto vrácených seznamů polygonů vytváří jeden dlouhý seznam všech výsledných polygonů obrazu. Tento seznam poté na základě nastavení hloubkově řadí, a to buďto přímo primitivním seřazením hloubky podle zvolené heuristiky pomocí metody `depth_sort_bb_depth()` třídy `DepthSorter`, a nebo postupným hledáním a eliminací konfliktů a následným řazením. V tomto případě je provedena korekce normál a lze provést řezání a řazení dle nastavení buďto metodou uložení do octree, Newellovým algoritmem, a nebo použitím BSP stromu. Výsledný seznam bez konfliktů u prvních dvou metod je poté také řazen primitivním seřazením hloubky podle zvolené heuristiky metodou `depth_sort_bb_depth()`. Jako poslední krok metoda pro každý polygon volá metodu `view_polygon_to_svg_string()` ze třídy `SVGFileGenerator` pro převod polygonu na řetězec ve formátu SVG, který zapíše do souboru. Poté vrací řízení metodě `main_export()`, odkud byla tato metoda volána.

9. Metoda `main_export()` po dokončení předchozí metody volá `gen_svg_tail()` ze třídy `SVGFileGenerator` pro generování konce souboru. Poté výsledný soubor zavře a vrací řízení metodě pro obsluhu tlačítka `execute()`, odkud byla tato metoda volána.

10. Metoda `execute()` končí, čímž je ukončen proces převodu a obsluha tlačítka.

Výše zmíněnou posloupnost lze doplnit diagramem 5.4, v němž jsou vyznačeny také místa s jednotlivými kroky (diagram popisuje pouze primitivní řazení a okrajově znázorňuje eliminaci konfliktů pomocí octree, ostatní metody řazení nejsou uvažovány).



Obrázek 5.4: Souhrnný diagram implementace převodu. Uživatel vybere modely a nastavení převodu, pro zahájení převodu stiskne tlačítko (1 a 2). Poté je vytvořen soubor, vygenerována hlavička a je zahájeno generování těla (3). Jednotlivé modely jsou předány třídě pro převod modelů (4), která provede transformaci na souřadnice scény, Backface culling a začne převádět jednotlivé plochy modelu (5). Převod ploch zahrnuje ořez částí za kamerou a částí mimo okno ve třídě pro ořez polygonů, převod na souřadnice obrazu a výpočet barvy (6). Seznam převedených polygonů je zpět navrácen metodě pro generování těla souboru (7). Ta tento seznam s pomocí hloubkového řadiče seřadí a převede na řetězce formátu SVG, které zapíše do souboru (8). Po generování těla je generován konec souboru a soubor je zavřen (9). Převod končí návratem z metody pro obsluhu stisknutí tlačítka (10).

Kapitola 6

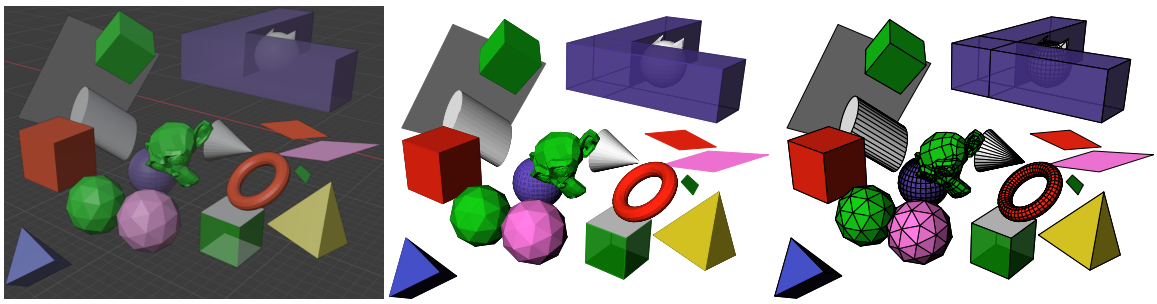
Dosažené výsledky

Následující kapitola slouží k představení několika příkladů výsledků, kterých lze dosáhnout za použití tohoto pluginu v Blenderu pro různé typy modelů a s různými kombinacemi nastavení.

Případná měření v této kapitole probíhala na sestavě s CPU AMD Ryzen 5 3400G a GPU NVIDIA GeForce GTX 1660 SUPER 6GB. Zápis polygonů probíhal na pevný disk s rychlostí 7200 otáček za minutu. Při testování byla použita verze Blenderu 2.90.0 na operačním systému Windows 10 64-bit.

Jako první příklad dosažených výsledků lze uvést část obrázků doposud použitých v tomto textu. Pro generování základů některých obrázků popisujících situaci v prostoru byly použity 3D modely v Blenderu společně s tímto pluginem. Základy těchto obrázků už poté byly pouze upraveny například změnou barvy nebo přidáním textu a křivek. Jedná se například o obrázky 3.1, 3.2, 3.3, 3.5, 3.10 nebo 3.11.

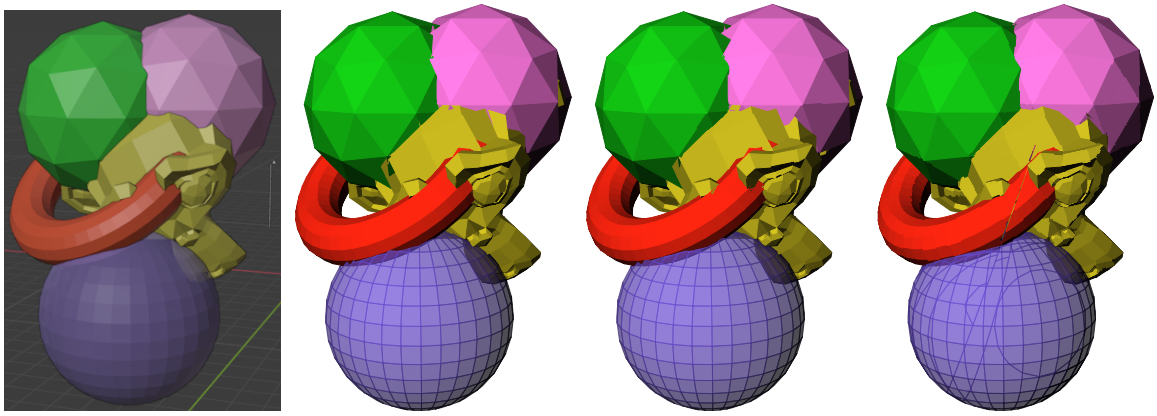
Další příklad výsledků dosažitelných při různých kombinacích nastavení je představen krátce na obrázku 6.1 a podrobněji v příloze A.1.



Obrázek 6.1: Převáděná scéna (vlevo) obsahuje 2376 polygonů. Zbylé obrázky zobrazují různé výsledky s různými kombinacemi nastavení. Při převodu nebyla použita metoda Backface culling a řezání konfliktních polygonů. Celkový převod ve všech případech trval 250 až 400 milisekund. Více obrázků ve větší velikosti lze najít v příloze A.1.

Jako další příklad dosažených výsledků lze uvést srovnání výsledných obrazů, časů a počtu řezů jednotlivých metod pro řezání složitější scény zaklíněných objektů na obrázku 6.2 a v tabulce 6.1. Toto srovnání slouží spíše jako ilustrační příklad, výsledky jednotlivých metod a poměry jejich rychlostí se můžou značně lišit v závislosti na velmi malých detailech převáděných modelů, které nelze jednoduše zobecnit a testovat. Například u metody BSP může přidání pouze několika pár malých polygonů do scény ovlivnit výběr prvního dělícího

polygonu, čímž se kompletně změní struktura BSP stromu, počet iterací, počet výsledných polygonů a celkový čas převodu. Jeden odvoditelný závěr je však fakt, že převod metodou BSP má vždy za následek větší počet výsledných polygonů a zbytečných řezů oproti ostatním dvěma metodám.



Obrázek 6.2: Převáděná scéna (vlevo) obsahuje 5 vzájemně zaklíněných objektů o celkem 934 přivrácených polygonech. Metoda Backface culling byla při převodu použita. První ze tří dalších obrázků vykresluje výsledek metody řezání pomocí octree, druhý výsledek metody řezání Newellovým algoritmem a třetí výsledek metody řezání a řazení pomocí BSP stromu.

	Octree	Newell	BSP strom
Převod a ořez ploch na výsledné polygony	0,105 s	0,105 s	0,105 s
Sestavení octree	0,015 s	–	–
Eliminace konfliktů a řazení	0,56 s	–	–
Newell řezání a řazení	–	1,455 s	–
Sestavení BSP stromu	–	–	1,116 s
Převod BSP stromu na seznam polygonů	–	–	0,008 s
Zápis polygonů	0,031 s	0,026 s	0,08 s
Celkem polygonů v souboru	**1487** (226 kB)	**1587** (245 kB)	**4443** (678 kB)
Celkový čas	**0,711 s**	**1,586 s**	**1,309 s**

Tabulka 6.1: Srovnání jednotlivých metod pro řezání a řazení polygonů použitých na obrázcích 6.2. Původní počet polygonů ve scéně byl 934.

Ačkoliv podnětem této práce byl především převod jednoduchých a pravidelných modelů určených pro ilustrace, na závěr lze jakožto experiment uvést příklad převodu komplexnějšího a rozsáhlejšího modelu postavy z počítačové hry. Model obsahuje celkem 7982 polygonů, převod byl proveden s aktivní metodou Backface culling a s primitivním řazením podle hloubky bez detekce konfliktů. Převod trval celkem 658 milisekund. Kvůli velikosti výsledného obrázku lze najít výsledek převodu na obrázcích v příloze A.2 a A.3. Časová náročnost konverze modelu při primitivním řazení podle hloubky roste lineárně s počtem polygonů modelu (nejvíce času při této konfiguraci totiž zabírá převod jednotlivých ploch modelů na výsledné polygony a zápis jednotlivých polygonů do souboru).

Kapitola 7

Závěr

Cílem práce bylo vytvořit plugin pro nástroj Blender, který umožňuje v jeho prostředí převádět pohled na scénu s 3D modely na obrázky vektorové grafiky. V rámci této práce byly dle jednotlivých bodů zadání postupně nastudovány základy modelovacího programu Blender a poté také základy skriptování v Blender Python API. Dále byl plugin i samotný proces pro převod modelů na formát vektorové grafiky SVG navržen a poté také implementován. Výsledek funkčnosti implementace byl demonstrován na různých typech modelů s různými kombinacemi nastavení. Výsledný plugin byl zdokumentován a zveřejněn pro použití jinými uživateli[1]. Pro demonstraci výsledků této práce bylo také vytvořeno krátké video[2].

Ačkoliv se podařilo jednotlivé body zadání v jejich původním znění splnit, je nutno podotknout, že se nepodařilo navrhnout proces převodu modelů takovým způsobem, aby byl zaručen korektní obraz za všech situací u všech typů modelů a jejich konfigurací. To bylo zapříčiněno poměrně složitější problematikou hloubkového řazení, která je oproti běžným situacím ještě o něco náročnější při práci s vektorovou grafikou a formátem SVG, neboť kombinace podmínek pro takový výstup eliminuje možnost použití většiny standardních algoritmů.

I přesto byl však navržen a implementován algoritmus pro nalezení a řezání konfliktních polygonů, který značně zlepšuje výslednou kvalitu obrazu v mnoha situacích, a to v řádu sekund pro stovky až tisíce polygonů. Obecně lze říct, že plugin dosahuje lepších výsledků v situacích, kdy jsou modely/scény složeny z většího množství menších ploch, a horších výsledků v situacích, kdy jsou modely/scény složeny z kombinací velmi velkých a malých ploch, které se z pohledu kamery vzájemně překrývají.

Výsledný plugin byl zveřejněn a lze jej volně stáhnout a nainstalovat do nástroje Blender. Plugin byl vyvinut pro verzi Blender 2.90 a v základu umožňuje uživatelům převádět pohled na vybrané modely scény v Blenderu na soubory vektorové grafiky ve formátu SVG a tím usnadnit vytváření ilustrací prostorových scén v případech, kde není vhodná rastrová grafika.

Tento převod je zároveň pluginem rozšířen o několik možností, kterými ho lze modifikovat. Podporován je výběr více modelů scény zároveň, nastavení barvy a průhlednosti modelů pomocí difuzní barvy jejich materiálu, nastavení typu a pozice zdroje světla a barvy světla pro stínování modelů, nastavení okolního světla pro snížení kontrastu vytvořeného primitivním stínováním, mód pro vykreslení pouze odstínů šedi, nastavení vlastních barev okrajů a vlastních barev výplní polygonů, nastavení použitých algoritmů a heuristik

[1] Publikovaný plugin online: https://github.com/Craszh/BlenderModelToSVG

[2] Video demonstrující použití pluginu dostupné online: https://youtu.be/Q0dkwK7mfqo

pro řešení problému viditelnosti a také nastavení některých vlastností výsledného souboru. V neposlední řadě je také podporován jak perspektivní, tak ortografický pohled na scénu.

Protože se jedná o plugin pracující ve velmi versatilním prostředí Blenderu, jehož rozhraní poskytuje širokou škálu nástrojů a tříd pro skriptování, lze uvažovat nad mnoha možnými budoucími rozšířeními. Mezi ně patří například rozšíření o možnosti převodu nejen 3D modelů, ale také křivek a textu, pro které Blender definuje speciální typy objektů. To by umožnilo snadnější tvorbu vektorových diagramů přímo v prostředí Blenderu bez nutnosti přidávat popisky v editoru vektorové grafiky. U křivek by bylo třeba řešit problém převodu různých podtypů křivek a problém vzájemného řazení křivek a modelů, popřípadě tento problém zanedbat a vykreslovat křivky na samostatné nejvyšší vrstvě, což by byl většinou u diagramů záměr. U textu by bylo potřeba brát ohled na určování jeho pozice ve výsledném obraze, neboť text by byl ve 3D scéně různě otočen a jeho výsledná pozice ve 2D obraze by nemusela odpovídat pozici očekávané. Dalším rozšířením by mohla být možnost pro uživatele vkládat vlastní skripty jazyka Python například pro řazení nebo stínování, kterými by byl uživatel schopen definovat vlastní funkce pro určování pořadí nebo barev polygonů, které by byly volány pluginem. Jednalo by se tak vlastně o primitivní rozhraní pro programování pluginu. Dalším rozšířením by samozřejmě bylo také vylepšení procesu řazení, které v současném stavu nefunguje s absolutní přesností ve všech konfiguracích. Z méně konkrétního hlediska lze také uvažovat nad více módy pro stínování modelů nebo nad více specifickými a speciálními módy vykreslení s různými efekty.

Literatura

[1] Beneš, B., Sochor, J., Felkel, P. a Žára, J. *Moderní počítačová grafika*. 2. vyd. Computer Press, 2004. ISBN 80-251-0454-0.

[2] Cohen Or, D., Chrysanthou, Y. L., Silva, C. T. a Durand, F. A survey of visibility for walkthrough applications. *IEEE Transactions on Visualization and Computer Graphics*. IEEE Computer Society. 2003, sv. 9, č. 3, s. 412–431. ISSN 1077-2626.

[3] Fuchs, H., Kedem, Z. M. a Naylor, B. F. On Visible Surface Generation by a Priori Tree Structures. In: *Proceedings of the 7th Annual Conference on Computer Graphics and Interactive Techniques*. Association for Computing Machinery, 1980, s. 124–133. SIGGRAPH '80. ISBN 978-0-89791-021-7.

[4] Hughes, J. F., Dam, A. van, McGuire, M., Sklar, D. F., Foley, J. D. et al. *Computer Graphics: Principles and Practice*. 3. vyd. Addison-Wesley, 2013. ISBN 978-0-321-39952-6.

[5] Lengyel, E. *Mathematics for 3D Game Programming and Computer Graphics*. 3. vyd. Course Technology Press, 2011. ISBN 978-1-4354-5886-4.

[6] Newell, M., Newell, R. a Sancha, T. A Solution to the Hidden Surface Problem. In: *Proceedings of the ACM Annual Conference - Volume 1*. Association for Computing Machinery, 1972, s. 443–450. ACM '72. ISBN 978-1-4503-7491-0.

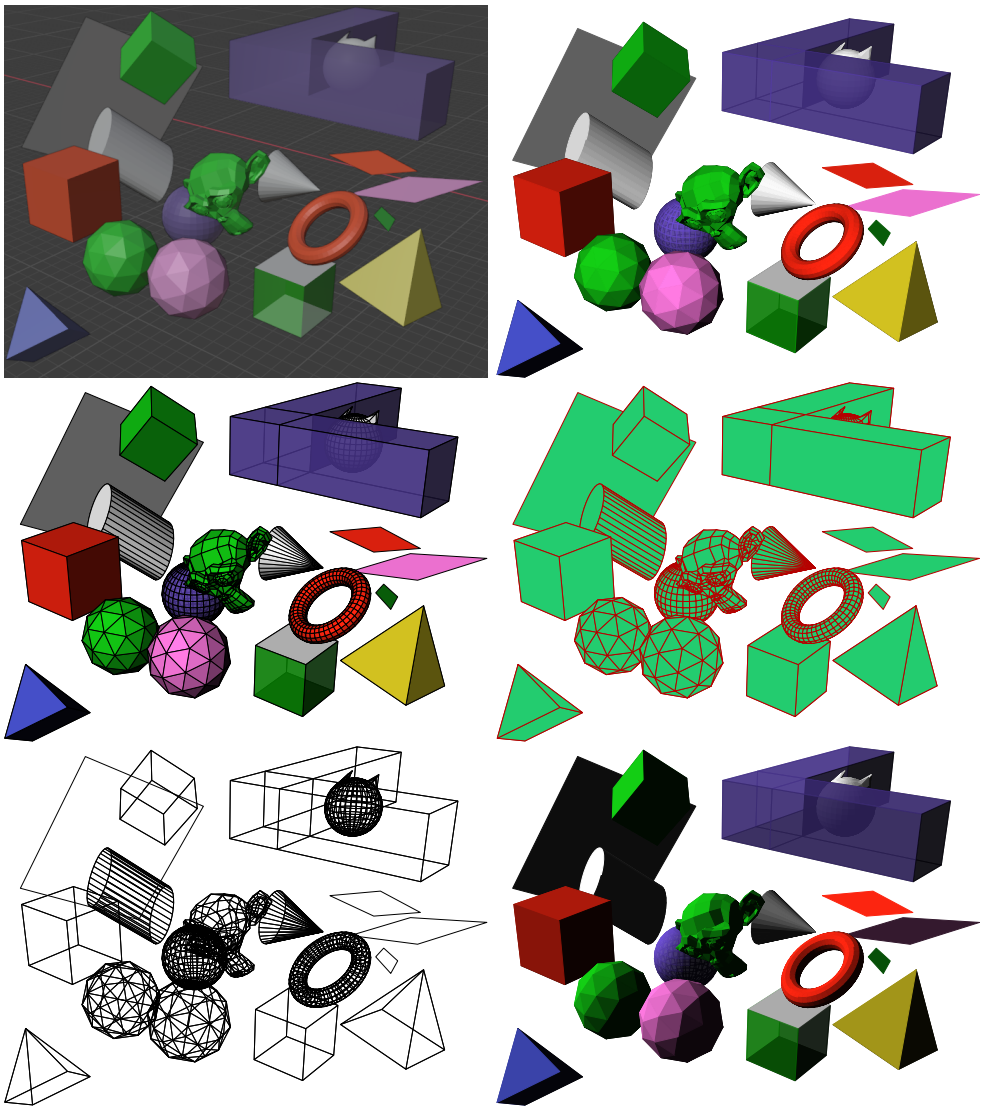
[7] Sutherland, I. E. a Hodgman, G. W. Reentrant Polygon Clipping. *Communications of the ACM*. Association for Computing Machinery. 1974, sv. 17, č. 1, s. 32–42. ISSN 0001-0782.

[8] Sutherland, I. E., Sproull, R. F. a Schumacker, R. A. A Characterization of Ten Hidden-Surface Algorithms. *ACM Computing Surveys*. Association for Computing Machinery. 1974, sv. 6, č. 1. ISSN 0360-0300.

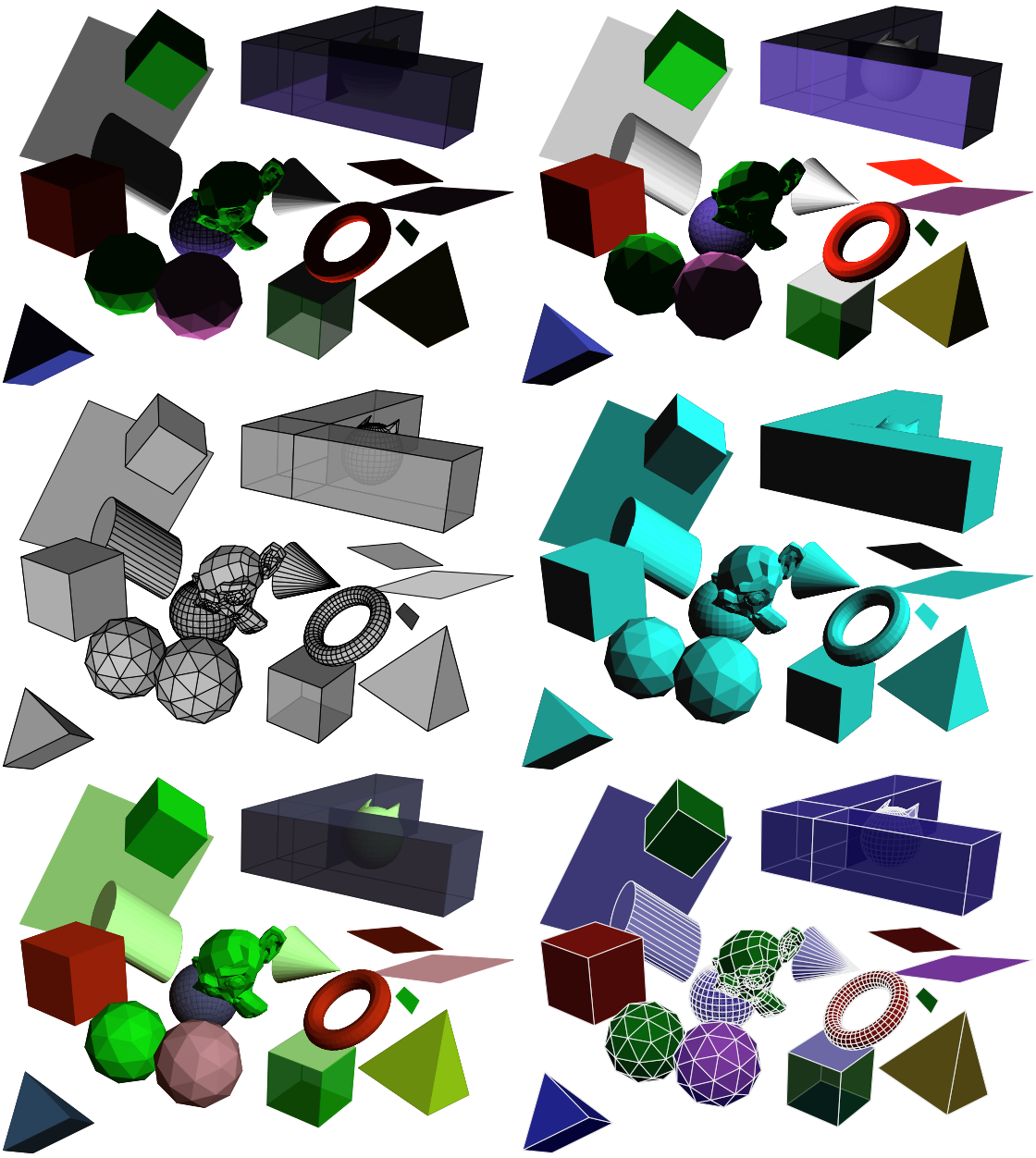
[9] Tunnel, R., Jaggo, J. a Luik, M. *Shading and Lighting* [online]. University of Tartu, 2014 [cit. 2021-03-31]. Dostupné z: https://cglearn.codelight.eu/pub/computer-graphics/shading-and-lighting.

Příloha A

Příklady dosažených výsledků



Obrázek A.1: Převáděná scéna (vlevo nahoře) obsahuje 2376 polygonů. Zbylé obrázky zobrazují různé výsledky s různými kombinacemi nastavení.



Obrázek A.1: Převáděná scéna obsahuje 2376 polygonů. Zbylé obrázky zobrazují různé výsledky s různými kombinacemi nastavení.

Obrázek A.2: Převáděný model postavy z počítačové hry obsahuje celkem 7982 polygonů, zhruba polovina je odvrácených.

Obrázek A.3: Převedený model postavy do vektorové grafiky. Převod s aktivním Backface culling a primitivním řazením podle hloubky trval 658 milisekund.