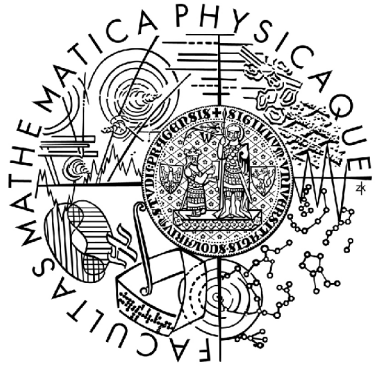
Univerzita Karlova v Praze
Matematicko-fyzikální fakulta

DIPLOMOVÁ PRÁCE



Jan Stuchl

Hardwarová podpora 3D grafiky pro JaGrLib

Katedra softwarového inženýrství
Vedoucí diplomové práce: *RNDr. Josef Pelikán*
Studijní program: *Informatika*

Na tomto místě bych rád poděkoval vedoucímu diplomové práce RNDr. Josefu Pelikánovi za odborné rady a náměty, které zásadním způsobem přispěly k dokončení této práce.

Prohlašuji, že jsem svou diplomovou práci napsal samostatně a výhradně s použitím citovaných pramenů. Souhlasím se zapůjčováním práce.

V Praze dne 16. dubna 2009

Jan Stuchl

Obsah

Název práce: *Hardwarová podpora 3D grafiky pro JaGrLib*

Autor: *Jan Stuchl*

Katedra (ústav): *Katedra softwarového inženýrství*

Vedoucí diplomové práce: *RNDr. Josef Pelikán*

e-mail vedoucího: *Josef.Pelikan@mff.cuni.cz*

Abstrakt: V *současné době je hardwarová podpora grafiky téměř nedílnou součástí všech počítačových platforem. Její zavedení do projektu JaGrLib, který slouží k testování a vylepšování grafických algoritmů a struktur a také k výuce počítačové grafiky, je tedy příhodné vzhledem k jejím účelům.*
Tato práce se zabývá implementací hardwarové akcelerace zejména pro její následné použití v JaGrLib. Je hledáno vhodné propojení mezi knihovnou a samotnou grafickou akcelerací, navrženo rozhraní pro další účely knihovny a moduly nezbytné k základní funkčnosti. Dále jsou také vytvořeny příklady kompozic do knihovny.
Druhá část práce se zabývá zobrazením CSG scény na grafickém akcelerátoru v knihovně JaGrLib a to převedením takové scény do b-rep za použití nového algoritmu. Algoritmus je založen na množinové operaci dvou těles procházením po jejich vrcholech, podobně jako množinová operace dvou polygonů v rovině.

Klíčová slova: *JaGrLib, jogl, hardwarová grafika, převod CSG na b-rep*

Title: *Hardware support of 3D graphics for JaGrLib library*

Author: *Jan Stuchl*

Department: *Department of Software and Computer Science Education*

Supervisor: *RNDr. Josef Pelikán*

Supervisor's e-mail address: *Josef.Pelikan@mff.cuni.cz*

Abstract: *Graphical hardware acceleration support is currently integral part of all computer platforms. Its implementation to the JaGrLib project, which serves for testing and improvement of graphical algorithms and structures as well as for educational purposes, is propitious given its purposes. This thesis deals with the implementation of hardware acceleration especially in the context of JaGrLib.*
The thesis strives for optimal connection between the library and graphical acceleration, the interface for other library tasks has been developed as well as modules needed to achieve basic functionality. Furthermore example library compositions have been developed.
Second part of thesis deals with CSG scene projection using graphical accelerator in JaGrLib by converting such scene into b-rep using new algorithm. The algorithm is based on set operation of two solids by walking their vertices, similar to set operation of two polygons in plane.

Keywords: *JaGrLib, jogl, hardware graphics, CSG to b-rep transformation*

1. Úvod

Hardwarová grafika je v dnešní době již běžná záležitost. Její implementace ve výukové knihovně *JaGrLib*[1] je tedy příhodná a rozšiřuje možnosti jak pro rychlé vykreslování 2D a 3D scén, tak pro samotné programování GPU[1]. Dosavadní verze projektu neobsahovala možnosti rychlého přepočítávání a vykreslování polygonů a možnosti výuky a práce v tomto projektu byly omezené. Zavedením hardwarové grafiky do projektu tedy vzniká mnoho nových možností, jak rozšiřovat vědomosti a vyvíjet nové metody či algoritmy, případně ukazovat schopnosti právě grafických akcelerátorů.

1.1 Cíl práce

Cílem této práce je zhodnotit možnosti a implementovat hardwarovou podporu ve výukové knihovně *JaGrLib* a to v souladu s projektovou ideou.

Základem je tedy volba a zavedení jedné z grafických knihoven pro hardwarovou akceleraci, a její spojení s ostatními moduly knihovny *JaGrLib*[1]. Vzhledem k tomu, že jedna z nejčastěji používaných reprezentací 3D scény je *CSG*[2], je součástí práce také zhodnocení a implementace takové scény. Je použita metoda převodu této reprezentace do reprezentace *b-rep*[3] a její následné vykreslení.

1.2 Rozsah práce

Práce je rozdělena do dvou hlavních částí.

První část se zabývá volbou a zavedením knihovny pro hardwarovou podporu grafiky v knihovně *JaGrLib*. Spočívá v popisu implementace modulů do knihovny tak, aby bylo jejich následné použití co nejvíce otevřené.

Druhá část popisuje jednu z možností zobrazení *CSG* scény a to jejím převodem na polygony, tedy *b-rep* novým algoritmem a jeho následná implementace.

1 Graphics Processing Unit – grafický procesor pro urychlení výpočtů a vykreslování na zobrazovacím zařízení.

2 Constructive solid geometry – modelovací technika pro vytváření 3D scén pomocí jednoduchých těles a množinových operací.

3 Boundary representation – reprezentace scény pomocí hranic – stěn, hran a vrcholů.

2. Hardwarová podpora v knihovně JaGrLib

Tato kapitola se zabývá vysvětlením pojmů zejména v knihovně *JaGrLib*, popisem potřebných částí knihovny *JOGL* a konečně popis zavedení knihovny do projektu – vytvoření nutných modulů a popis jejich použití s příklady. Předpokladem je znalost jazyka *Java*, jehož popis by byl nad rámec této práce. Pro přiblížení uveďme, že je to objektově orientovaný programovací jazyk od společnosti *Sun Microsystems* existující od roku 1995. Jeho výhodou je především v přenositelnosti na různé platformy.

2.1 Knihovna JaGrLib

Projekt *JaGrLib*[1] je framework sloužící pro testování a vylepšování jak algoritmů, tak datových struktur pro počítačovou grafiku, zvlášť pak pro výukové účely. Je napsán v jazyce *Java* od společnosti *Sun Microsystems* a díky tomu některé z prvků, jako např. správa alokované paměti, nevyžadují kontrolu od programátora.

Knihovna je modulární a prvky pro jednotlivé algoritmy či datové struktury jsou koncipovány jako třídy – označované jako **piece**. Jejich propojením přes rozhraní – **plug** – umožňuje pak vytvořit kompozici, která může předvést použitelnost a ukázat výsledky časových a operačních náročností. Každý z modulů pak je omezen na určité typy rozhraní a může být parametrizován. Pro prohlížení a spouštění kompozic slouží program **Skel**, který je součástí knihovny.

Vytvoření nového algoritmu či datové struktury obnáší vytvoření potomka třídy **`cz.cuni.jagrlib.Piece`**, definování jeho parametrů a rozhraní a jeho začlenění do nějaké nové či existující kompozice.

2.1.1 Používané prvky JaGrLib

Pro rozšíření knihovny o hardwarovou akceleraci poslouží jen zlomek z již implementovaných tříd a rozhraní. Pro detailní popis všech vytvořených modulů a rozhraní či popis práce a spouštění slouží dokumentace samotné knihovny [1]. Následuje popis takových modulů, které jsou při vytváření nových modulů pro účely práce nezbytné.

Rozhraní (všechny jsou součástí namespace **`cz.cuni.jagrlib.iface`**):

- **`GraphicsViewer`** – definuje obecné grafické rozhraní k zobrazení výsledků kompozice a rozhraní pro zpracování z vstupních zařízení (klávesnice, myš). Důležitá zde je meto-

da **repaintLoop**, jejíž implementace by měla provádět opakované překreslování okna.

- **Trigger** – umožňuje spouštět nějakou akci metodou **fire**. Obvykle se používá na spouštění algoritmů.
- **Property** – implementace tohoto rozhraní umožňují nastavovat vlastnosti modulů.
- **Worker** – interface pro spouštění kompozic. Přes toto rozhraní jsou kompozice spouštěny pomocí metody **run**.
- **Brep** – definuje metody pro práci s obecnou datovou strukturu scény definovaných pomocí okrajů (*bounds*). Tato struktura je vhodná pro vykreslování pomocí hardwarové akcelerace.
- **InputListener** – umožňuje implementovat reakce na uživatelské vstupy.
- **RTScene** – obecná scéna určená primárně pro ray-tracing. Umožňuje přístup k *CSG* stromu 3D scény.
- **DataFileFormat** – rozhraní pro nahrávání a čtení různých formátů souborů.
- **BitStream** – rozhraní, které umožňuje používat tok dat – *stream* (pro čtení nebo zápis).
- **RTScene** – interface pro scénu určenou pro *ray-tracing* – *CSG*. Využita je metoda pro získání stromu *CSG*.

Moduly (*piece* – jsou součástí namespace **cz.cuni.jagrlib.piece**):

- **InteractiveProjection** – převádí uživatelské vstupy z myši a klávesnice na transformační matice. V konečném důsledku umožňuje intuitivní prohlížení 3D scény pomocí napojení na rozhraní **Render3D** změnou jeho matic.
- **StaticCSGScene** – umožňuje přístup k definované *CSG* scéně ze souboru. Tato třída implementuje **RTScene**.
- **VEFDS** – základní implementace rozhraní **Brep**. Definuje scénu pomocí vrcholů, hran, stěn, jejich dělení a těles.
- **DefaultRender3D** – základní implementace **Render3D**. Má v sobě uložená data základních transformačních matic (*model-view*[4] a *projection*[5]) vykreslované scény.

4 Model-view matice – transformační matice sloužící k posouvání, otáčení a změně velikosti těles či celých scén

- **`DefaultFileFormat`** – základní implementace **`DataFileFormat`**, umožňující nahrávání a čtení různých datových typů. Umožňuje také napojení na zásuvku **`BitStream`** a číst data přímo z konkrétního *streamu*.

2.2 Výběr použité technologie pro HW podporu

Hardwarová podpora grafiky je v současné době používána nejvíce přes dvě nejznámější rozhraní – *OpenGL*[6] a *Direct3D*[7]. Pro použití v knihovně *JaGrLib*, která je, jak již dříve bylo napsáno, implementovaná v jazyce *Java*, je technologie *Direct3D* spíše omezujícím prvkem a to hlavně díky platformní závislosti na operačním systému *Microsoft Windows*. Dalším prvkem při výběru technologie byly existence implementací *OpenGL* v jazyce *Java*.

Vzhledem k tomu, že většina tvůrců implementace *OpenGL* rozhraní ke grafické akceleraci vytváří knihovny v jazyce *C*[8], bylo potřeba nalézt převodník rozhraní právě mezi platformami *Java* a *C*. Možnosti pak jsou tyto:

- **gl4java** – http://www.jausoft.com/gl4java/ . Implementace *OpenGL 1.3*. Projekt byl naposledy vydán v roce 2001 a je tedy zastaralý vzhledem k jeho verzi.
- **SWT OpenGL bindings** - http://www.eclipse.org/swt/opengl/ . Projekt od nadace The Eclipse Foundation jako součást jejich knihovny pro tvorbu okenních aplikací. Některé funkce však nejsou převedeny v souladu s omezeními jazyka *Java* (příkladem mohou být ukazatele, ke kterým nemá *Java* přístup).
- **JOGL** – *Java bindings for OpenGL* - https://jogl.dev.java.net/ . V současné době nejrozšířenější už jen díky podpoře od firmy *Sun Microsystems* – tvůrce jazyka *Java*.

Zkušenosti ukázaly, že nejlepší volbou je knihovna *JOGL*. V dalších kapitolách je popsáno její zavedení do projektu *JaGrLib*.

5 Projekční matice – transformační matice sloužící k zadání tváře výřezu zobrazovaného prostoru. Takovouto tváří může být jehlan (pro perspektivní zobrazení), nebo kvádr (pro ortogonální zobrazení).

6 Open Graphics Library – standard specifikující API pro používání grafických akcelerátorů. Viz také [2].

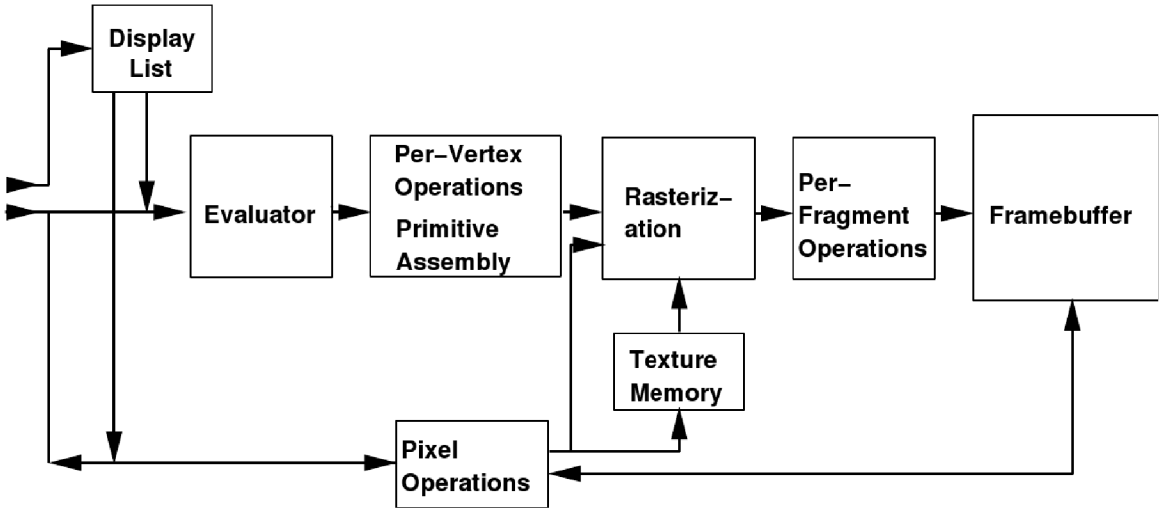
7 Microsoft® Direct3D® – knihovna firmy Microsoft™ pro přístup ke grafickým akcelerátorům. Je součástí větší knihovny DirectX® pro tvorbu multimediálních aplikací.

8 Jazyk C je nízkoúrovňový programovací jazyk vyvinutý Kenem Thompsonem a Dennisem Ritchie.

2.3 Rozhraní OpenGL

OpenGL je standard definující rozhraní pro tvorbu počítačové grafiky. Jeho hlavní výhodou, jak již bylo popsáno, je multiplatformní rozšíření. Rozhraní je implementováno na většině platformách a zejména v ovladačích nejpoužívanějších grafických karet od firem nV*idia* a *ATI*. Rozhraní definuje způsoby vykreslování do grafického *framebufferu* za pomoci programování vykreslovacího řetězce (*pipeline*). Dokumentace rozhraní je rozsáhlá [2], proto jsou popsány pouze základní vlastnosti programování *pipeline* potřebné pro použití v práci.

Na obrázku (Obr 1) je zobrazen vykreslovací řetězec *OpenGL*. Do řetězce vstupují data dvojího typu – *vertex data* a *pixel data*.



Obr 1: Blokový diagram OpenGL pipeline

Vertex data jsou údaje o vrcholech grafických primitiv, která jsou následně zpracovány a zobrazeny. Údaje musí obsahovat souřadnice vrcholu (možnosti jsou pro 2D, 3D jak v homogenní tak nehomogenní formě). Dále mohou obsahovat barvu, parametry materiálu, normálu, texturové souřadnice a další parametry popsané v dokumentaci.

Pixel data se používají na vykreslování bitmap a textur či jiných rastrových dat. V práci nejsou tato data použita, takže nejsou více rozebírána.

Pro uchování opakovaně používaných dat ať *vertex*, tak *pixel* slouží **Display list**, který urychluje předávání těchto dat z operační paměti počítače a uchovává je většinou přímo v paměti grafického akcelerátoru.

Evaluator slouží k vykreslování parametrických ploch a kvadrik. Dopočítává nové vrcholy takových objektů a posílá je spolu se zadanými dále po řetězci. Protože v práci nejsou použity, nejsou

více rozebírány. Více lze dočíst ve specifikaci *OpenGL* [2].

Per-Vertex Operations a **Primitive Assembly** slouží především k transformaci vrcholů a také případných texturových souřadnic. Další funkcí této části vykreslovacího řetězce je ořezávání vertexů, tedy pokud leží za nastavenou ořezávací rovinou, jsou buď vypuštěny, nebo v případě úseček a polygonů jsou vytvořeny vertexy nové. Nedílnou součástí je také výpočet osvětlení na základě světelného zdroje, materiálu a normál.

Pixel Operations dekóduje barvy zadané aplikací do interního formátu, který používá platforma akcelerátoru. Výsledné barvy jsou buď uloženy jako rastr pro texturu, nebo jsou poslány do rasterizační jednotky.

Modul **Rasterization** generuje fragmenty, které vstupní primitivum pokrývají. Jsou brány v úvahu transformované souřadnice vertexů a rastrová data, tedy zejména textury.

Texture Memory může uchovávat opakovaně používané textury. Uložení textur v této paměti urychluje přesun jejích dat k rasterizaci a uvolní se tím tok dat na sběrnici počítače. Tento modul také kombinuje či jinak zpracovává uložené textury, což je využíváno například při *mip-mappingu*[9].

Modul **Per-Fragment Operations** umožňuje modifikovat fragmenty. Pokud jsou použity textury, jsou fragmentům přiřazeny pixely z textury. V tomto modulu je také vypočítána mlha a *anti-aliasing*[10]. Dále jsou aplikovány operace s vnitřními *buffery* (jsou součástí *framebufferu*) grafického akcelerátoru. Mezi ně patří i *buffer* hloubky (*Z-buffer*), který slouží k určování viditelnosti jednotlivých fragmentů.

Výsledky procesu na pipeline jsou uloženy do **Framebufferu**, který obsahuje i ostatní *buffery* (mezi ně patří *color buffer*, *Z-buffer*, *stencil buffer* a *accumulation buffer* – více o nich je přímo ve specifikaci *OpenGL* [2]). Průřez těmito *buffery* jsou právě fragmenty. Nejdůležitější je zde *color buffer*, který bývá většinou výstupem na zobrazovacím zařízení.

Pomocí funkcí *OpenGL* lze pak jednotlivé moduly nezávisle modifikovat a docílit tak různých zobrazení. Nejdůležitější pro tuto práci jsou pak funkce pro nastavování vertexů a primitiv, mezi které patří zejména smyčky úseček (*line-loop*) a konvexní polygony, nastavování jejich normál a materiálů. Dále pak parametry scény jako osvětlení, *anti-aliasing*, *shading model* a transformační matice.

9 Mip-mapping – technika používaná pro zobrazování textur, kdy textura je zadaná ve více velikostech a pro vykreslení je použita nejbližší vhodná velikost.

10 Anti-aliasing – technika sloužící k vyhlazování objektů s velkým rozlišením na nižších rozlišeních.

2.3.1 Knihovna GLU a Tessalátory

Součástí knihoven *OpenGL* bývá také nadstavbová knihovna *GLU* (*OpenGL Utility Library*), která obsahuje množství často používaných funkcí umožňujících programování pipeline na vyšší úrovni. Jejich programování přímo pomocí knihovny *OpenGL* by bylo častým opakováním dlouhých kódů. Mezi nejdůležitější funkce pro tuto práci jsou intuitivní nastavování některých transformačních matic a tessalátory. Více o této nadstavbové knihovně lze nalézt v [3].

Tessalátory slouží k rozdělení polygonů pomocí předdefinovaných funkcí. Použity jsou na rozdělení nekonvexních polygonů na konvexní tak, aby je byla schopna pipeline zobrazit tak, jak bylo původně zamýšleno (tedy nekonvexně). Tessalátory také podporují díry v polygonech a to zadáním opačného pořadí vrcholů díry. Při rozdělování polygonu může dojít k vytvoření nových vrcholů. Tessalátor pak podporuje pomocí *callback* funkce u těchto vrcholů dopočítání případných parametrů jako normálu a materiál či barvu z dat okolních vrcholů.

2.4 Používání knihovny JOGL

Knihovna *JOGL* je v současné době nejpoužívanější implementací API *OpenGL* pro *Javu* a to už vzhledem k jejímu naplnění požadavku JSR234[11]. Pro použití grafického urychlení a vykreslování konečných scén je potřeba vytvořit jednak vykreslovací okno, tedy kontext vyrovnávací paměti (*framebuffer*) grafické karty, ve které bude scéna vykreslována, dále pak implementovat či namapovat metody, které umožňují pouštět API knihovny *OpenGL* a programovat její pipeline. Oproti jazyku *C*, ve kterém jsou knihovny *OpenGL* nejčastěji psány a ve kterém se *OpenGL* nejčastěji používá, je jazyk *Java* objektový. Funkce, které jsou obvykle volány v *C*, mají svojí alternativu jako členské metody nějaké z tříd knihovny *JOGL*. Základní rozdíl je tedy ve volání metod nějaké instance třídy místo volání funkcí.

Knihovna je můstkem mezi jazyky *Java* a *C*, proto knihovna používá nativní knihovny[12] napsané v jazyku *C*. Nativní knihovny jsou však závislé na platformě a je tedy nutné požít knihovnu odpovídající. Knihovny pro *JOGL* jsou dostupné pro nejpoužívanější platformy – *Windows*, *Linux*, *Solaris*, *MacOS* a je možné je přeložit na platformu jinou díky otevřeným zdrojovým kódům. Vše je dostupné přímo na stránkách projektu [4].

11 Java Specification Request – navrhované a konečné specifikace pro platformu Java. Více na stránkách http://jcp.org/en/jsr/overview

12 JNI - Java Native Interface – je rozhraní umožňující volat kód psaný v jazyku C z virtuálního stroje jazyka Java. Implementace takové knihovny je pak nativní knihovna.

2.4.1 Vytvoření GL kontextu

Okno, které bude vykreslovat scénu, se vytváří stejným způsobem jako jiná okna v jazyce *Java*. Pomocí knihoven **`java.awt`** a **`javax.swing`** se vytvoří okno, ve kterém bude scéna zobrazována. K tomu může posloužit např. třída **`javax.swing.Jframe`** – více o vytváření oken a GUI komponent je např. v [5]. Vykreslovací plochu pro scénu pak přidáme jako komponentu a to instanci třídy **`GLCanvas`**. Při její inicializaci dojde k vytvoření GL kontextu. Protože samotná třída neví co má vykreslovat, je nutné jí přiřadit objekt implementující rozhraní **`GLEventListener`** pomocí metody **`addGLEventListener`**. Rozhraní disponuje několika metodami, které umožňují dále pracovat s kontextem v různých stavech. Následuje jejich popis:

- **`init(GLAutoDrawable drawable)`** – je voláno jednou – ihned po vytvoření kontextu GL. Slouží k inicializaci pipeline a nastavení potřebných parametrů vykreslování. Předávaný objekt je základní abstrakcí pro vykreslování grafického akcelerátoru a umožňuje získat instanci třídy **`GL`** pro programování pipeline. Disponuje také dalšími možnostmi jako např. nastavení automatického přepínání zobrazovaného *framebufferu*.
- **`display(GLAutoDrawable drawable)`** – slouží vyvolání vykreslení scény pomocí příkazů pipeline. Obvykle je volána v nekonečném cyklu (který může ovšem být přerušen nějakou událostí k ukončení vykreslování).
- **`reshape(GLAutoDrawable drawable, int x, int y, int width, int height)`** – slouží ke změně vykreslovacích parametrů při změně či posunutí okna GUI. Nejčastěji se používá k modifikaci projekční transformační matice, která udává, jakým způsobem bude scéna vykreslena.
- **`displayChanged(GLAutoDrawable drawable, boolean modeChanged, boolean deviceChanged)`** – tato metoda je volána při změně vykreslovacího módu či výstupního zařízení. Jako příklad může být změna barevné hloubky či přesun okna na jinou obrazovku. Pro účely knihovny *JaGrLib* není potřeba.

Implementací těchto metod můžeme tedy přímo programovat vykreslovací řetězec díky přístupu k objektu **`GL`** a to v každém možném stavu a změně.

Objekt **`GL`** zpřístupňuje veškeré funkce pro programování řetězce dle specifikace *OpenGL*[2].

2.5 Vytvoření modulu pro vykreslování

Spojení znalostí z předchozích kapitol již umožňuje návrh implementace a definování základního rozvržení kompozice. Prvním cílem je tedy vytvořit zobrazovací okno GL kontextu a umožnění programování pipeline. Výsledkem bude také rozhraní, jehož realizací můžeme vykreslovací řetězec ovlivnit.

Základem nového modulu **`GLWindow`** je rozhraní **`GraphicsViewer`**, které disponuje metodami k překreslování okna a zachytáváním událostí z myši a klávesnice v tomto okně. Modul je implementací tohoto rozhraní a protože je to modul knihovny *JaGrLib*, je také potomkem třídy **`Piece`**. V metodě **`repaintLoop`** je při prvním volání vytvořeno okno běžnými prostředky knihoven **`java.awt`** a **`javax.swing`** jazyka *Java* vytvořením instance třídy **`JFrame`**. Do okna je přidána komponenta pro GL kontext, jak je popsáno v kapitole 2.4.1. Klíčovým prvkem je implementace rozhraní **`GLEventListener`** pro události kontextu. Protože se v takové implementaci provádí veškeré činnosti na grafickém akcelerátoru a takové činnosti mají být prováděny jiným nebo novým modulem, je vytvořeno nové rozhraní (*plug*) knihovny *JaGrLib* – **`GLGraphics`**. Toto rozhraní opisuje metody rozhraní **`GLEventListener`** tak, aby případný tvůrce implementace mohl ovládat pipeline za všech podmínek, zejména při změně velikosti okna. Jako parametry se těmto metodám předávají objekty **`GL`** a **`GLU`**, které umožňují právě ovládání pipeline. Obsahem implementace rozhraní **`GLEventListener`** je tedy získání objektů **`GL`** a **`GLU`** z parametru **`GLAutoDrawable`** a předání novému rozhraní **`GLGraphics`**. Toto rozhraní má následující metody:

- **`initRoutine(GL gl, GLU glu)`**
- **`paintRoutine(GL gl, GLU glu)`**
- **`reshapeRoutine(GL gl, GLU glu)`**

Názvy jsou odvozeny z příslušných metod rozhraní **`GLEventListener`** a jejich funkce je totožná.

Implementace takového rozhraní umožňuje plný přístup k pipeline *OpenGL* . Protože nové rozhraní je zásuvkou (*plug*) knihovny *JaGrLib*, je možné tvořit nové moduly a využívat tak plný přístup k řetězci.

2.6 Napojení na modul s daty

Jedním z hojně používaných modulů v knihovně *JaGrLib* je implementace rozhraní **`Brep`** a to **`VEFDS`**. Rozhraní a jeho implementace umožňují uchovávat a získávat data scény v *b-rep* reprezentaci (vícek ní je popsáno v kapitole 3.2). Základními metodami, které jsou potřebné pro získání stěn a vrcholů pro zobrazení scény jsou:

- **`solidIterator`** – vrací objekt, který umožňuje procházet všechny tělesa ve scéně.
- **`faceIterator`** – vrací objekt, který umožňuje procházet všechny stěny ve scéně.
- **`faceInSolidIterator`** – vrací objekt, který umožňuje procházet všechny stěny v tělese.
- **`getSolidFaces`** – vrací odkazy na všechny stěny jednoho tělesa.
- **`getAttributeId`** a **`getAttribute`** – umožňují získat pojmenovaný atribut jednoho z objektů – stěny, hrany či vrcholu. Do těchto atributů jsou ukládány také normály a materiál.
- **`getFaceVertices`** – vrací odkazy na vrcholy jedné stěny.
- **`getVertexCoords`** – pro daný odkaz na vrchol vrací jeho souřadnice.

Získáním seznamů těles, stěn a vrcholů je nyní možné programovat pipeline *OpenGL*. Vykreslovacímu řetězci jsou posílány data vrcholů, tedy jejich souřadnice, informace o normálách (uloženo v atributu vrcholu nebo stěny) a barvě (také uloženo v atributu vrcholu nebo stěny). Jako primitivum je zde zvolen polygon (buď přímo nebo za pomoci tessalátoru) nebo smyčka úseček (*line-loop*) pro každou stěnu.

Vytvořen je tedy modul **`GLBrepRender`**, který implementuje zásuvku (*plug*) **`GLGraphics`** a při každém překreslení (**`paintRoutine`**), získá data z připojeného modulu implementujícího **`Brep`** a ty pak posílá pomocí objektu **`GL`** pipeline. Zároveň tento modul rozšiřuje třídu **`DefaultRender3D`**, která má v sobě uchovány transformační matice pro zobrazení scény. Tyto matice jsou také předány pipeline, která je uzpůsobena na jejich použití s vykreslovanými objekty.

V modulu jsou vytvořeny parametry, kterými může uživatel řídit výsledek vykreslované scény. Tyto parametry jsou většinou obrazy stejných parametrů pipeline a jsou následující:

- **Počet světel** – protože vykreslovací řetězec potřebuje při inicializaci vědět, za jsou světelné zdroje použity a případně kolik, je zde tento parametr. Souřadnice světla jsou nastave-

ny pevně pro ukázku, avšak rozšířením scény *b-rep* o informacích o světelném zdroji (atributy scény), lze pak souřadnice světelných zdrojů nastavit.

- **Shading model** – pipeline umožňuje nastavením *shading modelu* dvě základní rasterizace primitiv. *Smooth* (*Gouraud*) vypočítává barvy fragmentů pomocí interpolace barev mezi třemi vrcholy, oproti tomu *flat* nastavuje barvu primitiv podle barvy jednoho z vertexů.

- **Wired model** – tento parametr zjišťuje vykreslování stěn buď pomocí polygonů nebo jen pomocí jejich hran (*wired model*). Rozdíl je v použití primitiva, kdy při nastaveném parametru je použita smyčka úseček (*line-loop*), v opačném případě pak přímo polygon.

- **Nekonvexní polygony** – při nastavení tohoto parametru je místo přímého nastavení vykreslovaného primitiva konvexního polygonu použit tessalátor pro nekonvexní polygony. Tessalátoru se předávají vrcholy takového polygonu a ten pak rozdělením na konvexní polygony nastaví pipeline sám. U tessalátoru je pak potřeba dopočítat další parametry nově vytvořených vrcholů, zejména normály a barvy. Pomocí *callback* funkce tessalátoru, která předává čtyři vrcholy s parametry a jejich váhu, je poměrově dopočítána normála a barva.

- **Parametry perspektivního zobrazení** – pakliže není projekční matice nastavována pomocí zásuvky `Render3D`, je použita perspektivní projekce. Nastavuje se zde úhel jehlanu projekce (*FOV – Field of view*) a dvě ořezávací roviny – přední a zadní. Tyto parametry jsou předány knihovně *GLU*, která z nich umí vyrobit projekční matici.

- **Barva pozadí** – barva pozadí vykreslované scény, tedy barva na kterou je inicializován *color buffer* pipeline před každým vykreslením.

- **Depth function test** – při zpracovávání viditelnosti na základě dat ze *Z-bufferu* tato funkce určuje, které fragmenty jsou považovány za viditelné porovnáním jejich hloubky. Jsou to funkce porovnání, tedy $<$, $=$, $\leq$, $>$, $\neq$, $\geq$, nikdy nebo vždy. Nejčastěji používanou je $\leq$, tedy fragment je zobrazen, pokud je hloubka menší nebo rovna ostatním.

- **Clear depth** – nastavuje do jaké hloubky má být smazán *Z-buffer*. Hodnota je v intervalu <0;1>, kde 1 znamená největší hloubku, tedy smazáno bude všechno. Naopak 0 značí hloubku nejnižší a při jejím nastavení nejsou smazány žádné informace o hloubce fragmentů a jsou použity při vykreslování další scény.

- **Mazání jednotlivých bufferů** – jednotlivé *buffery* mohou být před vykreslením další scény buď smazány nebo ponechány. Parametr umožňuje smazat *color buffer*, *Z-buffer*, *aku-*

mulační buffer[13] a *stencil buffer*[14]. Poslední dva nejsou při zobrazování využity a pouze ukazují možnost jejich použití.

- **Nastavení tzv. Hintů** – *OpenGL* disponuje možnostmi nastavit některé parametry závislé na implementaci. Mezi *hinty* patří vyhlazování bodů, úseček i polygonů, kvalita interpolace barev a souřadnic textur a přesnost výpočtu mlhy. Nastavené mohou být na nejrychlejší, nejhezčí nebo bez starosti (tedy provedení je ponecháno osudu implementace). Poslední z *hintů* není využit a pouze ukazuje možnost jeho použití.

Metoda **`initRoutine`** zděděná po rozhraní **`GLGraphics`** nastaví *shading model*, nastaví barvu pozadí a vytvoří světla (buď pevně umístěná, nebo získaná atributu scény *b-rep*), pokud je nastaven počet světel na kladnou hodnotu. Dále nastaví hloubku, do které se mají záznamy o hloubce mazat (*clear depth*), porovnávací funkci na hloubku (*depth function test*) a nastaví *hinty*. V případě nastavení použití nekonvexních polygonů inicializuje tessalátor.

Metoda **`paintRoutine`** taktéž zděděná po rozhraní **`GLGraphics`** tedy nejdříve smaže *buffery* dle nastavení, nastaví *model-view* transformační matici (buď z připojeného modulu přes zásuvku **`Render3D`**, nebo dle nastavené perspektivní projekce), získá data z **`Brep`** (vrcholy, normály a barvy) a předá je pipeline. V případě nekonvexních polygonů za použití tessalátoru.

Nakonec metoda **`reshapeRoutine`** mění projekční matici na základě velikosti okna, tedy změní *viewport* (zobrazovanou plochu) na velikost okna a dle poměru stran a velikosti změní buď jehlan perspektivního zobrazení, nebo parametry ortogonálního zobrazení (rovnoběžnostěn).

2.7 Spouštěcí modul

Každá kompozice v projektu *JaGrLib* by měla mít jeden modul (*piece*) ke spouštění. Takový modul musí implementovat třídu **`Worker`**, která v metodě **`run`** implementuje postup spouštění a použití ostatních modulů v kompozici. Pro účely spouštění hardwarové podpory grafiky je vytvořen modul **`GLWorker`**.

Modul při spuštění zavádí zobrazované okno (připojené pomocí zásuvky **`GraphicsViewer`**) vytvořením vlákna s nekonečným cyklem (až na uživatelská a systémová přerušení), tedy metodu **`repaintLoop`**, ve které opakovaně volá své překreslení, a toto vlákno spustí. Pro další potřeby je

13 Akumulační buffer – umožňuje shromažďovat (akumulovat) jednotlivé scény jejich fragmentový obraz) tak jak přicházejí pipelinou a provádět nad nimi operace. Je využíván např. k zobrazení motion blur, tedy rozmazání scény při pohybu.

14 Stencil buffer – slouží k maskování (ořezání) scény na fragmentové úrovni.

u tohoto modulu možnost připojení k zásuvce **`Trigger`**, pro spuštění nějaké akce při zavedení. Pokud je modul přes takovou zásuvku připojen, je spuštěn ještě před vytvořením okna. Nakonec je v modulu vytvořena ukázková scéna *b-rep* (jsou vytvořena data v připojeném modulu přes rozhraní **`Brep`**). Tedy pokud je modul implementující **`Brep`** připojen, jsou v něm vytvořena data jednoduché krychle s různými barvami stěn.

2.8 Shadery

Rozhraní *OpenGL* umožňuje měnit vlastnosti některých částí pipeline za pomocí tzv. **shaderů**. *Shadery* jsou programy, které jsou kompilovány před jejich použitím a většinou jsou nahrány přímo do grafického akcelerátoru. Díky tomu je pak jejich pouštění rychlejší a není potřeba přepočty provádět na procesoru počítače. *Shadery* jsou dvojího typu: **vertex shader** a **fragment shader**. První ze jmenovaných umožňuje měnit souřadnice vstupujících vertexů. Výstupem může tedy být vertex s jinými souřadnicemi a může tak zastupovat transformační matice. Pomocí takového shaderu lze pak vytvářet různé efekty jako např. hladinu vody. Druhý ze jmenovaných – *fragment shader* – umožňuje modifikovat fragmenty při rasterizaci. Může tak zastupovat např. mapování pixelů textur na fragmenty, nebo může měnit hloubku fragmentů.

Oba z *shaderů* pak mohou přijímat parametry, na základě kterých provádějí svojí modifikaci. Jako příklad může být transformační matice *vertex shaderu*, kdy při její neznalosti nevíme pohled nebo otočení scény. *Shadery* mívají předdefinované a automaticky předávané proměnné. Pokud je potřeba zvolit vlastní proměnné (třeba údaj o posunutí v závislosti na čase), je potřeba je definovat jak v programu *shaderu*, tak i při předávání z programu pipeline, ve které je již *shader* zaveden.

V současné době jsou dvě nejznámější specifikace obou shaderů – jazyk **CG** od firmy *nVidia* (popis je např. v [6] nebo v [7]) a jazyk **GLSL** vytvořený konzorciem *OpenGL ARB* specifikovaný v [8]. Každý z těchto jazyků je specifický svými vlastnostmi, avšak mají některé společné jako kód zapsaný v textové formě a předávání parametrů. Oba jazyky pak mají podporu v *OpenGL* a také v knihovně *JOGL*.

2.8.1 Implementace shaderů v JaGrLib

Pro podporu *shaderů* v knihovně *JaGrLib* je vytvořeno rozhraní **`Shader`**, které umožňuje implementovat inicializaci a použití *shaderu* při nastavování vykreslovacího řetězce. V rozhraní byly deklarovány metody s ohledem na jejich obecné použití hlavně pro obě specifikace (*CG* a *GLSL*). Jejich výčet následuje:

- **isSupported** – vrací zda je *shader* na dané platformě podporován.
- **initShader** – inicializuje *shader* a případně překompiluje pro GPU. Metoda je pouštěna jednou před použitím shaderu.
- **loadShader** – nahraje program do GPU tak, aby mohl být použit při dalším vykreslování.
- **start** – spustí používání již nahraného programu *shaderu*.
- **stop** – ukončí používání tohoto *shaderu*.
- **getShaderCode** – vrátí původní kód programu *shaderu*.
- **getShaderType** – vrátí typ *shaderu*, tedy *vertex* nebo *fragment*.
- **SetParamUniform*** – nastaví proměnnou daného jména a typu na určitou hodnotu. Hvězdička zde zastupuje jednu z hodnot **Scalar**, **Matrix**, **StateMatrix**, **Array**, **MatrixArray** a určuje tedy typ dané proměnné. Některé z nich jsou přetížené a mohou jako parametr přijímat různé typy, které mohou odpovídat typu ve jméně.

Všechny metody kromě **getShaderType** a **getShaderCode** přijímají jako parametr odkaz na objekt pro programování pipeline – typu **GL**. Jednotlivé metody **setParamUniform*** pak mají parametr název proměnné a samozřejmě hodnotu samotného parametru.

Implementovány jsou třídy pro obě specifikace – **GLSLFileFormat** a **CGFileFormat**. Obě jsou potomky třídy **DefaultFileFormat**, která je použita právě pro načtení programu *shaderu* přes *stream*, a implementují třídu **Shader**.

Samotné metody třídy **Shader** jsou implementovány dle popisu zavádění a používání *shaderů* v *OpenGL* [8] a [7].

2.9 Ukázkové příklady

Pro účely předvádění byl vytvořen modul **SimpleGLGraphics**, který dle nastavení parametru vykreslí krychli, kouli nebo konvici s ukázkovou texturou a materiálem a s objektem otáčí podle osy ve směru vektoru (1, 1, 1). Modul také dokáže použít případný další připojený modul přes zásuvku **Shader** a aplikuje shader na celou zobrazovanou scénu.

Pro ukázku byly vytvořeny jednoduché kompozice, jejichž rozvržení a náhledy výsledků jsou ukázány v příloze A a to následující:

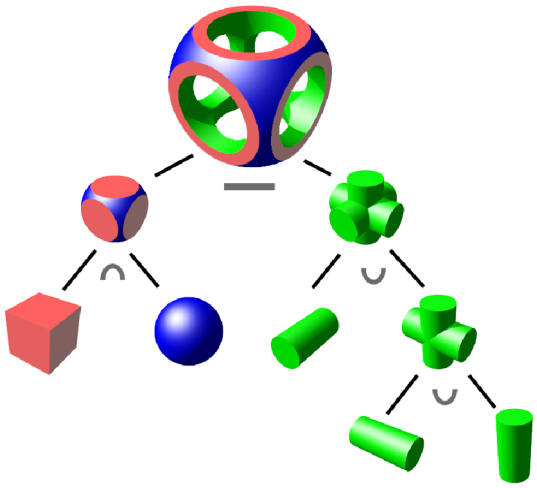
- zobrazení scény ze **`SimpleGLGraphics`**
- zobrazování scény generované v **`GLWorker`** za použití **`InteractiveProjection`**
- zobrazení objektu ve formátu *OBJ* přes *b-rep* scénu
- ukázka jednoduchého **CG** shaderu na scéně **`SimpleGLGraphics`**

3. Převod CSG do B-rep

Jednou z možných reprezentací scény v počítačové grafice je *CSG* (*Constructive solid geometry*). Je založena na analytickém popisu jednoduchých těles jejich objemem. Popis scény je pak chápán jako strom množinových operací nad těmito objemy. Tato reprezentace je vhodná zejména pro *ray-tracing*[15], kdy je výpočet průniku paprsku se scénou výhodný. Nevýhoda této metody je pak dlouhé trvání vykreslení celé viditelné scény, zejména pokud je potřeba měnit pozici pozorovatele, kdy je potřeba vrhat paprsky z jiných poloh. Tato kapitola popisuje převod takové scény do *b-rep*, která pak dále může být díky moderním grafickým kartám vykreslována v rychlém čase. Výsledkem by měla být scéna, která je až na omezení *b-rep*, tedy jemnost a hranatost, shodná.

3.1 Struktura CSG

CSG neboli *Constructive Solid Geometry* je popis scény pomocí množinových operací nad objemy jednoduchých těles. Jedná se o strom, v jehož listech jsou jednoduchá tělesa (krychle, válec, kužel, koule, poloprostor aj.) transformovaná do určité polohy a tvaru pomocí transformační matice. Vnitřní vrcholy stromu jsou množinové operace – pro jednoduchost sjednocení, průnik a rozdíl (ostatní operace mohou být dosaženy použitím těchto tří operací), které mohou mít jeden a více potomků, avšak záleží na jejich pořadí. Takový vrchol pak reprezentuje těleso vzniklé operací nad jeho potomky. Pořadí je zde hlavně pro operaci rozdílu, která není komutativní. Operace se provádějí nad potomky zleva doprava. Výsledná scéna je pak těleso, které reprezentuje kořen takového stromu. Příklad *CSG* stromu je na obrázku (Obr 2).



Obr 2: Příklad CSG stromu

15 Ray-tracing – metoda zobrazení scény založená na vrhání pomyslného paprsku světla opačně, tedy od oka ke scéně.

3.2 Struktura b-rep

B-rep, neboli *Boundary Representation*, je popis scény pomocí hranic útvarů. V prostoru 3D je popisována třemi základními prvky – stěnami, hranami a vrcholy. Stěna je část povrchu útvaru, tedy jeho hranicí, hrana je hranice stěny a vrchol hranicí hrany a reprezentuje jej bod v prostoru. Každý útvar je tedy reprezentován svými stěnami, které jsou zadány pomocí hran a ty pak pomocí dvou vrcholů. Tato reprezentace je zvlášť výhodná pro grafickou akceleraci, protože grafické jednotky jsou zaměřeny zejména na rychlé vykreslování polygonů, tedy v tomto případě stěn. Dalšími vlastnostmi takové scény mohou být sdílené hrany dvou stěn označovány jako *winged-edge* a nastavitelné parametry ke každému prvku.

3.3 Motivace

Grafické akcelerátory nejsou přímo uzpůsobeny na vykreslování scén zadaných v *CSG*, oproti tomu struktura *b-rep* je výhodná, protože obsahuje stěny jako polygony a ty mohou být vykreslené pomocí *OpenGL* primitiv. Zobrazení *CSG* scény přímo pomocí *OpenGL* je řešeno např. v [9]. Vychází z algoritmu Goldfeather [10], který ve vykreslovacím řetězci využívá *Z-bufferu* ke zjišťování fragmentů patřících do scény a *stencil bufferu* k ořezávání. *CSG* strom je však nutné před vykreslováním normalizovat. Toto řešení je rychlé a hojně využívané zejména v různých CAD systémech[16].

Algoritmus zde popsaný nedosahuje takových výsledků a je spíše zaměřen na převod právě do reprezentace *b-rep*, která může být pak dále využívána. Výhodou oproti přímému vykreslování je pak nezávislost na úhlu pohledu na scénu.

3.4 Současné metody

Převod jako takový je popsán již v [11], který vychází z popisu primitivních těles jako polytopů, které se skládají z konvexních polygonů. Algoritmus provádí množinovou operaci vždy na dvou tělesech ve vrcholech stromu *CSG* při jeho procházení, při níž hledá protínající se polygony různých těles a rozdělí je tak, aby vznikly nové, neprotínající se konvexní polygony. V poslední fázi pak zjišťuje, které polygony do scény patří na základě pozice vůči druhému tělesu.

Jiná metoda je popsána v [12], která je s popisovaným algoritmem v této práci podobná. Popisuje množinovou operaci na dvou tělesech, kde je u každé hrany zjišťována její část, která patří

16 CAD systém – computer-aided design systém – projektovací systémy obsahující množství grafických nástrojů jak pro design, tak i pro zobrazení.

do scény. Na základě sloučení těchto částí vzniká nové těleso.

Další algoritmus [13] je pak určený spíše pro reprezentaci *BSP* (*Binary Space Partitioning*), což je podobná struktura *CSG* založená na rozdělování prostoru pomocí ploch. Použití takové struktury by však obnášelo výpočet průsečíků všech takových ploch pro následné vykreslení polygonů v GPU.

3.5 Základní princip algoritmu

Algoritmus vychází z principu stejného převodu dvou polygonů v 2D popsané v [14] . Algoritmus nejdříve prochází rekurzivně *CSG* strom do hloubky. Začíná v kořeni stromu, kde získá tělesa, která jsou reprezentována jeho syny ve formě *b-rep*. Pokud je syn listem, je to jednoduché transformované těleso. Popis generování jeho *b-rep* reprezentace je popsáno v kapitole 3.12 . Pokud syn listem není provede se rekurzivní zjištění *b-rep* tělesa pro strom s kořenem v tomto synovi. Hlavní část algoritmu pak provádí operaci nad dvěma tělesy zadané v *b-rep*. Pokud je synů více, jsou operace prováděny zleva doprava. Tedy z prvních dvou synů vznikne těleso, které se použije k provedení operace se třetím tělesem a prostupuje se dál až k poslednímu.

Operace na dvou tělesech obnáší hledání nových vrcholů, které mohou tvořit výslednou scénu. Takovými vrcholy jsou průsečíky mezi stěnami jednoho tělesa s hranami tělesa druhého. Generování nového tělesa (výsledné těleso po množinové operaci) začíná v jednom z průsečíků. Následně se z tohoto vrcholu postupuje po všech hranách, které do nového tělesa mohou patřit. Z těchto vrcholů se stejným způsobem postupuje dál, dokud se neprojdou všechny takové hrany.

Výsledné těleso je pak dáno všemi hranami, kterými algoritmus prošel. Ty jsou k sobě uskupovány tak, že ve výsledném tělese tvoří součásti jedné stěny.

3.6 Předpoklady

Popisovaný algoritmus množinové operace dvou těles zadaných jako *b-rep* vychází z předpokladu, že jednoduchá tělesa jsou vytvořena kompaktně, tedy stěny sdílejí jak vrcholy, tak hrany (označované též jako *winged-edge*). Celé těleso pak musí být uzavřeno. Výjimky jako např. rovina nebo nekonečný válec je možné použít za předpokladu, že jsou po operaci jejich objemy konečné. Polygony tvořící povrch nemusejí být konvexní, avšak měly by být planární, tedy všechny vrcholy by měly ležet v jedné rovině. Zároveň také musí platit, že dvě různé stěny nemají buď žádný průnik, nebo je průnikem právě jedna hrana. Nenastane tedy případ, kdy dvě stěny v jednom tělese budou splývat. Vrcholy v tělese nesmí splývat, tedy vzdálenost dvou bodů musí být větší než nějaká

hodnota ϵ[17] vhodně zvolená pro platformu. Dalším omezením jsou normály vrcholů, které zejména v ostrých tělesech jako krychle mohou mít více směrů. Normálu v těchto případech je příhodné zadat jako průměr původních normál nebo je dodefinovat po skončení všech operací nad celým stromem. Všechny normály určují směr vně tělesa.

3.6.1 Předpoklady struktury b-rep

Struktura *b-rep*, která je očekávána pro algoritmus musí mít následující vlastnosti

- Hrany stěn jsou orientované, stěny jsou rovinné uzavřené jednosměrné smyčky takových hran.
- Hrany jsou okřídlené (*winged edge*), tedy každá hrana obsahuje odkazy na dvě stěny, kterým patří (levá a pravá), dále odkazy na vrcholy, jejichž pořadí udává orientaci hrany pro levou stěnu (pro pravou pak jejich opačné pořadí), a také odkazy na předchozí a následující hrany jak u levé tak u pravé stěny.
- Každý vrchol má záznam o hranách, kterým patří.
- Možnost procházení vrcholů a hran ve stěnách a procházení stěn v tělese.
- Je možné použít dodatečné atributy k vrcholům, hranám i stěnám, které mohou být měněny.
- Stěny mohou být definovány více orientovanými smyčkami hran a to pro informaci o „dírách“ v těchto stěnách. Hrany děr jsou v tomto případě orientovány opačně než hrany stěny samotné.

Protože implementace *b-rep* v *JaGrLib* (třída **`VEFDS`**) má všechny zmíněné vlastnosti, byl algoritmus implementován právě na ní.

3.7 Procházení CSG stromu

Převod *CSG* scény na *b-rep* začíná procházením stromu *CSG*. Začíná se u jeho kořene a získávají se tělesa, která reprezentují jeho syny. Synem může být buď list – jednoduché těleso, nebo podstrom. Pokud je synem podstrom, je na něm použito rekurzivní volání a získáno těleso reprezentující kořen podstromu. Na synech je pak provedena množinová operace dvou těles. Pokud je synů více provádí se operace po dvou zleva doprava. Následující pseudokód naznačuje takové pro-

17 Hodnota kladná blízká nule. Hodnoty nižší než tato hodnota mohou již být na dané platformě svojí reprezentací nepřesné.

cházení *CSG* stromem:

```
// parametry:
//   csgTree - odkaz na CSG strom scény
// výsledek:
//   brep reprezentace scény zadané jako CSG
function převedCSGnaBrep(CSGTree csgTree) : Brep
  return projdiVrcholCSG(csgTree.root)
end function

// parametry:
//   node - odkaz na vrchol v CSG stromu
// výsledek:
//   brep reprezentace scény kterou reprezentuje vrchol
function projdiVrcholCSG(CSGNode node) : Brep
  if(node je list) then
    vytvoř jednoduché těleso simpleSolid // kap.3.12
    return simpleSolid
  end if

  solidA = NULL
  foreach(child : node.children)
    if(solidA = NULL) then
      solidA = projdiVrcholCSG(child)
    else
      solidB = projdiVrcholCSG(child)
      solidA = solidSetOp(solidA, solidB, node.setOp) // kap.3.11.3
    end if
  end foreach

  return solidA
end function
```

Díky takovému procházení *CSG* stromu se redukuje problém na množinovou operaci dvou těles zadaných v *b-rep*. Další kapitoly se již zabývají právě touto množinovou operací.

3.8 Hledání průsečíků

Algoritmus se bohužel nevyhne výpočtu průsečíků hran jednoho tělesa se stěnami tělesa druhého. Takto vzniklé průsečíky – vrcholy – se pak stávají součástí nově vzniklého modelu pro jakoukoli množinovou operaci. Máme-li těleso A s počtem vrcholů v_A a počtem hran e_A a obdobně u tělesa B je počet vrcholů v_B a počet hran e_B, je časová složitost této operace $O(v_A \cdot e_B \cdot f + v_B \cdot e_A \cdot f)$, kde f je složitost výpočtu průsečíku hrana-stěna. Výpočet probíhá ve třech fázích.

V první fázi je spočítán hraniční kvádr stěny (polygonu), jehož hrany jsou rovnoběžné se souřadnými osami, v literatuře označován jako *Axis-Aligned Bounding Box* (zkráceně *AABB*). Je to nejmenší kvádr, který celý polygon obsahuje. Vzhledem k jeho následnému použití u ostatních hran jsou jeho parametry uloženy do paměti k příslušné stěně. Pokud se oba vrcholy testované hrany nacházejí v jedné polorovině definované stěnou *AABB* směrem od krychle, pak hrana polygon neprotíná.

Pokud je polygon definován vrcholy se souřadnicemi $(x_1, y_1, z_1), (x_2, y_2, z_2), \ldots, (x_n, y_n, z_n)$

a hrana vrcholy $(x_A, y_A, z_A), (x_B, y_B, z_B)$ pak lze hledání AABB a následný test formálně zapsat jako:

$$X_{min}^{AABB} = min(x_{1,} x_{2,} \dots, x_n)$$

$$Y_{min}^{AABB} = min(y_{1,} y_{2,} \dots, y_n)$$

$$Z_{min}^{AABB} = min(z_{1,} z_{2,} \dots, z_n)$$

$$X_{max}^{AABB} = max(y_{1,} y_{2,} \dots, y_n)$$

$$Y_{max}^{AABB} = max(x_{1,} x_{2,} \dots, x_n)$$

$$Z_{max}^{AABB} = max(z_{1,} z_{2,} \dots, z_n) \quad ;$$

$$(x_A < X_{min}^{AABB} \wedge x_B < X_{min}^{AABB})$$

$$(x_A > X_{max}^{AABB} \wedge x_B > X_{max}^{AABB})$$

$$(y_A < Y_{min}^{AABB} \wedge y_B < Y_{min}^{AABB})$$

$$(y_A > Y_{max}^{AABB} \wedge y_B > Y_{max}^{AABB})$$

$$(z_A < Z_{min}^{AABB} \wedge z_B < Z_{min}^{AABB})$$

$$(z_A > Z_{max}^{AABB} \wedge z_B > Z_{max}^{AABB})$$

Pokud je splněna alespoň jedna z šesti podmínek, hrana neprotíná polygon. Spočítání *AABB* v tomto případě trvá *O(n)* kde *n* je počet vrcholů jednoho polygonu. Pro celé těleso platí, že každá stěna má stejný počet vrcholů jako hran a každá hrana je sdílena právě dvěma stěnami. Předpočítání *AABB* všech stěn v obou tělesech tedy trvá $O(2 \cdot e_A + 2 \cdot e_B) = O(e_A + e_B)$. Následný test je konstantní a provádí se pro každou hranu - $O(e_A + e_B)$. Celkově tedy opět $O(e_A + e_B)$.

Tato fáze je prováděna z toho důvodu, že většina stěn složitějších těles se neprotíná a pro dvě tělesa s prázdným průnikem může urychlit výpočet.

Druhá fáze hledá průsečík hrany s rovinou polygonu. To předpokládá, že je známá rovnice roviny polygonu. Tato může být předpočítána při tvorbě polytopu primitivního tělesa. Její výpočet je popsán v kapitole 3.12.6. Hledání takového průsečíku je popsáno v [15] v kapitole 16.9 . Nejprve se provede změření vzdálenosti obou vrcholů od roviny. Pokud rovnice roviny má tvar $\vec{n} \cdot \vec{x} - d = 0$, pak vzdálenost vrcholu $\vec{A}$ je dána předpisem $s = \vec{n} \cdot \vec{A} - d$. Vzhledem k tomu, že vrcholy jsou sdílené více polygony, je dobré tyto vypočítané hodnoty uložit k patřičné dvojici vrchol-stěna. Test protínání je v tomto případě negativní, pokud hodnoty vzdáleností obou vrcholů hrany mají stejné znaménko. Testovány jsou všechny hrany které, prošly fází jedna jako potenciální kandidáti na průsečík. V nejhorším případě, kdy jsou testovány všechny hrany jednoho tělesa vůči všem stěnám druhého, je čas této fáze $O(e_A \cdot f_B + e_B \cdot f_A)$, kde f_A a f_B jsou počty stěn jednotlivých

těles.

Do třetí části jsou připuštěny pouze dvojice hrana-stěna, kde hrana protíná rovinu stěny. Tento test určí, zda vrchol vzniklý průsečíkem hrany a roviny je uvnitř nebo vně zkoumaného polygonu. Při tomto testu jsou spočítány souřadnice průsečíku díky známé rovnici roviny a také směrové rovnici přímky dané zkoumanou hranou – $L=\vec{X}+t\cdot(\vec{Y}-\vec{X})$, kde $\vec{X}$ a $\vec{Y}$ jsou souřadnice vrcholů hrany. Výsledný průsečík je dán vztahem

$$t=\frac{d-\vec{n}\cdot\vec{X}}{\vec{n}\cdot(\vec{Y}-\vec{X})} \quad ,$$

kde *t* udává pozici vrcholu na úsečce **XY** a díky poloze bodů na opačných stranách roviny je jeho hodnota v intervalu <0;1>. Výsledný bod pak vznikne dosazením hodnoty *t* do rovnice přímky.

Zjištění, zda daný bod leží v nekonvexním polygonu, lze pomocí průmětu polygonu a bodu na vhodné základní roviny. Vhodná rovina se dá určit dle [15] v kapitole 16.9 na základě absolutní hodnoty souřadnic normály roviny. Vypuštěním souřadnic s nejvyšší touto hodnotou jak u vrcholů polygonu, tak i u průsečíku, získáme problém náležení vrcholu v polygonu pro 2D systém.

Tato záležitost může být vyřešena např. podle [16], jakožto známého algoritmu řešícího tento problém v čase *O(n)*, kde *n* je počet vrcholů, tedy i hran polygonu.

Provedení této operace na všech stěnách a hranách obnáší složitost

$$O(e_A\cdot\sum_{f\in B}e(f)+e_B\cdot\sum_{f\in A}e(f))$$

kde *e(f)* je počet hran v dané stěně a sumy jsou počítány přes stěny v tělesech A a *B*. Díky duálnosti hran lze vztah zjednodušit na $O(2\cdot e_A\cdot e_B)=O(e_A\cdot e_B)$.

Celkové hledání průsečíků hran a vrcholů v tělesech má tedy složitost $O((e_A+e_B)+(a_A\cdot e_A\cdot f_B+a_B\cdot e_B\cdot f_A)+b_A\cdot b_B\cdot e_A\cdot e_B)$, kde a_A, a_B, b_A a b_B je procentuální rozložení variant, které mohou nastat a platí $0\le b_A\le a_A\le 1$ a $0\le b_B\le a_B\le 1$. V nejjednodušším případě se tělesa neprotínají a jsou v dostatečné vzdálenosti pro vyloučení průsečíků první fází a tedy hodnoty *a* a *b* jsou rovny 0. V nejhorším případě jsou pak hodnoty *a* a *b* rovny 1, tedy všechny hrany tělesa *A* protínají všechny roviny stěn tělesa *B* a naopak. Protože počet stěn je vždy menší, než počet hran, dá se složitost shora omezit jako $O(e_A\cdot e_B)$.

Pro další potřeby algoritmu je nutné uložit všechny vzniklé trojice **hrana-stěna-průsečík**, ke kterým se uloží informace o pozici průsečíku na hraně (vypočítaná hodnota *t*). Průsečíky jsou hledány i v krajních případech, tedy pokud se hrana dotýká hrany či vrcholu stěny a pokud se hrana dotýká pouze svým vrcholem stěny. Výjimku tvoří hrana, která splývá s rovinou stěny. Pokud alespoň jeden z vrcholů hrany se nachází uvnitř stěny, je tento vrchol vypočítán vedlejšími hranami polygonů, kterým hrana patří. Pokud i vedlejší hrana splývá, potom splývá i celý polygon. V tomto

případě nemusí být průsečík počítán vůbec a případ se řeší dle kapitoly 3.11.7.

3.9 Datová struktura pro průsečíky

Vzhledem k tomu, že v průběhu algoritmu budou vyhledávány průsečíky jak podle stěny, tak i podle hrany, je dobré mít uložené všechny trojice hrana-stěna-průsečík v indexovaném seznamu podle stěn i hran. Vyhledané trojice podle hrany je potřeba mít seřazené vzestupně podle pozice průsečíku na hraně (*t*). V následujícím textu bude označována trojice i s hodnotou t jako čtveřice $(F, E, V, t) : (F \in A \wedge E \in B) \vee (F \in B \wedge E \in A), V \in F \cap E, t \in \langle 0; 1 \rangle$, kde A a B jsou tělesa, na kterých se provádí operace, F je stěna jednoho z těles, E je hrana druhého tělesa, V je vrchol vzniklý jako průsečík stěny a hrany a t udává pozici vrcholu V mezi krajními vrcholy hrany E. Při implementaci v jazyce *Java* pro knihovnu *JaGrLib* byly použity dvě třídy **`HashMap`**, které pomocí hashovací tabulky přidávají a přistupují k příslušným hodnotám v konstantním čase za předpokladu dobře rozdělující hashovací funkce. Jako klíče do tabulky jsou použity hodnoty typu **`Integer`** a hashovací funkce vrací hodnotu klíče, tedy rozdělení konstantní čas nemění. Hodnotami v případě hledání podle stěn jsou odkazy na třídu **`HashSet`**, která používá stejné prostředky jako třída **`HashMap`** s tím, že jako hashovací funkce je použit odkaz na stěnu – typu **`int`**. Druhá hashovací tabulka odkazuje na setříděnou množinu – **`TreeSet`**, která zaručuje zápis a přístup k prvkům v čase $O(\log n)$. Tříděna je podle pozice na hraně, tedy hodnotě *t*. Vložení všech průsečíků do celkové struktury trvá $O(k \log k)$, kde k je počet vkládaných průsečíků. Vyhledání v takové struktuře trvá pro hledání podle stěn $O(1)$ a v případě hledání podle hran nejvýše $O(\log k)$.

Všechny uvedené časy struktur jazyka *Java* jsou popsány přímo v dokumentaci jazyka *Java*[17].

3.10 Tělesa bez průsečíků

Pokud nastal při předchozích testech na průsečíky případ, kdy nebyl nalezen ani jeden takový bod, pak jsou hranice tělesa disjunktní. Na základě množinové operace se pak vyberou ty tělesa, která do scény patří. V některých případech je potřeba na tělese otočit normály všech stěn a vrcholů. Tabulka 1 ukazuje výběr patřičného tělesa do scény.

operace	poloha	těleso A	Těleso B
$A \cup B$	A vně B	ano	ano
	A uvnitř B	ne	ano
	B uvnitř A	ano	ne
$A \cap B$	A vně B	ne	ne
	A uvnitř B	ano	ne
	B uvnitř A	ne	ano
$A \setminus B$	A vně B	ano	ne
	A uvnitř B	ne	ne
	B uvnitř A	ano	ano, s otočením normál

Tabulka 1: Výběr těles do výsledné scény při prázdném průniku obalů

Jak je vidět v tabulce, je potřeba zjistit pro tělesa vzájemnou polohu. Tuto polohu je možno zjistit vybráním libovolného vrcholu z jednoho tělesa a testovat jeho polohu vůči tělesu druhému. Tento test je popsán v následující podkapitole (3.10.1). Náročnost operace je tedy závislá na tomto zjišťování a je buď $O(e_A)$, nebo $O(e_B)$ (v závislosti na tom, ze kterého tělesa vybíráme vrchol a proti jakému tělesu testujeme). Dále pak v případě otáčení normál je čas závislý na změně znamének normál u všech stěn na tělese *B*. Reprezentace *b-rep* scény pak udává, jakým způsobem jsou tělesa sloučeny do jednoho. V případě knihovny *JaGrLib* a implementace třídou **`VEFDS`**, kdy je potřeba každé stěně přiřadit příslušnost k jinému tělesu, tato část trvá $O(f_B)$ a je realizovaná pouze ve dvou případech dle tabulky. Otáčení normál u stěn a vrcholů pak je realizováno v čase $O(f_B + v_B)$.

Celkově tedy vytvoření tělesa nového trvá $O(v_B + e_B + f_B)$ v nejhorším případě.

3.10.1 Poloha bodu vzhledem k tělesu

Test, zda bod leží uvnitř nebo vně uzavřeného tělesa zadaného polygony vychází z obdobného algoritmu ve 2D – pozice bodu vůči nekonvexnímu polygonu [16]. Stejným způsobem je vržen paprsek z vrcholu jedním směrem a testuje se počet protnutých stěn. Pokud je tento počet sudý je bod vně tělesa a naopak.

Paprsek můžeme vrhnout například rovnoběžně s osou **z** (vrhnutí paprsku rovnoběžného s ostatními osami je obdobný), přičemž testujeme všechny stěny tělesa. Nejdříve je určen směr ve kterém se průsečík s rovinou stěny nachází. Následující kroky jsou tedy provedeny jen v případě, že se stě-

na nachází v jednom směru od bodu po paprsku, tedy alespoň jedna ze souřadnic *z* vrcholů stěny je větší než souřadnice *z* bodu. Po dopočítání průsečíku s rovinou jedné stěny

$$[x, y, \frac{d - n_x x - n_y y}{n_z}] \quad ,$$

kde n_x, n_y, n_z a d jsou parametry rovnice roviny stěny a $[x, y, z]$ je souřadnice testovaného bodu, se použije test příslušnosti bodu do nekonvexního polygonu, popsaného v kapitole 3.8.

Při zjišťování mohou nastat také následující krajní možnosti:

- Zkoumaný bod leží přímo ve stěně – pak muže být automaticky považován za vnitřní bod. Vzhledem k předchozím testům na průsečíky mezi hranami a stěnami však tato možnost nenastane, protože by tento bod byl vyhodnocen jako průsečík a operace nad disjunktními obaly těles by se neprováděla.

- Průsečík prochází hranou či vrcholem – v tomto případě je zaznamenáno více průsečíků ač by měl být počítán jen jeden. Řešením je zapamatování hrany/vrcholu, který byl protnut. Pokud tedy při procházení stěn narazíme na průsečík v již zkoumané hraně či vrcholu, je tento průsečík ignorován.

- Paprsek je součástí roviny stěny ($n_z = 0$) – v tomto případě jsou všechny průsečíky ignorovány a jsou zaznamenány okolními stěnami.

Operace detekce vrcholu uvnitř tělesa A tedy trvá díky duálnosti hran a zkoumání vrcholu uvnitř polygonu $O(e_A)$.

3.11 Průchod soustavou dvou těles

Pokud se tělesa protínají a jsou spočítány průsečíky, je nyní možné konstruovat těleso nové, takové, které vznikne množinovou operací nad těmito tělesy. Konstrukce je založena na podobném principu jako ořezávání polygonu polygonem ve 2D popsaném v [14]. Tento algoritmus prochází polygon od prvního nalezeného průsečíku obou polygonů a poté z tohoto vrcholu prochází přes hrany nebo jejich části do dalších vrcholů a zařazuje tyto do výsledného ořezaného polygonu. Ořezání v tomto případě znamená provedení operace průniku obou polygonů, ale jak je v publikaci zmiňováno, úpravou lze docílit jakékoli množinové operace.

3.11.1 Datové struktury pro průchod

Při průchodu těles vznikají nové stěny z již známých vrcholů – buď z původních těles, nebo zjištěné průsečíky. Jednotlivé nové stěny jsou zaznamenány jako posloupnost vrcholů tvořících tuto

stěnu. Ve výsledku stěny vznikají pouze na původních stěnách jako jejich plošné podmnožiny. Na jedné stěně pak může nových vzniknout i více. Struktura, do které se budou nové vrcholy ukládat může tedy být mapa, kde klíčem je odkaz na původní stěnu a hodnotou je seznam uspořádaných množin vrcholů – tedy seznam nových stěn. Při provádění algoritmu mohou být nové stěny neuzavřené. Vkládání nových hran, tedy dvojic vrcholů pak je přizpůsobena tak, aby došlo ke sjednocení částí stěn, pokud je hrana spojuje. V knihovně *JaGrLib* je struktura realizována pomocí třídy **`HashMap`** s konstantním přidáváním a vyhledáváním s klíčem typu **`Integer`** jakožto odkazem na původní stěnu. Hodnotou je seznam (**`ArrayList`**) seznamů (opět **`ArrayList`**) typu **`Integer`**. Přidávání hrany do této struktury je popsáno v následující podkapitole (3.11.2).

Další potřebnou strukturou je seznam příznaků, zda vrcholy původní stěny byly použity či nikoli. Tato informace je potřebná pro detekci „děr“ ve stěnách, které však mohou vzniknout jen při operaci rozdílu a sjednocení. Příznaky jsou tedy přiřazeny každé původní stěně a mají jednobitovou informaci inicializovanou na hodnotu 0 (*false*). V implementaci je použita struktura **`HashSet`**, která postupně obsahuje odkazy na stěny, které použity byly – tedy s příznakem 1 (*true*) a inicializovaná je prázdnou množinou. Přístup a zápis do takovéto struktury je konstantní.

Pro záznam ještě nepoužitých průsečíků je použita jejich fronta. Při průchodu se odebírají použité průsečíky a v případě, že po průchodu nějaký zbyl, znamená to, že výsledné těleso nebude jedno, ale těles vznikne více. Je potřeba tedy průchod opakovat dokud nejsou použity průsečíky všechny. V realizaci je použita třída pro frontu **`ConcurrentLinkedQueue`**, která umožňuje odebírání prvků během jejího procházení. Její naplnění probíhá při získávání průsečíků a trvá tedy celkově *O(k)*, kde k je počet těchto průsečíků a mazání jednoho záznamu trvá stejný čas, tedy *O(k)*.

Při procházení tělesa, které oproti původnímu modelu ve 2D není lineární, je potřeba ukládat všechny vrcholy, do kterých se algoritmus dostal. Procházení může být realizováno jak do hloubky, tak i do šířky, tedy za použití obou přístupů – *FIFO* i *LIFO*. Realizace proběhla pomocí fronty *FIFO* opět třídou **`ConcurrentLinkedQueue`**, jejíž prvky jsou čtveřice

<vrchol, průsečík, předchozí vrchol, těleso>

Průsečík zde značí trojici **hrana-stěna-průsečík** včetně pozice průsečíku *t* a je prázdný v případě, že čtveřice symbolizuje pozici na nějakém původním vrcholu těles. Kromě této trojice jsou ostatní prvky typu **`int`**.

3.11.2 Vkládání nových hran

Každá nová nalezená hrana je přidávána k již známým. Při vkládání je známá původní stěna, ke které hrana patří a dva vrcholy této hrany. Protože vkládání nemusí být postupné, předpokládá se, že jsou hrany přijímány v různém pořadí. Výsledkem, po vložení všech hran pro danou stěnu, jsou uzavřené smyčky vrcholů. Struktura pro jednu stěnu je, jak již bylo dříve zmíněno, seznam seznamů vrcholů. Jedná se o seznam cest vrcholů, které jsou dány již vloženými vrcholy hran. Na začátku je tento seznam prázdný. Přidání jedné hrany pak obnáší:

- Zjištění, zda hrana není již použita.
- Vyhledání seznamu vrcholů, ke kterému muže být hrana přidána nakonec.
- Vyhledání seznamu vrcholů, před který může být hrana zařazena.
- Pokud hrana uzavírá cestu, tuto cestu označit příznakem uzavření.
- Spojení s patřičnými seznamy, případně založení seznamu nového.

Formálně lze zapsat přidání následujícím pseudokódem:

```
// parametry:
//   stěna - odkaz na stěnu ke které se má hrana přiřadit
//   v1 - odkaz na první vrchol nové orientované hrany
//   v2 - odkaz na druhý vrchol nové orientované hrany
// výsledek:
//   přidá hranu ke globálnímu seznamu cest pro stěnu
//   vrací true pokud hrana byla již přidávána, false v opačném případě
function přidej_hranu (Face stěna, Vertex v1, Vertex v2) : boolean
  cesty_stěny = seznam_cest_pro_stenu ( stěna )
  if(cesty_stěny je prázdné) then
    založ novou prázdnou a přidej do seznamu cest pro stěnu „stěna"
  end if

  přidej_za = NULL
  přidej_před = NULL

  foreach(cesta : cesty_stěny) do
    if(cesta je neprázdná) then
      if(v1 ∈ cesta & v2 ∈ cesta & v1 a v2 jsou za sebou) then return false
      if(cesta je uzavřena & v1 je poslední prvek cesta & v2 je první prvek
            cesta) then return false
      if(cesta není uzavřena) then
        if(v1 je poslední prvek cesta) then
          if(v2 je první prvek cesta) then
            uzavři(cesta)
            return false
          end if
          přidej_za = cesta
        end if
        if(v2 je první prvek cesta) then
          přidej_před = cesta
        end if
      end if
```

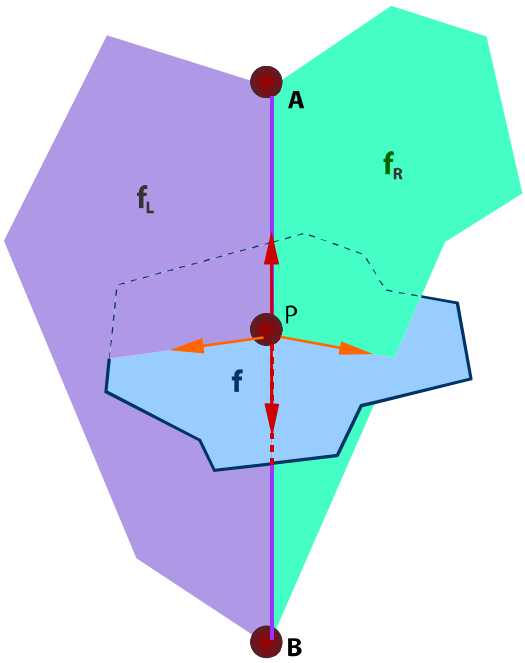
```
    end if
  end foreach
  if(přidej_za <> NULL && přidej_před <> NULL) then
    nová_cesta = spoj(přidej_za, přidej_před)
    přidej nová_cesta do cesty_stěny
    smaž přidej_před a přidej_za z cesty_stěny
    return false
  end if
  if(přidej_za <> NULL) then
    přidej v2 k přidej_za na konec
    return true
  end if
  if(přidej_před <> NULL) then
    přidej v1 k přidej_před na začátek
    return true
  end if
  založ nový seznam (v1,v2) a přidej ho do cesty_stěny
end function
```

Vstupem této funkce je identifikace stěny – ***stěna*** a identifikace vrcholů hrany ***v1*** a ***v2***. Předpoklad je také existence globální tabulky seznamu seznamů cest pro stěny (funkce `seznam_cest_pro_stěnu` získává takový seznam pro danou stěnu). Výstupem je pak jednobitový příznak, zda je potřeba po přidání hrany přidávat další hrany navazující na tuto. Tedy v případě, že struktura hranu již obsahuje nebo spojuje dvě cesty, jsou následující hrany již zařazeny a další procházení navazující na tuto hranu by generovalo hrany již použité. Výsledek také umožní ukončení celého algoritmu procházení obalů tělesa, tedy aby obaly nebyly procházeny donekonečna. U každé cesty je pak příznak jejího uzavření, který může být realizován jako speciální symbol na konci cesty.

Přidávání jedné hrany do struktury tedy není triviální. Získání seznamu seznamů vrcholů pro danou stěnu je konstantní. Pokud seznam seznamů obsahuje r cest a cesty mají délku $p(i), i \in \{1,\ldots,r\}$, pak kontrola na existenci hrany obnáší průchod všech prvků, tedy $O(\sum p(i))$. Při tomto průchodu se také zaznamenávají potenciální navazující cesty. Zařazení hrany do nalezených seznamů a jejich případné spojení je konstantní. Všechny uvedené časy jsou udané na základě použití třídy **`ArrayList`** v případě seznamu cest a **`LinkedList`** v případě seznamu vrcholů cesty. Celkově tedy přidávání hran trvá $O(1+2+\ldots+h)=O(h^2)$, kde h je počet vkládaných hran k jedné stěně.

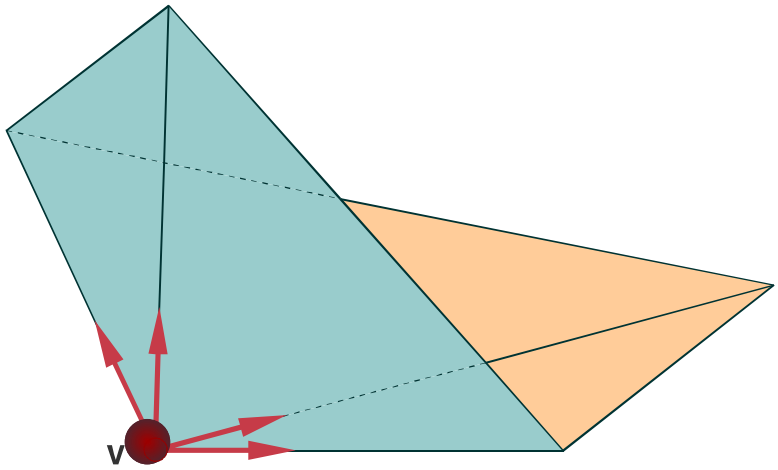
3.11.3 Procházení vrcholů

Procházení vrcholů začíná vždy v jednom, libovolném z nalezených průsečíků. Tento průsečík, stejně tak, jako i ostatní, bude patřit do scény vždy. Z tohoto průsečíku se dále postupuje k vrcholům sousedním, tedy jak po hraně, tak i po stěně, které tvoří průsečík. Situace je zakreslena na obrázku (Obr 3).



Obr 3: Směry, kterými se z průsečíku algoritmus při průchodu vydává

Na obrázku bod **P** vznikl jako průsečík stěny **f** a hrany **AB**. Z tohoto vrcholu je postupováno po oranžových šipkách po stěně a po červených šipkách po hraně k dalším vrcholům, které patří do scény. V případě postupu po hraně je pak vybrána nejvýše jedna možnost. Následující vrchol může být opět průsečík nebo původní vrchol jednoho z těles. V prvním případě se použijí pro další průchod opět směry naznačené na obrázku (Obr 3). Druhý případ je naznačen na obrázku (Obr 4). Z vrcholu původního tělesa se tedy postupuje všemi hranami které vrchol obsahují k dalším vrcholům (výjimku tvoří krajní případy popsané v kap. 3.11.7). Následující vrchol může být opět průsečík nebo původní vrchol stejného tělesa.



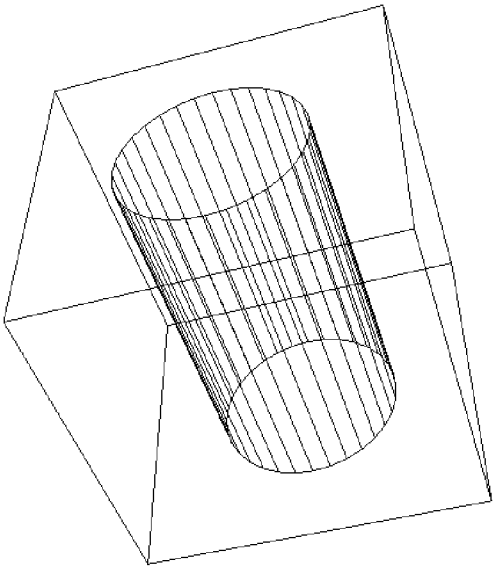
Obr 4: Směry, kterými se algoritmus vydává z původního vrcholu tělesa

K procházení vrcholů se tedy použije rekurze, která nám zaručí průchod vrcholů, které jsou z počátečního dosažitelné přes všechny probírané hrany. Zde tedy vzniká nutnost použití fronty či zásobníku právě pro vrcholy, které jsou aktuálně dosažené. Jednotlivé přechody z vrcholu do vrcholu jiného jsou přidávány jako hrany k patřičným stěnám pomocí algoritmu popsaného v kapitole 3.11.2. Díky tomu, že funkce přidávání vrací příznak, zda hrana byla již dříve zahrnuta, nebude se v případě kladné odpovědi postupovat dále. Toto zaručí, že algoritmus skončí a nebude procházet vrcholy donekonečna.

Při algoritmu jsou tedy hodnoty získávány z fronty. Záznam fronty pak udává vrchol, ve kterém se nacházíme, v případě průsečíku také informace o něm, tedy hranu, pozici na ní a stěnu. Dále je v záznamu hrany také obsažena informace o předchozím vrcholu, protože postup směrem k tomuto vrcholu by znamenal opakování záznamu hrany. Konečná informace udává, na jakém ze dvou těles se nacházíme. V případě průsečíku je to pak těleso, kterému patří průsečík tvořící hranu.

Každý průsečík je po průchodu přes něj odebrán ze seznamu dosud nepoužitých vrcholů. Tento seznam je na začátku naplněn všemi, viz kapitola 3.11.1. Po skončení algoritmu tedy v seznamu zůstávají nedosažené průsečíky. Jsou to takové, které tvoří další těleso ve výsledné scéně. Je tedy nutné algoritmus opakovat na těchto průsečících.

Další možnost, která může nastat jsou „děravé“ stěny. Příkladem může být úzký válec odečtený z krychle ukázaný na (Obr 5).



Obr 5: Rozdíl krychle a úzkého válce

Při průchodu, který začíná v jednom z průsečíků (nachází se vždy na hraně válce), se nikdy nedocílí vrcholu krychle. Duté stěny jsou v tomto případě stěny krychle bez kruhu uprostřed. Pro tento případ se u každé stěny zaznamenává, zda algoritmus prošel alespoň jednou její hranou. V případě, že se provádí množinová operace rozdílu či operace sjednocení, tedy operace, u kterých mohou duté stěny vzniknout, zkontrolují se všechny stěny, na kterých byly vytvořené stěny nové a pokud u stěny nebyla použita alespoň jedna původní hrana (v případě rozdílu se kontrolují pouze stěny na prvním tělese v rozdílu), zařadí se jeden z jejích vrcholů do fronty vrcholů ke zpracování.

Množinová operace nad tělesem pak může být zapsána následujícím pseudokódem:

```
// parametry:
//   solidA - odkaz na první těleso (v rámci operace)
//   solidB - odkaz na druhé těleso (v rámci operace)
//   op - množinová operace (sjednocení, průnik nebo rozdíl)
// výsledek:
//   odkaz na těleso, které vznikne množinovou operací nad předanými tělesy
function solidSetOp(Solid solidA, Solid solidB, SetOp op) : Solid
  průsečíky = najdi_průsečíky(solidA, solidB)   // kap. 3.8 a jejich zápis do
                      vyhledávací struktury viz kap. 3.9
  if(průsečíky je prázdné) then
    proveď triviální operaci nad tělesy (kap. 3.10)
    return nové vzniklé těleso
  end if


  fronta_vrcholů = empty_queue()
  nové_stěny = empty() // struktura popsaná v kap. 3.11.1, pro vkládání vrcholů
                          ke stěnám

  while(průsečíky není prázdné OR fronta_vrcholů není prázdné) do
    if(průsečíky není prázdné) then
      přidej do fronta_vrcholů první z průsečíky //pop()
    end if
```

```
    while(fronta_vrcholů není prázdné) do
      vrchol = fronta_vrcholů.pop()
      proveď postup přes vrchol // kap. 3.11.4 a 3.11.5
      // V tomto okamžiku jsou do nové_stěny přidány nové vrcholy a

      // zaznamenáno, zda vrchol je součást nějaké stěny (tknutá stěna)
    end while

    if(op je rozdíl) then
      najdi netknutou stěnu z solidA, u které byly zaznamenány nové vrcholy
      přidej jeden libovolný vrchol z ní do fronta_vrcholů
    else if(op je sjednocení) then
      najdi netknutou stěnu (obě tělesa), u které byly zaznamenány nové vrcholy
      přidej jeden libovolný vrchol z ní do fronta_vrcholů
    end if

  end while

  smaž původní tělesa solidA a solidB
  vytvoř nové_těleso z nových stěn z nové_stěny
  return nové_těleso
end function
```

Funkce tedy nejdříve zjistí průsečíky obou těles a zapíše je také do vyhledávací struktury. Jak již bylo popsáno v kapitole 3.8, vyhledání průsečíků trvá $O(e_A \cdot e_B)$. Zařazení průsečíků do vyhledávací struktury pak $O(k \log k)$ (viz kap. 3.9), kde k je počet průsečíků. Následuje cyklus, který kontroluje existenci nějakého neprobraného vrcholu. Jeho rychlost je tedy závislá na „spotřebě“ průsečíků v cyklu vnitřním a až na poslední kontrolu netknutých stěn je s ní časově spojená. Vnitřní cyklus prochází frontu vrcholů, které jsou do ní vkládány v průběhu procházení. Tímto cyklem prochází všechny vrcholy výsledného tělesa a díky vnějšímu cyklu jsou zahrnuty všechny, nikoli však duplicitně. Projít jím mohou tedy maximálně všechny původní vrcholy a nalezené průsečíky. Celkově tedy je nejhorší případ vnitřního cyklu při všech opakování $O((k+v_A+v_B)\cdot f^V)$, kde f^V je funkce, která určuje rychlost procházení jednoho vrcholu a skok na další. Hledání netknuté stěny je lineární – projdou se všechny stěny a hledá se taková, která je z prvního tělesa, má nastavený příznak netknutosti a vznikla na ní stěna nová. Protože vnější cyklus může brát v úvahu všechny průsečíky a také vždy jeden vrchol z každé stěny prvního tělesa, může tato akce trvat až $O((k+f_A)\cdot f_A)$.

3.11.4 Postup z původního vrcholu tělesa

Pokud procházíme vrcholem původního tělesa, je potřeba zjistit všechny hrany, které tento vrchol obsahují. Situaci zobrazuje obrázek (Obr 4). Z tohoto vrcholu se pak pokračuje po všech těchto hranách k nejbližšímu dalšímu vrcholu (výjimkou jsou krajní případy popsané v kap. 3.11.7). Tímto vrcholem může být buď druhý vrchol hrany, nebo nějaký dříve nalezený průsečík na této hraně. Zde je tedy použit dotaz do databáze průsečíků a to podle hrany (viz kap. 3.9), po které se právě vydává-

me. Není-li nalezen, pokračuje se u druhého vrcholu hrany. V případě, že nalezen byl, bere se v potaz nejbližší možný. Za určitých podmínek může dojít k tomu, že je nalezeno takových průsečíků více, nebo nejbližší splývá s vrcholem, na kterém se právě nacházíme. Tyto případy jsou rozebrány více v kapitole 3.11.7.

Nyní máme tedy vrchol ve kterém se nacházíme a vrcholy, ke kterým se bude postupovat. Ty budou tvořit hrany nového tělesa. Důležité zde je tedy, ke které původní stěně jsou přiřazeny. Vzhledem k tomu, že původní scéna má hrany orientované pro každou stěnu a u nové scény je potřeba mít hrany také orientované (musí vzniknout těleso, které splňuje podmínky k dalším operacím v *CSG* stromu), je výběr stěny proveden podle tabulky Tabulka 2.

<table>
<tr><th></th><th>v = V1</th><th>v = V2</th><th></th></tr>
<tr><td>Existuje průsečík na hraně e</td><td>Průsečík s nejnižším t</td><td>Průsečík s nejvyšším t</td><td rowspan="2">Druhý vrchol nové hrany</td></tr>
<tr><td>Neexistuje průsečík na hraně e</td><td>V2</td><td>V1</td></tr>
<tr><td>Operace je rozdíl (\) a nacházíme se na druhém tělese</td><td>pravá</td><td>levá</td><td rowspan="2">Stěna, ke které je nová, orientovaná hrana přiřazena</td></tr>
<tr><td>Nacházíme se na prvním tělese nebo operace není rozdíl (\)</td><td>levá</td><td>pravá</td></tr>
</table>

Původní okřídlená hrana (označena *e*), na které zkoumáme následující vrchol má záznam o vrcholech *V1* a *V2*, které jí patří v pořadí udávající orientaci pro levou stěnu (opačné pak udává směr pro stěnu pravou). Dále záznam obsahuje odkaz na levou a pravou stěnu, ke kterým se bude nová orientovaná hrana přiřazovat. Vrchol ve kterém se nacházíme je označen jako *v* a je to dle předpokladů buď *V1* nebo *V2*. Písmenem $t \in \langle 0;1 \rangle$ je míněna pozice průsečíků mezi vrcholy *V1* a *V2*.

Tabulka 2: Výběr následujícího vrcholu pro vrchol původní hrany, výběr přiřazení hrany k původní stěně

Podle tabulky je tedy přiřazena hrana s počátečním vrcholem v a s konečným vrcholem prvním následujícím na zkoumané hraně s touto orientací k jedné ze stěn. V případě, že provádíme operaci množinového rozdílu a nacházíme se na druhém tělese, je potřeba otočit normály jak vektorů, tak stěn. Tedy i orientace hran je ve výsledku opačná oproti původnímu směru, jak vystihuje Tabulka 2.

Po nalezení a přiřazení jsou zařazeny další vrcholy do fronty pro zpracování a to dle výsledku buď jako vrchol původního tělesa, nebo jako průsečík. Dále do záznamu fronty je také zaznamenáno stejné těleso, na kterém se vrchol nachází. Ve všech případech je stejné jako to, na kterém se nacházel vrchol, ze kterého jsme vycházeli. Nakonec jsou všechny stěny označeny příznakem, že se použila alespoň část jejich původních hran.

Postup obnáší nalezení všech hran z vrcholu, kterých ale nebude více jak $O(e_A+e_B)$ *a* vzhledem k uloženým informacím o hranách u každého vrcholu bude čas stejný. U každé hrany je pak hledán nejbližší vrchol, tedy dotaz do databáze průsečíků (viz kap. 3.9) v čase $O(\log k)$, kde k je počet průsečíků. Celkově tedy $O((e_A+e_B)\cdot\log k)$.

3.11.5 Postup z průsečíku

Došel-li algoritmus do vrcholu, který je průsečíkem nějaké stěny a hrany, je potřeba nalézt všechny nejbližší vrcholy, které se nacházejí ve všech možných směrech, které mohou tvořit novou hranu. Směry jsou buď k oběma vrcholům hrany tvořící průsečík nebo směry po stěně průsečíku a to takové, které jsou udány levou a pravou stěnou hrany průsečíku (viz Obr 3).

Protože stěna z průsečíku rozděluje hranu ve svém nejbližším okolí na dvě části a to na vnitřní a vnější část tělesa stěny, je na základě množinové operace brána pouze jedna tato část (viz Tabulka 3). V tabulce je zřejmé, že u jednoho tělesa patří do výsledné scény vždy právě jedna část.

Těleso	**Jeho část**	$A\cup B$	$A\cap B$	$A\setminus B$
A	uvnitř B	ne	ano	ne
	vně B	ano	ne	ano
B	uvnitř A	ne	ano	ano
	vně A	ano	ne	ne

Tabulka 3: Části těles, které na základě operace patří do scény

Část hrany, která patří do scény, lze jednoduše zjistit podle směru normály stěny, která tvoří průsečík. Pokud je tedy hrana definovaná dvěma vrcholy V_1 a V_2 v tomto směru (pro levou stěnu), spočítá se skalární součin $(\vec{V}_2-\vec{V}_1)\cdot\vec{n}$, kde $\vec{n}$ je normála stěny, a je-li jeho hodnota větší než nula, směr normály je na stejnou stranu jako směr vektoru vrcholů V_1 a V_2 hrany vzhledem ke stěně. Možnost výsledku rovného nule je ošetřen při vyhledávání průsečíků, kdy hrany, které jsou

na stejné rovině se stěnou (případ kdy je skalární součin roven nule) jsou ignorovány (viz kap. 3.8). Tabulka 4 ukazuje, který směr je pro dané operace brán v úvahu. Těleso, na kterém se nacházíme je to, kterému patří hrana průsečíku a je také předáváno jako jeden z parametrů záznamu fronty vrcholů ke zpracování.

operace	**těleso, na kterém se nacházíme**	$(\vec{V}_2-\vec{V}_1)\cdot\vec{n}>0$	$(\vec{V}_2-\vec{V}_1)\cdot\vec{n}<0$
$A\cup B$	nezáleží	k V_2	k V_1
$A\cap B$	nezáleží	k V_1	k V_2
$A\setminus B$	A	k V_2	k V_1
	B	k V_1	k V_2

Tabulka 4: Výběr směru hrany od průsečíku, který bude uvažován do výsledné scény

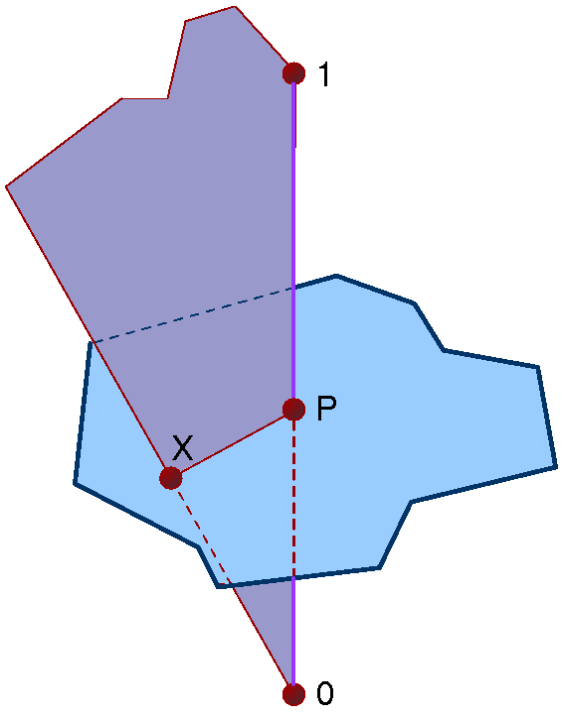
Nyní tedy stačí určit vrchol, který je ve zjištěném směru nejbližší. Hledá se v databázi průsečíků dle hrany (stejné jako ta, která tvoří průsečík) a to takový který má pozici průsečíku v daném směru. Předpokládáme-li, že se nacházíme ve průsečíku *P*, hledáme v případě směru „k V_1“ průsečík s hodnotou *t* největší takovou, která je menší nebo rovna než hodnota *t* průsečíku *P*. V případě směru „k V_2“ je to pak průsečík s nejmenší hodnotou *t* takovou, která je větší nebo rovna než hodnota *t* průsečíku *P*. Máme-li tedy čtveřici $P=(F,E,V,t)$ (průsečík ve kterém se nacházíme) a nalezené průsečíky na hraně $P_i=(F_i,E,V_i,t_i)$, pak pro směr „k V_1“ hledáme $i:t_i\le t\wedge\forall j:t_j\le t,t_i\ge t_j$ a pro směr „k V_2“ zase $i:t_i\ge t\wedge\forall j:t_j\ge t,t_i\le t_j$. Případy, kdy je těchto vrcholů nalezeno více nebo kdy nastává rovnost $t=t_i$, jsou rozebrány v kapitole 3.11.7. Pokud takový průsečík (takové *i*) nebyl nalezen, pak se hledá vrchol hrany v daném směru, tedy v případě směru „k V_1“ je to vrchol V_1, v opačném případě vrchol V_2. Vždy je tedy výsledkem pro daný směr právě jeden další vrchol. Nová hrana pro scénu z vrcholu průsečíku, ve kterém se nacházíme k dalšímu vrcholu na hraně, je přiřazena ke stěně dle tabulky Tabulka 5.

operace	**směr „k V_1“**	**směr „k V_2“**
A\B, hrana je na tělese B	levá	pravá
jinak	pravá	levá

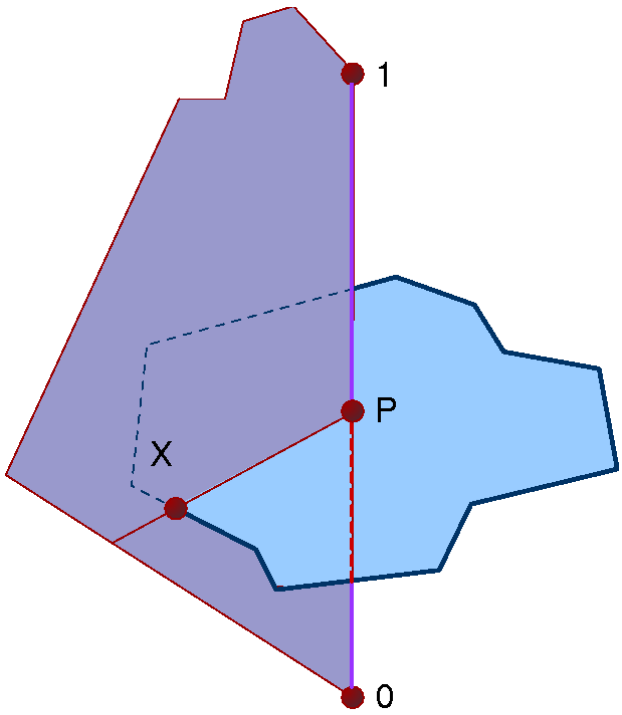
Tabulka 5: Výběr stěny pro novou hranu z průsečíku dle operace a směru

V tabulce je vidět, že výběr při operaci množinového rozdílu v případě druhého tělesa je dán opačným směrem normál a tedy i hran ve stěnách. Ve všech ostatních je směr správný.

Další možné nové hrany mohou vést z průsečíku pouze po stěně. Takové hrany jsou právě dvě a jsou určeny levou a pravou stěnou hrany průsečíku. V tomto případě jsou do výsledné scény přidány obě hrany. Situace, které mohou v jednotlivých směrech nastat jsou zachyceny na obrázcích (Obr 6 a Obr 7).



Obr 6: Následující vrchol po stěně (uvnitř stěny)



Obr 7: Následující vrchol po stěně (kraj stěny)

Na obrázcích je vidět, že další vrchol po stěně (X) se může nacházet jak uvnitř stěny tak i na její hraně. Takový vrchol je vždy průsečík. Průsečíky, které splývají s některým z vrcholů jsou detailněji popsány v kapitole 3.11.7. Na obrázcích je také zřejmé, že bod X vznikne buď jako průsečík na stejné stěně s nějakou jinou hranou buď levé, nebo pravé stěny hrany průsečíku (označeno 0 a 1) – (Obr 6), nebo jako průsečík levé nebo pravé stěny hrany průsečíku s hranou stěny průsečíku. Zde je tedy potřeba najít takové průsečíky pomocí dotazu do vyhledávací struktury. Poté je hledán takový, který má nejmenší vzdálenost od původního průsečíku. Jsme-li tedy v průsečíku (F, E, P, t) hledají se v databázi průsečíku (DB) takové i, které splňují:

$(F, E_i, V_i, t_i) \in DB, E_i \in \text{levá stěna } E, E_i \neq E$

$(F, E_i, V_i, t_i) \in DB, E_i \in \text{pravá stěna } E, E_i \neq E$

$(levá\ stěna\ E, E_i, V_i, t_i) \in DB, E_i \in F$

$(pravá\ stěna\ E, E_i, V_i, t_i) \in DB, E_i \in F$

Vzhledem k uzavřenosti těles je pro levou i pravou stěnu nalezen průsečík alespoň jeden. Z těchto je vybrán pro levou i pravou stěnu takový, který má nejmenší vzdálenost od průsečíku (*P*). Případy, kdy je vzdálenost nulová je popsána v kapitole 3.11.7. Vzdálenost dvou bodů $V_1 = [X_1, Y_1, Z_1]$ a $V_2 = [X_2, Y_2, Z_2]$ lze jednoduše spočítat jako $\sqrt{(X_2 - X_1)^2 + (Y_2 - Y_1)^2 + (Z_2 - Z_1)^2}$ a protože odmocnina je funkce rostoucí, pro výpočty stačí její vnitřní člen, tedy

$d = (X_2 - X_1)^2 + (Y_2 - Y_1)^2 + (Z_2 - Z_1)^2$.

Hrany, které vzniknou spojením průsečíku s nově nalezenými vrcholy (pro levou a pravou stěnu) mohou být přiřazeny buď stěně tvořící průsečík, nebo levé či pravé stěně. Tabulka 6 ukazuje výběr takové stěny.

	operace	**Směr „k V1“**	**Směr „k V2“**
levá strana	A\B, hrana je na tělese B	stěna průsečíku	levá stěna hrany průsečíku
	jinak	levá stěna hrany průsečíku	stěna průsečíku
pravá strana	A\B, hrana je na tělese B	pravá stěna hrany průsečíku	stěna průsečíku
	jinak	stěna průsečíku	pravá stěna průsečíku

Tabulka 6: Výběr stěny pro nové hrany z průsečíku po stěně

Opět je v tabulce vidět, že v případě rozdílu na druhém tělese, se volí stěna opačně, z důvodu obrácených normál a tedy i směrů hran.

Po zařazení hrany k příslušné stěně jsou všechny vrcholy, které operace našla, zařazeny do fronty vrcholů ke zpracování. Jako parametry záznamu pro frontu je přidána jak čtveřice dalšího vrcholu – průsečíku, tak i těleso, na kterém se nacházíme. Toto těleso je stejné v případě, že vrchol je nalezen dle obrázku (Obr 6). V opačném případě (je nalezen dle Obr 7) se těleso mění na druhé (hrana takového průsečíku je na druhém tělese, než současná). V ani jednom případě (levá a pravá strana) není označena žádná stěna jako tknutá, protože jsme nepoužili původní hranu tělesa a vytvořili jsme novou.

Okolní body průsečíku, ke kterým se bude postupovat, jsou buď po hraně nebo po stěně. Po hraně je vyhledáváno ve vyhledávací struktuře v čase $O(\log k)$ (*k* je počet průsečíků obou těles) a provádí se pouze jednou pro jednu hranu. Vyhledání vrcholů po stěně je složitější a v databázi

se hledá nejdříve podle stěny ($O(1)$) a poté ve vyhledaných průsečících, kde se hledají všechny takové, jejíchž hrana patří k nějaké stěně jiné. U hrany je však záznam právě o stěně, ke které patří, tedy opět v čase $O(1)$. V nejhorším případě mohou být všechny průsečíky na jedné stěně a tedy je potřeba prověřit *k*-krát náležení k jiné stěně. Tedy pro jeden průsečík by obnášelo vyhledání $O(k)$, což by vedlo k výsledku $O(k^2)$ pro všechny průsečíky. Pokud jsou však všechny průsečíky na stěně jedné, nemohou být na stěnách ostatních (výjimku tvoří průsečíky ve vrcholu nebo hraně stěny což je probráno v kap. 3.11.7). Vyhledání je proto v průběhu celého algoritmu dáno časem

$$O(4 \cdot \sum_{F \in DB} k(F)) \quad ,$$

kde *DB* je vyhledávací struktura, *F* je stěna která je ve struktuře zaevidovaná a $k(F)$ je počet průsečíků na této stěně. Číslem 4 je vyjádřeno, že průsečíky se hledají jak pro levou, tak pravou stranu a vždy jsou hledány případy podle (Obr 6) a podle (Obr 7). Suma ve výrazu je rovna *k*, celkový čas vyhledání (v průběhu celého algoritmu) je tedy pouze $O(k)$.

3.11.6 Ostré hrany a normály vrcholů

Protože ve vrcholu může být v *b-rep* nastavena pouze jedna normála, avšak v ostrých hranách může být takových normál více, jsou u každého průsečíku hrany a stěny vytvořeny vrcholy dva s různými normálami. Jeden pro hranu a jeden pro stěnu. Při provádění algoritmu celého *CSG* stromu je však používán vždy jeden – hranový. Po dokončení generování celé scény jsou vrcholy u patřičných stěn zaměněny za stěnové vrcholy. Taková scéna však již nesplňuje podmínky pro algoritmus a nemůže být následně použita znovu.

Normály pro hrany jsou počítány interpolací, stejně tak jako normály pro stěny. Stejným způsobem jako s normálami lze pak zacházet i s barvami a materiály.

3.11.7 Krajní případy

V průběhu procházení po vrcholech tělesa mohou nastat případy, kdy dva či více vrcholů a průsečíků splývají, nebo se průsečíky nacházejí na hraně mezi dvěma stěnami. V takových případech může dojít k nesmyslným výsledkům, protože mezi dvěma splývajícími vrcholy není udán směr, který je ve všech případech vyžadován. Takové vrcholy jsou pak sloučeny v jeden. Pro účely algoritmu je však nutné znát i původní vrcholy a průsečíky, a proto je zavedeno mapování splývajících vrcholů, tedy funkce $coincide : V \rightarrow V$, kde V je množina všech vrcholů a průsečíků. Pro takovou funkci pak platí:

$$(coincide(x)=y)\Leftrightarrow(dist(x,y)<\epsilon)$$
$$(coincide(x)=y)\Rightarrow(coincide(y)=y)$$
$$(x\neq y\wedge coincide(x)=y)\Rightarrow\forall z\in V: coincide(z)\neq x$$

Kde *dist* je funkce vzdálenosti dvou bodů a ϵ je vhodně zvolená konstanta pro číselné odchylky při výpočtu na nějaké platformě. Tato hodnota by měla být kladná co nejblíže nule.

Jinými slovy tedy každý vrchol odkazuje na jediného zástupce sdílených vrcholů.

Pro implementaci byla zvolena třída **`HashMap`**, u níž nejsou hodnoty, kde $coincide(x)=x$ nejsou zaznamenány. Funkce a tedy zařazování do struktury je prováděno v průběhu algoritmu při nalezení takových bodů. Do výsledných hran je pak zadáván zástupce namísto samotného bodu.

Situace samotná může nastat v těchto případech:

(1) Při postupu z původního vrcholu tělesa po hraně, kdy nejbližší vrchol(y) splývají s tímto vrcholem.

(2) Při postupu z původního vrcholu tělesa po hraně, kdy nejbližší vrchol(y) splývají sami se sebou.

(3) Při postupu z původního vrcholu tělesa po hraně, kdy nejbližší vrchol(y) splývají s druhým vrcholem hrany.

(4) Při postupu z průsečíku po hraně (tvořící průsečík), kdy nejbližší vrchol(y) splývají s tímto průsečíkem.

(5) Při postupu z průsečíku po hraně (tvořící průsečík), kdy nejbližší vrchol(y) splývají sami se sebou.

(6) Při postupu z průsečíku po hraně (tvořící průsečík), kdy nejbližší vrchol(y) splývají s vrcholem hrany.

(7) Při postupu z průsečíku po stěně (tvořící průsečík), kdy nejbližší vrchol(y) splývají s průsečíkem.

(8) Při postupu z průsečíku po stěně (tvořící průsečík), kdy nejbližší vrchol(y) splývají sami se sebou.

Případy (2), (3), (5), (6) a (8) jsou vyřešeny prostým postupem k jednomu z dalších vrcholů, kde nastane jeden z případů (1), (4) a (7), tedy kdy se nacházíme ve vícenásobném vrcholu.

V těchto případech jsou zařazeny všechny splývající vrcholy do fronty ke zpracování, avšak je u nich nastaven příznak o opatrnosti při postupu po hranách. Na takových vrcholech, včetně toho,

na kterém ke krajní situaci dojde, je postupováno tak, že na všech vycházejících hranách najde nejbližší další vrchol nebo průsečík, avšak s nenulovou vzdáleností. Následně je spočítán střed mezi oběma vrcholy a u něj je proveden test náležení do tělesa (do opačného než toho, kterému patří hrana) podle kapitoly 3.10.1. Dle tabulky Tabulka 7 je pak rozhodnuto, jestli z daného vrcholu po hraně postupovat k vrcholu dalšímu či nikoli.

operace	**bod na hraně tělesa A**		**bod na hraně tělesa B**	
	uvnitř B	**vně B**	**uvnitř A**	**vně A**
$A \cup B$	ne	ano	ne	ano
$A \cap B$	ano	ne	ano	ne
$A \setminus B$	ne	ano	ano	ne

Tabulka 7: Výběr hrany na základě polohy bodu v tělese

Pro každou hranu tělesa B vedoucí z takového vrcholu je tedy prováděn test náležení do tělesa v čase $O(e_A)$ a opačně. V nejhorším případě je tedy test proveden v čase $O(e_A \cdot e_B)$ v celé scéně.

Dalším krajním případem mohou být splývající stěny. Mohou být detekovány při hledání průsečíků hran se stěnami (kap. 3.8) a to tak, že jsou nalezeny alespoň dvě splývající hrany a alespoň jeden vrchol leží uvnitř stěny. Pakliže k tomu dojde, je nutné zjistit orientaci obou stěn, tedy jsou-li normály shodné či opačné. Poté je provedeno vzájemné ořezání takových stěn, které je možné provést např. podle [14]. Máme-li tedy takové stěny F_A na tělese A a F_B na tělese B, vytvoří se nové dle tabulky Tabulka 8.

operace	**normály shodné**	**normály opačné**
$A \cup B$	$F_A, F_B \setminus F_A$	$F_A \setminus F_B, F_B \setminus F_A$
$A \cap B$	$F_A \cap F_B$	žádná
$A \setminus B$	$F_A \setminus F_B$	F_A

Tabulka 8: Výběr ořezání u splývajících stěn

Každé ořezání je dáno množinovou operací a vlastnosti ořezané stěny jsou zděděny po prvním členu operace. Ořezání mohou být provedena po dokončení operace na tělesech (procházení vrcholů), avšak již na nových stěnách určených smyčkami.

3.12 Tvorba polytopů základních těles a útvarů

Protože listy *CSG* stromu jsou tvořeny základními tělesy, zejména krychlí, válcem, kuželem, koulí a poloprostorem a to v jednotkové formě v počátku, je potřeba pro převod do *b-rep* vytvoření jejich polytopu. Takový polytop je pak potřeba pomocí transformační matice zadané také v listu *CSG* stromu stejným způsobem transformovat. Tato kapitola popisuje tvorbu takových polytopů. Tělesa se při každém použití vytvářejí stejným způsobem, odlišné jsou pak pouze transformační matice. Proto mohou být data jednotkového polytopu uchována pro každé použití a čas na jejich vytvoření by obnášelo pouze jejich transformování. Popsány jsou zde způsoby získání takových dat. Samotná transformace je popsána v kapitole 3.12.7 .

3.12.1 Krychle

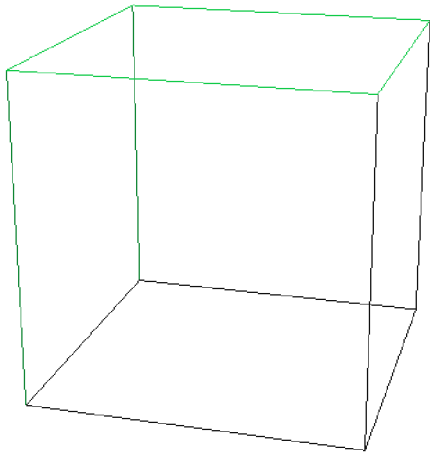
Jednotková krychle je krychle, jejíž každá hrana má délku 1, její hrany jsou rovnoběžné s některou souřadnou osou a její střed (průsečík tělesových úhlopříček) leží v počátku. Sestrojení takového tělesa obnáší tedy vytvoření vrcholů se souřadnicemi:

$V_{000}=(-0.5,-0.5,-0.5)$, $V_{001}=(-0.5,-0.5,0.5)$, $V_{010}=(-0.5,0.5,-0.5)$,
$V_{011}=(-0.5,0.5,0.5)$, $V_{100}=(0.5,-0.5,-0.5)$, $V_{101}=(0.5,-0.5,0.5)$,
$V_{110}=(0.5,0.5,-0.5)$, $V_{111}=(0.5,0.5,0.5)$.

Vytvořením stěn zadaných vrcholy $(V_{001,}V_{011,}V_{010,}V_{000})$, $(V_{110,}V_{111,}V_{101,}V_{100})$, $(V_{010,}V_{110,}V_{010,}V_{000})$, $(V_{101,}V_{111,}V_{011,}V_{001})$, $(V_{100,}V_{101,}V_{001,}V_{000})$ a $(V_{011,}V_{111,}V_{110,}V_{010})$ vzniká potřebná krychle pro další zpracování. Stěny jsou orientovány proti směru hodinových ručiček tak, aby normála po výpočtu dle kapitoly 3.12.6 byla orientována vně tělesa. Normály stěn mohou být také v tomto případě intuitivně nastaveny. Jejich vektory jsou jednotkové rovnoběžné se souřadnými osami a kolmé na stěnu. Normály vrcholů jsou nastaveny dle předpisu:

$$\vec{n_{V_x}}=\frac{V_x}{|V_x|}$$

Tedy normalizovaný směr od počátku k tomuto vrcholu.



Obr 8: Ukázka vygenerované krychle

3.12.2 Válec

Jednotkový válec v *CSG* stromu je definovaný jako válec s průměrem 1 s nekonečnou výškou na obě strany. Osa válce je totožná s osou **z** a prochází počátkem systému. Pro účely v knihovně *JaGrLib* je válec vytvořen konečný, tedy výška je nastavena parametricky v *CSG* stromu a je na tvůrci takového stromu, aby parametr nastavil tak, aby výsledná scéna byla stejná jako s válcem nekonečným.

Konstrukce začíná vytvořením vrcholů na podstavách. Podstavy jsou kružnice a protože *b-rep* používá pouze rovné hrany je potřeba vytvořit pravidelný *n*-úhelník jehož vrcholy leží na pomyslné kružnici podstavy. Pro zvolenou hodnotu *s*, což je jemnost válce, vytvoříme vrcholy pro dva (4*s*)-úhelníky následujícím způsobem:

```
// parametry:
//   s - jemnost válce, s > 0
//   height - výška válce
// výsledek:
//   vrcholy horní a dolní podstavy válce
function podstavy_válce(int s, double height) : (Vertex[], Vertex[])
  α = π / (2 * s)
  for(i = 0; i < s; i++)
    X = cos(α * i)
    Y = sin(α * i)
      // horní podstava
    VT[0*s + i] = ( X,  Y, -height)
    VT[1*s + i] = (-Y,  X, -height)
    VT[2*s + i] = (-X, -Y, -height)
    VT[3*s + i] = ( Y, -X, -height)
      // dolní podstava
    VB[0*s + i] = ( X,  Y,  height)
```

```
    VB[1*s + i] = (-Y,  X,  height)
    VB[2*s + i] = (-X, -Y,  height)
    VB[3*s + i] = ( Y,  X,  height)
  end for
  return (VT,VB)
end function
```

Nyní je potřeba vrcholy spojit do stěn. Boční stěny jsou vytvořeny pomocí následujícího kódu:

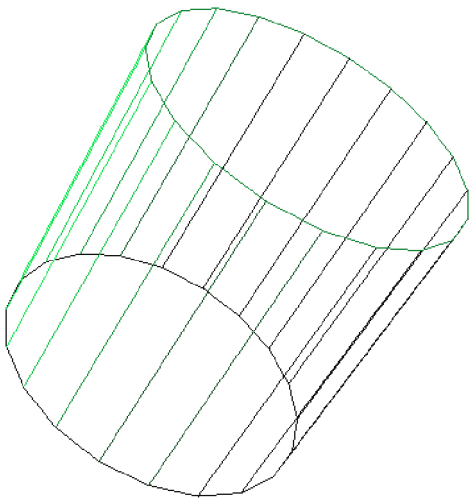
```
// parametry:
//   s - jemnost válce
//   VT - vrcholy horní podstavy (výsledek funkce podstavy_válce)
//   VB - vrcholy dolní podstavy (výsledek funkce podstavy_válce)
// výsledek:
//   vytvoří boční stěny válce
function postranní_stěny_válce(int s, Vertex[] VT, Vertex[] VB)
  for(i = 0; i < s; i++)
    for(j = 0; j < 4; j++)
      i1 = i + j * s
      i2 = (i1 + 1) mod (4*s)
      vytvoř stěnu z vrcholů (VT[i2], VT[i1], VB[i1], VB[i2])
    end for
  end for
end function
```

Stěny jsou po vytvoření orientovány proti směru hodinových ručiček tak, aby šly dopočítat normály stěn dle kapitoly 3.12.6 . Podstavy jsou vytvořeny přímo z proměnných **VT** a **VB** a to ve stejném pořadí podle indexů v případě proměnné **VT** a v opačném pořadí v případě proměnné **VB**. I v tomto případě jsou hrany stěn orientovány proti směru hodinových ručiček.

Normály jsou ve vrcholech dodefinovány normovaným vektorem od středu podstavy k vrcholu. Tedy má-li vrchol souřadnice $v = (x, y, z)$, pak normála vrcholu je

$$\vec{n}_v = \frac{(x, y, 0)}{|(x, y, 0)|} \quad .$$

Normály stěn jsou dopočítány podle kapitoly 3.12.6.



Obr 9: Ukázka vygenerovaného válce

3.12.3 Kužel

Jednotkový kužel v *CSG* stromu je nekonečný kužel oběma směry, jehož osa splývá se souřadnou osou **z**, vrchol kužele je v počátku a úhel kužele je 90°. Pro účely algoritmu je výška takového kužele od vrcholu k podstavě nastavitelná a kužel je uzavřen podstavami. Při nastavení dostatečné výšky lze dosáhnout stejných výsledků jako v případě kužele nekonečného. Vrchol v počátku má tedy souřadnice $v_0 = (0, 0, 0)$. Vrcholy podstav jsou dopočítány podle následujícího kódu podobně jako u válce:

```
// parametry:
//   s - jemnost kužele, s > 0
//   height - výška kužele
// výsledek:
//   vrcholy horní a dolní podstavy kužele
function podstavy_kužele(int s, double height) : (Vertex[], Vertex[])
  α = π / (2 * s)
  for(i = 0; i < s; i++)
    X = cos(α * i)
    Y = sin(α * i)
      // horní podstava
    VT[0*s + i] = ( X, -Y, -height)
    VT[1*s + i] = (-Y, -X, -height)
    VT[2*s + i] = (-X,  Y, -height)
    VT[3*s + i] = ( Y,  X, -height)
      // dolní podstava
    VB[0*s + i] = ( X,  Y,  height)
    VB[1*s + i] = (-Y,  X,  height)
    VB[2*s + i] = (-X, -Y,  height)
    VB[3*s + i] = ( Y, -X,  height)
  end for
  return (VT, VB)
end function
```

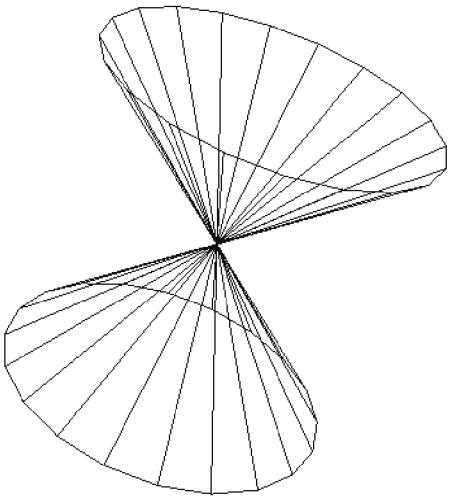
Dalším krokem je pak vytvoření samotných stěn, tedy spojení podstav s vrcholem kužele:

```
// parametry:
//   s - jemnost kužele
//   VT - vrcholy horní podstavy (výsledek funkce podstavy_kužele)
//   VB - vrcholy dolní podstavy (výsledek funkce podstavy_kužele)
// výsledek:
//   vytvoří boční stěny kužele
function prostranní_stěny_kužele(int s, Vertex[] VT, Vertex[] VB)
  V0 = (0, 0, 0)
  for(i = 0; i < VT.length; i++)
    i2 = i mod VT.length
    vytvoř stěnu z vrcholů (VT[i2], VT[i], V0)
    vytvoř stěnu z vrcholů (VB[i2], VB[i], V0)
  end for
end function
```

Vytvořené stěny mají hrany orientované proti směru hodinových ručiček, pro správný výpočet normály stěny. Podstavy jsou vytvořeny pomocí proměnných `VT` a `VB`, přímo podle pořadí jejich indexů a ve výsledku mají také správnou orientaci. Normály vrcholů podstav jsou nastaveny kolmo na hranu vedoucí ze středu kužele k tomuto vrcholu směrem od válce předpisem

$$\vec{n}_v = \frac{(x, y, z-2\cdot h)}{|(x, y, z-2\cdot h)|} \quad ,$$

kde $v = (x, y, z)$ je vrchol podstavy a h je výška kužele. Vrchol ve středu válce by měl mít normál více. Protože ale algoritmus nepočítá s vrcholem s vícenásobnou normálou, je potřeba zaznamenat k vrcholu odkaz na normálu pro každou stěnu, ve které je. Normála bude pak stejná jako normála stěny a může být dopočítána podle kapitoly 3.12.6.



Obr 10: Ukázka vygenerovaného kužele

3.12.4 Koule

Jednotková koule v *CSG* je koule o průměru 1 se středem v počátku souřadného systému. Taková koule může být převedena do polytopu dvěma základními způsoby. Prvním ze způsobů je vytvoření stěn s hranami podobně jako rovnoběžky a poledníky na zeměkouli, nebo postupnou triangulací pravidelného osmistěnu zvaném též *HTM* (*Hierarchical Triangular Mesh*) popsaném v [18]. Výsledek tvorby způsobem, popsaným v tomto článku, však není vyhovující, protože nepočítá se sdílenými vrcholy.

Koule jako zeměkoule

Při sestrojování polytopu koule podle vzoru rovnoběžek a poledníků zeměkoule jsou nalezeny nejdříve všechny vrcholy a to pomocí dvou úhlů (stejně jako šířka a výška při určování polohy na zeměkouli). Zadaná jemnost s říká na jaké úhly bude koule rozdělena. Následující kód vytváří takové vrcholy:

```
// parametry:
//   s - jemnost koule, s > 0
// výsledek:
//   vrcholy koule
function vrcholy_koule(int s) : Vertex[]
  a = π / (2 * s + 2)
  counter = 0
  for(e = -s; e <= s; e++)
    r_e = cos(a * e)
    y_e = sin(a * e)
    for(d = 0; d < (4 * s + 4); d++)
      z_d = r_e * sin(a * d) * (-1)
      x_d = r_e * cos(a * d)
      V[counter] = (x_d, y_e, z_d)
      counter++
    end for
  end for
  V[counter]   = (0,  1, 0)
  V[counter+1] = (0, -1, 0)
  return V
end function
```

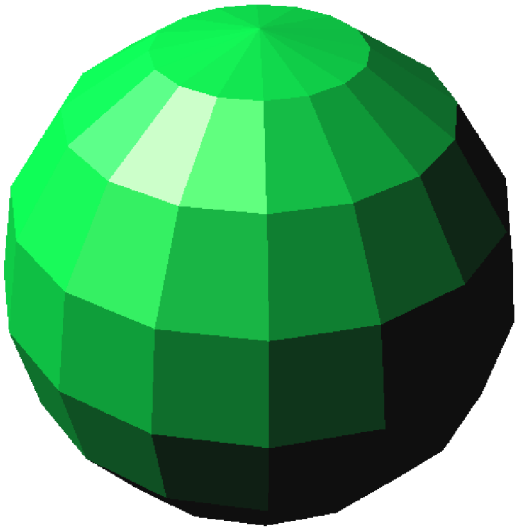
Po vytvoření vrcholů následuje sestrojení stěn polytopu koule:

```
// parametry:
//   s - jemnost koule, s > 0
//   V - vrcholy koule (výsledek funkce vrcholy_koule)
// výsledek:
//   vytvoří stěny koule
function stěny_koule(int s, Vertex[] V)

  for(e = 0; e < 2 * s; e++)
    for(i = 0; i < (4 * s + 4); i++)
      vytvoř stěnu z vrcholů:
          V[e * (4 * s + 4) + i],
          V[e * (4 * s + 4) + (i + 1) mod (4 * s + 4)],
          V[e * (4 * s + 4) + (i + 1) mod (4 * s + 4) + (4 * s + 4)],
          V[e * (4 * s + 4) + i + (4 * s + 4)]
    end for
  end for

  // trojúhelníkové stěny u horního a dolního vrcholu
  for(i = 0; i < (4 * s + 4); i++)
    vytvoř stěnu z vrcholů:
        V[e * (4 * s + 4) + i],
        V[e * (4 * s + 4) + (i + 1) mod (4 * s + 4)],
        V[(4 * s + 4) * (2 * s + 1)]
    vytvoř stěnu z vrcholů
        V[i],
        V[(4 * s + 4) * (2 * s + 1) + 1],
        V[(i + 1) mod (4 * s + 4)],
  end for
end function
```

Vytvořené hrany stěn jsou orientované proti směru hodinových ručiček, tedy tak aby šlo podle kapitoly 3.12.6 spočítat správné směry stěnových normál. Normály vrcholů pak mají shodné souřadnice jako bod samotný (směr je od středu k vrcholu a tato vzdálenost je díky jednotkové kružnici právě jedna).



Obr 11: Ukázka vygenerované koule "s rovnoběžkami a poledníky"

Koule pomocí HTM

Polytopový obraz koule v tomto případě vzniká ze základního pravidelného osmistěnu se souřadnicemi (1,0,0), (-1,0,0), (0,1,0), (0,-1,0), (0,0,1), (0,0,-1). Každý trojúhelník je pak rozdělen svými příčkami na čtyři menší trojúhelníky, jejichž vrcholy jsou následně zobrazeny na kouli. Rekurzivním voláním na každém trojúhelníku lze pak dosáhnout jemnějšího vzhledu. Protože vrcholy polytopu musí být sdílené, jsou v globální proměnné uloženy odkazy na vrcholy tvořící střed nějaké hrany. Pokud je tedy udaná hloubka s rekurzivního volání, lze generovat HTM tímto kódem:

```
//střed je globální mapování dvou vrcholů na vrchol třetí. Platí, že
// střed(X,Y) = střed(Y,X)
//na začátku je tato mapa prázdná

// parametry:
//   s - jemnost koule - počet iterací, s >= 0
// výsledek:
//   vytvoří stěny HTM koule
function HTM_koule(int s)
  // osmistěn
  O[0] = ( 1,  0,  0)
  O[1] = (-1,  0,  0)
  O[2] = ( 0,  1,  0)
  O[3] = ( 0, -1,  0)
  O[4] = ( 0,  0,  1)
  O[5] = ( 0,  0, -1)

  HTM_triangulace({O[4], O[0], O[2]}, s)
  HTM_triangulace({O[1], O[4], O[2]}, s)
  HTM_triangulace({O[5], O[1], O[2]}, s)
  HTM_triangulace({O[0], O[5], O[2]}, s)
  HTM_triangulace({O[0], O[4], O[3]}, s)
```

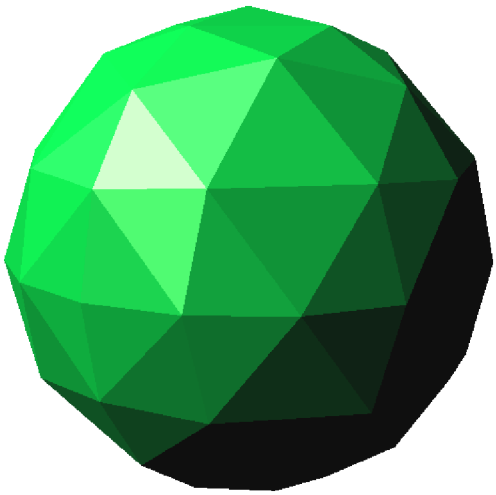
```
  HTM_triangulace({O[4], O[1], O[3]}, s)
  HTM_triangulace({O[1], O[5], O[3]}, s)
  HTM_triangulace({O[5], O[0], O[3]}, s)
end function

// parametry:
//   V - vrcholy trojúhelníku k jemnější triangulaci
//   iter - počet iterací dělení na tomto trojúhelníku
// výsledek:
//   rozdělí trojúhelník na menší a zavolá se rekurzivně na nich
//   pokud se rekurze dostala na konec, vytvoří stěny koule
function HTM_triangulace(Vertex[] V, int iter)
  if(iter = 0) then
    vytvoř stěnu z vrcholů (V[0], V[1], V[2])
    return
  end if


  for(i = 0; i < 3; i++)
    i2 = (i + 1) mod 3
    if(střed(V[i], V[i2]) neexistuje) then
      size  = (V[i].x + V[i2].x)^2
      size += (V[i].y + V[i2].y)^2
      size += (V[i].z + V[i2].z)^2
      size = sqrt(size)
      střed(V[i], V[i2]) = ((V[i].x + V[i2].x)/size,
                            (V[i].y + V[i2].y)/size,
                            (V[i].z + V[i2].z)/size)
    end if
    mid[i] = střed(V[i], V[i2])
  end for

  // rekurzivní volání
  HTM_triangulace({mid[2], V[0], mid[0]}, iter - 1)
  HTM_triangulace({mid[1], mid[0], V[1]}, iter - 1)
  HTM_triangulace({V[2], mid[2], mid[1]}, iter - 1)
  HTM_triangulace({mid[0], mid[1], mid[2]}, iter - 1)
end function
```

Algoritmus vytvoření polytopu koule je opět utvořen tak, aby stěny byly orientovány proti směru hodinových ručiček a výpočet normály (dle kapitoly 3.12.6) vyšel správně. Normály vrcholů jsou pak stejné jako souřadnice vrcholů samotných.



Obr 12: Ukázka vygenerované koule pomocí HTM

3.12.5 Poloprostor

Poloprostor je v *CSG* stromu definovaný rovinou splývající s rovinou os **x** a **y** souřadného systému. Normála této roviny je v kladném směru osy **z**. Pro účely algoritmu je však potřeba zadat okraje alespoň této roviny a to například pomocí čtverce o dostatečné nebo lépe nastavitelné velikosti. Poloprostor může být použit pouze v operaci průniku nebo v rozdílu jako druhé těleso a to s tělesem uzavřeným. Důležité také je, aby algoritmus nenašel žádný průsečík s žádnou pomocnou hranou roviny (proto je nutné mít nastavenou velikost dostatečně velikou).

Poloprostor je pak sestrojen jako jediná stěna s vrcholy (-*t*, -*t*, 0), (*t*, -*t*, 0), (*t*, *t*, 0), (-*t*, *t*, 0), kde *t*, je ona nastavitelná velikost. Normála je nastavena na (0,0,1).

3.12.6 Výpočet normály nekonvexního polygonu

Algoritmus pro výpočet normály nekonvexního polygonu je proveden pomocí metody „Newell's method“ popsané v [19]. Při použití vektorového součinu dvou hran polygonu (výsledkem je kolmice na ně, tedy i směr normály polygonu) může dojít v nekonvexních částech k výpočtu normály opačné. Je tedy možné použít vektorový součin avšak jen u takových polygonů, které jsou konvexní. „Newell's method“ vypočítává celou rovnici roviny polygonu v normalizovaném tvaru. Použit tedy může být i její poslední parametr (vzdálenost roviny od počátku – parametr *d* v rovnici roviny $n_x \cdot x + n_y \cdot y + n_z \cdot z - d = 0$. Hodnotu lze také získat dosazením jednoho vrcholu polygonu do rovnice při již známé normále. Celkový výpočet dle této metody trvá $O(n)$, kde *n*

je počet vrcholů, tedy i hran v polygonu.

3.12.7 Transformace polytopu

Je-li sestrojen polytop, je potřeba jej dle transformační matice (***M***) zadané u listu *CSG* stromu přesunout, otočit či změnit velikost. Je potřeba tímto způsobem přesunout všechny vrcholy, ale také všechny normály a to jak u stěn tak i vrcholů. Transformace vrcholu $\vec{v}$ se provádí jednoduchým způsobem a to maticovým součinem $\vec{v_T} = \boldsymbol{M} \cdot \vec{v}$, transformace normály $\vec{n}$ je pak dána předpisem $\vec{n_T} = (\boldsymbol{M}^{-1})^T \cdot \vec{n}$, tedy k transformaci se použije transformovaná inverzní matice. Tato může být pro každou matici v *CSG* stromu vypočítána jednou.

Pokud polytop A má v_A vrcholů a f_A stěn, je potřeba transformaci provést na každém vrchole (samotný vrchol a normála vrcholu) a na každé stěně (normála stěny), a transformace celého polytopu trvá $O(v_A + f_A)$.

3.13 Časová složitost

Popsaný algoritmus množinové operace dvou těles se sestává z následujících kroků s uvedenými časy:

- Vyhledání průsečíků – $O((e_A + e_B) + (a_A \cdot e_A \cdot f_B + a_B \cdot e_B \cdot f_A) + b_A \cdot b_B \cdot e_A \cdot e_B)$, nebo $O(e_A \cdot e_B)$ (kap. 3.8).
- Procházení po původních vrcholech těles – $O((e_A + e_B) \cdot \log k)$ (kap. 3.11.4).
- Procházení po průsečících – $O(k)$ (kap. 3.11.5)
- Kontrola netknutých stěn – $O((k + f_A) \cdot f_A)$ (kap. 3.11.3)
- Vkládání nových hran do struktury – $O(\sum_{f \in A \cup B} h(f)^2)$, kde $h(f)$ je počet vzniklých hran na stěně f (kap. 3.11.2).
- Krajní případy – v nejhorším případě $O(e_A \cdot e_B)$ (kap. 3.11.7)

Z uvedených časů je zřejmé, že nejvíce zatěžující částí je vyhledávání průsečíků. Všechny ostatní časy jsou nižší, tedy čas $O(e_A \cdot e_B)$ je horním odhadem. Budeme-li počítat čas procházení se zanedbatelným množstvím krajních případů, dosahuje algoritmus času

$$O((e_A + e_B) \cdot \log k + k + (k + f_A) \cdot f_A + \sum_{f \in A \cup B} h(f)^2),$$

kde e_A, e_B, f_A, f_B jsou počty hran v tělesech A a *B*, k je počet průsečíků.

Je tedy vidět, že čas závisí i na vzájemné poloze (*k*) a provedené operaci či jemnosti ($h(f)$).

3.14 Implementace v JaGrLib

V knihovně *JaGrLib* byl pro účely implementace vytvořen modul **`CSG2BrepImpl`**. Tento modul je potomek třídy **`Piece`** a implementuje rozhraní (plug) **`Trigger`**, pro spouštění samotného algoritmu. Výstupními zásuvkami jsou pro přístup k datům **`RTScene`** (pro získání *CSG* stromu) a **`Brep`** (pro modifikaci a zápis do *b-rep*). Metoda **`fire`** zděděná po rozhraní **`Trigger`** spouští algoritmus tím, že získá *CSG* strom (odkaz na jeho kořen) z připojeného modulu přes zásuvku **`RTScene`**. Na něm pak rekurzivně prochází strom (kap. 3.7), kde aplikuje množinovou operaci (kap. 3.11.3). Použité struktury jsou popsány dříve v kapitolách 3.9 a 3.11.1. Po skončení procházení vždy smaže staré stěny a hrany, přičemž vrcholy ponechává a následně vytváří stěny nové z již existujících vrcholů.

Vrcholy *CSG* stromu jsou reprezentovány třídou **`SceneNode`**, a odkaz na kořen stromu, získaný pomocí metody **`getRoot`** rozhraní **`RTScene`**, pak nese celý strom. Takovýto vrchol může být pak typu **`CSGNode`** nebo **`CSGLeaf`**, tedy buď vnitřní vrchol nebo list. Pokud se jedná o list, třída umožňuje získat typ jednoduchého tělesa (instanci jedné ze tříd **`Cube`**, **`Cylinder`**, **`Cone`**, **`Sphere`** a **`Plane`**, nebo jiného potomka třídy **`Solid`**) a transformační matici tohoto tělesa. Parametry jsou spolu s odkazem na **`Brep`** předány metodě **`createPolyhedr`**, která je implementována ve všech používaných jednoduchých tělesech (potomcích třídy **`Solid`**) a která vrací odkaz do **`Brep`** na nově vytvořený polytop.

Z rozhraní **`Brep`** jsou pak využívány tyto metody:

- **`getFaceVertices`** – získává uspořádané vrcholy stěny.
- **`edgeInFaceIterator`** – slouží k procházení hran stěny.
- **`getEdgeRecord`** – získá údaje o hraně (odkazy na vrcholy, levou a pravou stěnu a předchozí a následující hranu jak na levé tak na pravé stěně.
- **`setFaceVertices`** – nastavuje vrcholy stěně.
- **`faceInSolidIterator`** – prochází všechny stěny v tělese.
- **`vertexInFaceIterator`** – prochází vrcholy ve stěně.

- **`resetSolid`** – vymaže záznam o tělesu z *b-rep*, ponechává vrcholy, hrany i stěny.
- **`getVertexCoords`** – vrátí souřadnice vrcholu.
- **`createSolid`** – vytvoří nové těleso.
- **`createFace`** – vytvoří novou stěnu.
- **`createAttribute`** – vytvoří nový atribut.
- **`getAttribute`** – získá hodnotu atributu u daného objektu (vrchol, hrana, stěna).
- **`setAttribute`** – nastaví hodnotu atributu u daného objektu.
- **`insertFaceIntoSolid`** – vloží stěnu do zadaného tělesa.

Metody umožňují tedy veškeré operace popsané v algoritmu. V atributech jsou ukládány normály stěn (celá rovnice roviny) i vrcholů, uchovávány jsou spočítané vzdálenosti vrcholů od stěn a hraniční kvádry stěn.

Nové stěny mohou obsahovat díry a proto jsou všechny smyčky vrcholů, které byly přiřazeny k jedné stěně staré, uloženy k jedné nově vytvořené stěně. Při vykreslování nové scény pomocí **`GLBrepRender`** pak musí být použit tessalátor, který takovou strukturu (více smyček vrcholů různě orientovaných) umí správně zobrazit.

Protože je v průběhu provádění ovlivňována přímo datová struktura v implementaci **`Brep`**, metoda **`fire`** nevrací žádný záznam z této struktury. Pouze indikaci, že algoritmus dopadl dobře, tedy hodnotu **`true`**.

4. Závěr

Diplomová práce vznikala ve dvou fázích – zjištění možností a implementace hardwarové akcelerace v knihovně *JaGrLib* a návrh a implementace zobrazování scén *CSG* v knihovně.

Před samotnou implementací hardwarové akcelerace byly hledány možné prostředky na její realizaci. Byla nalezena vhodná knihovna *JOGL* implementující rozhraní *OpenGL* a bylo potřeba nastudovat její možnosti a způsoby použití. Dalším krokem bylo nastudování specifikace *OpenGL* pro zjištění možností samotné akcelerace. Před zavedením do projektu *JaGrLib* bylo potřeba prostudovat jeho dokumentaci a seznámit se s principy v něm používanými. Na základě těchto znalostí byly navrženy nové moduly a rozhraní v knihovně a následně také implementovány. Nakonec byly vytvořeny kompozice projektu s příklady použití nových modulů.

Druhá fáze obnášela seznámení se s problematikou zobrazování *CSG* scén na grafických akcelerátorech a zvolení jednoho ze způsobů – převodu do *b-rep* – vhodného pro knihovnu *JaGrLib*. Byly vyhledány a nastudovány materiály, které se převodem zabývají, a byl navržen nový postup při takovém převodu. Postup se skládá z dílčích algoritmů, u nichž byly hledány optimální varianty tak, aby celkový převod byl optimalizován. Postup byl nakonec realizován v knihovně *JaGrLib* jako samostatný modul, jehož výstup – scéna v *b-rep* – může být dále používán nejen k zobrazení na monitoru.

Jedním z přínosů práce je ve zvolené knihovně pro akceleraci pomocí *OpenGL – JOGL*. Jak se ukázalo, pokrývá většinu specifikace *OpenGL* a je tak použitelná nejen pro projekt *JaGrLib*. Dále lze za přínos považovat i samotnou implementaci v knihovně *JaGrLib*, která dále umožňuje vyvíjet a navrhovat nové algoritmy a datové struktury pro počítačovou grafiku. Přínos převodu *CSG* do *b-rep* je ukázání další možnosti jak problém realizovat za pomoci optimálních dílčích algoritmů.

Závěrem lze říci, že předem stanovené cíle byly splněny a realizovány. Přesto zde zůstávají možnosti jak na rozšíření hardwarové podpory *JaGrLib*, tak i na vylepšení algoritmu převodu *CSG* do *b-rep*. Specifikace *OpenGL* je rozsáhlá a byly pokryty pouze důležité parametry pipeline. Rozšířením by tedy mohlo být přidání dalších parametrů. Další možností je obohatit projekt o možnost využití nezobrazovaného *bufferu* (tzv. **pbuffer**), který umožňuje vykreslovat scénu pouze do paměti a dále ji využít např. jako texturu (tato metoda je používána k simulaci zrcadel). Algoritmus pro převod scén lze pak vylepšit např. optimalizací procházení vrcholů po stěnách bez průsečíků tím, že jsou stěny zařazeny do výsledné scény automaticky. Jinou možností je více optimalizovat hledání

průsečíků např. zvolením a vytvořením vhodných hraničních objektů různých částí *b-rep* těles.

5. Reference

[1] Josef Pelikán. *JaGrLib - library for computer graphics.* MFF-UK. http://cgg.mff.cuni.cz/JaGrLib/

[2] Mark Segal, Kurt Akeley. *The OpenGL Graphics System: A Specification (version 2.0).* Silicon Graphics. Inc., 2004.

[3] Norman Chin a kol. *The OpenGL Graphics System Utility Library (version 1.3)*, Silicon Graphics, 1998.

[4] *Java bindings for OpenGL.* Sun Microsystems. https://jogl.dev.java.net/

[5] *Creating a GUI with JFC/Swing.* Sun Microsystems. http://java.sun.com/docs/books/tutorial/uiswing/

[6] *Cg Language Specification.* NVidia Corp., 2009.

[7] Randima Fernando, Mark J. Kilgard. *The Cg Tutorial.* NVidia Corp., 2003.

[8] John Kessenich. *The OpenGL Shading Language.* Silicon Graphics, Inc., 2008.

[9] Nigel Stewart, Geoff Leach, Sabu John. *An Improved Z-Buffer CSG Rendering Algorithm.* 1998.

[10] J. Goldfeather, S. Molnar, G. Turk, H. Fuchs. *Near Real-Time CSG Rendering Using Tree Normalization and Geometric Pruning.* 1989.

[11] David H. Laidlaw a kol. *Constructive Solid Geometry for Polyhedral Objects.* 1986.

[12] A. Requicha, H. Voelcker. *Boolean Operations in Solid Modelling: Boundary Evaluation and Merging Algorithms.* 1985.

[13] William C. Thibault, Bruce F. Naylor. *Set operations on polyhedra using binary space partitioning trees.* 1987.

[14] Günther Greiner, Kai Hormann. *Efficient Clipping of Arbitrary Polygons.* 1998.

[15] Thomas Akeine-Möller a kol. *Real-Time Rendering, Third Edition.* 2008.

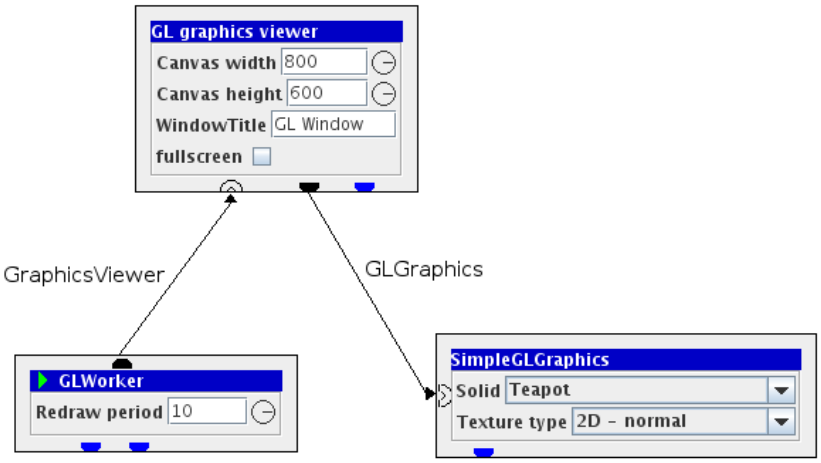
[16] Paul Bourke. *Determining if a point lies on the interior of a polygon.* 1987.

[17] *Java Reference documentation*, Sun Microsystems, inc., http://java.sun.com/reference/docs/

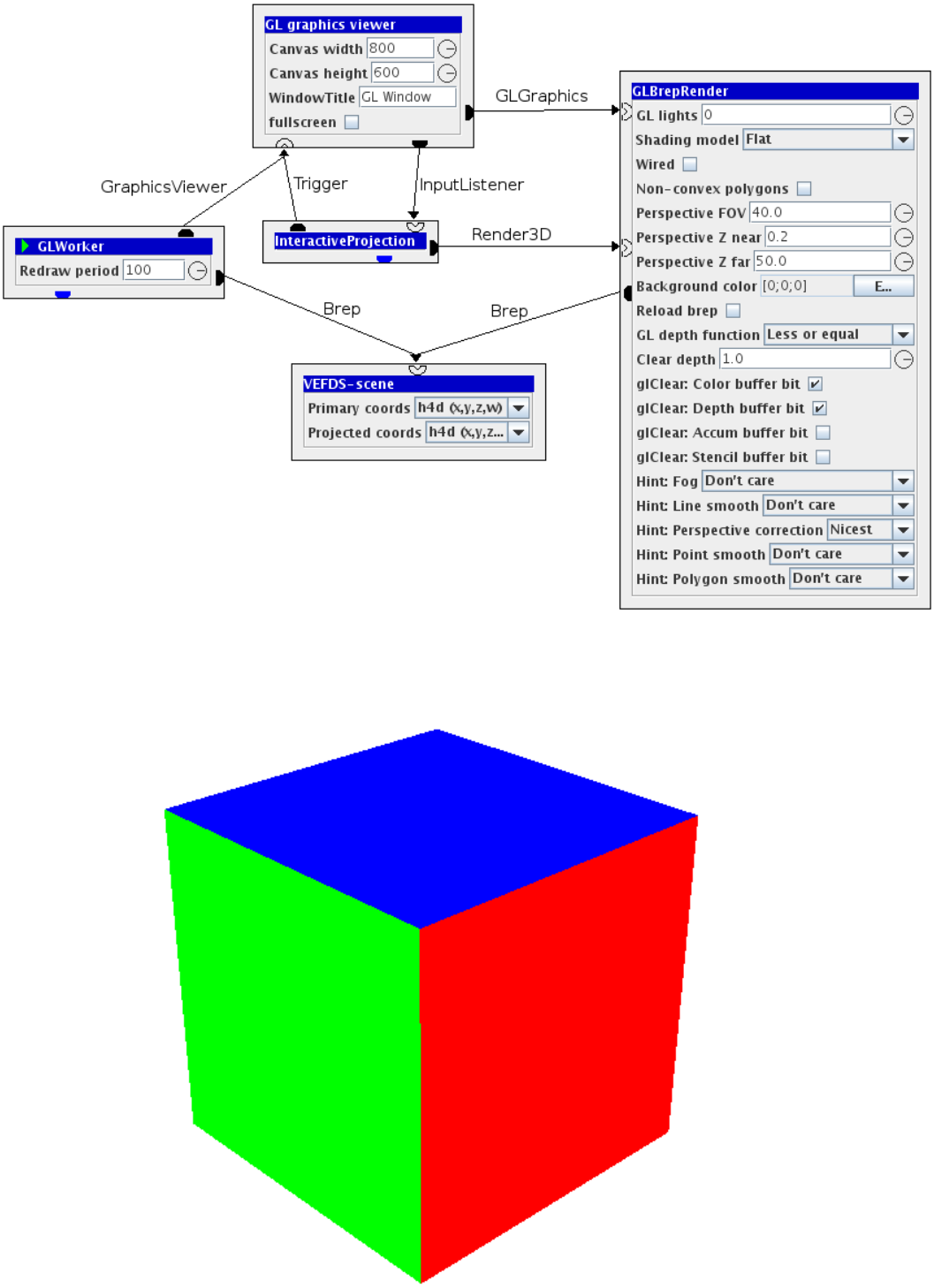
[18] P. Z. Kunszt, A. S. Szalay, A. R Thakar. *The Hierarchical Trangular Mesh.* 2001.

[19] Fillipo Tampiery. *Newell's method for computing the plane equation of a polygon.* 1992.

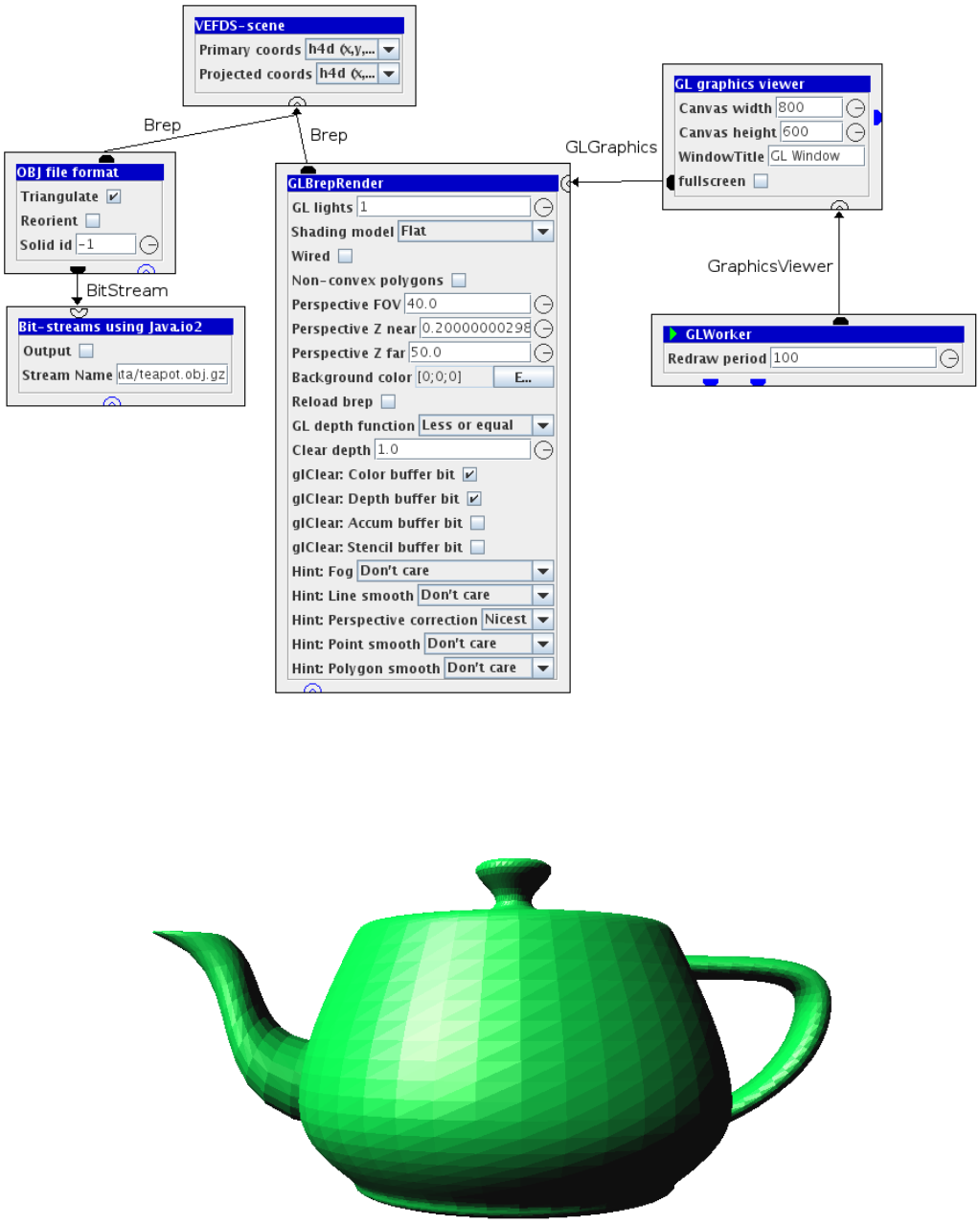
Příloha A. Příklady kompozic



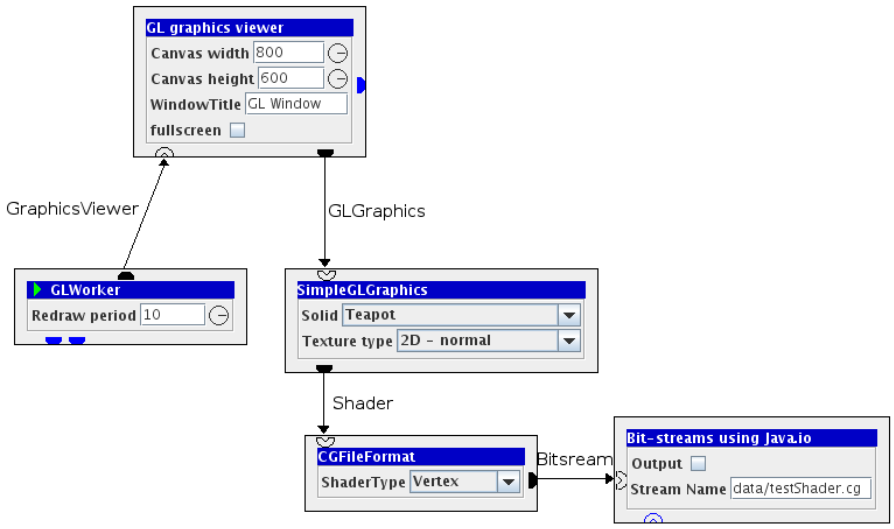
*Příklad kompozice a jejího výstupu jednoduché scény, vykreslené za pomoci **SimpleGLGraphics***



*Příklad kompozice a jejího výstupu jednoduché scény b-rep generované v modulu **GLWorker** s použitím **InteractiveProjection**.*

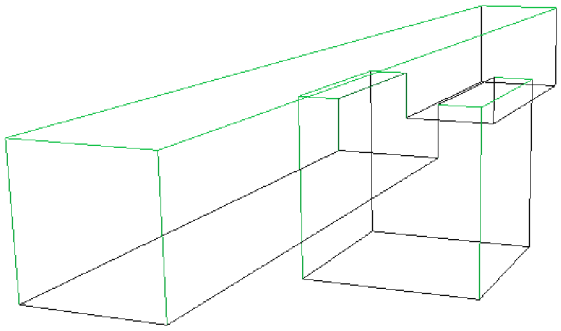


Příklad kompozice a jejího výstupu jednoduché scény b-rep získané ze souboru ve formátu OBJ (jednoduchý textový formát pro specifikování b-rep – zde je použita scéna s konvicí).

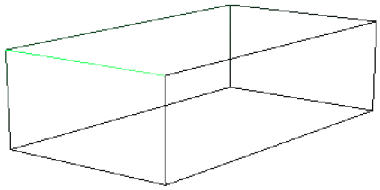


Příklad kompozice pro použití vertex shaderu (v tomto případě CG) ze souboru.

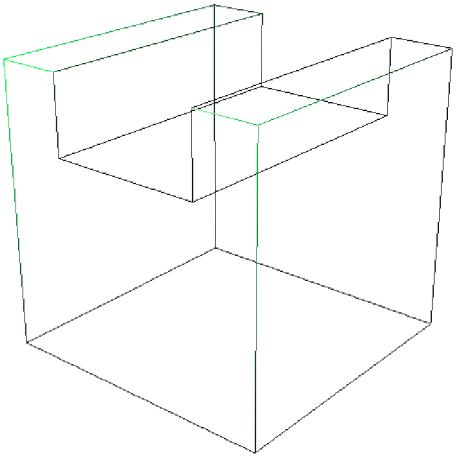
Příloha B. Ukázky převedených scén z CSG



Sjednocení

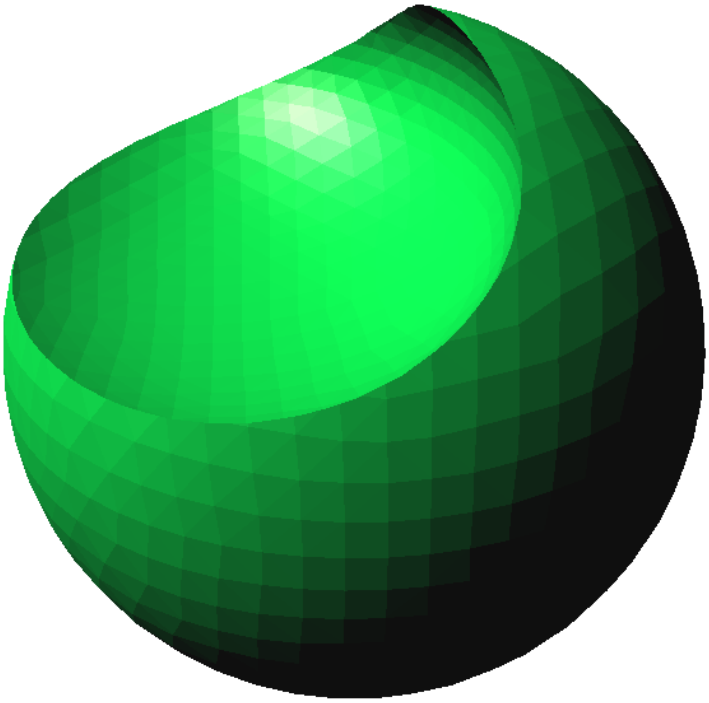


Průnik



Rozdíl

Ukázky výsledků různých operací nad dvěma kvádry.



*Ukázka rozdílu koule a elipsoidu HTM za použití **flat** stínování*

Příloha C. Obsah CD

Součástí diplomové práce je i přiložený CD-ROM obsahující text práce a zdrojové kódy projektu *JaGrLib* rozšířeného o třídy na podporu akcelerované grafiky a převod CSG na *b-rep*.
CD-ROM obsahuje následující soubory a adresáře:

- **`obsah.txt`** – soubor s popisem obsahu CD-ROM.
- **`/text/diplomova-prace.pdf`** – soubor s textem diplomové práce.
- **`/jagrlib`** – adresář se zdrojovými soubory knihovny *JaGrLib* včetně nových tříd. Struktura odpovídá dokumentované struktuře projektu.
- **`/jagrlib-opengl`** – adresář pouze s novými třídami a kompozicemi. Struktura opět odpovídá dokumentované struktuře projektu, avšak není možné projekt žádným způsobem v tomto adresáři spustit.