Univerzita Karlova v Praze
Matematicko-fyzikální fakulta

BAKALÁŘSKÁ PRÁCE

Jakub Gemrot

Mapování, zpracování a reprezentace map pro Unreal Tournament boty

Katedra software a výuky informatiky

Vedoucí bakalářské práce: Mgr. Cyril Brom

Studijní program: programování

2006

Na tomto místě chci poděkovat především svému vedoucímu práce Cyrilu Bromovi za jeho trpělivost a cenné rady.

Dále bych rád poděkoval svým rodičům, bez jejichž podpory by tato práce nikdy nevznikla.

Jsem vděčný Michalovi Bídovi a Ondrovi Burketovi za pomoc při práci na framework Pogamut a diskuze kolem Unreal Tournamentu.

V neposlední řadě patří poděkování GA UK, jelikož jsem měl možnost zúčastnit se prací v rámci grantu 351/2006/A-INF/MFF a získat tak cenné zkušenosti pro svou bakalářskou práci.

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

V Praze dne 7. 8. 2006 Jakub Gemrot

Obsah

strana

Název práce: Mapování, zpracování a reprezentace map pro Unreal Tournament boty
Autor: Jakub Gemrot
Katedra: Katedra software a výuky informatiky
Vedoucí bakalářské práce: Mgr. Cyril Brom
e-mail vedoucího: brom@ksvi.mff.cuni.cz

Abstrakt: Ve většině komerčních her se pro pohyb počítačem ovládaných postav ve hře využívají navigační body. Z navigačních bodů se vytváří souvislý graf, který pokrývá nejdůležitější místa ve hře, kam postava může dojít. Se zvyšujícím se výkonem stolních počítačů se začínají používat sofistikovanější postupy pro pohyb postav, jmenovitě pak metoda NavMesh. NavMesh je množina konvexních polygonů souvisle pokrývající místa ve hře, kudy postavy mohou chodit. Tento postup používají například nedávno vydané tituly Half-Life 2 a F.E.A.R. Bohužel tyto komerční produkty nejsou přístupny akademické sféře, a tak neexistuje možnost pro experimentování s NavMesh jako podkladem pro pohyb počítačem ovládaných postav ve hře. Možnost ovládat postavu však existuje pro starší hru Unreal Tournament, který používá navigační body. V práci diskutuji výhody NavMesh oproti starší metodě navigačních bodů. Přínosem této práce je možnost automatického generování NavMesh pro mapy Unreal Tournamentu. NavMesh vytvářím prostřednictvím algoritmu z oblasti počítačové grafiky pro převod prostředí Unreal Tournamentu z CSG reprezentace do hranové reprezentace, ze které lze již jednoduše zkonstruovat NavMesh. Součástí práce je také demonstrace použití takto vytvořeného NavMeshe pro navigaci postavy na mapách pro Unreal Tournament.

Klíčová slova: umělá inteligence, FPS hry, pohyb botů, NavMesh

Title: Emotion bots in the enviroment of the game Unreal Tournament
Author: Jakub Gemrot
Department: Department of Software and Computer Science Education
Supervisor: Mgr. Cyril Brom
Supervisor's e-mail address: brom@ksvi.mff.cuni.cz

Abstract: In many computer games the navigation points are used for the movement of computer-controled players. With the growing power of desktop computers more sophisticated methods are used for the movement of players, namely the method of NavMesh. NavMesh is a set of convex polygons continuously covering the reachable places inside the game by characters. This approach was used in the recently produced games Half-Life 2 and F.E.A.R. Those commercial products can't be used by the academy, therefore the chance to experiment with NavMesh doesn't exist. But there is a simple way to control the characters in environment of older game Unreal Tournament. This work is about the advantages of using NavMesh against older approach of navigation points. This work then presents the way how to automatically generate the NavMesh for the maps of Unreal Tournaments. NavMesh is created using the algorithm from computer graphics for the conversion from CSG representation of the map into boundary representation out of which the NavMesh can be generated. The work also contains a demonstration of usage of NavMesh for the movement of players in Unreal Tournament's maps.

Keywords: artifical inteligence, FPS games, bot's movement, NavMesh

1 Úvod

Vzrůstající výkon stolních počítačů dovoluje programovat hry, které jsou vizuálně čím dál tím blíže realitě. Výkonné grafické karty dovolují v reálném čase vykreslovat stíny, světelné lomy způsobené vodou, dokáží zaoblit ostré hrany polygonů. Objekty ve hrách mají také svůj fyzikální model, a proto veškeré manipulace s předměty vypadají věrohodně.

Hry se také snaží vytvořit virtuální umělé postavy. Tyto postavy mají ve světě plnit různé úkoly, které jsou buď jednoduché a omezují se na sdělení nějaké informace hráči, anebo mohou být velice komplexní. Postavy mohou mít za úkol například s hráčem bojovat, či s ním vyjednávat, nebo si jen tak povídat. Programování těchto postav tak spadá do oboru umělé inteligence. Hráči vysoce hodnotí ty hry, ve kterých se postavy chovají přirozeně a lidsky.

Jeden z typů her, jejichž hratelnost závisí na počítačem řízených postavách, jsou *first-person-shooter* (*FPS)* hry. Tyto hry vytváří virtuální 3D svět, který hráč vidí z pohledu postavy, za kterou hraje. Interakce mezi postavami se omezuje převážně na boj pomocí nejrůznějších střelných zbraní. V některých z těchto her hráč bojuje v týmu spolu s dalšími postavami ze hry a společně se snaží plnit nějaký úkol.

V rámci těchto her pak aplikace generuje a řídí rovněž postavy, které se aktivně účastní celé akce a jsou plně ovládány programem jako takovým (dále jen *bot*, *boti*).

Především pak hry, ve kterých hráči hrají v týmech, kladou na postavy kontrolované počítačem vysoké nároky, neboť ty by měly spolu navzájem spolupracovat na splnění úkolu týmu.

Výše popsané hry se tak stávají zajímavými pro obor umělé inteligence, neboť nabízí prostředí pro výzkum v oblasti lidského myšlení, řešení problémů v rámci first-person-shooter hry a zkoumání lidských dovedností. Hráči pak při hře interagují s boty a mohou zpětně říci, jak dobře boti napodobují lidské chování.

Aby boti byli schopni cokoli ve virtuálním prostředí udělat, potřebují nějaký jeho model. Čím více informací o virtuálním světě bude tento model obsahovat, tím více se boti budou schopni svým konáním přiblížit člověku, neboť i chování člověka je omezováno stejným virtuálním prostředím. Na druhou stranu není žádoucí, aby tento model byl příliš detailní, neboť by bylo výpočetně složité s ním pracovat. Právě FPS hry takový model hledají, protože potřebují značný výpočetní výkon na zobrazení virtuálního světa a na počítání fyzikálních modelů předmětů světa. Model by minimálně měl botovi umožnit se v prostředí pohybovat a umožnit mu vnímat svou polohu vůči ostatním předmětům a postavám ve hře.

2 Cíle práce

V práci se zabývám pohybem počítačem kontrolovaných postav ve virtuálním 3D prostředí reprezentovaném pomocí polygonů. Aplikaci zvoleného postupu pak ukazuji konkrétně v prostředí komerční hry Unreal Tournament [1] (dále *UT*). Protože techniky a teorie, které zde popisuji, vzešly původně z oblasti počítačových her, dovoluji se držet již zažitých pojmů z této oblasti.

Cílem práce je:

- ukázat problémy tvorby botů a vztah těchto problémů k modelu prostředí,
- popsat a porovnat modely virtuálního 3D světa, a to navigačních bodů a navigační mřížky,
- vytvořit a prezentovat způsob tvorby navigační mřížky pro mapy ze hry Unreal Tournament,
- demonstrovat použití zkonstruované navigační mřížky v UT.

3 Struktura práce

Nejprve se zmíním o historii her podobných Unreal Tournamentu a ukáži, v čem jsou zajímavé pro obor umělé inteligence. Zmíním se také o dvou dalších hrách – Half-Life [2] a Quake II [3] a poukáži na výhodu Unreal Tournamentu oproti nim, co se týče možnosti experimentování s programováním postav pro virtuální prostředí.

Zmíněné hry vytváří virtuální 3D prostředí blízké našemu, ve kterém jednotlivé postavy bojují proti sobě. Těmto prostředím se říká *lokace* nebo *mapy*. V prostředí se pak pohybují postavy *(avataři)*, které jsou buď ovládány člověkem *(hráč)* nebo počítačem *(bot)*. Popíši hlavní problémy, které boti musí v podobných hrách řešit, a po té se budu zabývat problémem vytvoření modelu prostředí pro potřeby botů. Prostředí se v převážné většině starších her reprezentuje pomocí navigačních bodů, které tvoří souvislý graf pokrývající dostupná místa ve hře. Bot nevnímá toto prostředí jako člověk, ale vnímá jej jako graf bodů v 3D prostoru. Tyto navigační body a jejich využití popíši podrobněji a ukáži omezení této reprezentace.

Následně představím blíže hru Unreal Tournament, ukáži, jak UT boti fungují a jak se dokáží vypořádat s problémy navigačních bodů. Pak popíši ideu navigační mřížky *(NavMesh)* a porovnám její výhody a nevýhody oproti navigačním bodům.

Hlavní část práce je věnována možnosti automatického generování NavMesh z dat, které reprezentují prostředí v UT. K vygenerování NavMesh bylo zapotřebí rozluštit strukturu T3D souborů, která není dokumentována a do kterých UT exportuje své prostředí. Pro zpracování dat z T3D souboru jsem implementoval algoritmus pro převod prostředí z *CSG* reprezentace, tj. reprezentace pomocí konstruktivní geometrie, do hranové reprezentace, ze které lze vygenerovat NavMesh [4].

Nakonec stručně představím middleware PoGamUT, který jsem vytvořil spolu s kolegy Michalem Bídou a Ondrou Burketem. Middleware spolupracuje s UT a poskytuje programátorovi možnost vytvářet nové boty. Pomocí tohoto middleware jsem vytvořil jednoduchého bota používajícího vygenerovaný NavMesh pro pohyb prostředím UT.

4 Počítačové hry typu first-person-shooter

4. 1 Unreal Tournament a FPS hry

Unreal Tournament je hra typu first-person-shooter, v českém žargonu „3D střílečka“. Hráč ovládá vždy jednoho avatara a dívá se na svět hry jeho pohledem – proto se tento typ her označuje jako *first-person*. Hráč má v prostředí většinou jednoduché cíle – pomocí střelných zbraní zneškodnit co nejvíce protivníků a přežít – odtud *shooter*.

FPS hry také nabízejí různé herní modifikace *(mody)*, které hráčům stanovují odlišné cíle. Nejznámější mod je *death-match*, ve kterém všichni avataři bojují proti sobě navzájem. Z dalších modů uvedu jako příklad *capture the flag*, který se hraje v týmech. Každý tým má svou vlajku, kterou musí bránit před ostatními týmy, a zároveň se snaží cizí vlajky ukrást. V tomto modu již nejde pouze o přesnost střelby, ale také o týmovou spolupráci a taktické uvažování. Hrám, nabízejícím pouze mody pro týmy (ne pro jednotlivce), se také říká *tactical shooter*.

Prostředí v hrách se nazývá *lokace* nebo také *mapa*. Každá hra Unreal Tournament může probíhat v jiném prostředí. V UT je takových prostředí značný počet. Můžeme se podívat například na vesmírnou základnu, do skladiště továrny nebo do jedoucího vlaku.

Každá postava má kromě polohy v prostoru několik dalších atributů, které určují její stav, například míra zdraví (nebo také počet životů), síla zbroje, zbraně a náboje, které vlastní. Je-li postava zasažena nějakou zbraní, tak se jí sníží míra zdraví, podle typu zbraně, kterou byla zasažena. Klesne-li míra zdraví na nulu, je postava zněškodněna a po chvilce se znovu objeví na nějakém místě mapy s počátečním stavem zdraví, zbroje a základními zbraněmi.

Unreal Tournament může hrát více hráčů současně, tzv. *multiplayer*. Hráči se mohou do jedné hry UT připojit pomocí místní sítě nebo internetu a hrát proti sobě. Primárně tedy hráči bojují proti dalším lidským hráčům.

UT také myslí na hráče, kteří nemají možnost hrát tuto hru multiplayer. Hru lze spustit s různým počtem počítačem řízených botů, se kterými pak hráč bojuje. Při tvorbě těchto umělých protivníků je kladen důraz na jejich inteligenci. Boti by měli být schopni pohybovat se prostředím, sbírat vhodné předměty a zvažovat situaci, kdy se ještě vyplatí bojovat a kdy utéci. Bot má nahradit lidského oponenta co nejvěrohodněji.

4. 2 Trochu historie

Fenomén FPS her odstartovaly hry Wolfenstein [5] a Doom [6] vydané firmou ID Software. Hra Doom se také stala nechvalně proslulá na pracovištích po celém světě, jako hra odvádějící od práce. Lidé si ji během pracovní doby spouštěli a hráli ji dohromady proti sobě. Příklad je to sice negativní, ale svědčí o nebývalé popularitě hry a zájmu o FPS hry obecně. Postupně různé firmy vydávaly další a další tituly, ve kterých se vylepšovala hlavně jejich grafická stránka. Slabinou těchto prvních 3D her byl pohyb a inteligence protivníků.

Jejich pohyb byl nekoordinovaný a omezoval se jen na přímý směr k hráči, případně jednoduché *zigzag* kličkování, či náhodný pohyb. Inteligencí protivníků se hry nezabývaly jednoduše proto, že veškerý výkon procesoru byl spotřebován na vykreslování 3D světa.

Postupem času se rozšířil trh s grafickými kartami. Grafická karta se stala druhým, úzce specializovaným procesorem stolního počítače. Karta přebírá práci

hlavního procesoru při vykreslování 3D scén, takže programátoři mohli obohatit hry o další aspekty.

Obr. 1. Vývoj grafiky v FPS hrách – Wolfenstein [5] (vlevo nahoře), Doom [6] (vpravo nahoře), HalfLife [2] (vlevo dole), Counter Strike: Condition Zero [7] (vpravo dole).

Prostředí FPS her jsou stále rozlehlejší a vzniká nový typ, který od hráčů vyžaduje nejen rychlé reflexy a přesnou střelbu, ale rovněž taktické uvažování. Průkopníkem her nového typu, který byl pojmenován *tactical shooter games*, je hra Counter-Strike (dále *CS*) [8].

V CS se stáváte buď teroristou nebo členem protiteroristického komanda. Cílem teroristů je například vyhodit do povětří továrnu a cílem komanda je továrnu bránit. CS bylo zprvu možné hrát pouze mezi lidmi stylem *multiplayer*. Vydáním *Software Development Kit (SDK)* bylo umožněno vytvářet boty pro tuto hru a experimentovat s jejich umělou inteligencí.

Jaké požadavky klade nový typ her na počítačem řízené protivníky? Oproti jednoduchým hrám typu death-match již boti nemohou běhat, kudy se jim za chce, a jednat každý sám za sebe. Náhodný pohyb po mapě začíná vypadat hloupě a nespolupracující tým nevěrohodně. Nemluvě o tom, že v žádném případě takový tým botů nemá šanci vyhrát proti spolupracujícímu lidskému týmu, a tedy taková hra se po chvíli stává zcela nudnou v porovnání s hrou mezi lidskými protivníky.

4. 3 Využití FPS her

Samotná hra Counter-Strike vychází ze hry Half-Life, která se spolu s Unreal Tournamentem stala úspěšnou právě proto, že dovoluje komukoli hru upravit nebo dokonce vytvořit na jejím základě hru novou. V dnešní době je již možné si k těmto hrám zdarma stáhnout editory map a upravovat stávající mapy nebo vytvářet úplně nové. K Half-Life poskytuje výrobce zdarma SDK (kód je psán v C++), pomocí něhož lze také vytvářet boty [9].

Naproti tomu hra Unreal Tournament je z velké části naprogramována v *UnrealScript*. UnrealScript je objektově orientovaný jazyk vyvinutý pro

programování stroje *UnrealEngine*, jenž se zaměřuje na vizualizaci 3D scén. Jazyk a stroj je inspirován jazykem Java, resp. Java Virtual Machine. UT je distribuován se všemi zdrojovými kódy napsanými v UnrealScript spolu s kompilátorem tohoto jazyka, takže kdokoli má možnost si UT upravit – tedy vytvářet i vlastní boty.

Kromě Half-Life a UT se pro tvorbu botů používá také prostředí starší hry Quake II. Zdrojové kódy této hry byly uvolněny pod licencí GPL. Kód Quake II je napsán v jazyce C a x86 assembleru.

4. 4 Výhoda UT oproti Half-Life a Quake II

Důvodem pro práci v prostředí UT je již zmíněná architektura. Nejprve byl naprogramován virtuální stroj UnrealEngine, nad kterým pracují skripty v jazyce UnrealScript. Na rozdíl od her Half-Life a Quake II rozšiřování UT nevyžaduje kompilátor jazyka C++, resp. C. Také skripty psané pro UT jsou přenositelné a nezávislé na systému.

Největší výhodu však vidím v tom, že v UT lze vždy pracovat s nejnovější ihned po jejím vydání. UT již existuje ve třech verzích – UT 2000, UT 2003, UT 2004 a pracuje se na UT 2007. Novější verze se zatím vždy prodávaly spolu se zdrojovým kódem v UnrealScript a editorem map UnrealEd. Je tedy možné novou verzi hry okamžitě začít používat pro experimenty. Nové verze přinášejí nejen vylepšení vizuální stránky hry, ale také nové prvky do prostředí hry. V UT 2004 to jsou například dopravní prostředky, které avataři ve hře mohou využívat.

5 Boti v FPS a problémy, které musí řešit

Jaké jsou botovy cíle v FPS hře? Jaké problémy musí řešit? Na tomto místě by bylo dobré připomenout, že stále hovoříme o postavě ve hře. Hra jako taková má vlastně jediný cíl – pobavit svého hráče a boti by měli být tomuto požadavku přizpůsobeni.

Nechci se zabývat tím, co je to vlastně zábava, protože má pro to každý svá vlastní měřítka. Faktem však je, že žádného hráče nebaví, jestliže lze jednoduše zvítězit, když jsou boti hloupí, nebo naopak nelze hru vůbec dohrát, když jsou boti nepřirozeně přesní a rychlí. Špatný dojem ze hry vzniká také tehdy, když se boti nedokáží pohybovat prostředím nebo když se pohybují chaoticky, ignorují hráče apod. Každý hráč vám potvrdí, že hra proti člověku je daleko zábavnější.

Botovy cíle v prostředí hry mohou být různé – zajmout a chránit rukojmí, vyhodit do povětří továrnu, bránit nějaký objekt aj. Při tom musí řešit tyto problémy:

- používání zbraní
- pohyb prostředím
- „být ve správném čase na správném místě“
- týmová spolupráce
- výběr další akce (mechanismus výběru akcí)

Problémy kladou různé nároky na botův model prostředí. U každého problému jsou zmíněny požadavky na model prostředí, které by měl model splňovat, aby bylo vůbec možné tyto problémy úspěšně řešit.

5. 1 Používání zbraní

Zbraně jsou součástí prostředí a mají definovanou sílu a způsob, jakým střílí. Vlastní střílení za bota řeší hra sama. Bot se jen rozhodne střílet na nějaké místo a hra už sama zjistí, zda něco zasáhl či nikoliv. Zde je také hodně prostoru pro odstupňování síly bota. Slabší boti budou zkrátka více nepřesní na větší vzdálenosti, zatímco silní boti budou schopni zasáhnout cíl i z velké dálky.

Výjimkou je střelba z raketometů nebo házení granátů, tedy používání zbraní, které mají buď plošný efekt nebo jejichž střely jsou pomalé. Při střílení z těchto zbraní je nutné odhadnout směr a rychlost pohybu postavy, neboť bot musí vystřelit do míst, kam objekt nebo postava dorazí před tím, než střela místo zasáhne.

Zvlášť zajímavé je pak házení granátů, které mají schopnost se několikrát odrazit od zdí před tím, než vybuchnou. Dobří hráči již mají natrénovanou techniku hodu granátu a dokáží jej hodit například za roh chodby a získat taktickou výhodu tím, že poškodí nepřítele dřív, než se s ním střetnou.

Chce-li bot odhadnout budoucí pozici postavy, musí vědět, kde všude se postava může pohybovat, a ve kterých místech se nachází zdi. Bot například může vidět postavu skrz úzké okno, ale parametry střely nemusí dovolovat skrz toto okno střílet. Tyto informace by měly být zahrnuty v modelu prostředí.

5. 2 Pohyb prostředím

Pohyb bota prostředím představuje složitý problém. Chce-li se bot dostat z bodu *A* do bodu *B*, měl by tak učinit po nějaké ideální trase. Neměl by při pohybu vrážet do zdí nebo se někde zaseknout. Měl by se také vyhýbat ostatním postavám, popřípadě nebránit jim v jejich pohybu, pokud jsou to týmoví spoluhráči.

Pod pojmem ideální trasa se pak nemusí skrývat pouze trasa nejkratší, ale můžeme si klást i jiná kritéria. Pokud se bot chce dostat někam nepozorovaně, pak by určitě neměl běhat po volném prostranství a prozrazovat svou polohu. Zvolená trasa by měla procházet místy, kde je možno se za něco schovat v případě střetu s nepřítelem. Pokud jste hráli nějakou FPS hru, tak určitě víte, co mám na mysli. Pokud ne, tak se asi ptáte, co to znamená místo, kam se mohu schovat? Když člověk vidí nějakou scénu ze hry, tak je mu jasné, že ta bedna vpravo před ním mu poslouží jako úkryt (viz obr. 2).

Obr. 2. Scéna ze hry Counter-Strike [8]

Tento závěr můžete učinit, protože víte, že bedna je neprůhledná a neprůstřelná a že nepřítel je v prostoru za bednou (ne před ní), a proto vás bedna zakryje. O těchto skutečnostech skoro ani nemusíte přemýšlet, nabízejí se samy. Je to logické, neboť prostředí her se snaží kopírovat skutečný svět, a v tom se pohybujeme dnes a denně.

Jak ale docílit, aby takový závěr učinil i bot? Aby bot byl schopen získat odpověď na tuto otázku, potřebuje nějakým způsobem vnímat prostředí (angl. *spatial awerness*). Vnímání prostředí by mělo zahrnovat také informaci o viditelnosti jednotlivých míst.

Pohyb prostředím je úzce spjatý s reprezentací prostředí. Pokud botovi dáme k dispozici jen seznam polygonů, ze kterých se svět skládá, tak budeme při pohybu bota odkázáni na náročný *ray-casting* [1)] a vyhodnocování toho, zda je před botem nějaká překážka či nikoliv. Je sice pravda, že díky ray-casting jsme schopni získat všechny informace, které potřebujeme, ale bylo by to časově velice náročné.

5. 3 „Být ve správném čase na správném místě“

Odpověď na tento problém je závislá na tom, co chce bot dělat, kde se nachází jeho spoluhráči a kde jeho nepřátelé. Uvedu příklad ze hry zkušených hráčů. Jestliže člověk hraje nějakou FPS hru delší dobu, tak získá dobrý přehled o mapách, po kterých se často pohybuje. Zejména pozná, jak lze dojít z jednoho místa na druhé v nejkratším čase, a zapamatuje si různé trasy mezi jednotlivými místy. Na základě těchto informací pak dokáže vhodně volit trasu svého pohybu tak, aby se svým nepřátelům buď vyhnul, anebo je odněkud překvapil. Pod tímto problémem se také skrývá výzva přizpůsobivosti botů. Každý hráč „střílečekˮ má nějaký svůj styl a oblíbená místa, kde číhat na nepřátele a podobně. Pokud je mapa složitější, tak můžeme u botů sledovat, že některými částmi mapy probíhají vždy stejně, dívají se stejným směrem nebo přeskakují překážky v jednom místě. Pro hráče pak není problém takového bota zneškodnit, neboť je vždy odněkud nechráněn. To je třeba typický problém UT botů.

Model prostředí by tedy měl obsahovat pro boty nějaké nápovědy, například „toto je dobré místo pro úkrytˮ, „odtud je výhled na většinu mapy, a tedy dobré místo pro ostřelovačeˮ. Také různé trasy mohou být předem spočítány. Například v modu hry capture the flag existují dvě vlajky, mezi kterými se hráči nejčastěji pohybují. Trasy mezi těmito vlajkami můžeme dopředu nalézt a uložit, neboť je budou boti často potřebovat.

5. 4 Týmová spolupráce

Dalším z problémů je týmová spolupráce botů a také spolupráce botů s lidskými postavami. To je však téma z jiné oblasti, než kterou se zabývá tato práce. V práci řeším reprezentaci prostředí pro boty a ne jejich spolupráci, i když samozřejmě spolupráce botů se musí opírat o prostředí, neboť se musí umět dohodnout na společném postupu. Problémem spolupráce botů se zabýval kolega Ondřej Burket [10].

5. 5 Mechanismus výběru akcí

Mechanismus výběru akcí (dále *mechanismus*) určuje botovi, co provede. Mechanismus vybírá ze souboru akcí, které má bot v daném prostředí k dispozici. Soubor akcí je definován stavem bota a stavem prostředí a v závislosti na těchto stavech se mění.

Mechanismus vybírá akce vždy podle toho, jaký cíl bot v prostředí sleduje. Programuje se buď jako plánovač hledající optimální řešení, anebo jako reaktivní architektura, která reaguje na podněty z prostředí. Pro boty se volí právě reaktivní architektury, neboť prostředí, ve kterém se pohybují je dynamické a nepředvídatelné.

Vztah k modelu prostředí je takový, že veškeré akce, které bot může dělat by se měly v tomto modelu projevit, jinak bot nebude vědět, zda-li jeho akce uspěla, či

1) vrhání paprsku, který vyhází ze zadaného bodu nějakým směrem, a vyhodnocení, kterými objekty prochází a v jakém pořadí

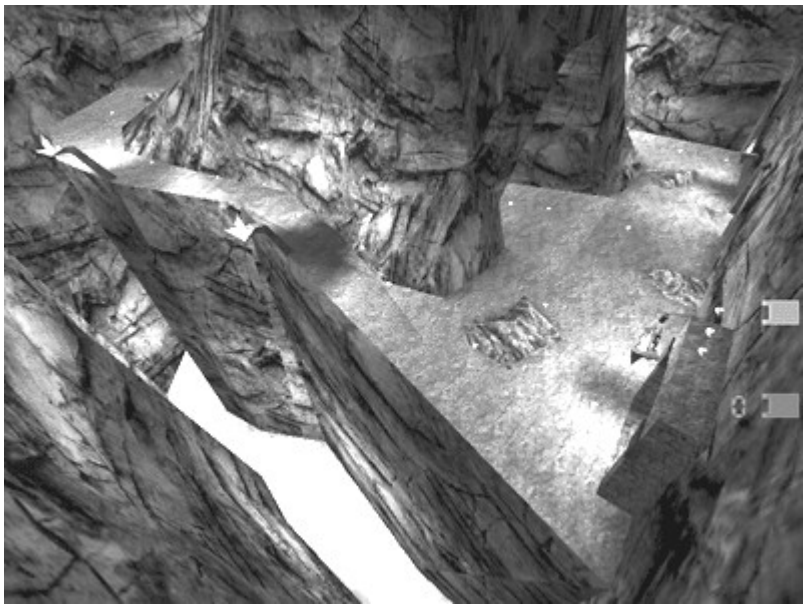
nikoli. Následně by mechanismus měl problém s tím, jestli zvolenou akci zopakovat, nebo vykonat logicky následující akci.

6 Navigační body

Jedním z cílů práce je bližší popis modelu navigačních bodů a popis, jak s nimi bot pracuje. V této kapitole ukáži principy tohoto modelu a následně popíši, jakým způsobem je model implementován a používán v Unreal Tournament.

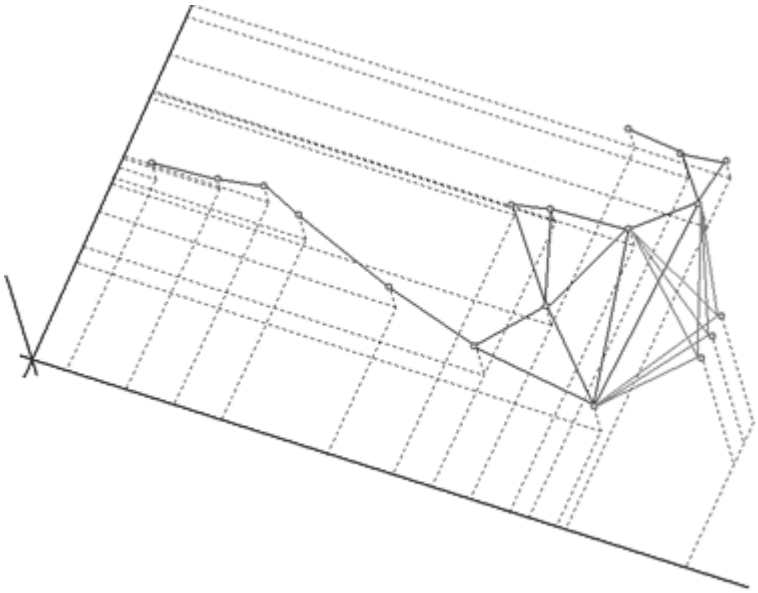
Model navigačních bodů zjednodušuje prostředí hry tak, aby se v něm boti dokázali rychle pohybovat. V místech, kterými se v mapě často prochází, jsou umístěny navigační body, které jsou před hráči skryty. Každý navigační bod se nachází těsně nad zemí (podlahou) prostředí a je popsán svými souřadnicemi. Ke každému navigačnímu bodu je dán seznam jeho sousedních navigačních bodů.

Navigační bod *A* je sousedem navigačního bodu *B* právě tehdy, když se lze přímým pohybem po zemi dostat z bodu *A* do bodu *B*. Vzniká nám tak graf z navigačních bodů umístěných v prostoru. Tento graf je zpravidla orientovaný, protože pádem z vyššího místa se lze dostat k navigačnímu bodu, který je níže, ale opačně to není možné. Jednotlivé hrany jsou ohodnoceny vzdálenostmi. Tato vzdálenost je většinou rovna vzdálenosti v prostoru mezi navigačními body tvořícími danou hranu. Můžeme však do ní také připočítat náročnost pohybu po daném terénu. Z výsledného grafu se také mohou vypustit hrany, které nejsou úplně potřebné. To znamená, existuje-li hrana *AB* a blízko cesty po zemi mezi body *A* a *B* se nachází bod *C* a existují hrany *AC*, *CB*, pak hrana *AB* je vypuštěna.



Obr. 3. Obrázek z UT [1] mapy CTF-EternalCave.

Bílé tečky na obrázku 3 těsně nad zemí znázorňují navigační body.



Obr. 4. Graf navigačních bodů odpovídající obrázku 3.

Takto vnímají prostředí boti. Všimněme si navigačních bodů úplně vpravo, které jsou na římse. Pouze z nich je možno postupovat směrem k ostatním navigačním bodům, ale ne opačně.

6. 1 Použití navigačních bodů pro pohyb prostředím, algoritmus A*

Bot se na prostředí dívá jako na graf bodů v prostoru. Jakmile se objeví na mapě nalezne v ní nejbližší navigační bod, na který se přesune. Obvyklou praxí je, že místa, kde se boti objevují, nejsou volena náhodně, ale jsou volena buď blízko navigačních bodů nebo jsou přímo součástí grafu. Chce-li se bot dostat z navigačního bodu *A*, kde stojí, na nějaký jiný navigační bod *B*, pak potřebuje znát cestu v tomto grafu. Nejčastěji potřebuje znát cestu nejkratší, a pro její nalezení použije jeden z algoritmů kombinatoriky. Jakmile již danou cestu má, začne se podle ní pohybovat.

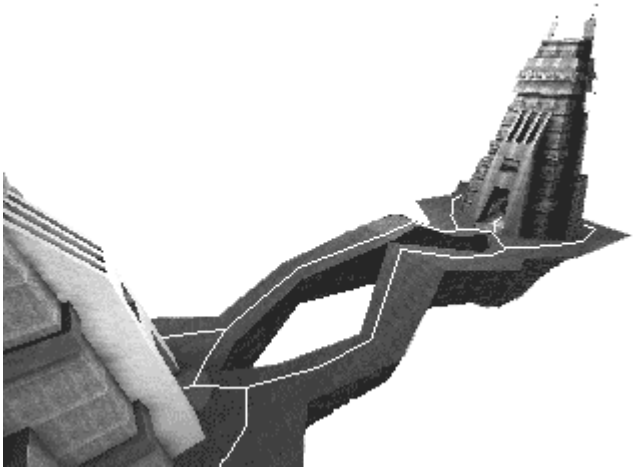
Pro hledání nejkratší cesty v grafu se v FPS hrách používá nejčastěji algoritmus A* *(A-Star)*. Jedná se o rozšíření Dijkstrova algoritmu o heuristiku, která odhaduje, jak daleko je od probíraného vrcholu grafu ke koncovému vrcholu hledané cesty. Díky této heuristice můžeme odhadnout délku cesty, která vede od počátečního do koncového vrcholu přes vrchol probíraný algoritmem. Jako první zkoumáme vrcholy, u kterých je tento odhad nejnižší a cesta přes ně povede pravděpodobně k cíli. Čím lepší je tato heuristika, tím rychleji algoritmus nalezne cestu v grafu. Aby algoritmus poskytl vždy nejkratší cestu grafem, musí být heuristika korektní. To znamená, že nesmí stanovit vzdálenost k cíli větší než ve skutečnosti je a odhadnutá vzdálenost musí být nezáporná. Více o algoritmu A* se píše v [11].

6. 2 Problémy navigačních bodů

Navigační body byly vytvořeny pro prostředí, která jsou spíše uzavřenější, skládající se z menších místností a chodeb, kde není velká možnost pro *manévrování*, tj. kličkování před nepřítelem za účelem ztížit mu míření. V takových mapách jsou jasné trasy, jak projít mezi místnostmi, a nemá smysl se příliš vzdalovat od průmětu

hrany mezi dvěma navigačními body na podlahu mapy. Jakmile jsou navigační body použity v rozlehlejších chodbách, má bot problém s tím, že neví, jak daleko se může odchýlit od trasy, která vede mezi dvěma sousedními navigačními boty.

Další problémem navigačních bodů spočívá v tom, že nedokáží dobře zachytit rozlehlá prostranství. Pokud do otevřeného prostranství vložíme málo navigačních bodů, pak boti budou chodit po neměnných trasách a po chvilce začnou působit strnule.



Obr. 5. Pohled shora na UT mapu CTF-Face.

Na obrázku 5 jsou bílou čarou vyznačeny hrany mezi navigačními body. Boti přebíhají mezi pevnostmi po jedné ze dvou tras a málo se od ní odchylují. Je pak velice jednoduché je zneškodnit pomocí ostřelovací pušky z vrcholku jedné či druhé pevnosti.

Problém velkých prostranství lze řešit tak, že do nich umístíme velké množství navigačních bodů. Ovšem to pro algoritmus hledající nejkratší cestu v grafu mezi navigačními body znamená zpomalení, které může být nepřijatelné, neboť by se bot musel zastavovat a čekat na výsledek výpočtu algoritmu.

S navigačními body mohou být spojeny také různé nápovědy pro boty, např. „zde je dobré se schovat“, „odtud máš rozhled na velkou část mapy“. Podrobněji se o nich zmíním v kapitole o prostředí Unreal Tournamentu, i když se samozřejmě jedná o obecnou techniku, kterou využívají i jiné hry.

6. 3 Matice viditelnosti – vylepšení navigačních bodů

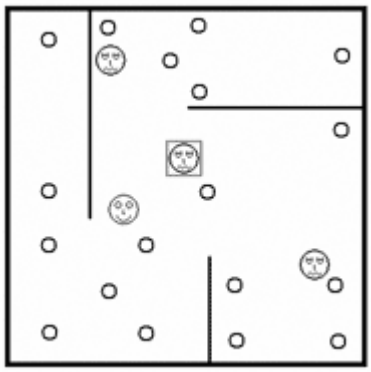
Navigační body samy o sobě nenesou žádnou informaci o viditelnosti. Dokonce ani nemusí platit, že z navigačního bodu jsou viditelné všechny jeho sousední body. Je možné, že mezi navigačním bodem *A* a jeho sousedem *B* bude kopec, který je nutné přelézt.

Matice viditelnosti [12] je technika, která botovi umožňuje lépe volit místo, odkud útočit na své nepřátele, nebo jak se před nimi schovávat.

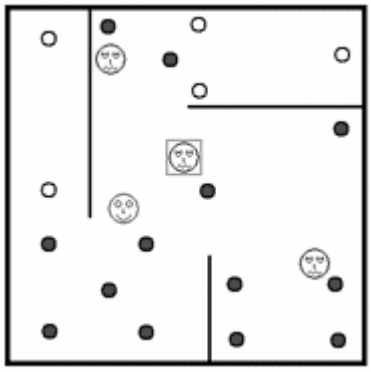
Odpověď na otázku, zda je místo *A* viditelné z místa *B*, získáme pomocí ray-castingu. Jestliže paprsek vržený z místa *A* do místa *B* nenarazí na žádnou překážku, jsou místa navzájem viditelná. Mapu tedy lze dopředu předzpracovat a udělat ray-casting mezi všemi navigačními body a výsledek si zapsat do bitové matice. Označíme-li navigační body N_1 až N_k, pak pro navigační body N_i a N_j máme výsledek ray-castingu uložen v pozici *(i, j)* v matici viditelnosti. Pro každý navigační bod N_i tak máme bitový vektor viditelnosti všech navigačních bodů – jedná se o i-tý řádek matice.

S bitovými vektory můžeme provádět logické operace *and*, *or*, *xor*, *not* a získávat zajímavé výsledky. Ukáži na příkladu, jak lze matici viditelnosti využít pro nalezení místa dobrého k útoku na zvoleného nepřítele.

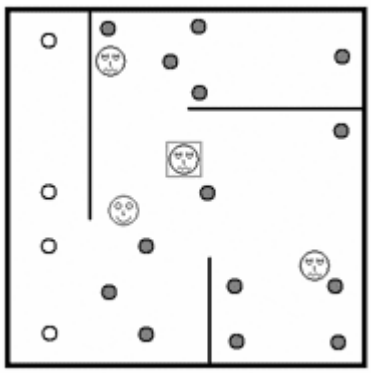
Na obrázku máme pohled shora na místnost se třemi přepážkami, třemi zamračenými nepřiteli a usměvavým botem. Prázdná kolečka znázorňují navigační body. Bot si zvolil, že chce zaútočit na nepřítele, který je mu nejblíže (je orámovaný). Zůstane-li stát na svém místě, bude to nerovný boj 3 proti 1. Bot by se tedy měl přemístit někam, odkud bude mít výhled na zvoleného nepřítele, ale ostatní nepřátele ho neuvidí.



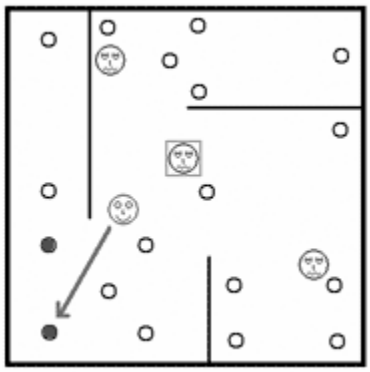
Na tomto obrázku jsou tmavě vyznačeny navigační body, ze kterých je možné vidět zvoleného nepřítele.



Zde jsou vyznačené ty navigační body, které jsou ve výhledu ostatních dvou nepřátel.



Ze seznamu navigačních bodů, ze kterých je zvolený nepřítel viditelný, jsme odebrali ty navigační body, na které mají výhled ostatní dva nepřátele. Výsledkem jsou dva tmavé navigační body, ke kterým je vhodné se přesunout.



Výsledek předcházejícího příkladu můžeme získat pomocí logických operací s bitovými vektory. Výsledek binární logické operace and, or a xor nad dvěma

bitovými vektory *a*, *b* (stejně dlouhými) definujeme jako bitový vektor *c*, kdy na i-té pozici vektoru *c* je výsledek logické operace and resp. or resp. xor *a[i]* a *b[i]*. Výsledek unární logické operace not nad bitovým vektorem definujeme jako bitový vektor *c*, na jehož i-té pozici je výsledek logické operace not *a[i]*.

Máme-li bota *B* a k němu bitový vektor *V* nejbližšího navigačního bodu *a*, máme-li seznam nepřátel E_1 až E_a a ke každému nepříteli E_i máme bitový vektor V_i nejbližšího navigačního bodu k nepříteli E_i, pak můžeme spočítat bezpečné navigační body a body nejvýhodnější pro útok.

Bezpečné navigační body (žádný z nepřátel je nevidí):

$S = \neg \bigvee_{i=1}^{k} V_i$ … je-li *S[i]* rovno 1 (pravda), pak N_i je bezpečný navigační bod

Chce-li bot zaútočit na nepřítele E_i, pak je pro bota nejvýhodnější přesunout se na jeden z navigačních bodů:

$S = V_i \wedge \neg \bigvee_{j=1..k,\, j \neq i} V_j$ … je-li *S[l]* rovno 1 (pravda), pak N_l představuje vhodný navigační bod, ze kterého lze zaútočit na nepřítele E_i.

Další využití matice viditelnosti jsou popsány v [11].

6. 4 Navigační body a Unreal Tournament

Unreal Tournament jako FPS hra již byla představena v úvodu práce. V této kapitole popíši, jak fungují boti UT, a dále pak speciální funkce navigačních bodů v UT spolu s informacemi, které botům poskytují.

Boti v UT. Boti v UT jsou naprogramováni v UnrealScriptu a fungují na principu hierarchických konečných automatů. Pro bota jsou definovány různé stavy, které se od sebe navzájem liší a aktivní z nich může být pouze jeden. Každý stav pak může mít další podstavy. UnrealEngine zasílá v průběhu hry botovi zprávy, co se kolem něho děje. Pro každou zprávu je definována metoda, která je po doručení zprávy spuštěna. Každý stav bota může definovat jinou metodu, která je při příchodu nějaké zprávy spuštěna. Bot tedy funguje na principu systému řízeného událostmi a jednotlivé události vyvolávají reakci podle toho, v jakém stavu se bot nachází.

Boti v UT jsou naprogramováni tak, aby bylo jednoduché odstupňovat jejich sílu. Při tvorbě nové hry s boty můžeme zvolit jejich sílu, podle které je pak bot rychlý a přesný. Silnější boti se také snaží různě manévrovat při střetu s nepřítelem a snaží se mu ztížit jeho střelbu.

Boti využívají navigační body pro pohyb v mapě. Body musí být do mapy umístěny manuálně, což je dost velká nevýhoda pro tvůrce map. Pokud se UT boti spustí na mapě bez navigačních bodů, nejsou schopni se po ní rozumně pohybovat. Většinou popocházejí kolem místa, kde se na mapě objevili.

Navigační body v UT. V UT jsou různé typy navigační bodů, které mohou obsahovat různé informace pro boty. Navigační bod je v UnrealScriptu vyjádřen třídou NavigationPoint a má množství potomků. Každý potomek této třídy má pro bota jiný význam.

Potomci třídy NavigationPoint jsou:

AlternatePath. AlternatePath se používá zejména v mapách pro mod capture the flag. Označují začátky přístupových cest k vlajkám. Boti je využívají k výběru trasy k vlajce na opačném konci cesty.

AmbushPoint. AmbushPoint představuje místo, kde lze v úkrytu číhat na nepřítele. Součástí této třídy je také booleovská vlastnost bSniping, která udává, jestli je z místa rozhled na nějakou větší část mapy a je-li místo vhodné pro ostřelovače.

DefensePoint. DefensePoint se používá hlavně při modu capture the flag. Označuje místo, ze kterého je vhodné hlídat vlajku.

InventorySpot. Každý předmět v UT je instancí třídy InventorySpot. InventorySpot obsahuje informace o předmětu.

JumpSpot. JumpSpot je místem, ze kterého lze někam přeskočit nebo vyskočit.

LiftCenter, LiftExit. LiftCenter je navigační bod, který je umístěn uprostřed výtahu. LiftExit pak určuje místa výstupu z výtahu.

PatrolPoint. PatrolPointy označují místa, která je třeba kontrolovat. Používají se například v modu capture the flag. Součástí tohoto typu navigačního bodu může být také pevný skript, který botovi sdělí trasu, po které se má pohybovat, aby tak hlídal určité místo (například vlajku).

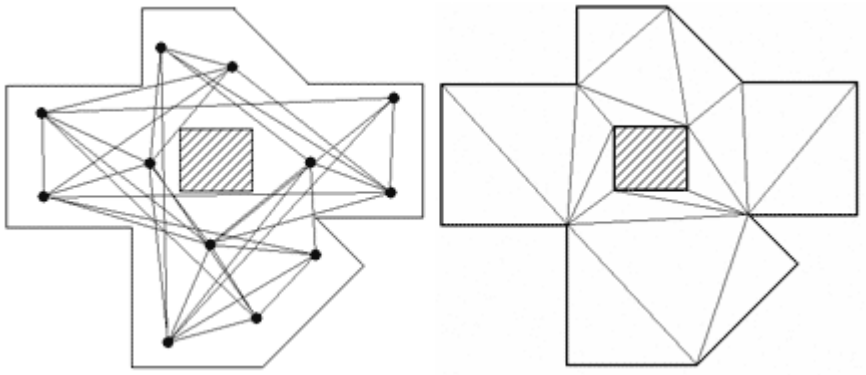
Navigační body jako DefensePoint nebo AmbushPoint zlepšují výkon UT botů. Pokud jsou tyto speciální navigační body správně umístěny, tak boti dokáží efektivně bránit svou vlajku nebo přepadat nepřátele. Všechny tyto body však musí být umístěny ručně tvůrcem mapy. Také hráči si po nějaké době zvyknou, kde boti stojí nebo číhají a dokáží se jim přizpůsobit.

Řešení vnímání prostoru. Jak bylo uvedeno v kapitole o navigačních bodech, hrany mezi sousedními navigačními body nenesou žádnou informaci o okolí této hrany. Boti neví, jak daleko se mohou od své trasy odchýlit a zda jim při nevhodném úkroku nehrozí například pád do lávy nebo náraz do zdi. Toto UT řeší zasíláním zpráv o nadcházející kolizi těsně před tím, než se tak stane. Bot má tak šanci reagovat změnou směru pohybu. Blíží-li se bot například k propasti, tak o tom dostane zprávu spolu s vektorem, který míří směrem od propasti, takže bot může změnit vektor pohybu a vyhnout se pádu. Bot tedy spoléhá na samotné prostředí, které ho má upozorňovat na uvedená nebezpečí.

7 Navigační mřížka – NavMesh

Novějším modelem prostředí pro boty je navigační mřížka – *navigation mesh (NavMesh)*. V této kapitole popíši model podrobněji a zvlášť se budu zabývat výhodami a nevýhodami navigační mřížky oproti staršímu modelu navigačních bodů.

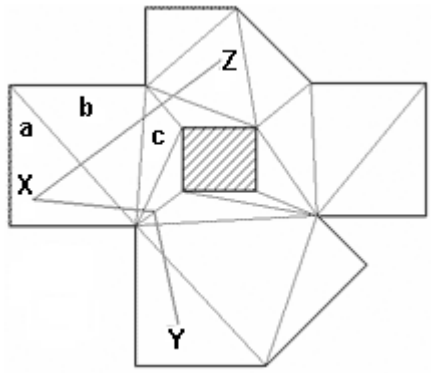
Navigační mřížku ve své práci popisuje G. Snook [4]. Snook vychází z toho, že pro potřeby vizualizace je mapa reprezentována jako seznam trojúhelníků, které se mají vykreslit. Snook navrhuje použití těch trojúhelníků, které tvoří podlahu mapy, jako podklad pro pohyb botů po mapě. Navigační mřížku tak tvoří všechny trojúhelníky, na kterých postavy mohou stát.



Obr. 6. Srovnání reprezentace místnosti (pohled shora)
pomocí navigačních bodů (vlevo) a NavMesh (vpravo)

7. 1 Použití NavMesh pro pohyb prostředím

Výhodou NavMesh je, že bot se nikdy neztratí. Pokud stojí na podlaze mapy, tak vždy stojí na nějakém trojúhelníku, který byl zahrnut do seznamu trojúhelníků tvořících NavMesh. Pro hledání cest se i zde používá algoritmus A* podobně jako u navigačních bodů, jen se obtížněji tvoří ohodnocení grafu, nad kterým A* pracuje. Na obrázku 7 vidíme, že bychom mohli považovat jednotlivé trojúhelníky za vrcholy grafu a mezi vrcholy vytvořit hranu tehdy, když příslušné trojúhelníky sdílí nějakou hranu. Pak ovšem máme problém s ohodnocením těchto hran v grafu, jak je patrno z obrázku 7. Ohodnocení hrany mezi vrcholy grafu je závislé na pohybu bota po daných trojúhelnících.

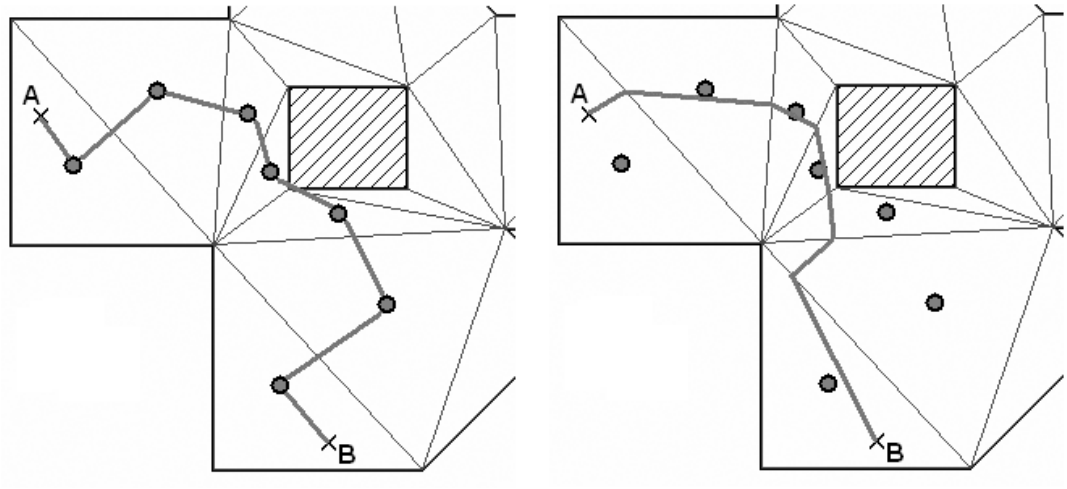


Obr. 7. NavMesh se znázorněním dvou cest *XY* a *XZ*.

Na obrázku 7 je bot v bodě *X* a jednou plánuje cestu do bodu *Y* podruhé do bodu *Z*. Obě cesty na začátku procházejí trojúhelníky *A*, *B*, *C*, ale jejich délky se liší. Počítat s rozdílným ohodnocením cest, by znamenalo při probírání nového trojúhelníku vždy přepočítat nějakou část trasy tak, aby odpovídala ideální trase z počátečního trojúhelníku do probíraného trojúhelníku. To by bylo však časově náročné, a tak stanovíme vzdálenost dvou trojúhelníků jako vzdálenost jejich těžišť a upravíme až výslednou trasu. Kdybychom výslednou trasu vedoucí přes těžiště trojúhelníku neupravovali, tak by se pohyb příliš nelišil od pohybu mezi navigačními body, protože těžiště bychom chápali rovněž jako navigační bod.

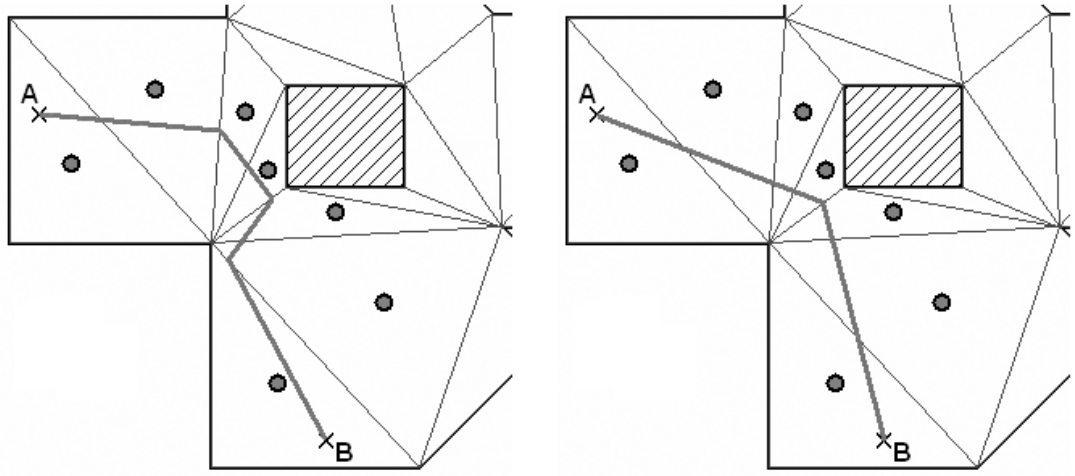
Výslednou trasu můžeme například upravit tak, že se na pohyb po trojúhelnících budeme dívat jako na pohyb od hrany k hraně. Vezmeme vždy pozici bota

v trojúhelníku a uděláme kolmý průmět bodu, na kterém stojí, na přímku, která prochází hranou dalšího trojúhelníku na nalezené cestě. Pokud tento bod není součástí hrany, tak vezmeme bod hrany, který leží nejblíže nalezenému bodu (viz obr. 8).



Obr. 8. Hledání trasy pro pohyb bota

Trasu můžeme dále upravovat například tak, že vždy vezmeme tři trojúhelníky, které jsou za sebou ve výsledné cestě, a podíváme se, zda lze jít přímo z prvního trojúhelníku do třetího, viz obr. 9.



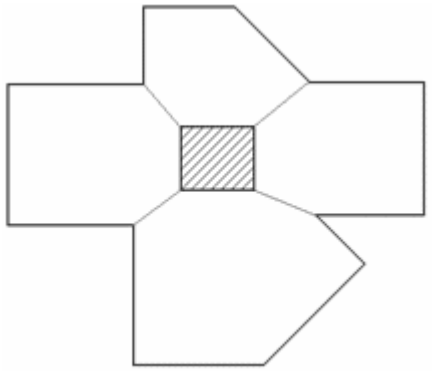
Obr. 9. Nalezenou trasu lze upravovat.

Navíc tyto úpravy nemusíme provádět najednou. Po nalezení cesty mezi těžišti trojúhelníků se bot může vydat směrem k prvnímu tojúhelníku cesty a během jeho pohybu můžeme další trasu vylepšovat. Bot se nemusí nikde zastavovat a čekat na výpočet své trasy.

Výsledný pohyb bota je tak závislý na jeho poloze v trojúhelníku a nesleduje již pevně stanovenou trasu, jako je tomu při používání navigačních bodů.

7. 2 Pohyb v trojúhelníku a vylepšení NavMesh

Při pohybu v trojúhelníku využíváme faktu, že každý trojúhelník je konvexní. Tedy z kteréhokoli bodu trojúhelníku se můžeme přímým pohybem dostat do každého jiného bodu trojúhelníku. Při pohledu na obrázek 9 zjistíme, že některé trojúhelníky by se daly spojit a výsledkem by byl stále konvexní polygon. Snížíme tím počet vrcholů grafu a urychlíme výpočet algoritmu A*. Tato alternativa je blíže popsána v [13].



Obr. 10. Ideální NavMesh složený pouze ze 4 konvexních polygonů.

7. 3 Srovnání s navigačními body

Navigační mřížka má oproti navigačním bodům několik výhod, ovšem také i nevýhod.

Výhody navigační mřížky:

- Reprezentuje celou podlahu mapy.
  Podle definice by měla navigační mřížka obsahovat všechny trojúhelníky, na kterých bot může stát, a tedy neexistuje zákoutí, kam se bot nemůže dostat.
- Je možné ji generovat automaticky.
  Navigační body se musí umísťovat a upravovat manuálně. Navigační mřížku lze vygenerovat z polygonů tvořících mapu.
- Pohyb nesleduje pevně stanovené trasy.
  Boti nejsou nuceni běhat po pevně stanovených trasách, jako je tomu u navigačních bodů. Pohyb vypadá přirozeněji.
- Bot ví, zda při úkroku do strany nenarazí nebo nespadne.
  Při střetu s nepřítelem může bot začít manévrovat a nemusí se bát, že někam spadne nebo narazí. Z navigační mřížky lze zjistit, jestli bot daný úkrok může udělat nebo nikoliv.

Nevýhody navigační mřížky:

- Problémy se statickými objekty, které nejsou součástí polygonů mapy.
  V editorech map pro FPS hry občas můžeme do mapy umístit objekty, které ale nejsou zahrnuty v polygonech mapy. Prostor pod takovým objektem ovšem považuje NavMesh jako průchozí, což není pravda.
- Problémy s viditelností.
  Zatímco s navigačními body můžeme asociovat jejich viditelnost na ostatní navigační body, u NavMesh to udělat nemůžeme. NavMesh se skládá z trojúhelníků definujících plochu a ne bod. Pokud bychom chtěli stanovit viditelnost mezi dvěma trojúhelníky, tak se můžeme dostat do problému, kdy z jednoho trojúhelníku je viditelná pouze určitá část trojúhelníku druhého.

- S navigačními body mohou být spojeny další informace.
  Jak je uvedeno v kapitole č. 6. 4 o navigačních bodech v UT, lze s nimi spojit další informace, na příklad, že je to místo pro ostřelovače aj. S navigační mřížkou to tak jednoduše udělat nelze, protože kritérium pro ostřelovače může splňovat pouze část trojúhelníku.
- Práce s navigační mřížkou vyžaduje složitější výpočty.
  Pohyb pomocí navigačních bodů je velice jednoduchý. Jsme-li na pozici nějakého navigačního bodu a chceme jít k jednomu z jeho sousedů, tak nemusíme nic počítat – hře se zašle zpráva o tom, že bot se chce dostat na souřadnice vybraného sousedního bodu. Používáme-li však NavMesh, tak test, jestli se mohu posunout přímým pohybem na nějaké souřadnice znamená:
  1. Lokalizovat trojúhelník, na kterém stojím.
  2. Otestovat, zda-li se zvolené souřadnice nachází stále ve stejném trojúhelníku.
  3. Pokud se zvolené souřadnice nenachází ve stejném trojúhelníku, tak musíme zjistit trojúhelníky, kterými by bot při pohybu měl projít – tedy zjistit, jestli vůbec tyto trojúhelníky existují, a podle toho rozhodnout, zda je přímý pohyb možný či nikoli.

  Bod tři je vlastně ray-casting, který se provádí v trojúhelnících NavMesh. Paprsek vychází z bodu, kde bot stojí, směrem, kterým se chce pohybovat. Tento test je dost náročný na výpočet. V tomto vidím hlavní důvod, proč se navigační mřížka začíná používat ve hrách až v současné době, kdy jsou k dispozici výkonnější procesory.

8 Automatické generování NavMesh pro mapy UT

Tato kapitola prezentuje můj postup, jak automaticky vygenerovat NavMesh pro mapy UT, což je hlavním cílem mé práce.

Jak bylo zmíněno v kapitole 6. 4, boti Unreal Tournament používají navigační body pro pohyb prostředím. Tato závislost na navigačních bodech s sebou nese nevýhody, jako je nedostupnost některých míst mapy a stále stejný způsob pohybu po mapě. Oba tyto problémy řeší použití modelu navigační mřížky, což jsem ukázal v kapitole 7.

Navigační mřížka je vytvářena ze seznamu polygonů, které tvoří celou mapu. Do navigační mřížky jsou vybrány ty polygony, po kterých se mohou boti pohybovat. U každé hrany každého polygonu je uložena informace o tom, zda je hrana sdílena s jiným polygonem, může-li tedy bot přes tuto hranu projít na jiný polygon navigační mřížky. Polygony tvořící mapu musíme rozpoznat a získat jejich popis.

Bohužel UnrealScript nenabízí žádné rozhraní pro získání polygonů, ze kterých se mapa skládá. Tyto informace musíme získat před zahájením hry.

Pro tvorbu map je spolu s hrou dodán editor UnrealEd, ve kterém lze editovat existující mapy UT a také vytvářet mapy nové. Mapy se implicitně ukládají do binárního formátu *UNR (Unreal Rousource)* nebo je lze exportovat do textového souboru T3D.

8. 1 Formát souboru UNR

Formát souboru UNR je binární a nedokumentovaný. Navíc není určen pouze pro ukládání map, ale jedná se o víceúčelový formát, se kterým UT implicitně pracuje. Další informace o něm lze získat na stránkách [14], které vytvořil jeden z vývojářů

Unreal Tournament a ze které je podstatná jedna věta: „Abandon all hope ye who try to parse this file format.“ [2)]

8. 2 Formát souboru T3D

Formát souboru T3D je sice nedokumentovaný zato textový. Způsob jeho interpretování jsem nikde nenašel. Jeho interpretace je netriviální a musel jsem ji experimentálně stanovit. Své výsledky jsem pak umístil na web zabývající se Unreal Tournament [15]. Podrobnosti, jak získat objekty tvořící mapu v UT, uvádím v příloze D.

8. 3 Zpracování T3D souboru

Z popisu T3D souboru v příloze A je důležité, že mapa v Unreal Tournament je reprezentována pomocí konstruktivní geometrie. V T3D souboru jsou uložena různá tělesa, která mají svou přesnou pozici v prostoru a navzájem se mohou protínat nebo dotýkat. U každého takového tělesa je specifikována operace, zda má být toto těleso do prostoru přidáno, či z něj má být vyňato *(subtracted)*. To znamená, že pokud chceme získat polygony mapy, musíme nejprve převést reprezentaci mapy z konstruktivní geometrie do hranové reprezentace, která je vhodná pro tvorbu navigační mřížky [4].

Implementace algoritmu převodu reprezentací tělesa je nejdůležitější částí tvorby NavMesh. Tento algoritmus a jeho aplikace na data z T3D souboru popisuji v kapitole 9.

8. 4 Hranová reprezentace mapy a NavMesh

Hranově reprezentovaný objekt je tvořen seznamy vrcholů tělesa, jeho hran a povrchů. Každá hrana je tvořena dvěma vrcholy a u každé z nich je zapsána informace, které dva povrchy spojuje. Každý povrch je reprezentován polygonem, který je tvořen svými hranami.

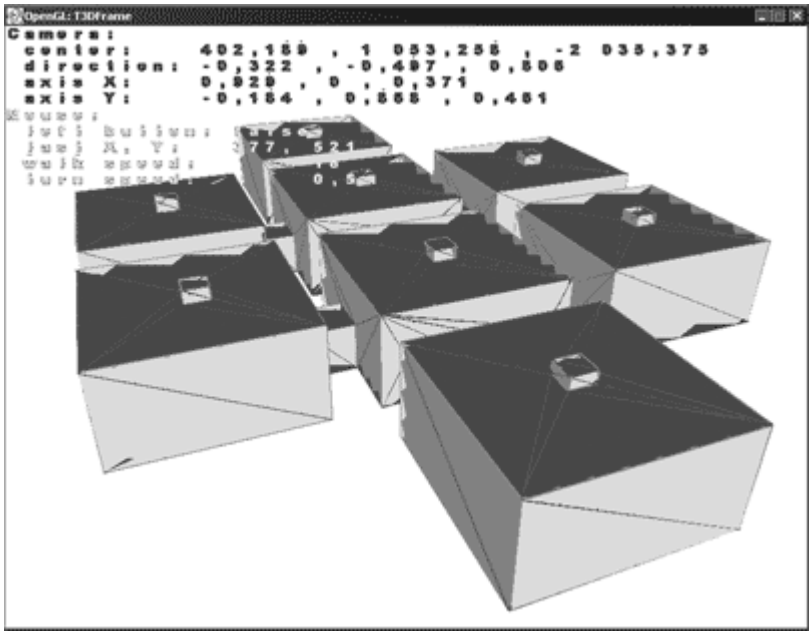
Tato reprezentace je pro tvorbu navigační mřížky ideální, neboť již v sobě nese informaci, jak na sebe jednotlivé polygony navazují prostřednictvím jejich hran.

Před tvorbou NavMesh je třeba zkontrolovat, zda veškeré polygony tvořící mapu, jsou konvexní. Nekonvexní polygony je nutno rozdělit.

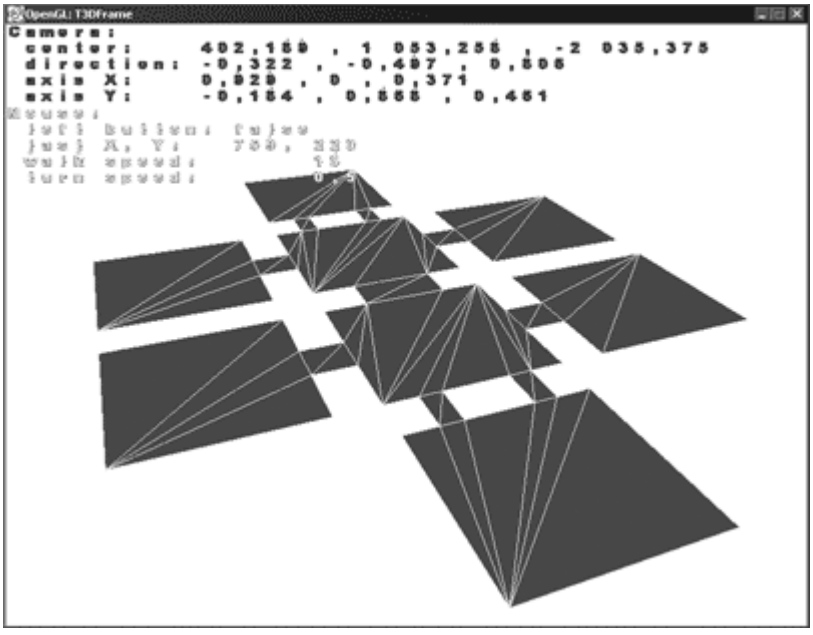
Pak již stačí z hranové reprezentace mapy vybrat ty polygony, na kterých se může bot pohybovat. To jsou takové polygony, jejichž opačný vektor k normále roviny nesvírá s vektorem (0,0,1) více než 50 stupňů. Vektor (0,0,1) je určen stojícím botem a 50 stupňů je zvolená konstanta, která byla experimentálně stanovena. Dále jsou do NavMesh zahrnuty polygony, jejichž rozdíl maxima a minima *z-tové* souřadnice nepřekračuje 50. Opět se jedná o zvolenou konstantu. Daný trojúhelník je vnímán jako schod. Není otestováno, zda na sebe nenavazuje více takovýchto schodů a netvoří celou stěnu.

Takto vybrané polygony spolu s informacemi o hranách, které je spojují, již tvoří NavMesh, jak jej popsal ve své práci Snook [4].

[2)] Marné jsou naděje těch, jenž hledají způsob, jak analyzovat tento formát. (Překlad J. Gemrot)



Obr. 11. UT mapa CTF-1Knight, kterou vytvořil můj kolega Michal Bída



Obr. 12. NavMesh pro mapu CTF-1Knight. NavMesh je tvořen trojúhelníky z důvodu použitého algoritmu na převod mezi reprezentacemi tělesa (viz kapitola 9).

Pro tvorbu navigační mřížky z T3D dat, jsem naprogramoval T3D aplikaci, která umožňuje vytvoření NavMesh a jeho uložení do XML souboru. Aplikace využívá algoritmus na převod mapy z reprezentace pomocí kontruktivní geometrie do hranové reprezentace, který popisuji v následující kapitole. O postupu použití T3D aplikace píši v kapitole 10.

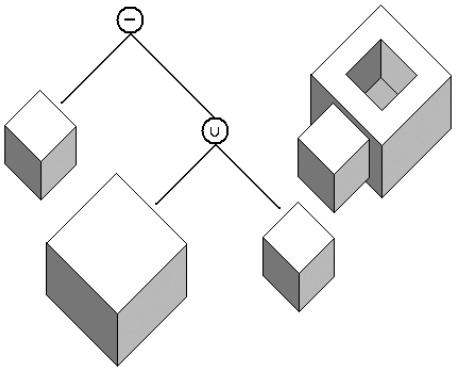
9 CSG Algoritmus

V této kapitole popisuji algoritmus, jehož pomocí lze převést těleso z reprezentace pomocí konstruktivní geometrie (dále jen *CSG reprezentace*) do hranové reprezentace (viz kapitola 8.4). Jde o známý problém z počítačové grafiky, jehož řešení uvedl Hubbard [16], který vylepšuje algoritmus Leidlawa [17].

Nejprve uvedu, co je vstupem algoritmu, co se o vstupu předpokládá a jaký je výstup algoritmu (9.1). Následně popíši Leidlawův algoritmus vylepšený Hubbardem (9.2). Dále stanovím horní odhad složitosti algoritmu, který v pracích Leidlawa a Hubbarda schází (9.3). Poté popíši aplikaci algoritmu na data z T3D souboru (9.4) a několik změn algoritmu, které jsem provedl (9.5). V závěru kapitoly uvedu problémy této aplikace, a jak se snažím tyto problémy řešit (9.6).

9. 1 Vstup a výstup algoritmu

Vstupem algoritmu je těleso reprezentováno pomocí konstruktivní geometrie. CSG reprezentace zapisuje těleso jako binární strom, jehož listy jsou tělesa a uzly jsou množinové operace, které se mají provést s levým a pravým podstromem uzlu. Množinová operace je buď sjednocení, rozdíl nebo průnik dvou těles (dále jako slučování těles). Výsledkem takovéto množinové operace je vždy nové těleso. Algoritmus pak rekurzivně vyhodnocuje jednotlivé uzly stromu tak, že tělesa levého a pravého podstromu uzlu sloučí dohromady podle množinové operace, která se váže s daným uzlem. Strom je tedy vyhodnocován postfixově – nejprve se stanoví výsledek levého a pravého podstromu a až pak se tyto výsledky slučují dohromady. Výsledkem vyhodnocení kořenu stromu je těleso v hranové reprezentaci, které bylo daným stromem reprezentováno konstruktivní geometrií.



Obr. 13. Ukázka reprezentace objektu pomocí konstruktivní geometrie

Algoritmus Leidlawa předpokládá, že tělesa v listech stromu jsou již hranově reprezentována a jsou to *2-manifoldy* (taková tělesa, ve kterých každá hrana tělesa spojuje právě 2 povrchy tělesa). Výsledkem sloučení těles je pak opět 2-manifold v hranové reprezentaci.

Hubbard pak na tělesa stanovuje ještě další požadavek, a to že jsou složena výhradně z trojúhelníků, což umožňuje rychlejší práci algoritmu (viz [16]). U každého trojúhelníku také musíme znát normálu jeho roviny, která ukazuje směrem ven z tělesa.

9. 2 Datová reprezentace těles

Každé z těles, které vstupuje do algoritmu a které je v průběhu algoritmu vytvářeno, je hranově reprezentováno a složeno výhradně z trojúhelníků.

Těleso je reprezentováno pomocí tří seznamů: vrcholy *(vertices)*, hrany *(edges)*, trojúhelníky *(triangles)*.

Každý vrchol je reprezentován svými souřadnicemi v prostoru (*x, y, z*).

Hrana je tvořena svými krajními body a je u ní poznamenáno, které dva trojúhelníky spojuje. U každé hrany je uložena informace, zda spojuje dva trojúhelníky ležící ve stejné rovině (*koplanární trojúhelníky*). Takové hraně se říká *vnější*, a pokud spojuje dva trojúhelníky, které neleží v jedné rovině (*nekoplanární trojúhelníky*) označuje se jako *vnitřní*.

Trojúhelník je tvořen svými třemi hranami. U každého trojúhelníku jsou také zaznamenány maximální a minimalní hodnoty jednotlivých souřadnic krajních bodů jeho hran. Pro každý trojúhelník je také známa normála roviny, ve které leží a která ukazuje směrem ven z tělesa.

9. 3 Popis algoritmu

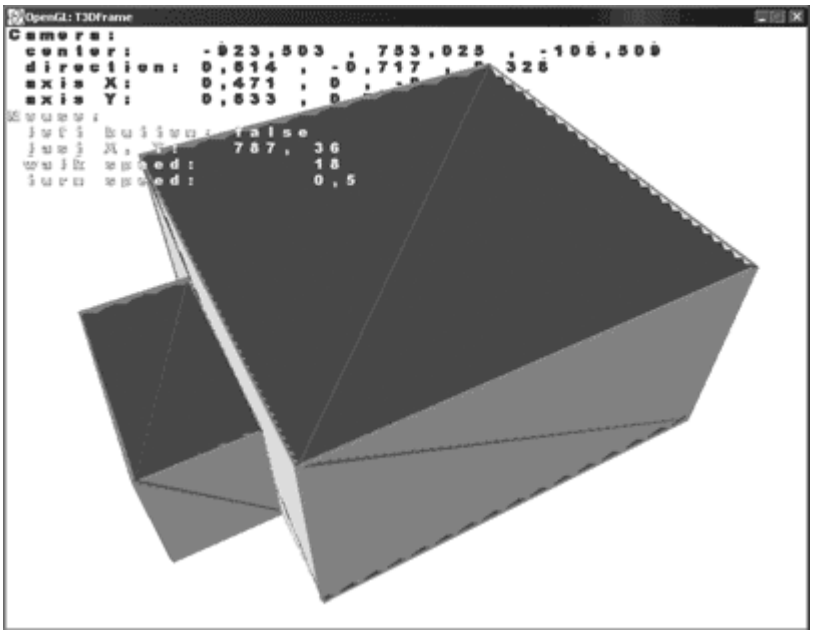
Jak jsem uvedl v 9.1, algoritmus zpracovává binární strom a slučuje postupně tělesa podle zadaných množinových operacích. Toto slučování těles popisuje tato část kapitoly.

Ideou algoritmu je nalézt místa, kde se tělesa protínají, a trojúhelníky v těchto místech rozdělit tak, aby existovaly pouze ty, které se nachází uvnitř nebo vně druhého tělesa (popřípadě v rovině stěny druhého tělesa). Všechny trojúhelníky tělesa *A* jsou pak klasifikovány vůči tělesu *B* (a naopak) jako ležící vně nebo uvnitř tělesa *B*. Pro koplanární případy trojúhelníků, kdy trojúhelník leží na stěně druhého tělesa, rozeznáváme případy *koplanární se stejnou orientací normál rovin* a *koplanární s opačnou orientací normál rovin*. Jakmile jsou trojúhelníky klasifikovány, tak se zachovají nebo vymažou z daného tělesa podle množinové operace, která je nad tělesy prováděna, a tělesa se sloučí do jednoho výsledného.

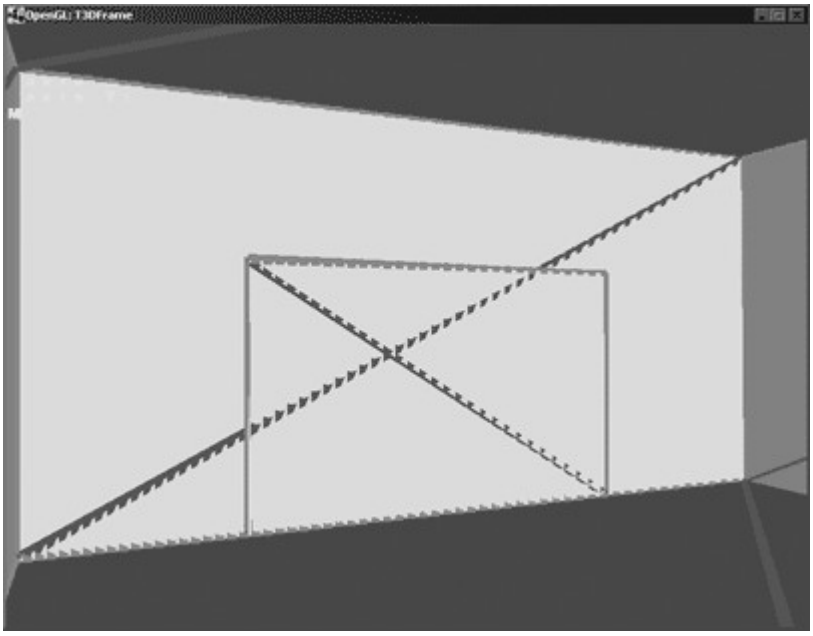
Algoritmus pro tělesa *A*, *B* a množinovou operaci *O* má tyto fáze:

1. nalezení úseček, ve kterých se tělesa A a B protínají,
2. rozdělení jednotlivých trojúhelníků podle dělících segmentů,
3. klasifikace trojúhelníků tělesa *A* vůči tělesu *B* a naopak,
4. smazání trojúhelníku z těles *A* a *B* podle operace *O* a sloučení těles *A*, *B* do výsledného tělesa $C = A\ O\ B$.

Jednotlivé fáze jsou znázorněny také na tělesech na obrázcích 14 a 15. Pro jednoduchost jsou jako slučovaná tělesa vybrány dva kvádry, které se dotýkají v jedné rovině a chceme je sloučit operací sjednocení.



Obr. 14. Dva kvádry, které chceme sloučit.



Obr. 15. Pohled zevnitř kvádru na stěnu, která je sdílena s druhým menším kvádrem, jehož hrany jsou taktéž zobrazeny (menší obdélník). Tělesa zatím nejsou sloučena.

Fáze 1 – hledání protínajících se trojúhelníků

Definice DLink. Než popíši tuto fázi, zavedu pojem *DLink* (dělící úsečka), což je segment, který vznikne průnikem trojúhelníku z tělesa *A* s nekoplanárními trojúhelníky z tělesa *B*. DLink vzniká pouze tehdy, když existuje neprázdný průnik takovýchto dvou trojúhelníků a průnikem není pouze jeden bod – tedy je to vždy úsečka.

Vstup. Vstupem do této fáze jsou dvě tělesa *A* a *B*, která jsou hranově reprezentovaná, tak jak je popsáno v 9.2. Výsledkem této fáze je seznam úseček

(DLinks), ve kterých trojúhelníky tělesa *A* protínají nekoplanární trojúhelníky tělesa *B*. Dalším výsledkem je seznam trojúhelníků, které obsahují aspoň jednu výše popsanou úsečku.

Popis. Chceme nalézt všechny DLinks, tedy segmenty, ve kterých se těleso *A* protíná s tělesem *B*. Testujeme všechny trojúhelníky z *A* oproti trojúhelníkům z *B* (n^2 testů).

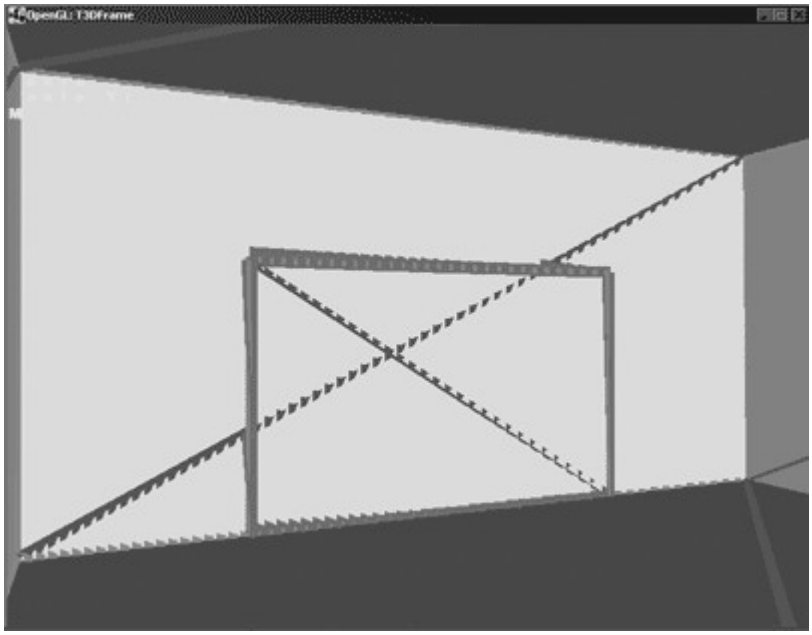
Pro každý trojúhelník T_A z tělesa *A* a pro každý trojúhelník T_B z tělesa *B*:

1. Nejprve otestujeme, zda se protínají krychle, jejichž všechny hrany jsou rovnoběžné s osami souřadnic a všechny body se nacházejí mezi minimy a maximy souřadnic bodů jednotlivých trojúhelníků. Pokud ano, pokračujeme dále, pokud ne, trojúhelníky se neprotínají.
2. Rozhodneme, zda jsou trojúhelníky koplanární, pokud ne, tak pokračujeme dále.
3. Jestliže průnik existuje, pak musí ležet na průniku rovin T_A a T_B. Nalezneme průnik rovin T_A a T_B, což je přímka *L*. Nalezneme průnik *L* s T_A jako úsečku T_{AL} a *L* s T_B jako úsečku T_{BL}. Nalezneme průnik T_{AL} s T_{LB}. Pokud je neprázdný a není tvořen pouze jedním bodem, pak tento průnik tvoří DLink, který asociujeme s trojúhelníkem T_A a s trojúhelníkem T_B.

Algoritmus nemusí zpracovávat koplanární trojúhelníky T_A a T_B proto, že jakmile dvě tělesa sdílí nějaké části stěny, pak celá část je buď zachována nebo smazána. Koplanární trojúhelníky musíme rozdělovat pouze tehdy, když je nějaká hrana *E* trojúhelníku T_A vnější. Pak musíme vytvořit DLink, který vznikne průnikem *E* a T_B. Ale ani to nemusíme dělat pro tyto trojúhelníky T_A a T_B, neboť hrana *E* je vnější a tedy existuje trojúhelník T_A', který také obsahuje hranu *E* a není koplanární s T_B. A tedy DLink vznikne tehdy, když jsou testovány trojúhelníky T_A' a T_B.

Výstup. Po ukončení fáze 1 algoritmu máme seznam DLinks a trojúhelníků, které je obsahují.

Složitost. Složitost této fáze je *O(m.n)*, kdy *m* je počet trojúhelníků tělesa *A* a *n* je počet trojúhelníků tělesa *B*.

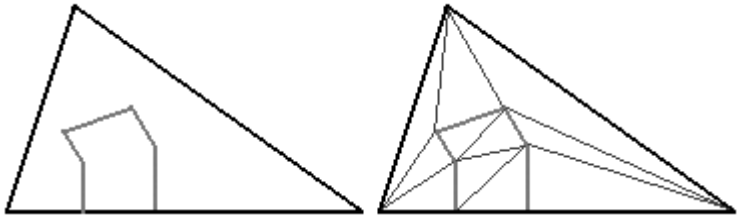


Obr. 16. DLinks byly nalezeny a zvýrazněny.

Fáze 2 – dělení trojúhelníků

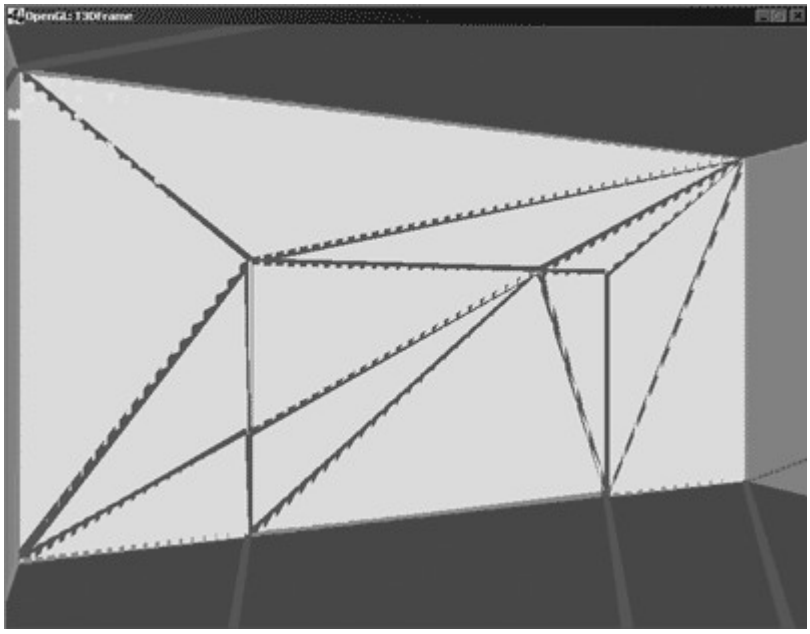
Vstup. Vstupem této fáze jsou trojúhelníky, které obsahují aspoň jeden z DLinks, které byly nalezeny ve fázi 1 a seznam všech DLinks, ve kterých se tělesa protínají.

Popis. Všechny trojúhelníky, které obsahují DLinks musíme nyní rozdělit na menší trojúhelníky, které ve sjednocení dávají původní trojúhelník. Na toto dělení jsou kladeny další požadavky, a to, že musí vést podél DLinks obsažených v původním trojúhelníku – viz obr. 17.



Obr. 17. Vlevo je trojúhelník obsahující DLinks a vpravo jeho rozdělení na menší trojúhelníky, podle DLinks, které obsahuje.

Navíc, pokud nějaký DLink se dotýká hrany E trojúhelníku A v bodě X, tak tento bod X propagujeme do trojúhelníku B, který s trojúhelníkem A sdílí hranu E. Tento trojúhelník také zahrneme do seznamu těch, které potřebují rozdělit. Musíme to udělat proto, aby po výsledném rozdělení bylo těleso opět 2-manifold. Hrana E bude totiž v bodě X rozdělena na dvě hrany, a kdybychom nerozdělili trojúhelník B, tak by hrana E trojúhelníku B již tento trojúhelník s ničím nespojovala.



Obr. 18. Jednotlivé trojúhelníky většího kvádru byly rozděleny podle DLinks. Zobrazen je pouze větší kvádr, vidíme jasně obrys stěny menšího kvádru.

Výstup. Po ukončení fáze 2 máme ke každému objektu seznam nových trojúhelníků, které vznikly po rozdělení trojúhelníků s DLinks. Všimněme si, že ke každému DLink vznikly (nebo byly zachovány) v každém z objektů 2 trojúhelníky obsahující jako hranu bývalý DLink.

Složitost. Složitost fáze je *O(m.n.log n)*, kdy *m* je počet trojúhelníků, které potřebují rozdělit a *O(n.log n)* je složitost použitého triangulačního algoritmu [18]. Velice hrubý odhad pro *n* bude větší z počtu vrcholů těles *A* a *B* plus 3. Příkladem může být jehlan, který sjednocujeme s krychlí, a podstava jehlanu leží na stěně jedné z krychlí. Pak kromě jednoho vrcholu jehlanu leží ostatní vrcholy ve stěně krychle a musí být tedy zavrhnuty do některé z triangulací.

Fáze 3 – značení trojúhelníků

Vstup. Vstupem této fáze je seznam DLinks a trojúhelníky těles *A* a *B*, pro které navíc platí jedna z následujících možností:

a) obsahuje alespoň jednu hranu, jež je shodná s jedním DLink,
b) ani jedna hrana trojúhelníku není shodná s žádným DLink.

Nemůže se stát, že by existovala hrana, která není shodná s nějakým DLink, ale obsahuje jej. Stejně tak nemůže existovat trojúhelník, který by DLink obsahoval a tento DLink nebyl shodný s jednou jeho hranou. Je to proto, že všechny takové trojúhelníky byly ve fázi 2 rozděleny podle DLinks tak, aby byly splněné výše zmíněné případy a) a b).

Popis. V této fázi probíráme postupně trojúhelníky z tělesa *A* a označujeme je jako *vnitřní* nebo *vnější* (případně *koplanární* atd. – viz 9. 3) trojúhelníky podle toho, zda jsou uvnitř nebo vně tělesa *B*. Stejně pak probíráme trojúhelníky z tělesa *B* a značíme je vůči tělesu *A*.

Následující popis se týká trojúhelníků z tělesa *A*, které označujeme vůči tělesu *B*. Případ pro trojúhelníky z *B* je analogický.

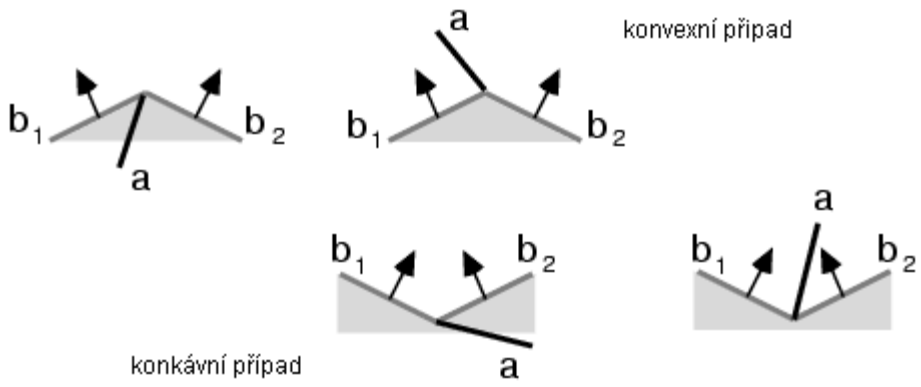
Případ a) Máme trojúhelník T_A z tělesa *A*, jehož hrana *E* je shodná s jedním s DLinks. Hrana *E* je tvořena body *X* a *Y*. V tělese *B* existují trojúhelníky T_{B1} a T_{B2}, které sdílí hranu *E*.

Je-li T_A koplanární s T_{B1} i s T_{B2}, pak podle orientace normál rovin určíme T_A jako *koplanární se stejnou orientací normál rovin* nebo jako *koplanární s opačnou orientací normál rovin*.

Je-li T_A koplanární s T_{B1} (resp. T_{B2}), pak nechť $T_A = DXY$, $T_{B1} = FXY$. Jestliže body *D* a *F* leží na stejných stranách přímky *XY* v rovině T_A (stejná jako rovina T_{B1}), pak T_A je označena jako *koplanární se stejnou orientací normál rovin* nebo jako *koplanární s opačnou orientací normál rovin* dle orientací normál rovin T_A a T_{B1}.

Pokud body *D* a *F* leží na opačných stranách, pak je tento případ shodný s případem, kdy T_A není koplanární ani s T_{B1} ani s T_{B2} a je klasifikován podle něj (viz níže).

Není-li T_A koplanární ani s T_{B1} ani s T_{B2} (nebo se jedná o speciální případ popsaný v předchozím odstavci), pak nechť $T_A = DXY$. K určení, zda bod *D* leží uvnitř nebo vně tělesa *B* (a tedy i celý trojúhelník T_A), potřebujeme normály rovin trojúhelníků T_{B1} a T_{B2} (b_1, b_2) a vektor *a* vycházející z projekce *D* na přímku *XY* a směřující k bodu *D*. Je-li hrana konvexní, pak *D* leží uvnitř tělesa *B* tehdy, když vektory b_1 a b_2 míří směrem od vektoru *a*. Je-li hrana konkávní, pak *D* leží mimo těleso, jestliže vektoru b_1 a b_2 míří směrem k vektoru *a*.



Obr. 19. Charakterizace trojúhelníku T_A vůči T_{B1}, T_{B2}. Převzato z [16]

Případ b) U takového trojúhelníku se díváme k jeho sousedům (trojúhelníky, se kterými sdílí hranu), jestli už byly ohodnoceny. Jestliže ano, pak přebereme jeho označení (propagace). Jestliže ne, označíme je způsobem podle toho, pod jaký případ spadají.

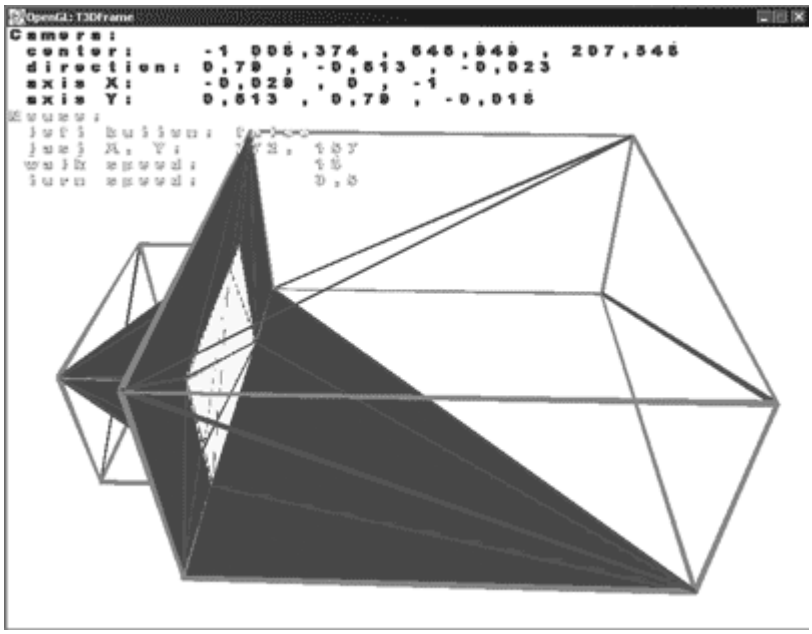
Speciální případ fáze. Speciální případ této fáze se vyskytne, když neexistují žádné DLinks. Tělesa se tedy vůbec neprotínají a jedno leží zcela uvnitř druhého nebo naopak zcela mimo něj. Toto je nutné rozhodnout pomocí jednoho ray-casting. Nechť těleso *A* je zcela obsaženo v krychli definované minimálními a maximálními hodnotami souřadnic vrcholů tělesa *B*. Pak vrháme paprsek z libovolného vrcholu tělesa *A* směrem k bodu, který leží mimo těleso *B*. Protne-li paprsek lichý počet trojúhelníků tělesa *B*, pak těleso *A* leží unitř tělesa *B*. Protne-li paprsek sudý počet trojúhelníků tělesa *B*, pak těleso *A* leží mimo těleso *B*. Analogicky postupujeme v případě, kdy těleso *B* je zcela obsaženo v krychli definované minimálními a maximálními hodnotami souřadnic vrcholů tělesa *A*.

Leží-li těleso *A* mimo těleso *B*, pak všechny trojúhelníky obou těles jsou označeny jako vnější.

Leží-li těleso *A* zcela uvnitř tělesa *B*, pak všechny trojúhelníky tělesa *A* jsou označeny jako vnitřní a všechny trojúhelníky tělesa *B* jako vnitřní. (Analogicky pro případ, kdy těleso *B* leží zcela uvnitř tělesa *A*.)

Výstup fáze. Na konci fáze 3 jsou veškeré trojúhelníky těles označeny jako vnitřní, vnější, koplanární se stejnou orientací normál rovin nebo koplanární s opačnou orientací normál rovin.

Složitost. Složitost je $O(m+n)$, kde m je počet trojúhelníků tělesa *A* a n je počet trojúhelníků tělesa *B*.



Obr. 20. Trojúhelníky byly označeny. Tmavé trojúhelníky jsou vnější, bílý obdélníček uprostřed tmavých je složen z několika trojúhelníků, které jsou označeny jako koplanární s opačnou orientací normál rovin. Některé trojúhelníky nejsou označeny vůbec (viz kapitola 9.6).

Fáze 4 – mazání trojúhelníků

Vstup. Vstupem této fáze jsou trojúhelníky těles *A* a *B*, které jsou označeny buď jako vnitřní, vnější, koplanární se stejnou orientací normál rovin nebo koplanární s opačnou orientací normál rovin.

Popis. V této fázi již pouze smažeme nebo zachováme trojúhelníky podle tabulky CSG operací a trojúhelníky z tělesa *B* přiřadíme tělesu *A*.

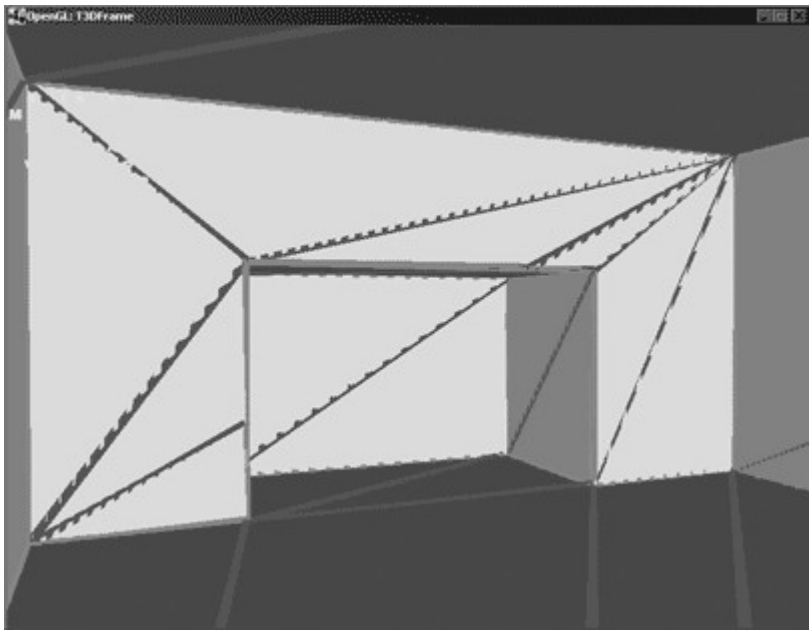
Pozice	***Trojúhelníky tělesa A***				***Trojúhelníky tělesa B***			
	vnější	*vnitřní*	*kopl. stejná orient.*	*kopl. opačná orient.*	*vnější*	*vnitřní*	*kopl. stejná orient.*	*kopl. opačná orient.*
sjednocení	ano	ne	ano	ne	ano	ne	ne	ne
průnik	ne	ano	ano	ne	ne	ano	ne	ne
rozdíl	ano	ne	ne	ano	ne	ano	ne	ne

Tab. 1. CSG operace pro trojúhelníky slučovaných těles, převzato z [16]

Tabulka 1 označuje trojúhelníky, které se mají smazat jako „ne“ a ty které zachovat pomocí „ano“. Trojúhelníky jsou rozděleny podle toho, zda patří tělesu *A* nebo tělesu *B* a podle jejich označení. V řádcích jsou pak množinové operace, které s tělesy provádíme.

Výstup. Výsledkem fáze 4 je těleso $C = A$ operace *B*.

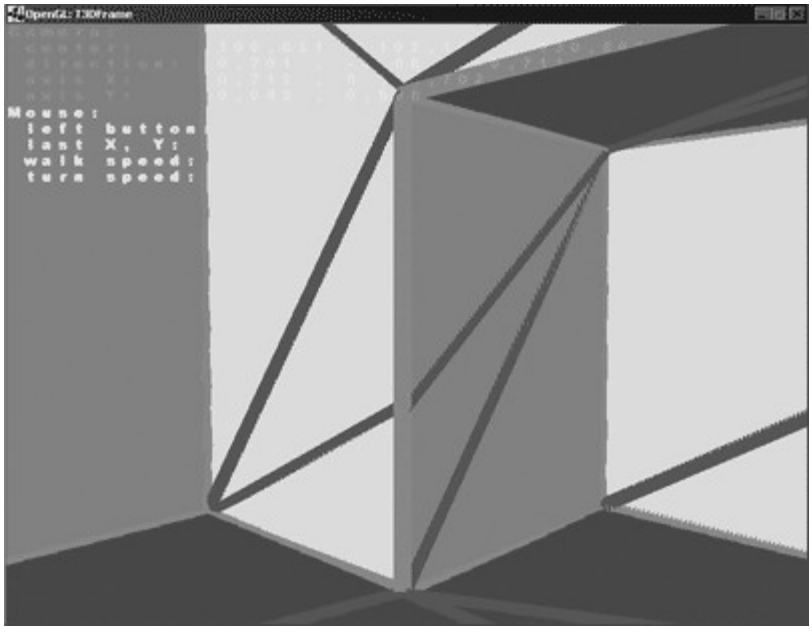
Složitost. Složitost fáze je $O(m+n)$, kde *m* je počet trojúhelníků tělesa *A* a *n* je počet trojúhelníků tělesa *B*.



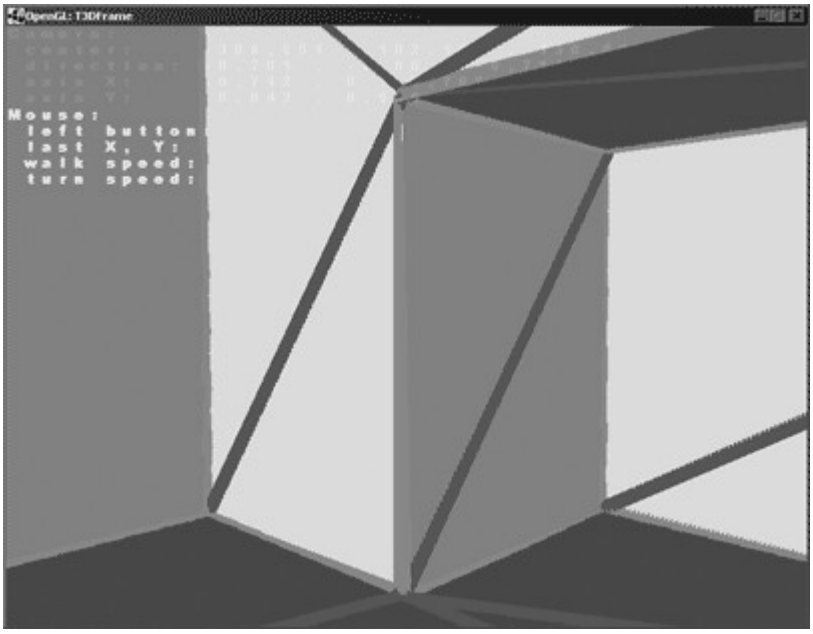
Obr. 21. Výsledek přidání menšího kvádru
k většímu z pohledu zevnitř velkého kvádru.

Poznámka k textu Hubbarda

Hubbard zmiňuje ještě odstranění vrcholů, které zbytečně zvyšují počet trojúhelníků výsledného tělesa. Jedná se o vrcholy, ze kterých vychází pouze vnitřní hrany, nebo vrcholy, ze kterých vychází právě dvě vnější hrany, které leží v jedné přímce. Hubbard píše, že tyto vrcholy se mají odstranit před rozdělením trojúhelníků těles *A* a *B* (fáze 2). Lze to udělat i dodatečně na výsledném tělese.



Obr. 22. Na obrázku uprostřed vidíme zbytečný „vrchol“ tělesa.



Obr. 23. Poslední fáze mé implementace odstraní tyto vrcholy.

9. 4 Celková složitost algoritmu

Ani práce Leidlawa a Hubbarda neuvádějí odhad složitosti algoritmu. Pro úplnost zde uvádím hrubý odhad složitosti celého převodu tělesa z CSG reprezentace do hranové reprezentace.

Vstupem algoritmu (podle 9.1) je binární strom, který má v listech tělesa a v uzlech množinové operace. Abychom získali hranovou reprezentaci tělesa popsaného takovým stromem, musíme vykonat tolik sloučení těles, kolik je uzlů daného stromu. Nechť je těchto uzlů *a*.

Složitost slučování tělesa *A*, které má *m* trojúhelníků a *k* vrcholů, a tělesa *B*, které má *n* trojúhelníků a *l* vrcholů:

Fáze 1. Složitost fáze je *O(m.n)*.

Fáze 2. Složitost fáze je *O((m+n).(k+l).log (k+l))*.
Počet trojúhelníků, které potřebují rozdělit, odhadneme součtem trojúhelníků obou těles. Počet vrcholů, které tvoří polygon určený k triangulaci odhadneme součtem počtu vrcholů těles *A* a *B*.

Fáze 3. Složitost fáze je *O(m+n)*.

Fáze 4. Složitost fáze je *O(m+n)*

Nejsložitější je tedy fáze 2, která má složitost *O(b.c.log c)*. Kde *b* je počet trojúhelníků těles a *c* je počet vrcholů těles. Složitost této fáze by se dala vylepšit implementováním rychlejšího algoritmu pro triangulaci.

Máme-li provést *a* slučování těles, je výslednou složitostí převodu tělesa z CSG reprezentace do hranové reprezentace *O(a.b.c.log c)*.

9. 5 Aplikace algoritmu na data z T3D souboru

Právě popsaný algoritmus očekává jako vstup binární strom CSG reprezentace tělesa, které chceme převést. O tělesech v listech předpokládá, že jsou reprezntována hranově a skládají se pouze z trojúhelníků. Navíc u každého trojúhelníku musíme znát normálu jeho roviny, která musí směřovat ven z tělesa (podrobněji viz 9. 1 a 9. 2).

Data v T3D souboru netvoří tento binární strom a ani jeho tělesa nejsou hranově reprezentována a složena pouze z trojúhelníků, což bylo třeba vyřešit. Co splňuje předpoklad algoritmu, je správná orientace normály, která je u každého polygonu přítomna.

Tělesa v T3D souboru jsou zapsána jako seznam polygonů, a tak je nejprve nutné převést je do hranové reprezentace a polygony triangulovat tak, jak to požaduje zvolený CSG algoritmus (viz 9. 1). Převod těles nacházejících se v T3D souboru s sebou nese problémy, které jsou popsány v 9. 7. Zde zmíním jen jeden problém, kvůli kterému jsem popsaný CSG algoritmus pozměnil.

Problém spočívá v tom, že T3D soubor neobsahuje binární strom, ale pouze posloupnost těles spolu s určením CSG operace. Těleso (mapa) se tedy vytváří tak, že se jednotlivé objekty sekvenčně slučují a k velkému obejktu jsou přidávány malé. Algoritmus je tím ve fázi 3 označování trojúhelníků zbytečně zdržován propagací u trojúhelníků velkého tělesa, které se z velké části nachází vně druhého tělesa. Tento problém řeším pozměněním algoritmu.

9. 6 Provedené změny v algoritmu

Předchozí podkapitola zmínila problém sekvence slučování těles. Tento problém řeším následovně.

Zakázání oddělených těles. Implementaci algoritmu jsem zjednodušil tak, že neuvažuje spojování těles, která se neprotínají. UnrealEd sice povoluje tvorbu takovýchto map, ale skoro nikde se to v mapách nevyskytuje. Díky tomuto omezení bylo možné zjednodušit fázi 3, tím pozměnit i fázi 4, a algoritmus zrychlit (i když ne výrazně – odhad složitosti zůstává stejný kvůli náročné fázi 2).

Přináší to ale také problém, v jakém pořadí tělesa slučovat, neboť tělesa v pořadí tak, jak jsou zapsána v T3D souboru, ne vždy na sebe navazují. Implementoval jsem metodu, která dokáže tělesa přeskládat tak, aby na sebe vždy navazovala, a je možné je postupně slučovat.

Změna ve fázi 3. Jelikož se tělesa vždy musí protínat, tak není nutné označovat všechny trojúhelníky těles. Klasifikujeme pouze ty, které byly vytvořeny při rozdělování trojúhelníků ve fázi 2.

Změna ve fázi 4. V této fázi pak neprocházíme všechny trojúhelníky těles, ale pouze ty, které byly označeny. Následně mažeme nebo zachováváme trojúhelníky této množiny. Pokud je nějaký trojúhelník smazán, tak si zapíšeme ty jeho hrany, které se neshodují s žádným z DLinks. Jako poslední krok mažeme trojúhelníky, které byly kompletně odděleny od zbytku tělesa. Procházíme seznam uložených hran smazaných trojúhelníků, a pokud je k hraně stále připojen nějaký trojúhelník, tak jej smažeme a rekurzivně i jeho sousedy. Nakonec trojúhelníky druhého tělesa přidáme do prvního.

9. 7 Implementace algoritmu

Zvoleným jazykem implementace je Java. Zvolil jsem jej pro jeho univerzálnost a přenositelnost mezi systémy.

Triangulace. Pro fázi 2 je nutná triangulace. Použil jsem dostupnou již implementovanou triangulaci z [19]. Triangulace byla implementována v C++ a musel jsem ji tedy přepsat ekvivalentně do Javy. Jedná se o algoritmus používající zametací přímku popsaného v [18]. Algoritmus byl implementován Wu Liang

a rozšířen tak, aby dokázal triangulovat polygony s dírami. Složitost algoritmu je *O(n.log n)*, kde *n* je počet vrcholů triangulovaného polygonu.

Jedná se o triangulaci v 2D, proto před triangulací je třeba udělat projekci úseček trojúhelníků (a DLinks) do vhodné roviny. Zjevně se nabízí přímo rovina definovaná vrcholy trojúhelníku. Jako počáteční bod osy souřadnic v této rovině jsem použil jeden z vrcholů a jako osy vždy jednu z hran trojúhelníku a vektor k ní kolmý v rovině trojúhelníku.

Algoritmus triangulace občas vyprodukuje „ne-trojúhelník“, čili označí tři body v přímce za trojúhelník. Tato chyba je mou implementací detekována a opravena. Složitost opravy je *O(m.(n-m))*, kde *m* je počet chybných trojúhelníků a *n* je celkový počet trojúhelníků vyprodukovaných triangulací.

9. 8 Problémy algoritmu a dat T3D souboru

Při zpracovávání dat z T3D souboru jsem narazil na tyto problémy:

- některé tělesa v T3D souboru nejsou součástí mapy,
- chybný předpoklad o tělesech (existují tělesa, která nejsou 2-manifold),
- nízka přesnost dat,
- numerická nestabilita mé implementace (důsledek řešení nízké přesnosti dat).

Tělesa, která nejsou součástí mapy. V datech T3D souboru se nachází vždy jedno těleso, které nepatří do výsledné mapy. Lze jej rozeznat podle toho, že neobsahuje žádnou CSG operaci a je vždy uvedeno jako první v souboru. Řešení tohoto problému ponechávám uživateli T3D aplikace (viz kapitola 10), který má možnost toto těleso nezahrnout do mapy.

V T3D souboru, se také mohou nacházet tělesa, která do mapy vůbec nepatří. Tato tělesa sloužila tvůrcům mapy pro urychlení práce a v souboru mapy UT je nechali pro případ, kdyby se k mapě chtěli někdy vrátit. Tělesa jsou jednoduše odhalitelná, neboť nejsou součástí mapy a jsou od těles tvořících mapu viditelně vzdálena. Tělesa lze z mapy smazat pomocí UnrealEd a exportovat do T3D souboru pouze relevantní soubory. Stručný návod k UnrealEd uvádím v příloze D.

Chybný předpoklad o tělesech T3D souboru. Při práci na CSG algoritmu jsem zjistil, že proti původnímu předpokladu některé objekty nejsou 2-manifold. Tyto objekty jsou tří typů.

Prvním typem jsou samostatné polygony, které jsou do map přidávány z estetických důvodů (například nápis na zdi, který obsahuje jinou texturu než samotná zeď), a tedy nemusí být algoritmem vůbec zpracovány.

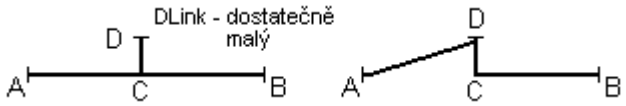
Druhým typem jsou tělesa, jejichž některé stěny jsou zapsány tak, že neuzavírají těleso. Tato chyba je při přechodu do hranové reprezentace detekována a upravena. Úprava objektů probíhá tak, že se najde seznam bodů, které k sobě mají „blízko“. Blízké body jsou takové, které nejsou vzdáleny více než zvolená konstanta. Z těchto bodů, je pak vybrán bod s největším zastoupením a souřadnice ostatních bodů jsou přepsány souřadnicemi vybraného bodu. Bohužel to někdy způsobí, že takto upravené objekty přestanou navazovat na jiné a nelze je sloučit.

Třetím typem jsou tělesa, která do mapy patří, ale ve skutečnosti nejsou 2-manifoldy. Chybí jim například nějaká stěna a nejsou tedy uzavřené. Neviděl jsem však těleso, které by bylo pro mapu podstatné a nebylo 2-manifoldem. Tedy všechna tělesa tohoto typu, na která jsem narazil, nemusela být do mapy vůbec zarnuta. Jsou v mapě jen kvůli zkrášlení prostředí.

Nízká přesnost. Veškerá číselná data v T3D souboru jsou zapsána s přesností na 6 desetinných míst. Jsou-li tělesa natočena, vzniká nebezpečí, že se přestanou

dotýkat, a tedy je nebude možno sloučit. Tato chyba se projevuje hlavně ve fázi 1, kdy algoritmus nenalezne některé DLinks, jelikož jsou tělesa od sebe vzdálena o větší hodnotu, než je přesnost, se kterou algoritmus počítá.

Numerická nestabilita mé implementace. Předchozí dva problémy jsem se snažil vyřešit změnami přesností, se kterými algoritmus počítá ve svých různých částech. Je to zejména v místě, kde počítá průmět bodu na přímku, a v testu, je-li daný bod ještě v přímce nebo mimo ni. Tento zásah sice přinesl zlepšení – algoritmus je schopen převést některé z objektů, které se skoro dotýkají, ale také vytvořil umělou chybu ve fázi 3 při dělení trojúhelníků. Pokud nějaký DLink má počátek na hraně trojúhelníku, ale celý v ní neleží a je dostatečně krátký, pak je algoritmem považován za ležící v hraně. Hrana je pak rozdělena na úsečky, které neleží všechny v přímce. Na hraně trojúhelníku se vytvoří zub.



Obr. 24. *AB* tvoří hranu trojúhelníku a *CD* je DLink,bod *D* je však blízko hrany *AB* a je považován za součást hrany. Při dělení trojúhelníků s hranou *AB* dojde k vytvoření zubu na původní hraně *AB*.

Poslední problém s přesností se vyskytuje při počítání průniku dvou rovin trojúhelníků ve fázi 1, kdy výsledná přímka někdy neprotne hranu některého z trojúhelníku, i když by měla.

Možné řešení problémů převodu. Objekty, které při převodu činí potíže, je možné upravit ručně v editoru map Unreal Tournamentu – UnrealEd. Lze je různě posunout nebo zvětšit či zmenšit tak, aby se objekty začaly překrývat. Algoritmus by pak měl být schopen je sloučit. Postup, jak upravit objekty v UnrealEd, je uveden v příloze D. Bohužel tento postup někdy také selhává ve fázi hledání DLinks kvůli problémům s přesností počítání průniku rovin.

10 T3D aplikace a postup získání NavMesh z T3D souboru

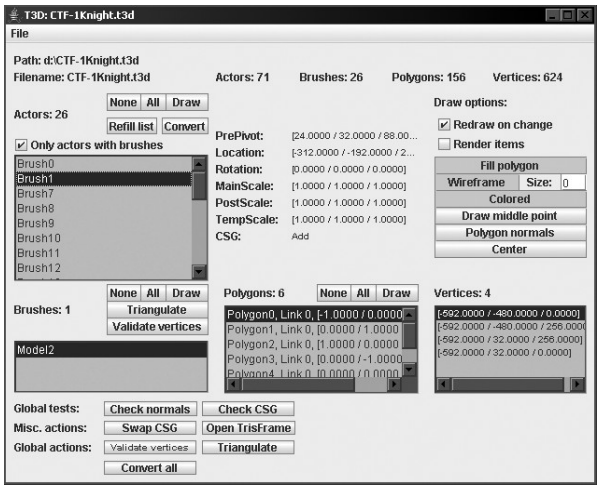
Pro tvorbu NavMesh z T3D dat, jsem naprogramoval T3D aplikaci, která umožňuje data načíst, převést do hranové reprezentace a sloučit pomocí algoritmu z kapitoly 9 do mapy. Ze získaného tělesa (mapy) je pak možné vygenerovat přímo NavMesh, jak je popsáno v 8. 4. Jako součást své aplikace jsem implementoval vizualizaci používaných těles. Uživatel má možnost si prohlížet tato tělesa z různých stran a určovat způsob jejich vykreslení. Lze také vizualizovat jednotlivé kroky algoritmu a sledovat případné chyby algoritmu.

Tato kapitola se zabývá pouze sekvencí kroků pro získání T3D souboru zvolené mapy a jeho převedení na NavMesh. Postup instalace, spuštění a uživatelský popis ovládacích prvků aplikace je uveden v příloze B. Další příloha D obsahuje popis editoru map UnrealEd.

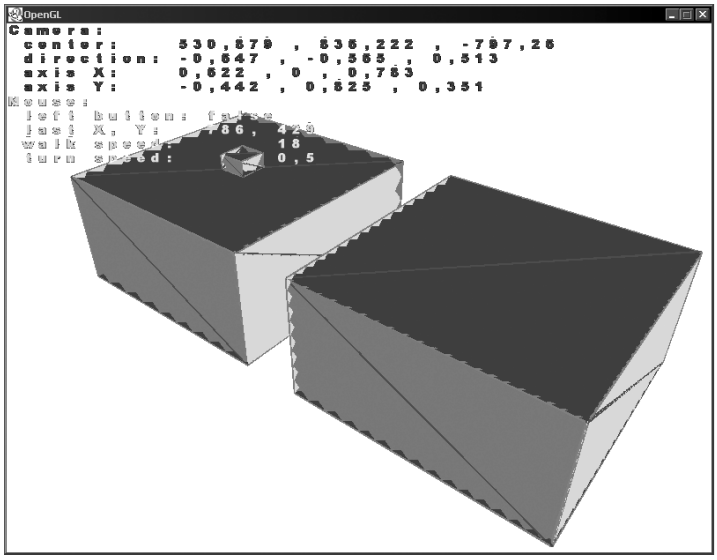
10. 1 Postup získání NavMesh z T3D souboru

NavMesh se z T3D souboru získá v T3D aplikaci následujícím způsobem:

1. Po spuštění aplikace se objeví následující 2 okna – T3DFrame, OpenGL. T3DFrame slouží pro ovládání objektů z T3D souborů a okno OpenGL objekty zobrazuje.



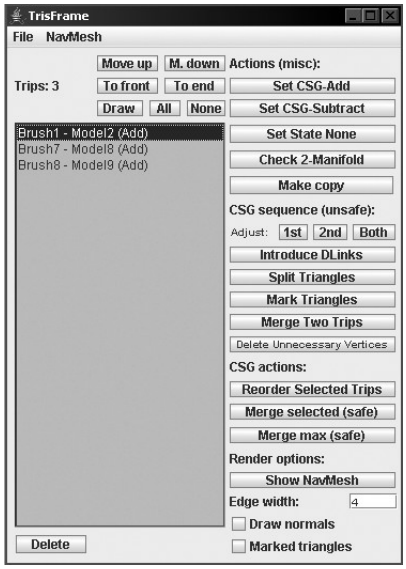
Obr. 25. T3D okno



Obr. 26. OpenGL okno.

2. V menu se naléza jediný příkaz 'Load', kterým je možné nahrát T3D soubor exportovaný z UnrealEd. UnrealEd exportuje také jeden nepotřebný objekt, který se vypisuje v 'Actors listu' jako první – není mu přiřazena žádná CSG operace a nepatří do výsledné mapy.
3. Tlačítkem 'All', které se nachází nad listem s actory, vybereme všechny objekty. Kliknutím na prvním objekt (který nepatří do scény) jej odznačíme a pak klikneme na tlačítko 'Convert'. Tím se objekty převedou do hranové reprezentace. Během převodu dochází k úpravám vrcholů polygonů a jejich triangulaci.

4. Po dokončení konvertování objektů do hranové reprezentace se může objevit chybová hláška se jmény objektů, které netvoří 2-manifoldy, či u kterých není uveden typ CSG operace.
5. Konvertované objekty se zobrazí v listu nového okna – TrisFrame.



Obr. 27. Okno TrisFrame.

6. Klikneme na tlačítko 'All' a pak na tlačítko 'Reorder Selected Trips' pro uspořádání objektů do posloupnosti tak, aby každý objekt navazoval na předcházející.
7. Dále klikneme na tlačítko 'All' a pak na tlačítko 'Merge selected (safe)' a algoritmus sloučí objekty podle jejich typu CSG operace.
8. V menu NavMesh vybereme položku 'Save NavMesh' a jako jméno souboru zadáme název mapy a příponu '.xml'. V této podobě ji očekává NavMesh bot.

Pokud chceme otestovat NavMesh bota na nové navigační mřížce, pak xml soubor s NavMesh nakopírujeme do adresáře Pogamutu. Další informace o možnosti T3D aplikace jsou uvedeny v příloze C.

10. 2 Převedené mapy

Na přiloženém CD v adresáři „ut_maps“ jsou mapy, které se podařilo úspěšně převést. Výše popsaný postup funguje bez problémů na datech z CTF-1Knight.t3d, DM-OblivionSimple a DM-Simplest. Při zpracovávání CTF-Simplest.t3d je nutné ručně upravit sekvenci slučování objektů.

Mapu CTF-1Knight vytvořil kolega Michal Bída. A před převodem nemusela být upravována pomocí UnrealEd.

Ostatní mapy byly kvůli problémům při převodu do hranové reprezentace popsaným v 9. 8 upraveny v UnrealEd. Objekty, které činily při slučování problémy, byly buď zvětšeny nebo smazány. Mapa CTF-Simplest však zachovává většinu těles původní mapy a příliš se od ní neodlišuje.

11 PoGamUT a jednoduchý bot používající NavMesh

PoGamUT [20] je middleware, který zpřístupňuje ovládání postav v Unreal Tournamentu. Middleware je naprogramován v jazyce Jython [21], což je kombinace jazyka Python [22] a Javy [23]. Pro komunikaci s UT je použit server GameBots (dále GB) [24], který je napsán v UnrealScriptu jako rozšíření UT.

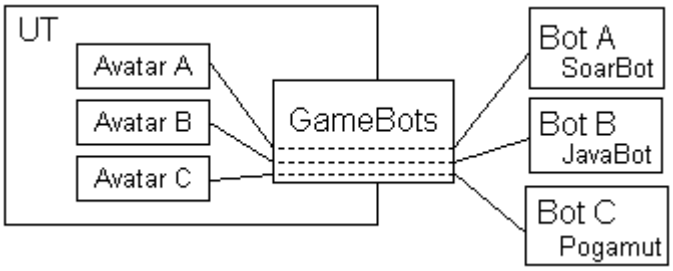
Pomocí middleware lze naprogramovat boty s libovolným algoritmem napsaným v Pythonu nebo Javě. Tím je prostředí UT otevřeno akademické obci pro experimentování.

Součástí middleware je také pyPOSH, který byl naprogramován Andy Kwongem [25] na univerzitě Bath v Anglii podle prací J. J. Bryson [26]. POSH je symbolická, reaktivní architektura, která staví na paradigmatu „behavior-oriented-design“, designu orientovaného na chování, který vychází z Brooksova přístupu k řízení robotů [27], nicméně Bryson jej pro potřeby své architektury rozšiřuje [28]. Podrobnější popis této architektury je však nad rámec této bakalářské práce.

11. 1 Popis middleware

GameBots

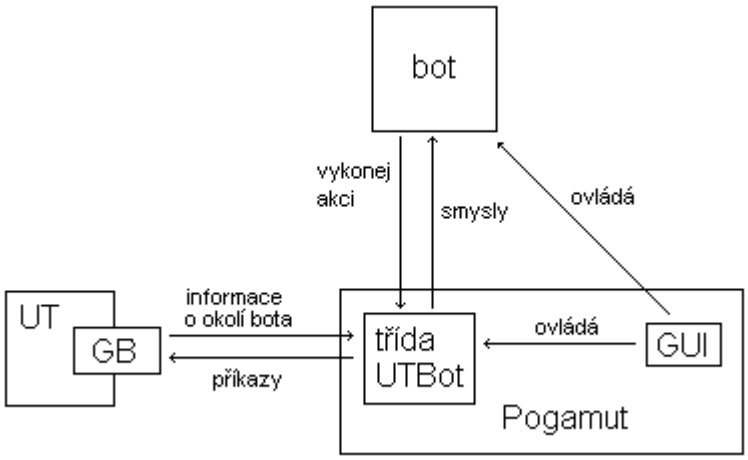
GameBots [24] funguje jako server, ke kterému je možné se připojit pomocí TCP protokolu. GB implicitně poslouchá na portu 3000, na kterém akceptuje nová spojení. GB definuje textový protokol, se kterým se dorozumívá s klientem. Při připojení ke GB je klient povinen poslat zprávu „INIT“. Po obdržení této zprávy se GB pokusí vytvořit nového bota v prostředí UT. Pokud se to zdaří, GB posílá klientovi zprávu „NFO“, která obsahuje základní informace o modu hry a mapě, která je v UT právě spuštěna. Po té GB začne posílat klientovi informace o tom, co bot vidí. Klient může zasílat příkazy, které má bot vykonat.



Obr. 28. GameBots jako server pro různé boty.

Pogamut

Každý bot vytvořený v tomto middleware musí být potomkem třídy UTBot. Tato třída zajišťuje komunikaci s GB, kterému zasílá příkazy, co má bot dělat, a přijímá od něj informace o tom, co právě bot vidí. Třída ukládá všechny důležité informace do vlastních datových struktur, které jsou pak z potomka přístupny a se kterými algoritmus bota pracuje.



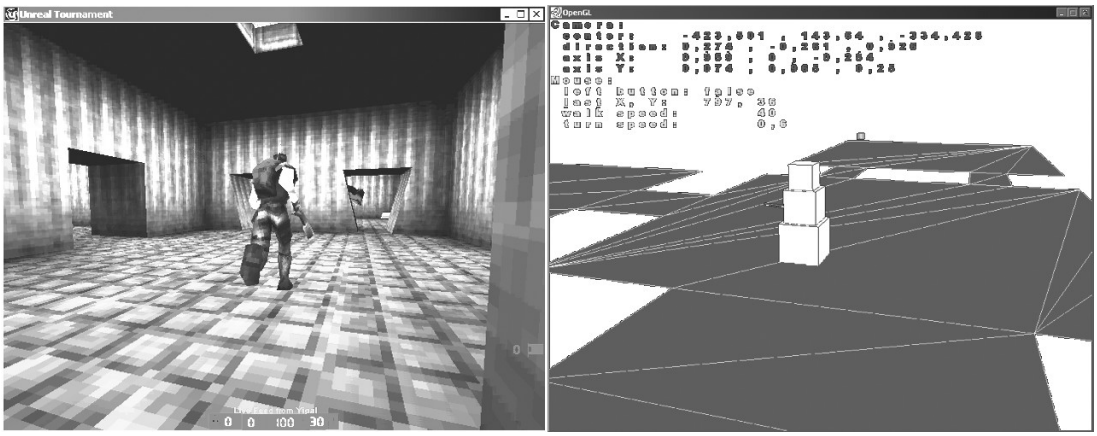
Obr. 29. Schéma práce Pogamutu

11. 2 NavMesh bot

NavMesh bot slouží jako demonstrace použití navigační mřížky pro pohyb prostředím. Pohyb v prostředí pomocí navigační mřížky je řešen elementárním způsobem popsaným v kapitole 7. Bot se snaží náhodně procházet prostředím a nalézt nějakou silnější zbraň. Pro nalezení cesty v NavMesh jsem použil algoritmus A*, jmenovitě pak implementace popsaná v [29] a nazvaná *A* machine*. Po obdržení cesty v NavMesh se bot pohybuje vždy směrem k nejbližšímu bodu následujícího trojúhelníku. To znamená, že se pohybuje kolmo na hranu, kterou sdílí trojúhelník, ve kterém právě stojí, s následujícím trojúhelníkem v cestě.

Bot se po mapě pohybuje náhodně.

Součástí bota je také vlastní vizualizér navigační mřížky, po které se právě pohybuje. Tato vizualizace slouží k lepší představě, jak bot vnímá prostředí, ve kterém se pohybuje – viz obrázek 30.



Obr. 30. NavMesh bot. Vlevo je zobrazen v prostředí UT,
vpravo v modelu prostředí navigační mřížky

12 Závěr

Tato práce se zabývá modely virtuálních prostředí pro potřeby botů v FPS hrách. Byly představeny modely navigačních bodů a navigační mřížky. Prezentoval jsem způsob, jakým je možné automaticky generovat navigační mřížku pro mapy Unreal Tournamentu a T3D aplikaci, pomocí níž lze navigační mřížku vygenerovat z T3D soubor.

Problémy s T3D daty a numerickou nestabilitou implementace CSG algoritmu způsobují, že T3D aplikace nemůže být použita pro jednoduché získání NavMesh z existujících map Unreal Tournamentu. Mapy musí být ručně upravovány pomocí UnrealEd, ale ani s jeho pomocí nelze převest většinu map. Nicméně se podařilo upravit původní mapu CTF-Simple do podoby, která byla nazvána CTF-Simplest, a která není zcela triviální.

Věřím, že důkladným rozborem práce implementovaného algoritmu a lokalizování částí, ve kterých algoritmus selhává při zpracování dat z T3D, bude možné zvýšit jeho robustnost a převést většinu map z Unreal Tournament.

Nic méně již teď je možné T3D aplikaci využívat na jednodušších mapách a experimentovat s NavMesh jako modelem prostředí UT, jak dokazuje použití navigační mřížky jednoduchým NavMesh botem.

Ve své další práci se chci zaměřit právě na popsané problémy CSG algoritmu a na experimentování s navigační mřížkou. Získanou mapu bych chtěl také dále zpracovávat a tvořit například automaticky matici viditelnosti, která by rozšířila model virtuálního prostředí UT.

LITERATURA

[1] Epic MegaGames: Unreal Tournament, http://www.unrealtournament.com, 6. 8. 2006.

[2] Valve: Half-Life, http://www.planethalflife.com/, 6. 8. 2006.

[3] ID Software: Quake II, http://www.idsoftware.com/games/quake/quake2/, 6. 8. 2006.

[4] Greg Snook (2000): Simplified 3D Movement and Pathfinding Using Navigation Meshees. *Game Programming Gems* (Mark DeLoura, ed.), Charles River Media.

[5] ID Software: Wolfenstein, http://en.wikipedia.org/wiki/Wolfenstein_3D, 6. 8. 2006.

[6] ID Software: Doom, http://en.wikipedia.org/wiki/Doom, 6. 8. 2006.

[7] Counter Strike: Condition Zero, http://www.ritual.com/cz/index.php, 6. 8. 2006.

[8] Valve: Counter-Strike, http://en.wikipedia.org/wiki/Counter-Strike, 6. 8. 2006.

[9] Valve: Half-Life SDK, http://www.planethalflife.com/half-life/files/, 6. 8. 2006.

[10] Ondřej Burket (2006): Bakalářská práce, Deathmatch Twins, MFF-UK, Praha.

[11] James Mattews (2002): Basic A* Made Simple. *AI Game Programming Wisdom* (Steve Rabin, ed.), Charles River Media, 105-113.

[12] Lars Lidén (2002): Strategic and Tactical Reasoning with Waypoints. *AI Game Programming Wisdom* (Steve Rabin, ed.), Charles River Media.

[13] Paul Tozour (2002): Building a Near-Optimal Navigation Mesh, *AI Game Programming Wisdom* (Steve Rabin, ed.), Charles River Media, 171-185.

[14] Formát souboru UNR, http://unreal.epicgames.com/Packages.htm, 6.8.2006.

[15] BeyondUnreal Wiki, http://wiki.beyondunreal.com/wiki/T3D_File, 6.8.2006.

[16] Philip M. Hubbard (1990): Constructive Solid Geometry for Triangulated Polyhedra, Department of Computer Science, Brown University, Providence, Rhode Island 02912.

[17] David. H. Leidlaw, W. Benjamin Trumbore, John F. Hughes (1986): Constructie solid geometry for polyhedral objects, *Proceedings of the 13th annual conference on Computer graphics and interactive techniques*.

[18] Mark de Berg, Marc van Kreveld, Mark Overmars, and Otfried Schwarzkopf: Triangulation, *Computational Geometry: Algorithms and Applications, 2nd Edition*.

[19] Wu Liang: Poly2Tri, http://www.mema.ucl.ac.be/~wu/Poly2Tri/poly2tri.html, 6.8.2006.

[20] Bída M., Burkert O., Brom C., Gemrot J. (2006): Pogamut – Platforma pro prototypování botů v Unreal Tournamentu. *Kognice a umělý život* **VI**, Slezská univerzita v Opavě, Fiozoficko-přírodovědecká fakulta, Ústav informatiky, 59 – 66.

[21] Jython, http://www.jython.org, 6.8.2006.

[22] Python, http://www.python.org, 6.8.2006.

[23] Sun: Java, http://java.sun.com, 6.8.2006.

[24] Gamebots, http://www.planetunreal.com/gamebots

[25] Andy Kwong (2003): A Framework for Reactive Inteligence through Agile Component-Based Behaviors, *Master's thesis*, Department of Computer Science, University of Bath.

[26] Joanna J. Bryson (2001): Inteligence by design: Principles of Modularity and Coordination for Engineering Complex Adaptive Agent. *PhD Thesis*, MIT, Departement of EECS, Cambridge, MA.

[27] Rodney A. Brooks (1986): A robust layered control system for a mobile robo*t*, *IEEE Journal of Robotics and Automation*.

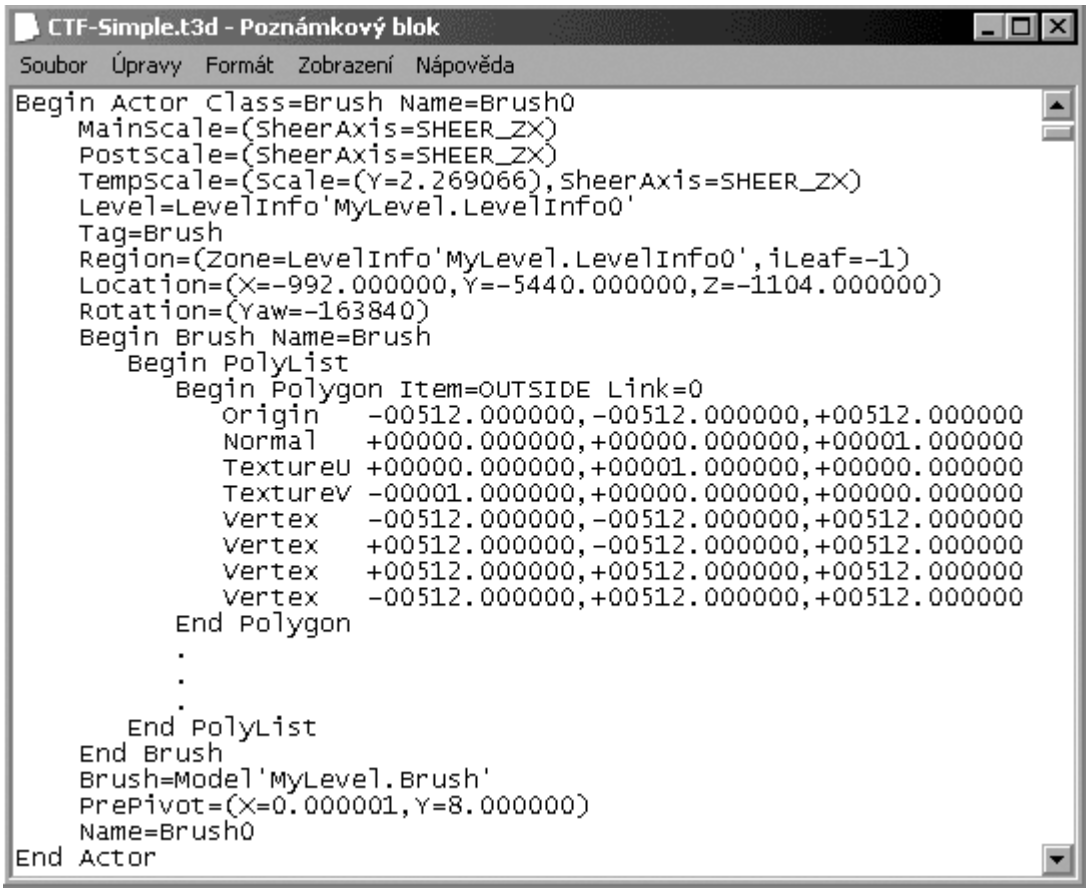
[28] Joanna J. Bryson (2003): The behavior-oriented design of modular agent intelligence, *Agent Technologies, Infrastructure, Tools, and Applications for e-Services* (R. Kowalszyk, Jörg P. Müller, H. Tianfield, and R. Unland, eds.), Springer.

[29] Dan Higgins (2002): Generic A* Pathfinding, *AI Game Programming Wisdom* (Steve Rabin, ed.), Charles River Media, 114-121.

Formát T3D souboru

T3D soubor nese v sobě informace o všech objektech a tělesech, ze kterých se mapa skládá.

Mapu v UnrealEdu tvoříme pomocí tzv. brushes (tvary, tělesa), které jsou ze scény buď odečteny nebo do ní přidány. Tvorba map tedy funguje na principu konstruktivní geometrie, ve které se pomocí množinových operací (sjednocení, rozdíl, průnik) spojují jednoduchá tělesa a vznikají tělesa složitější. Oproti klasické konstruktivní geometrii, která využívá několik typů těles (kvádr, koule, válec, jehlan, apod.), lze v UnrealEdu konstruovat a použít libovolné uzavřené těleso. Výjimku pak tvoří zvláštní polygony, které lze do mapy přidat z estetických důvodů. Příkladem je nápis na zdi, pro který je použita jiná textura než pro zbytek stěny.



Obr. 31. Příklad T3D souboru

T3D soubor je seznam jednotlivých objektů, které tvoří mapu nebo se na mapě nachází. Kromě jednotlivých těles tvořících mapu obsahuje T3D soubor také seznam předmětů, světel a jiných významných bodů mapy. Formát souboru funguje na principu otevírání a zavírání značek (Begin / End Actor, Begin / End Brush, atd.).

a) značky

Map – první značka v celém souboru, její otevření zahajuje výstup mapy a uzavření končí soubor

Actor – značka, která definuje objekt (*brush*, předmět) nacházející se na mapě

Brush – podznačka *Actor* obsahuje definici tělesa
PolyList – následující značka za *Brush* zahajuje seznam polygonů
Polygon – značka se vyskytuje v těle *PolyList*, obsahuje definici jednoho polygonu

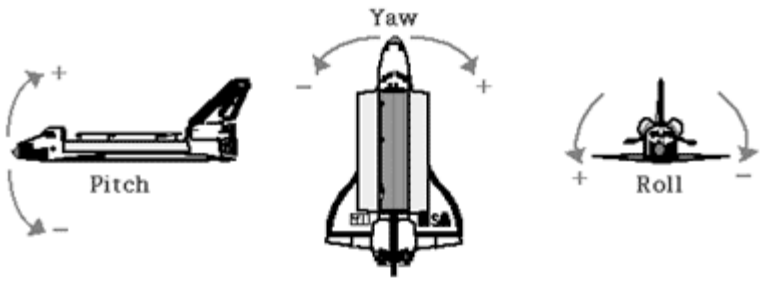
b) atributy značek

Značka	***Atribut***	***Význam***	***Interpretované hodnoty***	***Defaultní hodnota***
Actor	Name	název	název	
	MainScale	první zvětšení	(Scale=(osa=hodnota,...), …)	x=1, y=1, z=1
	PostScale	poslední zvětšení	(Scale=(osa=hodnota,...), …)	x=1, y=1, z=1
	TempScale	zvětšení	(Scale=(osa=hodnota,...), …)	x=1, y=1, z=1
	Location	umístění v prostoru	(osa=hodnota, …)	x=0, y=0, z=0
	Rotation	rotace (yaw, pitch, roll)	(typ=hodnota,...)	yaw=0, pitch=0, roll=0
	PrePivot	umístění polygonů před rotací	(osa=hodnota, …)	x=0, y=0, z=0
	CsgOper	CSG operace	CSG_Add, CSG_Subtract	
Polygon	Origin	neznámý význam	x, y, z	
	Normal	normála roviny definované polygonem	x, y, z	
	TextureU, TextureV	natočení textury na polygonu	x, y, z	
	Vertex	bod polygonu (x, y, z)	x, y, z	

Tab. 2. Atributy značek T3D souboru

c) rotace

Pojmy *pitch, yaw, roll* pochází z teorie letecké dynamiky a udávají rotaci postupně kolem os *y, z, x*.



Obr. 32. Znázornění významu veličin pitch, yaw, roll

d) sekvence zobrazení vrcholu

Po načtení celého actoru, který obsahuje aspoň jeden brush, je nutné všechny vrcholy posunout, zrotovat a jejich osy vynásobit podle jednotlivých atributů *Actor* (*MainScale*, *Location*, atd.).

```
FOR each vertex of each polygon of parsed brush DO:
   do MainScale ... x *= MainScale[x], y *= MainScale[y], z *= MainScale[z]
   do translation (-PrePivot[x], -PrePivot[y], -PrePivot[z])
   do rotation Yaw, Pitch, Roll
   do PostScale ... x *= PostScale[x], y *= PostScale[y], z *= PostScale[z]
   do TempScale ... x *= TempScale[x], y *= TempScale[y], z *= TempScale[z]
   do translation (Location[x], Location[y], Location[z])
ENDFOR
```

Obr. 33. Pseudokód sekvence zobrazení jednotlivých bodů Brush dle hodnot Actor

Hodnoty zvětšování

V hodnotách zvětšení se také objevují *SheerAxis=Sheer_OsaOsa*. Význam této hodnoty jsem nerozluštil, ale nezdá se to být důležité pro správné zobrazení těles z T3D souboru.

Hodnoty CsgOper

UnrealEd považuje prostor za pevný *(solid)*. A tedy veškeré první objekty umístěné do mapy mají hodnotu CSG operace CSG_Subtract *(rozdíl)*. Pro potřeby implementovaného algoritmu jsem považoval prostor za prázdný a všechny operace *subtract* a *add* jsou zaměněné za opačné.

e) polygony

Experimentálně bylo ověřeno, že veškeré polygony z T3D souboru jsou konvexní a normály rovin polygonů míří směrem ven z tělesa. S tímto předpokladem pak pracuje algoritmus popsaný níže. Nicméně tento předpoklad pro další verze UnrealEdu nemusí platit. Pracoval jsem s UnrealEd 2000.

Instalace a spuštění T3D aplikace

Na přiloženém CD se nachází instalační soubory pro MS Windows 2000 / XP, které jsou potřebné pro kompilaci a spuštění aplikací, a také vlastní práce – T3D aplikace a NavMesh bot. Aplikace byly vyvíjeny pro Javu 1.5.0_06 a prostředí MS Windows 2000/XP. Aplikace také vyžaduje podporu OpenGL, která je sice v systému Windows přítomna, ale pokud OpenGL není podporováno vaší grafickou kartou, tak zobrazení objektů bude pomalé. Aplikace nebyla v jiném prostředí testována. Instalační postup je popsán pro systém Windows XP.

Obsah CD:

java/jdk-1_5_0_06-windows-i586-p.exe	…	Sun Java JDK 1.5
jytohn/jython-21.class	…	Jython v2.1
java_extensions/	…	jars, dlls pro Javu
NavMesh/	…	T3D aplikace
Pogamut/	…	Pogamut s NavMesh botem
ut_maps/	…	ukázkové UT mapy

1. Nejprve si nainstalujte vývojové prostředí pro Javu (java/jdk-1_5_0_06-windows-i586-p.exe). Implicitně se instaluje do adresáře „C:\Program files\Java\jdk1.5.0_06“. V dalším textu budu používat tuto implicitní cestu pro popis přístupu ke složkám Javy. Pokud jste instalovali Javu do jiného adresáře, tak tuto cestu nahrazujte svou.

2. Po instalaci Javy doporučuji připsat do proměnné prostředí *path* cestu ke spustitelným souborům Javy. Ve Windows tuto proměnnou naleznete:
   Start / Nastavení / Ovládací panely / Systém / Upřesnit / Proměnné prostředí
   Editujte proměnnou *path*. Pokud jste Javu nainstalovali do implicitního adresáře, tak k její hodnotě připojte řetězec:
   „;c:\program files\java\jdk1.5.0_06\jre\bin“ (bez uvozovek).
   Následně je třeba restartovat operační systém, aby každá nová konzole byla spuštěna s novým nastavením cesty.
   Pokud nechcete měnit proměnné systému, tak doporučuji si otevřít konzoli a nastavit v ní cestu ručně pomocí příkazu:
   „path=%path%;c:\program files\java\jdk1.5.0_06\jre\bin“ (bez uvozovek)
   a všechnu další instalaci provádět z této konzole.

3. Po té nainstalujte Jython. V adresáři na CD „jython/“ zadejte příkaz „java jython-21“. Spustí se grafický instalátor, ve kterém zvolte operační systém Windows a nainstalujte jej např. do adresáře „c:\Jython“.

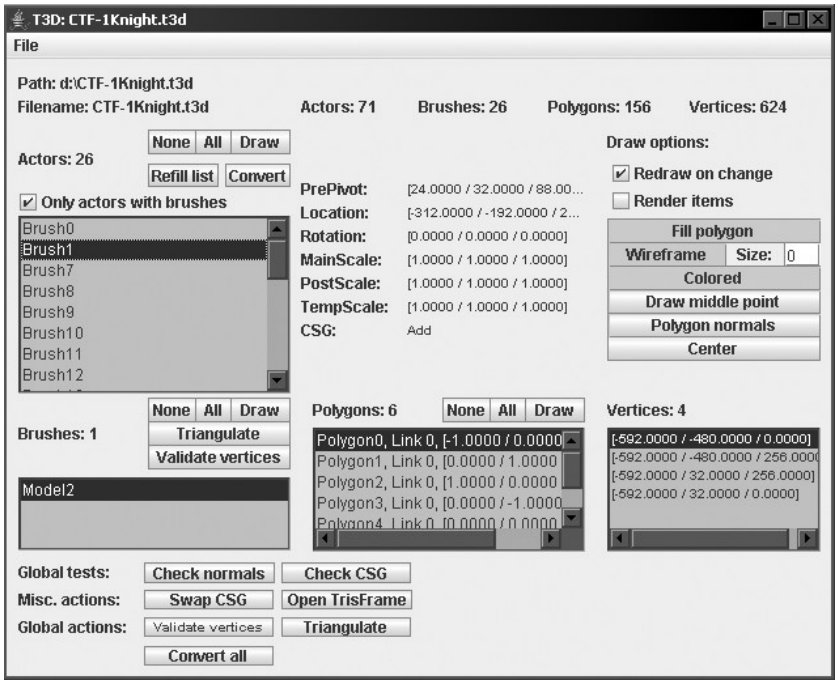
4. V adresáři, kam jste nainstalovali Jython („c:\Jython“), editujte soubor jython.bat. Poslední příkaz v něm změňte (zachovejte uvozovky a zrušte řádkování):
   "c:\program files\java\jdk1.5.0_06\jre\bin\java.exe" -Xmx256m "-Dpython
   .home=c:\jython" -classpath "c:\jython\jython.jar;%CLASSPATH%"
   org.python.util.jython %ARGS%
   Tím nastavíte, aby se Jython spustil ve správném prostředí Javy.

5. Nyní je třeba rozšířit prostředí Javy o jar a dll soubory. Do složky „c:\Program files\Java\jdk1.5.0_06\jre\lib\ext“ zkopírujte tyto soubory ze složky na CD „java_extensions/“:
   ABODEDebuggerParts.jar
   gl_utils.jar
   poly2tri.jar
   my_lwjgl_0.99.zip/lwjgl_source.jar
   xerces-2.8.0.zip/xerces-2.8.0/*.jar
   Do složky „c:\Program files\Java\jdk1.5.0_06\jre\bin“ zkopírujte tyto soubory ze složky na CD „java_extensions/“:
   my_lwjgl_0.99.zip/native_win32/*.dll
6. Zkopírujte na pevný disk z CD složky NavMesh a Pogamut. V adresáři „NavMesh/“ spusťte příkaz „compile.bat“. Pokud máte správně nastavenou cestu k Javě, tak tento příkaz zkompiluje T3D aplikaci a vytvoří soubor navmesh.jar. Tento soubor zkopírujte do adresáře „c:\program files\java\jdk1.5.0_06\jre\lib\ext“. T3D aplikaci spustíte příkazem z adresáře NavMesh „run.bat“.
7. Pogamut se spouší příkazem z adresáře Pogamut „pogamut.bat“. Pokud jste Jython nainstalovali do jiného adresáře než „c:\Jython“, tak změňte soubor „pogamut.bat“ a nastavte v něm správnou cestu do adresáře Jythonu.

Uživatelská dokumentace T3D aplikace

V příloze B naleznete postup, jak nainstalovat a spustit T3D aplikaci. Aplikace obsahuje celkem tři okna. Jsou to T3DFrame, OpenGL a TrisFrame.

T3DFrame slouží k náhrání T3D souboru exportovaného z UnrealEd a k práci s T3D objekty. TrisFrame pak slouží k práci s objekty, které byly převedeny do hranové reprezentace. Obsahuje také možnost jednotlivé objekty slučovat podle jejich CSG operace. OpenGL okno pak vykresluje jednotlivé zvolené objekty z T3DFrame nebo TrisFrame.



Obr. 34. T3DFrame

Popis jednotlivých komponent T3DFrame:

menu File/Load T3D
: Nahraje T3D soubor do aplikace.

tlačitka None, All, Draw
: Ovládají zobrazení prvků v OpenGL okně. Vztahují se vždy ke zvoleným položkám v listu pod nimi.

list Actors, tlačítko Refill list
: Naplní actors list původním obsahem.

list Actors, tlačítko Convert
: Konvertuje zvolené actors do hranové reprezentace.

list Brushes, tlačitko Triangulate
: Trianguluje polygony zvoleného brush.

list Brushes, tlačítko Validate vertices

Pro každý vrchol každého polygonu zvoleného brush otestuje, zda-li neleží v hraně jiného polygonu. Pokud ano, je tento bod přidán do polygonu, v jehož hraně vrchol leží.

Global tests, tlačítko Check normals

Otestuje, zda-li normály všech těles směřují ven z tělesa. Jedná se o heuristiku, která funguje pouze pro konvexní tělesa. Tělesa, která neprojdou testem jsou vypsány v Actors listu.

Global tests, tlačítko Check CSG

Zobrazí ty objekty, u kterých není definována CSG operace.

Misc. Actions, tlačítko Swap CSG

Mění typ CSG operace u všech actors z 'Add' na 'Subtract' a obráceně.

Misc. actions, tlačítko Open TrisFrame

Zobrazí TrisFrame.

Global actions, tlačítko Validate vertices

Provede validaci vrcholů ve všech brushes. Jedná se o hromadnou operaci případu 'list Brushes, tlačítko Validate vertices'.

Global actions, tlačítko Triangulate

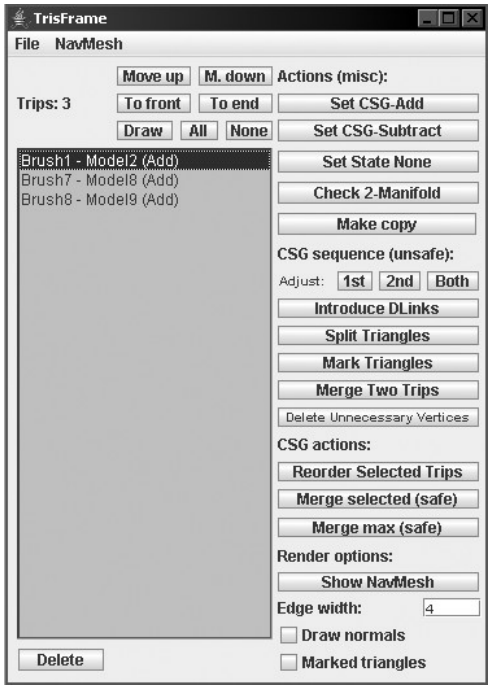
Provede triangulaci všech brushes. Jedná se o hromadnou operaci případu 'list Brushes, tlačitko Triangulate'.

Global actions, tlačítko Convert All

Konvertuje všechny brushes do hranové reprezentace a otevře TrisFrame. Jedná se o hromadnou operaci případu 'list Actors, tlačítko Convert'.

Draw options

Ovládání způsobu vykreslování T3D objektů.



Obr. 35. TrisFrame

Převedeným brushes do hranové reprezentace se v okně TrisFrame říká Trip. Je to zkratka pro Triangulated Polyhedra. Popis jednotlivých komponent TrisFrame:

Tlačítka Move Up, M. Down, To front, To end
: Přesouvají zvolené položky v trips listu.

Tlačítka All, None
: Označí resp. zruší označení všech položek trips listu.

Tlačítko Draw
: Překreslí scénu v OpenGL.

Tlačítko Delete
: Smaže zvolené actors.

Tlačítko Set CSG-Add
: Změní typ CSG operace zvolených actors na 'Add'.

Tlačítko Set CSG-Subtract
: Změní typ CSG operace zvolených actors na 'Subtract'.

Tlačítko Set State None
: Změní stav zvolených actors na None. Actors mění svůj stav vždy po provedení některé ze čtyř fází algoritmu.

Tlačítko Check 2-Manifold
: Testuje, zda-li jsou zvolené trips 2-manifold. Pokud dojde k chybě algoritmu v některých z jeho fází, tak je možné, že těleso přestane být 2-manifold.

Tlačítko Make copy
: Zkopíruje zvolené trips do nových instancí a připojí je na konec seznamu.

Tlačítka adjust 1st, 2nd, Both
: Provádí operace nad 2 zvolenými trips A a B.
1st opravuje vrcholy actor A vzhledem k actorovi B.
2nd opravuje vrcholy actor B vzhledem k actorovi A.
Both nahrazuje stisknutí tlačítek 1st a 2nd.
Operace, které spouští tyto tlačítka, se snaží opravit vrcholy trips tak, aby bylo možné trips sloučit.

První čtyři tlačítka CSG sekvence
: Odpovídají 1. až 4. fázi algoritmu. Lze pomocí nich sledovat, zda fáze proběhly v pořádku.

Tlačítko Delete Unnecessary Vertices
: Vyhledá a smaže ty vrcholy trips, které v nich nemusí být.

Tlačítko Reorder Selected Trips
: Pracuje nad zvolenými Trips. Snaží se vytvořit sekvence se zvolených trips tak, aby na sebe trips navazovaly a bylo možné je sloučit.

Tlačitko Merge selected (safe)
: Spustí algoritmus na zvolené trips. Slučuje trips shora dolů. Zastaví se při prvním neúspěchu slučování. Pokud dojde při běhu algoritmu k chybě, tak zvolené trips nejsou poškozené – algoritmus vždy pracuje nad kopiemi.

Tlačítko Merge max (safe)

Snaží se sloučit maximum trips. Při chybě se nezastaví, ale bere další trip ze zvolených trips a pokusí se ho sloučit s předchozím výsledkem. Taktéž pracuje nad kopiemi trips a případné chyby nepoškodí žádné z trips.

Tlačítko Show NavMesh

Zobrazí NavMesh ze zvoléného trip.

Menu File/Save selected

Uloží zvolené trips do souboru ve formě XML.

Menu File/Save All

Uloží všechny trips do souboru ve formě XML

Menu File/Load (loose precision)

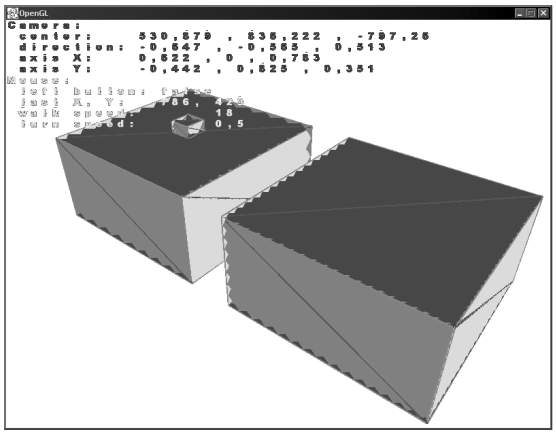
Nahraje trips ze souboru XML. Při ukládání dochází ke ztrátě přesnosti.

Menu NavMesh/Load NavMesh (show only)

Nahraje data z XML souboru, který obsahuje NavMesh a zobrazí jej.

Menu NavMesh/Save as NavMesh

Uloží zvolený trip do souboru jako XML. Pokud chcete, aby byl soubor nalezen a použit NavMesh botem, musí mít jméno stejné, jako je jméno mapy a příponu '.xml'. Daný soubor s NavMesh zkopírujte do složky 'Pogamut'.



Obr. 36. OpenGL okno

V okně OpenGL lze pohybovat kamerou pomocí klávesnice a otáčet s ní pomocí myši. Tím lze měnit pohled na zobrazovaná tělesa.

Ovládání myší

Levé tlačítko + pohyb myší … otáčení ve směru pohybu myši

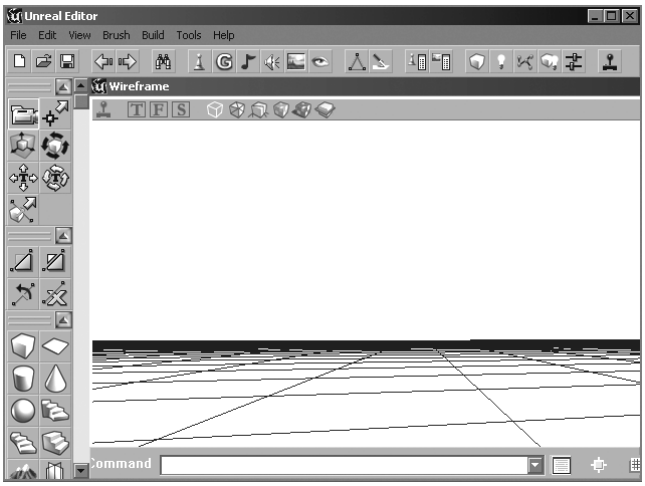
Kolečko … mění rychlost pohybu

Ovládání klávesnicí (qwerty rozložení kláves)

W	…	dopředu
S	…	dozadu
A	…	otočení doleva
D	…	otočení doprava
Z	…	úkrok doleva
C	…	úkrok doprava
Q	…	otočení nahoru
E	…	otočení dolů

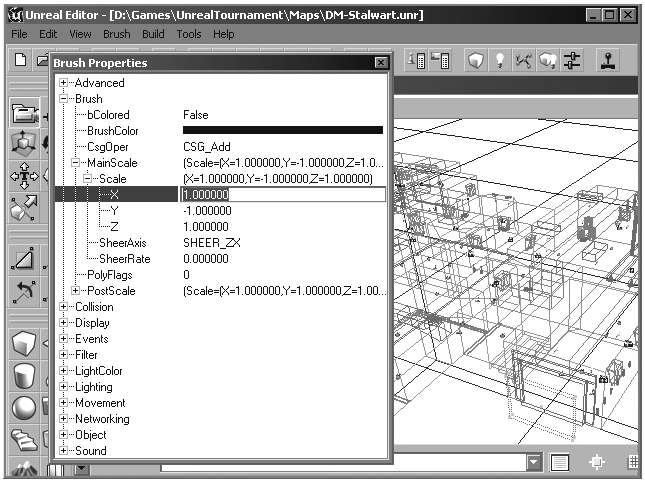
Práce v UnrealEd

UnrealEd se nachází v adresáři unreal_tournament_home/system/UnrealEd.exe. Po jeho spuštěním se zobrazí toto okno:



Obr. 37. UnrealEd

V menu File máte možnost nahrát do editoru libovolnou mapu (File/Open...) a také nahranou mapu exportovat (File/Export...) do T3D souboru. Po nahrání mapy se vám v Wireframe okně zobrazí všechny brushes, ze kterých se mapa skládá. Můžete kterýkoli objekt vybrat kliknutím na jeho hranu a posléze vyvolat pravým tlačítkem jeho vlastnosti z kontextového menu (Brush properties...).



Obr. 38. Vlastnosti brush v UnrealEd

Zde můžete změnou hodnoty Brush/Main scale/Scale/osa těleso zvětšit zadáním hodnoty vyšší než jedna. Tím se brush přestane s okolními tělesy dotýkat a začne je překrývat.

UnrealEd je komplikovaný nástroj na tvorbu map pro Unreal Tournament a popis všech jeho možností by byl zdlouhavý. Další informace jsou k dispozici na http://wiki.beyondunreal.com/wiki/Mapping_Lessons (6.8.2006).

Popis některých balíčků a tříd T3D aplikace

Externí balíčky

Používané externí balíčky jsou balíčky LWJGL (Lightweighted Java Gaming Library, pro OpenGL), Xerces (implementace SAX 2.0 API) a Poly2Tri (triangulace polygonů v 2D).

Jak T3D aplikace tak Pogamut využívají LWJGL pro vytvoření OpenGL kontextu a práce s ním. Stránky a aktuální verze LWJGL jsou k dispozici na http://lwjgl.org (6.8.2006). Pracoval jsem s verzí 0.99, ve které jsem musel upravit třídu Display přidáním metody releaseContext(), aby bylo možné s OpenGL pracovat v různých vláknech. Vytvořil jsem balíček lwjgl_source.jar, který spolu s .class soubory obsahuje i některé .java soubory.

Při náhrávání NavMesh a Trips jsou pak použity balíčky Xerces (resolver.jar, xercesImpl.jar, xercesSample.jar, xml-apis.jar), které implementují SAX 2.0 API pro zpracovávání XML souborů. Stránky a aktuální verze jsou k dispozici na http://xerces.apache.org/xerces-j/ (6.8.2006).

Balíček poly2Tri je používán vždy, když je třeba triangulizovat nějaký polygon. Jedná se zejména o fázi tři algoritmu na převod tělesa z CSG reprezentace na hranovou reprezentaci.

Balíček gemrot.glUtils

Obsahuje třídy, které používají LWJGL. Nejdůležitější je abstraktní třída AdvancedGL, v jejichž potomcích je nutné implementovat metody logicInner() a renderInner(). Metoda renderInner() je zodpovědna za vykreslení scény a metoda logicInner() je zamýšlena pro jakoukoli přípravu proměnných pro renderování. Třída AdvancedGL pak obsahuje metdou startGL(height, width), která inicializuje OpenGL okno o zadané velikosti. Tuto třídu AdvancedGL používá jak T3D aplikace, tak i NavMesh bot.

Balíček math

Balíček obsahuje definici tříd 2D a 3D geometrických primitiv, jako jsou trojice (Tuple3D), přímky, roviny, apod. Důležité jsou pak třídy v balíčku math.extended, které obsahují třídy, které definují hashCode() a equals() a mohou se vkládat do kolekcí HashSet a použít jako klíč v mapě HashMap. Hešování u např. Vertex3D probíhá zaokrouhlením souřadnic na určitý počet desetinných míst (podle konstanty math.M.EPSILON_DIGITS = 8) a zahešováním této hodnoty. Equals() pak s touto přesností porovnává souřadnice instancí. Podobně jsou pak hešovány i ostatní třídy v baličku.

Balíček gemrot.unreal.gui

Balíček obsahuje třídy potomků JFrame – T3DFrame, TrisFrame a ProgressWindow, potomka třídy AdvnacedlGL – NavMeshGL. T3DFrame i TrisFrame definuje různé metody pro manipulaci s objekty ve svých listech. Některé z těchto metod jsou public a lze je využít pro ovládání daného okna. T3DFrame je pak hlavní třída celé aplikace, která také definuje metodu main a může být tedy spuštěna. Pro další informace o metodách viz javadoc.

Balíček gemrot.unreal.navMesh

Balíček obsahuje třídy, pomocí nichž je konstruován NavMesh. Důležité jsou třídy LoadNavMesh, NavMesh a NavMeshTriangle.

Třída LoadNavMesh používá Xerces pro zpracování NavMesh uloženého v XML souboru a vytváří instanci třídy NavMesh. Tuto třídu používá také NavMesh bot.

Třída NavMesh zastřešuje celou navigační mřížku. Obsahuje metody pro vytváření nových trojúhelníků (getVertex(Tuple3D), getEdge(Tuple3D, Tuple3D), addTriangle(NavMeshTriangle)), ze kterých se skládá NavMesh a dále obsahuje metodu getTriangle(Tuple3D), která vrací nejbližší trojúhelník NavMeshe, který se nachází pod zadaným bodem (ve smyslu Z-souřadnice). NavMesh dělí prostor do hranolů, které mají šířku a délku rovnu NavMesh.PRISM_WIDTH a do kterých si ukládá trojúhelníky, které do daného hranolu zasahují. To umožňuje metodě getTriangle(Tuple3D) testovat pouze některé trojúhelníky NavMesh a tím výrazně urychlit proces hledání trojúhelníku.

Třída NavMeshTriangle je potomkem třídy math.extended.Triangle3D. Z hlediska pohybu botů je nejdůležitější třída getPath(Tuple3D point, NavMeshTriangle triangle). Metoda k bodu *point* hledá nejblížší bod hrany, který trojúhelník sdílí s *triangle*.

Balíček gemrot.unreal.T3D

Balíček obsahuje třídy, které reprezentují objekty z T3D souboru. Jednotlivé třídy sebe navzájem obsahují T3DActorList <- T3DActor <- T3DBrush <- T3DPolygon. Tyto třídy také obsahují metody pro triangulaci, úpravu vrcholů polygonů apod. – viz javadoc. Důležitá je třída T3D2TRIS, která převádí T3D objekty do hranové reprezentace.

Balíček gemrot.unreal.tris

Balíček obsahuje třídy pro hranovou reprezentaci 3D těles. Tato tělesa jsou nazvána Trip (Triangulated Polyhedra) a pracuje se především s třídami Trip2, Edge, TrisTriangle a TrisVertex3D. Balíček také obsahuje další balíčky, kterými jsou:

coplanarTrangulation – implementuje retriangulaci

CSG – vlastní algoritmus pro slučování trips

projection3DTo2D – používá se kvůli triangulaci, která je psaná pro 2D polygony, obsahuje třídy, které počítají 2D souřadnice koplanárních objektů v 3D

Balíček gemrot.unreal.tris.CSG

Balíček obsahuje algoritmus pro slučování trips podle zadané CSG operace. Vše je implementováno v třídě CSG, která obsahuje jednotlivé metody, pro každou fázi algoritmu. Vše pak zastřešuje metoda mergeSafe(Trip2, t1, Trip2 t2, int csgOperation), která slučuje trips dohromady.

Při slučování se pro každý trojúhelník, který má být rozdělený DLinks, používá instance třídy TriangleArea, do které se jednotlivé DLinks zapisují. TriangleArea pak identifikuje jednotlivé polygony, na které je dělena a vrací jejich triangulaci.

Balíček gemrot.unreal.utils

Balíček obsahuje pomocné třídy například pro hešování (HashCodes).