ČESKÉ VYSOKÉ UČENÍ TECHNICKÉ

Fakulta Elektrotechniky

DIPLOMOVÁ PRÁCE

3D model bytu pro vizualizaci / simulaci a ověřování řídicích systémů

2011 Bc. Antonín Pošusta

České vysoké učení technické v Praze
Fakulta elektrotechnická

Katedra kybernetiky

ZADÁNÍ DIPLOMOVÉ PRÁCE

Student: Bc. Antonín P o š u s t a

Studijní program: Elektrotechnika a informatika (magisterský), strukturovaný

Obor: Biomedicínské inženýrství

Název tématu: 3D model bytu pro vizualizaci / simulaci a ověřování řídicích systémů

Pokyny pro vypracování:

1. Seznamte se s již existujícími systémy pro 3D zobrazení modelů bytů/domácností určených pro vizualizaci z pohledu domácích inteligentních řídicích systémů (obsluhy domácího prostředí).
2. Zhodnoťte jejich výhody a nevýhody zejména z pohledu přehlednosti, názornosti a vypovídací hodnoty nejen pro uživatele, ale současně i například pro servisního/testovacího technika.
3. Sestavte seznam mnoha běžných domácích zařízení/spotřebičů, jenž často bývají (mohou být) řízeny z (pomocí) inteligentního domácího systému. U každého zařízení rovněž uveďte seznam možných akcí/úkonů (nejen činnost vlastního zařízení, ale rovněž jeho obsluhy).
4. Navrhněte možnost snadného vytvoření vhodného 3D modelu bytu/domácnosti pouze pomocí několika konfiguračních souborů obsahujících nejen stručný plán bytu, tak rovněž rozmístění jednotlivých typů zařízení.
5. Vytvořte programový modul/knihovnu, jenž z dodaných konfiguračních souborů sestaví požadovaný 3D model bytu/domácnosti a současně bude poskytovat vhodný interface pro řízení v něm umístěných zařízení/spotřebičů.
6. Pro ladící účely přidejte schopnost ukládání všech vykonaných (zaslaných) akcí a jejich pozdější opětovné přehrávání z tohoto záznamu (popis akce, časový údaj a další).

Seznam odborné literatury:

[1] Internetové stránky prodejců/výrobců domácích řídicích systémů.
[2] Knihy/manuály Microsoft .NET Framevork, jazyk C# (2005, 2008, 2010) a jeho komponenty.

Další literaturu dodá vedoucí práce.

Vedoucí diplomové práce: Ing. Petr Novák

Platnost zadání: do konce letního semestru 2011/2012

prof. Ing. Vladimír Mařík, DrSc.
vedoucí katedry

prof. Ing. Boris Šimák, CSc.
děkan

V Praze dne 2. 2. 2011

Prohlášení

Prohlašuji, že jsem svou diplomovou práci vypracoval samostatně a použil jsem pouze podklady (literaturu, projekty, SW atd.) uvedené v přiloženém seznamu literatury.

V Praze dne **11.5.2011**

.....

podpis

Poděkování

Rád bych tímto poděkoval vedoucímu své diplomové práce, Ing. Petru Novákovi za vstřícný přístup v průběhu jejího vzniku.

Abstrakt

Tato práce se zabývá implementací programové komponenty pro vizualizaci nejen inteligentních domácností / budov, ale rovněž například pro simulaci a zobrazování pohybu osob v budovách. Zmíněná komponenta řeší problematiku snadného a srozumitelného zobrazení dat uživatelům (v případě inteligentních domácností i snadného nastavování).

Komponenta je určena pro zobrazování 3D modelů bytů / vícepodlažních domů ze snadno editovatelných bitmapových obrázků, označování objektů a dále například zobrazování pohybu / polohy osob (z možných pohybových čidel nebo při simulacích). Součástí komponenty je i „logger“, umožňující opakované přehrání situací a událostí během simulace/ běhu aplikace.

Za tímto účelem tedy byla vytvořena universální zobrazovací komponenta doplněná editorem, umožňujícím snadné rozšíření o další objekty.

Abstract

This work is concerned with implementation of software component for rendering 3D models of intelligent homes and simulations (of person movement etc.). This component solves problem of easy and readable displaying of data to users.

The component is rendering 3D models of flats/ houses from easily editable bitmap files and can also render movement of persons. The important part of this component is „logger“, that is able to record situations and events that happened during the simulation / run of application.

For that purpose was created the universal component with editor, that is capable of extending by adding new objects.

Obsah

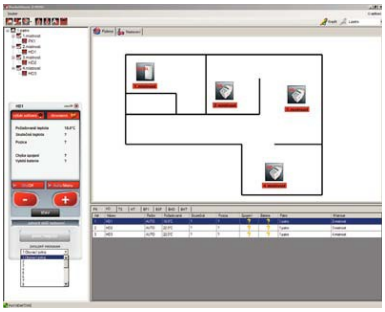
1 Úvod

1.1 Co je inteligentní domácnost

Tímto pojmem se bude následující text zabývat, jelikož to byl primání cíl vytvářené komponenty. Pojem inteligentní domácnost se stal fenoménem poslední doby. Co si pod tímto pojmem představit? Jedná se domácnost, která je řízená centrální jednotkou, kterou může být jednoduchá PLC logika, FPGA nebo dokonce přímo PC server (což je v dnešní době na vzestupu). Inteligentní domácnost může kontrolovat vlhkost vzduchu v domě, teplotu, obsahovat bezpečnostní systém, umožnit libovolné přiřazení vypínačů v domě libovolnému světlu, ztlumení světel při zapnutí televize a mnoho dalšího. Často si pod pojmem inteligentní domácnost představíme své domácí okolí, které je schopno do jisté míry interagovat s uživatelem a tím mu vhodně zpříjemnit pobyt v tomto prostředí. Například když je zima, zapne topení, když je velmi teplo, otevře okno a další akce, nejspíše vykonané uživatelem. A toto vše současně podle osob, vyskyujících se právě v domácnosti.

1.1.1 Nynější stav

Mezi firmy zabývající se těmito technologiemi patří například firma Loxone [6]. Tato firma ke svým produktům dodává software, s jehož pomocí je možné byt nastavit / nakonfigurovat, avšak tento proces pro laika již nemusí být tak jednoduchý (jedná se o nastavení podobné ovládacím panelům ve Windows). Uvedená firma se zabývá ovládáním ventilací, žaluzií, světel, audiosystémů a televizí. Jejich systém lze ovládat i aplikací určenou pro Iphone. Další českou firmou, zabývající se touto problematikou je Elektrobock [7]. Ta se spíše specializuje na inteligentní vytápění a osvětlení domu. Vyvíjí systém Pocket-Home obsahující základní grafické rozhraní s nákresem půdorysu bytu (obrázek 1) s ikonkami aktivních spotřebičů. Tento systém může být pro laika o něco přehlednější (zpravidla topných jednotek a světel). Navíc obsahuje i možnosti plánování akcí na týden dopředu, tento software již není dodávaný zdarma, jeho cena je přibližně 3000 korun.



Obrázek 1 - Pocket-Home

Jinou firma zbývající se touto problematikou komplexněji je AMX [8], která společně s českým partnerem Insight Home nabízí řešení inteligentních domácností jako celku, včetně vlastního software pro ovládání. Jejich systém je založen na spojení s PC serverem. Jejich ovladače se podobají PDA a umožňují ovládat prakticky vše.

Poslední firma, jež bude zmíněna v tomto textu je ABB. Jedná se o velkou

nadnárodní korporaci, jež se zaobírá automatizací a jejím projektováním obecně a tudíž inteligentní domácnosti nejsou jejím hlavním záměrem.

Průzkum internetu ukázal, že zatím žádná z firem 3D zobrazení pro inteligentní domácnosti nevyužívá.

Tato komponenta by měla sloužit pro vizualizaci a snadnější ovládání, nastavování nebo simulaci inteligentní domácnosti. Dále by ji mělo být možné využít například jako dohledový systém pro pohyb osob pod anestezií v nemocnici.

1.1.2 Intetligentní systémy spotřebiče v domácnosti

Dnes už je téměř v každém zařízení kolem nás přítomen i sebemenší mikroprocesor, avšak spotřebiče přímo řízené PC lze základně zmapovat. Mezi nejzákladnější systémy, ovládané v domě patří zřejmě vytápění domu – klimatizační systémy. Regulují se zpravidla teploty a vlhkosti v místnostech bytu nebo domu. Obvyklé nastavení bývá, že v obdobích kdy je očekávána přítomnost obyvatel domu je teplota vyšší (respektive nížší v případě klimatizace v létě). Dalšími ovládanými systémy v inteligentních domech jsou rolety u oken, ventilace, televize a audiosystémy (firma Loxone) a samozřejmě zámky u dveří nebo garážová vrata, veškeré osvětlení a lampy. V domech upravených pro handicapované osoby lze mezi komponenty řadit i ovládání výtahu, popřípadě otevírání dveří a oken.

1.2 Použité technologie

1.2.1 C# 3 a .NET v3.5

Implementována byla proveden v jazyce C#, což je vlastně vysokoúrovňový objektově orientovaný programovací jazyk vyvinutý firmou Microsoft zároveň s platformou .NET Framework, schválený standardizačními komisemi ISO a ECMA. C# je založen na jazyku C++ a rovněž čerpá ze syntaxe jazyku Java. Lze jej využít k tvorbě všech druhů běžných aplikací – databázových programů, formulářových aplikací ve Windows, webových aplikací a služeb, ale také softwaru pro mobilní zařízení (PDA a mobilní telefony).

Historie jazyka C# začíná rokem 2002, kdy byla zveřejněna první verze společně s první verzí .NET Frameworku. Obsahovala základní podporu objektového programování, ve které vycházela z jazyka C++ a zkušeností z jazyku Java.

Mezi první verzí a verzí 3, v níž je celá práce implementována uplynulo 5 let vývoje tohoto nového jazyka. Verze 3 na rozdíl od předešlých obsahuje spoustu nových vlastností – nativní podporu generických tříd, vycházející z podpory na úrovni CLI, dále částečné třídy (možnost rozděli třídu na více souborů), iterátory, anonymní metody pro pohodlnější užívání delegátů (to jsou bezpečné odkazy na metody), null typy a operátor

koalescence.

Mezi další novinky patřily lambda výrazy, tedy jednodušší metoda pro zápis anonymní metody, inicializátory objektů a kolekcí, Rozšiřující metody, Anonymní třídy umožňující např. rychlé vytvoření objektů přenášejících informace vyžádané z databáze přes LINQ, Klíčové slovo var, nutná to podmínka pro využití anonymních tříd a Výrazové stromy (expression trees) umožňující za jistých podmínek kompilátoru místo vyhodnocení výrazu vytvoření jeho objektové reprezentace.

Nevýhodou tohoto prostředí je použitelnost (v současné) výhradně na systémech s operačním systémem Windows. Ikdyž operační systém Windows je v evropském prostředí právě ten nejrozšířenější. Existují dokonce i určité snahy o implementaci podpory .NET Frameworku i na operačním systému Linux – projekt MONO.

Nyní je již sice k dispozici C# 4 společně s .NET 4.0, jež byly zpřístupněny společně s vývojovým IDE Visual Studio 2010. Vzhledem však k započetí implementace na platformě 3.5 byla práce na této platformě i dokončena. Implementace byla po dokončení převedena do verze 2010, aby byla schonpna činnosti i na 64bitových systémech (což bylo i odzkoušeno).

1.2.2 DirectX

Nejzákladnější technologií, použitá v této práci je DirectX. Jedná se o jednu z technologií pro zobrazování 3D vektorových dat. Nejvíce se s ní pravděpodobně lze setkat v počítačových hrách. Její výhoda spočívá v podpoře většiny současných grafických karet. Veškeré grafické operace se zpracovávají převážně na grafické kartě, což přináší velkou výhodu – není zatížen procesor a lze jej využít pro jiné výpočty. DirectX byl uvolněn v roce 1995 s nástupem Windows 95, avšak v povědomí širší veřejnosti začal být až s veremi 3 a 5. Verze 5 měla HW podporu 3D akcelerace grafiky, následkem čehož začala být ve velkém měřítku užívaná zejména v počítačových hrách. DirectX vždy nabízel velmi dobrou implementaci grafického 3D prostředí pro zobrazování v reálném čase své doby, ale svého open-sourcového soupeře OpenGL podle názoru většiny odborníků překonal až s verzí 9. V této verzi totiž byly odstraněny některé zastaralé a těžkopádné procedurální metody implementace.

Aktuální verze DirectX pracují spolehlivě na všech počítačích s operačním systémem Windows. Pokud se nepoužívají speciální stínovací nebo zjemňovací efekty je možnost využití takzvaného „softwarového renderingu“ a není tedy potřeba v PC mít 3D grafickou kartu (v dnešní době to je ale téměř samozřejmostí). Knihovny DirectX navíc umožňují na starších grafických kartách nedostupné hardwarem podporované nové technologie softwarově nahradit a dopočítat přes CPU. Pokud tedy je v PC 3D grafická karta, využívá se hardwarového vykreslování, což uvolní mnoho prostředků výpočetního času procesoru.

Pro správný běh této komponenty je doporučena instalace nejnovějších ovladačů

obsahujících DirectX verze 9.0c (doporučen SDK balík DirectX9-10-11). Komponentu bylo možné implementovat i pro novější verze DirectX, ale ukázalo se, že DirectX 9.0c je zpětně kompatibilní a je vysoká pravděpodobnost jeho podpory na většině dnešních počítačů.

Cíle vytvářené komponenty

Mezi cíle mé práce patří seznámení se s existujícími systémy pro 3D zobrazení modelů bytů / domácností určených pro vizualizaci z pohledu domácích inteligentních řídicích systémů a obsluhy domácího prostředí. Zhodnotit jejich výhody a nevýhody z pohledu přehlednosti a vypovídací hodnoty.

Na vytvářenou komponentu jsou tedy kladeny tyto požadavky:

- Možnost vytvořit model ze snadno editovatelného univerzálního formátu, pomocí několika konfiguračních souborů
- Vytvořeno formou knihovny / komponenty pro využití v libovolném větším (komplexnějším) systému
- Zobrazení a možnost interakce s modely domácích spotřebičů
- Možnost konfigurace vlastního zobrazení (otočení, náhled, …)
- Možnost vkládání modelů spotřebičů a některých dalších typů výbavy domu
- Možnosti nastavení grafického zobrazení (zobrazením pouze jednoho patra...)
- Aktivace / deaktivace zobrazení některých prvků.
- Možnost ukádání (logování) událostí nastalých v komponentě
- Ukázkovou aplikaci, například s několika modely domácích spotřebičů

Výstupem by tedy měla být programová knihovna (neboli komponenta) pro pozdější využití v libovolném komplexnějším systému. K této knihovně by měla být rovněž naimplementována jednoduchá vzorová aplikace pro její demonstraci a testování / využití. Pro testování a základní simulace byla tedy implementována jednoduchá aplikace, obsluhující komponentu.

V práci se nevěnuji vlastní logice řízení nebo komunikaci s koncovými zařízeními v domácnosti. Komponenta slouží pouze jako zobrazovací element, schopný ukládat a logovat nastalé události během provozu. Práce se rovněž nezaobírá rozmanitou tvorbou všemožných 3D modelů, nýbrž obsahuje pouze několik základních pro simulace ukázkovým programem a pochopení jejich tvorby (uživatelem komponenty). Pro pozdější účely obsahuje editor, jenž umožňuje tyto modely kdykoli později podle potřeby přidat.

2 Návrh řešení

2.1 Rozbor

Prvotním cílem této komponenty bylo její užití pro dohled nad stavem inteligentní domácnosti, její kontrolu a simulace takového prostředí. Během vývoje se však ukázalo, že by bylo možné ji užít i pro jiné účely simulace – například pohybu pacientů omámených sedativy v nemocnicích. Proto byla komponenta rozšířena i o funkce pro rozpoznání nárazu osoby do zdi nebo jiných předmětů přítomných ve zobrazení a rozšířené možnosti ovládání těchto virtuálních osob – dále v textu jako „herci".

2.1.1 Přístroje v inteligentní domácnosti

Průzkum internetu, především stránek firem nabízejících různá řešení v inteligentních domácnostech, umožnil sestavit určitý seznam přístrojů v domácnosti a ovládatelných přes počítač.

Prvky inteligentní domácnosti

Světla

Mezi hlavní vlastnosti inteligentní domácnosti patří především flexibilní vypínače a regulátory světel. Jedná se o zapojení světel do systému ovládaného mikropočítačem. Tento mikropočítač je pak možné řídícím serverem nastavit a tím přiřadit dané vypínače a regulátory daným světlům.

Komponenta by tedy měla být schopná zobrazit rozvícená světla nebo lampičky v místnostech, popřípadě mít i možnost jas těchto světel regulovat.

Vytápění / Klimatizace

Vytápění představuje nedílnou součástí každé domácnosti. V inteligentních domácnostech je navíc řízeno sofistikovanými metodami pro zajištění maximální efektivity vytápění při nejmenší spotřebě energií (respektive surovin – plyn). Například by tedy mohlo být možné v každé místnosti nastavit různou teplotu, příjemnou konkrétnímu obyvateli, nebo například vhodné řízení teplot v domě tak, aby se netopilo době, kdy v něm nikdo není.

Samozřejmě zobrazovací komponenta by neměla plnit funkci logiky řízení tohoto systému (předání nastavení už by obsluhoval SW samotného výrobce systému), ale pouze přehledný vstup / výstup pro uživatele. Ten by si mohl snadno nastavit například teplotu v dané místnosti kliknutím na radiátor, apod.

Dále to může být například ventilace / klimatizace, kontrola vlhkosti v domě a

podobně. Tyto systémy by bylo možné shrnout do této kategorie.

TV / Audiosystém

Jak již bylo zmíněno v úvodu, systémy některých firem umožňují i ovládat ozvučení domu (HiFi soustava) a televize. Bylo by vhodné pokud by komponenta uměla s těmito systémy rovněž interagovat (v rámci simulace).

Například by uživatel posouváním reproduktorů po místnosti mohl nastavit vyvážení prostorového zvuku nebo kliknutím na model TV televizi vypnout.

Kuchyňské spotřebiče

Kuchyňské spotřebiče ovládány v inteligentních domácnostech dnešní doby zatím nejsou, ale komponenta by mohla například upozornit na zapomenutý zapnutý sporák nebo troubu na pečení.

Komponenta by reagovala na hrozící nebezpečí umístěním ikonky vykřičníku do místa problému.

Bezpečnostní systém

Bezpečnostní systémy jsou především dvojího druhu – k ochraně života a k ochraně majetku. Systémy pro ochranu života jsou míněny především protipožární hlásiče, popřípadě jističe v případě přetížení některého elektrického okruhu. Systémy k ochraně majetku jsou především pohybová čidla v mísnostech a kontaktní čidla v oknech.

Komponenta by v případě narušení ochrany mohla logovat pohyb nežádoucího hosta po domě, popřípadě ukázat varování o nefunkčnosti nebo poruše spotřebiče.

Kontrola oken a dveří

Jedná se kontrolu zavřenosti všech oken nebo dveří, či elektronický vrátný. V případě bytu upraveného pro handicapovanou osobu by mohlo být možné i otevírat okna vzdáleně.

Bylo by vhodné aby komponenta dokázala zobrazovat otevírající se okna a dveře.

2.1.2 Užití jako asistivní technologie

Komponentu by mělo být možné využít pro dozor nad nemohoucími osobami nebo seniory, tento dozor potřebující. Bude obsahovat možnost zobrazení virtuálních osob nebo například vozíků.

2.1.3 Užití pro simulace

Komponentu bude možné užít jako zobrazovač libovolného simulátoru a pro tyto účely bude obsahovat funkce pro zobrazení trasy v prostoru, ukládání stavu komponenty a změn v ní nastalých v čase a možnosti pro přehrání takovýchto záznamů. Dále bude schopna registrovat interakci virtuálních osob z předměty – například náraz do objektu nebo do zdi.

Další významnou vlastností komponenty budou definovatelné oblasti zájmu – zóny, ty by samozřejmě mimo využití pro simulace našly využití i v nastavování inteligentních domácností – například by si obyvatel domácnosti mohl nastavit určité místo, ve kterém když bude, rozsvítí se například světlo apod.

2.2 Návrh řešení programové knihovny

Komponenta se bude skládat ze 3 prvků ovládaných aplikací – vlastního zobrazení, loggeru a editoru. Logger bude prvek ovládající zobrazení a zároveň ukládající veškeré operace provedené s komponentou. Ke knihovně bude současně vytvořena vzorová aplikace sloužící pro ukázku možností a způsobu užití. Editor bude určen k úpravě hlavního konfiguračního souboru.

2.2.1 Zobrazení

Komponenta bude zobrazovat 3D modely pater ležících nad sebou. Zobrazené zdi budou říznuté v polovině, aby bylo možné do bytu vidět. V patrech budou zobrazeny modely jednotlivých předmětů. Tyto modely budou mít libovolné stavy – akční objekty, nebo budou mít pouze dekorativní charakter – statické objekty. Objekty bude možno ovládat programově přes interface komponenty a také klikáním myši do prostoru jejich zobrazení (pro uživatelské pohodlí).

Dalším důležitým prvkem zobrazovaným komponentou jsou modely osob - „herci“, rovněž ovládaných programově. Komponenta bude schopna rozeznat náraz herce do stěny nebo objektu, což by mohlo být vhodně využito v některých simulacích. Modelu herce bude možno nastavit libovolnou barvu pro lepší přehlednost. Herci mohou být více druhů – reprezentováni různými modely a typem - vozík nebo člověk.

Bude možno definovat zóny v bytě / domu. Pokud do některé z těchto zón herec vstoupí, bude na to reagovat navrácením události. Tímto způbem lze nakonfigurovat zóny zájmu jako například „obývák“ nebo „podkroví“.

Každému objektu (předmět, herec, dveře,..) lze přiřadit své unikátní jméno dle kterého jej bude možné nastavovat.

Ovládání pohledů

Pro uživatelské pohodlí bude komponenta obsahovat funkce, které voláním s parametrem názvu objektu v zobrazení automaticky a animovaně nastaví kameru na tento objekt nebo ji posunou na žádanou vzdálenost a úhel. Animovaný pohyb kamery je důležitý pro lepší prostorovou představivost – ze statického obrazu s okamžitým skokem by uživatel mohl ztratit orientaci v prostoru.

„Kameru“ bude rovněž možné nastavovat manuálně pomocí pohybu myši a kláves. Bude poskytovat několik režimů – volná kamera, fixace v patře (mezi patry se bude možné přesouvat například kolečkem myši) a náhledová kamera (aby byl ve zobrazení vidět celý model domu).

V pohledu bude režim „průhledných stěn“. Tento režim bude užitečný, když uživatel bude chtít mít přehled nad celým domem. Objekty v tomto režimu budou neprůhledné.

Konfigurace

Zobrazení bude konfigurováno jedním souborem s nastaveními. V tomto souboru budou uloženy umístění modelů na disku, jejich nastavení, názvy a barvy podle nichž budou přiřazeny plochám.

Model bytu / domu / ústavu, včetně vybavení, bude vytvořen z nekomprimovaného obrázku (BMP, GIF, PNG) s půdorysem patra. Jednotlivé části stavby a objekty se rozlišují barvami. Pro modely zdí, oken a dveří sou rezervovány barvy odstínů šedi a černá. Bílá barva bude brána jako prázdné místo.

Jednotlivým objektům – předmětům dekorace bude možné přiřadit libovolné barvy (v editoru). Objekty budou těmito barvami reprezentovány v obrázku s půdorysem patra. V obrazovém souboru s půdorysem domu budou tyto objekty reprezentovány plnými obdélníky odpovídající barvy. Komponenta modely těchto předmětů naškáluje v prostoru zobrazení na přesnou velikost odpovídající obdélníkům. Orientace těchto předmětů se určí bílou tečkou u kraje obdélníku v půdorysu. Předmětům budou přiřazeny automaticky názvy tvořené názvy typu objektu a pořadím jejich výskytu. (Číslováno shora-dolů, zleva-doprava obrázku)

Akčním předmětům (TV, rádio, větrák,...) budou rovněž přiřazeny barvy, ovšem nebudou v půdorysu reprezentovány obdélníky, ale pouze symboly určujícími jejich orientaci. Tyto předměty budou mít svou určenou velikost modelem, popřípadě je bude možno přeškálovat uživatelsky. Tento způsob byl navržen také proto, aby bylo možné je v půdorysu umístit například na jiný statický objekt dekorace – stůl, skříň nebo třeba noční stolek.

Aby bylo možné přiřadit různé textury podlah a také ponechat místa bez podlah – například kvůli výtahu, bude pro každé patro ještě druhý konfigurační obrázkový soubor.

V tomto souboru budou jednotlivé barevné plochy odpovídat texturám umístěným na podlahy nebo prázdným místům bez podlah (pro prázdná místa to bude opět bílá).

Editor

V komponentě bude implementován editor k vytvoření a úpravě hlavního konfiguračního souboru. Editor umožní barvy přiřadit modelům, nastavit názvy typů předmětů, konfiguraci těchto předmětů a načíst do konfiguračního souboru obrázkové soubory textur v libovolném obrazovém formátu (JPG, PNG, GIF, BMP). Rovněž sloužicí k nastavení umístění souborů s půdorysy jednotlivých pater domu.

2.2 Logger

Logger bude možné aktivovat nebo nechat deaktivovaný. V případě jeho aktivace bude ukládat veškeré druhy operací prováděné s komponentou, včetně uživatelských editací (přesunutí nábytku, přidání nového předmětu) a operací s herci (změna poloh, vytvoření, zánik).

Logger bude mít také možnost navrácení komponenty do libovolného stavu v čase a záznam libovolnou rychlostí přehrát normálně nebo pozpátku. Záznam bude možné uložit do souboru a ze souboru opětovně načíst.

2.3 Vzorová aplikace

Aplikace bude sloužit jako testovací prostředí komponenty. Bude demonstrovat především možnosti ovládání jednotlivých prvků, přidávání a ovládaní herců, různých režimů ovládání kamery a testování událostí – nárazu herce do zdi nebo objektu, kliknutí na předmět, apod.

Pro demonstraci možného užití bude mít aplikace implementovanou také primitivní logiku automatického ovládání herců. Ta bude spočívat v posouvání herce vpřed pod náhodným úhlem a v případě nárazu do stěny zvolí úhel jiný.

V aplikaci bude rovněž možné spustit režim uživatelské editace a manipulovat s loggerem. Dále bude možné spustit editor, upravovat konfigurační soubor a tedy přidávat libovolné nové modely.

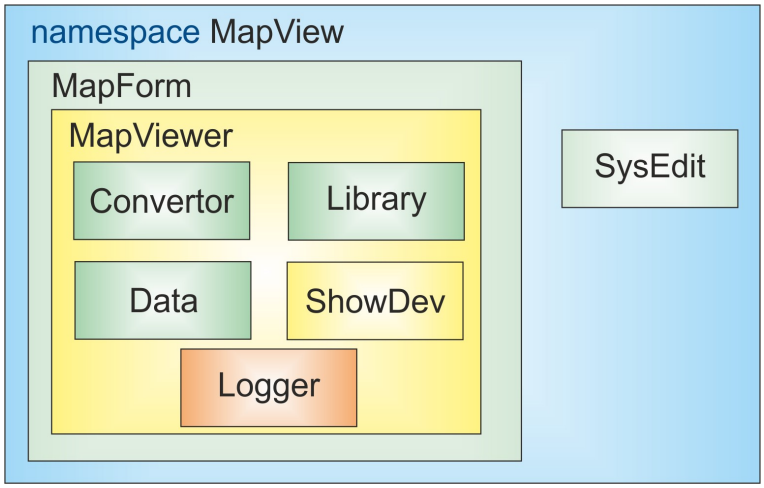
3 Popis řešení

3.1 Řešení komponenty

V této sekci bude popsáno vnitřní uspořádání grafické komponenty MapView.dll (název vytvářené komponenty).

3.1.1 Logické uspořádání

Následující diagram ulehčí představu o vnitřním logickém uspořádání jednotlivých částí vytvořené komponenty. Jednotlivé dílčí součásti budou dále v textu podrobně rozepsány a principy vysvětleny. V další kapitole textu jsou vybrané význačné a klíčové úseky kódu názorně vysvětlující problematiku vykreslování a funkce komponenty jako celku.



Obrázek 2 – vnitřní uspořádání komponenty

Pro volání „zvenčí“ (jako interface s programem) byly implementovány 4 třídy – **MapForm, MapViewer, SysEdit a Logger**.

MapForm zapouzdřuje **MapViewer** pro snadnější práci s komponentou. Jedná se o využívání grafického rozhraní jakožto samostatného okna. Objekt třídy **MapForm** automaticky inicializuje **MapViewer,** roztažený do velikosti okna a automaticky načte všechny data a objekty do operační paměti ze souboru. Rovněž zajišťuje automatické změny velikosti, při úpravách velikosti okna.

MapViewer je poděděný od **System.Windows.Forms.Panel**, jedná se o objekt, sloužící jako interface s komponentou, obsahuje logiku pro automatické roztažení obrazu (reinicializaci některých vlastností GPU zobrazení) do celé plochy panelu je roztažen a přímo ovládán objekt **ShowDev**. Objekt **ShowDev** slouží pro obsluhu samotné **Direct3D.Device**, jde tedy o plochu do níž se vykresluje vlastní 3D vizualizace. Dalšími

objekty obsaženými v **MapViewer** jsou **Logger**, **Library** a **Data**.

Objekt **Logger**, jak už název napovídá slouží k ukládání historie a událostí, jež nastaly v komponentě (bude probrán níže). Objekt **Library**, uchovává veškeré informace o grafických objektech, nastaveních pro komponentu a textury, je volán při inicializaci. Grafické objekty z **Library** jsou nahrány (metodami **MapViewer**u za užití **Coverter**u) po inicializaci do objektu **Data**, kde se již nachází samotný 3D model bytu s jednotlivými grafickými objekty a okamžitými nastaveními.

Posledním objektem patřícím do **MapViewer**u je **Convertor**. Tato třída obsahuje veškeré metody pro vytvoření modelů z bitmap a metody pro správné umístění objektů do prostoru. Od třídy **MapViewer** byla oddělena z důvodu přehlednosti a její aktuální rozsáhlosti. Obsahuje veškeré metody pro konstrukci prostorového obrazu z bitmap.

Pro editaci slouží objekt třídy **SysEdit**. Ten ve své podstatě představuje dialogové okno, obsahující v sobě další dialogová okna, umožňující editovat nastavení komponenty (objekt třídy Library) a ukládat jej do souboru. Z tohoto souboru si pak komponenta načítá nastavení a odkazy na umístění souborů s modely. Editor rovněž umožňuje komponentu testovat a graficky editovat po vytvoření tohoto kofiguračního souboru.

3.1.2 Třída Library

Tato třída zastává základní úložiště pro nastavení celé komponenty (nejedná se o nastavení užívaného zobrazení koncovým uživatelem, v případě inteligentních domácností by se o tato nastavení staral technik). Objekt této třídy je součástí třídy **MapView**. Obsahuje informace a nastavení všech typů grafických objektů, jejich výchozí názvy, barvy (podle nichž jsou rozpoznány z bitových map, bude probráno níže), dimenze, umístění a veškeré názvy souborů (map pater, modelů objektů, apod.) ze kterých jsou modely třídou **MapViewer** a **Convertor** načteny do **Data**. Veškeré uložitelné objekty v této třídě jsou označeny jako serializovatelné, a třída se také pomocí vlastní metody dokáže uložit do souboru a poté zas opětovně načíst.

Každý grafický objekt je reprezentován dvěma slovníky (Dictionary<Key, Value>). V jednom slovníku jsou objekty rozděleny podle jejich názvů, a v druhém podle barev. Třída **Convertor** tedy pouhým voláním metody **GetObjectByColor(Color)** získá klon objektu, umístitelný do prostoru. Objekt musí být klonován, jelikož v případě předání pouhé reference by při změně parametru došlo ke změně všech objektů daného typu. Aby toto klonování nebylo příliš paměťově náročné jsou klonovány pouze informace o transformacích a samotný model je v paměti pouze jednou (více viz. sekce grafické objekty). V knihovně je ukládán do souboru nastavení pouze jako název souboru obsahující model.

Důvodem proč nejsou v této třídě uloženy přímo samotné modely je neserializovatelnost jakéhokoliv objektu Direct3DX (jako je například mesh, jež je užitá pro uchování geometrie a normál). Navíc při vytváření objektů Direct3D je vždy nutné

mít inicializovanou samotnou Direct3D.Device (ta se po ukončení aplikace ukončí). Bylo by tedy nutné vyvinout vlastní způsob ukládání modelů do souboru, což je ale díky schopnosti objektů třídy Mesh – načtení ze souboru psaného skriptem DirectX zbytečné.

Dalším účel této třídy spočívá v načtení a uchování textur podlah a stěn, názvů souborů s půdorysem pater domu / bytu, nákresu podlah a popřípadě zón (komponenta může pomocí **event** pak oznamovat přítomnost osoby v daných místech). Uchování konstanty pro přepočet kolika centimetrům odpovídá jeden pixel bitmapy. Implicitně je tato hodnota nastavena na 4 cm/pix.

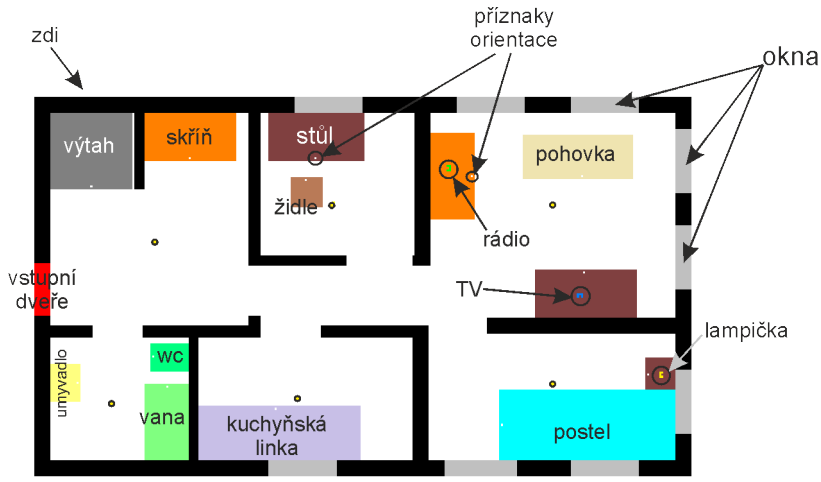
3.1.3 Třída Convertor

Slouží k převodu bitmap na grafické objekty komponenty.

Tato třída obsahuje metody a funkce, založené na principu primitivních hranových detektorů pro rozpoznání stěn (topologie patra), umístění oken, dveří, rozměrů a natočení grafických objektů. Je volána metodou pro načtení dat z třídy **MapViewer** nebo přímo konstruktorem třídy **MapForm** v případě, že není uložen „cachefile“ (umožňuje mnohem rychlejší načtení a obsahuje i informaci o posledním nastavení zobrazení – viz sekce Data).

Objektům v bitmapách jsou přiřazené barvy. Podle umístění těchto barev v nákresech pater (v bitmapách) jsou objektem této třídy (ve spolupráci s Library) přiřazeny a vytvořeny žádané předměty určitých typů podle odpovídajících barev (v editoru je možné tyto barvy měnit, kromě barev pro okna, dveře a stěny). Stejně tak jsou zde nastavené barvy odpovídající texturám podlah. Metody pro rozpoznání objektů jsou založené na podobném principu, ale v detailech se liší.

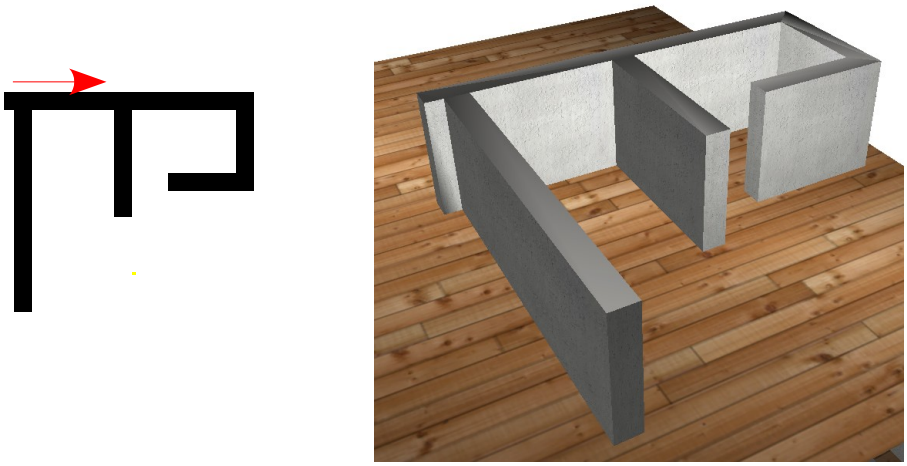
V mapách lze zanechat komentáře nevyužitou barvou pro objekty nebo zdi – **Convertor** převádí pouze barvy, které zná – z databáze v **Library**. Pro předměty jsou klíčové především hrany a příznak natočení, celá plocha obdélníku být vyplněna nemusí.



Obrázek 3 – bitmapa s půdorysem patra

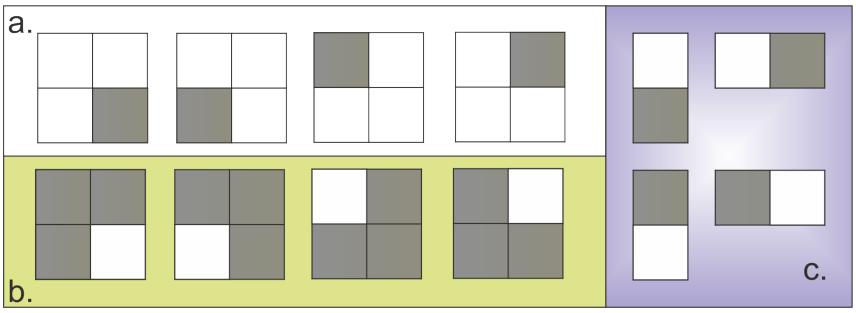
Základní myšlenka detekce hran

V nejzákladnějším případě – detekce hran pro vytvoření zdí je princip následující. Nejprve je vybírána barva – pro stěny je to černá. Z obrázku je vytvořena maska v podobě 2 rozměrného pole **bool**eanů o rozměrech obrázků. Tam kde byla nalezena žádaná barva (nyní černá) se v poli vyskytuje true, jinak false. Pole se poté prochází a v případě, že detektor v podobě for cyklů narazí na místo výskytu objektu, volá funkci pro vytvoření modelu stěny (WallContructor) s parametry v podobě souřadnic bodů a ta navrátí již zkonstruovanou stěnu (podrobnosti v sekci grafické objekty). Daný zkonstruovaný nákres je pak z pole odstraněn a pokračuje se ve vyhledávání dalších nákresů zdí až do doby, kdy pole obsahuje pouze hodnoty **false**.



Obrázek 4 – Vlevo bitmapa, vpravo výsledek konstrukce

V metodě WallConstructor se děje vlastní hranová detekce v podobě porovnávání vzorů. Postupem po směru hodinových ručiček se opisuje po obvodě tvar objektu, přičemž jsou do datového objektu **List** ukládány jednotlivé kraje objektu (rohy) a informace zdali se jedná o bod vnější (na obrázku **a**) nebo vnitřní (**b**). Toho je rovněž docíleno porovnáváním vzorů.



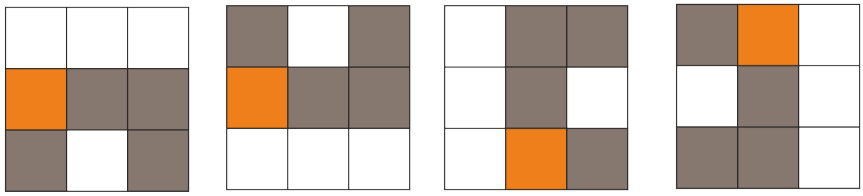
Obrázek 5 – vzory pro určení tvaru

Pokud na krajový bod sedí některý ze vzorů ze skupiny **a**, je bod uložen a označen jako vnější, pokud ze skupiny **b**, je označen jako vnitřní. Pokud na okolí bodu sedí pouze vzory ze skupiny **c**, pokračuje se v daném směru vyplývajícím ze vzoru po směru hodinových ručiček. (obrázek 4, červená šipka)

Z nalezených bodů, určujících tvar zdi je nakonec sestaven seznam bodů. Tento seznam bodů je pak poslán do konstruktoru grafického objektu **Wall**, kde je z něj vypočten 3D model zdi.

Metoda pro konstrukci statických předmětů (GetObjects)

Hranový detektor pro vyhledávání předmětů je zjednodušený a ochuzený o hledání vnitřních hran Naopak je rozšířen o vyhledávání příznaku orientace objektu vyjádřeného jedním z vzorů na obrázku 6.



Obrázek 6 – vzory pro určení orientace

Výsledkem funkce hranového detektoru je umístění a rozměry detekovaného objektu. Ty jsou nastaveny do objektu, získaného z Library. Každý takovýto objekt je tedy možné naškálovat na libovolnou velikost.

Obrázek 7 – nákres a výsledek vytvořený objektem třídy Convertor

Při umísťování objektů se stále kontroluje, zdali pod objektem umísťovaným do scény již některý neleží. Tato funkce byla implementována z důvodu variability komponenty. Před implementováním této funkce bylo nutné například u akčních objektů nastavovat již v modelu pevně jejich výšku umístění v ose z (například model televize umístěný na stole byl již modelován do dané výšky stolu). Avšak za pomoci této funkce je

jejich elevace určena automaticky (více v popisu grafických objektů) a tudíž je možné do komponenty vložit libovolně vysoký model (například stolu) a objekty na něj umístěné budou vždy ve správné výšce.

Metoda pro konstrukci akčních objektů (GetActionObjects)

V případě akčních objektů se vyhledávají přímo vzory orientace jako takové (obrázek 7 – modrý obrazec), vzory hranového detektoru pro určení velikosti tělesa již užity nejsou. Objekty jako je telvize nebo rádio se nezvětšují totiž přímo na určitou velikost, mají svojí velikost pevně danou již ze souboru modelu.

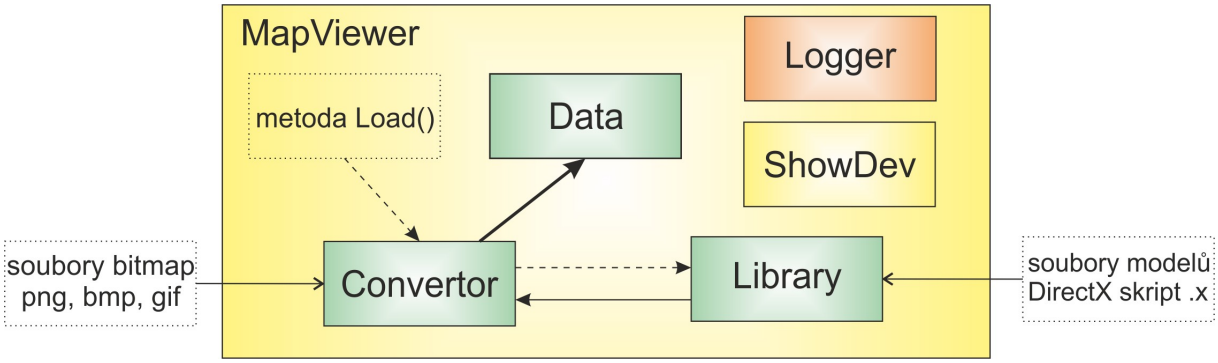
Metoda pro konstrukci zón zájmu (GetZones)

Používá detektor na stejném principu jako detekce stěn a podlah. Pouze s tím rozdílem, že výstupem je objekt **GZone** stejné barvy jako v bitmapě. Tento konstruktor není volán při načítání komponenty, ale pouze při její editaci – během vkládání nových zón ze souboru bitmap.

Během vývoje bylo zjištěno, že převedením této třídy (Convertor) na statickou by se dalo přibližně třikrát zrychlit načítání, ale z důvodu možnosti použití více nezávislých zobrazení v rámci jednoho programu byla třída ponechána jako dynamická.

3.1.4 Třída Data

Slouží pouze pro aktuální potřeby zobrazení, právě do této třídy se umisťují konkrétní grafické objekty, tvořené modely (podlaží, herců, předmětů) a zóny z třídy **Library**, vygenerované třídou **Convertor**. Jednotlivé objekty v ní uchované jsou pak modifikovány během používání komponenty podle informací o propojení nebo aktuálním stavu předmětů (vypnuto/zapnuto, regulace).

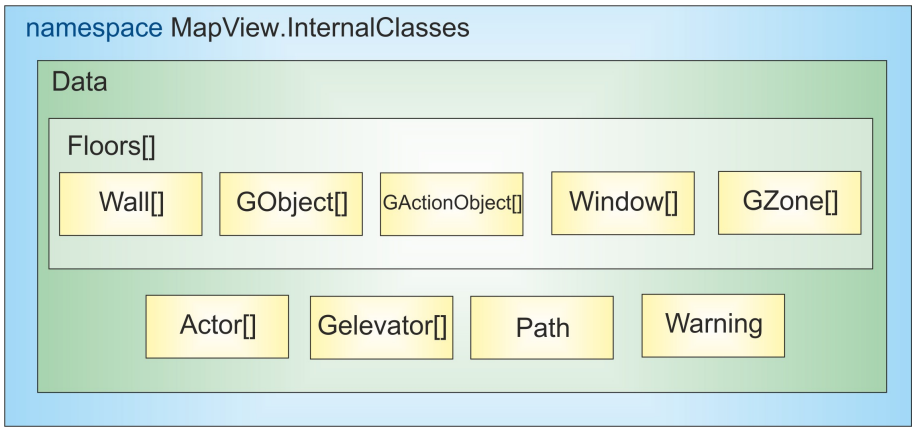


Obrázek 8 – diagram participace jednotlivých tříd při načítání dat

Třída je serializovatelná a je možné ji uložit do souboru „cachefile“, což mimo jiné také usnadní příští načítání komponenty (přeskočí se některé operace třídy **Covertor**) a zároveň uchová všechna nastavení provedená před jejím ukončením (například informace o propojení objektů, aktivované prvky, definované herce, uživatelsky pozměněné rozmístění předmětů a podobně). V případě užití „cachefile“ se tedy přeskočí část úseku kódu umísťující jednotlivé grafické objekty do scény a místo toho jsou tyto grafické objekty načteny ze souboru „cachefile“.

Samozřejmě se tak neděje automaticky, jelikož se jedná pouze o komponentu pro zobrazování dat a uložení nastavení je především na uživateli (programátorovi, technikovi) používajícím komponentu ve své aplikaci. Takováto vlastnost může být v mnoha případech velice užitečná – obzvláště při simulacích.

Třída obsahuje pole objektů **Floors** – objekt pro uchování jednotlivých pater bytu. **Floors** obsahuje pole **Wall**, kde je uložena topologie patra, pole **GObject a GActionObject**, kde jsou uložené statické a akční objekty a pole **GZone**, kde jsou jednotlivé zóny zájmu v patře bytu.



Obrázek 9 – struktura objektů ve třídě data

Pole **Window**[] bylo původně součástí pole **GActionObject**[]. Z důvodu správného vykreslování, později během implementace přidané průhlednosti, (vysvětleno v kapitole implementace – průhlednost) bylo ale nutné objekty této třídy (Window) od akčních objektů (GActionObject) oddělit, přestože se rovněž jedná též o akční objekt (Window je od GActionObjekt poděděna).

Ve **Floors** jsou tedy uloženy objekty patřící pouze do daného patra bez možnosti se mezi patry dynamicky přesouvat (například skříně na rozdíl od osob). Objekty mimo **Floors** jsou pole **Actor**[], **GElevator**[] a objekty **Warning** a **Path**.

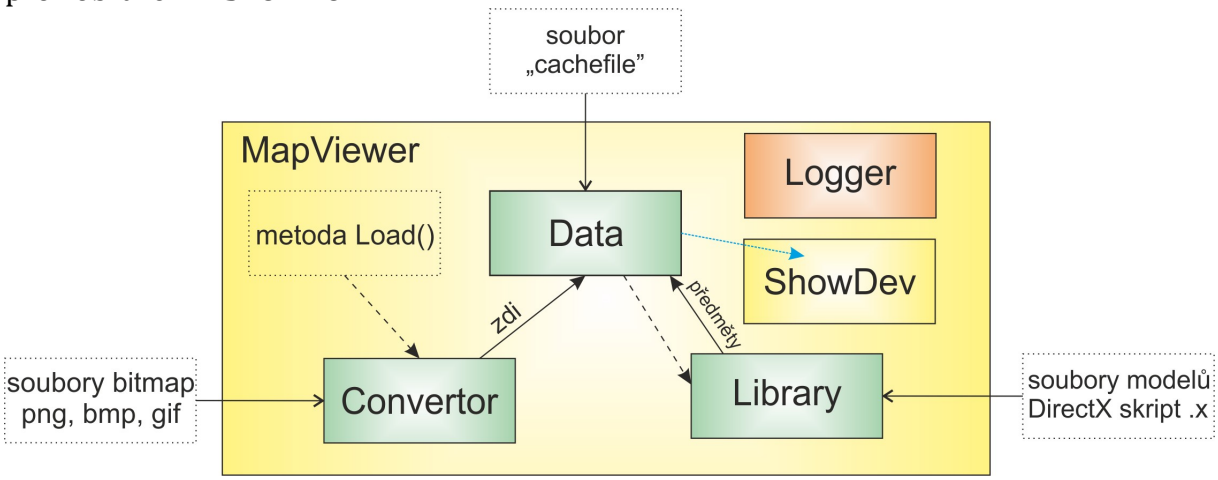
Pole **Actor**[] obsahuje herce, tedy objekty reprezentující osoby nebo vozíčky. Pole **GElevator**[] obsahuje grafické a funkční objekty výtahů. **Warning** je grafický objekt pro vykreslení varování nebo upozornění ve scéně a **Path** je grafický objekt k vykreslení cesty (například přesunu osoby).

Uložení do „cachefile“

Všechny implementované třídy komponenty reprezentující grafický objekt jsou serializovatelné a je možné je tedy uložit do souboru. Do souboru ale ovšem nejsou uloženy přímo modely, nýbrž jen jejich nastavení, poloha, transformace apod. Tento soubor má dle velikosti scény velikost řádově desítky kilobyte (pro scénu se 40 objekty je to 11kB, velikost jednoho modelu je v závislosti na složitosti jeho geometrie 20 až 400 kB).

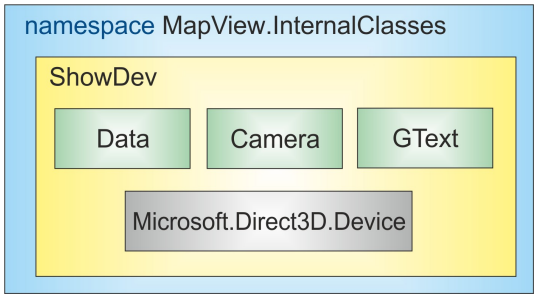
Jelikož tedy nelze přímo používaný model, sestavený pomocí prvků datových struktur Direct3D, přímo uložit do souboru, je při načítání použita zároveň třída Library. Ta načte ze své databáze stejné geometrické modely, každému z objektů ve třídě Data, ze souboru, za pomoci metody grafických objektů GiveMeshAndTextures – tedy vzorové objekty s modely v **Library** voláním této metody předají svůj model, materiál a textury objektu reprezentovaným parametrem metody. Tuto metodu obsahuje každý objekt a slouží pro poskytnutí svého modelu jinému grafickému objektu (podrobněji viz kapitola implementace).

Výsledná načtená scéna v objektu **Data** je pak jako reference předána rovněž třídě pro zobrazení – **ShowDev**.



Obrázek 10 – diagram participace tříd při načítání s „cachefile“

3.1.4 Třída ShowDev



Obrázek 11 – diaram vnitřního uspořádání ShowDev

V této třídě jsou implementovány všechny metody pro ovládání komponenty klávesnicí a myší, rovněž zapouzdřuje samotnou „Direct3D.Device“, ta slouží pro inicializaci grafické karty a rovněž pro posílání dat pro vykreslení do grafické paměti. Třída je poděděna od UserControl, jelikož právě do ní lze Direct3D přímo vykreslovat.

Obsahuje nepostradatelné metody pro inicializaci grafiky a smyčku pro vykreslování jednotlivých objektů a tři třídy zajišťující její funkce – **Data**, **Camera** a **GText**. Třída **ShowDev** vykresluje všechny objekty uchované ve třídě **Data**. Třída **Camera** slouží pro nastavování pohledů a animovaných průletů kamerou a **GText** slouží pro vypisování textu do plochy zobrazení.

Ovládání klávesami a myší

Jak již bylo zmíněno, v této třídě se rovněž nachází metody pro ovládání myší a klávesami. Toho je docíleno přesměrováním nativních funkcí komponenty **Windows.Forms.UserControl** – **OnMouseMove**, **OnMouseDown**, **OnMouseWheel** a **OnKeyPress**, od níž byla poděděna. Ovládání je využito především v editoru, při uživatelském nastavování pohledu kamery a při klikání na objekty v prostoru. Při inicializaci komponenty ve vlastní aplikaci je toto ovládání zakázáno a je na programátorovi, zdali jej povolí nebo si vytvoří vlastní (za pomoci události OnKey nebo jiných implementovaných) a tím například zamezí cílovému uživateli některé zásahy do zobrazení.

Vykreslovací metoda – Render()

Třída **ShowDev** slouží především pro vykreslování všech objektů. Tohoto lze docílit voláním její metody **Render**. Tato metoda je volána třídou **MapViewer** automaticky 20 krát za sekundu. V této metodě je nejprve nastavena kamera a poté jsou volány jednotlivé vykreslovací metody všech objektů na scéně, dále obsahuje také logiku pro nastavování právě editovaných grafických objektů, to z důvodu optimalizace. Není totiž například nutné přepočítat velikost právě editované skříně vždy při změně polohy kurzoru myši (tu je možné změnit vícekrát než 20krát za sekundu – tažením), ale stačí to pouze těsně před jeho vykreslením.

Ovládání pohledů

Pro ovládání pohledů uživatelem je využito metod **OnMouseMove** a **OnMouseWheel**. V komponentě je implementováno několik možných režimů práce s kamerou – floor camera (kamera fixovaná v dané výšce, změna výšky možná kolečkem myši), ree camera (kamera volného pohybu prostoru) a actor camera (kamera z pohledu očí objektu herce).

U „floor“ camery je možné měnit směr pohledu myší stisknutím levého tlačítka myši a výšku kamery kolečkem myši.

„Free“ kamera má 2 režimy. Režim stále aktivní a stiskem tlačítka myši aktivní. V případě použití režimu stále aktivní se zamkne kurzor myši vprostřed obrazovky a pohybem se mění úhly natočení kamery. V případě druhého režimu je funkce obdobná „floor“ kameře, jen s tím rozdílem, že výška není ovládána kolečkem, ale je dopočítávána ze směrnice pohledu – pohybu.

U obou typů práce s kamerou je pohyb vpřed proveden stisknutím nebo držením klávesy „w“ a pohyb vzad stisknutím nebo držením klávesy „s“.

Výběr objektů v prostoru

Pro výběr objektů v prostoru jsou užity metody **OnMouseMove** a **OnMouseDown**. V DirectX (v 9.0c bez použití nějakého rozšíření) neexistuje metoda pro kliknutí do prostoru, proto bylo nutné takovou metodu implementovat (MouseHit). Nachází se v základní třídě grafických objektů **GObject** (v sekci o grafických objektech je také podrobně popsána). V případě vybírání objektů jsou všechny grafické objekty postupně testovány voláním této metody s parametry podle pozice kursoru myši. V případě kliknutí na objekt tedy tato funkce navrátí pozitivní odezvu a vzdálenost od předmětu. Předmět s nejkratší vzdáleností od pozorovatele je potom označen za vybraný a v případě kliknutí (OnMouseDown) může být navrácena událost **OnClickableClick** nebo **OnClickableDoubleClick**. V případě povolení označování objektů uživatelem, je po kliknutí na objekt nastaven příznak **Selected**, následkem čehož dojde k jeho označení poloprůhledným červeným modelem kvádru, v případě pouhého zanechání kurzoru nad tímto předmětem je označen pouze drátovým modelem kvádru a objektem třídy **GText** je potom na pozici kurzoru myši zobrazen jeho název.

Po výběru objektu, a pokud objekt měl nastaven příznak **IamClickable**, je možné odchytit z komponenty událost **OnClickableClick** nebo **OnClickableDoubleClick**. Tyto události vracejí název kliknutého objektu, jeho vzdálenost od pozorovatele a koordináty kurzoru myši.

Další možností je objekt vstupy z myši a klávesnice například editovat – nastavovat velikost, umístění, rotaci, elevaci a výšku.

Uživatelský mód editace objektů v prostoru

V DirectX rovněž neexistuje žádný objekt, metoda nebo událost umožňující uchopit objekt v prostoru nebo jej popotáhnout myší, obsahuje objekty a metody určené povětšinou pro vykreslování. Tudíž bylo nutné tyto metody rovněž implementovat.

Pro uživatelské pohodlí byla do komponenty implementována snadná editace objektů v prostoru klávesovými příkazy a myší. Tyto metody lze omezit nebo zakázat, tudíž má i programátor, tuto komponentu používající, možnost koncovému uživateli tuto

možnost zakázat nebo povolit jen v určitém žádoucím rozsahu.

Za možností snadné editace se skrývá poměrně rozsáhlá logika implementovaná v třídách grafických objektů, kde je popsána podrobněji. Možnosti uživatelské editace jsou následující:

Editace polohy objektu s gravitací – v tomto režimu komponenta zjišťuje, zdali pod právě umisťovaným objektem neleží již objekt jiný a v tomto případě jej umístí nad needitovaný. Gravitace je počítána vždy pouze pro právě editovaný objekt, aby nedocházelo k nežádoucím změnám ve scéně.

Editace polohy objektu s volitelnou elevací – v tomto režimu komponenta gravitaci nepočítá, pouze je možné elevaci objektu měnit kolečkem. V metodě pro odchytávání události použití kolečka myši se tedy přičítá nebo odečítá konstanta elevace.

Editace polohy s rotací objektu – kolečkem myši je ovládáno natočení objektu, gravitace se počítá pro editovaný objekt.

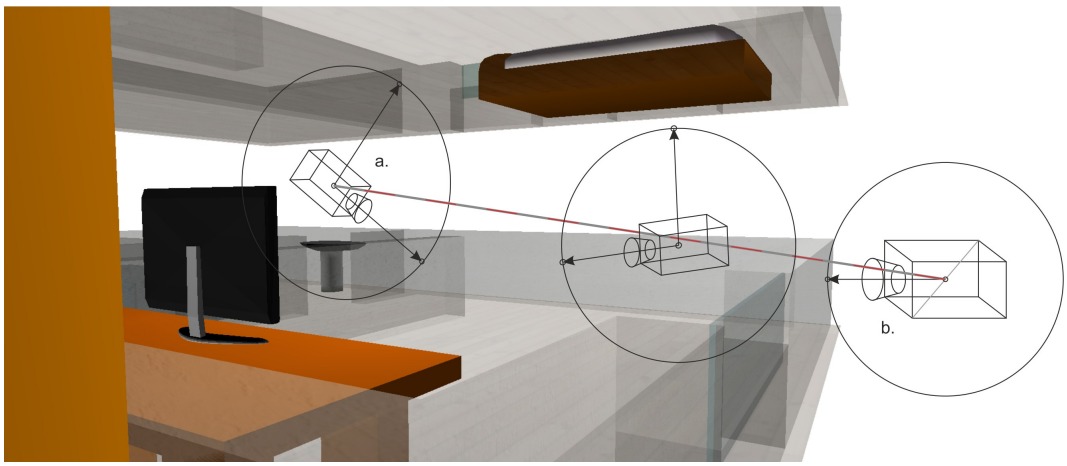
Editace velikosti objektu v ose z – kolečkem myši se škáluje objekt v ose Z.

Editace velikosti objektu

Pro tyto editace jsou použity události metody **ShowDev** – **OnKeyDown**, **OnMouseMove**, **OnMouseDown** a **OnMouseWheel**.

Třída Camera

Nezbytnou součástí třídy **ShowDev** je objekt třídy **Camera**. Ten obstarává animace a především výpočty natočení a trasování pohledu kamery. Tyto operace byly implementovány za použití jednoduché analytické geometrie a tříd **Microsoft.DirectX Vector2** a **Vector3 –** tyto třídy usnadnily implementaci základních matematických vzorců pro výpočet vzdálenosti vektoru od počátku souřadného systému, a početní operace s vektory - (Add - součet, Subtract - rozdíl, Multiply – násobení). Na zbytek operací (jako sin, cos, asin a acos) byla použita statická třída **Math**. Vzhledem k tomu, že veškeré početní operace týkající se grafického zobrazení a DirectX jsou ve **float**, je potřeba části počítané třídou Math vždy přetypovat, jelikož počítá v datových formátech **double**.



Obrázek 12 – ilustrace animované změny pozice kamery (SetTrack)

Obrázek nastiňuje princip funkce animovaného přesunu kamery. Funkci je na vstupu předána aktuální poloha kamery s úhly natočení a žádaná cílová poloha s úhly natočení. Pomocí analytické geometrie je pak trajektorie kamery rozdělena na několik elementárních příspěvků. V každé chvíli vykreslení scény se pak jeden tento elementární příspěvek přičte k aktuální pozici kamery. S úhly natočení je výpočet obdobný pouze s tím rozdílem, že se nejdříve zjišťuje, zdali se bude točit kamerou doleva nebo doprava, jelikož nikdy není třeba se otočit o více než 180 stupňů, podle toho je pak elementární příspěvek pro rotaci kamerou kladný nebo záporný.

Pro efektivní práci s kamerou byla do komponenty navíc implementována logika, umožňující natočení kamery na objekt pouhým zadáním jeho názvu. Komponenta se už sama postará o vyhledání tohoto objektu v prostoru a funkcím kamery předá vhodné parametry. Metod pro tyto účely je několik druhů – pro natočení pohledu na objekt, pro natočení pohledu na objekt s přiblížením do určité vzdálenosti, pro natočení pohledu na objekt zepředu a přiblížení do určité vzdálenosti nebo úhlu pohledu.

Všechny metody pro animované změny pozic kamery lze volat zvenčí komponenty. Programátor užívající komponentu má tedy kontrolu nad tím jak dlouho bude animace kamery trvat a za použití jednoduchých metod, jejichž parametry jsou buď koordináty, nebo přímo název předmětu, může vytvořit velice efektní animovaný posun kamery na vybraný předmět nebo do žádané pozice.

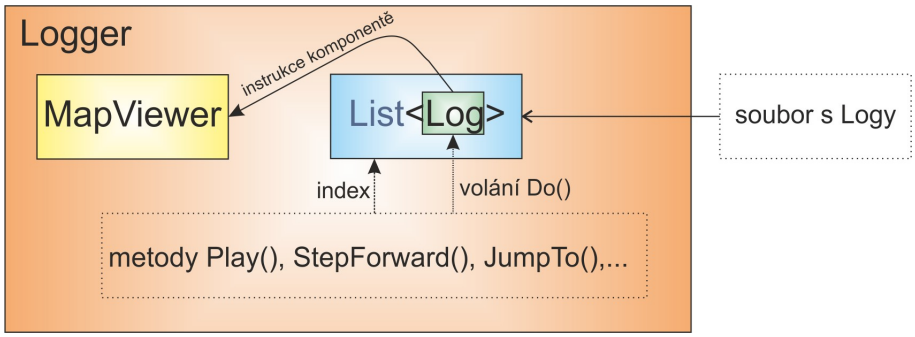
Třída GText

Tato třída byla implementována k vykreslování textu na plochu zobrazení. Pokud by byl umístěn přímo Windows.Forms.Label nebo jiný prvek z knihovny Windows.Forms do plochy zobrazení, mohlo by docházet k nepředvídatelnému chování komponenty. Pro tyto případy je v Direct3D třída Font. Ta umožňuje zobrazit prostý text kamkoliv do plochy zobrazení. Metoda třídy Font pro vykreslení slouží pouze pro vykreslení jednoho řádku textu a velmi se podobá metodám pro vykreslení jakéhokoliv jiného objektu v Direct3D, proto byla zapouzdřena do třídy GText. GText má implementované metody pro

vykreslení nápovědy, více řádků textu a textu na určitou pozici obrazovky, o vykreslování se již není nutné starat, jelikož je GText implementovaný tak aby to nebylo nutné.

2.1.5 Třída Logger

Logger dokáže zasílat instrukce třídě **MapViewer** a rovněž dokáže veškeré instrukce, poslané aplikací (do níž je komponenta umístěna) třídě **MapViewer**, ukládat (což se děje v případě aktivace ukládání). Obsahuje metody pro návrat na začátek záznamu, přetočení na konec záznamu, či skok do určité časové pozice záznamu dle času nebo čísla položky. Dále obsahuje i časovač a je možné Logger nastavit na určitou rychlost přehrávání dle toku času nebo pozpátku.



Obrázek 13 – proces předávání instrukcí třídě MapViewer

Ukládání jednotlivých operací a událostí (přidání osoby – Actor, událost vstoupení herce do zóny, propojení objektů, výjimka, stav objektu – vypnuto, zapnuto, otevřeno, zavřeno) je provedeno jako přidávání položek do datového objektu **List<Log>**, kde **Log** je objekt uchovávající provedené změny. Ukládají se tedy pouze změny, jež nastaly v daných časových úsecích, a informace o předchozím stavu měněného atributu, jelikož ukládání stavu všech objektů v každém okamžiku by bylo paměťově náročné a neúsporné.

Samozřejmě by mohla nastat situace, kdy uživatel vypne nahrávání a něco změní. V tomto případě by pak přehrávání neproběhlo správně a změny stavů objektů by mohly mít za následek úplně jinou výslednou scénu než jaká by měla správně být. Proto byla do **Logger**u ještě implementována funkce **Snapshot**(). Tato funkce je volána automaticky vždy, když se započne nebo skončí nahrávání (logování), ukládá celý aktuální stav komponenty (podobně jako „cachefile“) do jednoho **Log**u. Má ve své podstatě stejný význam jako klíčové snímky v MPEG4 komprimovaném filmu. S touto funkcí již nemůže nastat zmíněná chyba v přehrávání, když uživatel nahrávání zastaví, změní, vymaže nebo přesune objekty a znovu nahrávání spustí. Velikost jednoho takového klíčového snímku pro scénu obsahující kolem 40 objektů je 11kbyte.

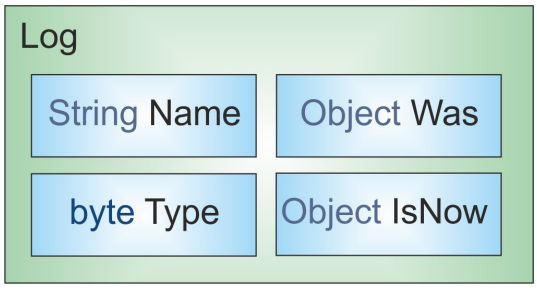
V případě, že se tedy uživatel chce podívat na nějakou určitou pozici v čase, je nutné projít celý list od časového úseku, kde se nachází nyní nebo kde se nachází „snapshot“ až do času žádaného. V případě procházení listu vpřed, tedy po směru toku času, jsou prováděny změny **IsNow**, v případě procházení proti směru toku času, jsou brány instrukce předchozích stavů objektů (nelze provádět přímo negace předchozích stavů, jelikož uživatel mohl nastavit atribut objektu komponenty 2x za sebou na stejnou hodnotu).

Při přehrávání je možné aktivovat vykonávání přehrávaných eventů, komponenta tudíž bude poté hlásit veškeré eventy, tak jak je hlásila během normálního běhu (například i kliknutí na objekt, což může být užitečné v některých simulacích).

Pro užití ve vlastní aplikaci má Logger event **OnAddLog**. Tento event lze využít na odchytávání právě přidávaných logů a jejich zobrazování například do seznamu v dialogovém okně - prvek **Windows.Forms.ListBox** a následné zobrazení detailů logu nebo pro ukládání do vlastní databáze logů.

Třída Log

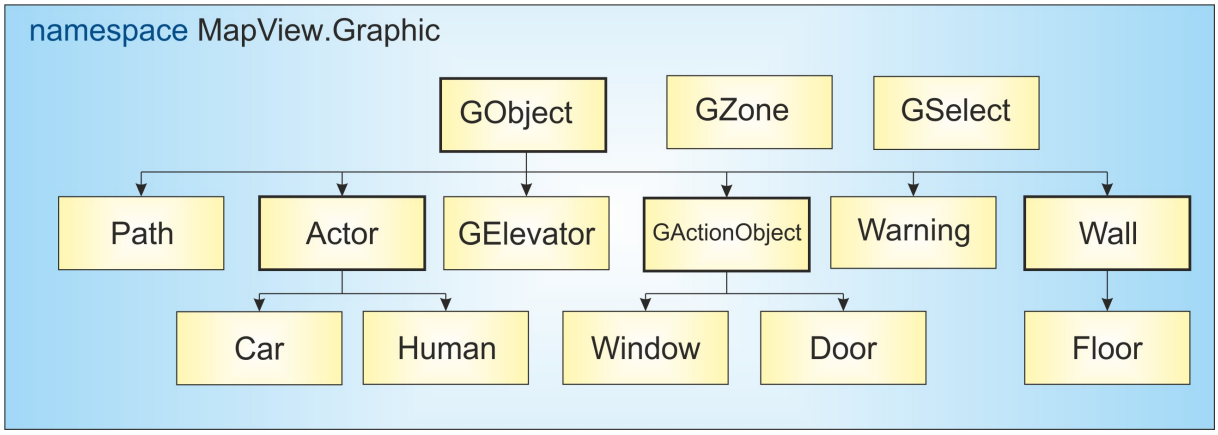
Třída Log je výkonným prvkem loggeru a zároveň také úložnou entitou. Obsahuje metody pro ovládání komponenty. Tyto metody jsou užívány v případě přehrávání záznamu. Log v sobě obsahuje základní logiku pro směr přehrávání. Pro tento účel jsou v Logu 2 objekty uchovávající parametry stavu objektů, řetězec se jménem objektu, ke kterému parametry náleží a byte Type, ten udává typ provedené akce (událost, změna stavu, pohyb, vytvoření nebo smazání objektu apod.). V případě změn objektů a přehrávání po směru toku času jsou jím prováděny akce uložené v objektu **IsNow**, v případě přehrávání proti toku času se provádí akce uložené v objektu **Was**. V případě zanikajících a vznikajících grafických objektů bývá v jednom z těchto úložišť **null** (tedy prázdný prvek). Dle umístění a informace o toku času pak Log rozhodne, zda-li se objekt vytvoří nebo zanikne.



Obrázek 14 – vnitřní uspořádání třídy Log

3.2 Grafické objekty

Nyní budou v logické posloupnosti popsány třídy tvořící grafické objekty ve vlastním zobrazení. Pro snadnější orientaci byl vytvořen diagram jejich hierarchie.



Obrázek 15 – diagram dědičností tříd

3.2.1 GObject

Základní třída pro grafický objekt uchovávající informace o jeho geometrii, viditelnosti, pozici a velikosti. Veškeré ostatní grafické objekty, vyjma **GZon**e a **GSelect**, jsou dědici této třídy.

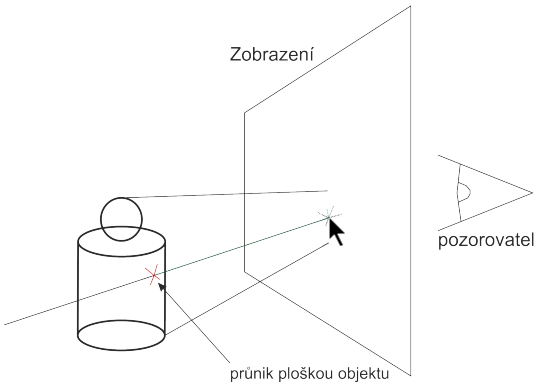
Skládá se z metod pro načtení svého modelu ze souboru (volaných objektem třídy Library), vykreslení a pro transformování objektu na žádanou velikost. Další potřebné metody implementované do **GObject** slouží k výpočtu průniku přímky stěnami modelu, což je využito zejména při zjišťování, zda-li na objekt bylo kliknuto, rovněž obsahuje metodu pro simulaci gravitace v případě, že se na určitý objekt umístí jiný objekt (aby nepropadl na zem – popsáno částečně v sekci **Convertor**u).

Kromě informace o souboru modelu obsahuje informaci o umístění předmětu, škálování, natočení a barvě. Rovněž uchovává případné použité textury nebo Direct3D materiály (Direct3D.Material) a podrobnosti o typu objektu – zdali se jedná o ovladač (příznak Controller) nebo ovládaný předmět (Controlled) a také jestli je označitelný. Pro účely propojení objektu obsahuje objekt ve funkci ukazatele na objekt, jímž je ovládán (respektive předmět, který jej ovládá).

Každý model, načítaný z DirectX formátu .x musí být velikosti 1x1 v rovině xy, aby komponenta mohla objekt naškálovat přesně na požadovanou velikost (dle mapy nebo uživatelského vstupu). Objektem reprezentovaným touto třídou může být libovolný statický objekt, jako například skříň, stůl nebo vypínač.

Kliknutí myší

Důležitou vlastností **GObject** je možnost detekce kliknutí na objekt v prostoru – k zajištění funkce režimu uživatelských editací či aktivací předmětů jednoduše klikáním na ně. K tomuto byla implementována funkce (MouseHit) se vstupními parametry kursoru myši na zobrazovací ploše, schopná rozeznat zdali se nachází kursor právě nad příslušným objektem.

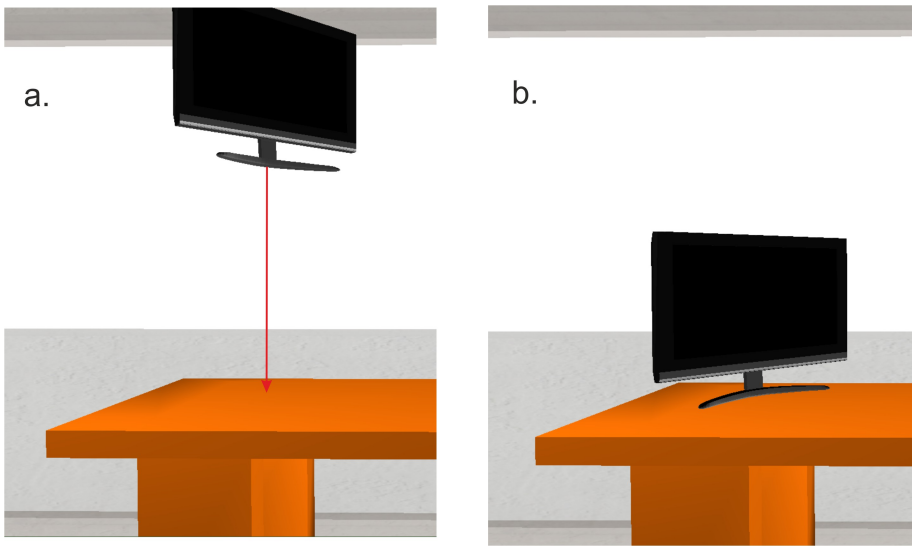


Obrázek 16 – klik do prostoru, průsečík

Tato funkce, transformuje souřadnice myši z 2 rozměrného prostoru obrazu, vnímaného pozorovatelem, do prostoru modelů a následně z transformovaných souřadnic vytváří polopřímku, u níž se zjišťuje průnik s elementární ploškou modelu předmětu.

Gravitace

Funkci pro zjištění kliknutí na předmět (MouseHit) se velice podobá funkce **GravityDistance**. Funkci **GravityDistance** volá třída **Convertor** (respektive **ShowDev**, v módu editace uživatelem) pro účely výpočtu „gravitace“ při umisťování předmětů do scény. Tato funkce se liší především svým vstupem. Tím je vektor umístění „padajícího“ předmětu do scény. Každý objekt je při vytvoření spouštěn z maximální výšky patra, v němž se nachází. Pokud tedy pod tímto padajícím předmětem leží libovolný jiný předmět, navrátí parametrem **dist** vzdálenost od tohoto předmětu.



Obrázek 17 – umístění objektů s gravitací

V případě počítání „gravitace“ se vyšle polopřímka ze středu předmětu ležícího nad objektem se směrnicí (0, 0, -1). Metodou pro počítání průsečíků polopřímky s plochou je zjištěna vzdálenost protnutého předmětu od právě umisťovaného předmětu. O tuto vzdálenost je poté objekt, pro který počítáme gravitaci posunut níže. Tímto se tedy docílí efektu, že objekty jsou těsně na sobě a nevznáší se ve vzduchu.

Vykreslování

Zřejmě nejdůležitější implementovanou metodou této hlavní třídy pro grafické objekty je **Render**(), sloužící pro vykreslení předmětu jako takového. Tato metoda je volána při každém vykreslení scény objektem třídy **ShowDev**. Mimo jiné obsahuje i přidružené metody schopné vykreslit označení takového předmětu drátovým modelem kvádru nebo polopropustným kvádrem o velikosti předmětu.

Dalšími významnými součástmi této třídy jsou metody pro změnu barvy, umístění, rotace a velikosti modelu předmětu.

3.2.2 GActionObject

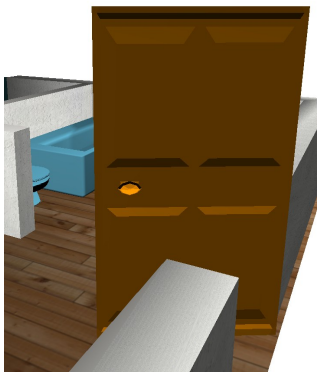
Rozšíření třídy GObject → tento objekt může být animovaný, může mít několik stavů. Typičtí zástupci **GActionObject** jsou například TV, rádio, okna, dveře, světla, lampičky, větrák, atp.

V tomto objektu mohou být uchované předměty typu zapnuto/vypnuto, regulovatelné nebo stavové. Dokáže v sobě rovněž uchovat animaci – posloupnost po sobě jdoucích modelů **GObject**. Dle nastavení příznaku typu objektu je pak animace přehrávána různou rychlostí - v případě nastavení příznaku „regulátor“. V případě nastavení příznaku „stavový předmět“ se zobrazuje jen určitý model z posloupnosti animace, . Dále obsahuje možnosti pro složení modelu ze dvou různých souborů a možnost nastavení jedné z částí modelu k vykonávání pohybu (rotujícího, pulsujícího).

Současně dle nastavení svých proměnných o stavu lze pak některým elementem modelů například otáčet (větrák), pulsovat (repro u rádia) nebo spustit animaci tvořenou posloupností modelů.

GActionObject také může mít **Convertor**em od Direct3D.Device přiřazené světlo. Toto světlo obsluhuje v závislosti na své poloze a stavu. Pokud je tedy světlo vypnuté nastaví mu barvu na černou a intenzitu svítivosti na 0 a naopak, v případě nastavení příznaku regulovatelnosti, je pak možné nastavit přímo intenzitu světla. V **GActionObject** jsou tedy uchovávána i hlavní světla jednotlivých místností.

3.2.3 Door

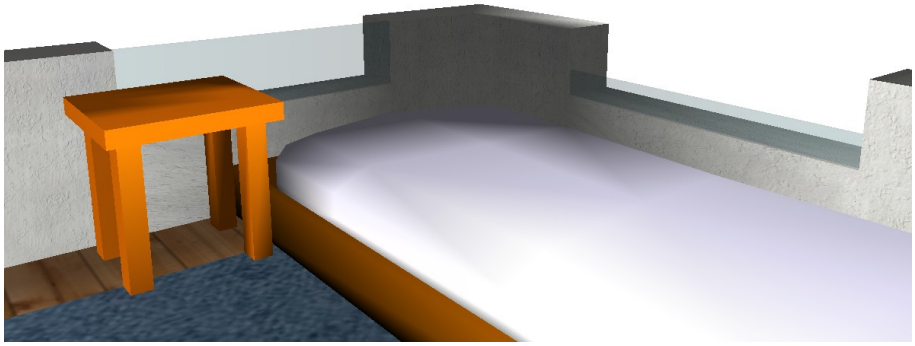


Obrázek 18 - Dveře

Je objekt poděděný od **GActionObject**. Jak název napovídá, slouží pro zobrazování modelů dveří a jejich obsluhu. Může nabývat hodnot otevřeno nebo zavřeno a podle toho natočí v něm obsažený model. Od GactionObject se liší ještě v konstruktoru – dveře jsou naškálovány na přesnou plochu v bitmapě třídou Convertor na rozdíl od jiných akčních objektů.

3.2.4 Window

Je rovněž poděděný od **GActionObject**. Částečně souvisí s objektem **Wall**, ten totiž volá, v případě výskytu okna ve zdi, konstruktor **Window** s hodnotami velikosti a umístění okna, na základě čehož vygeneruje model otevíratelného okna. Okno je pak animovaně otevíráno nastavením nových hodnot vrchním bodům okna v ose Z.



Obrázek 19 – Okna - vlevo zavřené, vpravo otevřené

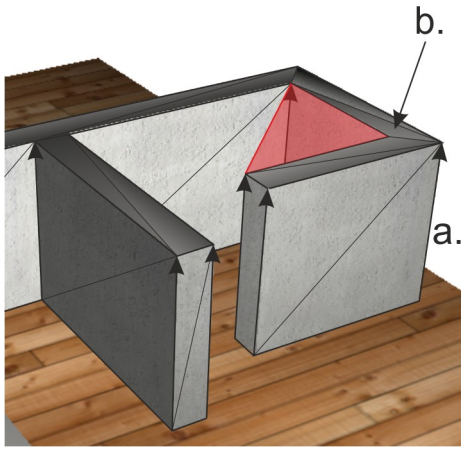
3.2.5 Wall

Objekt této třídy uchovává geometrii právě jednoho řezu zdí. V konstruktoru tohoto objektu je umístěn algoritmus pro převedení seznamu bodů na model zdi. Vstupem (parametry konstruktoru) je seznam bodů rohů a seznam **bool** informujících zdali se jedná o vnitřní nebo vnější bod. Tyto seznamy byly vygenerované hranovým detektorem třídy **Convertor**.

Dále obsahuje konstruktor pro sestavení zdi pod oknem – tato zeď je nižší.

Algortimus pro konstrukci 3D modelu stěn lze dělit na 2 části – vytažení stěn do výšky (**a** na obrázku 19) a přiklopení zdi stěnou shora (**b** na obrázku 19). Algoritmus pro vytažení zdí do výšky je poměrně snadný – pouze prochází seznam bodů a vytváří dvojice trojúhelníků (základních stavebních prveků v DirectX), tvořících čtverce (stěny zdi – **a** na obrázku 19).

Algoritmus pro vytvoření vrchních stěn řezů zdí již tak snadný není. Jelikož stěny mohou být libovolného tvaru (aby nemusela být omezena složitost tvaru) a mohou obsahovat vnitřní rohy, nelze pouhým procházením seznamu s body generovat jednotlivé trojúhelníky, jelikož by mohlo dojít k vytvoření horních stěn v místech, kde se zeď nevyskytuje (obrázek 19 – červený trojúhelník). Pro tento účel byl vytvořen algoritmus umístěný v třídě **EdgeEar**.



Obrázek 20 – Generování zdí ze seznamu bodů

Algoritmus v třídě EdgeEar

Vstupem tohoto algoritmu je seznam bodů v rovině. Algoritmus testuje zdali 3 body po sobě následující tvoří trojúhelník, tak aby nevyčníval ven z tělesa a aby neprotínal těleso v některé jeho hraniční čáře. Pokud právě vybraný trojúhelník splnil všechny podmínky je uložen do seznamu triangles a prostřední vrchol tohoto trojúhelníku je odstraněn ze seznamu bodů, tvořících těleso. Algoritmus tedy tímto způsobem postupně odebírá body ze seznamu, až nakonec zbydou jen body 2, z nichž již nejde vygenerovat trojúhelník, a je tedy ukončen.

Tento algoritmus byl implementován na základě teorie popsané na webu – geometrictools.com [4].

Kliknutí na model patra

Jelikož by bylo vhodné aby si uživatel mohl vybrat patra, která ho zajímají nebo například určit pozici právě umisťovaného předmětu, bylo nutné model patra pře-konvertovat do objektu DirectX třídy **Mesh**.

Pro tento účel (klikání na objekt do prostoru) slouží třída **WallMesh**. Tato třída sice technicky mezi grafické objekty patří také, ale nikdy se nevykresluje. Slouží pouze k uchování objektů mesh, tvořících patro, vygenerované metodou třídy **Wall**, aby bylo možné využít již hotovou metodu v **GObject** pro počítání průsečíku vektoru s tělesem v prostoru a nebylo nutné ji implementovat složitě znovu (jiným způsobem).

Kliknutí na model patra je implementováno dvěma metodami – jedna metoda slouží po výběr patra jako takového a druhá pouze pro výběr pozice v ploše daného patra – toho je využito v simulačním programu pro výběr umístění vkládaného herce. Obě metody lze využít na libovolné jiné účely.

Floor

Floor je objekt poděděný od Wall. Uchovává pouze podlahy a jejich textury. Na

generování těchto podlah rovněž využívá algoritmu **EdgeEar**.

3.2.6 Actor

Actor je poděděná rozšířená třída objektů od třídy **GObject**, sloužící jako základní třída pro „Herce“. Herec je přidáván uživatelsky až během používání komponenty (nebo může být načtený z „cachefile“ či logu). Obsahuje informace o pozici v patře, patře samotném a úhlu natočení, ale na rozdíl od **GObject** transformační matici počítá v každém vykreslení (u statických objektů to není nutné).

Obsahuje důležité metody pro zjištění nárazu herce do zdi nebo objektu (GetHitWallStatus a GetHitObjectStatus). Tyto metody jsou komponentou vždy volány v případě změny umístění herce v prostoru.

Náraz je zjištěn při změně umístění herce vysláním polopřímky ve směru posunu, vypočteného ze změny umístění, z polohy kde se nachází. V případě protnutí předmětu nebo stěny se zjišťuje, zdali tato stěna není blíž než je velikost herce v rovině x-y. Pokud tato situace nastane, je vyslána událost o nárazu herce do stěny. V případě užitečnosti této informace ji lze pomocí těchto událostí (OnActorHitWall a OnActorHitObject) z komponenty odchytit.

K výše zmíněným metodám pro zjištění nárazu se také váže příznak **MovementCorrection**. Na základě nastavení tohoto příznaku bude pak pohyb herce přímo kontrolován a komponenta mu nedovolí projít zdí nebo objekty (lze nastavit obojí zvlášť). Tyto pokusy o projití zdí nebo objekty ve scéně jsou samozřejmě logovány a rovněž vraceny formou událostí. Tohoto efektu je dosaženo pomocí 2 proměnných uchovávajících informaci o předchozí poloze. Pokud by tedy nastal po zadání nových souřadnic průchod herce zdí, okamžitě by byly nastaveny souřadnice předchozího kroku. Pokud je tedy **MovementCorrection** aktivován, nelze herce ani přesunout z místnosti do jiné, je nutné příznak nejprve deaktivovat nebo jej přesouvat tak, aby vždy procházel dveřmi.

3.2.7 Human

V objektu **Human** jsou implementovány datové struktury pro uchování tvaru těla a nohou a metody pro jejich vykreslení. Dále obsahuje funkce pro animovanou chůzi. Příznak pro aktivaci těchto funkcí je možné aktivovat metodou komponenty ze strany uživatele. Chůze je implementována jako cyklické zmenšování a zvětšování velikosti modelů nohou střídavě po sobě.

Obrázek 21 – model člověka

3.2.8 Car

V objektu třídy **Car** je uchován model vozíku se 4 koly. Je rovněž poděděn od třídy **Actor** a rozšiřuje jí o otáčení kol při změně polohy. Také obsahuje vlastní metodu pro vykreslování. Model kola je vykreslován 4 krát po sobě v různých polohách dle nastavení vektorů jejich umístění. O tato nastavení se v případě přidání nového modelu stará programátor nebo uživatel používající editor komponenty, je totiž nutné nastavit správné umístění připojení koleček k modelu vozíku.

Obrázek 22 – model vozíku

V komponentě lze v editoru nadefinovat neomezený počet typů herců třídy Car nebo Human (tudíž můžou být například modely „Žena“, „Muž“, „Dítě“, … a podobně).

3.2.9 GZone

Uchovává plochu zóny s nastavenou barvou a názvem. Obsahuje funkci (IsIn) pro zjištění výskytu herce v zóně i metodu pro vykreslení. Metoda pro vykreslení je volána třídou **ShowDev** jen v případě nastavení příznaku **ShowZones** metodou komponenty. Zóny jsou pak vykresleny jako polopropustné plochy odpovídající barvy dle jejich nastavení.

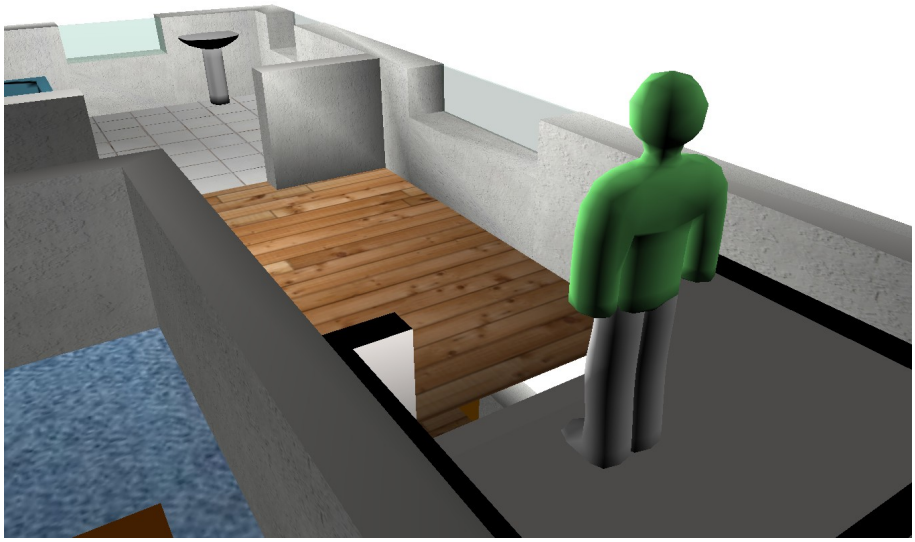
Obrázek 23 – pohled na 2 zóny zájmu s herci uvnitř

Funkce sloužící k detekci herce uvnitř zóny spočívá v zjišťování zdali bod určený pozicí herce leží v daném patře a uvnitř některého z trojúhelníků tvořících zónu.

Poslední důležitou vlastností je možnost vytvářet zónu za běhu komponenty kreslením obvodových čar. Uživatel si tak může nadefinovat zónu, ve které když se někdo nachází se například rozsvítí světlo nebo aktivuje audio systém (samozřejmě komponenta pouze navrátí obslužné aplikaci událost o vstoupení herce do zóny a na té, záleží jak s informací naloží).

3.2.10 GElevator

Objekt třídy GElevator slouží k uchování modelu výtahu a obstarává jeho funkce. Především schopnost přesouvat se animovaně mezi patry a zároveň také pojmout herce. Pro účel přesouvání herců mezi patry v sobě obsahuje datovou strukturu seznam a jednu zónu (GZone), s jejíž pomocí zjišťuje, zdali herec nenastoupil / nevyskytuje se na ploše výtahu. Tato zóna se posouvá společně s výtahem. V případě výskytu herce v této zóně se zašle reference do seznamu herců ve výtahu a výtah mu pak pravidelně dle své polohy upravuje umístění a elevaci. Herec bude ze seznamu odebrán, pokud z této plochy vystoupí



Obrázek 24 – herec jede výtahem do spodního patra

3.2.11 Path a Warning

Objekt třídy **Path** slouží k vykreslení trajektorie v prostoru dle zadaných bodů. Do komponenty rovněž byla implementována funkce pro vykreslení trasy pohybu herce. Funkci stačí jako vstupní parametry název objektu herce a časová období. Funkce prohledá logy a na základě poloh daného herce vykreslí cestu.

Objekt třídy **Warning** slouží pouze pro vykreslení 3 druhů varování na určitém místě. Pokud není varování příliš podstatné je zobrazen na místě varování jen malý hranol. Pokud je varování nastavena větší důležitost je nad tímto hranolem vykreslen navíc rotující vykřičník a v případě nejvyššího nebezpečí vykřičník navíc bliká. Využití tohoto objektu modelu je zřejmé – například upozornění požáru nebo nefunkčnosti nějakého zařízení.

3.2.12 GSelect

GSelect byla implementována jako jednoduchá pomůcka pro zobrazení uživateli umístění v ploše patra, do něhož ukazuje kursorem myši. Objekt třídy GSelect se využívá, když uživatel naklikává do plochy patra body ohraničující nově vytvářenou zónu nebo když po něm je žádán výběr umístění v ploše libovolného patra obecně.

3.3 Popis simulátoru (Simulator.exe)

Simulátor byl implementován pro základní testování většiny možností komponenty a případné přehrávání uložených logů. Simulátor by mohl být samozřejmě napsán mnohem složitěji, jelikož komponenta svou variabilitou umožňuje mnoho možností – například by mohl odchytávat události kliknutí do prostoru a posílat herce na

dané pozice apod. Ale takovýto vývoj by byl rovněž časově náročný a čas již byl především vynaložen na tvorbu komponenty jako takové.

Simulátor používá implementovanou knihovnu **MapView.dll** s komponentou, skládá se ze tříd **ActorBehavior**, **ListBoxActualizator MainWindow**, **AddActor**, **Controller**, **CamerControlForm**, **EventTestForm**, **GlobalSettingsForm**, **KeyForm**, **LogForm** a struktury **KeyControl**.

Jak už názvy napovídají třída **ActorBehavior** slouží pro simulaci chování „herců", třída **Controller** slouží pro nastavení a samotné ovládání herců na klávesnici a **MainWindow** je hlavní grafický interface pro nastavení simulátoru a jednotlivých objektů v komponentě. Rovněž lze z hlavního okna spustit editor komponenty. Jednotlivé dialogy a formuláře budou v tomto popisu vynechány, jelikož jsou dostačujícím způsobem osvětleny v kapitole 4 – Manuál užití, v sekci Simulátor.

3.3.1 ActorBehavior

Jedná se o jednoduchou třídu, uchovávající v sobě jména herců na scéně. Tyto jména získává z komponenty pomocí funkce k tomu určené a v případě nastavení automatického chování některých herců, posouvá tyto herce vpřed. Odchytává z komponenty události o nárazu do zdi a v případě nárazu herce do stěny tomuto herci zvolí náhodný nový úhel pohybu. Druhé možné nastavení je manuální ovládání herců klávesami. K tomu je užito jednoduché struktury **KeyControl**. Tato struktura se skládá ze 4 znaků, uchovávajících pro jednotlivé manuálně ovládané herce klávesy pro chůzi vpřed, vzad, otočení doleva a doprava.

3.3.2 ListBoxActualizator

Objekt této třídy slouží, jak název napovídá k aktualizaci jednotlivých ListBoxů v dialozích simulátoru. Jelikož některé dialogy v sobě uchovávají stejné seznamy položek a v některých dialozích je možné tyto položky přidávat nebo ubírat je nutné je mít při otevření více dialogů vždy aktuální. Tato třída byla implementována, jelikož nelze předávat kolekce položek seznamů ListBox referencemi a tudíž musí mít každý ListBox svojí vlastní kolekci položek. Předávat si mezi dialogy přímo reference samotných ListBoxů by bylo rovněž nemožné, jelikož by bylo nutné pokaždé nastavit jiného „rodiče" - tedy okno, v němž se vyskytuje.

Objektu **ListBoxActualizator**, se tedy za pomoci metody předají reference jednotlivých typů seznamů **ListBox** a **ListBoxActualizator** do všech těchto listboxů v případě volání metody pro aktualizaci umístí vhodné položky – ve všech otevřených dialozích.

Ostatní třídy jsou dialogy, sloužící pro ovládání komponenty, jejich význam je dostatečně popsán v kapitole manulálu a užití.

4 Popis implementace

V této kapitole budou popsány a vysvětleny některé důležité a užitečné úseky zdrojových kódů komponenty. Zároveň také obsahuje užitečné poznatky na co si dávat pozor při používání DirectX. Vzhledem k rozsáhlosti celé komponenty byly některé výše zmíněné třídy a funkce vynechány.

4.1 ShowDev

4.1.1 Správná inicializace grafické karty

Na následující části zdrojového kódu je vysvětlena správná inicializace grafické karty s vysvětleními významu jednotlivých kroků:

```
public void InitializeDevice()            // metoda pro inicializaci grafické karty
        {
            PresentParameters presentParams = new PresentParameters();     // třída s nastavením pro zobrazení
            presentParams.Windowed = true;                                 // komponenta poběží v okně
            presentParams.SwapEffect = SwapEffect.Discard;                 // nastavení bufferu

            //if(set!=null)presentParams.MultiSample = set.MultiSampling;  //  pokud  je  aktivní,  budou  vyhlazovány  hrany,  nemusí
podporovat GPU
            presentParams.AutoDepthStencilFormat = DepthFormat.D24S8;   //24bitů pro hloubku bufferu vzdálenosti objektů, 8bit pro stiny
            presentParams.EnableAutoDepthStencil = true;                   // zapne buffer pro vykreslování objektů ve správném pořadí
            if (!init) device = new Microsoft.DirectX.Direct3D.Device(0,   // pokud nebyla GPU ještě inicializována, inicializuje se
                      Microsoft.DirectX.Direct3D.DeviceType.Hardware, this, // nastavení HW akcelerace
                      CreateFlags.HardwareVertexProcessing,                // jednotlivé vertexy bude rovněž počítat GPU
                      presentParams);                                      // vložení nastavení zobrazení
            device.RenderState.CullMode = Cull.None;                       // chceme "oboustranné stěny"
            device.Transform.World = Matrix.Identity;                      // nastavení matice objektů v prostoru
            device.Transform.Projection = Matrix.PerspectiveFovLH((float)Math.PI / 4, // nastavení projekční matice - perspektiva
                      (float)this.Width / this.Height, 0.5f, 1000f);
            device.Transform.View = Matrix.LookAtLH(new Vector3(x, y, z),  // nastavení matice pohledu
                      new Vector3(x + (float)Math.Cos(azimut),
                      y + (float)Math.Sin(azimut),
                      z + (float)Math.Sin(zenit)), new Vector3(0, 0, 1));
            device.RenderState.Lighting = true; device.RenderState.NormalizeNormals = true;     // aktivace světel, normalizování normál
            init = true;                                                   // inicializace dokončena
            //FX = Effect.FromString(device, HLSL.GetData(), null, ShaderFlags.None, null);
            //FX = Effect.FromFile(device, "x.fx", null, ShaderFlags.None, null);
        }
```

Důležité je především nastavení velikosti **ZBuffer**u, zde pod názvem **AutoDepthStencilFormat** na 24bitový. Implicitní 16bitový nedostačuje a v zobrazování pak dochází k „prosakování" objektů ležících blízko za sebou. V komentáři zdrojového kódu se ještě pojednává o 8bitovém bufferu pro stíny, bohužel ale funkce pro generování stínů objektů nebyla dokončena.

K inicializaci ještě patří blok, umístěný do metody **Repair**(). Tyto příkazy byly od inicializace odděleny, jelikož je potřeba provést tyto příkazy v případě změny velikosti okna nebo po minimalizaci. V případě jednodušších aplikací je tedy možné je umístit přímo do inicializace. Samozřejmě by bylo možné vykonávat při každém zvětšení nebo zmenšení okna celou inicializaci znovu, bylo by to ale časově neúsporné a „trhané".

Tato procedura je umístěna v bloku try-catch, jelikož v případě, kdy by naskočil spořič, nebo došlo k jiné neočekáváné výjimce (např. na grafické kartě), je nutné opět inicializovat zobrazení znovu. Před inicializací je nastavená krátká prodleva, aby nedošlo

k zacyklení příliš rychle po sobě jdoucích pokusů o inicializaci, končící bez žádaného výsledku.

```
Repair(){
try
            {
                device.Transform.Projection = Matrix.PerspectiveFovLH((float)Math.PI / 4,       //  opět  nastavit  projekční  matici  -
perspektiva
                     (float)this.Width  /  this.Height,  0.5f,  1000f);       // úhly určují ohnisko, dále nejbližší a nejvzdálenější
vykreslovaný bod
                device.Transform.World = Matrix.Identity;              // matice umístění objektů ve světě
                device.RenderState.Lighting = true; device.RenderState.NormalizeNormals = true; //   inicializace   světel,   normalizace
normál
                device.RenderState.CullMode = Cull.None;                                        // oboustranné stěny
                device.SamplerState[0].MagFilter = TextureFilter.Linear;                        //   nastavení   lineárního   filtrování
textur
                device.SamplerState[0].MinFilter = TextureFilter.Linear;                        // jinak by bylo jejich zobrazení na
kratší
                device.SamplerState[0].MipFilter = TextureFilter.Linear;                        // vzdálenost „rozpixelované“
                device.RenderState.ShadeMode = ShadeMode.Phong;                                 // metoda stínování podle normál

            }
            catch (NullReferenceException e)                    // v případě, že došlo k chybě a tedy k deinicializaci device -
zobrazení
            {
                device = null;                                  // zrušíme instanci device
                init = false;                                   // odnastavit příznak inicializace
                System.Threading.Thread.Sleep(2000);            // malá časová prodleba 2s
                InitializeDevice();                             // pokus o opětovnou inicializaci
                text = new Gtext(device);                       // reinicializace zobrazování textu na prostor vykreslování
                Repair();                                       // a zbytek inicializace a zároveň opakování pokusu v případě neúspěchu
            }
}
```

4.1.2 Vykreslování

Samotná vykreslovací smyčka se vykonává dle nastavaného **Timer**u každých 50ms, tedy rychlost zobrazování je 20 snímků za vteřinu. Ve smyčce jsou nejprve inicializována „globální“ světla (aby objekty při vypnutém lokálním osvětlení v místnostech nebyly černé), poté je nastaven pohled kamery objektem třídy **Camera** (popsán v další části) a transformací pohledové matice (**device**.**Transform**.**View**).

V DirectX je klíčovým prvkem správné nastavení všech zobrazovacích matic. Pro správné zobrazení je nutné nastavit matici **View**. Do této matice je možné uložit kombinaci matic (násbením): perspektivu, rotaci, posuv a všemožné další operace vyjádřené maticemi 4x4. Pro umisťování objektů je potom nutné nastavit matici **World**, nebo je možné objekty počítat přímo do prostoru. (Více v sekci grafické objekty.)

Po inicializaci světel a pohledu začíná samotný vykreslovací blok. Nejprve je smazána obrazovka (metoda **Clear**), poté je inicializován začátek scény (BeginScene). Vykreslování probíhá po patrech. První jsou vykresleny objekty statické (skříně, stoly, postel), poté objekty akční, zóny (pokud jsou zobrazeny) a nakonec stěny. Toto pořadí má svůj účel – tím je problematika vykreslování průhledných objektů (vysvětlena dále v textu). DirectX má v sobě implementovaný prostorový buffer – **Zbuffer**. V případě aktivace tohoto bufferu (což bylo provedeno a vysvětleno v inicializaci) je možné objekty vykreslovat neuspořádaně, aniž by docházelo k objevování objektů z pozadí scény v popředí a naopak. Pokud buffer nepoužijeme budou v popředí objekty nejpozději vykreslené.

Problémem je, že tento **Zbuffer** nepočítá s možností objevení průhledných objektů ve scéně. Proto je nutné **Zbuffer** v částech vykreslování průhledných objektů

vypnout a tyto objekty kreslit až nakonec. V případě, že by byly kresleny na začátku, staly by se neprůhlednými, respektive objekty za nimi by nebyly vidět, ikdyž by to jejich světelná propustnost měla dovolovat.

Po vykreslení všech pater následuje vykreslení případné cesty, varování v prostoru, jednotlivých „herců“ (třída **Actors** – mohou být vozíčky, lidé apod.) a nakonec textu (nápověda, status, apod.). Následuje už jen ukončení scény (metoda **EndScene**) a voláním metody **Present**() je vše vykresleno do oblasti panelu.

Bylo počítáno i s užitím na velmi pomalém HW nebo s přetížením zobrazení spoustou modelů – v tomto případě se metoda Render volá pouze tak často, jak to dovolují systémové prostředky PC. Toho bylo docíleno za použití semaforu v podobě boolovské proměnné – pokud by byla vykreslovací metoda volaná v době, kdy ještě není vykreslená scéna, nebude se vykreslovat a ani čekat na uvolnění této metody a snímek se vynechá.

4.2 Camera

Nejpodstatnější metodou z třídy Camera je **SetTrack**. Tato metoda je v různých modifikacích vstupů vždy volána při požadavku na animovaný přesun kamery. Nejprve je vektorově vypočtena přímka, vzniklá z rozdílu vektorů udávajících body odkud a kam se bude kamera přesunovat. Tato přímka je implicitně rozdělena na 40 elementů (což odpovídá 2s pohybu). Obdobný postup následuje i pro úhly pohledu kamery. Ty jsou nejprve normalizovány, jelikož po nastavování pohledů myší to mohou být téměř jakékoliv hodnoty (periodicita konstanty PI). Po té následuje výpočet směru pohybu (doprava/doleva, nahoru/dolu), tak aby se pohled nenatáčel vzdálenější stranou (nikdy se nebude otáčet o víc jak 180 stupňů). Rozdíl úhlů je rovněž rozdělen na 40 dílčích příspěvků. Tyto elementy jsou pak při každém volání funkce GetActPos přičteny k aktuální pozici kamery v prostoru a tím je docíleno pohybu.

```
class Camera
    {
        public bool Tracking = false;         // je-li true, pohyb kamery je aktivní
        Vector3 From, To, Phase;              // uchování kde kamera je, kam putuje, o jaké příspěvky
        float Azimut, Zenit, phA, phZ;        // úhly azimut a zenit aktuální a jejich příspěvky (phA, phZ)
        int elements = 40, actual;            // celkový počet elementů (40 = 2spohybu) a aktuální fáze (actual)

        public void SetTrack(Vector3 from, Vector3 to, Vector2 fromAngle, Vector2 toAngle)  // nastavení cesty kamery
        {
            From = from;             // odkud se přesunuje
            To = to;                 // kam se přesumuje
            Phase.Add(to);           // vnejprve vypočíat fázi – nastaví Phase = to
            Phase.Subtract(from);    // od fáze pohybu odečté odkud se letí
            Phase.X /= elements;     // a fázi vydělí počty elementů, nejprve X
            Phase.Y /= elements;     // poté dělení v ose Y
            Phase.Z /= elements;     // a nakonec v ose Z
            actual = elements;       // proměnnou actual, udávající kolik kroků zbývá nastavit na počet elementů
            Normalize(ref fromAngle);         // zde se normalizuje úhel, aby ležel na intervalu od -pi do pi
            Normalize(ref toAngle);           // stejná operace s cílovým úhlem
            if (from == to && fromAngle == toAngle) return;      // pokud se úhly rovnají i cílové souřadnice, nebude
                                              se dít nic, tato operace může být až po normalizaci úhlů (periodicita)
            Zenit = fromAngle.Y;     // přiřazení aktuálních úhlů pohledu kamery
            Azimut = fromAngle.X;
            phA = SetAnglePhase(toAngle.X, Azimut);     // určí směr otáčení a příspěvek (aby se netočil vzdálenější
            phZ = SetAnglePhase(toAngle.Y, Zenit);      // stranou) pro azimut a zenit
            Tracking = true;         // nastavení příznaku pohybu kamery
        }
```

```
        // (...)

        float SetAnglePhase(float to, float frm)       // výše volaná metoda nastavení jednho příspěvku úhlu
        {
            float ph=0;                                // inicializace proměnné
            ph = Math.Abs(to - frm);                   // absolutní rozdíl úhlů
            if (ph > Math.PI)                          // pokud je větší než pi
            {
                ph -= (float)Math.PI;          // je nutné pi od rozdílu odečíst (není třeba víc než 180 stupňů)
                if (to < frm)                  // pokud je úhel do které ho se natáčí kamera menší
                    return ph / elements;      // bude příspěvek kladný (nahoru pro zenit nebo doleva pro azimut)

                    return -ph / elements;     // jinak bude záporný (dolů pro zenit nebo vpravo pro azimut)
            }
            else                               // v opačném případě není nutné PI odečítat a podmínky budou obráceně
            {
                if (to < frm)
                    return -ph / elements;

                    return ph / elements;
            }
        }
         //(...)
}
```

V renderovací metodě třídy **ShowDev** je vždy volána metoda **GetActPos**(...). V případě inicializace změny polohy umístění kamery (některou z metod) tato metoda vrací nové (aktuální) hodnoty umístění kamery v prostoru. Přesun kamery je nastaven na trvání 2 vteřin, aby změny pohledů nebyly příliš trhané a uživatel mohl získat lepší představu a prostorový dojem z vizualizace. Nastavení doby přesunu lze nastavit změnou konstanty **elements**, ta je implicitně nastavena na 40 (zobrazení se překresluje 20 krát za sekundu) a vyjadřuje rovněž počet změn pozic kamery během změny pohledu.

```
public void GetActPos(out float x, out float y, out float z, out float azimut, out float zenit)
        {
            From.Add(Phase); actual--;         // přičte jeden pohybový element k aktuální pozici, a sníží počet
            Azimut += phA; Zenit += phZ;       // přičte jeden úhlový element k úlhům natočení
            if (actual == 1) Tracking = false;          // pokud se jedná o poslední element, zrušti příznak
            x = From.X; y = From.Y; z = From.Z;         // nastaví aktuální pozici kamery (navrácené hodnoty)

            azimut = Azimut; zenit = Zenit;             // nastaví úhel pohledu
        }
```

Důležitým prvkem této třídy je natočení na objekt dle zadaného názvu. Při volání této funkce předá **MapViewer** třídě **Camera** souřadnice předmětu daného názvu v prostoru. Pomocí analytické geometrie se z bodu aktuální pozice kamery a cílové pozice vypočte úhel natočení, případně nový bod pro umístění kamery ležící na cestě k objektu. Tato funkce byla impementována ve 3 modifikacích – pouhé natočení na objekt, natočení a přiblížení k objektu a pohled na objekt z určené vzdálenosti a úhlu pohledu. Po určení nových úhlů natočení, popřípadě i nové pozice kamery, se volá opět metoda SetTrack a kamera se začne animovaně přesouvat.

4.3 Ukládání dat

4.3.1 Library

Ukládání

Vzhledem k serializovatelnosti všech objektů lze tedy ukládat do souboru pomocí nativní třídy C # - **BinaryFormatter**.

```
public void SaveMe()
        {

            Stream stream = File.Open("mapview.conf", FileMode.Create);   // vytvoření nebo přepsání souboru
            IFormatter bin = new BinaryFormatter();                       // objekt realizující serializaci
            bin.Serialize(stream, this);               // převod a uložení do vytvořeného souboru této třídy
            stream.Close();                            // zavření streamu / souboru

        }
```

Získávání objektů

Jak již bylo zmíněno v kapitole popisu řešení – modely předmětů a grafiky, vyskytující se na scéně nejsou načítány pro každý předmět na scéně zvlášť, nýbrž jednotlivé typy předmětů jsou načteny pouze jednou při inicializaci celé komponenty a stejným typům předmětů je jen předána reference na umístění tohoto modelu v operační paměti. Pro tento účel mají grafické objekty implementovanou funkci **Clone**() - tato funkce vytváří nový grafický objekt, do něhož vloží referenci na texturu a model tohoto objektu.

Právě tuto metodu funkci volá třída Library v případě požadavku od třídy Convertor nebo MapViewer na vygenerování objektu určitého typu. Třída Convertor získává z Library předměty na základě jejich barev bitmapě s půdorysem. Třída MapViewer získává objekty na základě názvu jejich typu (zpravidla požadavek uživatele na vložení nového objektu při běhu komponenty).

V Library jsou grafické objekty uchovány za pomoci dvou datových struktur slovník (Dictionary). Jeden překládá názvy na samotné objekty a druhý barvy, jimiž jsou reprezentovány v bitmapách, na názvy.

```
public GObject GetByColor(Color c)            // metoda třídy Library pro získání grafického objektu dle barvy
        {
            String s; GObject x;
            gobjectsColorDict.TryGetValue(c, out s);    // pokus o získání názvu objektu dané barvy pomocí slovníku
            gobjectNameDist.TryGetValue(s,out x);       // pokus o získání samotného objektu dle názvu
            return x.Clone();                  // v případě úspěchu navrátí klon objektu, jinak null
        }
```

4.3.2 Data, Logger – ukádání scén

Pro ukládání třídy **Data** a seznamu **Log**ů z **Logger**u bylo nutné vyvinout rovněž metodu pro klonování, tentokráte samotné třídy **Data** – jelikož v případě předání třídy by se předali i stejné reference na grafické objekty v paměti a tedy funkce **Snapshot** pro uchování „klíčového snímku“ scény by ztratila svůj význam. Při změně vlastností z některého objektu by se změnily vlastnosti ve všech zaznamenaných časových okamžicích.

Vzhledem k tomu, že se neukládají modely, což by bylo i velmi neúsporné na paměť, bylo nutné doimplementovat současně metodu pro předání referencí na aktuálně načtené typy grafických objektů. Při nahrání logu ze souboru se tato metoda volá pro

všechny „klíčové snímky“ a všem grafickým objektům daných typů jsou předány reference na jejich modely.

V době přehrávání se pak při výskytu snímku pouhým předáním reference na „klíčový snímek“ vymění celý datový set ve třídě **Data** a to již není prakticky vůbec HW ani paměťově náročné.

4.4 Logger

Princip loggování spočívá ve volání metody **AddLog(Log log)** komponentou v každém okamžiku užití libovolné ovládací metody komponenty, pokud je aktivován příznak pro nahrávání (**InsertActor** – vložení herce, **SetObjectParameters** – změna parametrů objektu, apod.)

Loggování události vložení nového herce tedy vypadá takto
:

```
logger.AddLog(3, name, null, new Object[] {type, x, y, floor, angle, c, alpha });
```

První parametr udává o jakou událost se jedná (3 odpovídá vložení nebo odebrání herce), druhý parametr název objektu, další parametr udává stav minulý (null – neexistoval) a poslední parametr udává stav aktuální – tedy například jakého je herec typu (Human – člověk, Car – vozíčkář, apod.), kde se nachází (x, y, floor, angle) a barvu s případnou průhledností (c, alpha).

4.5 Grafické objekty

4.5.1 Wall - zdi

Konstrukce stěn

Jelikož základní stavební prvek DirectX je trojúhelník, je nutné obrys stěny určený body, vygenerovaný z bitmapy, převést na model složený z trojúhelníků. Každý trojúhelník reprezentující část modelu je tvořen třemi body – v DirectX vertexy. Jelikož bylo žádoucí vykreslovat v domě osvětlení a mít stěny zdí texturované, bylo nutné kromě pozic bodů stěn v prostoru, ještě rovněž dopočítat normály (pro dopočítávání osvětlení) a koordináty textur na ploše stěny. Pro tento účel bylo tedy užito struktury DirectX – CustomVertex.PositionNormalTextured. Každý trojúhelník je tedy tvořen třemi těmito vertexy. Každý tento vertex je tvořen 8 souřadnicemi – 3 souřadnice umístění v prostoru, 3 souřadnice pro normálu a 2 souřadnice pro koordináty textury na elementárním trojúhelníku.

Nevhodné určení normál může mít za následek nedokonalosti ve stínování objektů – například se model rozsvítí, když na něj bude dopadat světlo z druhé strany, nebo se bude zdát podivně ohnutý.

Nevhodné určení koordinátů pro textury má za následek vykreslení textur všemožně zkosených, roztažených nebo převrácených.

Vykreslování, průhlednost

Vykreslovat do **Device** v DirectX lze mnohými způsoby. V této implementaci byly užity tři z těchto způsobů. Vykreslení uživatelských dat vygenereovaných ze struktur **CustomVertex.PositionColored** (pro objekty jako je označení, vykreslení cesty, varování) a **CustomVertex.PositionNormalTextured** (pro zdi, okna) a vykreslování pomocí objektu třídy **Mesh** z knihovny Direct3DX (extended Direct3D).

Dvě datové struktury (**PositionColored** a **PositionNormalTextured**) lze pak vykreslovat buď přímo za pomoci metody objektu **Device** – **DrawUserDataPrimitives** nebo za pomoci indexace.

Pro vykreslování zdí je užito metody **DrawUserPrimitives**.

```
if(walls != null) d.DrawUserPrimitives(PrimitiveType.TriangleList, walls.Length / 3, walls);
// walls – pole verteců CustomVertex.PositionNormalTextured, vykesluje se počet trojúhelníků walls.Length / 3
```

V hlavní vykreslovací metodě je rovněž implementována možnost vykreslování průhledných stěn. V případě aktivace příznaku pro vykreslování průhledných stěn je nutné vypnout **ZBufferWrite**. V případně ponechání aktivního zápisu do **ZBufferu** by totiž došlo k označení míst výskytu stěn v ploše zobrazení a při vykreslování předmětů do místonsti by pak části předmětů, nacházejících se z pohledu pozorovatele za stěnami, nebyly vidět.

Proto (jak již bylo popsáno v sekci ShowDev) je nutné průhledná tělesa s vypnutým zápisem do ZBufferu vykreslovat ihned po sobě.

Úsek kódu z vykreslovací smyčky:

```
 if (data.floors[aa].objects != null)                          // pokud jsou v data objekty k vykreslení
         for (int a = 0; a < data.floors[aa].objects.Length; a++)   // všechny vykresli
                 data.floors[aa].objects[a].Render(device);
if (data.floors[aa].action != null)                            // pokud jsou v data akční objekty
         for (int a = 0; a < data.floors[aa].action.Length; a++)   // všechny vykresli
                 data.floors[aa].action[a].Render(device);
if (data.floors[aa].walls != null)                             // stěny již mohou být průhledné
         for (int a = 0; a < data.floors[aa].walls.Length; a++)    // kreslí se tedy společně s průhlednými
                 data.floors[aa].walls[a].Render(device);          // jako okna a zóny
if(ShowZones) if (data.floors[aa].zones != null)
         for (int a = 0; a < data.floors[aa].zones.Length; a++)
                 data.floors[aa].zones[a].Render(device);
if (data.floors[aa].windows != null)                           // průhledná okna nakonec, ikdyž
         for (int a = 0; a < data.floors[aa].windows.Length; a++)  // jsou poděděny od akčních objektů musí
                 data.floors[aa].windows[a].Render(device);        // být kresleny oddéleně (Zbuffer vyp)
```

Konverze na mesh

Vzhledem k potřebě užití některých funkcí třídy Mesh, bylo nutné implementovat koverzi geometrie zdí na objekt třídy Mesh. Následující výpis kódu tuto problematiku nastiňuje.

```
public Mesh GetMesh(Device d)
        {
            //vytvoření objektu mesh s počtem trojúhelníků points.Lengths/3, flag nutné na Managed, formát vertexů a
             //Direct3D Device
 Mesh m = new Mesh(steny.Length / 3, steny.Length, MeshFlags.Managed, CustomVertex.PositionNormalTextured.Format, d);
            short[] indices = new short[steny.Length];  // deklarace pole indexů
            short a;
            for (a = 0; a < steny.Length; a++) { indices[a] = a; }  // indexy se rovnají bodům
            // nastavení vertexů do vertex buffer
            GraphicsStream data = m.VertexBuffer.Lock(0, 0, LockFlags.None); // zamknutí vertex bufferu mesh
            for(a = 0; a < steny.Length; a++) data.Write(steny[a]); // zápis jednotlivých vertexů do proudu
            m.VertexBuffer.Unlock();

            // nastavení indexů do index bufferu
            m.IndexBuffer.SetData(indices, 0, LockFlags.None);


            // nastavení datové struktury pro uchování atributů mesh
            int[] attribBuffer = m.LockAttributeBufferArray(LockFlags.None);
            // všechny trojúhelníky budou jako subset 0
            for (a = 0; a < steny.Length / 3; a++) attribBuffer[a] = 0;

            m.UnlockAttributeBuffer(attribBuffer);  // vložení atributů a odemčení attributového bufferu

            // nastavení atributů pro subset 0
            AttributeRange subset = new AttributeRange();
            subset.AttributeId = 0;                 // ID atributu bude 0
            subset.FaceStart = 0;                   // budeme začínat trojúhelnikem s indexem 0
            subset.FaceCount = steny.Length / 3;    // počet trojúhelníků je počet bodu děleno 3
            subset.VertexCount = steny.Length;      // počet bodů
            subset.VertexStart = 0;                 // budeme začínat prvním bodem

            m.SetAttributeTable(new AttributeRange[] { subset }); // vložení atributů
            return m;
        }
```

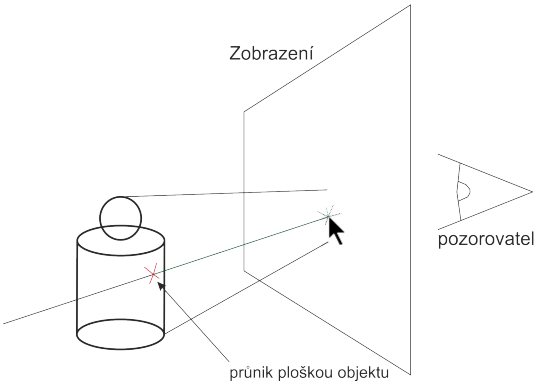
Při konverzi je třeba si dát pozor na **MeshFlags** nastavené na **Managed**. V případě opomenutí tohoto nastavení je při každé změně na zoborazení nutné objekty třídy **Mesh** reinicializovat jinak hrozí chyba zobrazení a ztráta kontroly nad samotnou Direct3D.Device.

WallMesh

Tento objekt se sice nikdy nevykresluje, ale s grafickými objekty velmi úzce souvisí. Vzhledem k využívání metody třídy **Mesh** pro spočtení průsečíku polopřímky s elementární ploškou modelu, bylo nutné vygenerovanou zeď **Wall** převést na objekt typu **Mesh**. Třída **WallMesh** uchovává všechny zdi v podobě objektů třídy Mesh a slouží pro zjišťování nárazu herce do zdi bytu a také pro účely režimu uživatelské editace objektů v prostoru.

4.5.2 Kliknutí do prostoru, náraz a gravitace

Pro účely detekce kliknutí do prostoru byla implementována metoda **MouseHit**. Kliknutí myší do prostoru nelze zjistit žádnou nativní metodou Direct3D.Device (zobrazení), avšak v objektu třídy **Mesh** se nachází metoda Intersect. Intersect navrací zdali došlo k průniku s polopřímkou v prostoru. V případě průniku rovněž navrací podrobnosti o průniku (jako je vzdálenost, pozice v protnutém trojúhelníku). Metoda umí na základě zadané polopřímky vrátit, zdali protnula model uchovávaný v **Mesh** a vzdálenost od počátečního bodu této polopřímky.



Obrázek 25 – klik do prostoru

Metoda **MouseHit**, implementovaná v **GObject**, spočívá ve spočtení průsečíku přímky, projektované od pozorovatele směrem do scény s elementární ploškou (trianglem) modelu zobrazovaného předmětu. Zajišťuje tedy transformování bodu a vektoru z obrazové roviny pozorovatele do prostoru s objekty (viz obrázek 25).

Aby komponenta byla univerzální a snadno rozšiřitelná, všechny modely v mesh leží v počátku souřadného systému zobrazení a mají rozměr v rovině x-y 1x1. Modely jsou teprve až během vlastního vykreslování transformovány do prostoru na žádanou pozici a velikost. Tudíž mimo transformace z obrazového prostoru do prostoru objektů bylo nutné provést i transformace do prostoru každého konkrétního objektu (v kódu matice Trans). Lépe situaci nastíní vybraný úsek ze zdrojového kódu.

```
public virtual bool MouseHit(out float dist, Device device, float posX, float posY)
       {          // posx a posy jsou souřadnice kurzoru myši
           IntersectInformation ii; // informace o průniku
         // vektor near jou souřadnice počátku polopřímky v rovině obrazovky, direction směrnice
           Vector3 near = new Vector3(posX, posY, 0), direction = new Vector3(posX, posY, 1);
         // obrácená transformace přes projekční matici, matici pohledu a matici transformace světa – objektu (Trans)
         // proloží vektor z prostoru zobrazení do objetkového prostoru
           near.Unproject(device.Viewport, device.Transform.Projection, device.Transform.View, Trans);
         // vektory je třeba transformovat do prostoru objektu (který reálně leží v počátku soustavy)
           direction.Unproject(device.Viewport, device.Transform.Projection, device.Transform.View, Trans);
           direction.Subtract(near);          // odečtení transformovaných koordinátů myši - vypočtení směrnice
         // vektor směrnice musel být transformován i se souřadnicemi myši
           if (mesh.Intersect(near, direction, out ii))        // pokud došlo k průniku modelem v mesh
           {
               dist = ii.Dist;               // navrátí vzdálenost nejbližší protnuté plošky
               return true;                  // došlo k protnutí
           }
           dist = 0;               // nedošlo k protnutí nastavit vzdálenost
           return false;           // navrať neprotne
       }
```

Klíčový parametr z navrácené struktury je „dist“. Ten udává vzdálenost nejbližší protnuté plošky. Ve třídě **ShowDev** se potom v případě detekce několika výskytů objektů za sebou vyhodnocuje nejkratší vzdálenost a tento objekt je brán poté za označený.

Obdobně jsou implementovány i metody pro gravitaci a náraz – pouze s rozdílem na vstupu (parametry) a vynecháním transformace z obrazové plochy do prostoru, transformuje se pouze do prostoru modelu.

Metoda pro spočtení nárazu:

```
public bool HitMe(Vector3 where, Vector3 heading, out float dist)
        {
            IntersectInformation ii;
            Vector3 hitDirection = Vector3.Normalize(heading - where);    // určení směrového vektoru
            where.TransformCoordinate(Matrix.Invert(Trans));    // transformace do prostoru objektu inverzní t
                                                                //ransformační maticí
                                                               // transformuje se jako koordinát
            hitDirection.TransformNormal(Matrix.Invert(Trans)); // transformace směrového vektoru, je nutné
                                                                // transformovat jako normálu
            if (hitMesh.Intersect(where, hitDirection, out ii)) // metoda pro spočzení průsečíku
            {
                dist = ii.Dist; // vzdálenost v prostoru objektu -> nutnost transformace do prostoru zobrazení
                hitDirection.Multiply(dist);    // úprava směrového vektoru na správnou vzdálenost z prostoru objektu
                hitDirection.TransformNormal(Trans);    // transformace směrového vektoru do prostoru scény
                dist = hitDirection.Length();   // správná vzdálenost v prostoru scény
                return true;
            }                   // navrátí protnuto a vzdálenost, ta je pak testována zdali není menší
                                                               // než velikost objektu → náraz
            dist = 0;
            return false;           // neprotnuto
        }
```

V konstruktoru objektu je vytvořena – hitMesh – kvádr opsaný kolem předmětu, jelikož se v objektu mohou vyskytovat prázdná místa (například prostor pod stolem) a hledat průsečíky pro celou výšku předmětu by bylo neúsporné.

Metodu pro spočítání nárazu volá objekt třídy **Actor** (herec). Potom jednoduchým porovnáním své šířky v rovině x-y s navrácenou vzdáleností od objektu (dist) rozhoduje zdali do objektu narazil nebo ne.

MapViewer pak při instrukci na změnu polohy voláním metody nárazu herce pak pro dané objekty v patře zjišťuje, zdali do některého nenarazil a v případě nárazu vytvoří událost.

Vykreslování předmětu, označení

V grafických objektech je užito vykreslování tříd **Mesh**. Výhodou tohoto vykreslování je, že **Mesh** má přímo metodu sloužící pro vykreslení sebe sama do **Device** a pro načtení modelu ze skriptového souboru DirectX .x, nevýhodou však je, že model uchovaný v **Mesh** je nedotknutelný a téměř nedostupný pro zásah. Před vykreslením modelu je na rozdíl od vykreslování zdí nutné modely transformovat do prostoru, jelikož na rozdíl od zdí nemají jednotlivé souřadnice tvořící model absolutní hodnoty umístění v prostoru. To je z důvodu již výše zmíněné variability komponenty a možnosti vykreslení libovolně velkého a natočeného předmětu. Modely mají tedy souřadnice v rovině x-y vždy o velikosti 1x1. Následující úsek zdrojového kódu ukazuje správnou implementaci pro vykreslení takového objektu.

Nejprve bude nastíněno nastavení prvotní transformační matice předmětu, aby bylo vykreslování optimální je tato matice spočtena pouze jednou při vkládání objekt do scény nebo při manipulaci z objektem.

```
protected Matrix Trans;
        public virtual void TransMatrix()
        {
            Trans = Matrix.Identity;    // nastavení jednotkové matice
            if (rotation == 0)  // rotation je příznak rotace z Convertoru
            {
                if (zoomx != 0) Trans = Matrix.Scaling(zoomx, zoomy, 1); // škálování na požadovanou velikost
                Trans *= Matrix.Translation(x, y + zoomy, 0);   // translace na požadovanou pozici
            }
            if (rotation == 3) // příznak pro otočení o 90 stupňů vpravo
            {
                Trans = Matrix.RotationZ(-(float)Math.PI / 2); angle = -(float)Math.PI / 2; // matice otočení
                if (zoomx != 0) Trans *= Matrix.Scaling(zoomx, zoomy, 1); // násobená maticí pro škálování
                Trans *= Matrix.Translation(x + zoomx, y + zoomy, 0);     // násobená maticí pro posun
            }
            if (rotation == 2) // příznak pro otočení o 180 stupňů
            {
                Trans = Matrix.RotationZ((float)Math.PI); angle = (float)Math.PI;   // operace jsou stejné
                if (zoomx != 0) Trans *= Matrix.Scaling(zoomx, zoomy, 1);
                Trans *= Matrix.Translation(x + zoomx, y, 0);
            }
            if (rotation == 1) // příznak pro otočení o 90 vlevo
            {
                Trans = Matrix.RotationZ((float)Math.PI / 2); angle = -(float)Math.PI / 2;
                if (zoomx != 0) Trans *= Matrix.Scaling(zoomx, zoomy, 1);
                Trans *= Matrix.Translation(x, y, 0);
            }
            Trans *= Matrix.Translation(0, 0, 50 * floor + GravityElevation); // nastavení umístění v patře a elevace
        }
```

Pořadí jednotlivých transformací nelze prohodit – následkem by bylo například zobrazení zkoseného předmětu nebo by se předmět nacházel na zcela jiném místě.

V případě uživatelské editace objektu je pak matice počítána metodou **SetCustomMatrix**(), jelikož uživatel již může zadat jiný libovolný úhel a navíc si může i zvolit libovolnou výšku objektu.

```
public virtual void SetCustomTransMatrix()
        {
            Trans = Matrix.Scaling(zoomx, zoomy, CustomZscale) *            // vlastní škálování
                Matrix.RotationZ(angle) *                                   // úhel natočení
                Matrix.Translation(x, y, floor * 50 + GravityElevation);    // umístění v prostoru, elevace
        }
```

Následné vykreslování je implementováno v metodě **Render**() volané třídou **ShowDev**, předávající v referenci samotnou Direct3D Device pro vykreslení.

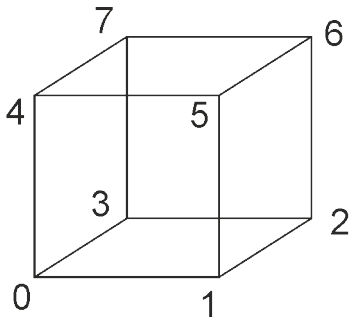
```
public virtual void Render(Device d)
        {
            d.Transform.World = Trans;      // nastavení transformační matice pro vykreslování
            for (int i = 0; i < meshmaterials.Length; i++)  // mesh má tolik vrstev, kolik materiálů
            {
                d.Material = meshmaterials[i];          // nastaví materiál pro vykreslení
                d.SetTexture(0, meshtextures[i]);       // nastaví texturu
                mesh.DrawSubset(i);                     // vykreslí subset z daným materiálem a texturou
            } if (WireSelection) DrawWireSelection(d);  // je li příznak výběru, drátový model krychle
            if(Selection)DrawSelection(d);          // je-li vybrán, vykreslit polopropustnou krychli výběru
            d.Transform.World = Matrix.Identity;    // nastavení matice na jednotkovou matici
        }
```

Metoda je nastavena jako virtuální, aby bylo možno ji v děděných objektech upravovat – **GActionObject** vykresluje více objektů nebo také jiným způsobem (rotace).

Pro vykreslování označení objektu je užito dalšího způsobu vykreslování a to pomocí indexace – metoda **DrawIndexedUserPrimitives**. Výhoda tohoto vykreslování spočívá v možnosti neukládat do paměti stejné body vícekrát (jako je tomu při užití **DrawUserPrimitives**), ale za pomoci indexů, uložených v poli short (v případě užití

16bitových indexů) nebo poli int (v případě 32bit) předat metodě indexy, určujících posloupnosti bodů definující jednotlivé trojúhelníky.

Pro vykreslení označení jsou definovány body typu **CustomVertex.PositionColored** a indexy **short** krychle o rozměrech 1x1xH, transformovány na velikost předmětu. Velikost krychle je vypočtena pomocí implementované metody **Size**(), využívající třídu **Geometry** pro zjištění hranic modelu, uchovaného v **Mesh**.




```
        protected CustomVertex.PositionColored[] selLines;
        protected short[] indices={     // definice indexů
                        0,1,2, // Spodek
                        0,2,3,
                        0,1,5, // Predek

                        0,5,4,
                        4,5,6, // Vrch
                        6,7,4,

                        3,2,6, // Zadni
                        3,6,7,
                        0,3,7, // Leva
                        0,7,4,
                        1,2,6, // Prava
                        1,6,5
                    };
        public bool Selection, WireSelection;   // deklarace příznaklů označení

        void SelectionConstruct()   // metoda pro vytvoření označení (volána při načítání modelu)
        {
            selLines = new CustomVertex.PositionColored[8];     // bude se vykreslovat krychle pomocí indexace, stačí
                                                                // tedy 8 bodů (jinak by jich bylo tříba 36 (12 trianglu)
            int c = Color.FromArgb(128, Color.Red).ToArgb();    // barva bude červená s propustností alpha 50 procent
                                                                // (128 v byte)
            float size = Size();                                // metoda spočte výšku předmětu
            selLines[0] = new CustomVertex.PositionColored(0, 0, 0, c);   // jednotlivé vertexy určují body
            selLines[1] = new CustomVertex.PositionColored(1, 0, 0, c);   // první až třetí parametr jsou souřadnice,
            selLines[2] = new CustomVertex.PositionColored(1, -1, 0, c);   // 4 barva v int
            selLines[3] = new CustomVertex.PositionColored(0, -1, 0, c);
            selLines[4] = new CustomVertex.PositionColored(0, 0, size, c);
            selLines[5] = new CustomVertex.PositionColored(1, 0, size, c);
            selLines[6] = new CustomVertex.PositionColored(1, -1, size, c);
            selLines[7] = new CustomVertex.PositionColored(0, -1, size, c);

        }
```

V případě vykreslování **CustomVertex.PositionColored**, je důležité zrušit nastavení dopočítávání světel ve struktuře **RenderState**, jelikož vertexy tohoto typu nemají žádné normály a v případě vykreslení s dopočítáváním odrazů světel by byly tyto objekty vždy černé. Následující úsek kódu je částí z metody pro vykreslování objektu, sloužící k vykreslení označení objektu.

```
void DrawSelection(Device d)
        {
            d.RenderState.AlphaBlendEnable = true;  // zapnutí alpha kanálu
            d.RenderState.Lighting = false;         // vypnutí počítání světel
            //d.RenderState.ReferenceAlpha = 0x80;
            d.Material = selMat;
            d.RenderState.SourceBlend = Blend.SourceAlpha;  // určení funkce pro výpočet průhlednosti
            d.RenderState.DestinationBlend = Blend.InvSourceAlpha; // to samé pro pozorovatele
            d.VertexFormat = CustomVertex.PositionColored.Format;   // nastavení vykreslovaného formátu vertexů
                // vykreslování indexovaných trojůhelníků, začínáme indexem 0, indexů je 36, stěn 12, zdroj indexů
indices, zdroj bodů selLines
            // true - značí užití 16 bitových indexů
            d.DrawIndexedUserPrimitives(PrimitiveType.TriangleList,0,36,12,indices,true,selLines);
            d.Transform.World = Matrix.Identity;    // vrátíme původní transformační matici
              if (renderCLink) d.DrawUserPrimitives(PrimitiveType.LineList, 1, line); // pokud je objekt propojen s
jiným, vykreslí spoj
            d.RenderState.Lighting = true;          // opět povolí počítání osvětlení
            d.RenderState.AlphaBlendEnable = false; // zakáže průhlednost
        }
```

Takového způsobu vykreslování je rovněž užito ve třídě **GSelect** (pro vykreslení ukazatele sloužícího k výběru umístění v patře) a třídě Warning. Kombinace tohoto způsobu a způsobu vykreslování zdí je využita ve třídě Path. Zbylé třídy poděděné od **GObject** užívají metody stejné nebo velice podobné. Pro názornost bude probrán ještě **GActionObject**, jež obsahuje vykreslování animovaných objektů.

4.5.3 GActionObject

Jak již bylo zmíněno v předešlé kapitole – třída slouží k uchování stavových nebo animovaných objektů a tedy rozšiřuje především metodu **Render**() z **GObject**. Animace byla implementována pomocí 2 proměnných a pole **GObject**ů. První proměnná je typu float a udává rychlost přehrávání animace. Druhá je typu integer a jednduchým výpočtem se do ní ukládá index právě zobrazovaného objektu pole **GObject**ů.

V případě příznaku pro stavové chování objektu se proměnná pro rychlost neužívá a metodou pro nastavování akčních objektů se nastaví přímostav objektu, tedy index pro pole **GObject**[].

Podobným způsobem bylo implementováno i vykreslování zmíněných předmětů tvořených dvěma modely, kde se jedna část točí nebo pulsuje (větrák, reproduktory). V těchto případech se neužívá proměnná stavu, ale pouze proměnná pro rychlost, z níž se dopočítávají transformační matice pro rotaci nebo škálování.

4.5.4 Window

Konstruktor **Window** kombinuje metody **GActionObject** i **GObject**. Třída obsahuje také metodu pro vytvoření **mesh**, ale jelikož se nepočítá s přesunem okna při běhu komponenty, jsou koordináty mesh počítány absolutně, navíc je mesh použita obdobně jako v případě **WallMesh** pouze pro spočtení průsečíku myši s objektem okna. Stejně tak je i upravená metoda pro konstrukcu označení okna (například při otevírání oken klikáním do zobrazení) – hodnoty opsaného kvádru jsou počítány absolutně.

Poděděný **GActionObject** má implicitně nastaven příznak **OnOff** (otevřené / zavřené okno).

Další grafické objekty již v tomto textu popsány nebudou, jelikož se jejich logika a metody vykreslování výše popsaným v drobných nuancích velmi podobají. Ostatní třídy a vnitřní mechanismy komponenty jsou buďto rozsáhlé nebo vzájemně velmi provázané a jejich popsání by bylo velmi komplikované a prostorově náročné. V případě zájmu o jejich implementaci je k dispozici zdrojový kód u autora nebo vedoucího práce.

5 Manuál užití

Pro testování komponenty byl vytvořen jistý simulátor, obsahující funkce pro ovládání herců („Actor“) , prvků prostředí a všech ostatních nastavení a režimů komponenty. Slouží především jako vzorová aplikace pro užití popisované komponenty. Dále v textu bude popsána jeho obsluha, používání editoru komponenty a užití samotné komponenty ve vlastní aplikaci.

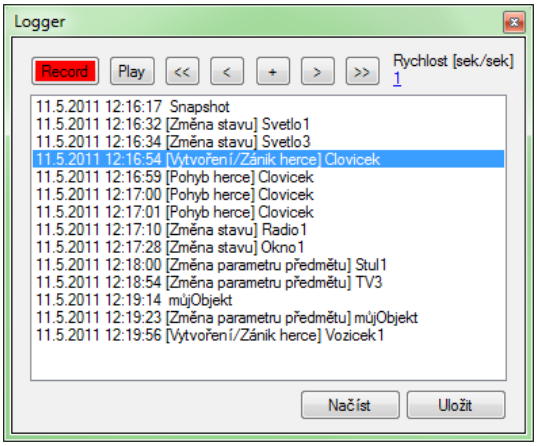
5.1 Simulátor (simulator.exe)

Simulátor je vzorová aplikace užití komponenty. Před použitím komponenty ve vlastní aplikaci je vhodné si projít jeho vzorový kód.

Hlavní okno aplikace obsahuje 6 kontrolních tlačítek. 1. tlačítko „Editor“ spustí editor komponenty. Druhé tlačítko „Spustit komponentu“ otevře komponentu v režimu vlastního okna a aktivuje zbylé tlačítka (na ty do doby spuštění komponenty není možné kliknout).

Po spuštění komponenty je možné otevřít dialogy pro ovládání a globální nastavení komponenty, testování událostí, ovládání kamery a akčních prvků komponenty a také logger.

5.1.1 Dialog „Logger“

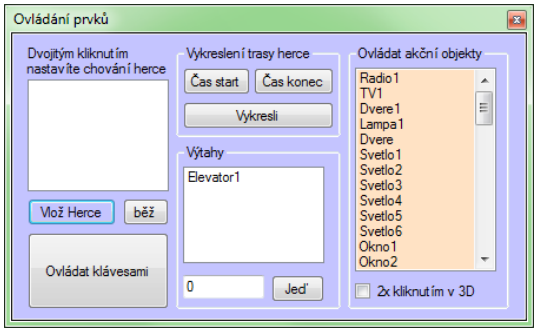


Slouží pro testování loggeru komponenty. V dialogovém okně je možné spustit nahrávání logů. Během práce s komponentou bude, v případě aktivovaného nahrávání, vše zaznamenáno a zobrazeno v seznamu s časovými kódy. Daný log je pak možné přehrát nebo komponentu dvojklikem na položku seznamu uvést do žádaného stavu. Dále pak lze celý záznam uložit do souboru a později opětovně načíst.

5.1.2 Dialog „Ovládat prvky“

Tento dialog slouží pro ukázku nejčastějšího užití komponenty. V první části je možné přidat herce do scény. V případě kliknutí na tlačítko ovládat herce lze všechny

herce ve scéně ovládat nastavenými tlačítky. Při dvojkliku na položku seznamu herců lze roněž nastavit automatické chování herců a otestovat tak schonpost komponenty hlásit náraz herce do stěny. Stiskem klávesy 'd' lze vybraného herce odstranit.



V pravé části dialogu je seznam akčních objektů – v něm je možné dvojklikem na položku změnit stav daného předmětu (například zapnout rádio nebo TV, či rozsvítit). Pod seznamem je zaškrtávací políčko „klik do 3D“. Zaškrnutím tohoto políčka je možné poté měnit stav objektů přímým kliknutím do prostoru, zobrazeného komponentou.

Prostřední část dialogu – „vykreslení trasy herce“ - lze užít jen v případě byl zaznamenán log s pohyby herce. Dvě tlačítka „Čas start“ a „Čas konec“ slouží pro označení časových úseků, od kdy a do jaké doby bude tento záznam prohledáván. Pro zobrazení trasy herce je nutné jej označit v seznamu herců

Poslední část dialogu slouží pro ovládání výtahů ve scéně. Pokud chceme výtah uvést do pohybu, je nutné jej nejprve vybrat v seznamu, poté do pole pod seznamem napsat číslo patra a klikout na tlačítko „Jeď“. V případě, že bude ve výtahu osoba, dostane se do žádaného patra.

Přidat herce

Dialog slouží k umístění herce do scény. Pro základní testování komponenty byly vytvořeny 2 modely – pro každý typ herce jeden – symbolický model člověka a vozíčku. Herci je nutné nadefinovat unikátní jméno a dále možné definovat barvu, patro a kliknutím na tlačítko „Kam“ také umístění v prostoru. Umístění v prostoru je tedy možné po kliknutí na tlačítko „Kam“ a výběru patra kliknutím do prostoru patra. Při tažení myší v nad zobrazením komponenty se v této chvíli ukáže hranol symbolizující pozici v prostoru.

5.1.3 Dialog „Ovládání kamery“

Dialog slouží pro ukázku možností nastavení kamery. První skupinou nastavení je nastavení kamery do libovolné pozice v prostoru, nastavení doby animovaného přesunu kamery a náhled na celou scénu.

Druhá skupina nastavení slouží pro ukázku

možností manuálního ovládání kamery uživatelem. Lze aktivovat režim fixované kamery v dané výšce, kde výšku lze měnit kolečkem myši, režim volného průletu prostorem s nutným stiskem tlačítka myši nebo se zamčeným kurzorem vprostřed zobrazení.

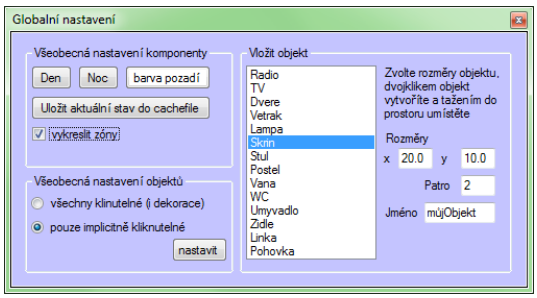
Třetí skupina nastavení ukazuje možnosti automatického chování kamery, kdy kliknutím na libovolný objekt v seznamu objektů na scéně, se kamera daným směrem natočí. V případě zaškrtnutí políčka „Leť k“ pak kamera přiletí k vybranému objektu na vzdálenost uvedenou v textovém poli. V případě zaškrtnutí políčka „Zepředu“ kamera přiletí před předmět z jeho přední strany. Poslední políčko k zaškrtnutí je „Z úhlu“. Po vyplnění dvou textových polí vedle tohoto políčka kamera přiletí k objektu z uvedených úhlů. První úhel značí azimut a lze jej zadat v podstatě libovolný, druhý úhel značí vertikální natočení kamery a tudíž je nutné jej zadat v rozmezí -90 až +90 stupňů (samozřejmě při zadání jiné hodnoty se nic nestane vzhledem k periodicitě funkce sin).

5.1.4 Dialog „Globální nastavení komponenty“

V tomto dialogu je možné přidávat nové objekty na scénu, barvu pozadí komponenty, nastavení dne nebo noci a ukládání aktuálního stavu komponenty. V případě uložení aktuálního stavu komponenty do souboru se bude komponenta po opětovném spuštění nacházet v přesně stejném stavu (se všemi změnami umístění a nastavení předmětů i herců).

Pro zobrazování zón je nutné právě zde zaškrtnout políčko zobrazit zóny – jinak nebudou viditelné a to i v případě stisku klávesy „k“ pro vytvoření nové zóny.

Dále je zde možnost nastavit všechny objekty ve zobrazení na kliknutelné a tudíž umožnit jejich editaci.



5.1.5 Dialog „Testování událostí“

Poslední testovací dialog je velmi jednoduchý. Ukazuje události komponenty a vypisuje jimi navrácené údaje.

5.2 Editor

Editor je přístupný buďto přímo přes vzorovou aplikaci – simulator.exe, nebo jej můžeme umístit přímo do vlastní aplikace vytvořením objektu SysEdit. Tedy:

```
SysEdit sysEdit = new SysEdit();
sysEdit.Show();  nebo  sysEdit.ShowDialog();
```

5.2.1 Vytváření nových modelů

Modely obsažené v této práci byly vytvořeny ve freewarovém programu Blender[10]. Tento program se vyznačuje možností exportu do mnoha souborových formátu včetně DirectX skriptu .x, podporovaného třídou mesh, v níž se uchovává model. Všechny modely byly vytvořeny v rámci této práce, a tudíž se na ně nevztahují žádná cizí autorská práva. Pouze modely pohovky a kuchyňské linky byly převzaty ze stránek s volně stažitelnými modely [5]. Tyto modely byly získány ve formátu 3D Studia MAX .3ds, importovány do Blenderu a přepracovány do aktuální podoby.

Běžný statický objekt

Běžný statický objekt, kterým je například skříň, postel nebo stůl je nutné po domodelování naškálovat na velikost 1x1xH, kde H je výška předmětu. Předmět bude v komponentě naškálován na správnou velikost dle obrázku patra domu / oddělení / bytu, či uživatelským vstupem.

Dalším požadavkem správného vykreslení je poloha

modelu v souřadném systému. Levý dolní okraj modelu by měl být posunut do počátku souřadného systému (0, 0, 0).

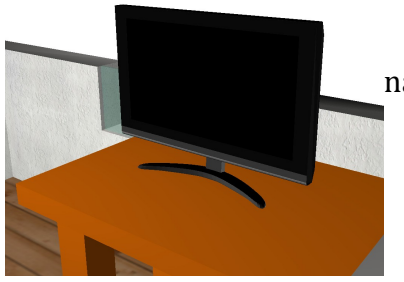
Před exportem modelu je nutné zkontrolovat, zdali jsou správně normály objektu a zdali jsou orientované směrem ven z objektu. V případě špatně nastavených normál může nastat situace, kdy bude objekt úplně černý nebo se na světle bude chovat neobvyklým způsobem. Při exportu je nutné zkontrolovat nastavení „Z-axis up“, tedy osa z směr nahoru a „right-handed system“, jinak hrozí špatné natočení objektu, či vykreslení vzhůru nohama.

Některé struktury DirectX sice nabízejí funkce pro normalizace normál a jejich konstrukci, bohužel ale zobrazení po aplikaci těchto metod už nedosahuje takové kvality, jako správné nastavení normál se zjemňováním, proto tyto metody nebyly v komponentě použity.

Komponenta umí rovněž načíst nastavené materiály, tudíž objekt bude barevně vypadat přesně tak jako v Blenderu.

Akční objekt

Akční objekt se může skládat z jednoho či více objektů. Na rozdíl od běžného statického objektu se akční objekt neškáluje a proto není třeba jej upravovat na specifickou velikost. Přepočet pro objekty k reálným velikostem je 4cm/1.0 float. Tedy reálný objekt o rozměrech 10x10x10cm by měl být 2.5x2.5x2.5 float.



Editor poskytuje všemožné nastavení pro objekty. Například větrák se skládá s točivého elementu (točícím se promněnlivou rychlostí) a pevné základny. Model rádia se rovněž skládá z 2 objektů, v případě jeho zapnutí bude objekt reproduktorů pulzovat. Lepší vysvětlení poskytne část popisující vkládání modelů do editoru.

Před exportem modelu do skriptu .x je důležité zkontrolovat, leží-li jeho střed ve středu souřadného systému (0, 0, 0).

Herec humanoidního typu

Každý objekt humanoidního typu je složen ze dvou objektů – nohy (ta je naklonována na 2 vedle sebe) a těla. V editoru je možné nastavit offset nohou od těla. Doporučený postup je namodelovat celého člověka a rozdělit model do dvou souborů. Model trupu člověk pak nechat ve stejné výšce, jako s nohami, pouze jej umístit do horizontálního středu soustavy x=0, y=0. Model nohy do druhého souboru poté umístit tak aby kyčel byla ve středu souřadného systému.

Herec – vozíček

Postup vytváření je obdobný jako v případě vytváření modelu člověka, pouze s tím rozdílem. Že model kolečka bude kolem vozíku nakopírován 4x. Kolečko je rovněž nutné umístit středem do počátku souřadného systému a model vozíku (například i s modelem člověka) ponechat ve stejné výšce, pouze jeho střed umístit do x = 0, y=0.

5.2.2 Vytváření plánů pater a zón

Plány pater lze vytvářet v obyčejném malování pro Windows (mspaint.exe), jelikož se jedná o obyčejné bitmapy. Pozor na barevné palety mspaint ve Windows Vista a Windows 7, palety byly upraveny a barvy o pár tónů povětšinou zesvětleny, proto uvádím u základních barev i RGB souřadnice barevného prostoru, v těchto verzích mspaint už se tyto základní barvy nevyskytují a je nutné si je namíchat.

Každé patro se skládá ze 2 souborů – půdorysu s objekty a půdorysu podlah. Pro základní entity jsou vyhrazeny tyto barvy:
černá (RGB: 0, 0, 0) – pro stěny
šedá (RGB: 192, 192, 192) – pro okna
bílá (RGB: 255, 255, 255) – prázdné místo
červená (RGB: 255, 0, 0) – vstupní dveře
žlutá (RGB 255, 255, 0) – světla v místnostech

Ostatní barvy lze použít na libovolné statické a akční objekty.

V bitmapě plánu podlah je rezervovaná pouze jedna barva – bílá (RGB 255, 255, 255). V místech, kde se vyskytuje bílá barva nebude podlaha žádná. Ostatní barvy je možné použít na různé druhy podlah. Textury pro tyto podlahy a barevné přiřazení lze v editoru nastavit.

Dále z bitmap mohou být do komponenty importovány také zóny a to v několika vrstvách. Pro tyto účely je opět rezervována bílá a ostatní barvy budou přiřazeny jednotlivým zónám (pouze pro estetický dojem).

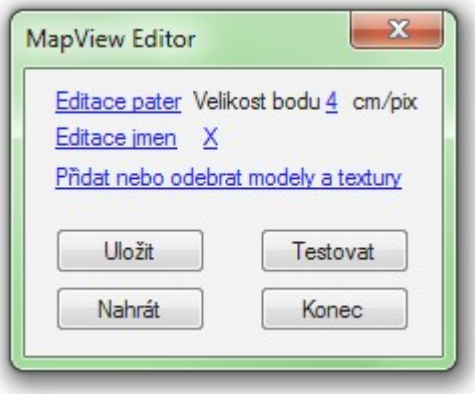
Soubory s mapami lze z důvodů úspory místa možné uložit v libovolném bezeztrátovém formátu: například GIF nebo PNG, komponenta je umí načíst také.

5.2.3 Práce s editorem

Editor slouží k editaci hlavního konfiguračního souboru komponenty „mapview.res“, kde jsou nastaveny lokace a názvy souborů s modely, nákresy pater, zóny a vlastnosti jednotlivých objektů.

Hlavní okno

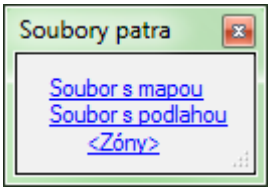
Na hlavním okně editoru si můžeme všimnout 4 základních tlačítek. „Načíst ze souboru“, „Uložit do souboru“, „Testovat“ a „Konec“. V případě, že chceme komponentu pouze editovat, pozměnit, přidat nebo odebrat objekty, je nutné nejprve kliknout na tlačítko „Načíst ze souboru“. V opačném případě se naskýtá možnost vytvořit konfigurační soubor komponenty od začátku.

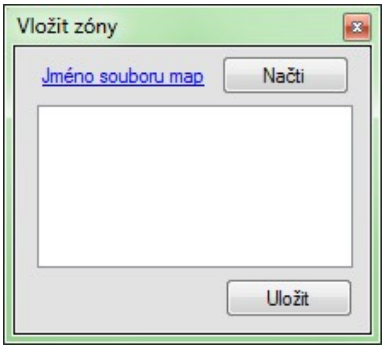


Dalšími prvky hlavního okna jsou „Editace pater“ (nastavení souborů s nákresy pater), „Editace jmen“ (pro nastavení jiných jmen objektům), „X“, nastavení rozlišení výkresů a „Přidat nebo odebrat modely a textury“.

Editace pater

Dialog „Upravit patra“ slouží k přidávání, odebírání a editaci názvů pater a importu zón zájmu ze souboru. K přidání a mazání položek slouží odkazy „přidat a odstranit“. Odkaz „odstranit“ odstraní aktuálně vybranou položku v seznamu. Pat bude komponenta vykreslovat dle pořadí položek seznamu od shora dolů. Tedy první položka seznamu bude nulté patro v zobrazení. Pro změnu patra slouží 2 šipky v pravém horním rohu dialogu „v“ a „^“.

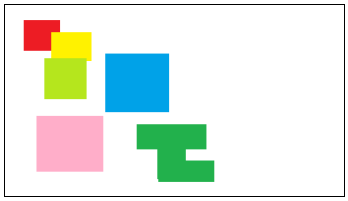


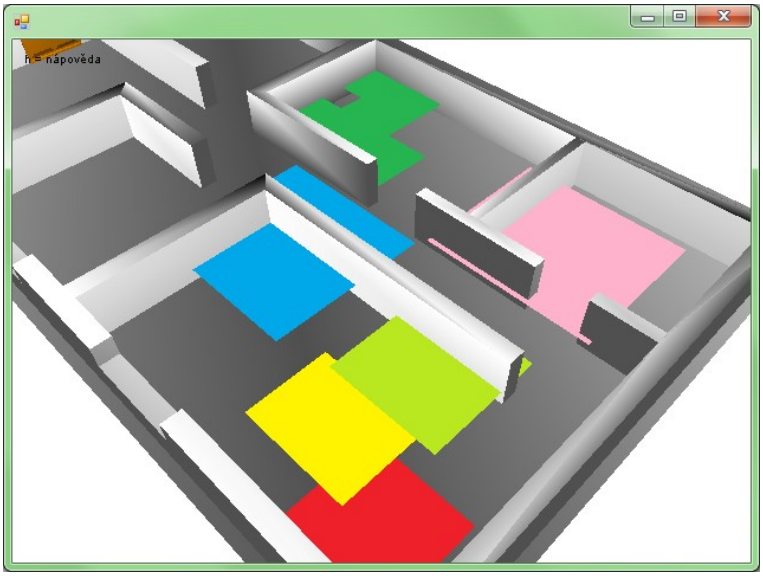


Při dvojkliku na vybranou položku v seznamu je možné zobrazit podrobnosti o názvech souborů patra a editaci zón. Po kliknutí na odkaz „přidat“ se objeví dialogové okno „Soubory patra“. Zde je možné nastavit jména pater dvěma způsoby. Kliknutím levého tlačítka myši na odkaz se objeví dialog s textovým vstupem, kde je možné přímo napsat název souboru (případně s cestou). Kliknutím pravého tlačítka na odkaz se otevře klasický dialog s možností procházení souborů a složek a v něm žádaný soubor vybrat.

Těmito způsoby lze vybrat bitmapu s půdorysem a bitmapu s nákresem podlah. Soubory mohou být v jakémkoliv bezztrátovém formátu (PNG, GIF, BMP).

Poslední položkou dialogu je odkaz „<Zóny>“. Ten nás odkáže na další dialog - „Vložit zóny“ daného patra. Před spuštěním tohoto dialogu je nutné aby uživatel nastavil názvy souborů s půdorysem patra a podlah, jelikož se společně s tímto dialogem otevře ještě náhledové okno s modelem daného patra (viz. obrázek). Nastavení zón samozřejmě není pro funkčnost komponenty klíčové a je možné zóny kdykoliv během používání komponenty přidat.





Obrázek 26 – pohled na vygenerované zóny z bitmapy (vlevo)

Kliknutím na položku „Jméno souboru map“ je možné obdobným způsobem jako u nastavení názvu souboru pater nastavit název souboru, z kterého budou načteny jednotlivé zóny. Zóny se vypíší poté od seznamu a v náhledovém okně komponenty se pak vykreslí stejnými barvami, jako měli v souboru pro import. Tento postup je možné několikrát opakovat a vložit tak více vrstev zón.

Názvy jednotlivých zón lze změnit dvojklikem na jednotlivé položky seznamu. Rovněž jim lze nastavit jiné barvy (což je využito pouze pro estetický dojem).

Editace jmen

Komponenta automaticky pojmenovává objekty názvy odvozených z jejich typu s číslem pořadí. Objekty jsou číslovány zleva doprava a shora dolů. V případě nutnosti změnit název těchto objektů tedy slouží okno „Editace jmen“, kde je možné implicitní názvy objektů změnit. Změnit názvy objektů je rovněž možné metodami komponenty nebo v režimu testování komponenty (vysvětleno níže).

Přidat nebo odebrat modely a textury

Otevře dialog „Nastavení objektů komponenty“. Zde je možné přidávat, editovat nebo mazat jednotlivé předměty a objekty zobrazované komponentou. Označenou položku v každém seznamu lze mazat stiskem klávesy „d“ nebo editovat dvojklikem na položku. Pod každým seznamem je tlačítko pro přidání nového objektu. Vkládat lze různé typy položek – statické objekty (skříně, poličky, postele, kuchyňská linka, umyvadlo), akční objekty (objekty vypnutelné nebo regulovatelné – TV, větrák, hifi soustava, rolety u oken, …), textury podlah a modely herce.

Vložit objekt nebo akční objekt

Jde o dialog zobrazený po kliknutí na tlačítko „Vložit objekt“ nebo „Vložit akční objekt“. V případě požadavku na vytvoření běžného objektu se dialog zobrazí bez některých nastavení. Je nutné zadat název předmětu (typ), jeho barvu v nákresu půdorysu a název souboru nebo cestu k jeho modelu kliknutím na tlačítko „<MESH>“.

Dále lze nastavit příznaky: „Clickable“, „Controllable“ a „Controller“. Příznak „Clickable“ nastavuje, zdali objekt ve zobrazení bude kliknutelný a označitelný. Příznak „Controlable“ určuje, zdali je objekt možné ovládat jiným objektem (například vypínačem) a příznak „Controller“ určuje, zdali se jedná o vypínač/ovladač.

Je důležité, aby název objektu a barva objektu byly unikátní. Potom už stačí jen kliknout na tlačítko vložit.

Pozn: jelikož jsem zdrojový kód komponenty psal v angličtině, je toto okno v angličtině, aby bylo jasné jaké prvky třídy GObject (respektive GActionObject) se nastavují.

Vložit model herce

Slouží ke stejnému účelu jako předešlý dialog. Tedy pro vložení nového herce. Je nutné nastavit název „Name“ (například „Člověk“, „Žena“, „Muž“, „ET“). A po té jméno souboru nebo cestu k modelu, kliknutím na první tlačítko „<MESH>“. Nakonec příznak nastavení zdali se jedná o vozík nebo člověka a cestu na model nohy/ kola.

Herec může být buď humanoidního typu – v tom případě bude model nohy zobrazen 2 krát vedle sebe nebo model kola – model bude zobrazen 4x (2 přední a 2 zadní kola).

Podle tohoto nastavení „Positions“ se určuje, kde v prostoru bude noha napojena na tělo – první řádek (x, y, z) koordináty 1. nohy a druhý řádek koordináty druhé nohy.

Ostatní objekty

Poslední položkou je dialog nastavení pro ostatní objekty, jako je model výtahu, model pro varování, základní textura země a textura zdi.

Režim testování

Umožňuje nahlédnout na model vygenerovaný s použitím aktuálně nastaveného konfiguračního souboru, popřípadě provést drobné úpravy v modelu bytu. Komponentu v tomto režimu je možné ovládat klávesami. Tyto klávesy je možné zobrazit stisknutím písmene / klávesy „h“. Nyní budou probrány jednotlivé režimy a příkazy.

Možnost editace je povolena i v simulátoru, aby bylo komponentu možné dobře otestovat a nebylo nutné spouštět editor. Samozřejmě při užití komponenty v novém čistém projektu má implicitně režim editací blokovaný (viz další kapitola – příkazy).

Klávesy pro ovládání kamery a pohyb v prostoru

„c“ - slouží pro přepínání mezi různými způsoby ovládání kamery. Při prvním stisku klávesy je nastaven způsob ovládání s fixací v patře, kolečkem myši lze možné měnit elevaci a se stisknutím levého tlačítka myši úhly pohledu. Při druhém stisknutí je nastaven režim volného průletu vyžadující pro pohyb kamery stisk levého tlačítka. Při třetím stisku je nastaven režim průletu.

„i“ - slouží pro vypnutí nebo zapnutí ovládání kamery. (užitečné při umisťování předmětu do zobrazení nebo změnách jeho parametrů)

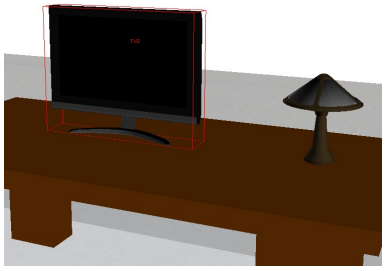
„m“ - náhled na všechny objekty ve scéně

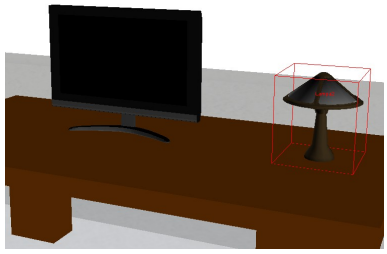
„w“ - pohyb kamerou vpřed, „s“ - pohyb vzad

Klávesy pro editace objektů

„a“ – povolit všechny objekty jako označitelné

„t“ - vybrat předmět, při výběru objektu se vždy ukazují názvy objektů u kurzoru myši a drátěný model opsaného kvádru (obrázky 27 a 28), po kliknutí na předmět se kolem něj vykreslí červený poloprůhledný kvádr.





Obrázky 27 a 28 – označování objektů kursorem myši

„g“ - změnit polohu předmětu myší s gravitací. Po označení objektu a aktivace tohoto režimu je možné tahy myší v daném patře určit novou polohu objektu, v případě umístění předmětu na jiný, a pokud je tento jiný objekt větší než předmět, je editovaný předmět umístěn na něj.

„e“ - změnit polohu předmětu myší s nastavením elevace kolečkem myši.

„r“ - změnit polohu předmětu myší s nastavením rotace kolečkem myši.

„z“ - změnit polohu předmětu myší s nastavením výšky předmětu kolečkem myši.

„y“ - nastavení šířky předmětu v ose Y.

„x“ nastavení šířky předmětu v ose X.

„q“ nastavení velikosti předmětu v obou osách naráz.

„o“ - odstraní objekt

„p“ - režim propojování objektů (například vypínač – světlo)

Klávesy pro práci se zónami

„k“ - nová zóna

Po stisknutí této klávesy se objeví dialog, vyzívající uživatele k zadání názvu zóny, ten musí být unikátní. Po zadání názvu se objeví dialog pro výběr barvy nové zóny

„l“ - nová obdélníková zóna

„n“ - odstraní zónu

„u“ - uložit stav komponenty (do „cachefile“, pro zrušení nutn „cachefile smazat“)

5.3 Užití komponenty (mapview.dll)

Komponentu lze použít buďto v režimu panelu vloženého do vlastní aplikace (objekt **MapViewer**) nebo v režimu vlastního okna (**MapForm**). Nejprve bude probráno použití v režimu panelu.

5.3.1 Režim panelu – objekt MapViewer

Inicializace

Komponentu inicializujeme vložením objektu **MapViewer** do kódu programu nebo tak lze učinit přímo v design módu Visual Studia 2008 nebo 2010.

Pokud požadujeme vložit do programu komponentu v design módu je nutné ji nejprve přidat mezi prvky kontroly pravým kliknutím tlačítka do postranního panelu „Toolbox“, následkem čehož se otevře dialogové okno. V dialogovém okně je nutné zadat cestu ke knihovně s komponentou (mapview.dll) a následkem toho se mezi prvky toolboxu zobrazí žádaný „**MapViewer**“ (a také „**MapForm**“, „**SysEdit**“, „**Logger**“ a „**Log**“). V tomto okamžiku lze postupovat již běžným způsobem, jako při běžné práci v design módu (vložit panel do prostoru okna, roztáhnout na požadovanou velikost, a podobně).

Pokud bychom ale program s pouze inicializovanou komponentou spustili, naskytl by se pohled na bílou plochu, jelikož v komponentě nejsou zatím načteny žádné objekty ani mapy. Dalším krokem pro získání žádaného zobrazení je tedy volání metody našeho objektu třídy MapViewer: **Load**(). Aby bylo volání této metody úspěšné, je nutné dbát na to, aby ve složce „debug“, kde se debugguje náš právě vznikající program, byl přítomný mimo knihovny (tu nahraje Visual Studio automaticky) i soubor nastavení pro knihovnu – „mapview.conf“. Samozřejmě, pokud jsou modely a mapy nastavené v „mapview.conf“ adresovány relativními cestami a ne absolutními, je nutné rovněž nahrát do správné složky i soubory modelů a map.

Po těchto prvních krocích by se měl zobrazit celý model bytu/domu i se všemi objekty.

Práce s objekty

```
Object GetStatus(String name)
```

Funkce navrátí status jednoho akčního objektu názvu „name“. Status je navrácen ve formě Object, ten může obsahovat typ bool, int nebo float, dle typu nastavitelného akčního objektu (regulovatelný, stavový, on/off)

```
Set(String name, Object setting)
```

Umožňuje nastavit libovolný akční objekt s názvem „name“. Dle typu objektu je potom položkou setting předat float, int nebo bool.

```
String[] GetElevatorNames()
```
Navrátí názvy výtahů ve scéně.

```
SetElevator(String name, int floor)
```
Nastaví cílové patro výtahu a uvede výtah do pohybu.

```
ChangeObjectParams(String name, Object[] parameters)
```
Umožňuje změnit parametry libovolného objektu na scéně – vyjma výtahů a herců. Parametr „name“ je název objektu a do parametru „parameters“ se píší instrukce s názvy měněných atributů. Do pole parametrů je vždy nutné nejprve uvést název měněného atributu a za něj například číslo ve float, int nebo booleanovské proměnné. Atributy lze tedy zadávat v libovolném pořadí.
Příklad volání:

```
ChangeObjectParams("Stul_u_TV", new Object[] { "x", 120.4f, "floor", 1 });
```

Takovéto volání by změnilo atribut umístění „Stul_u_TV“ v patře do pozice x a přesunulo by stůl do 1. patra.
Výčet měnitelných atributů:
„x“, číslo ve float – změna umístění předmětu – souřadnice v ose x
„y“, číslo ve float – změna umístění předmětu – souřadnice v ose y
„floor“, číslo v int – změna umístění předmětu – patro
„elev“, číslo ve float – změna elevace předmětu v patře
„angle“, číslo ve float – změna úhlu natočení předmětu
„width“, číslo ve float – změna šířky objektu
„height“, číslo ve float – změna délky předmětu
„zscale“, číslo ve float – změna výšky předmětu (škálování v ose z)
„clickable“, bool – změna atributu klikutelnosti objektu

```
GetConnections(out String[] controllers, out String[] controlled)
```
Získá seznamy propojených objektů v poli „controllers“ jsou prvky ovládající prvnky „controlled“.

```
SetConnection(String nameA, String nameB)
```
Umožňuje propojit 2 objekty názvu „nameA“ a „nameB“, volání je účinné pouze v případě má-li jeden z objektů nastaven příznak „controlled“ a druhý „controller“, na pořadí nezáleží. Při nesplnění se nic nestane a není provedena žádná akce.

```
RemoveConnection(String name)
```
Zruší spojení mezi objekty. Parametr „name“ je název jednoho předmětu z propojených předmětů.

```
String[] GetAllObjectsTypes()
```
Navrátí všechny druhy předmětů pro vložení.

```
String[] GetObjectsTypes()
```
Navrátí názvy typů veškerých statických objektů v knihovně, které je možno vložit do scény.

```
String[] GetActionObjectsTypes()
```
Navrátí názvy všech typů akčních objektů, jenž je možno vložit do scény.

```
PlaceObject(String type, String name, float x, float y, float xsize, float ysize, float angle, int floor)
```
Vloží do scény objekt typu „type“, názvu „name“ na pozici „x“, „y“ v patře „floor“ o velikost „xsize“, „ysize“ a úhlu natočení „angle“. Je důležité aby název předmětu byl unikátní, jinak není provedena žádná operace.

```
RemoveObject(String name)
```
Odstraní objekt s názvem „name“ ze scény.

Práce s „herci“

Do komponenty je možné přidávat „herce“ - dle definovaného typu z editoru. V základní verzi jsou typy 2: „Clovek“ a „Vozicek“. V editoru je možné typy přidat nebo rozčlenit (například na muže a ženy, obézní, pojízdné lůžko, apod). Nyní budou probrány jednoduché základní operace s nimi:

```
InsertActor(String type, String name, float x, float y, int floor, float angle, Color c, byte alpha)
```
Tuto metodu je možné volat i bez parametru alpha nebo dokonce i bez parametrů alpha i color, v těchto případech je objektům implicitně nastavena oranžovo-růžová barva. Prvním parametrem je type, ten odpovídá typům herců definovaných v editoru (tedy základně „Clovek“ nebo „Vozicek“). Druhým parametrem je zvolené jméno a zbývající parametry udávají umístění v prostoru, patře a natočení (ve stupních). I vložení nebo odstranění herce je možné loggovat (více viz sekce logger).

```
SetActor(String name, float x, float y, int floor, float angle)
```
Nastavuje umístění herce v prostoru. V případě zadání neexistujícího názvu se nic neděje a ani neloguje.

```
SetActor(String name, Color color)
```
Nastaví barvu herci.

```
String[] GetActorList()
```
Navrátí seznam všech herců na scéně.

```
GetActorParameters(String name, out bool hitWallCorrection, out Color color)
GetActorParameters(out String[] name, out float[] x, out float[] y, out int[] floor, out float[] angle)
```
Dvě variace metody navracející informace o hercích – zda-li je nastavena korkce nárazu do zdí (hitWallCorrection), informace o jejich barvě a především souřadnicích a úhlu natočení.

```
SetActorWalk(String name, bool on)
```

V případě humanoidního typu herce spustí animaci chůze. (Podrobnosti sekce popisu třídy Human). V opačném případě se nic nestane.

```
SetActorHitWallCorrection(String name, bool on)
```

Zapnutí korekce pohybu herce – v případě nárazu do zdi jí herec neprojde. V případě aktivovaného logování je vše zalogováno.

```
RemoveActor(String name)
```

Odstraní herce ze scény.

Nastavování zón

```
ShowZones(bool yes)
```

Povolí nebo zakazuje vykreslování zón ve zobrazení.

```
CreateZone(String name, Color color, float floor, PointF points)
```

Vytvoří novou zónu, určenou body „points“, barvy „color“ v patře „floor“.

```
RemoveZone(String name)
```

Odstraní zónu názvu „name“.

```
SetUserDrawZone(String name, Color color)
```

Nechá uživatele nakreslit scénu názvu „name“ a barvy „color“.

```
SetUserDeleteZone()
```

Dovolí herci kliknutím odstranit zónu.

Nastavování pohledů a kamery

Pro práci s kamerou a nastavení pohledů slouží následující metody. Nyní budou popsány jejich funkce:

```
SetCamera(float x, float y, float z, float azimut, float zenit)
```

Tato metoda nastaví kameru do pozice x, y, z v prostoru a natočí v azimutálním a vertikálním směru dle velikostí úhlů azimut a zenit. Do této metody (i zbývajících) se úhly rovněž nastavují ve stupních (tedy 0° až 360°). V komponentě je implementována funkce normalizace úhlů, tudíž lze zadat třeba i 934 stupňů a kamera se natočí bez jakékoliv ztráty stability aplikace. Kamera se po zadání této metody okamžitě nastaví na udané souřadnice, bez animace.

```
SetCameraMovementTime(float seconds)
```

Nastaví dobu pro posun kamery.

`SetCameraOn(String name)`

Animovaně natočí kameru na objekt s názvem „**name**“.

`SetCameraOn(String name, float distance)`

Animovaně natočí kameru na objekt s názvem „**name**“ a přesune se do žádané vzdálenosti „**distance**“.

`MoveCameraAngle(float azimut, float zenit)`

Plynule otočí kameru do žádaného úhlu o nejkratší možný úhel.

`MoveCameraTo(float x, float y, float z)`

Plynule přesune kameru v prostoru.

`MoveCameraTo(float x, float y, float z, float azimut, float zenit)`

Plynule přesune a natočí kameru v prostoru nejkratší možnou cestou.

`StopCameraMovement()`

Okamžitě zastaví pohyb kamery.

`ContinueCameraMovement()`

Dokončí poslední pohyb.

`Overview(bool transparentwalls)`

Přesune kameru do režimu náhledu na celý byt/dům/komplex. Jsou zde 2 možné nastavení – s průhlednými stěnami (aby bylo vidět na objekty) nebo s neprůhlednými stěnami.

`MovingCamera(bool on, bool free, bool lockmouse)`

Povolí nebo blokuje (v závislosti na parametru **on**) nastavování pohledu uživatelsky – myší. Kolečkem se mění výška pohledu, stisknutým tlačítkem a pohybem úhel natočení. V případěatavení i „free“ bude aktivován režim volné kamery a „lockmouse“ uzamkne kursor myši vprostřed zobrazení.

Režimy uživatelské editace objektů na scéně

Je doporučeno při povolení uživateli měnit rotaci, elevaci nebo výšku objektů zakázat režim pohybu kamery s užitím kolečka myši. Kolečko myši je totiž právě pro nastavení těchto vlastností využito.

Pro případ nutnosti vstupu uživatelem byl implementován přehledný a snadný mód editace objektů v prostoru, pro tento účel slouží tyto metody:

`ActivateEditKeys(bool yes, bool editkeys)`

Povolí všechny klávesy pro editace – komponentu nyní bude možné editovat stejně jako v testovacím módu. V případě aktivovaného nahrávání Loggerem se editace budou logovat a bude tedy možné změny navrátit i zpět.

SetUserEditMode(byte mode, String name)

Dle nastavení proměnné „mode“ aktivuje editaci objektu s názvem „name“. Pro mode=0 je režim přesunu předmětu s gravitací (tedy automatickou kontrolou výskytu předmětu pod editovaným)
mode=1 je režim změny velikosti objektu
mode=2 je režim změny polohy s gravitací a změny natočení objektu kolečkem myši
mode=3 je režim změny polohy bez gravitace s možností nastavení vlastní elevace v prostoru kolečkem myši
mode = 4 je režim změny polohy s gravitací a možnosti změny výšky objektu (v ose z) kolečkem myši

SetObjectSelecting(bool on)

Povolí nebo blokuje označování objektů v prostoru scény v závislosti na hodnotě proměňné „on“. Výsledek uživatelské vstupu je možné zachytit událostí OnObjectSelected.

SetAllObjectsClickable(bool on)

Tato metoda, v závislosti na hodnotě proměnné „on“, nastaví komponentu do režimu „všechny objekty kliknutelné“. V případě povolení uživateli označovat a klikat na objekty je tedy možné klikat na libovolný objekt (nejen ty s nastaveným příznakem klikutelný), popřípadě s ním manipulovat.

SetActorsClickable(bool on)

Povolí klikání na herce ve scéně.

SetNoObjectsClickable()

Nedovolí užiateli kliknout na žádný objekt. (Může se hodit s GetFloorClick)

GetFloorClick(int floor)

Tato metoda může být užitečná v případě vkládání nových objektů do již spuštěné komponenty (herců, statických objektů, …). Nastaví komponentu do režimu uživatelského vstupu pro nakliknutí pozice v prostoru vybraného patra proměnnou „floor“. Uživatelskou volbu je poté možné odchytit jako událost – OnFloorClick.

Práce s eventy komponenty

Komponenta obsahuje několik možností odchycení události:

OnActorHitWall

Událost o nárazu některého z herců do zdi. Těchto událostí je využíváno v simulátoru při simulaci omámených lidí (například po anestézi). Událost navrací jméno herce, pozici v prostoru patra, patro a informaci o nápravě nárazu.

OnActorHitObject

Událost o nárazu herce do některé z objektů ve zobrazení. Navrací mimo položek OnActorHitWall ještě navíc název objektu, do něhož herec narazil,.

OnActorInZone

Jak již bylo zmíněno, lze komponentě vytvářet zóny. Tato událost slouží k detekci příchodu herce do některé z nich. Například nás zajímá, zda-li prošel prostorem dveří nebo v

které místnosti herec je. (Můžeme mít například zónu „ložnice“ nebo pokoj „307“)

`OnActorClicked`
V případě povolení kliknutí na objekt herce tato událost vrací jeho název, polohu a směr pohledu.

`OnActorDoubleClicked`
Má obdobnou funkci jako předchozí událost, pouze se jedná o dvojklik.

`OnClickableClick`
Událost oznamující klinutí uživatelem do prostoru na určitý objekt. Tento objekt musí mít editorem nastaven příznak na kliknutelný nebo je možné voláním AllObjectsClickable() pak nakliknout libovolný objekt. Událost navrací pozici kursoru myši, název kliknutého objektu a pozici na objektu.

`OnClickableDoubleClick`
Obdoba minulé události pouze se jedná o dvojklik.

`OnObjectSelected`
Předchozí události oznamují o kliknutí na objekt i v případě deaktivovaného označování objektů. Tato událost oznamuje označení objektu uživatelem, v případě povolení této možnosti metodou SetObjectSelecting(true).
Navrací název označeného předmětu.

`OnKey`
Událost oznamující stiskutí klávesy.

Ostatní nastavení

`SaveCacheFile``()`
Uloží aktuální stav komponenty do „cachefile“. Pokud chceme úpravy odstranit, musí být ze složky s komponentou odstraněn soubor „cachefile“.

`ActivateKeys``(bool on)`
Zakáže nativní příkazy komponenty, včeteně kláves pro editaci (metoda EditKeys).

`SetBack``(Color c)`
Nastaví pozadí okna pro vykreslování.

`SetFont``(Font font)`
Nastaví font pro výpis textů v okně pro vykreslování.

Na panel do něhož se vykresluje pomocí Direct3D není možné vložit objekt Windows.Forms (například Label pro text), proto byla implementován do komponenty objekt toto umožňující. Následujícími metodami jej lze ovládat:

`WriteOnScreen``(String[] lines)`
Vypíše text v řádcích pod sebou v levém horním rohu okna pro vykreslování

```
WriteOnScreenAt(int posx, int posy, String text)
```
Napíše text na udané souřadnice okna pro vykreslování

Správné ukončení

Pro správné ukončení komponenty bez vyhození vyjímky je nutné volat metodu **Die**(). Tato metoda ukončí časování komponenty a nedojde tak o pokus k vykreslení v případě neexistující plochy pro vykreslování. Metodu není nutné volat při použití komponenty přes třídu **MapForm**, ta se o některé nastavení komponenty stará automaticky.

5.3.2 Logger

Logger ukládá všechny změny a události, které nastaly v komponentě. Komponentu lze využít buď přímo bez loggeru nebo s aktivovaným loggerem a tím pádem lze pak mít absolutní přehled i o všech událostech, které nastaly.

Práce s Loggerem

Pokud tedy chceme události loggovat, je nutné nejdříve z komponenty získat odkaz na objekt Logger. Toho je možné docílit voláním:
Logger mujLogger = mapViewer.GetLogger();

Aby se události začaly „nahrávat“, je nutné volat metodu SetRecordMode(true), v případě volání metody s parametrem false se nahrávání přeruší. Tímto způsobem je tedy možné nahrávat jen určité žádané úseky.

V případě spuštění přehrávání se nahrávání vždy přeruší. V případě jeho opětovného spuštění je doporučeno logger „přetočit“ na konec záznamu, jelikož v opačném případě bude záznam pokračovat s aktuálním stavem komponenty.

Základní operace

Nyní se již lze zabývat několika základními metodami loggeru, jejichž voláním můžeme dosáhnout těchto výsledků:
Play(bool backward) – bude přehrávat ve směru toku času (backward=false) nebo pozpátku (backward = true) do té doby, než se docílí konce nebo začátku záznamu
Stop() - zastaví přehrávání
SetSpeed(int secondsPerSecond) – nastavení rychlosti přehrávání, tedy kolik sekund nahraného času se má přehrát za 1 vteřinu reálného
JumpForward() - uvede komponentu do stavu konce záznamu
JumpBackward() - uvede komponentu do stavu počátku záznamu
StepForward a **StepBackward** fungjí obdobným způsobem, ale pro skok o jeden záznam dopředu nebo dozadu

JumpTo(DateTime dateTime) – nalezne a uvede komponentu do stavu, ve kterém se nacházela v čase dateTime
JumpTo(int index) – skočí na Log index-átý v pořadí
Snapshot() - uloží celý aktuální stav komponenty do logu
SimulateEvents(bool on) – pokud je on=true, bude nutit komponentu vyhazovat nahrané vyjímky

Pro případ potřeby uložení nebo opětovného načtení záznamů (logů) do/ze souboru byly do Loggeru připraveny i metody toto obstarávající. Jsou jimi **SaveToFile**(**String** filename) a **LoadFromFile**(**String** filename).

Rovněž bylo uvažováno s možností vlastního způsobu ukládání a práce s logy, proto byla implementována i funkce umožňující získat z Loggeru přímo list s jednotlivými Logy. Je jí funkce **List<Log> GetList**(). Pro bližší porozumění práci s logy doporučuji pročíst kapitolu třída Log z kapitol návrhu řešení a popisu implementace.

5.3.3 Režim okna – objekt MapForm

Pokud komponentu inicializujeme v režimu okna, není nutné se starat o hlídání změny velikosti okna, načítání nebo správné ukončení. Tyto rutiny si tento objekt obstarává sám. Pokud chceme pracovat s grafickými objekty nebo herci, je možné tyto metody volat přes objekt Map v MapForm například:

```
mapForm.Map.InsertActor("Clovek", "Clovicek", 100, 100, 0, 90);
```

Eventy se nastavují obdobným způsobem jako v režimu panelu například:

```
mapForm.OnActorInZone +=new MapForm.ActInZone(obsluznaMetoda);
```

Závěr

Cílem této práce bylo vytvoření programové komponenty pro zobrazování několika pater bytů / domu či ústavu, včetně vybavení a ovladatelných předmětů, schopnou ukládat vzniklé události. Téma práce bylo splněno.

Řešení bylo provedeno v jazyku C#, za použití zobrazovací technologie DirectX 9.0c (9.0c – z důvodu větší kompatibility se staršími GPU).

Vývoj a implementace zabraly téměř 85 procent času věnovaného práci, zhruba 10 procent zabral tento text a přibližně 5 procent vyhledávání informací.

Nejprve bylo řešeno zobrazení komponenty, systém výstavby modelu bytu z bitmapového souboru a systém ukládání grafických objektů v paměti, po vyřešení této problematiky byla komponenta rozšířena o možnosti uživatelských editací, výběry objektů a možnost uložení stavu zobrazení do souboru. V posledních fázích vývoje byly přidány funkce pro zjištění nárazu herců do objektu nebo zdi, gravitace působící na předměty a rovněž byl implementován Logger schopný veškeré události nebo změny nastalé v komponentě ukládat a v případě potřeby i načíst nebo uložit do souboru.

Pro komponentu bylo vytvořeno několik základních modelů – model TV, rádia, větráku, skříně, člověka, vozíčku, lampy, židle, stolu, WC, vany, umyvadla a dveří. Tyto modely slouží k základním demonstracím užití komponenty a je možné je velmi snadno rozšířit o mnohé další. Pro tento účel byl sepsán i manuál využití - v části o editaci se lze o problematice vytváření modelů naučit více.

Komponenta téměř nezatěžuje procesor počítače. Byla vyvíjena na notebookovém procesoru Intel Core 2 Duo, kde při užití aplikací v běhu vyžaduje 0 až 3 procenta výkonu CPU (vyjma vytváření modelů objektů a pater z bitmap při inicializaci).

Možnosti užití

Komponentu lze využít v SW pro ovládání inteligentních domácností, jakožto element ulehčující představu o ovládaných předmětech nebo jakožto asistivní technologie pro sledování samotných seniorů se zdravotními problémy. Další možné užití by bylo možné v programech pro simulace s pohybujícími se osobami – komponenta s největší pravděpodobností bude užita ve vznikajícím projektu na katedře kybernetiky „ Sledování / detekce pohybu pacientů omámených sedativy v nemocnicích“.

Jedna z počátečních verzí mé komponenty se na katedře kybernetiky již využívala – pro zobrazování polohy invalidního vozíku v bytě pro invalidy.

Možnosti vylepšení

Komponenta obsahuje přípravu pro přechod na vykreslování pomocí HLSL (High Level Shader Language) a obsahuje také jeden soubor naprogramovaného vlastního shaderu. Možné vylepšení by tedy bylo v přechodu na tuto metodu vykreslování. Výsledkem by byl lepší estetický dojem z realistického stínování a možnosti zobrazení

textur s prostorovou povrchovou úpravou. V grafických objektech jsou i připravené metody pro renderování touto metodou. (Brender). Bohužel tento způsob vykreslování nebyl dotažen do konce časovou náročností vývoje komponenty (ukázky kódu – příloha A).

Pro komponentu byla vytvořena ukázková aplikace (simulator.exe), která rovněž slouží pro testování komponenty jako takové. V aplikaci je možné otestovat většinu funkcí a vlastností komponenty – operace s kamerou, editace předmětů, nahrávání akcí a událostí a následné ukládání nebo nahrávání záznamů ze souboru a přidávání a operace s virtuálními osobami (herci). Byla do ní implementována jednoduchá logika pro ukázku funkčnosti události nárazu herce do stěny – automatická chůze vpřed se změnou úhlu pohybu v případě nárazu.

Literatura

[1] MSDN Library. [online]. [cit. 2011-4-24]. http://msdn.microsoft.com/en-us/library/

[2] GameDev forum[online]. [cit. 2011-4-24]. http://GameDev.net

[3] MDX info web [online]. [cit. 2011-4-24]. http://mdxinfo.com

[4] Triangulation by ear clipping [online]. [cit. 2011-4-30].
http://www.geometrictools.com/Documentation/TriangulationByEarClipping.pdf

[5] Free 3d model download for interior design [online]. [cit. 2011-5-2].
http://www.total-3d.com/

[6] WEB firmy Loxone [online]. [cit. 2011-5-5].
http://www.loxone.com/Pages/cz/default.aspx

[7] WEB firmy Elektrobock [online]. [cit. 2011-5-5]. http://elektrobock.cz

[8] WEB firmy AMX [online]. [cit 2011-5-5]. http://www.amx.com/

[9] WEB firmy InsightHome [online]. [cit. 2011-5-5]. http://www.insighthome.eu/

[10] Blender – 3D content creation suite [online]. [cit 2011-5-6]. http://blender.org

[11] Free textures web. [online]. [cit. 2011-5-6].
http://www.imageafter.com/textures.php

Příloha A – HLSL

HLSL – High Level Shader Language

Jedná se o syntaxi jazyka C, sloužící pro naprogramování shader jednotek na grafické kartě umožňující preciznější zobrazení scény a realistické stínování. V DirectX lze tento kód potom zkompilovat do shader jednotek pomocí objektu třídy Effect.

```
FX = Effect.FromFile(device, "x.fx", null, ShaderFlags.None, null);
```

Voláním objektu **Effect** pak lze docílit vykreslování přes vlastní shader. Proměnné vytvořeného zkompilovaného souboru se shaderem se nastavují přes metodu objektu **Effect** – **SetValue**.

Metoda **Brender**() v grafických objektech:

```
public virtual void Brender(Effect FX)
        {
            int numpasses = FX.Begin(0);                    //nastavení počtu průchodů
            for (int a = 0; a < meshmaterials.Length; a++)  //pro všechny materiály, obdobně v Render
            {
                for (int i = 0; i < numpasses; i++)     //renderují se jednotlivé průchody
                {
                    FX.BeginPass(i)
                    FX.SetValue("xColoredTexture", meshtextures[a]);    //nastavení textury do shadru
                    mesh.DrawSubset(a);             // vykreslení na Device
                    FX.EndPass();                   // ukončení a-tého průchodu
                }
            }

            FX.End();
        }
```

Ve vykreslovací smyčce **ShowDev** je přípraven blok kódu, který bude proveden v případě definování třídy **Effect**, tímto se mimo jiné i přestanou volat metody **Render**() v grafických objektů, nýbrž se budou volat metody **Brender**().

```
if (FX != null)
                    {
                        FX.Technique = "Simplest";
                        FX.SetValue("xLightPos", new Vector4(x, y, z, 0.1f));      //nastavení proměnných shaderu
                        FX.SetValue("xLightPower", 2);
                        FX.SetValue("xRot", Matrix.Identity);
                        FX.SetValue("ModelWorld", device.Transform.World);
                        FX.SetValue("xWorldViewProjection", device.Transform.View * device.Transform.Projection);
                    }
```

Příloha B – obsah CD

.\

DIP.pdf	(tento soubor)
simulator.exe	(ukázkový program využívající komponentu)
MapView.dll	(komponenta)
mapview.conf	(konfigurační soubor komponenty)
cachefile	(soubor posledního nastavení komponenty)

.\maps\

Složka s nákresy jednotlivých pater domu.

.\resources\

Ve složce „resources“ se nachází veškeré modely ve skriptu DirectX .x, používané komponentou. Tyto modely lze editovat programem Blender [10].

Příloha C – ukázky zobrazení