Na tomto místě bude oficiální zadání vaší práce

- Toto zadání je podepsané děkanem a vedoucím katedry,
- musíte si ho vyzvednout na studiijním oddělení Katedry počítačů na Karlově náměstí,
- v jedné odevzdané práci bude originál tohoto zadání (originál zůstává po obhajobě na katedře),
- ve druhé bude na stejném místě neověřená kopie tohoto dokumentu (tato se vám vrátí po obhajobě).

České vysoké učení technické v Praze
Fakulta elektrotechnická
Katedra počítačové grafiky a interakce

Diplomová práce

Komponentový přístup při tvorbě objektů herní knihovny

Bc. Martin Štýs

Vedoucí práce: Ing. Felkel Petr, Ph.D.

Studijní program: Otevřená informatika

Obor: Počítačová grafika a interakce

16. prosince 2013

Poděkování

Rád bych poděkoval panu Ing. Petru Felkelovi, Ph.D. za cenné rady, věcné připomínky a vstřícnost při konzultacích a vypracovávání diplomové práce.

Prohlášení

Prohlašuji, že jsem práci vypracoval samostatně a použil jsem pouze podklady uvedené v přiloženém seznamu.
Nemám závažný důvod proti užití tohoto školního díla ve smyslu §60 Zákona č. 121/2000 Sb., o právu autorském, o právech souvisejících s právem autorským a o změně některých zákonů (autorský zákon).

V Praze dne 16. prosince 2013 .....

Abstract

There are many game objects in modern computer games that need to be represented in a code. The most common way to represent them is via a class hierarchy which unfortunately introduces many problems such as the difficulty of hierarchy modification or a non-reusable code. This thesis focuses on a modern component-based game objects representation, which is a proper replacement for the class hierarchy. The component-based approach is very user friendly and does not suffer from problems the class hierarchy does. In the thesis a robust game object management system, which uses dividing a code to components and entity systems, was designed. This designed system was implemented as a set of library classes. Its usability was verified by specialization of the generic library to a library for 2D computer games and implementation of two separate games. The text can be used as a complete introduction to a virtual objects representation. The thesis proves the advantages of the component-based approaches over the class hierarchy and shows these advantages in practice by implementing two games. It is demonstrated that the components are easily reusable and that they can be managed with constant complexity thus the presented approach is suitable for real-time applications.

Abstrakt

Při tvorbě počítačových her je nutné reprezentovat mnoho objektů a jejich chování. Nejčastějším způsobem je reprezentace pomocí hierarchie tříd. Ta má ale řadu nevýhod, např. obtížnou modifikaci hierarchické struktury, či nemožnost opětovného použití vytvořeného kódu. Práce se zabývá moderním způsobem reprezentace herních objektů pomocí komponent, který zcela nahrazuje hierarchii tříd a odstraňuje problémy, kterými hierarchie trpí. Používání komponent je uživatelsky velmi pohodlné. Po pečlivé analýze byl sestaven návrh robustního systému pro správu herních objektů, který využívá dělení kódu na komponenty a systémy entit. Navržený systém byl implementován ve formě knihovny tříd. Kvalita a použitelnost návrhu byla ověřena specializací knihovny pro 2D hry a implementací dvou her. Text práce je použitelný jako shrnující úvod do problematiky reprezentace objektů virtuálního světa. Práce prokázala výhody a silné stránky návrhu pomocí komponent. Implementovaná knihovna je použitelná pro pohodlné vytváření nových her, což bylo předvedeno na implementací dvou her, při které byla prokázána pohodlná práce s komponentami a možnost opakovaného použití vytvořeného kódu. Také bylo předvedeno, že knihovna umožňuje práci s komponentami s konstantní složitostí, a je tedy velmi vhodná pro aplikace v reálném čase.

Obsah

Seznam obrázků

Seznam ukázek kódu

Kapitola 1

Úvod srovnávající metodu hierarchie tříd a komponentový přístup

Většina počítačových her obsahuje virtuální svět a uvnitř tohoto světa virtuální objekty. Virtuálním objektem v počítačové hře může být například budova, zbraň a strom. Tyto objekty jsou ve hře viditelné. Hry obsahují i objekty, které jsou pro hráče neviditelné. Neviditelným objektem může být např. kvádr, který vymezuje určitou oblast, která způsobí vyvolání nějaké akce, pokud do ní hráč vstoupí. Viditelnost je pouze jednou z vlastností, ve které se objekty mohou lišit. Herní objekty se tedy v určitých vlastnostech liší, ale jiné vlastnosti mají totožné.

Představme si trojrozměrnou hru, ve které zvolíme dva objekty: postavu a neonový nápis na budově. Na první pohled nápis a postava nemají nic společného. Postava může být ovládána (ať počítačem, či hráčem), může se pohybovat, můžeme s ní interagovat, má přiřazeny animace pro jednotlivé akce, může držet zbraň a mnoho dalších vlastností. Kdežto neonový nápis "neumí takřka nic". Přesto má s postavou společné některé vlastnosti. Oba objekty mohou být vykresleny na obrazovku, mají transformaci, lze s nimi interagovat (lze ovládat pohyb postavy, nápis je možné zhasnout), mají fyzikální reprezentaci reagující na zásah (postava přehraje animaci zásahu, nápis zhasne). Jednoduchý neonový nápis může blikat, může tedy také obsahovat určitou animaci (přestože je animace ve srovnání se skeletálními animacemi postav zcela primitivní, časování animací může být všem objektům společné).

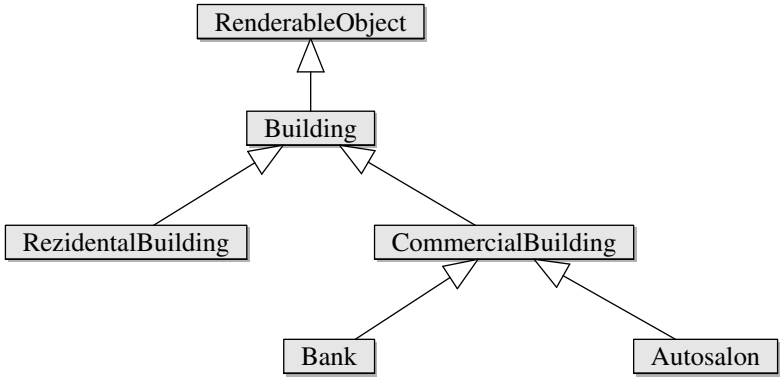
Všechny objekty a jejich vlastnosti je nutné ve hře reprezentovat. Nejpoužívanější metodou reprezentace je *hierarchie tříd*, přestože trpí mnoha vadami, které budou popsány v další sekci. Mnohem lepší z mnoha ohledů je reprezentace pomocí komponent, proto se jimi práce detailně zabývá.

1.1 Hierarchie tříd

Reprezentace pomocí hierarchie tříd využívá konceptu zvaného dědičnost[1]. Zjednodušeně řečeno, dědičnost shlukuje společné vlastnosti do hierarchicky výše postavených tříd a odvozené třídy jejich funkcionality mohou využívat. Například třída `Člověk`, může obsahovat

[1] http://en.wikipedia.org/wiki/Inheritance_(object-oriented_programming)

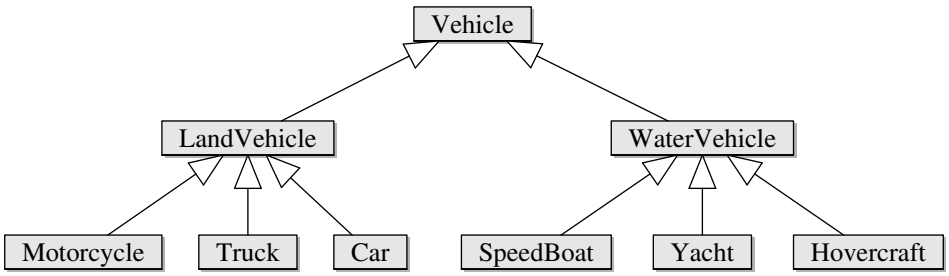
data jako je jméno či věk osoby. Třída `Zaměstnanec` odvozená od třídy `Člověk` automaticky tato data přejímá a může je rozšiřovat o další, jako např. plat. S využitím vlastností objektově orientovaného programovacího jazyku dělíme objekty taxonomicky. Je to jednoduché a přirozené. Pokud bychom hierarchií tříd měli reprezentovat např. budovy ve virtuálním městě, mohl by výsek z hierarchie vypadat jako na obrázku 1.1.



Obrázek 1.1: Hierarchie tříd pro budovy

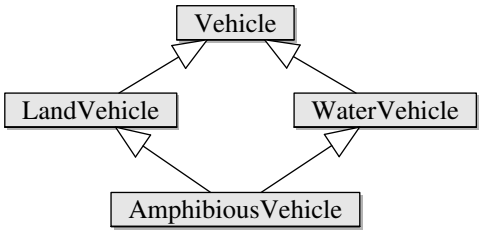
`RenderableObject` bude třída zodpovědná za vykreslování objektů. Bude poskytovat metody jako např. `draw()`. Ve třídě `Building` bude zapouzdřeno chování spojené s budovami obecně, bude například řešit vcházení do budov. Také zde bude nadefinována metoda `draw()` z nadřazené třídy, ve které bude již konkrétně vyřešeno vykreslování budov (vykreslování komerčních a rezidentních budov se nebude lišit). Pak se již hierarchie dělí na konkrétní typy budov. Banka a Autosalon mohou mít nějaké chování sdílené. Obě jsou to komerční budovy, tedy prostory, kde hra může stanovit nějaké omezení oproti chování uvnitř obytné budovy. Tato omezení tedy budou logicky umístěna ve třídě `CommercialBuilding`, abychom předešli duplikaci kódu. Takovéto rozdělení je intuitivní a na první pohled nejsou vidět žádné problémy.

Podívejme se na obdobný příklad, na kterém si ukážeme, že zpočátku zcela logické využití hierarchie tříd se časem může ukázat jako problematické. Navrhujeme hru, kde hráč bude moci ke své přepravě používat množství vozidel - automobily, motocykly, nákladní automobily, čluny, jachty a vodní vznášedla. Reprezentace pomocí hierarchie tříd by vypadala obdobně jako na obrázku 1.2.



Obrázek 1.2: Hierarchie tříd pro jednoúčelová vozidla

Proč může být uvedená reprezentace vozidel špatná? Především kvůli tomu, že při vývoji her nastávají neočekávané změny. Vývoj kvalitní počítačové hry je velmi náročný proces. Není možné vždy vše dopředu navrhnout a považovat za neměnné. Můžeme např. vyvinout umělou inteligenci pro nepřátelské postavy, která bude nesmírně sofistikovaná a bude dělat složitá rozhodnutí, ale mnohem důležitější než technologická vyspělost je výsledný pocit hráče z nepřátel [4]. Mnohem jednodušší umělá inteligence může působit výrazně chytřejším dojmem. Často tedy nastává situace, kdy designeři mění herní prvky uprostřed vývoje. Představme si tedy situaci, kdy jsou již naimplementovaná chování vozidel uvedených výše a přijde na řadu jejich testování. Všechna vozidla se chovají přesně jak mají, jenže herní svět obsahuje spoustu silnic a spoustu vodních ploch, což to nutí hráče často přesedat mezi vozidly. Je to drobnost, která však může kazit celkový dojem ze hry. Designeři tento problém vyřeší tak, že do hry přidají ještě obojživelná vozidla - tedy vozidla která umožňují jízdu jak po vodě, tak po souši. Reflektování této změny v hierarchii vozidel by vypadalo jako na obrázku 1.3. [13]



Obrázek 1.3: Část hierarchie tříd pro víceúčelová vozidla

Toto řešení znamená zanesení problému zvaného *Deadly diamond of death*. Tento problém spočívá v tom, že jedna třída dědí z více tříd, které mají společného předka. Z principu fungování dědičnosti je instance třídy `LandVehicle` také instancí třídy `Vehicle`. Stejně tak instance třídy `WaterVehicle` je instancí třídy `Vehicle`. Po definování třídy `AmphibiousVehicle` jako třídy, která dědí jak z `LandVehicle`, tak z `WaterVehicle`, jsme se dostali do problému, kdy jedné instanci `AmphibiousVehicle` náleží dvě instance `Vehicle` (z každého přímého rodiče jedna). To vede k rozporům (pokud k objektu náleží dvě instance třídy `Vehicle`, tak při volání metody této není jednoznačné, kterou instanci pro volání použít. Další rozpor je popsán v následujícím odstavci.

Představme si, že třída `Vehicle` obsahuje abstraktní (virtuální) metodu `steer(direction)` (metoda určená k zatočení do daného směru). Pozemní vozidlo tuto metodu překryje svou implementací, ve které je definováno zatáčení pomocí otočení jednoho, či více kol. Podobně metodu překryje její implementace uvnitř vodního plavidla. Tam bude zatáčení realizováno otočením kormidla. Pokud poté na instanci obojživelného vozidla zavoláme např. metodu `steer(RIGHT)`, nebude jasné, jakou implementaci pro zatočení má využít (z pozemního vozidla, či plavidla).

Deadly diamond of death není problém, který by nešel vyřešit, je však nutno namísto klasické dědičnosti využít dědičnosti virtuální. Ukázka je uvedena v kódu 1.1. Virtuální dědičnost je však málo známým konceptem a mnoho programátorů v jejím použití (nepoužití) chybuje. Nově příchozí programátor ji může snadno přehlédnout (v kódu C++ je mechanizmus virtuální dědičnosti spuštěn pouhým jedním klíčovým slovem). Virtuální dědičnost má

za následek mnohem náročnější údržbu kódu [27]. Při refaktoringu a zavádění nových typů je také třeba ke struktuře tvaru diamantu v hierarchii přistupovat zcela odlišně než k běžné stromové struktuře. Více v [17].

```
class Vehicle{ /* ... */ };

class LandVehicle : virtual public Vehicle{ /* ... */ };
class WaterVehicle: virtual public Vehicle{ /* ... */ };

class AmphibiousVehicle :
    public LandVehicle, public WaterVehicle{ /* ... */ };
```

Kód 1.1: Ukázka virtuální dědičnosti

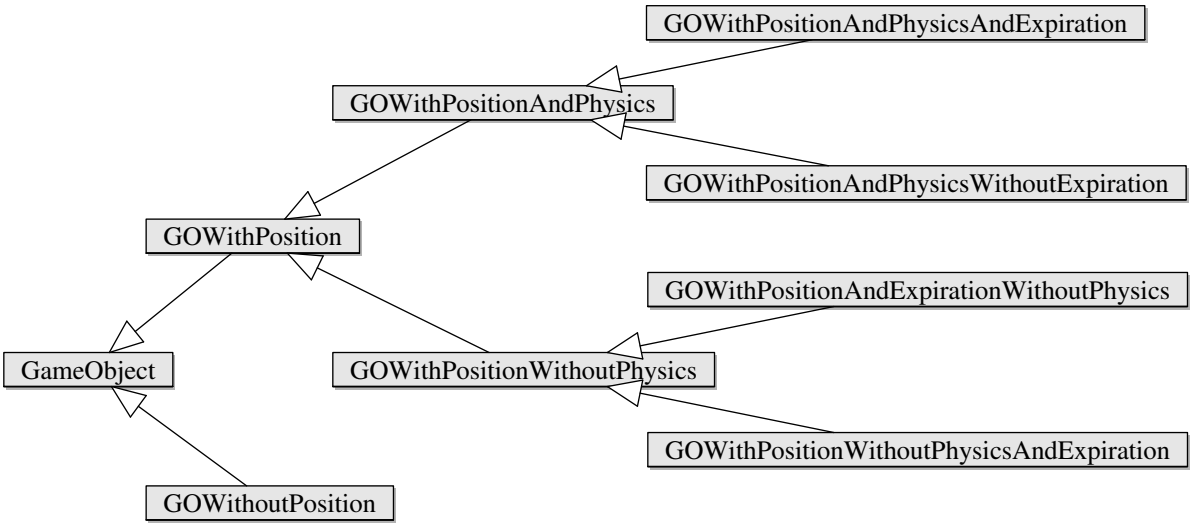
Bez využití vícenásobné dědičnosti nastávají další problémy. Představme si, že herní objekt může mít (mimo jiné) následující vlastnosti: *Transformaci* - pro určení jeho pozice, rotace a měřítka; *Fyzikální reprezentaci* - dodává objektu jeho tvar pro detekci a řešení kolizí; *Expiraci* - uchovává čas, po který se objekt bude ve světě nacházet. V tabulce 1.1 jsou uvedeny příklady objektů, které mají pouze určitou podmnožinu z uvedených vlastností.

Transf.	Fyz. rep.	Expir.	Příklad objektu
Ano	Ano	Ano	Projektil
Ano	Ano	Ne	Země po které se hráč pohybuje
Ano	Ne	Ano	Stoupající nápis zobrazující zdraví protivníka po zásahu
Ano	Ne	Ne	Obrázek na pozadí
Ne	Ne	Ano	Objekt reprezentující dočasně získanou speciální vlastnost
Ne	Ne	Ne	Zvuk

Tabulka 1.1: Ukázka možných herních objektů s různými kombinacemi vlastností

Pokud bychom chtěli předejít redundanci v datech a přiřazování funkcionality objektu, kterému nepatří, vypadala by hierarchie zhruba jako na obrázku 1.4 (obrázek pro přehlednost zachycuje pouze jednu větev), vidíme, že to vede nejen k exponenciálnímu nárůstu počtu tříd, ale i ke značné duplikaci kódu.

Takovýto přístup je prakticky nepoužitelný. V praxi se používá spíše přesný opak, tedy objekt má přiřazenou funkcionalitu, kterou ve skutečnosti nemá. Taková situace může vzniknout velmi jednoduše. Ve hře mohl být navržen speciální gravitační nástroj, kterým bylo možné určitému objektu změnit působení gravitační síly a nechat ho např. viset ve vzduchu. To se ukáže jako velmi efektní a podporující spád hry, designeři se tedy rozhodnou, že takto půjde ovlivnit více objektů. Původní hierarchie však nevznikala dělením na objekty, které takto ovlivnit jdou a které nejdou. Řešením by byla vícenásobná dědičnost. Vzhledem k jejím nevýhodám popsaným výše programátoři sahají k jinému přístupu. Naleznou třídu, která je předkem všem třídám, které chtějí využívat změny gravitace a funkcionalitu umístí právě tam. Často ještě třídu doplní o logickou proměnou, která určuje, zda právě tato instance může funkcionalitu změny gravitace využívat. Kód pro toto chování se tak může např. objevit ve třídě `MovableObject`, která bude v hierarchii velmi vysoko, přestože polovina pohyblivých objektů manipulaci gravitací nepodporuje. Je zřejmé, že takovýto přístup vede ke značnému znepřehledňování kódu. Čím více podobných případů bude nastávat,



Obrázek 1.4: Možný exponenciální nárůst počtu tříd v hierarchii

tím více budou obecnější třídy zahlceny kódem, který do nich z logického pohledu rozhodně nepatří. To vede ke zvyšování času potřebného k údržbě a refaktoringu kódu a zvyšuje to pravděpodobnost programátorských chyb.

Dalším problémem, který je nutno řešit, je traverzace napříč hierarchií. Jenou z možností je použít *Non-Virtual Interface* idiom [24]. Takovýto přístup ale vede k mírnému nafukování kódu, protože je počet funkcí zdvojnásoben. Nejčastěji se tedy používá průchod od potomka k rodiči. Uvnitř rodičovské třídy je umístěna virtuální metoda (zpravidla `update()`), která způsobí zavolání metody nejhlubšího potomka. Ten vykoná kód potřebný pro svou aktualizaci a jeho zodpovědností je zavolat metodu `update` svého přímého rodiče (nebo prvního předka směrem ke kořeni, který tuto metodu překrývá). Vraťme se k ukázce hierarchie z obrázku 1.1. Kód pro traverzaci uzlů `CommercialBuilding` a `Bank` by vypadal následovně:

```
1  void CommercialBuilding::update()
2  {
3      Building::update();
4      //do some CommercialBuilding related stuff
5  }
6
7  void Bank::update()
8  {
9      CommercialBuilding::update();
10     //do some Bank related stuff
11 }
```

Kód 1.2: Kód pro traverzaci uzlů `CommercialBuilding` a `Bank`

Problém nastává tehdy, pokud hierarchii upravujeme. Pokud budeme postupem času ještě více oddělovat funkcionalitu tříd (velmi běžný proces), musíme opravit kód na více místech. Rozhodneme-li se `CommercialBuilding` rozdělit ještě na budovy, kde jsou čistě jen kanceláře a budovy se službami pro zákazníky (`CustommerOrientedBuilding`), musíme projít veškeré třídy, které původně dědily ze třídy `CommercialBuilding`, a upravit volání předka na správného. Takových tříd může být hned několik. Někdy je třeba upravit i vnuky či

ještě hlubší potomky. Je velmi snadné na nějakého potomka zapomenout. Kód se zkompiluje v pořádku, hra bude fungovat, jen např. po vstupu do jednoho konkrétního typu budovy se hra bude chovat neočekávaně. Chyba tedy může vyjít na povrch až po nějaké době a její hledání stojí vývojáře čas. Tento postup je tedy velmi náchylný k zanesení chyb, které se neprojeví okamžitě a takovéto chyby jsou velice nepříjemné.

Při použití hierarchie však vznikají i další problémy. Kvůli velmi těsným vztahům mezi objekty (dědičnost je jednou z nejsilnějších možných vazeb mezi objekty v C++) se prodlužuje čas kompilace. Změna rozhraní třídy uvnitř hierarchie vede k nutnosti rekompilace všech potomků této třídy. Dalším problémem je srozumitelnost kódu. Pokud je kód rozsáhlý, není možné, aby si programátor pamatoval, kde a jak se provádí jaké operace (situace je poté podobná jako pro programátora, který se přidal k týmu později a kód vidí poprvé). Chce-li programátor nastudovat, jak funguje daný herní objekt, musí také nastudovat všechny třídy v hierarchii výše.

Nevýhodou hierarchie tříd je tedy především její malá flexibilita. Taxonomické dělení je prováděno vždy jen dle jedné osy (např. má fyzikální reprezentaci / nemá fyzikální reprezentaci), což vede buď k velkému počtu tříd a duplikaci kódu, nebo ke vkládání kódu tam, kam nepatří. Také její změny v pozdějších fázích vývoje jsou obtížné.

1.2 Komponentový přístup

V předchozí sekci byly detailně popsány značné nevýhody hierarchie tříd. V této sekci bude obecně popsána reprezentace objektů pomocí komponent. Podrobněji se jí budeme zabývat v kapitolách následujících.

Jaké jsou tedy hlavní požadavky na objekt reprezentovaný komponentami? Ideální by bylo, aby:

1. objektu byla přiřazena jen ta data, které má skutečně mít
2. objektu byla přiřazena jen ta funkcionalita, kterou má skutečně mít
3. komponenty bylo možné použít opakovaně
4. aktualizace dané funkcionality byla na jednom logickém místě
5. komunikace mezi komponentami efektivní a pro programátora pohodlná

Nyní budou body z výčtu vysvětleny podrobněji s pomocí příkladu reprezentace zdraví a smrti herních objektů. V hierarchii tříd by to bylo možné reprezentovat třídou `VulnerableObject` (zranitelný objekt). Uvnitř této třídy bychom kontrolovali, zda aktuální hodnota zdraví neklesla pod hodnotu nula (mohli bychom reagovat i na více hodnot a např. spouštět animace částečného zranění, pro jednoduchost ale dále bude na zdraví pohlíženo pouze binárně - žije / nežije). V případě poklesu pod nulu bychom provedli požadovanou akci, např. zavoláním metody `onLifeEnd()`. Tuto metodu by bylo možné předefinovat v potomcích a implementovat různé způsoby smrti pro různé objekty. Zdraví se netýká ale pouze herních postav. Může se týkat i věcí. Hodnota zdraví může reprezentovat např. aktuální zničení automobilu a pokud klesne na nulu, automobil přestane jezdit.

Bod č. 1 hovoří o přiřazování pouze relevantních dat. Tím je myšleno, aby objekt obsahoval pouze data, která sám využívá. Protože některé živé objekty ve hře půjdou zabít (protivníci) a některé zabít nepůjdou (např. tráva) a některé neživé věci půjdou zcela rozbít (automobil) a některé nepůjdou rozbít (stůl), je pravděpodobné, že třída `VulnerableObject` skončí v hierarchii velmi vysoko a bude z ní dědit řada tříd, které by z ní dědit neměla. Tohoto se zbavíme tak, že *zranitelnost* budeme reprezentovat komponentou a tuto komponentu přiřadíme pouze objektům, které jsou opravdu zranitelné. Vznikne tak komponenta `VulnerableComponent` a ta bude obsahovat pouze data popisující zdravotní stav (aktuální zdravotní stav, nejlepší možný zdravotní stav, ...)

Bod č. 2 zní obdobně jako bod předešlý, týká se však funkcionality, tedy určitých reakcí a chování na základě dat. Funkcionalitou pro uváděný příklad je kontrola zbývajícího zdraví a případné vyvolání akcí při poklesu pod danou hranici. Protože některé objekty budou *umírat* vyvoláním skeletální animace (protivníci) a jiné např. simulací fyzikálním systémem (váza), je opět pravděpodobné, že bude funkcionalita v případě hierarchie obsažena i v objektech, kam rozhodně v dané podobě nepatří. Při komponentovém přístupu je více způsobů, jak funkcionalitu implementovat, tyto přístupy budou popsány dále v textu.

Opakované použití již hotového kódu, jak je uvedeno v bodu č. 3, je velmi žádané. Herní studia často vyvíjejí více herních titulů a použití již hotového kódu vede k ušetření peněz. Každá hra je ale jiná (ať méně, či více). Kód pro umírání postavy může být napříč hrami zcela odlišný. V jedné hře může postava pouze zmizet, v jiné může vyvolat změny v chování umělé inteligence a spuštění dalších složitých akcí. Aby bylo možné znovu používat implementovanou logiku, je třeba kód psát velmi genericky. V další hře však může být vymyšlen nový herní prvek, se kterým původní programátoři nepočítali, a je tedy nutné implementaci pracně měnit. Čím více generický kód, tím méně zpravidla bývá efektivní pro danou situaci. Protože komponentový přístup je postaven na oddělování zájmů jednotlivých funkčních celků, lze předpokládat, že znovupoužití vybraných komponent bude jednodušší, než opětovné použití vybraných tříd z hierarchie. Samotný komponentový přístup však v žádném případě opětovně použitelný kód nezaručuje.

Bod č. 4 zdůrazňuje požadavek na logické umísťování v logických částech kódu. Každá elementární funkcionalita by měla být oddělena od ostatních funkcionalit a tedy umožňovat práci na ní bez znalostí ostatních aktualizací.

Bod č. 5 se zaměřuje na komunikaci mezi komponentami. V hierarchii jsou objekty těsně svázány a pokud např. postava dědí ze třídy `VulnerableObject`, tak pro získání aktuálního zdraví stačí zavolat metodu `getHealth()`. Je jasné, že pokud se komponentový přístup snaží rozbít co nejvíce vazeb mezi objekty, tak komunikace v rámci objektu již nebude tak přímočará. Přesto je velmi důležité, aby komunikace byla efektivní a pro uživatele(programátora) pohodlná. Pokud by nebyla, mohlo by to výrazně zastínit ostatní přínosy komponentového přístupu.

Kapitola 2

Rešerše

V předchozí kapitole byl pojem *komponenta* používán jako abstraktní pojem, který bude nyní formálněji nadefinován. Z pohledu UML dle [1]: "Komponenta zastupuje modulární část systému, jenž ukrývá svůj obsah, a jehož projev může být okolním prostředím nahrazen." Ukrytý obsah je přístupný pouze přes pevně stanovené rozhraní a komponenta je tedy jakási černá skříňka, která může být nahrazena jinou komponentou, která se řídí stejným komunikačním protokolem - má stejné rozhraní. Je zřejmé, že definice komponenty je do značné míry obecná. Neříká, jestli má komponenta být jedna třída z pohledu C++, či celý balík z pohledu programovacího jazyku Java. Pojem komponenta se běžně používá i mimo software. Automobil či počítač jsou také složeny z komponent. Pokud potřebujete např. vyměnit grafickou kartu v počítači za výkonnější, vyměňujete vlastně komponentu - vyjmete určitý kus, o kterém nemusíte vědět, jak uvnitř funguje, nahradíte ho jiným kusem se stejným rozhraním (aby bylo možné jej zapojit do základní desky) a počítač bude nadále fungovat. Dále se text bude zabývat výhradně komponentami softwarovými.

Jak již bylo řečeno, softwarová komponenta může nabývat různých velikostí. Vždy se spolu s komponentami mluví o znovupoužitelnosti. Herní vývojáři dnes mají možnost zakoupit tzv. *Commercial of the Shelf* (COTS) komponenty. To jsou již hotové komponenty, které jsou připraveny na zapojení do vyvíjeného herního stroje (*game engine*). Například 3D fyzikální systém. Při vývoji velikých komerčních titulů jde především o peníze, a proto je nákup již hotových součástí na denní pořádku. Zakoupení již hotového celku má své klady a zápory. Mezi klady patří finanční stránka. Často bývá výrazně levnější investovat do již hotové věci, než ji vyvíjet znovu. Prodávající firma se povětšinou specializuje na danou problematiku dlouhodobě. Pokud se nějaká komponenta vyvíjí od nuly a programátoři s tím nemají patřičné zkušenosti, může to vést např. k prodloužení doby vývoje. Zakoupené komponenty také bývají důkladněji otestovány, protože jsou prodávány více klientům. Nevýhodou však bývá obecnější rozhraní, které sice umožní zapojení do širokého spektra herních strojů, avšak často na úkor efektivity (nutné kopírování a konverze dat...). Více o COTS se můžete dočíst v článku [11].

Další část textu se bude věnovat komponentám při správě herních objektů. I v tomto užším kontextu je použití komponent velmi široký pojem a různí autoři si ho vykládají různě. Nejprve budou detailněji popsány dva konkrétní příklady, poté budou zmíněny další přístupy a nakonec bude představen přístup, který bude dále v práci navrhnut a implementován.

2.1 Správa objektů pomocí komponent - První přístup

Bob Nystrom je autorem knihy **Game Programming Patterns** [19], kterou zveřejnil na (gameprogrammingpatterns.com). Kromě jiných návrhových vzorů autor popisuje i *návrhový vzor Komponenta*. Není to však přímo návrhový vzor, jen určitý autorův postoj k použití komponent.

Autor si v textu propůjčil koncept známé hry Mario[1], na kterém názorně své myšlenky demonstruje. Popisuje, jak jednoduše může vzniknout třída `Mario`, která bude obsahovat stovky řádků špatně udržitelného kódu. Mohl by např. obsahovat výsek kódu na ukázce 2.1.

```
if (collidingWithFloor() && (getRenderState() != INVISIBLE))
{
    playSound(HIT_FLOOR);
}
```

Kód 2.1: Ukázka špatně udržitelného kódu

Pokud by chtěl programátor kód pozměnit, musí nutně vědět něco o fyzice, grafice a zvuku, jen aby něco nerozbil. Je tedy třeba oddělit zájmy. Jak to provést pomocí komponent, je demonstrováno na ukázce celé monolitické třídy `Mario` (ukázka 2.2).

```
class Mario
{
public:
  void update(World& world, Graphics& graphics)
  {
    // Apply user input to hero's velocity.
    switch (Controller::getJoystickDirection())
    {
      case DIR_LEFT:
        m_velocity -= WALK_ACCELERATION;
        break;

      case DIR_RIGHT:
        m_velocity += WALK_ACCELERATION;
        break;
    }

    // Modify position by velocity.
    x_ += m_velocity;
    world.resolveCollision(m_volume, m_x, m_y, m_velocity);

    // Draw the appropriate sprite.
    Sprite* sprite = &m_spriteStand;
    if (m_velocity < 0) sprite = &m_spriteWalkLeft;
    else sprite = &m_spriteWalkRight;

    graphics.draw(*sprite, m_x, m_y);
  }

private:
  static const int WALK_ACCELERATION = 1;

  int m_velocity;
  int m_x, m_y;

  Volume m_volume;

  Sprite m_spriteStand;
```

[1] http://en.wikipedia.org/wiki/Super_Mario_(series)

```
39   Sprite m_spriteWalkLeft;
40   Sprite m_spriteWalkRight;
41 };
```

Kód 2.2: Monolitická třída `Mario`

Třída `Mario` má metodu `update`, která je volána v každém kroku hry. Nejprve se přečte případný vstup od hráče, aby mohlo být postavičkou pohnuto v požadovaném směru. Poté jsou vyřešeny případné kolize a nakonec je postavička vykreslena na obrazovku. Logika této metody je velmi primitivní, není zde řešena gravitace, animace ani podobné nezbytné herní prvky. Přesto je kód poměrně nahuštěný a nepřehledný. Představme si situaci, kdy kód obstarává desetinásobek uvedeného. Vyznat se v takovém kódu je časově náročnější a takový kód je náchylnější k chybám. Pokud navíc na postavičce pracuje více lidí (jeden např. na správě animací a další na řešení pohybu), dříve či později se objeví problémy s verzováním, protože programátoři editují stejný soubor na stejném místě.

Autor tedy navrhuje oddělit logické celky mimo, do separátních celků, které nazývá *komponenty*. Kód pro spravování uživatelského vstupu byl tedy přesunut do komponenty nazvané *InputComponent* jako v ukázce 2.3:

```
1 class InputComponent
2 {
3 public:
4   void update(Mario& mario)
5   {
6     switch (Controller::getJoystickDirection())
7     {
8       case DIR_LEFT:
9         mario.velocity -= WALK_ACCELERATION;
10        break;
11
12      case DIR_RIGHT:
13        mario.velocity += WALK_ACCELERATION;
14        break;
15    }
16  }
17
18 private:
19   static const int WALK_ACCELERATION = 1;
20 };
```

Kód 2.3: Kód komponenty `InputComponent`

Stejným způsobem byla z kódu vyčleněna fyzikální komponenta, která zodpovídá pouze za pohyb s postavičkou a vyřešení kolizí 2.4:.

```
1 class PhysicsComponent
2 {
3 public:
4   void update(Mario& mario, World& world)
5   {
6     mario.x += mario.velocity;
7     world.resolveCollision(m_volume, mario.x, mario.y, mario.velocity);
8   }
9
10 private:
11   Volume m_volume;
12 };
```

Kód 2.4: Kód komponenty `PhysicsComponent`

Poslední logicky separabilní část kódu, která ve třídě `Mario` zbyla, je část starající se o vykreslování hrdiny. I ta byla přesunuta do separátní grafické komponenty 2.5.

```
class GraphicsComponent
{
public:
  void update(Mario& mario, Graphics& graphics)
  {
    Sprite* sprite = &m_spriteStand;
    if (mario.velocity < 0) sprite = &m_spriteWalkLeft;
    else sprite = &m_spriteWalkRight;

    graphics.draw(*sprite, mario.x, mario.y);
  }

private:
  Sprite m_spriteStand;
  Sprite m_spriteWalkLeft;
  Sprite m_spriteWalkRight;
};
```

Kód 2.5: Kód komponenty `GraphicsComponent`

Nyní, když jsou veškeré logické celky z monolitické třídy `Mario` přesunuty do separátních tříd - komponent, je nutné je propojit v hlavního hrdinu. Autor navrhuje využít kompozice, tedy ponechat třídu `Mario`, ve které budou uloženy instance jednotlivých komponent (2.6). Komponenty budou aktualizovány pří aktualizaci postavičky.

```
class Mario
{
public:
  int velocity;
  int x, y;

  virtual void update(World& world, Graphics& graphics)
  {
    m_input.update(*this);
    m_physics.update(*this, world);
    m_graphics.update(*this, graphics);
  }

private:
  InputComponent    m_input;
  PhysicsComponent  m_physics;
  GraphicsComponent m_graphics;
};
```

Kód 2.6: Upravená třída `Mario`

Již nyní je zřejmé, že komponenty jsou velikým přínosem. Části kódu, které spolu logicky nesouvisí, jsou odděleny a mohou být spravovány nezávisle na sobě. Ve výše uvedené definici komponenty bylo řečeno, že při splnění daného rozhraní může být komponenta vyměněna za jinou. Stačí tedy nemít uloženy přímo instance komponent, ale pouze ukazatel na rozhraní. Ze vstupní komponenty tedy vytvoříme rozhraní `InputComponent`, které bude obsahovat metodu pro aktualizaci (ukázka 2.7).

Kód obstarávající logiku ohledně pohybu Maria je přesunut do nové komponenty `PlayerInputComponent`, která splňuje rozhraní `InputComponent`. Pokud tedy bude uložen pouze ukazatel na instanci `InputComponent`, bude možné komponenty vyměnit. Může například

vzniknout komponenta `DemoInputComponent`, která také bude splňovat rozhraní `InputComponent` a bude ho implementovat pomocí umělé inteligence. Nyní je možné pouhou záměnou komponenty "odpojit"uživatelský vstup a "zapojit"vstup od umělé inteligence.

```
1 class InputComponent
2 {
3 public:
4   virtual void update(Mario& mario) = 0;
5 };
```

Kód 2.7: Abstrakce vstupní komponenty

Pokud podobným způsobem schováme za rozhraní i grafickou a fyzikální komponentu, bude možné hlavní postavičce zaměňovat i fyzikální reprezentaci a chování a také grafickou podobu. Tím jsme ze třídy `Mario` udělali třídu obecnější. Pokud např. zmenšíme fyzikální reprezentaci na polovinu, změníme grafickou podobu v grafické komponentě a dosadíme komponentu využívající umělé inteligence na místo vstupní komponenty, dosáhneme zcela jiného objektu než byl Mario. Můžeme tak sestavit např. jinou postavičku - počítačem řízeného protivníka. Autor zachází tak daleko, že třídu `Mario` obsahující ukazatele na komponenty, přejmenoval na `GameObject`(kód 2.8).

```
1 class GameObject
2 {
3 public:
4   int velocity;
5   int x, y;
6
7   GameObject(InputComponent* input,
8              PhysicsComponent* physics,
9              GraphicsComponent* graphics)
10  : m_input(input),
11    m_physics(physics),
12    m_graphics(graphics)
13  {}
14
15  virtual void update(World& world, Graphics& graphics)
16  {
17    m_input->update(*this);
18    m_physics->update(*this, world);
19    m_graphics->update(*this, graphics);
20  }
21
22 private:
23   InputComponent*    m_input;
24   PhysicsComponent*  m_physics;
25   GraphicsComponent* m_graphics;
26 };
```

Kód 2.8: Třída `GameObject`

Tento přístup je na první pohled sofistikovaný, avšak existují určité vady, které brání použití kódu tak, jak je. Všimněte si např. pozice a rychlosti přímo uložené v `GameObject`. Většina objektů herního světa pozici má, ale zdaleka ne všechny (viz. výše). Parametr rychlost bude nadbytečný pro všechny statické objekty ve scéně. Díky tomu, že jsou tyto informace uloženy přímo v `GameObject`, mohou je komponenty přímo číst a zapisovat do nich. To je však velmi omezená možnost komunikace mezi komponentami.

Další nevýhodou je to, že objekt nutně musí být složen právě ze 3 komponent - grafické,

vstupní a fyzikální. Některé objekty však budou potřebovat další komponenty (např. expirační komponentu pro granát) a jiné naopak využijí jen jednu z nich (nápis využije pouze grafickou komponentu). Pokud tedy přibude nějaká komponenta, je nutné přidat ukazatel na ni do členských parametrů herního objektu. Také je nutné počítat s tím, že daná komponenta nemusí existovat, tedy počítat s možností `null`.

2.1.1 Instanciace objektů herního světa

Vytváření objektů při komponentovém přístupu je v porovnání s přístupem hierarchie tříd popsané v sekci 1.1 složitější. Výhodou objektu v hierarchii je jeho snazší vytvoření. Pokud by takto byl modelován Mario, byla by třída `Mario` odvozena od tříd jako `PhysicsObject` a `RenderableObject`. Pro vytvoření Maria bychom pouze vytvořili instanci `Mario`, např. `Mario * m = new Mario;`. Nyní ale žádná třída `Mario` neexistuje, existuje pouze třída `GameObject`, která agreguje komponenty. Pro vytvoření hráče musíme tedy postupovat např. jako na ukázce 2.9.

```
1 GameObject * createMario()
2 {
3     return new GameObject(new PlayerInputComponent(),
4                           new MarioPhysicsComponent(),
5                           new MarioGraphicsComponent());
6 }
```

Kód 2.9: Možná instanciace herního objektu

Samozřejmě je možné zachovat dříve představenou třídu `Mario` (kód 2.6), poté by se instance vytvářela stejně jako v případě hierarchie. To je jistě pohodlné. Další výhodou je to, že pokud objekt neumožní příjímání libovolných komponent, ale napevno si vytvoří množinu komponent při svém vzniku, je jisté, že objekt vždy bude mít všechny komponenty, které potřebuje. Tyto výhody jsou však zastíněny výhodami dosazování komponent z vně objektu. Díky dosazování komponent nemusíme vytvářet třídy pro všechny objekty, které se liší např. jen jednou komponentou. Je tedy snazší rekonfigurovat objekty. Možná výměna komponent při běhu ještě zvyšuje flexibilitu.

2.1.2 Komunikace mezi komponentami

Přestože bylo výše popsáno, jak se jednotlivé logické celky oddělí do zcela nezávislých komponent, bylo by naivní myslet si, že komponenty nepotřebují komunikovat mezi sebou. Například komponenta umělé inteligence bude komunikovat s komponentou spravující zdravotní stav hrdiny, aby se dle míry zranění mohla rozhodnout, jaké chování aplikuje. Stejně tak fyzikální komponenta pro fyzikální simulaci potřebuje někam zapsat aktuální polohu objektu, která bude využita k vykreslení.

Oproti aktualizaci v monolitické třídě, musí být aktualizace přes komponenty provedena velmi pečlivě. V případě monolitické třídy je aktualizace provedena jako předem stanovená posloupnost operací. Tato posloupnost není tak zřejmá, pokud je kód rozdělen do komponent. Snáze se tak přehodí aktualizace dvou komponent a zanese se do programu chyba. Prohození aktualizace fyzikální a grafické komponenty může nastat situace, kdy budeme v současném snímku vykreslovat hrdinu na pozici z minulého snímku. To může být problém zejména pro rychle se pohybující objekty.

2.1.2.1 Komunikace přes herní objekt

Ve výše uvedeném příkladě (kód 2.8) byla pro jednoduchost pozice přímo uložena v herním objektu a komunikace mezi fyzikální a grafickou komponentou byla vedena přes tato data. Tento přístup není příliš vhodný. Kdykoli by spolu potřebovaly libovolné dvě komponenty komunikovat, tak by musely používat data ve třídě `GameObject` jako prostředníka. Třída by pak neúměrně narůstala a vrátily by se problémy, kterým jsme se snažili vyhnout v první řadě. Třída by musela např. obsahovat proměnou `currentAnimationState`, do kterého by mohl být zapisován typ aktuální animace (dle směru pohybu a akce, kterou hrdina vykonává), dále by data obměňovala komponenta starající se o zdraví (upravení typu animace dle aktuálního zranění) a nakonec by data použila animační komponenta, která by danou animaci přehrála. Takováto metoda komunikace plní do jisté míry účel, avšak je nepraktická. `GameObject` se stane centralizovaným prvkem, a navíc by obsahoval zbytečná data - statický objekt by `currentAnimationState` vůbec nevyužil.

2.1.2.2 Přímá komunikace mezi komponentami

Dalším způsobem komunikace je přímé adresování jedné komponenty ze druhé. Ukažme si to na příkladu výše popisovaných animací. Mějme komponenty: `ActionComponent` (komponenta zodpovědná za vykonávání akcí - pohyb vlevo, vpravo, sebrání předmětu...), `HealthComponent` (komponenta spravující aktuální zdravotní stav hrdiny) a `AnimationComponent` (komponenta, která přehraje danou animaci, dle prováděné akce a aktuálního zdraví). Je zřejmé, že `AnimationComponent` je závislá na zbylých dvou komponentách. Její implementace by mohla vypadat jako na ukázce 2.10:

```
1  class AnimationComponent
2  {
3  public:
4    AnimationComponent(ActionComponent & ac, HealthComponent & hc):
5        m_actionComponent(ac),
6        m_healthComponent(hc)
7    {}
8
9    void Update(Graphics & graphics)
10   {
11     ActionType at   = m_actionComponent.getCurrentAction();
12     unsigned health = m_healthComponent.getCurrentHealth();
13
14     Animation currAnimation = m_animations.getByType(at);
15
16     if      (health > 80) currAnimation.setModification(NORMAL);
17     else if(health > 30) currAnimation.setModification(WOUNDED);
18     else                 currAnimation.setModification(ALMOST_DEAD);
19
20     //play animation...
21   }
22
23 private:
24   ActionComponent & m_actionComponent;
25   HealthComponent & m_healthComponent;
26
27   AnimationTable m_animations;
28 };
```

Kód 2.10: Ukázka animační komponenty

Animační komponenta tedy muže být vytvořena jedině tak, že ji do konstruktoru poskytneme další potřebné komponenty. Při aktualizaci pak z `ActionComponent` zjistíme aktuální vykonávanou akci a z `HealtComponent` zjistíme aktuální zdraví. Dle prováděné akce poté vybereme patřičnou animaci z tabulky animací. Takovou animací může být například běh. Dle zranění upřesníme, jakou animaci běhu požadujeme (normální běh, běh zraněné postavy, či běh umírající postavy) a poté můžeme zvolenou animaci přehrát. Tento příklad je velmi nereálný, slouží však pouze k demonstraci komunikace mezi komponentami.

Komunikace přímo mezi komponentami je jednoduchá a rychlá, komponenty máme přimo k dispozici, můžeme tedy přímo volat jejich metody. Obrovskou nevýhodou však je úzká vazba komponent. `AnimationComponent` je nyní napevno spojena s `HealthComponent` a `ActionComponent`. To zamezuje opětovnému použití komponenty v mírně odlišné hře.

2.1.2.3 Komunikace pomocí zpráv

Komunikace pomocí zpráv je nejkomplexnějším řešením. Spočívá ve vybudování systému zpráv, pomocí kterých budou komponenty komunikovat. Všechny komponenty, které mají umožňovat komunikaci, budou dědit ze třídy `Component`(2.11), která jejich rozhraní bude rozšiřovat o metodu `receive(int message, void * data)`.

```
1 class Component
2 {
3 public:
4     virtual void receive(int message) = 0;
5 };
```

Kód 2.11: Komponenta přijímá zprávy

Nyní je možné zadefinovat určitou zprávu, na kterou mohou ostatní komponenty reagovat. Např. fyzikální komponenta po provedení simulace odešle zprávu (odeslání zprávy vyvolá zavolání metody `recieve` na všech komponentách) `MSG_PLAYER_MOVED`, spolu se kterou odešle aktuální pozici hráče. V grafické komponentě je možné tuto zprávu přijmout a vykreslit hráče na správné pozici. Toto je velmi jednoduchý komunikační systém, který však zachovává nezávislost komponent. Problematičtější situace však nastávají, pokud při aktualizaci jedné komponenty požaduji určité informace z komponenty jiné. Zprávami bychom museli nejprve požádat druhou komponentu o požadovaná data a poté čekat, než obdržíme zprávu o tom, že nám jsou data posílána. Více o komunikaci pomocí zpráv bude uvedeno dále v textu.

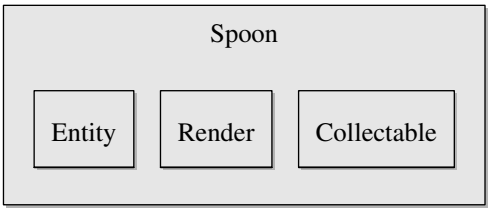
Shrnutí

Autor přistupuje k použití komponent tak, že z původního herního objektu odděluje logické celky. Odděluje data i funkce, které se týkají správy jedné vlastnosti, např. vykreslování. Každý logický celek je oddělen do nové třídy, jejíž instance je poté agregována v původním herním objektu. Mezi oddělenými logickými celky je možné komunikovat přímým odkazováním na jiný logický celek, čímž vzniká těsnější vazba mezi dvěma celky. Další možností komunikace je rozesílání a zachytávání zpráv.

2.2 Správa objektů pomocí komponent - Druhý přístup

Dalším přístup ke správě herních objektů pomocí komponent byl zveřejněn v knize Game Programming Gems 5 [23]. Autor popisuje problémy spojené s hierarchií a navrhuje je řešit zavedením komponent. Dle autora je komponenta třída, která obsahuje všechna členská data a metody použité pro danou úlohu.

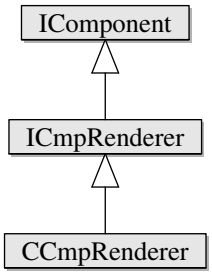
Příkladem by mohla být lžíce, která je složena ze tří komponent. První komponentou je `Entity`, která umožní umístění objektu do herního světa. Druhou komponentou je `Render` komponenta, která umožňuje vykreslení objektu, tedy například přiřazení objektu určité povrchové mřížky (*mesh*) atp. Poslední komponentou je `Collectable` komponenta, která umožňuje sebrání objektu a jeho umístění do inventáře.



Obrázek 2.1: Lžíce jako složený objekt

2.2.1 Podoba komponent a herního objektu

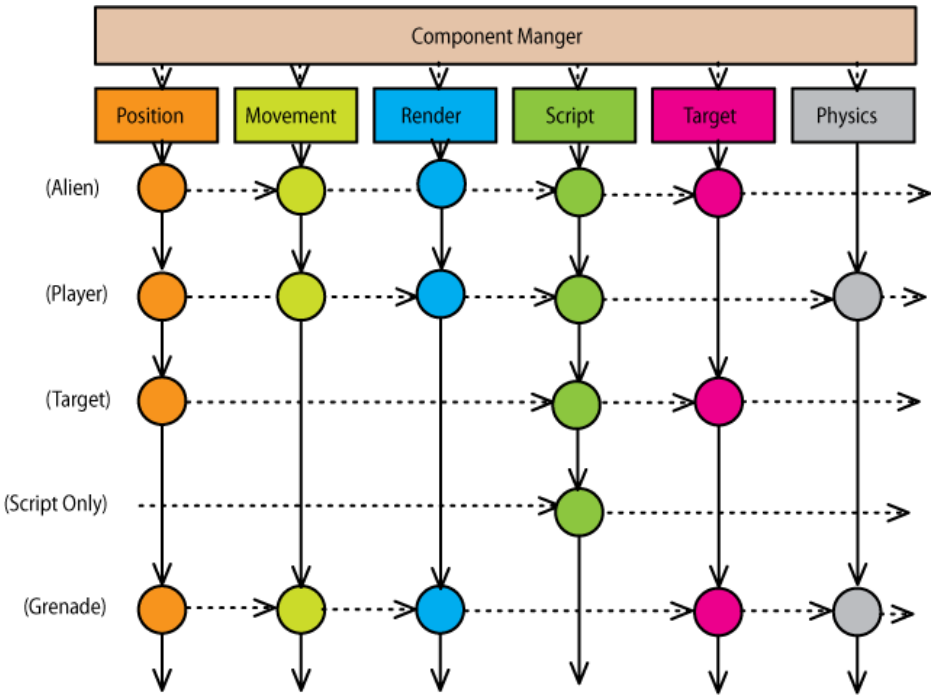
Všechny komponenty budou dědit ze společného rozhraní `IComponent`, které bude obsahovat potřebné metody pro správu komponent. Dále také bude vytvořeno pevné rozhraní komponenty pro daný úkol. Příkladem může být vykreslovací komponenta (ukázka 2.2).



Obrázek 2.2: Ukázka rozhraní komponent

Vidíme, že tedy máme společné rozhraní `IComponent`, ze kterého dědí `ICmpRenderer`. `ICmpRenderer` je rozhraní všech vykreslovacích komponent. Bude obsahovat např. metodu `render()`. Teprve z tohoto rozhraní dědí `CCmpRenderer`, neboli samotná implementace komponenty. Tímto přístupem získáme možnost snadno zaměňovat vykreslovací komponenty za jiné, bez nutnosti měnit kód, který vykreslovací komponentu používá (ten přistupuje pouze k `ICmpRenderer`). Na první pohled se může zdát, že takového dědění opět povede k problémům, které se týkaly hierarchie tříd. Autor však uvádí, že takto budou vznikat pouze úzce zaměřené množiny tříd a úkolů a problémy by tedy nastat neměly.

Vraťme se nyní k příkladu se lžící. Viděli jsme, že lžíce je složena ze tří komponent, ale ještě nebylo řečeno, co samotná lžíce je, jak je reprezentována v kódu. Zde se autor shoduje s Mickem Westem, autorem článku *Evolve Your Hierarchy* [26]. Oba navrhují, aby neexistovala žádná třída `GameObject`, ale aby herní objekt byl pouhou agregací požadovaných komponent, tedy aby byl objekt reprezentován pouze pomocí unikátního identifikátoru. K tomuto identifikátoru budou připojeny požadované komponenty, např. pomocí správce komponent (`ComponentManager`).



Obrázek 2.3: Skládání objektů z komponent zobrazeno v mřížce (převzato z [26])

Pospojování komponent ve výsledný objekt naznačuje obrázek 2.3. Vidíme, že obrázek strukturou připomíná tabulku. Na příkladu je ukázáno, jak sestavovat objekty pomocí pěti komponent - poziční komponenty (*Position*), pohybové komponenty (*Movement*), vykreslovací komponenty (*Render*), script komponenty (*Script*), komponenty, která umožňuje, aby na daný objekt mohl hráč zamířit (*Target*) a fyzikální komponenty (*Physics*). Například hráč (*Player*) ve hře pro jednoho hráče bude mít všechny komponenty, až na komponentu *Target* (nemůže mířit sám na sebe). Dalším objektem může být pomyslný statický cíl, ten má pozici, může mít přiřazen určitý skript a hráč ho může zamířit. Nemůže se hýbat, není vidět a nemá fyzikální reprezentaci. Přestože jde o poměrné vykonstruovaný příklad, je zřejmé, že takovéto objekty mohou vznikat pouhým pospojováním komponent, kdežto v hierarchii by musely mít někde své místo. Jak tedy reprezentovat objekt pouze pomocí identifikátoru? Identifikátor může být například řádek v popisované tabulce. Mimozemšťan (*Alien*) by tedy měl identifikátor 0 (předpokládejme indexování od nuly), hráč identifikátor 1 atd.

2.2.2 Komunikace mezi komponentami

Komunikační prostředky jsou velmi podobné jako v prvním způsobu (část 2.1.2.3). Máme opět volbu adresovat komponenty přímo, nebo použít systém zpráv.

V prvním případě však nebudeme adresovat komponenty tak přímo, jak již bylo uvedeno, ale budeme adresovat pouze jejich rozhraní, ze kterého dědí. Můžeme tedy požádat herního správce objektů o ukazatel na patřičné rozhraní daného objektu a volat jeho metody.

V případě komunikace pomocí zpráv autor navrhuje robustnější systém, než výše popisovaný. Systém bude umožňovat rozeslat zprávu všem objektům (`BroadcastMessage`), ale také adresovat přímo určitý objekt (`SendMessage`). Jednotlivé komponenty budou zaregistrovány na typ zpráv, které je zajímají a budou je obsluhovat. Po přijmutí zprávy navrátí stav, s jakým zpracování zprávy proběhlo (např. MR_FALSE, MR_TRUE, MR_IGNORED).

Autor se ve své implementaci vydal cestou komunikace pomocí zpráv. Její konkrétní použití bude popsáno v další sekci.

2.2.3 Implementace

Nyní se dostáváme k jednomu z nejdůležitějších faktorů metody a tím je její implementace. Autor v článku uvádí větší množství kódu a celou ukázkovou implementaci dodává na CD přiloženém ke knize. V textu tedy proto nebude uvedeno, jak takový systém naimplementovat, ale bude zde ukázáno, jak se se systémem pracuje, tedy jak vytvářet komponenty a jak s nimi pracovat. Pro každý systém objektů jsou toto dvě nejdůležitější věci. Použití musí být pohodlné, při vytváření komponent by mělo vznikat co nejméně kódu, který přímo nesouvisí s danou komponentou a to celé by mělo být poháněno efektivním systémem.

Vše bude ukázáno na příkladu. Mějme objekt, u kterého nás zajímá jeho zdravotní stav. Abychom ho řádně reprezentovali ve hře, je třeba vytvořit `HealthComponent`, která:

- umožní udržovat aktuální zdravotní stav objektu
- umožní dotazy na aktuální zdraví objektu
- upraví aktuální zdraví, pokud obdrží zprávu `MT_TAKE_HIT`
- odešle zprávu MT_HEALTH_DEPLETED, pokud aktuální hodnota zdraví klesne pod nulu

Jak již bylo řečeno, nebude zde popsán systém uvnitř, ale pouze jeho používání, třída `IComponent` zde tedy popsána nebude. První, co musí uživatel udělat, je vytvořit rozhraní `ICmpHealth`, které z `IComponent` dědí.

```
1 class ICmpHealth : public IComponent
2 {
3 public:
4     virtual int GetHealth()=0;
5 protected:
6     static void RegisterInterface(EComponentTypeId);
7 };
```

Kód 2.12: Vytvoření rozhraní pro zdravotní komponenty

Díky vzniklé metodě `GetHealth()` nyní můžeme komunikovat s libovolnou komponentou spravující zdraví přes tuto metodu. Implementace druhé metody vypadá následovně:

```
void ICmpHealth::RegisterInterface(EComponentTypeId compId)
{
    GetObjectManager().RegisterInterfaceWithComponent(
                                        IID_HEALTH,
                                        compId);
}
```

Kód 2.13: Implementace registrace rozhraní

Funkci je nutné zavolat z inicializace každé komponenty, která splňuje uvedené rozhraní. Funkce říká správci objektů, že daná komponenta implementuje rozhraní `IID_HEALTH`.

Přesuňme se k definici třídy `CCmpHealth`, která je již samotnou implementací představeného rozhraní.

```
class CCmpHealth : public ICmpHealth
{
public:
  //static methods
  static void          RegisterComponentType();
  static IComponent*   CreateMe();
  static bool          DestroyMe(IComponent*);

  //from IComponent interface
  virtual bool              Init(CParameterNode &);
  virtual void              Deinit();
  virtual EMessageResult    HandleMessage(const CMessage &);
  virtual EComponentTypeId  GetCmpTypeId() { return CID_HEALTH; }

  //from ICmpHealth intervace
  virtual int GetHealth() { return m_health; }
private:
  int m_health;
};
```

Kód 2.14: Konkrétní implementace zdravotní komponenty

Metoda `GetHealth()` a členská proměnná `m_health` je nepostradatelným prvkem této třídy, o tom není pochyb. Také metoda `HandleMessage(const CMessage&)` je nepostradatelná, protože řada v ní bude komponenta reagovat na zprávy a spravovat zdraví. V komponentě je však plno dalších metod, které je nutné implementovat, aby celý systém fungoval. Jsou to metody na vytváření a destrukci objektu, registraci objektu, jeho inicializaci a deinicializaci. Pro ukázku bude ukázána implementace dvou vybraných metod:

```
void CCmpHealth::RegisterComponentType()
{
    ICmpHealth::RegisterInterface(CID_HEALTH);
    GetObjectManager().RegisterComponentType(
                            CID_HEALTH, CCmpHealth::CreateMe,
                        CCmpHealth::DestroyMe, CHash("Health"));
    GetObjectManager().SubscribeToMessageType(CID_HEALTH,
                                              MT_TAKE_DAMAGE);
}

IComponent * CCmpHealth::CreateMe() { return new CCmpHealth; }
```

Kód 2.15: Ukázka nutné nežádoucí implementace

Vidíme, že kód komponenty nabobtnává kódem, který je sice třeba, ale již se netýká přímo správy zdraví (registrace komponenty, vytváření komponenty...). Obdobný kód se bude "duplikovat"ve všech komponentách, což zřejmě není vhodné. Podívejme se ale na výsledek, tedy jak komponenta interaguje s pomyslným objektem.

```
EMessageResult CCmpHealth::HandleMessage(const CMessage & msg)
{
  int newHealth;
  switch(msg.type)
  {
  case MT_TAKE_DAMAGE:
      newHealth = m_health - reinterpret_cast<int>(msg.mpData);
      if(newHealth <= 0 && m_health > 0)
      {
          GetObjectManager().PostMessage(
                                 getObjectId(), MT_HEALTH_DEPLETED);
      }
      m_health = newHealth;
      return MT_TRUE;
  }

  return MT_ERROR;
}
```

Kód 2.16: Ukázka zpracování zprávy

Po přijetí zprávy `MT_TAKE_DAMAGE` víme, že data ve zprávě obsahují množství zdraví, které máme odebrat. Odebereme ho tedy, a pokud jsme přešli přes nulovou hranici, odešleme zprávu `MT_HEALTH_DEPLETED`, na kterou již může reagovat další komponenta. Navrátíme hodnotu dle průběhu zpracovávání zprávy. Tento způsob práce s komponentou je velmi pohodlný. Nebýt příkazu `reinterpreter_cast`, tak kód neobsahuje jediný příkaz, který by byl "navíc".

Autor také popisuje, že jeho systém je velmi vhodný pro *data-driven* aplikace, neboli aplikace řízené daty, která se načtou např. po startu aplikace. Objekty vůbec nemusejí být vytvářeny z kódu. Stačí, aby se každá komponenta uměla sama inicializovat z dodaného bloku dat (např. část XML, či CParameterNode v případě autorova přístupu). Pak je možné objekty definovat v externím souboru, ten načíst a patřičné bloky posílat do konstruktorů daných komponent.

Shrnutí

Tato metoda přistupuje ke komponentám jako k logicky oddělitelným částem, které jsou zapouzdřené do třídy odvozené ze třídy `IComponent`. Na rozdíl od přístupu 2.1 neexistuje třída reprezentující herní objekt. Komponenty jsou uloženy a spravovány v autorem vytvořeném systému. Tento přístup umožňuje přidávání libovolných komponent bez zásahu do existujícího systému. Vytvoření systému pro správu komponent je však mnohonásobně náročnější. Autorův systém pro správu komponent má také určité nedostatky (uživatel musí v komponentě implementovat metody, které přímo nesouvisí s implementovaným chováním).

2.3 Správa objektů pomocí komponent - Další přístupy

V této sekci budou představeny další přístupy k používání komponent pro správu herních objektů. Přístupy nebudou popsány tak podrobně, jako přístupy v sekcích 2.1 a 2.2. Budou představeny především hlavní myšlenky.

Buchananův přístup

Warrick Buchanan popsal v knize [25] přístup, který je postaven na komunikaci přímo mezi objekty (viz. výše v textu). Třída, která má reprezentovat objekt složený z komponent, dědí ze třídy `Component`, která obsahuje šablonovou metodu `QueryForInterface()`. Pomocí této metody můžeme v rámci jednoho objektu získat rozhraní poskytující požadovanou funkcionalitu.

Ukažme si to na příkladu objektu, kterému chceme umožnit pohyb. Vznikne rozhraní `Movable`, které bude splněno ve třídě `Player`, viz. kód 2.17

```
1  struct Movable
2  {
3      virtual void GetPosition(float & x, float & y) = 0;
4      virtual void SetPosition(float x, float y) = 0;
5  };
6
7  struct Player : Component, Movable
8  {
9      float m_x, m_y;
10
11     void GetPosition(float & x, float & y) { x = m_x; y = m_y; }
12     void SetPosition(float x, float y) { m_x = x; m_y = y; }
13 };
```

Kód 2.17: Buchanan - rozhraní `Movable`

Na výpisu kódu 2.18 je ukázáno, jak získat požadované rozhraní z instance objektu, se kterým pracujeme. Slouží k tomu metoda `QueryForInterface()`. Detaily k implementaci metody lze dohledat ve zdrojových kódech ke knize [25]. Obdobná metoda (tedy metoda vyhledávající dle datového typu) bude představena v kapitole 3.

```
1 Movable * movable = player->QueryForInterface<Movable>();
2
3 if(movable)
4 {
5     movable->SetPosition(10.0f, 12.0f);
6 }
```

Kód 2.18: Buchanan - komunikace přes získané rozhraní

Namísto dědičností, `Player` může implementovat pohyb kompozicí přes pomocný objekt (přes *prostředníka*) `MovableHelper` viz. kód 2.19. `MovableHelper` půjde znovu použít v libovolném dalším objektu. Poté se s instancí pracuje např. `p.m_movableHelper.GetPosition()`.

```
1 struct MovableHelper : Movable
2 {
3     float m_x, m_y;
4
5     void GetPosition(float & x, float & y) { x = m_x; y = m_y; }
```

```
6       void SetPosition(float x, float y) { m_x = x; m_y = y; }
7   };
8
9   struct Player : Component
10  {
11      MovableHelper m_movableHelper;
12      //...
13  };
```

Kód 2.19: Buchanan - prostředník

Stoyův přístup

Další metodu, kterou představil Chris Stoy [6], si vysvětlíme na příkladu kódu. Stoy navrhuje zavedení třídy `GameObject`, která bude obsahovat tabulku komponent, ze kterých je objekt složen. Bude také poskytovat metody pro práci s tabulkou (vyhledání komponenty, přidání komponenty...). Dále představuje třídu `Component`, která obsahuje metody pro správu typu komponenty. Komponenty řadí do *rodin*. Vznikne tedy např. rodina vykreslovacích komponent, které budou vzájemně zaměnitelné pro vykreslování, protože všechny budou obsahovat metodu `render()`. Přístup je naznačen na volně převzatých ukázkách kódu. Kód 2.20 ukazuje `GameObject` a `Component`, kód 2.21 představuje příklad využití těchto struktur pro vykreslovací komponenty a kód 2.22 ukazuje použití při vykreslování.

```
1   struct GameObject
2   {
3       Component * getComponent(std::string & family_id);
4
5       std::map<std::string, Component*> m_componentMap;
6   };
7
8   class Component
9   {
10      virtual const std::string componentId() = 0;
11      virtual const std::string familyId() = 0;
12  };
```

Kód 2.20: Stoy - základy

```
1   struct VisualComponent : public Component
2   {
3       virtual const std::string familyId() { return "VISUAL"; }
4
5       virtual void render() = 0;
6   };
7
8   struct SphereVisualComponent : public VisualComponent
9   {
10      virtual const std::string componentId() { return "VISUAL_SPHERE"; }
11
12      float m_radius;
13
14      virtual void render() { /*render sphere*/ }
15  };
```

Kód 2.21: Stoy - vykreslovací komponenty

```
for (unsigned i = 0; i < gameObjects.size(); ++i)
{
    Component * c = gameObjects[i].getComponent("VISUAL");
    VisualComponent * vc = static_cast<VisualComponent*>(c);

    if(vc)
    {
        //we don't need to know what VisualComponent this is
        vc->render();
    }
}
```

Kód 2.22: Stoy - použití při vykreslování

Harmonův přístup

V této sekci také musíme zmínit Matthewa Harmona [16]. Ve článku *A System for Managing Game Entities* představuje systém, kde je vše založeno na zprávách a herní objekt je pouze datová přepravka. Datovou přepravkou může být např. struktura v C++ (`struct`). Autor ukazuje přístup na příkladu raketové střely. Vytvořil přepravku, která obsahovala pouze směrový vektor, translaci a akumulátor vektoru rychlosti. Jinde v kódu vytvoří funkci, ve které se budou zpracovávat zprávy týkající se právě raketové střely. V případě zprávy `MSG_CREATE` vytvoří raketu, v případě `MSG_SETPOS` upraví pozici rakety atd. Přestože celý přístup trpí problémy, které byly popisovány výše (podobný objekt bude duplikovat kód, atd.), přináší dvě značné výhody.

První z nich je způsob komunikace zprávami v oddělené části programu. To přináší několik výhod:

1. **Homogenita:** Nejrůznější herní entity mohou být spravovány stejným systémem.
2. **Oddělení funkcionality:** Pomáhá vynutit oddělení vykreslovací části od části simulační a logické, v případě síťové hry mezi klientským a serverovým kódem.
3. **Jeden přístupový bod:** Veškerá kontrola herních entit je prováděna přes jeden jediný přístupový bod - `EntitySendMessage`.
4. **Rozšiřitelnost:** Přidání nových vlastností je stejně jednoduché jako vytvoření nové zprávy. Již hotové entity mohou novou zprávu prostě ignorovat, aniž by se rozbilo jejich původní chování.
5. **Abstrakce:** Proces vystřelení kulky bude totožný jako start řízené rakety, jen pošleme zprávu jiné třídě.

Druhou výhodou, které přístup přináší, je možnost znovu použít části kódu. Kód týkající se celého jednoho herního objektu je obtížné znovu použít. Pokud ale provedeme rozdělení datové části a funkcionality, je možné jednoduše znovu použít datovou část. Chování rakety se může v příští hře zásadně změnit, může se vykreslovat jinak, simulovat jinak, ale zcela jistě ji zůstanou data, která jsou v datové přepravce (translace, vektor rychlosti, ...)

Shrnutí

V této sekci jsme si uvedli tři přístupy. Přístup dle Buchanana. Přístup byl postaven na skrývání funkcionality za rozhraní, která konkrétní herní objekt implementuje (přímo, či přes prostředníka). Stoyův přístup zavedl používání rodin komponent, které jsou zcela zaměnitelné. Harmonův přístup striktně odděluje data popisující objekt od kódu řešícího logiku věci.

Pro další přístupy a myšlenky je možné studovat literaturu [2, 5, 7, 8, 9, 14, 21, 18, 20].

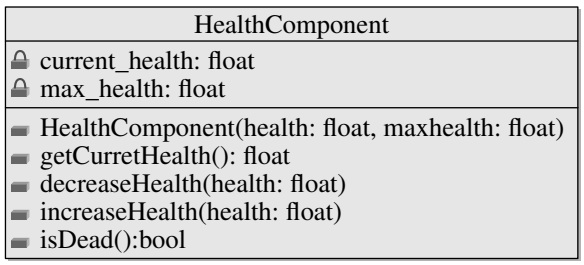
2.4 Správa objektů pomocí komponent - Systémy entit

V této sekci bude představen pokročilý systém pro komponentovou správu herních objektů. Dále v textu bude proveden detailní návrh, popsána implementace a testování systému. Je také nutné oddělit dva pojmy. Prvním pojmem je *systém*. Slovo systém bývá použito jako abstraktní pojmenování určitého kusu softwaru. Kus může být libovolně velký, můžeme mít např. *systém na správu objektů ve scéně*, tedy takovou část kódu, která obstarává přidávání herních objektů, jejich mazání, ukládání atd. Druhým pojmem je systém entit, neboli *Entity System*, což je pojmenování pro část kódu, kde může uživatel implementovat funkcionalitu související s konkrétními komponentami. Detailnější popis je uveden v následujících odstavcích.

Nyní popíšeme další možnost využití komponent. Hlavním prvkem tohoto způsobu je systém entit - *Entity System*. Základní princip systému entit byl popsán v [15] a [3]. Myšlenka použití systému entit je taková, že se herním objektům oddělí datová část a funkcionalita (to je proces, na který mnoho programátorů není zvyklých, z pohledu objektově orientovaného programování jsou zvyklí zapouzdřovat data i logiku do jednoho logického celku). Datová část bude uložena v komponentách a funkcionalita bude prováděna právě v systémech entit. Vše bude nyní detailněji vysvětleno a představeno na příkladech.

Mějme situaci, kdy potřebujeme reprezentovat zdravotní stav nějakého herního objektu - entity. Nezávisle na hře je jasné, že budeme potřebovat aktuální hodnotu zdraví a maximální hodnotu zdraví (např. pokud hráč nalezne lékárničku, která zvýší hodnotu zdraví o 80 ze 100 a hráč má momentálně hodnotu zdraví 50, nechceme, aby nová hodnota byla 130 (50+80)). Toto chování je zcela nezávislé na konkrétní hře. Co je ale závislé, je funkcionalita okolo těchto dat. V jedné hře můžeme kontrolovat, zda neklesla hodnota zdraví pod nulu, a pokud ano, objekt zemře - odstraníme ho ze hry. V jiné hře můžeme chtít navíc kontrolovat, jestli zdraví nekleslo pod hodnotu 30 a na základě toho změnit chování entit ve hře. To jsou věci, které není možné navrhnout obecně a využívat v každé hře. V následujícím odstavci bude popsáno, jak toto realizovat probíranou technikou.

Nejprve data, které jsou znovupoužitelná, přemístíme do oddělené komponenty. V tomto případě vznikne `HealthComponent`, která bude vypadat jako na ukázce 2.4. Komponenta obsahuje data nezávislá na hře, ke kterým je umožněn přístup přes poskytnuté metody. Pokud bychom chtěli komponentu využít i pro reprezentaci rozbití objektu (např. automobilu) a nechtěli bychom ho nazývat *health*, je možné pouze přidat do komponenty funkce `getCurrentDamage`, `increaseDamage` atd., které budou vnitřně používat proměnné `current_health`. Tím ještě rozšíříme možnost znovupoužití komponenty. Komponenta sama o sobě však nic nedělá. Pro přidání funkcionality je zapotřebí *EntitySystem*.



Obrázek 2.4: Možná podoba zdravotní komponenty

Systém entit je část kódu, kde jsou entity aktualizovány. Každý systém entit je zodpovědný za aktualizaci jednoho typu funkcionality. Při založení systému entit stanovíme, jaké entity se v něm mají aktualizovat. To se zrealizuje tak, že se provede požadavek na komponenty - systém entit bude požadovat, aby entity měly zadané komponenty. Může požadovat např. jednu komponentu, ale i třeba šest komponent. Do tohoto systému entit pak budou přiřazeny takové entity, které vyjmenované komponenty obsahují.

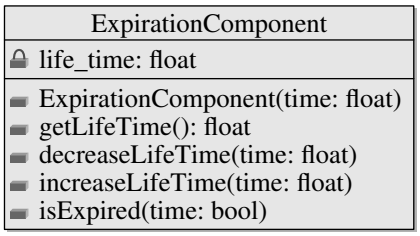
Při aktualizaci entity je možné používat komponenty, které byly stanoveny v požadavku. Je možné z nich data číst, ale také zapisovat. Jedna entita tedy bývá aktualizována ve více než jednom systému entit. Jeden systém entit např. aktualizuje její polohu na základě vstupu, další změní animaci na základě prováděné akce, další vyřeší případné kolize, další zkontroluje zdraví atd... Nejen, že jedna entita je aktualizována přes více systémů entit, ale také jedna komponenta příslušící jedné entitě může být vyžadována ve více systémech entit.

Představme si hrdinu, se kterým pohybujeme pomocí vstupu např. z klávesnice. Hrdina má grafickou podobu a je animovaný. Při chůzi je tedy přehrávána animace chůze a při překonávání překážky (např. zdi) je přehrávána animace přelézání. Pokud zmáčkneme klávesu pro skok, hrdina vyskočí a opět dopadne na zem. Pokud ale hrdina stojí u zdi, při výskoku se chytne vršku zdi a vyleze na ni. V jednu chvíli je tedy pozice hrdiny řízena uživatelským vstupem, jindy je řízena animací. V průběhu animace lezení se bude jeho pozice měnit (bude stoupat), až nakonec hrdina bude na zdi. Kdybychom pouze přehráli animaci lezení a neměnili pozici, hrdina by zůstal před zdí.

V tomto příkladě by byly důležité tři komponenty. V první komponentě by byla uložena pozice hrdiny a jmenovala by se např. `PositionComponent`. V druhé komponentě by byly ukládány požadavky na pohyb získané ze vstupu a mohla by se jmenovat `MoveRequestComponent`. Třetí komponenta by se jmenovala `AnimationComponent` a uchovávala by stavy animací, možné animace apod. Pro implementování popsaného chování bychom vytvořili dva systémy entit. První by hýbal s hrdinou dle kláves, které hráč mačká. Požadavek by tedy byl na `PositionComponent` a `MoveRequestComponent`. Veškeré entity obsahující tyto dvě komponenty by se zde aktualizovaly (nejčastěji to bude právě jedna entita - hlavní hrdina), byly by přečteny požadavky na pohyb a změnila by se pozice hrdiny. V druhém systému entit by byl požadavek na `PositionComponent` a `AnimationComponent`. Tam by se aktualizovaly entity, které tyto komponenty obsahují (již to kromě hrdiny můžou býti i nepřátelé). Při aktualizaci se zkontroluje stav animace a pokud se přehrává animace lezení, pozmění se pozice. Jedna komponenta může být tedy aktualizována ve více systémech entit.

Ukažme si to celé ještě na příkladu projektilu (představen v 1.1). Připomeňme, že projektil potřebuje expirační, fyzikální a poziční komponentu. Expirační komponenta by mohla

vypadat jako na obrázku 2.5. Dále by byl vytvořen systém, který by aktualizoval objekty s expirační komponentou, jak je zobrazeno na ukázce kódu 2.23. Objekt se vytvoří složením z jednotlivých komponent - kód 2.24.



Obrázek 2.5: Příklad expirační komponenty

```
for all GameObjects GO which have ExpirationComponent
{
    GO.decreaseLifeTime(dt)
        if(GO.isExpired())
        {
            remove GO from game world
        }
}
```

Kód 2.23: Ukázka expiračního systému entit

```
GameObject createProjectile ( def )
{
    GameObject p;
    p.add(ExpirationComponent(def.lifetime));
    p.add(PhysicsComponent(def.shape));
    p.add(SpatialComponent(def.transformation));
    return p
}
```

Kód 2.24: Vytvoření složeného objektu

Za předpokladu, že by byly obdobně doimplementovány komponenty *PhysicsComponent*, *SpatialComponent* a příslušné systémy pro aktualizace, je reprezentace jednoduchého projektilu hotova. Se složeným objektem se ideálně komunikuje prostřednictvím komponenty, tedy datové části, zatímco logická část zpracování objektu zůstává nezměněná. Vystřelení projektilu může být realizováno např. jako na 2.25.

```
shoot(direction)
{
    Weapon w = getCurrentWeapon;
    GameObject projectile = createProjectile(w.projectileDefinition);
    projectile.getPhysicComponent().addLinearForce(
                                        w.muzzleVelocity * direction);
}
```

Kód 2.25: Vystřelení projektilu

Zavoláním funkce *shoot* je již projektil vyslán do světa a systém se k němu dále chová pouze jako ke shluku komponent, které jsou separátně aktualizovány. Vzhledem k tomu,

že jsme již vytvořili expirační komponentu a její aktualizaci, lze nyní libovolnému hernímu objektu omezit životnost a to pouhým přiřazením komponenty.

Představme si např. hru typu RPG, kde hrdina vyčaruje ochrannou bariéru. Bude třeba vytvořit komponenty reprezentující její vzhled a případné další efekty, ale základní bariéru můžeme složit z již hotových komponent. Přiřadíme jí transformaci pomocí `SpatialComponent` a fyzikální reprezentaci pomocí `PhysicsComponent`. Pokud kouzlo není permanentní a bariéra má po určité době zmizet, doplníme do kódu jeden jediný řádek: `barrier.addExpirationComponent(time)`. Touto změnou se bariéra zařadí mezi objekty které mají být po určité době smazány a toto smazání se děje zcela automaticky. Stejně jednoduše libovolnému objektu přiřadíme fyzikální komponentu, nebo komponentu reprezentující zdraví.

Tento přístup je uživatelsky velmi příjemný. Komponenty lze používat znovu v dalších projektech, aktualizace jsou logicky oddělené a celkově se zpříjemnila práce s herními objekty. Programátor, který má např. opravit konkrétní chování, se může zaměřit pouze na vybrané komponenty a jeden, či více systémů. Pokud tedy opravuje umělou inteligenci nepřátel, nemusí vidět ani jediný řádek kódu, který řeší animace, fyzikální simulaci, aj.

Zde byl popsán systém pouze z hlediska uživatelského používání. Implementace metody je výrazně náročnější ve srovnání s výše popsanými metodami. Návrh a implementace však bude popsána v dalších kapitolách, kde budou také představeny konkrétní podoby systémů entit.

Kapitola 3

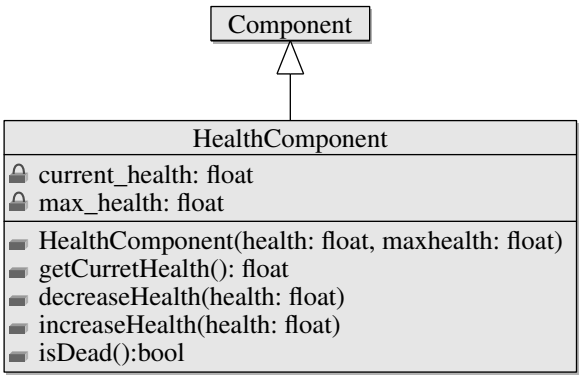
Návrh prototypu

V této kapitole bude popsána analýza a návrh vyvíjeného softwaru. Kapitola bude rozdělena na dvě hlavní části. Nejprve bude navržen komponentový systém herních objektů. V druhé části kapitoly bude navrženo zapojení komponentového systému do hry. Důležité pojmy z této kapitoly jsou také stručně vysvětleny ve slovníčku v příloze A.

3.1 Návrh komponentového systému herních objektů

Návrh bude představen po částech a poté bude ukázáno, jak jednotlivé části interagují. Návrh je tvořen s vědomím, že výsledná implementace bude provedena v C++. V návrhu jsou tedy zachyceny určité konstrukty, které budou využívat šablonového metaprogramování. Konkrétní implementace bude předvedena v další kapitole.

Výchozí třídou je třída `Component`. Třída funguje pouze jako společný předek všech uživatelských komponent. Představme si to na známém příkladu se zdravím. Vytvořenou komponentu `HealthComponent` zachycuje obrázek 3.1. Všimněte si, že uživatelská komponenta neobsahuje jedinou metodu či proměnou, která by úzce nesouvisela s logikou správy zdraví. Jediná věc, která se bude v komponentách opakovat, bude kód, říkající, že máme dědit, tedy kód ve stylu `class HealthComponent : public Component`. Tímto řádkem sdělíme systému *"Toto je komponenta"*.



Obrázek 3.1: Vytvoření zdravotní komponenty odvozením od třídy Component

Než přejdeme k představení reprezentace entit a entity systémů, musí být představen `Bitset`, protože je základním prvkem celého frameworku, který svazuje entity s komponentami a systémy entit. Jak již název napovídá, `Bitset` je uspořádaná množina bitů (bitset tedy může být implementován jako `int`). Počet bitů je pevně stanoven uživatelem dle rozsahu hry. Pomocí uspořádané množiny bitů lze ve frameworku reprezentovat například příslušnost komponent k entitě. Každá komponenta má pevně přiřazenou unikátní sekvenci bitů. Pokud tedy k entitě přiřadíme určitou komponentu, můžeme v sekvenci bitů patřičný bit nastavit na 1. Tím řekneme, že entita má danou komponentu. Je nutné, aby každá komponenta měla pevně stanovený unikátní bit. Z toho důvodu byla zavedena šablonová třída `ComponentType`, která obsahuje metodu `getBit`. Metoda pro jakoukoli komponentu vrátí sekvenci bitů, která ji reprezentuje a je unikátní napříč ostatními komponentami. Implementace této třídy je zajímavá a bude představena v další kapitole. Protože zaznamenávání příslušnosti komponent k entitě je stěžejním bodem celého systému, bude `Bitset`, operace s ním a třída `ComponentType` představena podrobněji.

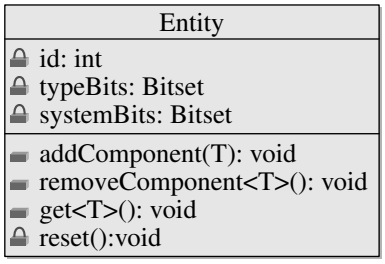
Třída `ComponentType` mimo poskytování sekvence bitů pro komponenty ještě poskytuje unikátní identifikátor každé komponenty. Vše si ukážeme na příkladu. Uvažujme velikost sekvence bitů stanovenou na čtyři. Uživatel vytvoří dvě komponenty, komponentu `A` a komponentu `B`. Pokud zavoláme `ComponentType<A>::getId()` získámé identifikátor s hodnotou `0`. Pokud zavoláme totéž pro komponentu `B` (`ComponentType<B>::getId()`) získámé identifikátor s hodnotou `1`. Pro získání sekvence bitů voláme metodu `getBit()`, přičemž volání pro komponentu `A` vrátí sekvenci `0001` a volání pro komponentu `B` vrátí sekvenci `0010`. Další komponenta by měla identifikátor o hodnotě `2` a sekvenci bitů `0100` (hodnota 2 se binárně nerovná `0100`, ale `0010`, sekvence bitů z identifikátoru se dá získat jako `1 « id`, kde « značí bitový posun vlevo). Vzniklé sekvence bitů lze uložit do jedné sekvence pomocí bitové operace `or`. Pokud má tedy entita uloženou sekvenci `0101`, znamená to, že má právě komponentu `A` a komponentu `C`.

Proč je vůbec potřeba `Bitset` a proč není popsán až v kapitole implementace? Protože pomocí sekvence bitů bylo navrženo efektivní získávání komponent v konstantním čase. Jak již bylo řečeno, efektivita celého systému je klíčová a je tedy nutné ji reflektovat již v návrhu. Jak tedy sekvence bitů zaručuje konstantní získávání komponent? Představme si hru, ve které používáme pouze 10 druhů komponent. Budeme tedy u každé entity ukládat, jaké komponenty agreguje. Může jich mít libovolný počet v rozmezí 0-10. Pomocí sekvence bitů můžeme říci, složením jakých komponenty entita vznikla. Pokud budeme hledat entity, které mají určitou komponentu, můžeme velmi jednoduše (pomocí bitových operací) rozhodnout, která entita komponentu obsahuje a která ji neobsahuje. I jednoznačný identifikátor napomáhá s manipulací s komponentami v konstantním čase. Je možné pro každou entitu vytvořit pole o délce 10 a v tomto poli ukládat konkrétní instance komponent(případně hodnotu `null`, pokud entita danou komponentu neobsahuje), které entitě náleží. Víme-li, že komponenta `X` má identifikátor `5`, můžeme v konstantním čase přistoupit do páté buňky pole a tím komponentu adresovat v konstantním čase.

Jak je reprezentována entita, tedy herní objekt? Entita je složena z komponent. Třída `Entity` však neobsahuje přímo v sobě uložené komponenty. Všechny komponenty budou uloženy na jednom centralizovaném místě v tabulce ve třídě `EntityManager`, která bude popsaná dále. Entita bude mít pouze odkaz na tuto třídu. Dále bude mít přiřazený unikátní identifikátor, množinu bitů udávající jaké obsahuje komponenty a množinu bitů udávajících,

k jakým systémům entit náleží. Systémy entit budou popsány dále, je ale třeba zmínit, že stejně jako komponenty, každý systém bude jednoznačně opatřen sekvencí bitů pomocí šablonové třídy `SystemType`. To je z datové části vše, co bude herní objekt, entita, obsahovat.

Třída `Entity` obsahuje metody na přidání, odstranění a získání komponenty. Protože entita přímo komponenty neobsahuje, budou volání zmíněných metod přesměrována na volání relevantních metod ve třídě `EntityManager`. Třída tak bude plnit úlohu proxy objektu. Je to z toho důvodu, že uživatel pracuje právě s instancemi třídy `Entity` a nemusí vědět, že třída `EntityManager` existuje. Třída zapouzdřuje správu entit, která se jednou vytvoří a uživatel by k ní neměl mít přístup (je to vnitřní třída knihovny). Entita má také metodu `reset()`, tato metoda bude popsána dále. Diagram znázorňující třídu `Entity` můžete vidět na obrázku 3.2.



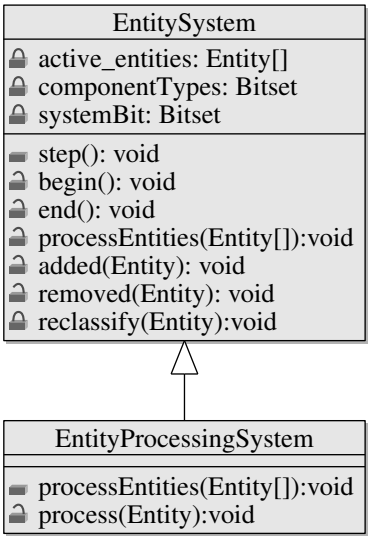
Obrázek 3.2: Třída Entity

Další prvek, se kterým bude interagovat uživatel, jsou systémy entit. Uživatel bude vytvářet vlastní systémy entit. Vytváření by mělo být co nejjednodušší, ideálně opět bez psaní kódu, který přímo nesouvisí s implementovaným chováním. Uživatel popíše, jaké entity má entity systém aktualizovat a napíše kód samotné aktualizace. Více by psát neměl. Třída `EntitySystem` uchovává pole entit, které k ní náleží. Dále má uloženou množinu bitů reprezentující typy komponent uvedených v požadavku. Důležitou metodou je metoda `step()`, tu volá uživatel v požadovaný okamžik (uživatel volá metodu `step` více systémů entit, je tedy na něm, aby určil pořadí aktualizací) a vede k aktualizaci systému entit. Třída `EntitySystem` je navržena tak, aby při vytváření nových systémů entit uživatel ze třídy dědil. Třída tedy obsahuje několik virtuálních metod, které může uživatel ve svém systému entit implementovat. Těmito metodami jsou:

- `begin()` Metoda volaná před aktualizací systému entit.
- `end()` Metoda volaná po aktualizaci systému entit.
- `added(Entity)` Metoda volaná, pokud je k tomuto systému entit přiřazena nová entita.
- `removed(Entity)` Metoda volaná, pokud je entita odstraněna z tohoto systému entit.

Pro uživatele nejdůležitější metodou, kterou může překrýt, je metoda `processEntities (Entity[])`. V této metodě uživatel dostane pole s podmnožinou entit, které splňují jeho požadavek a může s nimi dále nakládat. Je možné provádět operace nad polem, např. zjišťovat velikost pole (tím zjistíme kolik v herním světě existuje entit splňujících požadavek na komponenty) a dle toho měnit herní logiku (např. při malém počtu živých nepřátel můžeme vytvářet

další). Často ale uživatel bude polem iterovat a provádět operace s jednotlivými entitami. Z toho důvodu byla ze třídy `EntitySystem` odvozena třída `EntityProcessingSystem`, která prvky iteruje namísto uživatele a pro každý volá virtuální metodu `process(Entity)`, kterou může uživatel překrýt. Třídy `EntitySystem` a `EntityProcessingSystem` jsou zachyceny na obrázku 3.3.



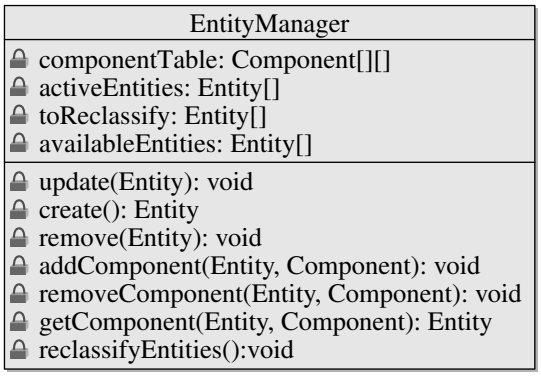
Obrázek 3.3: Třídy EntitySystem a EntityProcessingSystem

V následujících odstavcích bude popsána třída `EntityManager`. Je to jedna z nejdůležitějších tříd celého systému, protože má na starosti správu komponent i entit. Také je to třída, se kterou uživatel přímo nekomunikuje. Pro uživatele je tato třída skryta za voláními třídy `Entity` (viz. výše) a třídy `World` (bude popsána dále). Strukturu třídy můžete vidět na obrázku 3.4. Jednotlivé prvky třídy budou nyní vysvětleny. Třída je navržena tak, aby splnila požadavek na efektivitu, tedy na získávání požadovaných komponent k dané entitě v konstantním čase.

Nyní se zaměříme na ukládání komponent. Pro uložení komponent je využito dvoudimenzionální tabulky. Na pomyslné ose v jedné dimenzi jsou entity a na druhé komponenty. Přesněji řečeno, k uložení komponent je použito pole polí. Protože počet entit v herním světe není předem znám, musí být tabulka v jedné dimenzi rozšiřitelná. Komponenty jsou tedy uloženy v poli dynamických polí, o kterém však dále budeme hovořit jako o tabulce. Bylo řečeno, že každá entita má unikátní identifikátorů a každý typ komponenty má unikátní identifikátor. Tyto identifikátory jsou přirozená čísla a nula. Pomocí dvou identifikátoru lze z tabulky snadno v konstantním čase získat požadovanou komponentu. Pokud tedy chceme získat komponentu `A`, která náleží entitě `E`, získáme jí z tabulky na dvoudimenzionálním indexu `ComponentType<A>::getId()` a `E.getId()`.

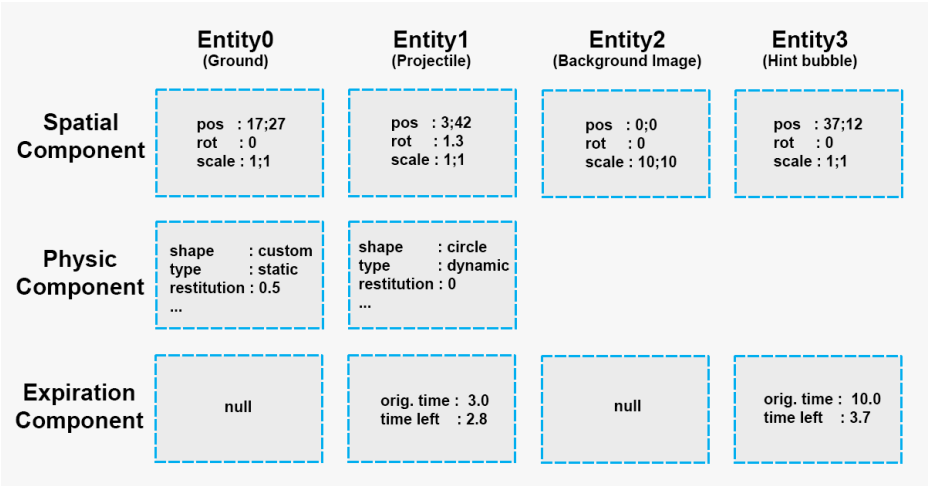
Samozřejmě mohou nastat situace, kdy entita danou komponentu nemá. V tom případě je v tabulce *prázdné místo*. Vzhledem k povaze tabulky bude velikost tohoto prázdného místa zanedbatelná. Pokud bychom měli např. 1000 herních entit a 32 možných komponent, při nejhorší možné situaci (ani jedna z entit nemá žádnou komponentu) bychom vyplýtvali 32000 tabulkových buněk. V buňce by mohl být uložen např. ukazatel `nullptr`, který na 32 bitovém

systému má velikost 4 byte. V tom případě bychom *plýtvali* 125kB paměti (4*32000/1024), což je vzhledem k velikosti dnešních pamětí zcela zanedbatelné množství. Pokud bychom se chtěli vyhnout nevyužité paměti, museli bychom mít komponenty uloženy způsobem, který vyžaduje prohledávání a neumožňuje tedy přístup s konstantní časovou složitostí.



Obrázek 3.4: Třída EntityManager

Ukázka tabulky komponent je na obrázku 3.5. Ukázka zachycuje situaci pro čtyři entity (zem, projektil, obrázek na pozadí, bublina s nápovědou) a tři komponenty (transformační, fyzikální, expirační). Každý řádek tabulky je reprezentován dynamickým polem, pokud mají fyzikální komponentu pouze první dvě entity, není důvod, aby bylo pole delší než dva prvky. Pokud by přibyla pátá entita, která by fyzikální komponentu měla, pole by se dynamicky zvětšilo. Před výběrem dat z tabulky se provádí kontrola, zda daná entita má požadovanou komponentu. Do tabulky je tedy přistupováno pouze na validní buňky. Pokud entita komponentu nemá, je v příslušné buňce pro úplnost uloženo *null*.



Obrázek 3.5: Ukázka tabulky komponent se třemi komponentami a čtyřmi entitami

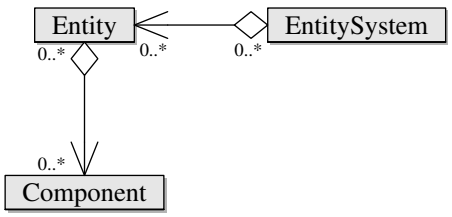
Problém s uvedeným přístupem však nastává tehdy, když entity ve hře zanikají a vznikají nové. Příkladem je vystřelení kulky. Pokud by se měla tabulka rozšiřovat dle identifikátoru entity, došla by paměť a vše by časem přestalo fungovat. Vystřelená kulka by např. měla

identifikátor 37, další kulka tedy 38 atd... Po chvíli hraní hry by mohlo dojít k tomu, že nově vystřelená kulka má identifikátor 30000. To by znamenalo, že pole, které uchovává komponenty, které používá kulka, by muselo být nejméně 30000 prvků dlouhé a bylo by takřka prázdné. A s každým dalším výstřelem se situace zhorší. Řešením je znovu používání entit.

Uvnitř třídy `EntityManager` budeme udržovat seznam aktivních entit (*activeEntities*), tedy entit, které jsou nyní ve virtuálním světě a seznam volných entit (*availableEntities*), tedy entit, které jsou k dispozici. Do seznamu entit k dispozici se zařadí každá entita, která bude vyřazena ze světa (a tedy i ze seznamu aktivních entit). Při odstranění entity ze světa se zavolá výše zmíněná metoda `reset()` uvnitř třídy `Entity`. Tato metoda je zodpovědná za to, že instanci třídy `Entity` uvede do původního stavu s tím, že zachová identifikátor. Při přidání nových entit (metodou `create()`) se nejprve provede kontrola, zda není k dispozici nějaká entita k opakovanému použití.

Další zodpovědností třídy `EntityManager` je přiřazování entit k existujícím systémům entit. Pokud je entita vytvořena a přidají se komponenty, je nutné zavolat metodu `update()`. Tím je entita uložena do seznamu `toReclassify`. Z tohoto seznamu jsou poté entity vyjmuty a opětovně zařazeny k systémům entit. Každý systém entit má uloženou sekvenci bitů, které udávají požadované komponenty. Obdobně instance třídy `Entity` má uloženo, které komponenty obsahuje. Jednoduchými bitovými operacemi lze tedy zjistit, zda entita má být k systému přiřazena, či odebrána. Tento proces se však netýká pouze nově vytvořených entit. I v průběhu hry je možné entitě komponenty přidávat či odebírat. Pokud např. hrdinovi odebereme komponentu zajišťující uživatelský vstup a přidáme komponentu umělé inteligence, entita se automaticky přeřadí do patřičných systémů entit a bude dále aktualizována jako postavička s umělou inteligencí (to je vhodné např. do kooperativních síťových her - pokud jede z hráčů přestane hrát, spoluhráči na něho nemusejí čekat a jeho pozici může dočasně zastoupit umělá inteligence).

Vztah mezi `Entity`, `EntitySystem` a `Component` je zachycen na obrázku 3.6. Entita agreguje libovolný počet komponent a stejný typ komponenty může náležet k libovolnému počtu entit. Systém entit agreguje pro aktualizaci libovolný počet entit a každá entita může zároveň náležet do libovolného počtu systémů entit.



Obrázek 3.6: Vztah mezi `Entity`, `EntitySystem` a `Component`

Přestože entity jsou zařazeny do systémů entit a tam jsou aktualizovány, může nastat situace, kdy chceme získat entitu, která do systému nepatří. V systému řešícím umělou inteligenci nepřátel můžeme např. požadovat získání entity reprezentující hráče, abychom zjistili jeho pozici. K tomu byly navrženy třídy `TagManager` a `GroupManager`, které slouží k označení entity, resp. skupiny entit. Systém se sám stará o označené skupiny entit a entity. Pokud je např. jedna entita ze skupiny odstraněna ze světa, je nutné ji odstranit i ze skupiny.

Pokud by si uživatel sám udržoval nějaký kontejner ukazatelů na entity, musel by ho pracně udržovat validní. Díky tomu, že se o označené entity stará systém, lze se vždy spolehnout na správnost poskytnutých entit a na efektivní implementaci.

Třída `World` je třídou, která spojuje popsané třídy. Je to hlavní třída, se kterou bude uživatel interagovat. Třída obsahuje instanci třídy `EntityManager` (pro správu entit), `System Manager` (pro správu systémů entit), `GroupManager` a `TagManager`. Třída je znázorněna na diagramu 3.7.

Mezi členské metody patří metoda `beginStep` a `endStep`. Tyto metody jsou volány na začátku aktualizace herního světa a na konci aktualizace. Uvnitř metody `beginStep` je vyřešena příslušnost entit k systémům entit. Pokud byla nějaká entita pozměněna, je na začátku aktualizace znovu přiřazena do správných systémů entit. Po aktualizaci herního světa jsou odstraněny entity, které mají být smazány. Mazání entit v průběhu aktualizace by bylo velmi nepohodlné. Nejen, že by byly narušeny iterátory, museli bychom také řešit okamžité odstranění entity ze všech systémů entit. Dalším velikým problémem je násobné mazání. Pokud budeme chtít smazat entitu poté, co ji zasáhne projektil, jak bychom postupovali, pokud ji zasáhnou dva projektily ve stejném aktualizačním cyklu? Entity se tedy neodstraňují okamžitě, jsou označeny pro pozdější smazání (přidání do kontejneru, který nedovoluje ukládat duplicitní data, např. `std::set`) a jsou smazány až při volání `endStep`.

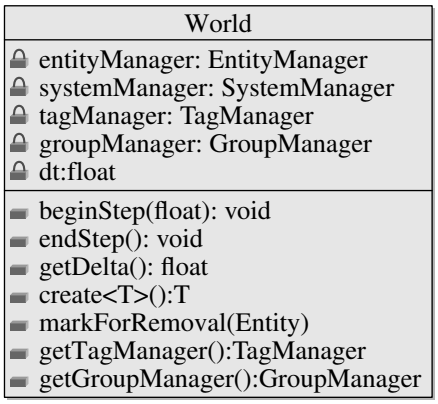
Stejně jako má třída zodpovědnost za mazání entit, umožňuje a zodpovídá i za jejich vytváření. Vytváření entit není triviální, entity jsou opakovaně používány. Není tedy možné nechat uživatele vytvářet entity stylem `Entity * e = new Entity`. Namísto toho uživatel žádá `World` o poskytnutí nové instance entity. Třída sama vnitřně vyřeší veškerou logiku a poskytne uživateli instanci. Tento přístup je známý jako návrhový vzor *Factory Method* [12]. Pokud bychom se rozhodli změnit vytváření instancí (např. zavést efektivnější alokátor), vyřešíme to pouze uvnitř třídy `World` a uživatelský kód zůstane zcela nezměněn. Stále bude vytvářet nové instance jako: `Entity * e = world.create<Entity>()`.

Třída `World` nevytváří pouze entity. Stejným principem vytváří i komponenty a systémy entit. Uživatel volá pouze metodu `create` s požadovaným datovým typem. Pokud chce vytvořit instanci systému entit `MyEntitySystem`, bude volat `world.create<MyEntitySystem>()`. Systém entit je automaticky přidán do správce systémů entit (`SystemManager`). Pokud bude chtít vytvořit zdravotní komponentu, která v konstruktoru přijímá číslo, které udává výchozí (a zároveň maximální) hodnotu zdraví, bude volat `world.create<HealthComponent>(100)`. Jak je metoda `create` implementována se můžete dočíst v následující kapitole.

Při aktualizaci objektů je nutné znát čas, který uplynul od minulého snímku (značen *dt*). Pomocí *dt* jsme schopni např. hýbat s objekty v závislosti na uplynulém čase. Pokud bychom hýbali objektem při každé aktualizaci o pevnou konstantu, objekt by se pohyboval rychleji na počítačích, které zvládnou spočítat více snímků za sekundu (vícekrát by tedy stihly objektem posunout o konstantu). Častým přístupem je předávání aktuální hodnoty *dt* do každé metody, která souvisí s aktualizací. To není ovšem příliš pohodlné, protože ne všechny aktualizace potřebují vědět o uplynulém čase (např. při aktualizaci vzájemné viditelnosti entit čas nehraje žádnou roli).

Z toho důvodu byla přesunuta správa *dt* do třídy `World`. Systémy entit, které budou potřebovat *dt* znát, jej mohou získat pomocí metody `getDelta`. Změnami reálného *dt* můžeme ve hře získat efekty jako zpomalení nebo zrychlení času. Uvedené řeší třída `World` a její metoda `setTimeFactor`. Pokud chceme docílit zpomalení hry na 50% původní rychlosti, stačí

skutečné *dt* vydělit dvěma. Tím veškeré aktualizace budou "ošáleny"a budou si "myslet", že ve hře uplynulo méně času, než tomu je ve skutečnosti. Obdobně je možné hru také zrychlovat. Je však důležité mít na paměti, že výrazné zpomalení či zrychlení hry může způsobit numerické problémy např. ve fyzikálním systému (více např. v [10]).



Obrázek 3.7: Třída World

3.2 Zapojení navrženého systému do hry

V této části bude popsáno, jak navržený systém zapojit do hry. Máme třídu `World`, která zapouzdřuje práci s entitami, komponentami a systémy entit, to ale nestačí.

Celá hra je reprezentována třídou `Game`. Třída má hlavní metodu `start`. Pro spuštění hry tedy vytvoříme instanci třídy `Game` a zavoláme `start`. Zpravidla je toto volání přímo uvnitř funkce `main`, která je vstupním bodem programu. V metodě `start` se děje více věcí. První z prováděných věcí je inicializace celé hry. Při inicializaci se provádí:

- **inicializace vstupů**: Je založen tzv. `InputProvider`. To je třída, která zapouzdřuje vlákno, které sbírá uživatelský vstup na dané platformě (např na platformě Windows pomocí *Raw Input*) a poskytuje ho do metod, které se zaregistrovaly pro odběr uživatelského vstupu. Vstup je potřeba např. pro pohyb s hlavním hrdinou, nebo pro používání vývojářské konzole (konzole bude popsána dále).

- **inicializace virtuálního systému souborů (*virtual filesystem*)**: Počítačové hry kromě spustitelného souboru obsahují spoustu *assetů*, např. 3d modelů, textur, zvukových souborů, popisů scény aj. Virtuální systém souborů umožňuje všechny tyto soubory (celou hierarchii souborů) umístit do jednoho šifrovaného souboru[1]. Při prohlížení struktury hry tedy hráč uvidí pouze jeden *nesmyslný* soubor, ale programátor může přes virtuální systém souborů se souborem pracovat jako s běžnou hierarchií souborů. Díky tomu nemůže hráč např. měnit herní data. Další výhodou je např. zavedení vlastního relativního adresování. Relativní adresa od spustitelného souboru může být např. `"../../data/myGameData"`. Virtuální filesystem umožňuje nastavit uvedenou relativní adresu jako výchozí a programátor ji již nikdy nemusí opakovat.

[1] Příkladem virtuálního systému souborů je např. PhysicsFS https://icculus.org/physfs/

- **inicializace systému zdrojů**: V počítačových hrách je velké množství dat načítáno ze souborů. Protože čtení souborů není tak rychlé (v porovnání s přímým čtením z paměti), není přípustné načítat data až v okamžiku, kdy jsou potřeba. Pokud bychom např. při každém výstřelu měli načítat podobu kulky ze souboru, tak by se hra při každém výstřelu na okamžik zasekla. Je tedy vhodné data načíst předem do paměti a poté se na ně pouze odkazovat. Dnešní počítačové hry však mívají i přes 10GB assetů (videa, animace, zvuky,...). Načtení všech dat tedy na většině dnešních strojích není možné. Tento problém řeší právě systém zdrojů. Předem načítá data, která se budou používat a naopak odstraňuje z paměti data, která se po určitou dobu používat nebudou.

- **inicializace vykreslovacího systému**: Je vytvořeno okno, založen grafický kontext, jsou zaregistrovány signály na akce vykonávané při změně velikosti okna apod.

- **inicializace grafického uživatelského rozhraní (GUI) a správce písem**: Pro herní nabídku a ladící nástroje je potřeba mít ve hře grafické uživatelské rozhraní.

Velmi užitečným ladícím nástrojem při vývoji her je konzole. Do konzole je možné psát příkazy pro načtení určité herní mapy, vytvoření herních objektů, upravení parametrů aj. Po spuštění hry jsou automaticky prováděny určité příkazy (načtení úvodní obrazovky apod.). Příkazy může uživatel zadat do souboru, které je při inicializaci načten a příkazy jsou provedeny.

Poté, co je hra inicializována a jsou vykonány všechny požadované konzolové příkazy dochází ke spuštění hry - zavolání herní smyčky. Smyčka běží do té doby, dokud není ukončena uživatelem (ukončením hry z herní nabídky, zavřením okna,...). V těle smyčky se provedou nezbytné aktualizace a aktualizace současné instance třídy `GameState`(popsána dále). Na konci smyčky je vykreslen současný stav hry na obrazovku. V každé iteraci se měří reálný čas, za kterou se vykonala a tento čas je poté použit jako *dt*.

Pokud spustíme počítačovou hru, bylo by nepříjemné, kdybychom se okamžitě ocitli v nějaké herní úrovni. Stejně tak by bylo nepříjemné, kdyby hra nešla pozastavit. Z toho plyne, že počítačová hra se nesestává pouze z módu, kde opravdu hrajeme. Správná hra má herní nabídku, ve které můžeme zvolit jakou část hry chceme hrát, nastavit ovládání aj. Do této nabídky také můžeme odskočit při přerušení hry. Z toho důvodu byla založena třída `GameStateBase`. To je třída, která reprezentuje určitý stav hry, který může být uložen ve třídě `Game` a aktualizován v herní smyčce. Je jedno, jestli instance `GameStateBase` reprezentuje herní level, nebo herní menu. Menu je také určitá hra. Je interaktivní, jen má pozměněnou logiku (namísto ovládání hrdiny ovládáme tlačítka).

Třída `Game` má tedy ukazatel na aktuální instance `GameStateBase`, kterou aktualizuje. Instance je samozřejmě možné měnit (po kliknutí na tlačítko *Nová hra* v menu dosadíme jinou instanci). Třída `GameStateBase` je čistě abstraktní třída, která má následující virtuální metody:

- `init()`: Provede potřebné inicializace, např. založení systémů entit aj.

- `deinit()`: Je volána při destrukci instance, provedou se v ní tedy potřebné operace jako uvolnění určitých dat aj.

- `step()`: Tato metoda bude volána v herní smyčce. Zde bude uživatel aktualizovat své systémy entit aj.
- `render()`: Tato metoda bude volána vždy na konci herní smyčky pro vykreslení stavu hry na obrazovku.
- `load()`: V této metodě se načtou a zpracují data ze souborů, která jsou potřebná pro danou část hry.
- `unload()`: Zde proběhne vyčištění současné instance `GameStateBase` od načtených dat.
- `activated()`: Tato metoda je zavolána, pokud je instance dosazena jako aktuální. Pokud např. přecházíme z herního menu do hry, je zavolána tato metoda a je možné v ní provést akce, jako např. zaregistrování konzolových příkazů, které mají fungovat pouze při hraní, nikoli v menu.
- `deactivated()`: Opačná metoda k metodě `activated`.
- `input()` V této metodě je možné zpracovávat uživatelský vstup.

Jaký je hlavní rozdíl mezi `init` a `load` resp. `deinit` a `unload`? Metoda `init` je volána při založení instance. Má na starosti spuštění a inicializace požadovaných systémů, avšak nezodpovídá za jejich naplnění daty, to má na starosti metoda `load`. Mějme situaci, kdy hrajeme nějakou herní úroveň a potřebujeme přepnout na úroveň jinou (například při dohrání úrovně, či konzolovým příkazem). Herní úroveň jsme nastartovali metodou `init` a naplnili daty metodou `load`. Při přechodu na další úroveň není nutné znovu startovat instanci `GameStateBase`. Jen ji naplníme novými daty. Načteme tedy novou geometrii herní úrovně, založíme nepřátelé na správných pozicích aj. Logika hry a tedy systémy entit apod. zůstává stejná. Pokud budeme hrát např. sekvenci dvou herních úrovní, volání budou následující: `init - load - unload - load - unload - deinit`.

Třída, která implementuje rozhraní `GameStateBase`, je třída `GameState`. V této třídě je vytvořena instance třídy `World`, spravující entity, komponenty a systémy entit. Třída obsahuje dvě hlavní metody, metodu `getWorld` a metodu `step(dt)`. Pomocí metody `getWorld` může uživatel získat instanci třídy `World` a následně např. zakládat entity. Metoda `step(dt)` je volána z herní smyčky a nejprve zavolá `World::beginStep`, poté zavolá metodu `step`, což je virtuální metoda, ve které bude moci uživatel provádět aktualizace a nakonec metodu `World::endStep` (metody `World::beginStep` a `World::endStep` byly popsány při představování třídy `World`).

Shrnutí

V této kapitole byla představena reprezentace herního objektu, jeho komponent a systémů entit. Také bylo ukázáno, jak vše bude provázáno a jak bude celý vytvořený systém pro správu herních objektů zapojen do hry. V kapitole 4 budou popsány zajímavé konstrukty použité při implementaci. Celá implementace je přiložena jako příloha k této práci. Představený framework je zcela obecný, není vázán na 2D ani 3D hry. V kapitole 5 bude ukázáno, jak framework specializovat pro využití pro 2D hry a bude diskutováno jeho použití pro 3D hry. Budou již představeny konkrétní systémy využitelné pro všechny 2D hry, herně nezávislé komponenty aj.

Kapitola 4

Implementace

V této kapitole budou popsána zajímavá místa v implementaci vzniklého frameworku. Celá implementace je k dispozici jako příloha k této práci a byla provedena v jazyku C++.

4.1 Implementace Bitset

Nejprve bude detailně představena implementace `Bitset`, která slouží k efektivnímu provázání entit, komponent a systémů entit. Pro práci s komponentami v konstantním čase je nutné stanovit velikost sekvence bitů. Velikost musí být větší než počet komponent či systémů entit ve hře. K reprezentaci sekvence bitů lze využít základní datové typy (`short`, `int`, ...), ale ty dostačují pouze do určité velikosti. Po překročení této velikosti je potřeba použít jinou reprezentaci, např. `std::bitset`[1]. Na ukázce kódu 4.1 je ukázáno, jak lze `Bitset` a operace nad ním implementovat tak, aby podporoval jak základní datové typy, tak `std::bitset`.

```
1  typedef uint64 Bitset;
2  //also possible for example:
3  //typedef int32 Bitset;
4  //typedef std::bitset<128> Bitset;
5
6  template <size_t size> struct GetLength<std::bitset<size>>
7  {
8      enum { Value = size };
9  };
10
11 template <typename T> struct GetLength
12 {
13     enum { Value = sizeof(T) * 8 };
14 };
15
16 template <typename T> bool to_bool(const T & t)
17 {
18     return t != Bitset(0);
19 }
20
21 template <size_t size> bool to_bool(const std::bitset<size> & b)
22 {
23     return b.any();
24 }
25
```

[1] http://en.cppreference.com/w/cpp/utility/bitset

```
26 template <size_t size> bool hasSetBit(const std::bitset<size> & b,
27                                        unsigned bit)
28 {
29     return b.test(bit);
30 }
31
32 bool hasSetBit(const Bitset & b, unsigned bit)
33 {
34     return to_bool(b & Bitset(Bitset(1) << bit));
35 }
```

Kód 4.1: Reprezentace a operace nad sekvencí bitů

4.2 Implementace ComponentType a získávání komponent v konstantním čase

Nyní bude popsána třída `ComponentType`. Díky této třídě má každá komponenta unikátní identifikátor a sekvenci bitů získanou v konstantním čase. Takto získané identifikátory slouží k rychlému (opět v konstantním čase) získávání komponent z tabulky ve třídě `EntityManager`. Třída je vyobrazena na ukázce 4.2. Při prvním volání `getId` se do statické proměnné uloží hodnota z atomického čítače. Tím má každá komponenta jednoznačně přiřazený identifikátor. Tento identifikátor je použit k sestavení sekvence bitů v metodě `getBit`. Identifikátor i sekvence bitů jsou po prvním zavolání metod uloženy ve statických proměnných a při každém dalším zavolání jsou okamžitě vráceny.

```
1 atomic_counter s_nextComponentTypeId = 0;
2
3 template <typename CT>
4 class ComponentType
5 {
6 public:
7     static unsigned getId()
8     {
9         static unsigned compId = ComponentType<CT>::next();
10        return compId;
11    }
12
13    static const Bitset & getBit()
14    {
15        static Bitset b = Bitset(1) << (ComponentType<CT>::getId());
16        return b;
17    }
18
19 private:
20    static unsigned next()
21    {
22        return s_nextComponentTypeId++;
23    }
24 };
```

Kód 4.2: Implementace `ComponentType`

Kód 4.3 ukazuje, jak lze využít `ComponentType` pro získání komponenty v konstantním čase. Uvnitř třídy `EntityManager` je uloženo pole polí `m_componentMap`, ve kterém jsou uloženy komponenty. Pole poskytuje `operátor[ ]`, který umožňuje adresovat jeho prvky v konstantním čase. Pomocí dvou identifikátorů můžeme získat požadovanou komponentu.

Protože, že jsou komponenty uloženy jako ukazatele na třídy `Component`, využijeme typu `T` a komponentu přetypujeme na správný typ. Tím opět ušetříme uživateli práci.

```
1 template <typename T>
2 T * EntityManager::getComponent(Entity * from)
3 {
4     typedef ComponentType<T> CT;
5
6     return static_cast<T*>(m_componentMap[CT::getId()][from->getId()]);
7 }
```

Kód 4.3: Získání komponenty

Použití je zobrazeno na ukázce 4.4. Komponentu tedy v konstantním čase získáme a poté na ni přímo můžeme volat metody. Abychom uživatele ušetřili ještě dalšího psaní, vznikla metoda `call`, která používání komponent ještě zjednodušuje (ukázka 4.5). Implementace byla prováděna ve Visual Studio 2010, které plně nepodporuje standard C++11, nebylo možné využít variadické šablony. Z toho důvodu musela být metoda `call` přetížena pro různé počty argumentů.

```
1 Entity * e = //...
2
3 HealthComponent & hc = e->get<HealthComponent>();
4
5 if(hc.getHealth() < 60)
6 {
7     AnimationComponent & ac = e-><AnimationComponent>();
8     ac.setModification(WOUNDED);
9 }
```

Kód 4.4: Použití kompnent

```
1 template <typename T, typename RT>
2 RT call(RT (T::*func)())
3 {
4     T & component = get<T>();
5     return (component.*func)();
6 }
7
8 template <typename T, typename RT, typename AT1>
9 RT call(RT (T::*func)(AT1), AT1 at1)
10 {
11     T & component = get<T>();
12     return (component.*func)(at1);
13 }
14 //...
15
16 //pouziti
17 e->call(&AnimationComponent::setModification, WOUNDED);
```

Kód 4.5: Pomocná metoda `call`

4.3 Implementace kontroly vztahu mezi třídami

Při zakládání entit, komponent a systémů entit se volá metoda `World::create`. Metoda založí novou instanci a provede další nutné kroky pro správné použití. Aby uživatel nemohl

požádat o založení jiného datového typu, než který `World` podporuje, bylo nutné kontrolovat, zda typ poskytnutý uživatelem odpovídá, nebo dědí ze třídy `Entity`, `EntitySystem` nebo `Component`. Kód, který toto dovede zkontrolovat, byl vytvořen pomocí pokročilého šablonového metaprogramování a můžete si ho prohlédnout na ukázce 4.6[2]. Pro zjištění, zda např. typ `T` je odvozen od typu `Component`, zavoláme `is_base_of<Component, T>::value`.

```
1  template <typename B, typename D>
2  struct is_base_of
3  {
4      typedef char (&yes)[1];
5      typedef char (&no)[2];
6
7      struct Host
8      {
9          operator B*() const;
10         operator D*();
11     };
12
13     template <typename T>
14     static yes check(D*, T);
15     static no check(B*, int);
16
17     static const bool value = sizeof(check(Host(), int())) == sizeof(yes);
18 };
```

Kód 4.6: Kód pro ověření vztahu mezi třídami

[2] Jak kód funguje je popsáno např. na http://stackoverflow.com/questions/2910979/how-is-base-of-works

Kapitola 5

Použití vytvořeného frameworku a jeho specializace

5.1 Framework

Vytvořený framework je možné použít pro 2D i 3D hry. Tyto hry mají něco odlišného a něco společného. Např. fyzikální systém či skeletální animace jsou odlišné ve 2D a ve 3D. Na druhou stranu reprezentace zdraví je společná oběma typům her. V této kapitole budou nejprve popsány komponenty a systémy entit, které jsou nezávislé na dimenzionalitě. Poté bude představno rozšíření frameworku na framework2D a ukázány komponenty a systémy entit typické pro 2D hry.

Jedním z prvků nezávislých na počtu dimenzí jsou určité části umělé inteligence. Počítačem řízení protivníci se samozřejmě budou chovat odlišně ve 2D a 3D světě, ale rozhodování bude fungovat totožně. Pro rozhodování byly naimplementovány behaviorální stromy.

Behaviorální stromy jsou v dnešních hrách hojně používané[1]. Jedná se o uspořádání akcí do stromové struktury, která mimo akcí obsahuje ještě uzly označené jako *sekvence* a *selekce*. Každé chování je rozděleno na atomické akce. Například chování *Zabít hráče* bude složené z akcí *Seber zbraň, pokud nemáš*, *Dostaň se k hráči na dostřel*, *Střílej na něho*. Např. chování *Dostaň se k hráči na dostřel* bude opět dále rozděleno na *Zjisti pozici hráče*, *Najdi cestu k hráči*, *Kontroluj, zda již není hráč na dostřel*. Takto můžeme dělit akce dokud nedostaneme atomické akce. Spojení akcí do jednoho chování se provede pomocí uzlu *sekvence*.

Pokud při traverzování stromu narazíme na uzel *sekvence*, víme, že pro provedení akce je potřeba vykonat všechny jeho potomky. Každý uzel ve stromu má vstupní podmínky. Pokud jsou vstupní podmínky splněny, můžeme do uzlu vstoupit. Pokud nejsou splněny, musíme pokračovat s jinými uzly. Traverzace uzlu akce vrátí stav, s jakým vykonávání akce skončilo. Pokud vše dopadlo dobře, uzel vrátí hodnotu *SUCCES* a pokračuje se s dalším uzlem v pořadí. Uzel *sekvence* tedy předpokládá, že všechny jeho potomci navrátí tuto hodnotu. Potomci mohou být opět *sekvence*. Může se stát, že akce nedopadne dobře, např. není nalezena cesta, kterou by se měl agent vydat. V tom případě uzel vrátí hodnotu *FAILURE* a musíme traverzovat do jiného chování.

[1] http://aigamedev.com/insider/presentations/behavior-trees/

Opakem *sekvence* je uzel *selekce*. Uzel *selekce* vyžaduje splnění pouze jednoho potomka. Představme si modelový příklad. Máme agenta, který hlídkuje u brány a pokud vidí hráče, zaútočí na něho. Máme tedy dvě chování *Attack* a *Patrol*. Obě chování nelze vykonávat zároveň. Budou tedy potomky uzlu *selekce*. Pokud tedy jedno chování skončí neúspěšně, vykonáme chování druhé. Prvním chováním bude *Attack*, které bude mít jako vstupní podmínku test, zda je možné, aby agent viděl hráče. Pokud agent hráče nevidí, celé chování skončí neúspěšně a pokračuje se dalším chováním - *Patrol*. Pokud agent uvidí hráče, zaútočí na něho, v opačném případě bude dále hlídkovat. Hlídkování a útok budou reprezentovány uzlem *sekvence*. Pro hlídkování agent nejprve musí najít cestu, poté ji se po ní začne pohybovat, při tom se bude rozhlížet atp.

Toto bylo stručné seznámení s behaviorálními stromy. V praxi mohou uzly vracet ještě hodnotu *RUNNING*, která udává, že akce je dlouhodobějšího rázu a zásadně upravuje traverzaci. Také existují další uzly, jako např. *dekorátor*, které upravují chování v podstromech. Behaviorální stromy však nejsou tématem této práce, proto zde nebudou podrobněji popsány. Více informací o behaviorálních stromech je možné zjistit např v [22].

Je zřejmé, že samotné behaviorální stromy jsou nezávislé na počtu dimenzí hry. Byly proto implementovány a zabaleny do komponenty `AIComponent`. Pokud nyní libovolné herní entitě přiřadíme `AIComponent`, můžeme ji přiřadit behaviorální strom, podle kterého by se měla chovat. Byl také vytvořen systém entit `AIEntitySystem`, který aktualizuje všechny entity, které obsahují `AIComponet`. Systém entit z komponenty získá strom a traverzuje jej.

Další ukázkou komponenty, která byla vytvořena nezávisle na dimenzionalitě je `NavigationComponent`. V této komponentě je uložen např navigační graf scény a poskytuje informace o navigačních cestách. Pokud v jedné akci nalezneme cestu, kterou chceme, aby agent traverzoval, zapíšeme ji právě do navigační komponenty. V další akci, kterou může být např vyhlazení cesty nebo její traverzace, se opět na patřičná data dotážeme navigační komponenty. Ještě zmíníme `ActiveComponent`, která udržuje informaci, zda je daná entita aktivní, neaktivní, zapnutá, vypnutá, v činnosti apod. (aktivní komponenta je tvořena jen jednou booleovskou hodnotou, ke které je umožněn přístup přes více metod s odlišnými názvy).

5.2 Framework2D

Vzhledem k tomu, že málo která hra se obejde bez fyzikální interakce mezi objekty, je nutné, zapojit do frameworku fyzikální systém. Zde se již zásadně liší, zda cílíme na 3D, nebo 2D hry. Pro 3D hry by bylo možné použít např *Bullet Physics Library*[2]. Protože v této práci jde především o komponentový návrh objektů, nikoli o výslednou hru, byla pro otestování funkčnosti celého systému zvolena jednodušší 2D hra. Je tedy nutné specializovat framework právě pro 2D herní tituly. Začneme tím, že do frameworku integrujeme fyzikální systém. Jako fyzikální systém byl zvolen *Box2D*[3].

5.2.1 Fyzikální systém

Integrace byla provedena odvozením nové třídy `World2d` od třídy `World`. Třída `World2d` v sobě skrývá fyzikální systém a stará se např. o správné mazání entit. Pokud odstraníme

[2] http://bulletphysics.org
[3] http://box2d.org/

ze hry entitu, musíme odstranit i její případnou fyzikální reprezentaci z fyzikálního systému. Právě o to se stará metoda `World2d::remove`, odstraní fyzikální reprezentaci a poté zavolá původní `World::remove`.

Pokud chceme entitě přiřadit fyzikální reprezentaci, uděláme to pomocí komponenty `PhysicsComponent`. Třída `PhysicsCompnent` zapouzdřuje fyzikální objekt (v našem případě `b2body`) a poskytuje fyzikální operace nad ním. Umožňuje volání např. metod `applyForce`, `applyTorque`, `setDensity` aj. Entita, která obsahuje `PhysicsComponent`, je zařazena do fyzikální simulace a můžeme na ní působit určitými sílami apod. O spuštění a synchronizaci s fyzikální simulací se stará `PhysicsUpdatingSystem`. Aktualizuje podmnožinu entit, které obsahují `PhysicsComponent` a `SpatialComponent`(komponenta, která zapouzdřuje transformaci objektu). Pří každé aktualizaci spustí krok fyzikální simulace a poté nové transformace objektů uloží do instance `SpatialComponent`. Je třeba dávat pozor, jak často a s jakým *dt* fyzikální systém aktualizujeme, abychom nezpůsobili numerické problémy. Z toho důvodu byla do fyzikálního systému entit zavedena aktualizace pomocí akumulované časové diference (více v [10]).

Od fyzikálního systému nepožadujeme pouze fyzikální simulaci. Rádi bychom také dostávali informace, které objekty spolu kolidují a patřičně na tyto kolize reagovali. Pokud např. nastane kolize mezi projektilem a nepřítelem, chceme projektil odstranit z herního světa a nepříteli ubrat zdraví. *Box2d* umožňuje získání těchto informací, ale v omezené míře. Poskytuje např. metody `BeginContact` a `EndContact`, které jsou typu *callback* a jsou volány pokud libovolné dva objekty začaly kolidovat, nebo právě přestaly kolidovat. Tímto se ale dozvíme pouze informaci, která se týká libovolných dvou objektů. Musíme tedy pracně zjišťovat, kterých objektů se zpráva týká.

Byl proto nad výchozím systémem box2d postaven robustnější systém na zpracovávání *callbacků* o kolizích. Implementace nového *callbacku* je velmi jednoduchá. Vznikla šablonová třída `CollisionCB`, kterou uživatel vydědí pro požadovanou informaci o kolizi. Pokud by měl např. v herním světě objekty reprezentující zem (označené jako `GROUND`) a projektily (označené jako `PROJECTILE`) a chtěl by se dozvědět o každé kolizi mezi nimi, dosáhl by toho kódem na ukázce 5.1. Uvnitř metody `onBeginContact` by mohl např. smazat kulku a vytvořit částicový efekt odlétající země. Metoda bude volána pouze pro pevně stanovenou dvojici, což je uživatelsky velmi příjemné.

```
1 struct ProjectileGroundCB : public CollisionCB<PROJECTILE, GROUND>
2 {
3     void onBeginContact(Entity & projectile, Entity & ground)
4     {
5         //...
6     }
7 };
```

Kód 5.1: Použití nadstavby nad informací o kolizi

Z původní třídy `GameState` byla odvozena její specializace `GameState2d`. Uvnitř aktualizace instance `GameState2d` je aktualizován `PhysicsUpdatingSystem`. Pokud tedy budeme vytvářet novou 2d hru, využijeme připraveného `GameState2d`, který nám zcela automaticky poskytne podporu pro simulaci fyzikálních objektů a registrování *callbacků* o kolizích. Jediné, co programátor musí udělat, je přiřadit herní entitě fyzikální komponentu s daty o její fyzikální reprezentaci. Pokud má ve hře např. míč, bude ho reprezentovat pomocí kruhu, kterému

nastaví poloměr, hustotu, odrazivost, tření apod. O zbytek se již postará sám framework2d a bude uživateli entitu korektně simulovat.

5.2.2 Prostorové dotazy nad scénou a další funkcionalita

Dalším důležitým systémem entit, který je poskytován v rámci framework2d, je `SpatialSystem`. Tato třída aktualizuje podmnožinu entit, které obsahují `SpatialComponent`. Uvnitř této třídy je možné udržovat prostorovou datovou strukturu, která nám urychlí dotazy o pozicích. Prozatím má `SpatialSystem` implementovány dva druhy dotazů, `rectangularQuery` a `circularQuery`. Jak již názvy napovídají, první dotaz slouží k získání všech entit v dané obdélníkové oblasti a druhý dotaz pro získání všech entit v kruhové oblasti. Dotaz bychom mohli využít např. pro výbuch granátu. Po výbuchu granátu bychom provedli dotaz na kruhovou oblast a tím získali všechny entity, kterým ubereme zdraví.

Příklad s výbuchem má jeden nedostatek. Pokud provedeme dotaz na entity v daném regionu, můžeme dostat i entity, které žádné zdraví nemají (země, ležící zbraň, kulka, ...). Mohli bychom tedy specifikovat, že máme zájem pouze o entity, které obsahují `HealthComponent`. Výbuch ale může způsobit to, že ve svém okolí aplikuje radiální sílu na všechny fyzikální objekt (ležící zbraň bude výbuchem posunuta). Ne všechny objekty, které obsahují zdravotní komponentu, musí obsahovat fyzikální komponentu a naopak. Problém by šlo vyřešit aplikováním dvou dotazů. Nejprve se dotázat na entity v okolí výbuchu, které mají zdravotní komponentu a poté provést nový dotaz na entity, které mají fyzikální komponentu. Vyhledávání však může být velmi náročnou operací, a proto dvojité dotazy nejsou ideální. Z toho důvodu byla založena třída `SpatialQueryResult`, která je popsána v následujícím odstavci.

`SpatialQueryResult` je třída reprezentující výsledek dotazu a umožňující manipulace s ním. Pokud zavoláme `circularQuery`, jsou všechny entity v dané oblasti uloženy do instance `SpatialQueryResult`. Poté je možné je velmi efektivně filtrovat (pouhým porovnáváním `Bitset`). `SpatialQueryResult` poskytuje např. metodu `processingFilter`, která po zavolání provádí požadovanou akci pouze s entitami, které obsahují požadovanou komponentu. Ukázku můžete vidět v kódu 5.2. Provedeme dotaz na kruhovou oblast okolo bodu `center` o poloměru `radius` a výsledek je uložen do `res`. Poté je možné volat vícekrát `processingFilter`, pokaždé s jiným požadavkem na komponenty. Metoda `processingFilter` jako parametr přijímá funkci, je tedy možné použít *lambda funkce* z C++11 a kód je poté velmi kompaktní.

```
1  SpatialQueryResult res;
2  spatialSystem->circularQuery(center, radius, res);
3
4  res.processingFilter<HealthComponent>([&](Entity *e){
5      //...
6  });
7
8  res.processingFilter<PhysicsComponent>([&](Entity *e){
9      //...
10 });
```

Kód 5.2: Použití `SpatialQueryResult`

`SpatialSystem` je také obsažen přímo v `GameState2d`, tedy uživatel pouhým přiřazením `SpatialComponent` libovolnému objektu získává možnost začlenit ho do scény a získávat ob-

jekt při prostorových dotazech. Mimo správy fyzikální simulace a transformací framework2d poskytuje řadu dalších komponent a systémů, které může uživatel použít. Příkladem je `RenderComponent`, neboli vykreslovací komponenta. Komponenta dává uživateli možnost nadefinovat, jak bude daná entita vykreslena. Všechny entity, které obsahují vykreslovací komponentu jsou poté vykreslovány pomocí systému entit `WorldRenderSystem`.

Abychom mohli herní svět vykreslit, je nutné vytvořit kameru. Proto byla vytvořena komponent `ProjectionComponent`. Ta umožňuje práci s výškou a šířkou obrazu, poměrem stran, přibližováním obrazu apod. Samotná projekční komponenta není pro vytvoření kamery dostatečná. Kameře je nutné přiřadit ještě `SpatialComponent`, která bude reprezentovat transformaci kamery, tedy její pozici, rotaci a měřítko. Pro určitý typ her by stačilo, aby kamera byla reprezentována pouze dvěma zmíněnými komponentami. Pokud ale ve hře může být více kamer současně, je potřeba kameře přiřadit `ActiveComponent` (popsána výše), která bude udržovat informaci o tom, která kamera je zrovna aktivní.

Framework2d také poskytuje možnost používat skeletální animace. Poskytuje komponentu `SkeletonComponent`, ta udržuje informace o skeletu a `SkeletonAnimationComponent`, která má na starosti správu animačních sekvencí nad skeletem. Pro obsluhu skeletálních animací jsou samozřejmě k dispozici systémy entit, které komponenty aktualizují. Systém okolo skeletálních animací je poměrně složitý, nebude zde tedy detailněji popsán, ale lze si ho prohlédnout v přiložených zdrojových kódech.

5.2.3 Lokální zpomalení času

Efekt zpomaleného času ve hrách je velmi populární. Lze ho dosáhnout modifikací dt, to má ale za následek zpomalení času napříč celým herním světem. Všechny pohybující se entity se budou pohybovat pomaleji. My si zde představíme, jak lze dosáhnout efektu lokálního zpomalení času. Lokální zpomalení času znamená, že čas bude zpomalen pouze v určité oblasti herního světa. Entity mimo oblast se budou pohybovat normálně, kdežto entity uvnitř oblasti budou zpomaleny. Lokální zpomalení času je samozřejmě nereálné, musel být tedy vymyšlen přístup, jakým efektu lokálního zpomalení bude dosaženo. Fyzikální svět aktualizuje vždy všechny objekty naráz, řešením tedy není posílání různým objektům různé dt.

Fyzikální simulace ve hře nesimuluje odpor vzduchu, proto se letící objekty pohybují po parabolické trajektorii. S použitím vzorců pro výpočet trajektorie lze odvodit, aby se dva objekty pohybovaly po stejné trajektorii, ale každý jinak rychle, musí platit: $g_1 = g_2 \cdot f^2$, kde g_1 je gravitační zrychlení prvního objektu, g_2 je gravitační zrychlení druhého objektu a f je faktor, kterým se mají lišit rychlosti objektů. Musíme tedy počítat s různým gravitačním zrychlením u objektů. Kdybychom objektu pouze zmenšili rychlost, objekt by spadl dříve. Pokud ještě objektu snížíme gravitační zrychlení, tak můžeme dosáhnout efektu zpomalení. Jak to reprezentovat pomocí frameworku je popsáno v dalších odstavcích.

Při vývoji frameworku ještě fyzikální systém *box2d* neumožňoval nastavování různým objektům různé gravitační zrychlení, proto musela tato funkcionalita být implementována na straně uživatele. Byla vytvořena komponenta `GravityComponent`, která uchovávala aktuální gravitační zrychlení entity. Spolu s ní vznikl systém entit `GravitySystem`, který při každé aktualizaci aplikoval entitám sílu dle vektoru gravitačního zrychlení, kterou měly v gravitační komponentě. Přiřazením gravitační komponenty entitě jsme schopni na každou entitu působit jiným gravitačním zrychlením. Některé entity tak mohou být simulovány normálně, kdežto

některé mohu např. *padat nahoru*. Gravitační komponenta byla využita i pro projektily, kterým bylo nastaveno nulové gravitační zrychlení, a tak projektily mohou létat po přímce.

Další komponentou, která musela pro daný efekt vzniknout, byla `SlowedComponent`. Komponenta udržuje informace o aktuálním zpomalení entity. Zpomalení entity se netýká jen pohybu. Představme si, že entitou je nepřítel se zbraní. Zbraň má parametry, které mimo jiné udávají, kolik kulek za vteřinu zbraň zvládne vystřelit. I tento parametr musí být tedy ovlivněn zpomalením, protože zpomalený nepřítel, jehož zbraň chrlí kulky běžnou kadencí, vypadá velmi nevěrohodně. V kapitole 3 bylo popsáno, že kdykoli je potřeba hodnota *dt*, je získána voláním `World::getDelta`. Třída `World2d` metodu `getDelta` překrývá metodou `getDelta(Entity)`. Metoda vrátí hodnotu *dt* pro danou entitu. Uvnitř metody se zjistí, zda entita obsahuje komponentu `SlowedComponent`, případně jak moc je zpomalená a je vrácena modifikovaná hodnota *dt*. Tato hodnota je využita všude, kde je chování závislé na čase. Agent ve zpomalovací zóně pomaleji míří, granátům trvá delší dobu, než vybuchne apod.

Zpomalovací zóny jsou reprezentovány třemi komponentami. První komponentou je známá `SpatialComponent` pro udání transformace. Druhou komponentou je `PhysicsComponent`, která popisuje kolizní těleso, tedy jaký má zóna tvar. Také fyzikální těleso je simulováno pouze jako *sensor*, ostatní objekty s ním tedy fyzicky nekolidují, ale uživatel přesto dostává informace o kolizi (stejným způsobem se reprezentují *trigger* zóny). Třetí komponenta je komponenta `TimeModificationComponent`. Ta entitě přidává informaci o tom, kolikrát se v zóně ostatní entity zpomalí.

Komponenta `SlowedComponent` o entitě říká: *Jsem entita, která se zpomalí, pokud narazí na zónu s lokálně zpomaleným časem.* Komponenta `TimeModificationComponent` na druhou stranu říká: *Pokud se mnou koliduje entita obsahující `SlowedComponent`, ja ji zpomalím.* Pokud se do zpomalené zóny dostane entita, která neobsahuje `SlowedComponent`, zpomalení času se ji nebude týkat. Pouhým přiřazením komponenty můžeme rozhodnout, které entity se mohou zpomalovat a které nikoli. Zbývá již jen implementovat *callback*, který pozmění vlastnosti entity, když se dostane do zóny, resp. když zónu opouští.

Hodnoty při kolizi upravuje `TimeCollisionCallback`. Při vletu entity do zóny upraví gravitační zrychlení, vektor rychlosti a úhlovou rychlost pomocí výše uvedené rovnice. Hodnoty modifikuje pomocí faktoru zpomalení, která má zóna uložen v komponentě `TimeModificationComponent`. Zóny lze do sebe zanořovat a tím vytvořit efekt plynulého přechodu do zóny se zpomaleným časem. Efekt lokálního zpomalení času je poskytován z framework2d a je možné ho použít v libovolné 2d hře. Hra která byla vytvořena spolu s touto prací efekt využívá a je možné si ho prohlédnout.

Kapitola 6

Implementace ukázkové hry a testování

Tato kapitola se věnuje tomu, jak využít Framework2d k vytvoření hry. Bude popsáno, jaké komponenty a systémy entit byly použity, jak naprogramovat jednoduchá chování agentů za pomoci behaviorálních stromů a další kroky, které jsou nutné pro vytvoření funkční hry. Také bude popsáno, jak vytvořit a používat prostředí, ve kterém se dají vytvořené komponenty testovat.

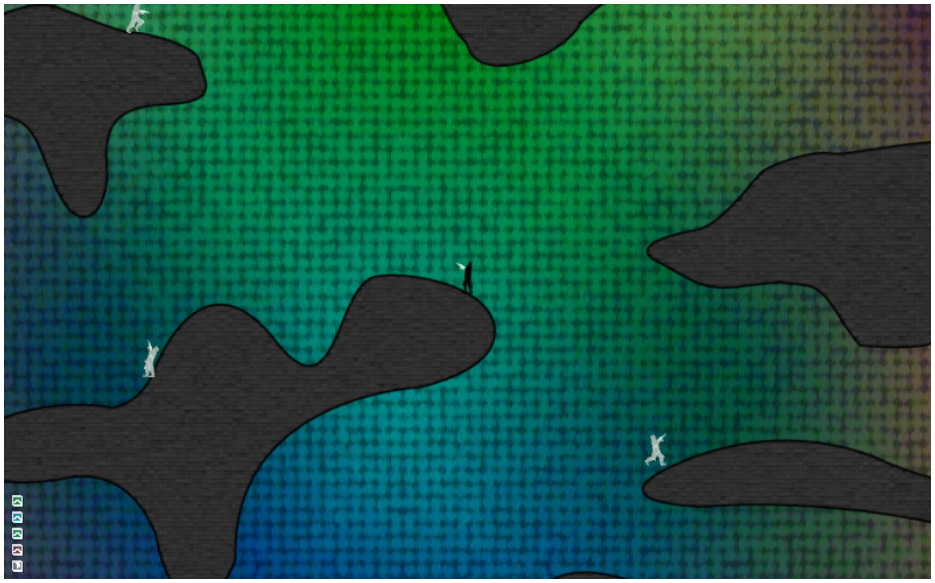
6.1 Koncept hry

Byla zvolena 2D hra pro jednoho háče s pohledem ze strany. Hra je akčního typu, kde se hlavní hrdina pohybuje v rámci jedné arény. Hráč ovládá jednu postavičku, pohybuje se po uzavřené mapě a snaží se přežít co nejdéle. Nepřátelé budou vlastnostmi (např. výdrži) převyšováni hlavním hrdinou, avšak nepřátel může být větší množství. Hra bude využívat efektu lokálního zpomalení času. Na obrázku 6.1 můžete vidět základní podobu vytvořené hry.

6.2 Specializace frameworku

Stejně jako byl rozšířen framework o prvky 2d her, bude nyní framework2d rozšířen tak, aby plně podporoval všechny požadované herní mechanizmy. Hra dostala název *Rifle*, a tak se specializace třídy `GameState2d` bude nazývat `RifleGameState`. Také byla implementována třída `RifleGame`, která je odvozena od třídy `Game`(popsané v kapitole 3) a řeší inicializace, které hra vyžaduje. Také se ve třídě zpracovává část uživatelského vstupu, která se týká hry jako celku, nikoli nějaké její části. Uživatelský vstup v této části umožňuje např. pozastavení všech aktualizací a krokování po jednom snímku či zadávání příkazů do konzole. Další uživatelský vstup, jakým je např. ovládání hlavního hrdiny, není řešen v této třídě, protože aktivní instancí `GameState` může být herní nabídka.

`RifleGameState` je třída odvozena od třídy `GameState` a je v ní implementována logika hraní jedné herní úrovně. Třída je poměrně rozsáhlá, mimo jiné shlukuje všechny systémy



Obrázek 6.1: Vytvořená hra

entit pro hru. Bude tedy představována průběžně. Mimo shlukování systémů entit má na starosti registraci požadovaných zpětných volání z fyzikálního systému a spouštění testovacích scénářů.

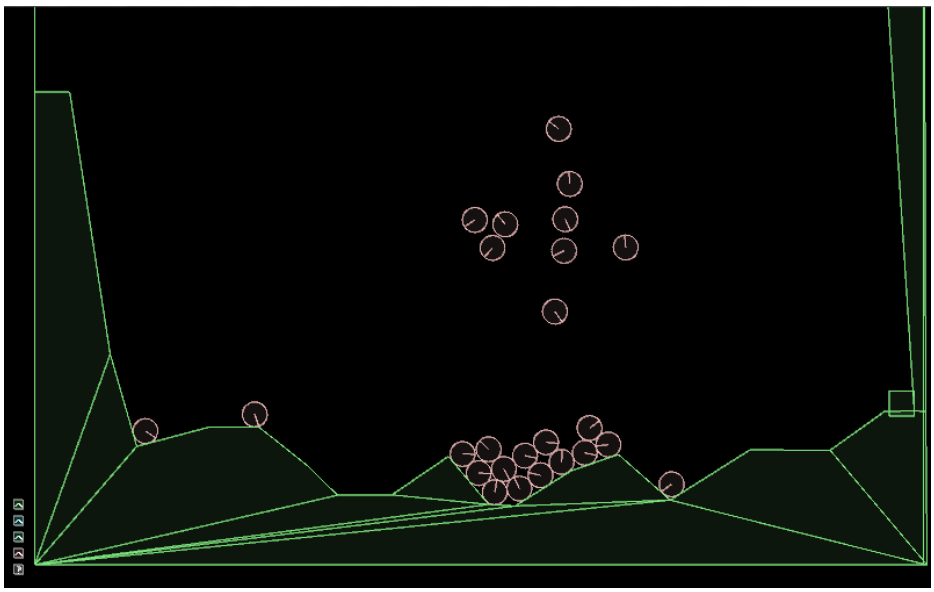
6.3 Testování základních komponent a systémů entit

Testovat vytvořené komponenty a systémů entit přímo hraním hry může být někdy velmi obtížné. Do situace, kterou chceme otestovat, se můžeme dostat až po několika minutách hraní, či se do ni nemusíme dostat nikdy (pokud je např. chování agentů ovlivněno generováním pseudonáhodných čísel). Vznikla tedy potřeba umožnit v herních úrovních spouštět testovací scénáře. Scénář umožňuje upravit výchozí stav scény a změnit průběh provádění herní logiky. Nové scénáře se vytváření odvozením nové třídy od třídy `Script`.

Při odvození nové třídy může uživatel překrýt tři metody - `init`, `step` a `draw`. V metodě `init` je možné nastavit výchozí stav ve scéně, např. zakládat nové objekty. Metoda `step` umožňuje upravit herní logiku v každé iteraci herní smyčky a metoda `draw` dává uživateli možnost vykreslit dodatečná data na obrazovku. Scénář je aktualizován spolu s herním světem. Načtená herní úroveň je aktualizována jako v průběhu hry, scénář jen upravuje její logiku. V následujících odstavcích si ukážeme příklady testovacích scénářů.

V rámci hry byla vytvořena třída `RifleEntityFactory`, která je zodpovědná za vytváření entit herního světa. Abychom vyzkoušeli, zda vytváření entit funguje správně, vznikl první testovací scénář. Scénář testuje vytváření kruhů a jejich zapojení do fyzikální simulace. Kruh je tvořen z `SpatialComponent` a `PhysicsComponent`. O aktualizaci se starjí systémy entit `SpatialSystem` a `PhysicsUpdatingSystem`, které jsou součástí framework2d. Pro simulaci kruhů tedy není nutné více programovat. Správnost chování byla ověřena vytvořením scénáře `Script_FallingBalls`. Při inicializaci scénáře je vytvořeno 25 kruhů, které se nechají volně

padat k zemi. Pokud se pozice kruhu a jeho přímých sousedů ustálí (fyzikální těleso přestává být simulované), je kruh opětovně přemístěn nad zem, ve scéně tak kruhy padají stále znovu (chování bylo implementováno v metodě `step`). Vytvořeným testem byla otestována funkčnost komponent a systémů entit týkajících se fyzikální simulace, vykreslování, transformací objektů aj. Vizuélně lze totiž poznat, zda se objekty na obrazovce chovají tak, jak bylo zamýšleno, jestli se neprotínají, jestli se správně odráží apod. Pokud ano, tak je zřejmé, že fungují komponenty a systémy entit týkající se fyzikálního, pozičního i vykreslovacího chování. Náhled na test je zobrazen na obrázku 6.2

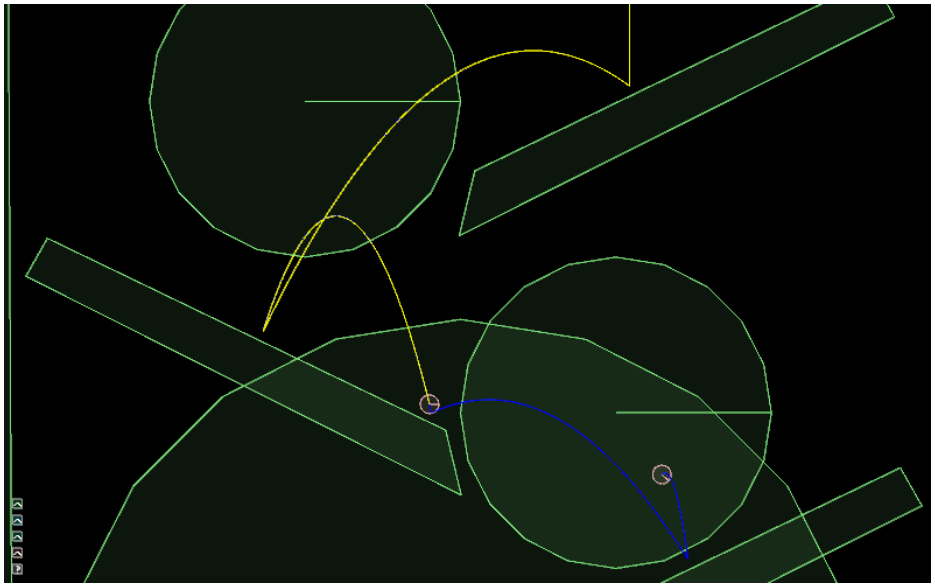


Obrázek 6.2: Testovací scénář padajících kruhů

Další testovací scénář ukazoval funkčnost lokálního zpomalení času. Do herní scény byly umístěny kruhové zóny lokálního zpomalení času a dva kruhy. První kruh padal scénou bez ovlivnění lokálním zpomalováním času a zanechával za sebou modrou stopu. Druhý kruh byl ovlivněn při průletu zónami lokálního zpomalování času a zanechával za sebou žlutou stopu. Vizuálně lze tedy kontrolovat, zda druhý kruh opravdu zpomaluje, kde má, a zda se jeho trajektorie i přes zpomalení shoduje s trajektorií prvního kruhu (žlutá stopa překrývá modrou). Náhled na popsaný testovací scénář lze vidět na obrázku 6.3.

Pokud objekt koliduje se zpomalovací zónou, je okamžitě zpomalen. Pro efekt pozvolnějšího zpomalení lze použít zanoření více zón do sebe. Tento efekt je testován v dalším scénáři a je vyobrazen na obrázku 6.4. Scénář zároveň testuje správné škálování sil, které objekt dostane při kolizi, uvnitř zpomalovací zóny. Opět byly vytvořeny 2 kruhy, na jeden se lokální zpomalení nevztahovalo (žlutá stopa) a na druhý působilo (světle modrá stopa). Byla vykreslována nejen stopa po středu objektu, ale také po jeho rotaci. Vizuálním porovnáním stop lze opět ověřit správnost chování.

Ve třídě `RifleGameState` je udržován ukazatel na aktuální testovací skript a při každé aktualizaci třídy je skript aktualizován. Při přiřazování nového skriptu je nejprve zavolána popsaná metoda `init`. Scénáře jsou užitečné také k prototypování nové funkcionality, což je vidět na skriptu testujícím střelbu na pohybující se cíl (`Script_LinearMovementPredict.h`).



Obrázek 6.3: Testovací scénář lokálního zpomalování času

Dříve, než byla implementována střelba na letící cíl přímo do umělé inteligence ve hře, byla vytvořena scéna, ve které se pohyboval objekt, na který byly vysílány projektily. Po odladění funkcionality je možné ji přesunout ze skriptu na požadované místo.

6.4 Hra, její komponenty a systémy entit

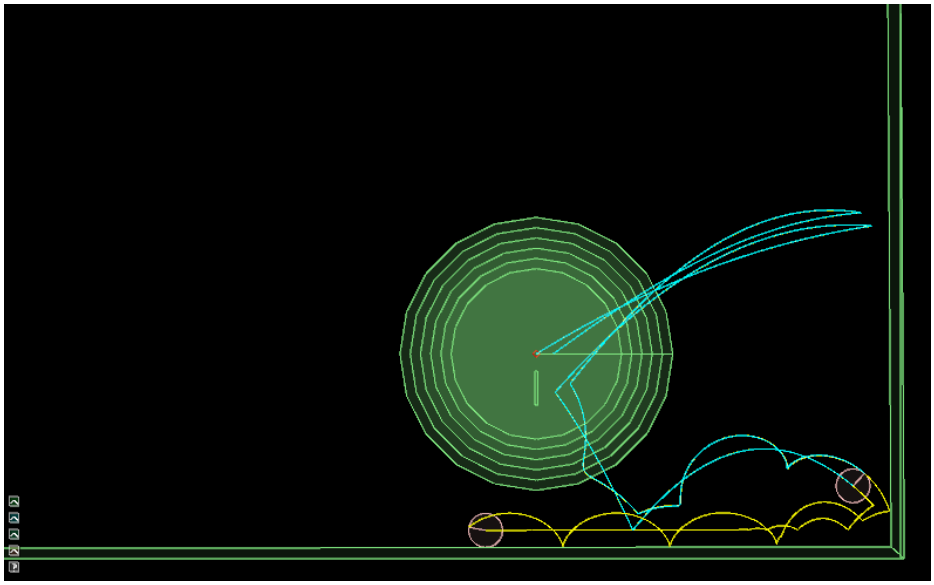
6.4.1 Postavy

Hlavní hrdina a nepřátelé jsou postavy vzpřímené povahy (dle grafické reprezentace mohou být nepřátelé např. vojáci nebo zombie). Pro reprezentaci postavy byla vytvořena stěžejní komponenta hry - `CharacterComponent`. Komponenta je rozsáhlá, proto zde nebude představena celá, ale pouze bude naznačeno, jaké jsou její hlavní zodpovědnosti (detaily o komponentě lze nalézt v implementaci přiložené k této práci). O aktualizaci komponenty se stará `CharacterControlSystem`.

Pár `CharacterComponent` a `CharacterControlSystem` má řadu zodpovědností, mezi hlavní patří např.:

- **pohyb postavy:** Pro správný požitek ze hry je velmi důležité, jak se hlavní hrdina ovládá. Jestli reaguje rychle, jak dlouho setrvá v pohybu, jak moc je jeho pohyb realistický / arkádovitý atd. Je třeba zjišťovat, zda je postava na povrchu, nebo ve skoku, kontrolovat, po jak strmém povrchu se pohybuje. Také je třeba zajistit, aby se postava nezasekla v žádné části herní úrovně apod. Ke zjišťování popisovaných informací se využívají např. fyzikální tělesa a vrhání paprsků.

- **animace postav:** Animace postav ve hře je prováděna přehráváním animací dle současného pohybu a směru míření. Postava v rukou drží zbraň, na animaci rukou je použita inverzní kinematika.



Obrázek 6.4: Testovací scénář pozvolného lokálního zpomalování času

- **míření:** Před vystřelením je nutné zamířit na cíl. Je třeba vyřešit rychlost a způsob míření (postava nesmí ve zpomalovací zóně mířit stejně rychle jako mimo ni, atd.).
- **poskytnutí informace o postavě:** Dotazy typu *Kde má postava hlavu* jsou nezbytné nejen pro míření agentů, ale také např. při řešení viditelnosti mezi postavami.

Tento výčet uvádí pouze některé zodpovědnosti. Bylo by možné každou zodpovědnost umístit do separátní komponenty a vytvořit nové systémy entit, v tomto případě však bylo rozhodnuto vše zapouzdřit do `CharacterComponent` a `CharacterControlSystem`. Zodpovědnosti jsou tak úzce provázané, že jejich oddělení by bylo pracné a zbytečné. Je vždy na uživateli, jakou granularitu komponent zvolí. `CharacterControlSystem` je také součástí `RifleGameState`, kde je vytvořen i aktualizován.

6.4.2 Zbraně

Zbraň je ve hře reprezentována třídou `Weapon`. Třída uchovává informace o současném stavu zbraně (čas od posledního výstřelu, aktuální počet nábojů, atd.) a informace o zbrani obecně. Mezi obecné vlastnosti zbraně patří např. její vzhled, kadence, velikost zásobníku aj. Tyto vlastnosti jsou sdíleny napříč všemi instancemi stejné zbraně, a aby se zabránilo jejich opakování v každé instanci, jsou zabaleny do třídy `WeaponTemplate`, která data spravuje.

Třída `Weapon` obsahuje virtuální metodu `onPrimaryAttack`. Pro vytvoření nové zbraně tedy odvodíme novou třídu od třídy `Weapon` a metodu překryjeme. V metodě provedeme samotný výstřel. U nějakého typu zbraně to může znamenat pouze vytvoření malého kruhu (kulky) a jeho vyslání v aktuálním směru míření. U jiných zbraní (např. brokovnice) je nutné kulek vytvořit více. Uživatel má v metodě úplnou volnost, namísto klasických projektilů lze např. vyslat sledovaný paprsek (*RayCast*) a simulovat tak výstřel laseru.

U každé kulky lze ještě nastavit dvě zpětná volání. Jedno bude volané v každém snímku letu kulky a druhé při expiraci kulky. Je možné implementovat např. řízenou střelu, a to tak, že v ve zpětném volání budeme upravovat v každém snímku trasu projektilu dle požadavků. Zpětné volání při expiraci kulky přidává možnost volby, co se stane s kulkou, která má zaniknout. Jedním řešením je projektil pouze smazat, pokud je ale projektilem raketa či granát, můžeme namísto pouhého smazání simulovat výbuch.

Pro přidání zbraní k postavě slouží `WeaponComponent`. Komponenta udržuje např. seznam držených zbraní a aktuální zbraň. Komponenta také udržuje požadavky na akce. Pokud hráč zmáčkne klávesu pro výstřel či nabití zbraně, uloží se požadavek do `WeaponComponent`. Požadavky jsou zpracovávány v systému entit `WeaponSystem`. Pokud je uložen např. požadavek na výstřel, systém entit zkontroluje čas od posledního výstřelu, kadenci zbraně, dostupnost munice a případně ze zbraně vystřelí. Vystřelení vyvolá popsanou metodu `onPrimaryAttack` na aktuální instanci zbraně (dle `WeaponComponent`).

Vytvořená kulka obsahuje komponentu `ProjectileComponent`. Komponenta ukládá informace o kulce, např. z jaké zbraně byla vystřelena. Po zásahu jsme např. schopni určit, kolik zdraví máme postavě ubrat. Výše popsaná zpětná volání jsou uložena právě uvnitř instance `ProjectileComponent`. Jednoduchý systém entit `ProjectileSystem` aktualizuje podmnožinu entit obsahující komponentu a v každé aktualizaci vyvolává zpětné volání. Ve zdrojovém souboru `CollisionCallbacks.h` si lze prohlédnout, jaká byla implementována zpětná volání při kolizi kulky. Např. při kolizi kulky se zemí kulka zmizí, při kolizi kulky z granátem kulka zmizí a granát vybuchne atd.

6.4.2.1 Příklad - střílení min

Nyní si ukážeme, jak vytvořit zajímavou zbraň - ruční minomet. Zbraň bude fungovat tak, že při výstřelu vystřelí minu, která má určité vlastnosti. Pokud mina narazí do země (stropu, stěny,...) přichytí se tam. Pokud mina za letu narazí na protivníka, exploduje. Pokud je v daném okolí miny nepřítel (je jedno jestli mina letí, nebo je přichycena), mina spustí časový odpočet, po kterém exploduje. Pokud je více min vedle sebe a jedna vybuchne, vybuchnou i miny blízké. Miny, které vystřelí hráč, budou aktivovány pouze nepřáteli, hráč okolo nich bude moci bezpečně projít.

Mina se skládá ze sedmi komponent, čtyři komponenty jsou společné pro všechny projektily:

- **SpatialComponent:** Uchovává transformaci projektilu.
- **PhysicsComponent:** Umožňuje fyzikální simulaci projektilu.
- **SlowedComponent:** Letící mina bude zpomalena v zónách lokálního zpomalování času.
- **ProjectileComponent:** Uchovává informace o projektilu.

Nyní ukážeme zbylé 3 komponenty, které jsou využity pro reprezentaci miny.

- **MineComponent:** Obsahuje poloměr okolí, které mina detekuje, čas, který trvá, než vybuchne a informaci o tom, jestli je mina přichycena k povrchu.

- **ActiveComponent:** Je využita pro informaci o tom, jestli již byla mina aktivována (nepřítel se k mině přiblížil, začal odpočet).
- **ExpirationComponent:** Uchovává čas do vybuchnutí. Před aktivací miny může být čas např. nekonečno, po aktivaci se nastaví odpočet dle vlastností miny (z `MineComponent`).

Dalším krokem je vytvoření fyzikálních zpětných volání při kolizích. Náraz miny do nepřítele je vyřešen zpětným voláním implementovaným obecně pro všechny projektily, které zasáhnou postavu (náraz miny do nepřítele se tedy neliší od nárazu běžné kulky do nepřítele). Projektilu se zavolá expirační akce, tedy mina vybuchne. Muselo vzniknout ještě další zpětné volání, které se vyvolá, pokud mina zasáhne zemský povrch. V tom případě se mina z dynamického fyzikálního objektu změní na statický fyzikální objekt. Přestane být simulován její pohyb, ale je stále zapojena v procesu detekce kolizí. Pokud do přichycené miny střelíme, odpálíme ji. Výbuch miny je simulován vysláním fragmentů do jejího okolí. Tyto fragmenty jsou opět projektily, pokud tedy zasáhnou další minu, mina také exploduje.

Výsek z implementace `MineSystem`, který se stará o aktualizaci min, můžete vidět na ukázce 6.1. V metodě proces jsou aktualizovány všechny entity, které mají uvedené komponenty. Pokud mina není aktivní, má smysl provádět kontrolu okolí. Pokud je již aktivní, `MineSystem` již nic dělat nemusí, o odpočet a výbuch se postará `ExpirationSystem` (pravidelně snižuje čas do expirace a poté zavolá příslušné expirační zpětné volání). Okolí miny je prohledáno výše popsaným prostorovým dotazem. Zjišťujeme, zda se okolo pozice miny ve stanoveném poloměru (uloženém v `MineComponent`) nachází entita, která obsahuje `AIComponent`, tedy nepřítel. Pokud v okolí miny je nepřítel, mina se aktivuje je upraven její čas do výbuchu. `MineSystem` je obsažen v `RifleGameState`, který zodpovídá ze jeho vytvoření a aktualizaci během hry.

```
1  void MineSystem::process( Entity * e )
2  {
3      auto & mc = e->get<MineComponent>();
4      auto & ac = e->get<ActiveComponent>();
5      auto & bc = e->get<ProjectileComponent>();
6      auto & sc = e->get<SpatialComponent>();
7      auto & ec = e->get<ExpirationComponent>();
8
9      if(ac.isDisabled() && bc.isPlayersProjectile())
10     {
11         const Vec2 & position = sc.getTransformation().getPosition();
12
13         SpatialSystem & ss =
14             *m_world2d.getSystemManager()->getSystem<SpatialSystem>();
15
16         SpatialQueryResult res;
17
18         ss.circularQuery(position, mc.getDetectionRadius(), res);
19
20         res.processingFilter<AIComponent<Agent>>([&](Entity * e)
21         {
22             ac.setActive(true);
23             ec.setLifeTime(mc.getTickingTime());
24
25             return;
26         });
27     }
28 }
```

Kód 6.1: Implementace `MineSystem`

6.4.3 Umělá inteligence

Zadáním práce bylo vytvořit hru s prvky umělé inteligence. Vytvořil jsem tedy dva typy *inteligentních* protivníků. První protivník se snaží dostat na dohled k hráči a poté ho zabít. Druhý protivník naopak vybírá náhodná místa na mapě, kde se usadí a hráče ostřeluje. K programování umělé inteligence byly použity behaviorální stromy popsané v kapitole 5. Abychom mohli behaviorální stromy naplnit vhodnými daty, je třeba agentům poskytnout nějaké znalosti, dle kterých se budou rozhodovat. Pro umělou inteligenci byly také vytvořeny komponenty a systémy entit a v následujících odstavcích si některé ukážeme. Poté předvedeme, jak vše složit ve výsledné chování agenta.

Již v generickém frameworku je k dispozici komponenta `VisionComponent`, která uchovává informace o vzájemné viditelnosti. Komponenta zapouzdřuje informace o vzájemné viditelnosti mezi postavami. Např. agent může žádat od komponenty informace typu: *Vidím hráče?*, *Vidí mě hráč?* apod. Viditelností je myšlena přímá viditelnost. Postavy na sebe mohou úspěšně útočit i přesto, že se přímo nevidí, např. hozením granátu. Řešení samotné viditelnosti je však závislé na hře.

Ve hře *Rifle* bylo zvoleno řešení viditelnosti pomocí vrhání paprsků. O to se stará třída *VisibilitySystem*. Implementace funguje tak, že se vyšle daný počet paprsků, které směřují od očí jedné postavy do bodů na druhé postavě. Body na druhé postavě jsou voleny v závislosti na počtu paprsků tak, aby rovnoměrně pokryly celou postavu. Pokud alespoň jeden z paprsků po celé své délce nenarazí na žádnou překážku, můžeme prohlásit, že je testovaná postava viditelná z pohledu testující postavy.

Další implementovanou komponentou je `AIAgentKnowledgeComponent`, která udržuje nejrůznější znalosti agenta. O jejich správu se stará přidružený `AIAgentKnowledgeSystem`. Mezi znalosti patří např.:

- **poslední pozice:** Díky sledování vlastních pozic spolu s požadavky na pohyb můžeme detekovat, zda se agent zasekl či ztratil a můžeme mu najít novou cestu.
- **doba viditelnosti hráče:** Touto znalostí se dá modelovat např. reakční doba. Pokud by agent např. vystřelil okamžitě, jakmile vidí hráče, střílel by i v situacích, kdy se hráč pouze mihne. Takové chování nepůsobí přirozeně. Pokud však vystřelí až tehdy, pokud hráče vidí déle, působí to mnohem realističtěji.
- **doba posledního výstřelu:** Pomocí této znalosti je možné simulovat střelbu s rozvahou. Pokud je agent od hráče daleko, tak může vystřelit jednou za sekundu a mezi tím pozorovat, jak dopadl předchozí výstřel. Pokud by na dálku střílel s takovou kadencí, jakou to zbraň umožňuje, působilo by to nereálně a agent by brzy mohl přijít o všechnu munici.
- **doba hlídkování:** Implementované chování ostřelovače si zde ukládá, jak dlouho již je na dané pozici. Může si tak hlídat dobu, po které se přesune na jiné stanoviště.

Jak tedy navrhnout behaviorální strom pro požadované chování? Ukážeme si to na příkladu prvního protivníka, který se snaží dostat k hráči a pokud ho nějakou dobu vidí, vystřelí po něm. Je zřejmé, že chování se skládá ze dvou částí: pronásledování hráče a útok. Každá část je složena z dalších chováních, která jsou reprezentována uzly v behaviorálním stromu.

Strom bude mít kořenový uzel *selekci*, pod kterým budou zavěšeny další dva uzly typu *sekvence*. Jeden uzel pro pronásledování a druhý pro útok. Oba jsou uzly typu *sekvence*, protože se skládají z dalších již atomických akcí, které musejí být všechny popořadě vykonány, aby bylo uskutečněno celkové chování. Strukturu lze znázornit takto:

- *Selekce* **Chování** | Zastřešuje celé chování agenta.
  - *Sekvence* **Navigace** | Naviguje agenta k hlavnímu hrdinovi.
    - *Akce* **Zjisti cíl cesty** | Zjistí, kde se nachází hlavní hrdina, dle jeho pozice vybere nejvhodnější uzel v grafu jako cílový.
    - *Akce* **Najdi cestu** | Vyhledá v navigačním grafu cestu mezi cílovým uzlem a uzlem nejblíže současné polohy hráče.
    - *Akce* **Vyhlaď cestu** | Vyhladí cestu, aby se agent choval přirozeně. Např. počáteční uzel cesty (nejbližší k agentovi) může být přesně na opačné straně, než směřuje zbytek cesty. V případě nevyhlazené cesty se tak agent nejprve kousek vrátí a poté teprve jde za cílem.
    - *Akce* **Traverzuj cestu** | Zde již agent traverzuje cestu, tj. směřuje vždy k následujícímu uzlu dle traverzované hrany (skokem, během, ...), dokud k němu nedorazí. Poté směřuje k uzlu dalšímu, dokud není v cíli. Traverzace je déle trvající chování a má dvě vstupní podmínky, je možné traverzovat pokud: 1) hráč je stále nejblíže k uzlu, který považujeme za cílový, 2) agent nevidí hráče déle jak X vteřin. Porušení jedné z podmínek vede k nové traverzaci stromu. Buďto se naplánuje nová cesta (v případě podmínky 1) a nebo se přejde na uzel Útok (podmínka 2).
  - *Sekvence* **Útok** | Zastřešuje útočení agenta na hráče.
    - *Akce* **Zamiř** | Dle vzájemné polohy agenta a hráče podá požadavek na zamíření na hráče. Míření má určitou rychlost, není tedy okamžité. Může se tedy stát, že akce se pouze přiblíží k požadovanému směru míření. Bližší přiblížení se provede v další iteraci.
    - *Akce* **Vystřel** | Pokud předchozí akce míření skončila úspěšně (již je zamířeno na hráče), tak tato akce vydá požadavek ke střelbě.

Struktura stromu druhého typu agenta (ostřelovače) je komplikovanější, proto zde nebude uvedena. Obě chování si můžete prohlédnout implementované v souboru `AI.cpp`. V souboru je např. ještě behaviorální strom pro otočnou střílnu (*Turret*). Střílna může být přidělána např. na strom v herní úrovni a skenuje paprskem zadanou oblast. Pokud v oblasti zaregistruje hráče, tak "zbystří". Pokud hráč neopustí oblast v daném časovém limitu, tak po něm začne střílna střílet.

Zbývající systémy

`RifleGameState` mimo popsaných obsahuje velké množství dalších systémů entit, např.:

- **HudRenderSystem:** Pro vykreslování informací o zdraví, zbrani, apod.

- **BulletTimeSystem:** Pro správu celkového zpomalení hry.
- **RifleCameraSystem:** Pro pohyb s kamerou, která sleduje hráče.
- **BasicDirectorSystem:** Pro centrální *mozek* reprezentující umělou inteligenci.
- **ProgressiveRaySystem:** Pro simulování pomalu letícího útočného paprsku.
- **PositionHeatmapSystem:** Pro správu *heatmapy*, tedy datové struktury, kde jsou uchovávány záznamy, kde se hráč na mapě pohybuje nejvíce (vhodné pro řízení AI).
- **ProjectedTrajectorySystem:** Pro simulování trajektorie letu granátu.

Vytvořená hra toho obsahuje mnohem více, než je zde popsáno, avšak samotná hra slouží pouze k demonstraci a ověření funkce komponent a systémů entit a není těžištěm této práce. Nebylo zde např. popsáno, jak se vytváření a načítají herní úrovně, jak funguje základní vykreslování, jak se dá postupovat pro dosažení efektu hadrové panenky, jak registrovat a používat konzolové příkazy a mnoho dalšího. Proto pro další detaily doporučuji prozkoumání zdrojových kódů přiložených k této práci.

Kapitola 7

Diskuze

Herní objekty se dají reprezentovat více způsoby, nejvíce používaným způsobem je reprezentace pomocí hierarchie tříd. Alternativou je použití komponentového přístupu.

Kdy zvolit reprezentaci pomocí hierarchie tříd?

Hlavní nevýhodou hierarchie je její špatná modifikace, která sebou přináší řadu problémů (popsaných v 1.1). Přesto je hierarchie hojně používaná a to především kvůli intuitivní reprezentaci objektů. Její použití je vhodné pro herní tituly, které mají detailní dokument popisující hru ještě před napsáním prvních řádek kódu (*Design document*). Design document také musí být brán jako neměnný, poté může být hierarchie objektů navržena v již finální podobě. Hierarchie tedy není vhodná, pokud při vývoji her mohou nastat změny (byť malé) v konceptu hry, což při moderním vývoji her bývá časté.

Proč nebyla již dávno hierarchie nahrazena komponentovým přístupem?

Komponentový přístup je flexibilní a netrpí problémy spojenými s hierarchií. Nevýhodou je však to, že při prvním pohledu se komponentová reprezentace objektů jeví neintuitivní a komplikovaná. Vývojáři se často drží pravidel správného objektově orientovaného programování, proto se myšlenka, že by jeden herní objekt nebyl zapouzdřen v jedné třídě, jeví jako chybná. Narušení zapouzdření však vede k tomu, že je možné oddělit datovou a logickou část kódu a vytvořit tak opakovaně použitelný kód.

Kterou variantu komponentového přístupu použít?

Při bližším zkoumání komponentových přístupů se ukázalo, že existují desítky způsobů, jak reprezentaci herních objektů pomocí komponent pojmout. Vývojář vždy musí stanovit, co od systému očekává a na základě toho zvolit vhodný přístup. Pokud chce vytvořit co nejrychleji jen jednu hru, ale chce se vyhnout problémům spojených s hierarchií tříd, může použít přístup Boba Nystroma (2.1). Není nutné vytvářet podpůrný framework, vývojář přímo dělí kód na komponenty, které shlukuje. Pokud vývojář nevylučuje vytvoření pokračování hry, či další hru, vyplatí se investovat čas do vytvoření jednoduchého systému, který umožní

komponenty lépe oddělovat, zaměňovat a opakovaně používat ve více projektech. Příklad takového systému byl popsán v sekci 2.2).

I robustnější systém ze sekce 2.2 však nemusí být ideální v případě, že vývojář, či vývojové studio vytváří a plánuje tvořit větší množství her. Pro vytvoření komponent je nutné psát kód, který úzce nesouvisí s programovanou funkcionalitou. Abychom umožnili pohodlné vytváření a efektivní používání komponent, je nutné vytvořit poměrně složitý systém na správu objektů. Takový systém byl popsán a vytvořen v rámci této práce. Počáteční časová investice je výrazně vyšší (autorovi práce trvalo vytvoření frameworku asi 500 hodin), než v ostatních přístupech, přináší ale sebou značné ušetření práce (tedy i času) při vývoji dalších a dalších komponent a celých her.

Lze komponenty a systémy entit opravdu tak lehce opakovaně použít?

Na obrázku 7.1 je možné vidět ukázku ze hry *Space Invaders*. Díky vytvořenému frameworku a hře *Rifle* mohla tato hra vzniknout za méně než dvě hodiny. Přestože obě jsou velmi odlišné hry, bylo možné znovu použít komponenty, jako např. fyzikální, poziční, zdravotní, vykreslovací a další. Pokud by hra *Rifle* (z kapitoly 6) byla implementována pomocí hierarchie tříd, tak by např. projektil dědil ze třídy umožnující lokální zpomalení času (specifický efekt implementován v rámci 2D frameworku), které se ve *Space Invaders* vůbec neobjevuje. Znovu použít objekt z hierarchie by bylo problematické. V případě komponentového přístupu stačilo vzít komponenty z původního projektilu s výjimkou `SlowedComponent` a implementace projektilu byla hotova.

Obrázek 7.1: Další hra vytvořená pomocí vytvořeného frameworku

Jakou zvolit granularitu komponent?

Abychom předešli redundanci dat při opakovaném používání komponent, je nejlepší vytvářet atomické komponenty, tedy takové, které se týkají jedné dále nedělitelné funkcionality. Tím naroste počet komponent použitých ve hře, ale díky práci s komponentami v konstantním čase

nečiní veliký počet komponent problém. Pokud víme, že danou funkcionalitu (neatomickou) opakovaně používat nebudeme, je výhodné vytvořit větší komponentu a systém entit, kde bude funkcionalita zapouzdřena a pohodlně vyřešena logika dané věci. Tento postup byl zvolen např. v komponentě `CharacterComponent`.

Kapitola 8

Závěr

Výsledkem této práce je kompletní návrh a prototyp frameworku, který reprezentuje herní objekty pomocí komponent. Vytvořený framework je pro programátora her velmi vhodný, nové komponenty a systémy entit se vytvářejí a propojují velmi jednoduše a intuitivně. Komunikace mezi komponentami probíhá v konstantním čase, a proto je implementovaný prototyp velmi vhodný pro použití v rozsáhlých herních projektech. Implementovaný komponentový přístup je obecný a lze ho využít nejen pro hry (textové, 2D, 3D,...), ale i pro další aplikace pracující v reálném čase.

V rámci práce byla také vytvořena specializace frameworku pro 2D počítačové hry. Uživatel může využít předpřipravených komponent a systémů entit např. pro skeletální animace, prostorové dotazy a efekty zpomalování času. Specializovaný framework byl využit pro demonstraci tvorby hry, na které bylo ukázáno, že tvorba probíhá intuitivně, a že vzniká opakovaně použitelný kód. Jako důkaz, že vytvořené komponenty a systémy entit je možné opětovně použít, vznikla druhá hra, která až na jednu novou komponentu sestává pouze z komponent již existujících.

Text práce obsahuje detailní uvedení do problematiky, a je proto vhodný i pro čtenáře, kteří nemají zkušenosti s použitím komponent a s problémy spojenými s jejich nepoužitím. Dále text obsahuje rozbor různých přístupů ke komponentám, návrh a popis implementace robustního komponentového systému a ukázku implementace hry pomocí navrženého prototypu. Napříč textem je mnoho příkladů a k práci je přiložena funkční implementace. Práce může tedy sloužit jako zdroj informací i vývojářům, kteří s komponentami zkušenosti mají.

Předložený framework je možné dále rozšiřovat. Vhodným rozšířením je např.:

- **vytvoření dalších obecných komponent**: Přidáním vhodných komponent by se usnadnil vývoj dalších her. Vytvořit hru na hrdiny (RPG) či závodní hru bude vyžadovat implementaci nových komponent (např. `CollectableItemComponent`, `CarPhysicsComponent`).
- **vytvoření editoru pro tvorbu objektů**: S narůstajícím počtem komponent by bylo dobré vytvořit grafický editor, který by umožňoval vytváření nových herních objektů pouze z existujících komponent. Grafický editor by byl pro vytváření objektů uživatelsky příjemnější.

Literatura

[1] J. Arlow. *UML 2 and the unified process : practical object-oriented analysis and design*. Addison-Wesley, Upper Saddle River, NJ, 2005.

[2] S. Bilas. A data-driven game object system. In *Gas Powered Games, Game Developers Conference Proceedings*, 2002.

[3] T. Brown. Fast entity component system. http://www.openprocessing.org/sketch/18023, Jan. 2011. [Online; accessed 29-October-2013].

[4] M. Buckland. *Programming game AI by example*. Wordware Pub, 2005.

[5] M. Chady. Theory and practice of game object component architecture. In *Game Developers Conference Canada*, Vancouver, BC, 2009.

[6] S. Chris. *Game Programming Gems 6*, chapter Game object Component System, pages 393–403. Cengage Learning; 1 edition, 2006.

[7] D. Church. Object systems - methods for attaching data to objects, and connecting behaviors. In *Game Developers Conference Proceedings*, 2002.

[8] A. Duran. Building object systems - features, tradeoffs, and pitfalls. In *Gas Powered Games, Game Developers Conference Proceedings*, 2003.

[9] D. Eberly. *3D Game Engine Design: A Practical Approach to Real-Time Computer Graphics - Apendix A*. Morgan Kaufmann Elsevier Science distributor, San Francisco, CA Oxford, 2007.

[10] G. Fiedler. Fix your timestep! http://gafferongames.com/game-physics/fix-your-timestep/, Sept. 2006. [Online; accessed 29-October-2013].

[11] E. Folmer. Component based game development – a solution to escalating costs and expanding deadlines? In *Component-Based Software Engineering*, volume 4608 of *Lecture Notes in Computer Science*, pages 66–73. Springer Berlin Heidelberg, 2007.

[12] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-oriented Software*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1995.

[13] J. Gregory. *Game engine architecture*. A K Peters/CRC Press, 2009.

[14] M. Haller, M. Haller, and W. Hartmann. A generic framework for game development. In *In Proceedings of the ACM SIGGRAPH and Eurographics Campfire*, 2002.

[15] A. Martin. Entity systems are the future of mmog development - blog series. http://t-machine.org/index.php/2007/09/03/entity-systems-are-the-future-of-mmog-development-part-1/, Sept. 2007. [Online; accessed 29-October-2013].

[16] H. Matthew. *Game Programming Gems 4*, chapter A System for Managing Game Entities, pages 69–83. Cengage Learning; 1 edition, 2004.

[17] S. Meyers. *Effective C++ : 55 specific ways to improve your programs and designs*. Addison-Wesley, Upper Saddle River, NJ, 2005.

[18] D. Michael. *A software architecture for games*. PhD thesis, University of the Pacific Department of Computer Science Research and Project Journal, 2009.

[19] B. Nystrom. Component. http://gameprogrammingpatterns.com/component.html, 2010. [Online; accessed 29-October-2013].

[20] E. B. Passos, J. W. S. Sousa, E. W. G. Clua, A. Montenegro, and L. Murta. Smart composition of game objects using dependency injection. *Computers in Entertainment (CIE) - SPECIAL ISSUE: Games*, 7(4):53:1–53:15, Jan. 2010.

[21] J. Plummer. *A flexible and expandable architecture for computer games*. PhD thesis, Arizona State University, 2004.

[22] S. Rabin. *AI game programming wisdom 4*. Course Technology, Cengage Learning, Boston, MA, 2008.

[23] B. Rene. *Game Programming Gems 5*, chapter Component Based Object Management, pages 25–37. Cengage Learning; 1 edition, 2005.

[24] H. Sutter. Virtuality. http://www.gotw.ca/publications/mill18.htm, Sept. 2001. [Online; accessed 29-October-2013].

[25] B. Warric. *Game Programming Gems 5*, chapter A Generic Component Library, pages 177–187. Cengage Learning; 1 edition, 2005.

[26] M. West. Evolve your hierarchy. http://cowboyprogramming.com/2007/01/05/evolve-your-heirachy/, Jan. 2007. [Online; accessed 29-October-2013].

[27] K. Wilson. Game object structure: Inheritance vs. aggregation. http://gamearchitect.net/Articles/GameObjects1.html, 2002. [Online; accessed 29-October-2013].

Příloha A

Slovníček

V této příloze jsou stručně vysvětleny nejdůležitější pojmy, které se v práci objevují:

- **entita** (*entity*): herní objekt
- **komponenta** (*component*): část herního objektu (dle komponentového přístupu může část obsahovat data i funkcionality, nebo pouze data)
- **systém entit** (*entity system*): část kódu, ve které se řeší jedna konkrétní funkcionalita pro podmnožinu entit, které obsahují komponenty (pouze datové) týkající se dané funkcionality
- **framework** (*framework*): podpůrná knihovna pro programování her
- **framework2d** (*framework2d*): specializace frameworku pro vytváření 2d her
- **Rifle** (*Rifle*): název hry vytvořené v rámci této práce

Příloha B

Obsah přiloženého DVD

Na přiloženém DVD je v kořenovém adresáři soubor `readme.txt`, ve kterém je popsáno, jaké jsou požadavky na cílový počítač, jak vytvořit projekty ze zdrojových souborů a jak projekty sestavit ve výsledné spustitelné aplikace. Soubor také popisuje ovládání aplikací.

- V adresáři **executable** lze nalézt spustitelné verze vytvořených her.
- V adresáři **source** se nachází veškeré zdrojové kódy, závislosti a další potřebná data pro projekty.
- V adresáři **text** je umístěna tato práce ve formátu PDF.
- V adresáři **video** jsou dostupné video soubory demonstrující vytvořené aplikace.