VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
FACULTY OF INFORMATION TECHNOLOGY

ÚSTAV POČÍTAČOVÉ GRAFIKY A MULTIMÉDIÍ
DEPARTMENT OF COMPUTER GRAPHICS AND MULTIMEDIA

KOMPONENTNÍ SYSTÉM PRO HERNÍ GRAFICKÝ ENGINE

GRAPHIC ENGINE BASED ON ENTITY COMPONENT SYSTEM

BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE — **TOMÁŠ POLÁŠEK**
AUTHOR

VEDOUCÍ PRÁCE — **MICHAL ŠPANĚL, Ing., Ph.D.**
SUPERVISOR

BRNO 2017

Abstrakt

Cílem této bakalářské práce je návrh a implementace knihovny pro správu entit, která umožňuje návrh pomocí *Entity-Component-System* paradigmatu. Součástí práce je analýza aktuálního stavu návrhu software, z pohledu vývoje her a rozbor dnes používaných technik pro práci s entitami včetně srovnání jejich výhod. Dále se práce zabývá důvody, proč tento nový způsob návrhu her lépe vyhovuje architektuře hardware moderních počítačů, primárně se zaměřením na paměťovou hierarchii.

V práci je dále navržen entitní systém, který umožňuje dynamickou kompozici entit za běhu aplikace z komponent definovaných před jejím spuštěním. Výsledný systém je navržen s ohledem na snadnou paralelizaci a umožňuje uživatelské rozšíření. Následně je provedena implementace výše zmíněného návrhu v programovacím jazyce *C++*.

Výsledkem této bakalářské práce je multiplatformní knihovna, pojmenovaná *Entropy*, která uživatelům zpřístupňuje návrh aplikace pomocí *ECS* paradigmatu.

Abstract

The goal of this bachelor thesis is the design and implementation of a library for entity management, which allows software design using *Entity-Component-System* paradigm. Analysis of the current state of software design is presented, including comparison of techniques used for entity representation. Potential advantages are discussed from the hardware point of view, primarily using the memory hierarchy found in modern computers.

Thesis proposes design of an entity system, using the *ECS* paradigm, which allows dynamic composition of entities during runtime of application, from predefined components. The resulting system is designed with easy parallelization and user customizability in mind. The design is then implemented using the *C++* programming language.

The result of this bachelor thesis is multi-platform library, called *Entropy*, which enables the users to design applications using *ECS* paradigm.

Klíčová slova

Komponentní systém, Entitní systém, ECS, Kompozice, Data-oriented design, Paralelismus, C++

Keywords

Component system, Entity-component system, Entity-component-system, ECS, Composition, Data-oriented design, Parallelism, C++

Citace

POLÁŠEK, Tomáš. *Komponentní systém pro herní grafický engine*. Brno, 2017. Bakalářská práce. Vysoké učení technické v Brně, Fakulta informačních technologií. Vedoucí práce Španěl Michal.

Komponentní systém pro herní grafický engine

Prohlášení

Prohlašuji, že jsem tuto bakalářskou práci vypracoval samostatně pod vedením pana Ing. Michala Španěla, Ph.D. Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

.....
Tomáš Polášek
1. května 2017

Poděkování

Děkuji vedoucímu bakalářské práce, panu Ing. Michalu Španělovi, Ph.D. za odborné vedení a konzultace problémů, které se při její tvorbě vyskytly.

Obsah

Kapitola 1

Úvod

Díky rostoucím požadavkům na moderní hry – jak ve složitosti herních principů, tak i věrnosti grafické reprezentace – se stále zvyšují požadavky na *herní engine*, použitých v jejich tvorbě. Mezi hlavní požadavky patří široké využití v mnoha herních žánrech, možnosti multiplatformního nasazení, efektivní použití dostupného hardware, ale také jednoduchost a efektivita práce ve velkých týmech. Tyto, ale i další problémy řeší návrhový vzor *entity-component-system*, který v základu používá *kompozici* místo *dědičnosti*.

Jelikož je tato metoda vcelku nová, širší využití začíná až v posledních letech, existuje mnoho návrhů, jak by *entitní systém* založený na kompozici měl vypadat. Nejčastěji je návrh rozdělen do tří celků – *entita*, *komponent* a *systém*. Často se také využívá tzv. *datově orientovaného návrhu*[31], který specifikuje návrh aplikace za použití analýzy vstupních a výstupních dat jednotlivých částí.

Cílem práce je návrh a implementace *entitního systému* založeného na *datově orientované kompozici*. Hlavními požadavky na tento systém jsou:

- Použitelnost ve vícevláknových aplikacích.
- Efektivní využití hardwarových prostředků.
- Jednoduché rozhraní a integrace do existujících projektů.

Mezi další požadavky, z pohledu *herního enginu*, jsou efektivní práce s velkým množstvím *entit*, kde většina z nich nemusí být aktuálně používána a také komunikace mezi jednotlivými *komponenty* a *entitami*.

Práce je rozdělena do čtyř logických celků – teoretická část, návrh systému, implementace a vyhodnocení. Kapitola 2 popisuje aktuální stav návrhu *software*, primárně z pohledu herního vývoje. Nejdříve zde jsou přiblíženy požadavky na *entitní systémy*. Následuje porovnání aktuálně používaných způsobů reprezentace *entit*, jejich výhody a nevýhody. Závěr obsahuje způsoby využití vícevláknových konfigurací. Kapitola 3 zahrnuje kompletní návrh *entitního systému*, který je obecný – nezávislý na implementačním jazyce. Nejdříve je uveden přehled celého systému, následuje podrobnější popis jeho částí a rozbor různých způsobů řešení. Další kapitola (4) je věnována implementaci v jazyce C++. Následuje kapitola **??**, ve které je předvedena funkčnost systému na návrhu a implementaci demo hry. Poslední kapitola shrnuje výsledný systém z pohledu výkonosti a srovnává jej s podobnými volně přístupnými knihovnami.

Kapitola 2

Teoretický rozbor

Tato kapitola se věnuje teorii *entitních systémů*, jejich historii a požadavkům na ně kladeným. Dále obsahuje rozbor běžně používaných způsobů reprezentace *entit* a srovnává jejich výhody a nevýhody. Následuje popis vlastního *entitního systému* založeného na kompozici a definice hlavních pojmů – *entita*, *komponent* a *systém*. Na závěr se věnuje metodám paralelizmu, které jsou dnes používány.

2.1 Entitní systémy

Entitní systém je část herního enginu, který zprostředkovává správu *entit* – objektů ve virtuálním světě. Alternativní název, který se také často používá, je *herní objekt* (*Game Object* [15]). Primární funkcí *entit* je propojení jednotlivých modulů a částí aplikace, které jsou vyvíjeny odděleně – např. herní logika a simulace fyziky.

2.2 Požadavky na návrh

Stále zvyšující se požadavky na složitost herních principů, věrnost grafické reprezentace a velikost virtuálních světů mnohonásobně ztěžují návrh *herních enginů*, na kterých jsou hry stavěny. Dalším problémem je znovupoužitelnost již vytvořených částí, nejen ve stejném projektu, ale stále častěji i v dalších hrách. Z výše zmíněných důvodů vznikají techniky pro organizaci kódu a knihy *návrhových vzorů*[33][40], které obsahují zkušenosti a prověřené způsoby jak navrhovat *software*. Důležitou vlastností správného návrhu, je také modularita – oddělení částí systému. Tyto techniky umožňují pracovat velkému množství programátorů na jednom projektu.

Mezi techniky dnes používané patří primárně *objektově orientované návrh*, ale stále častěji také návrh *datově orientovaný*, kterým se zabývá část 2.4.

Způsob reprezentace *entit* – objektů ve virtuálním světě – je jedním z důležitých rozhodnutí v návrhu *herního enginu*. Entity jsou často používány pro komunikaci mezi jednotlivými podsystémy – např. vykreslení efektu, který vznikl v *gameplay* kódu. Přílišná provázanost však většinou znamená zvýšený výskyt programovacích chyb (*bugs*) [40]. Podrobněji se tímto zabývá sekce 2.5.

2.3 Architektura moderních počítačů

Důležitou součástí tvorby her je analýza cílového hardware, na kterém hra poběží. Velkou výhodou je v dnešní době velmi vysoká podobnost všech různých herních systémů. Stolní počítače, ale i některé nové konzole (*Playstation 4* a *Xbox One* [45]), již všechny používají architekturu procesorů *x86* [22] [19]. Stále důležitější platformou jsou také mobilní zařízení, které používají architekturu procesorů *ARM* [17]. Výhodou, z pohledu návrhu *software*, je velmi podobná architektura paměti, která umožňuje optimalizace, které fungují na všech často používaných platformách.

Moderní procesory dokáží velmi rychle vykonávat jednoduché instrukce a za použití mechanizmů (např. *pipelining* [11]) se stále zvyšuje počet instrukcí za jeden cyklus (*Instructions per Cycle* [8]). Problém nadchází v případě, kdy procesor operuje s daty, které nejsou k dispozici v jeho registrech. Rozdíl v rychlosti procesorů a pamětí stále roste [46], způsobem minimalizace tohoto rozdílu jsou procesorové *cache* (*vyrovnávací paměť*).

Právě díky rozdílům rychlostí jednotlivých typů pamětí [39] vznikají nové způsoby, jak navrhovat aplikace, které umožňují efektivně využívat hierarchii procesorových *cache*. Jedním z nich je *datově orientovaný návrh*, kterým se zabývá následující sekce. Mezi důležité parametry pro efektivní využití *cache* patří lokalita odkazů [6] a udržování jejich koherence [5]. Lokalita odkazů je primárně dělena na 2 typy – časová a prostorová. Časovou lokalitou je myšleno opakované používání stejných dat, kdy se data při prvním použití zapíší do *cache* a dále již není třeba přístup k hlavní paměti. Načítání dat do *cache* je prováděno v blocích (*cache line*), které jsou specifikovány pro každý procesor. Program, který využívá data, které jsou v paměti blízko (vejdou se do jedné *cache line*) má dobrou prostorovou lokalitu odkazů. Udržování koherence procesorových *cache* se primárně projevuje ve více-jádrových systémech a podrobněji se jím zabývá sekce 2.8.

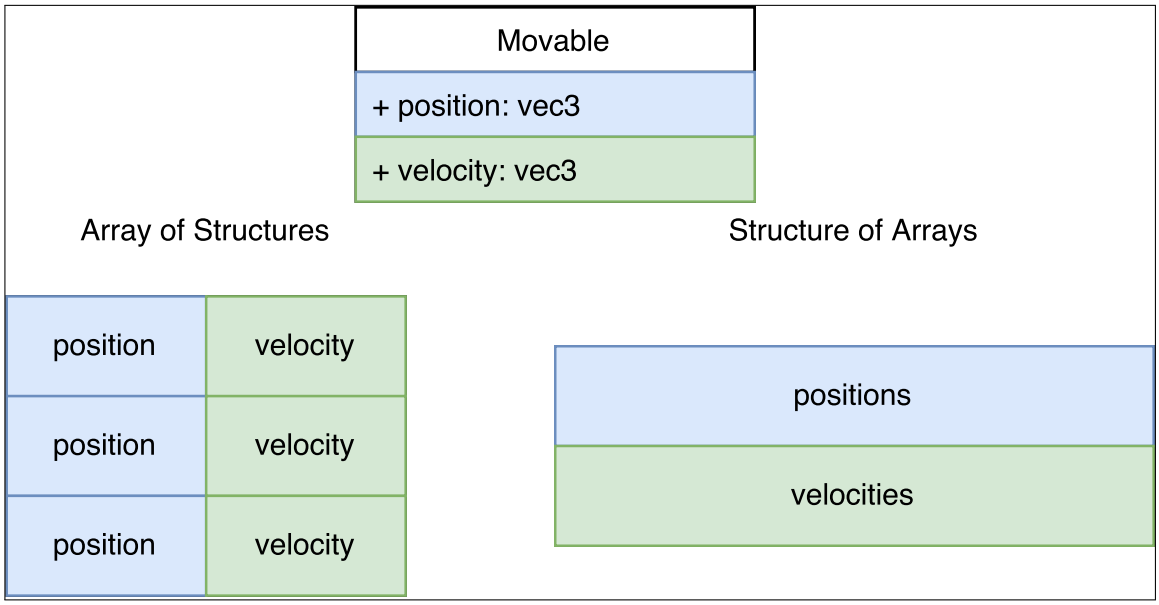
2.4 Datově orientovaný návrh

Dnes nejpoužívanější způsob návrhu – *objektově orientovaný* (*OOD*) – umožňuje abstrahovat od fyzického hardware, na kterém výsledná aplikace běží a řešit daný problém jeho dekompozicí do objektů. Základní stavební kámen je v tomto případě objekt – agregace hodnot a proveditelných operací. Toto umožňuje řešení problémů transformací objektů z reálného světa na objekty virtuální, se kterými dokáží lidé pracovat a zároveň jsou interpretovatelné i překladači programovacích jazyků. Toto je však problematické pro moderní výpočetní hardware, který je optimalizovaný na jednoduché datové transformace.

Kvůli výše zmíněnému problému a stále vyšší potřebě optimalizace se objevuje *datově orientovaný návrh*[31] (*DOD*), který se naopak zaměřuje na data, které aplikace používá. Nevýhodou je snížení čitelnosti výsledného kódu a složitější transformace návrhu aplikace – řešení problému – ve výsledný program. Základní myšlenkou je oddělení dat a operací nad nimi, toto umožňuje efektivnější využití paměti a procesorových *cache*.

Častým příkladem rozdílů mezi *OOD* a *DOD* je transformace seznamu struktur (*SOA*) ve strukturu seznamů (*AOS*), který lze vidět na obr. 2.1.

Základem *DOD* je důkladná analýza aplikačních dat – jejich obor hodnot, transformace, definice vstupních dat, požadovaný výstup atp. Často je využíváno datové struktury pole, jehož vlastnosti umožňují rychlou iteraci. Mezi výhody *DOD* patří také dobrá lokalita dat a odkazů, což umožňuje efektivní využití procesorových *cache*. Díky těmto vlastnostem se *DOD* stále častěji používá v herním průmyslu[9][23].



Obrázek 2.1: Příklad transformace pole struktur na strukturu polí.

Entitní systém navržený v této práci vychází z principů *DOD – datově orientovaná kompozice*[30]. Blíže se tímto způsobem skládání *entit* zabývá sekce 2.5.3.

2.5 Reprezentace entit

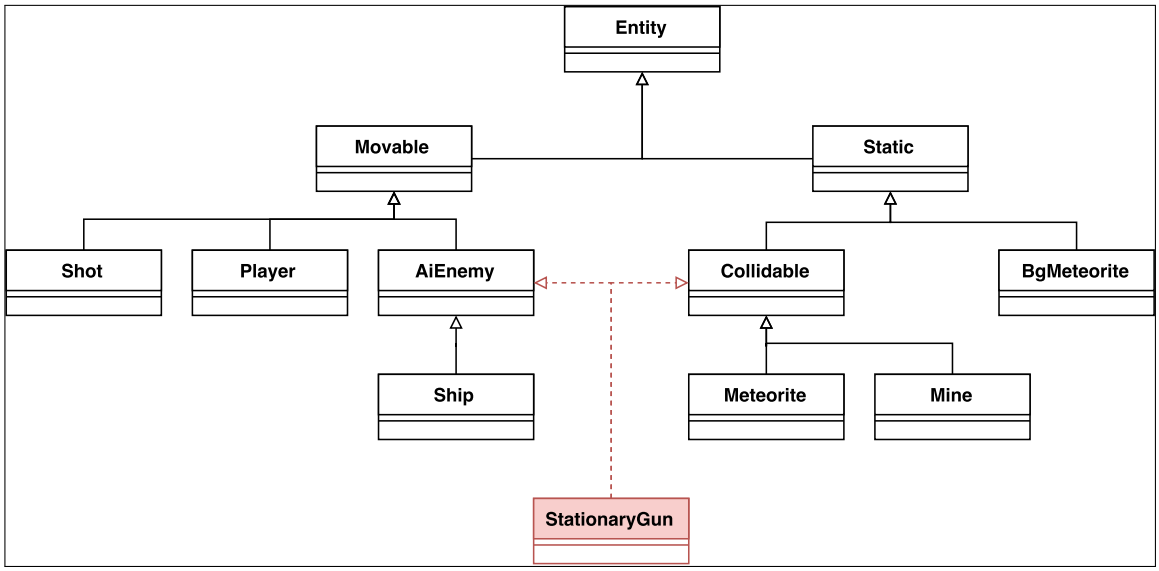
Tato sekce obsahuje rozbor nejpoužívanějších způsobů reprezentace *entit*[47] a jejich *chování*. Pod pojmem *entita* je myšlen objekt ve virtuálním světě a *chování* jsou operace, které *entita* dokáže provést – pohyb, vykreslení, kolize atp.

Jednotlivé reprezentace jsou srovnány na příkladu jednoduché 2D *shoot'em up* hry. Metody jsou hodnoceny na základě složitosti implementace, návrhu s jejich použitím a výhodnosti z pohledu hardware. Závěry zde vyvozené jsou primárně zaměřené na *třídní* objektově orientované jazyky (*C++*, *JAVA*, *Python* atp.), ale částečně je lze aplikovat i na *objektově orientované* programovací jazyky obecně.

2.5.1 Objektově orientovaná hierarchie

Pod pojmem *objektově orientovaná hierarchie* (*OOH* [40]) je míněn způsob skládání nových typů entit za využití *dědičnosti* a často také *polymorfismu*. Příklad hierarchie, navržené pro ukázkovou 2D hru, lze vidět na obr. 2.2. V kořenu stromu hierarchie je, v případě *OOH*, bázová třída, která umožňuje uniformní skladování entit. Konkrétní entity, které existují v herním světě, jsou listy ve stromy dědičnosti.

Množina akcí entity je nastřádána průchodem stromu dědičnosti od kořene k listu, kde se konkrétní entita nachází. Existují dva často používané způsoby definice těchto akcí. První z nich je přímá definice metody v děděné třídě, tohoto je využíváno v případě akcí, které nezávisí na typu entity – např. vykreslení. Druhým je potom využití polymorfizmu – virtuálních metod.



Obrázek 2.2: Příklad objektově orientované hierarchie.

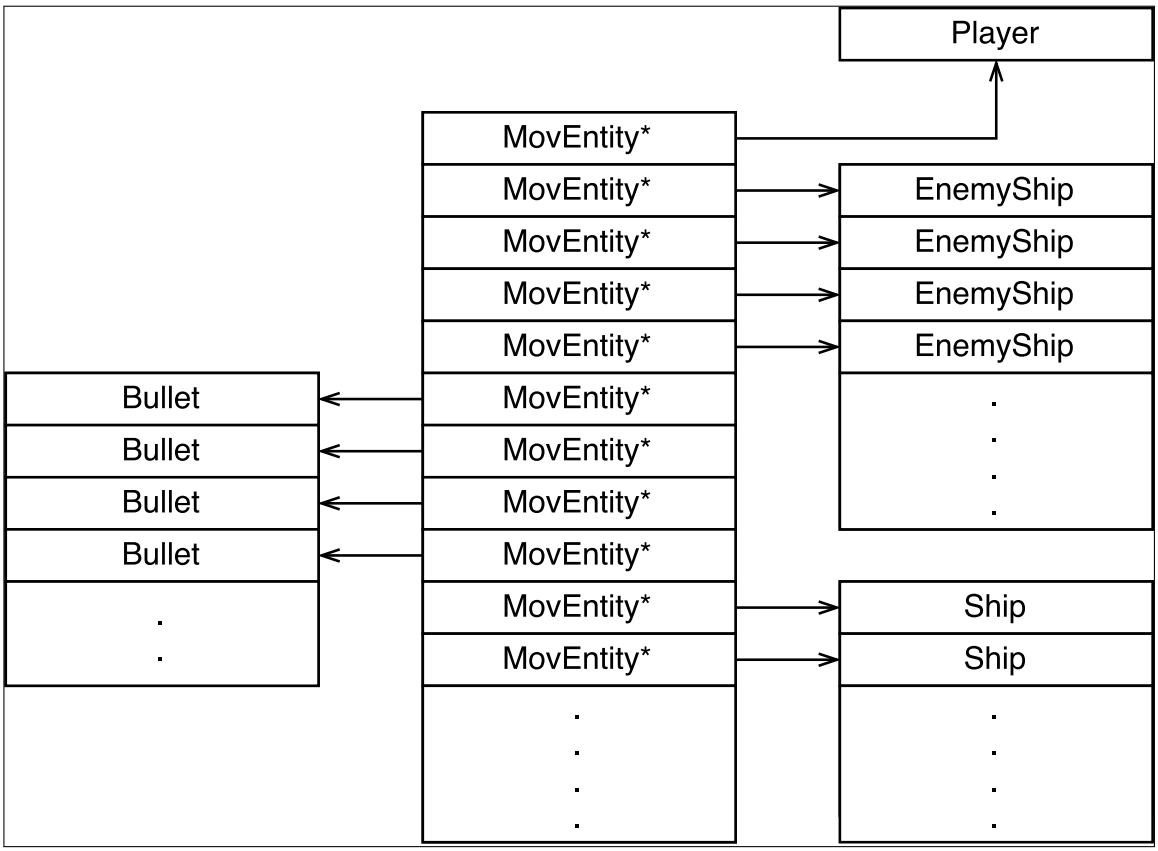
Hierarchie dědičnosti, která je v tomto příkladě použita lze vidět na obr. 2.2 – pouze ilustrační, pro předvedení problémů. Bázová třída **Entity** obsahuje kód pro vykreslování, dále se hierarchie dělí na entity pohyblivé a statické. Prvním problémem je umístění entit, které jsou v pozadí – nelze do nich narazit. Dalším příkladem problémů s *OOH* je přidání třídy **StationaryGun** (nepohyblivá zbraň), u které není optimální třída, ze které by měla dědit.

Pro příklad využití paměti je využit seznam pohybujících se entit, jejichž pozice musí být každý snímek hry aktualizována o jejich rychlost. Ilustrace možné organizace paměti lze vidět na obr. 2.3. Hlavní seznam obsahuje ukazatele na entity, které je potřeba aktualizovat. Instance jednotlivých konkrétních typů jsou uloženy ve vlastních seznamech.

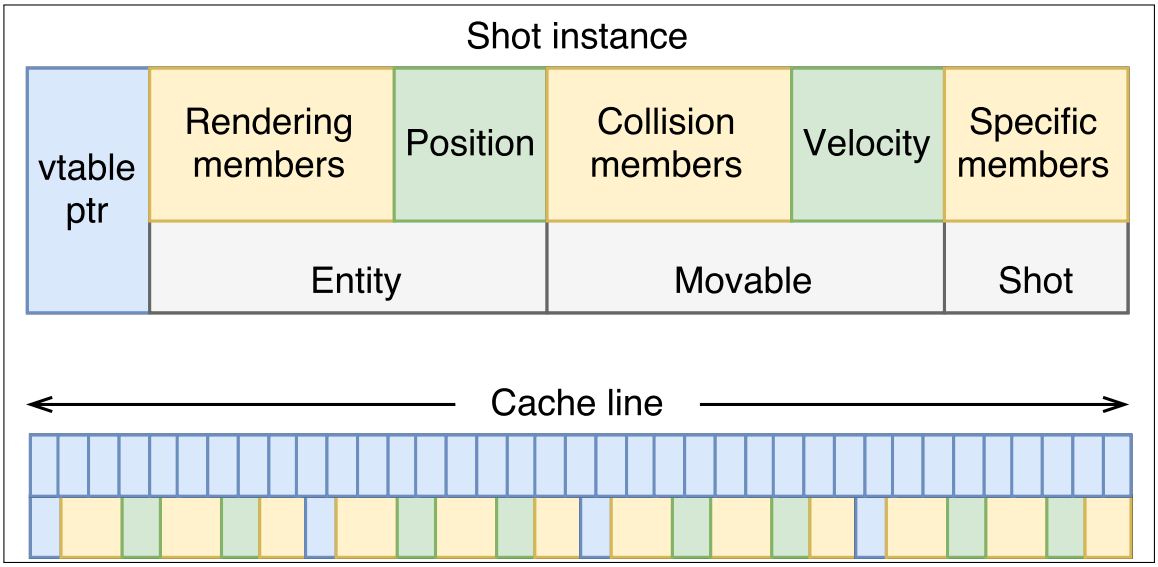
Při operaci aktualizace je iterováno přes hlavní seznam ukazatelů a na každý objekt je zavolána metoda, která aktualizuje pozici o jeho rychlost. Postup práce s hlavní pamětí a *cache* je následující [1]. Prvním krokem je načtení bloku paměti o velikosti řádku vyrovnávací paměti (*cache*), který obsahuje požadované ukazatele. Následuje dereference prvního z ukazatelů, která zapříčiní čtení dalšího řádků paměti, který již obsahuje iterované objekty. Ilustrace obsahu paměti *cache* lze vidět na obr. 2.4. Díky způsobu, kterým jsou entity skládány, obsahují jednotlivé objekty kromě požadovaných informací – pozice a rychlost – také data, která nejsou používána. Výsledkem je neoptimální využití procesorových *cache* [9].

Komunikace jednotlivých částí je při použití *OOH* implicitní – pomocí *public* a *protected* členů.

[1] Postup neuvažuje různé vrstvy paměti *cache* a předpokládá, že k požadovanému bloku paměti zatím nebylo přistoupeno. Velikost řádku vyrovnávací paměti je nastavena tak, aby nedošlo k překrytí paměti ukazatelů a paměti instancí.



Obrázek 2.3: Obsah paměti u objektově orientovaných hierarchií.



Obrázek 2.4: Využití procesorové *cache* při použití objektově orientované entitní hierarchie.

Mezi výhody *OOH* patří:

- Jednoduchá implementace.
- Podpora tříd zabudována do mnoha programovacích jazyků.
- Použití standardních návrhových metod z objektově orientovaného návrhu.
- Implicitní propojení a komunikace mezi děděnými částmi.

Její nevýhody jsou:

- Akumulace stavu a chování, které není nutně entitou vyžadováno.
- Duplikace kódu v různých větvích stromu dědičnosti.
- Zvyšující se složitost umisťování nových typů entit.
- Typy entit jsou specifikovány ve zdrojovém kódu.
- Statické typování [2], nemožnost tvorby nových typů za běhu programu.

2.5.2 Objektově orientovaná kompozice

Pod pojmem *objektově orientovaná kompozice* (*OOC* [40]) je myšlena tvorba entit z menších částí – komponent – kde komponenty obsahují data i množinu proveditelných akcí. Entita je při použití *OOC* kontejner, který obaluje seznam komponent (kompozice). Příklad návrhu množiny komponent lze vidět na obr. 2.5, kontejnerovou třídou je v tomto případě **Entity**.

Množinu akcí, které lze nad entitou provést je definována komponenty, které jsou v entitě přítomny. Každá komponenta obsahuje, kromě dat, také akce, které lze nad entitou, která danou komponentu obsahuje, provést.

Tato metoda je často používaná (*composition over inheritance*) v návrhu software a je mezikrokem od *objektově orientované hierarchie* k *datově orientované kompozici*.

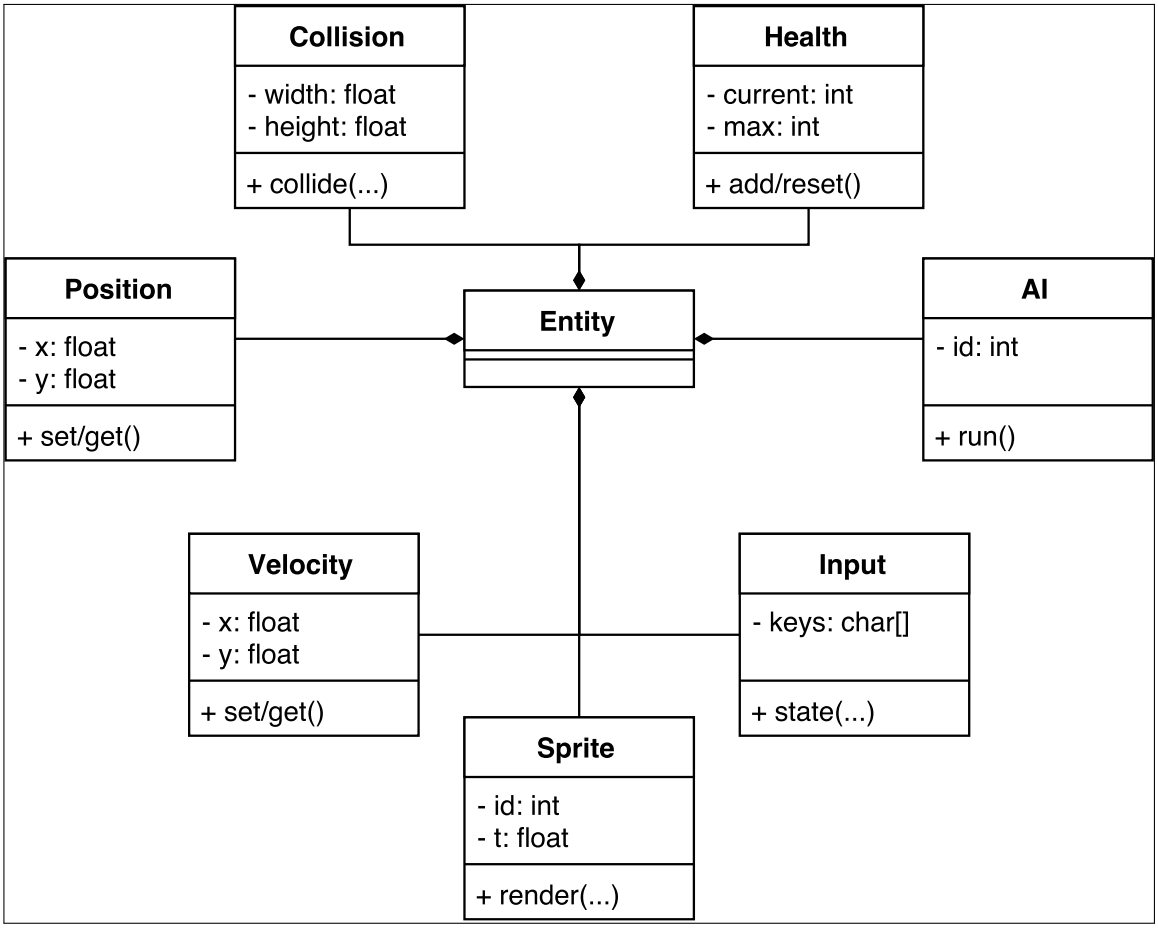
Příkladem využití těchto komponent, pro implementaci stejné množiny konkrétních entit, jako v případě použití *OOH*, lze vidět na obr. 2.6. Oproti využití dědičnosti se zde již lze jednoduše vyhnout problému s umístěním entit do stromu dědičnosti. Přidání typu **StationaryGun** je již také bezproblémové, díky možnosti libovolné kombinace jednotlivých komponent.

Existuje mnoho způsobů, jak tuto základní myšlenku kompozice z menších částí implementovat. Jednou z možností je vyhradit pro každý typ komponenty pozici v seznamu ukazatelů [3]. V tomto návrhu je entita redukována na seznam ukazatelů, kde každá komponenta je buď přítomna (ukazatel je nastavený), nebo nepřítomna (ukazatel je *NULL*). Výhodou této implementace je rychlost. Primární nevýhodou je potom neefektivní využití paměti, pro větší množství druhů komponent. Ilustraci této implementace lze vidět na obr. 2.7.

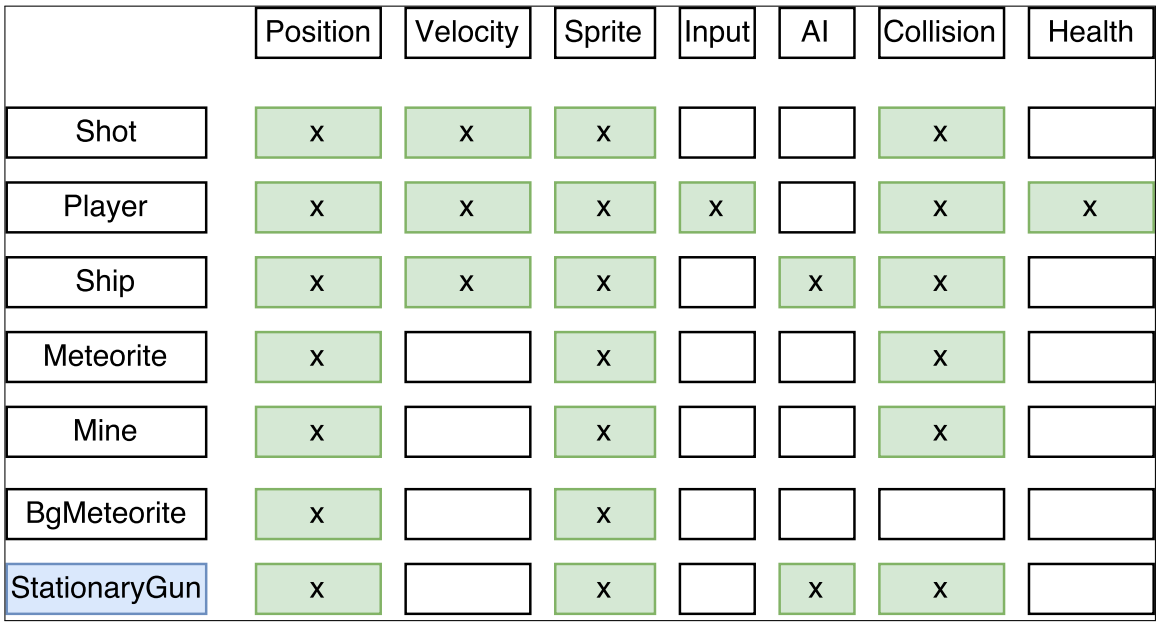
Jako příklad práce s pamětí je použita iterace nad seznamem entit, u kterých je nutno provést aktualizaci pozice, podle jejich rychlosti. V tomto příkladě je použita implementace pomocí statického seznamu ukazatelů, podle obr. 2.7.

[2] Může být výhodou v některých případech, např. optimalizace, které vykonává překladač.

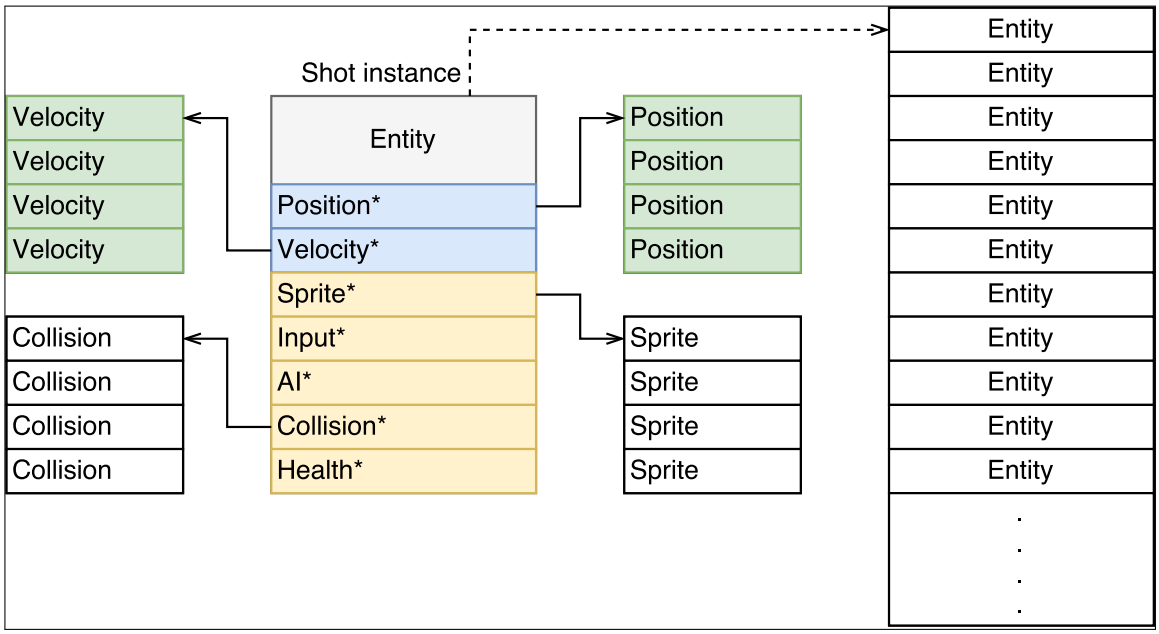
[3] Mezi další způsoby patří mapy nebo dynamické seznamy. Možným řešením je také použití obecných ukazatelů a typových proměnných.



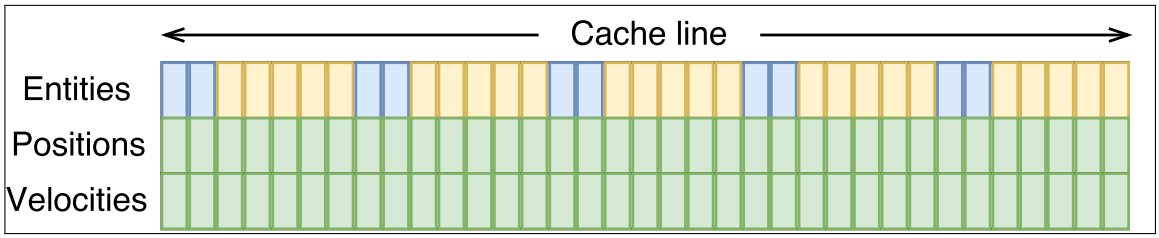
Obrázek 2.5: Příklad objektově orientované kompozice.



Obrázek 2.6: Využití kompozice pro tvorbu herních objektů. Konkrétní typy entit jsou zde reprezentovány řádky, jednotlivé komponenty jsou potom sloupce. Přítomnost komponenty je vyznačena znakem „x“.



Obrázek 2.7: Obsah paměti u objektově orientované kompozice. Zde je entita implementována pomocí seznamu ukazatelů.



Obrázek 2.8: Využití procesorové *cache* při použití objektově orientované kompozice.

Podobně, jako u použití *OOH*, je zde iterováno přes seznam, který však tentokrát obsahuje již jednotlivé instance typu **Entity** [4]. Využití vyrovnávací paměti, za stejných předpokladů, jako u příkladu v případě *OOH*, je následující. Nejdříve je načten seznam entit a jejich ukazatelů na jednotlivé komponenty. Požadovaná operace potřebuje pouze ukazatele na komponenty typu **Position** a **Velocity**, ostatní jsou v tomto případě zbytečné [5]. Následuje přístup k požadovaným komponentám skrz ukazatele v první entitě. Tímto je načten blok paměti, pro každou komponentu, do vyrovnávací paměti. Po provedení operace nad první entitou a přístupu k dalším jsou již následující komponenty přístupné z *cache*.

Komunikace mezi jednotlivými komponenty je při použití *objektově orientované kompozice* problematické. Příkladem by mohla být *umělá inteligence*, která potřebuje změnit rychlost entity. Jelikož jsou jednotlivé komponenty samostatné (*encapsulation*) objekty, oddělené od ostatních komponent, které jsou součástí stejné entity, není mezi nimi možná přímá komunikace. Jedním možným řešením je přidání systému zpráv [6], který má přístup k entitě, jako celku a předat odkaz na tento systém jednotlivým komponentám.
Mezi výhody *OOC* patří:

- Uniformní instance všech typů entit, liší se pouze v přítomných komponentách.
- Lepší využití procesorových *cache*.
- Možnost definování nových typů entit za běhu, použitím kompozice.
- Definice typů entit v datech.
- Komponenty, které entita vlastní, jsou přístupny z jednoho místa.
- Odstraněna duplikace kódu a akumulace nechtěného stavu.

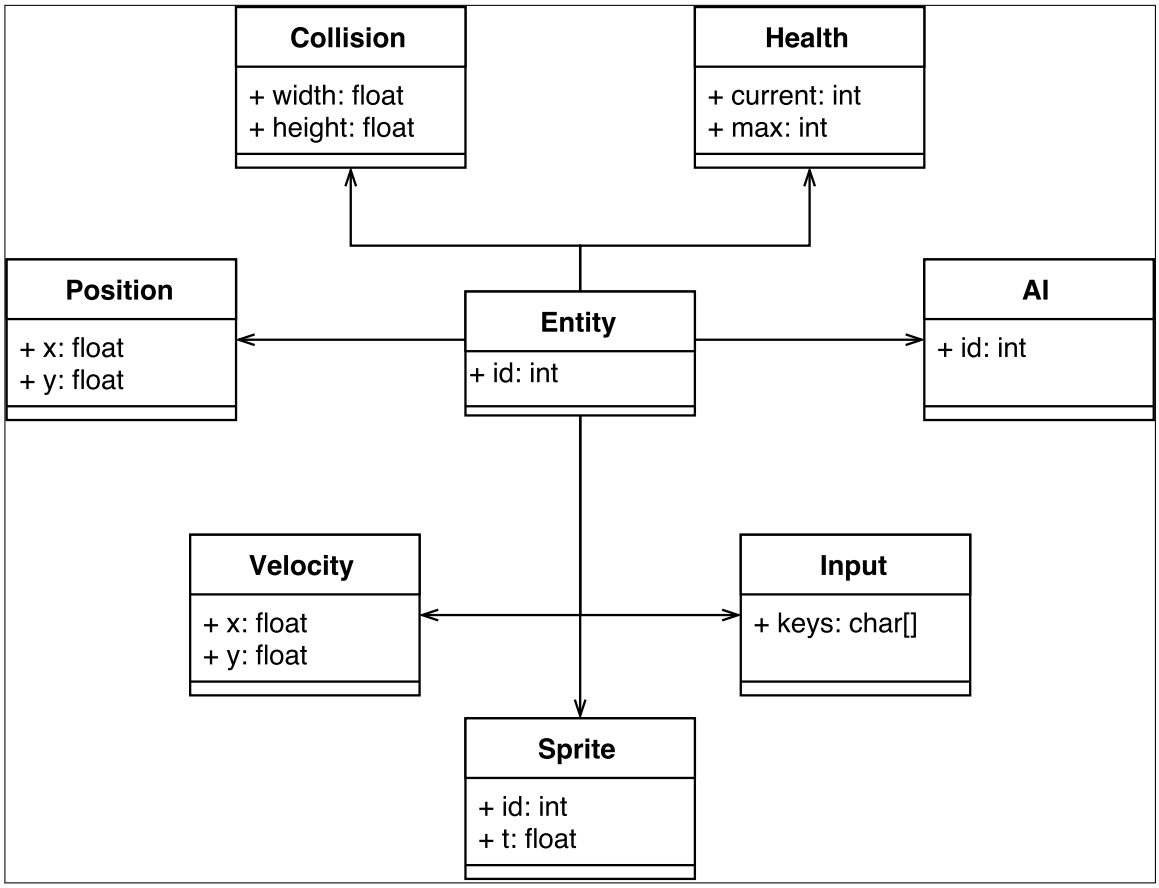
Její nevýhody jsou:

- Složitější implementace, většinou bez podpory v programovacím jazyce.
- Komunikace mezi komponentami není implicitní, je nutno ji implementovat.

2.5.3 Datově orientovaná kompozice

Datově orientovaná kompozice (*DOC*) je dalším krokem v rozdělování entit do logických celků – do komponent. Podobně, jako *objektově orientovaná kompozice*, *DOC* rozděluje

[4] Předpokládá se, že všechny entity obsahují komponenty typu **Position** a **Velocity**
[5] Záleží na implementaci, tuto neefektivitu je možné řešit optimalizacemi.
[6] Zprávy, ale i události jsou možným řešením.



Obrázek 2.9: Příklad datově orientované kompozice.

entity do menší části (komponenty), což umožňuje lepší modularitu výsledného software. Oproti *OOC* však jednotlivé komponenty neobsahují žádnou logiku [7].

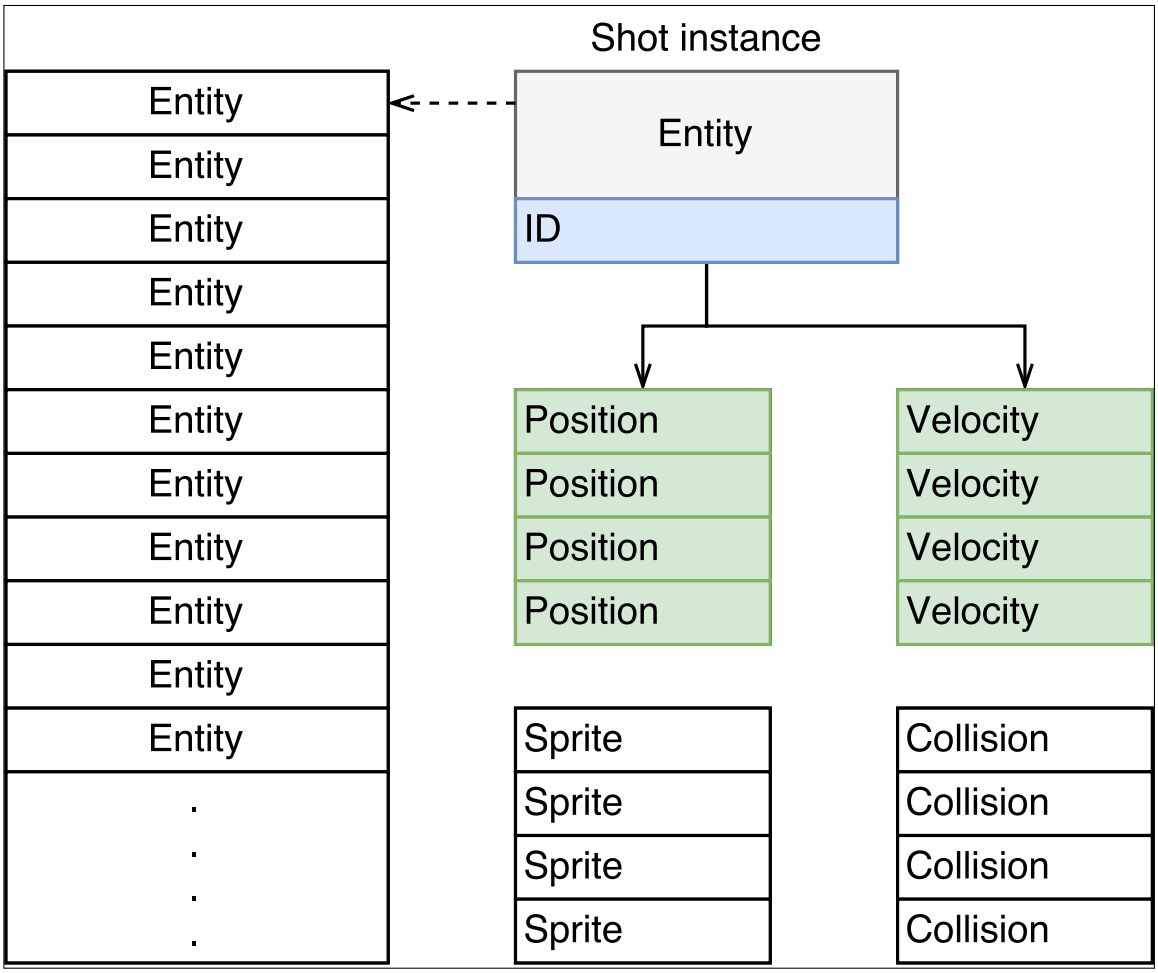
Množinu akcí, kterou lze nad entitou vykonat, je opět definována přítomnými komponentami. Jelikož komponenty samy o sobě neobsahují logiku, je nutno akce definovat v okolním kódu. Jedním způsobem je přímý přístup ke komponentám a jejich datům, tato metoda bude použita v následujícím příkladě. Pro rozsáhlejší aplikace, kde je třeba lepší kontrola nad přístupy k jednotlivým komponentám, lze například využít *systémy*, které implementují akce nad komponenty [8]. Tento způsob je využit i v návrhu *entitního systému*, o kterém je napsána tato práce.

Příklad návrhu entity a komponent lze vidět na obr. 2.9. Mezi důležité změny, oproti *OOC*, patří veřejný (*public*) přístup ke členům entit. Entita je v tomto návrhu [9] reprezentována číslem (identifikátor), pomocí kterého lze komponenty jednoznačně přiřadit k entitám, které je vlastní. Implementace konkrétních entit je konceptuálně stejná, jako při použití *objektově orientované kompozice*, příklad lze vidět na obr. 2.6.

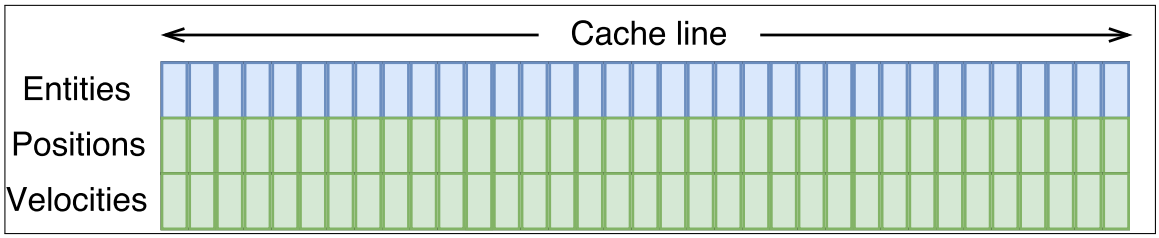
[7] Logikou je v tomto případě myšleno kód, který pracuje na vyšší úrovni, než pouze s daty dané komponenty. Blíže k tomuto v 2.6

[8] Tato metoda je blíže popsána v sekci 2.6

[9] Toto je pouze jeden způsob implementace *datově orientované kompozice*.



Obrázek 2.10: Obsah paměti u datově orientované kompozice.



Obrázek 2.11: Využití procesorové *cache* při použití datově orientované kompozice.

Příklad organizace paměti, při použití *DOC*, lze vidět na obr. 2.10. Entity jsou uchovávány v homogenním seznamu. Jednotlivé identifikátory entit je možné mapovat na komponenty přímou indexací polí komponent.

Pro ukázku práce s vyrovnávací paměti je opět použit příklad iterace nad seznamem entit, které je nutno posunout v prostoru, podle jejich rychlosti. Pro vykonání této akce je tedy nutný přístup ke komponentám typu **Position** a **Velocity**. Nejdříve je opět přistoupeno k poli entit, čímž je načten příslušný řádek do vyrovnávací paměti. Jelikož je mapování identifikátorů entit realizováno pomocí přímé indexace [10] je možné po načtení řádku *cache* pro dané dva typy komponent již následující přístupy do paměti realizovat přímo z vyrovnávací paměti. Obrázek 2.11 obsahuje ilustraci stavu procesorové *cache*.

Komunikace mezi jednotlivými částmi je v případě *datově orientované kompozice* rozdělená do dvou úrovní. První z nich, komunikace mezi komponentami, je vyřešena implicitně, jelikož akce mohou přistupovat k několika komponentám. Druhou je potom komunikace mezi zprostředkovateli akcí (např. *systémy*). Tento typ komunikace je možné opět řešit pomocí systému zpráv, nebo událostí.
Mezi výhody *DOC* patří:

- Efektivní skladování entit, které jsou reprezentovány identifikátorem.
- Lepší využití procesorových *cache*.
- Možnost definování nových typů entit za běhu, použitím kompozice.
- Definice typů entit v datech.
- Odstraněna duplikace kódu a akumulace nechtěného stavu.
- Možnost efektivní implementace akcí, mimo komponenty.
- Možná vysoká úroveň optimalizace celého systému.

Její nevýhody jsou:

- Bez podpory v programovacích jazycích [11].
- Náročná implementace.

[10] Za předpokladu, že identifikátory všech iterovaných entit nedovolují díry v poli komponent.
[11] Vzniká programovací jazyk, který bude tento typ kompozice podporovat [28].

Mezi další vlastnosti patří:

- Ztráta abstrakce při práci s komponentami.
- Možnosti *serializace* entit a komponent.
- Akce mohou přistupovat k několika komponentám.

2.6 Entity-Component-System

Entitní systémy, postavené za použití *datově orientované kompozice*, s využitím *systémů* pro generování chování, se nazývají *Entity-Component-System* [12]. Tato část obsahuje popis obecných konceptů tohoto typu systémů, a vlastností, které z nich vyplývají.

2.6.1 Motivace a koncepty

Entity-Component-System (*ECS* [21] [20]) je programovací paradigma [34], založené na kompozici. Základní myšlenkou je separace logiky (*systémy*) a dat (*entity*, *komponenty*), čímž je umožněna jemnější modularita, celý entitní systém lze přirovnat k *relační databázi*. Tato metoda vychází z *datově orientované kompozice*, která je podrobněji popsána v části 2.5.3.

Entita je základním stavebním blokem [34], který lze přirovnat k objektu z *objektově orientovaného* návrhu. Oproti objektům nejsou však entity tvořeny z předem daného vzoru (např. třídy). Jejich chování a data, definují komponenty, které jsou k dané entitě přiřazeny. Entity lze reprezentovat pomocí jednoznačných identifikátorů (čísel), které mají podobnou funkci, jako *primární klíče* v databázi, ilustrace tohoto přirovnání lze vidět na obr. 2.12.

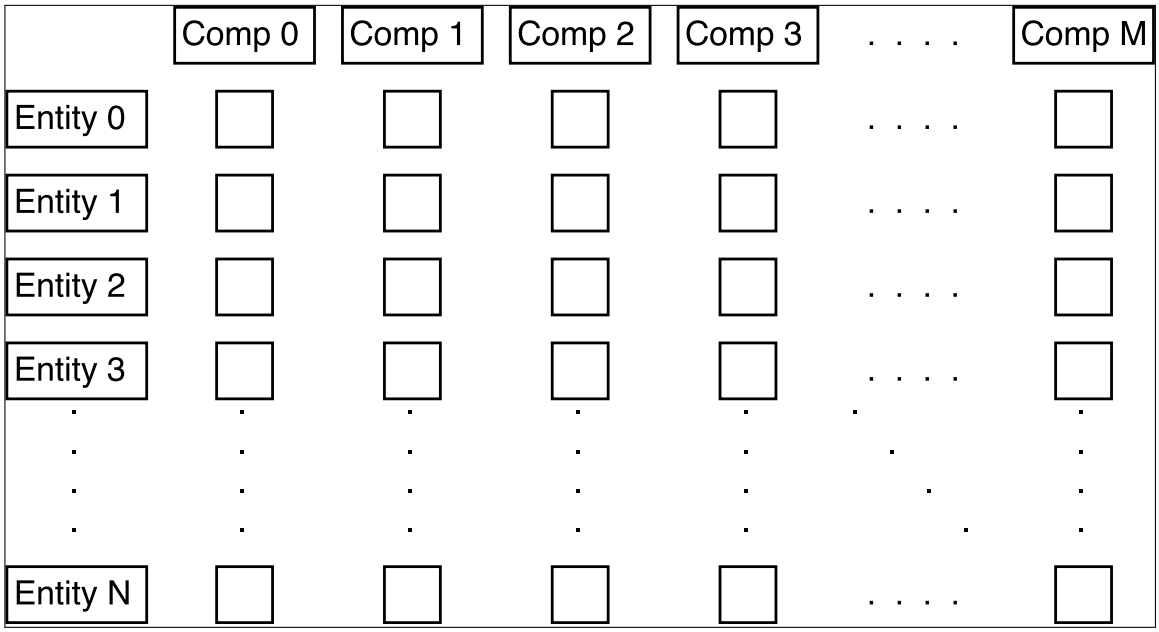
Komponenty jsou základními nosiči dat [13] v *ECS*. Po přiřazení komponent k entitám však mají komponenty i důležitější funkci – určují chování entit. Kromě explicitních dat obsažených v komponentách existuje také, implicitně, informace o tom, zda určitá entita obsahuje (má přiřazen) daný typ komponenty. Často používaným názvem pro komponenty, je také „aspekt“ [34]. Každá entita má k sobě přiřazeno $0 - 1$ komponent daného typu, kde přítomnost komponenty znamená, že entita splňuje daný aspekt. V přirovnání *ECS* k *relační databázi*, lze o komponentách smýšlet jako o sloupcích (obr. 2.12) tabulky entit.

Pokud bychom pokračovali s přirovnáním k *OOP* – entity jsou jednotlivé objekty, komponenty umožňují polymorfizmus – potom systémy implementují chování, nebo metody jednotlivých komponent. Pro mnoho různých typů dat, kde každý typ vyžaduje vlastní chování, je *OOP* výhodné, protože je možné data i operace uložit do jednoho nerozdělitelného celku. Pokud je však nutné vykonat stejnou akci na velkém množství různých typů, *OOP* není optimálním řešením a *ECS* je v tomto případě vhodnější [34].

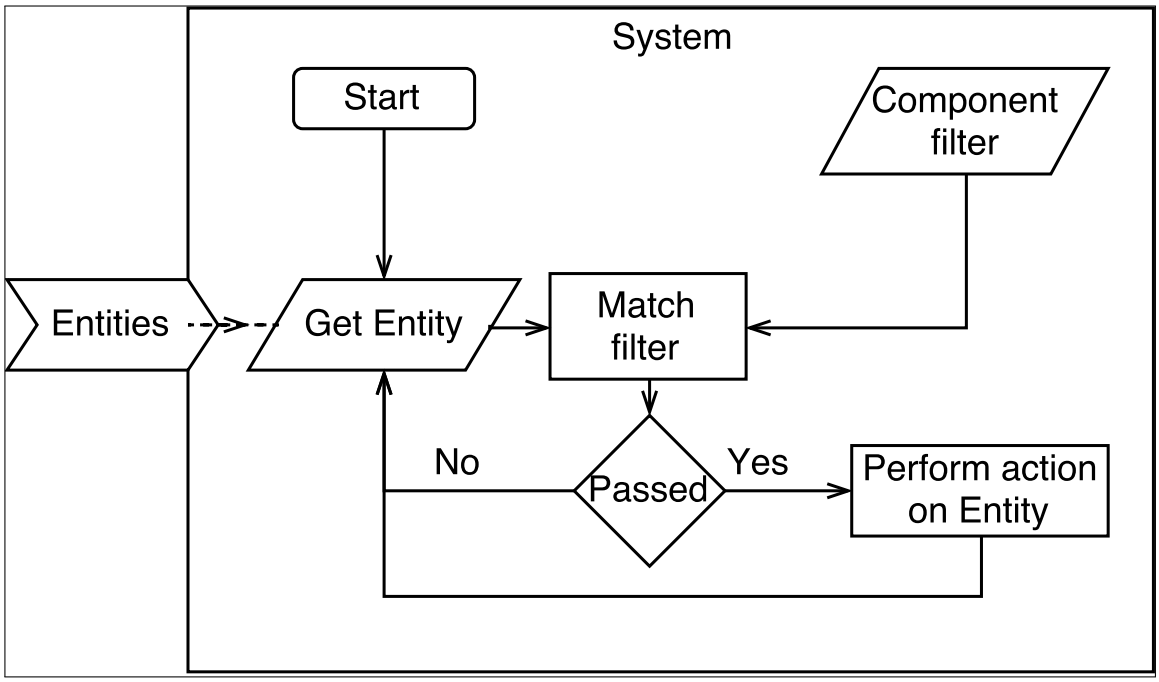
Každý systém provádí danou operaci na všech entitách, které obsahují požadované komponenty (*aspekty*). Základní schéma systému lze vidět na obr. 2.13. Výběr vhodných entity je proveden pomocí filtru, který operuje nad přítomností jednotlivých komponent (obr. 2.14).

[12] Často používaným označením je také „component base entity system“, „Component-Entity-System“, nebo i „Entity System“.

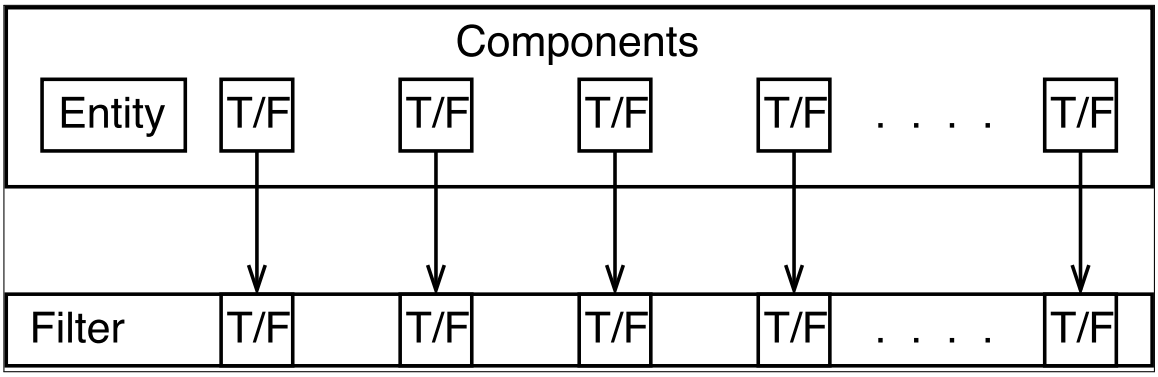
[13] Některé typy komponent – např. značky – nemusí nutně ani data obsahovat.



Obrázek 2.12: Entita reprezentována jako řádek v databázi. Primárním klíčem je identifikátor entity.



Obrázek 2.13: Vstupem *systému* jsou entity, ze kterých si vybere pouze ty, které mají odpovídající komponenty. Na vybraných entitách provádí požadované akce.



Obrázek 2.14: Filtr operuje nad entitou a informací o přítomnosti komponent.

2.6.2 Vlastnosti

Důvody, proč se *ECS* často používá při návrhu moderních her je mnoho a většina z nich vzniká díky striktnímu oddělení logiky a dat. Komponenty je možné skladovat v sekvenčních blocích paměti, toto zlepšuje efektivitu využití procesorových *cache*. Lokalita dat se vztahuje i na instrukce [32] [14] a díky opakovanému provádění stejných operací nad velkým počtem entit dochází k vyšší efektivitě využití *instrukční cache*.

Inherentní modularita *ECS* dovoluje lepší dělení práce ve velkých týmech vývojářů. Každý systém musí předem specifikovat, o jaké entity (které aspekty musí splňovat) má zájem, tímto je zamezeno nechtěnému ovlivňování okolního kódu. Další výhodou, z pohledu organizace kódu, je přímočará *refaktorizace* systémů a komponent.

Jelikož komponenty obsahují pouze čistá data, bez propojení do externích modulů, je možné přistupovat k entitnímu systému také z vestavěného skriptovacího jazyka (např. *LUA*). Skriptovací jazyky bývají hojně využívány jako součást „gameplay“ [15] funkcí, nebo *umělé inteligence*. Pokročilou vlastností *ECS* může také být přidávání nových typů komponent za běhu aplikace. Tyto, ale i další (např. *serializace* komponent), funkce umožňují rychlé prototypování nových herních mechanik.

Při správném rozvržení systémů a jejich požadovaných aspektů, je možné zaručit, že k ostatním komponentám nebude přistupováno. Toto umožňuje dosáhnout vyšší úrovně paralelizace celého systému, i přestože několik systémů může v jeden čas přistupovat ke stejné entitě. Paralelizace je dále podporována opakovaným prováděním stejných akcí nad velkým množstvím entit, které lze rozdělit na různá vlákna a zpracovat je odděleně.

2.7 Existující řešení

Tato část obsahuje porovnání existujících knihoven, které umožňují práci s entitními systémy založenými na *ECS* paradigmatu. Kromě základního popisu jsou zde také uvedeny výhody a nevýhody daných přístupů. Mezi další známé knihovny pro tvorbu her, které používají komponentní přístup, patří *Unity*, *Unreal Engine*, nebo např. *libGDX* (knihovna *Ashley* [44])

[14] *Instrukční cache* [46]
[15] Kód, který se stará o interakce virtuálních objektů ve hře.

2.7.1 EntityX

EntityX [24] je implementace entitního systému v programovacím jazyce *C++*, která silně využívá vlastností standardu *C++11* a *šablonového metaprogramování* [29]. Toto umožňuje příjemnější práci s typy, pro uživatele této knihovny. Entity jsou v tomto případě nepřímo mapovány (pomocí identifikátorů) na komponenty, které jsou uloženy v souvislých polích, což umožňuje efektivní využití vyrovnávací paměti.

Kromě základní implementace entitního systému, podle *ECS*, knihovna také umožňuje uživatelům knihovny reagovat, když entity „vcházejí“, nebo „vycházejí“ do *systémů*, čímž dovoluje entitní systém propojit s okolním kódem.

Mezi hlavní nevýhody patří delší doba překladu výsledné aplikace – výsledek použití šablon. Vlastnost, která je na jednu stranu výhodou, ale má i jisté nepříjemné stránky, je nemožnost registrace nových typů komponent za běhu aplikace. Výhodou je v tomto případě možnost vyšší optimalizace výsledné přeložené aplikace.

2.7.2 Artemis

Artemis [41] je původně knihovna implementovaná pro jazyk *Java*, ale existuje i mnoho implementací pro další jazyky, např. *C++* [37]. Knihovna využívá *generického programování* jazyka *Java*, čímž opět umožňuje uživateli příjemnější práci. Oproti ostatním knihovnám podporuje vyšší granularitu filtrování entit, kdy je možné specifikovat, které komponenty entita musí, může a nesmí obsahovat. Další zajímavou vlastností je speciální typ komponent – značky – které neobsahují žádná data, ale umožňují entity dělit do skupin.

Knihovna také umožňuje využití pokročilých funkcí, jako sdružování entit a jejich tvorba ve vyšších počtech, nebo entitní *archetypy*, které fungují jako předloha pro další entity.

Knihovna **Artemis** je ze všech představených knihoven nejvíce dokončená a existují již i projekty, které ji používají.

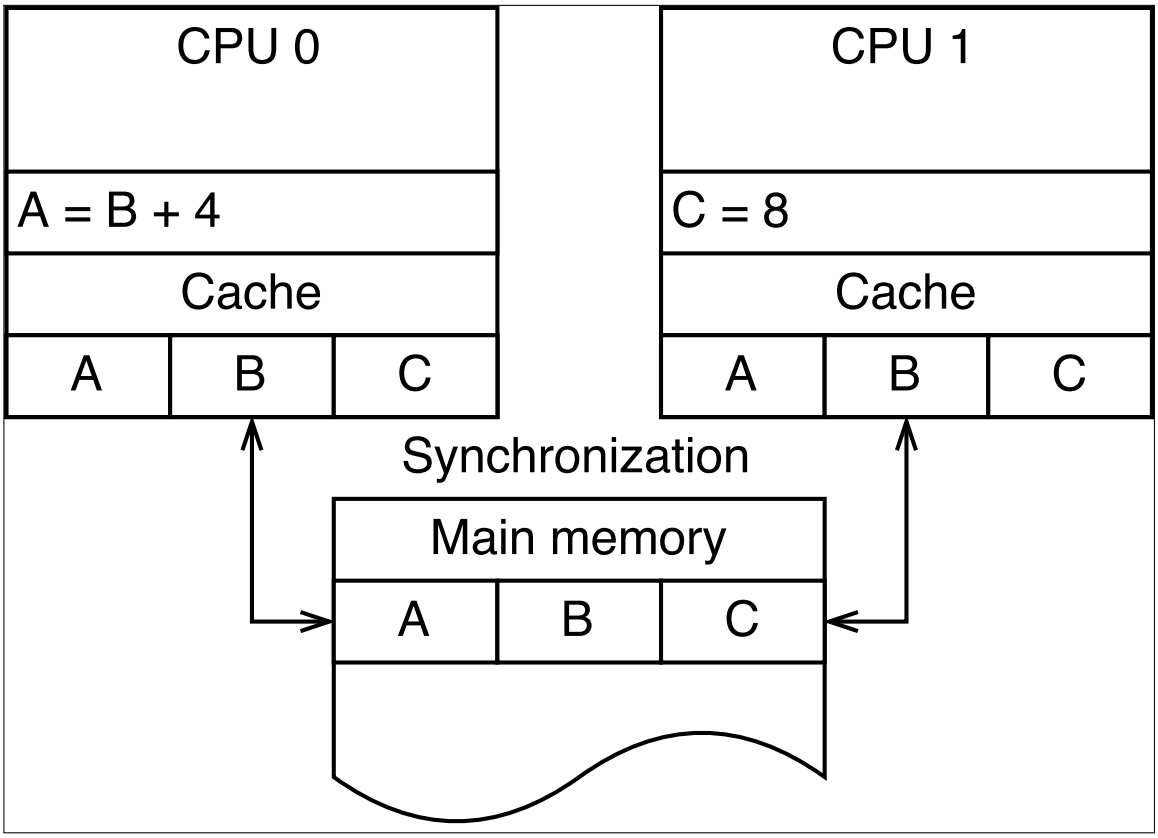
2.7.3 Anax

Kromě základní implementaci vlastností *ECS*, obsahuje knihovna **Anax** [36] také pojem *skupin*. *Skupiny* obsahují před generovaný seznam entit, skrz který *systémy* iterují. Výhodou tohoto přístupu je možnost rychlé práce s celým seznamem entit, které vyhovují určitému filtru. Mezi nevýhody patří nutnost tyto seznamy udržovat. Podobně, jako **EntityX**, je tato knihovna implementována v jazyce *C++*.

2.8 Paralelizmus

Vývoj moderních procesorů se ubírá směrem zvyšování počtu výpočetních jader [18] a proto i moderní aplikace musí být schopné tato jádra využít. To je zvláště pravda pro vývoj her, které se snaží s hardware vytěžit maximální výkon. Paralelizace aplikací, jejichž části jsou úzce provázané je obzvláště problematické a proto vznikají nové způsoby tento typ software navrhovat – jedním z nich je právě *ECS*.

Paralelizmus lze obecně rozdělit na 2 typy – datový a úkolový [10]. Při použití datového paralelizmu je prováděna jedna akce, na různých datech (obdoba *SIMD* u procesorů). Opačným přístupem je paralelizmus úkolový [12], u kterého mohou běžet různé operace nad stejnými daty (*MISD*). Ve většině případů je však používáno kombinací těchto dvou přístupů. Speciálním případem je vyhrazení celého vlákna jednomu modulu – např. manažer zvuku – který vyžaduje nízké odezvy [38].



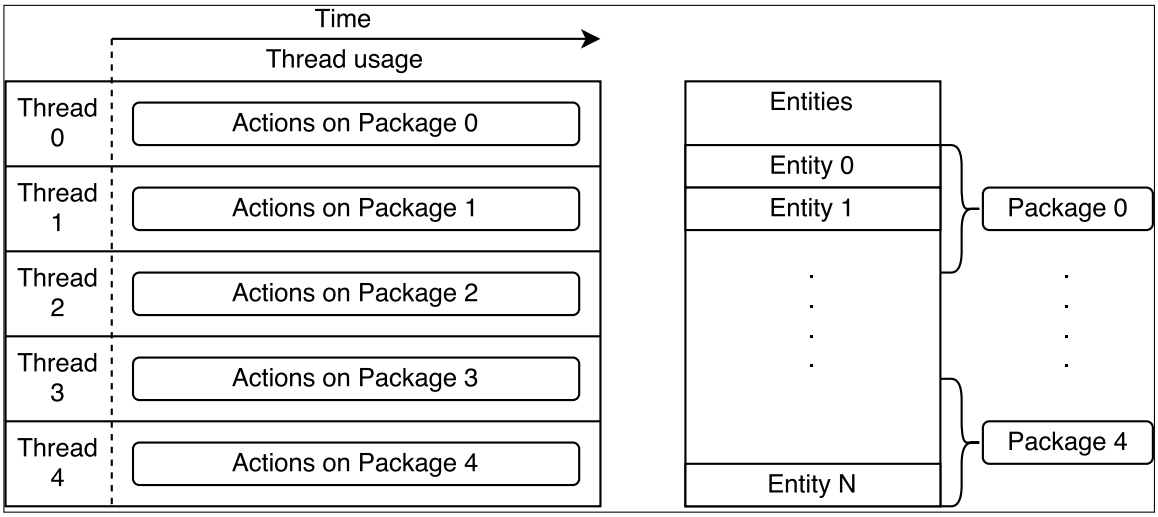
Obrázek 2.15: Udržování koherence vyrovnávací paměti a „false sharing“ [2]. Synchronizace *cache* je prováděna mezi jednotlivými jádry, přes specializovanou sběrnici.

Výhodou paralelizmu je možnost rozložení práce na několik výpočetních jednotek, což umožňuje požadovanou akci provést rychleji. Využití paralelizmu však představuje mnoho nových typů problémů, které se u sekvenčního programování nevyskytují. Kromě obecně známých překážek, jako jsou synchronizace vláken, potenciální uváznutí (*deadlock*), různých typů souběhu (*race condition*), existují také problémy na hardwarové úrovni. Hlavním z nich je udržování koherence procesorových *cache* [5] a skrz to problém, který se nazývá „false sharing“ [2]. V případě, že několik vláken pracuje se stejným blokem paměti – je načtený v jeho lokální vyrovnávací paměti – je nutné *cache* jednotlivých jader synchronizovat tak, aby každé z nich nepracovalo s jinými daty. Tento proces se nazývá „udržování koherence vyrovnávacích pamětí“. Problém, který v tomto systému může nastat – „false sharing“ – vzniká v případě, kdy obě vlákna pracují nad stejnou pamětí [16], ale nepracují se stejnými daty. Při každé změně bude nutno provést synchronizaci, i když není nutná, ilustraci tohoto jevu lze vidět na obr. 2.15.

Entity-Component-System paradigma je výhodné z pohledu více-jádrových systémů, díky možnosti téměř [35] dokonalé paralelizace na několika úrovních [25]. První úrovní paralelizace, kterou *ECS* umožňuje, je datový paralelizmus, kde se uvnitř systému množina všech entit, které odpovídají požadavkům, rozdělí do množiny vláken, které mohou dané akce provádět odděleně [17]. Ilustrace principu, jak tento způsob pracuje lze vidět na obr.

[16] Paměť je součástí stejného řádku vyrovnávací paměti.

[17] Některé typy akcí tento paralelizmus neumožňují - např. výpočet pozic objektů v grafu scény.

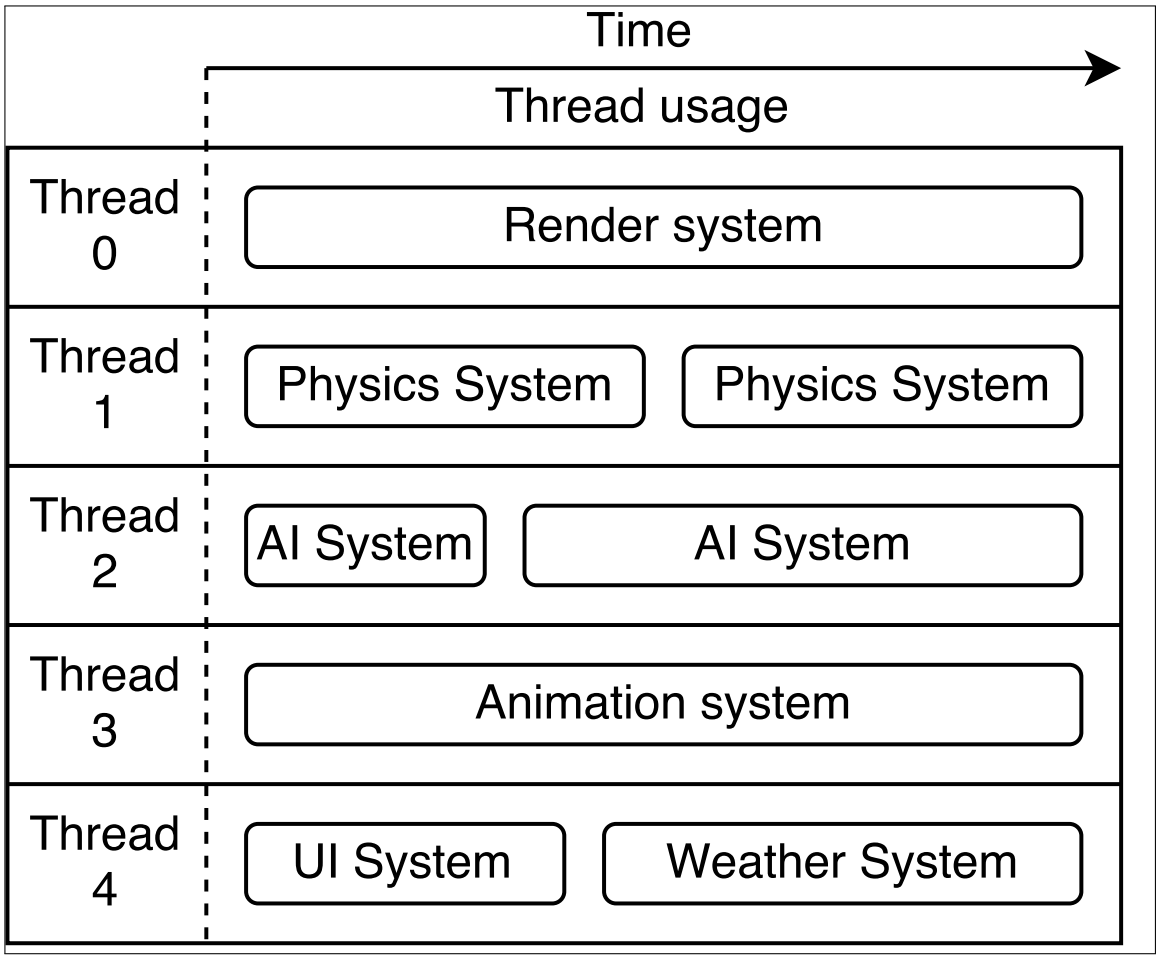


Obrázek 2.16: Paralelizmus uvnitř systémů je příkladem datového paralelizmu [10].

2.16. Dalším způsobem paralelizace je možnost spouštět jednotlivé systémy na oddělených vláknech (obr. 2.17), tato metoda je ovšem použitelná pouze pro systémy, které nezapisují do komponent, které jsou zároveň používány v jiném systému.

Jednoduchý systém může pracovat způsobem transformační funkce, jejíž vstupní parametry jsou komponenty, které systém požaduje a výstupem je zápis (změna) jedné z těchto komponent. Jelikož je ovšem vhodné, při paralelním přístupu k aktuálnímu stavu herního světa, aby všechny systémy dokázaly přečíst původní data beze změn, lze funkci systémů změnit. Systémy mohou místo zápisu nových hodnot do stejných komponent (vstupů) zapsat výsledek do „následujícího stavu“ [38] [18]. Při generování nového stavu je nutné určit okamžik, kde se následující stav stane stavem aktuálním, čímž se celý herní svět „posune“.

[18] Tento způsob je často používán ve funkcionálních jazycích.



Obrázek 2.17: Paralelizmus na úrovni systémů, kdy několik systémů běží zároveň.

Kapitola 3

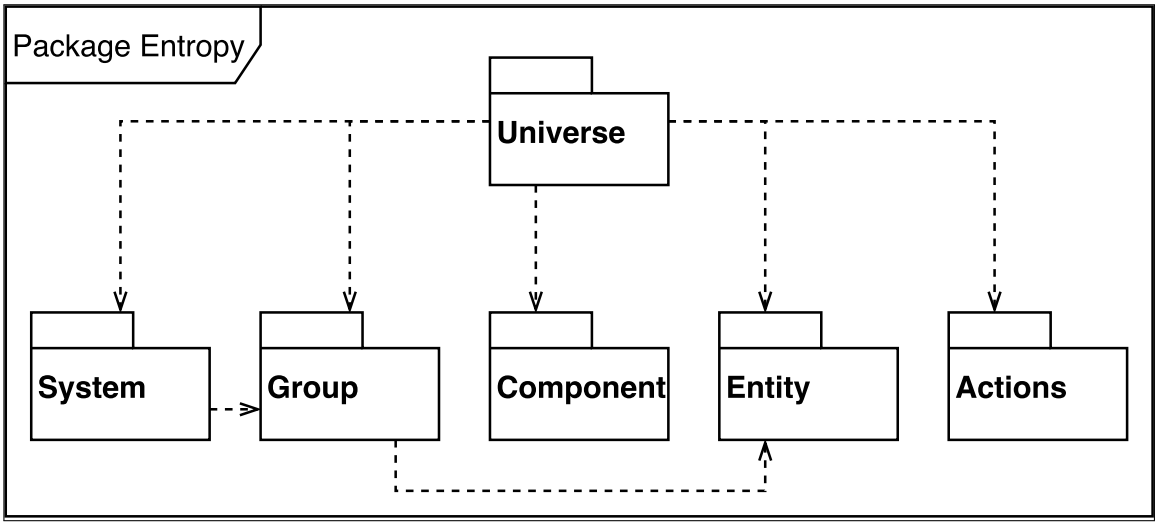
Návrh entitního systému

Obsahem této kapitoly je popis návrhu entitního systému založeného na *ECS* paradigmatu, které bylo představeno v předchozí kapitole. Nejdříve je prezentováno základní rozdělení systému na jednotlivé části a komunikace mezi nimi. Následuje popis návrhu podsystémů – *komponenty*, *systémy* a *entity*. Každá část obsahuje představení návrhu a jeho zdůvodnění. Následuje návrh paralelního přístupu k entitnímu systému a jeho tok řízení.

Při návrhu entitního systému byly použity poznatky z teoretické části této práce a *open-source* implementace entitních systémů založených na *ECS* paradigmatu – *EntityX* [24], *ArtemisCpp* [37], *Anax* [36], *Artemis* [41] a *Ashley* [44].

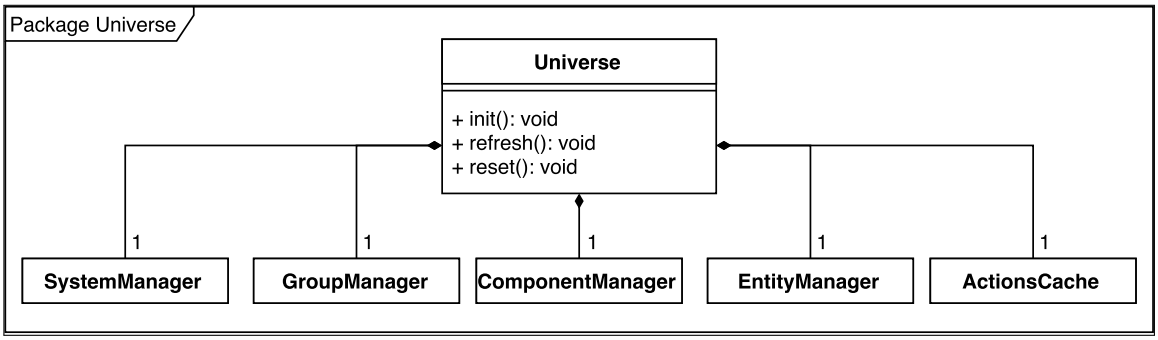
3.1 Přehled a komunikace

Entitní systém je, podobně jako *ECS* paradigma, rozdělený do několika částí – modulů – kde cílem jednotlivých částí je správa některé z domén *ECS*. Diagram reprezentující toto rozdělení lze vidět na obr. 3.1. Mezi tyto moduly patří – správa systémů a skupin, správa komponent, správa entit a správa akcí. Bližší popis a návrh jednotlivých částí obsahují následující části této kapitoly.



Obrázek 3.1: Rozdělení entitního systému do modulů.

Funkcionalita celého systému je zastřešena třídou **Universe** (obr. 3.2), skrz kterou je uživateli umožněn přístup k jednotlivým podsystémům. Pro každou operaci nad entitním systémem existuje metoda uvnitř třídy **Universe** – tento návrh umožňuje jednotný vstupní bod. Nevýhodou je vysoké množství metod v této třídě. Další důležitou funkcionalitou třídy **Universe** je zprostředkování komunikace mezi jednotlivými moduly – např. přidání komponenty má dvě části, skutečná operace nad nosičem komponent a úpravu metadat (obr. 3.4 a obr. 3.7).



Obrázek 3.2: Třída **Universe** je rozhraním entitního systému. Diagram neobsahuje specifikaci všech metod třídy, kvůli jejich vysokému počtu.

3.2 Komponenty a jejich nosiče

Komponenty jsou definovány jako základní datové jednotky v *ECS*, kde každá entita má k sobě přiřazeno 0-1 komponent daného typu. Komponenty, jak s nimi pracuje tento návrh, mohou být jakékoliv třídy [1] a nemusí obsahovat žádné specifické akce. Každá komponenta má přiřazen unikátní identifikátor – **CompId** – tento identifikátor je neměnný po dobu běhu entitního systému. Komponenty by měly být pasivní datové struktury [2], se kterými lze pracovat jako s čistou pamětí (přesuny, kopírování, atp.).

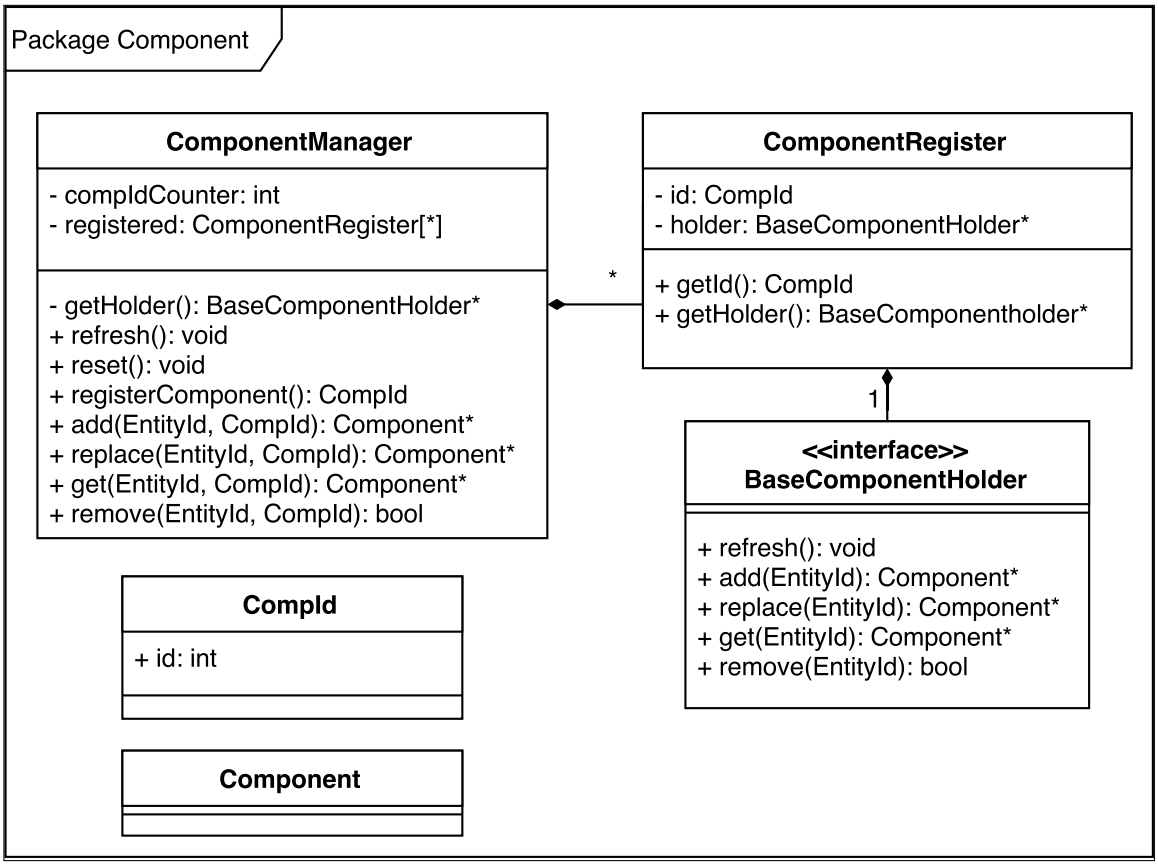
O správu komponent se stará podsystém správy komponent, jehož diagram lze vidět na obr. 3.3. Modul se skládá dvou částí – registrace typů komponent a nosiče komponent. Veškerá komunikace probíhá skrz třídu **Component Manager**, která obsahuje metody pro registrace komponent a jejich následná asociace k entitám.

V první fázi práce s tímto modulem je třeba registrovat typy komponent, které budou následně používány. Nové komponenty jsou registrovány pomocí metody **registerComponent** [3], která vytvoří mapování z daného typu na jeho registr (**ComponentRegister**). Tohoto registru je následně využito pro získání specifického nosiče komponent.

[1] Obecný typ komponenty je **Component**. Pokud to implementační jazyk vyžaduje, je možné požít abstraktní třídu stejného jména, ze které budou všechny konkrétní komponenty dědit.

[2] *Plain old data structure* (*POD*) – struktura reprezentovaná kolekcí hodnot.

[3] Této metodě je třeba předat typ komponenty, která má být registrována, čehož lze docílit pomocí např. *template* (C++), nebo *generics* (Java)

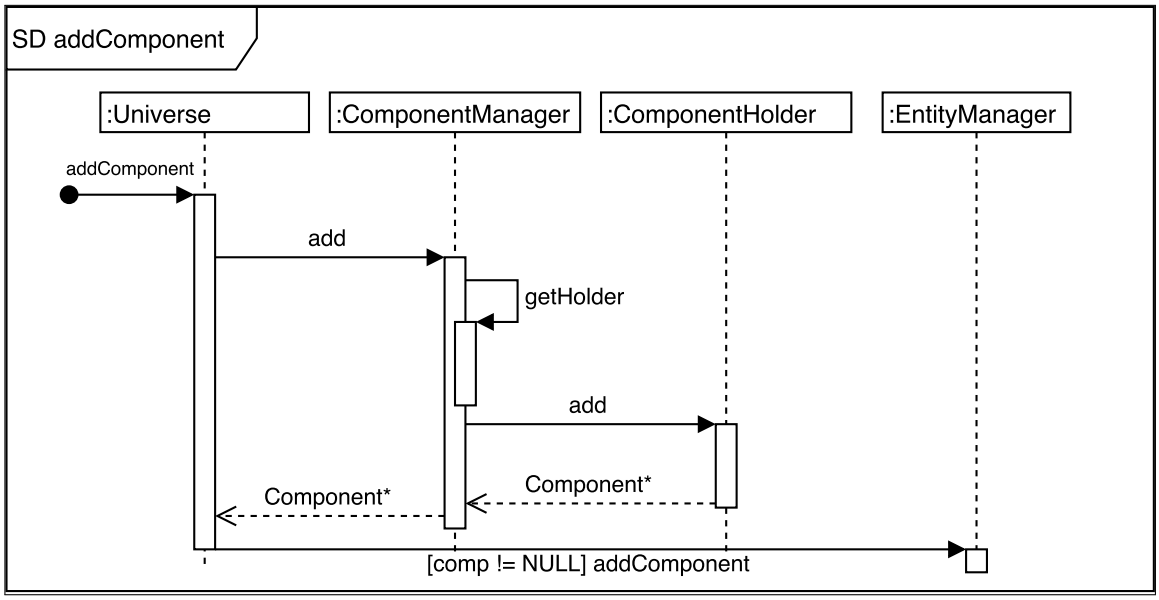


Obrázek 3.3: Diagram tříd podsystému správy *komponent*.

Každý typ komponent je uložen ve vlastním kontejneru – tzv. nosič komponent. Nosičem je třída, která dědí z **BaseComponentHolder** a implementuje všechny požadované operace, přičemž si může každá komponenta zvolit typ nosiče [4]. Hlavní funkcí nosiče je mapování identifikátorů entit (**EntityId**) na přiřazené komponenty.

Různé implementace nosičů, specifické pro určené komponenty, umožňují vyšší úroveň optimalizace pro předpokládané využití komponent. Pokud je např. jisté, že téměř všechny entity budou obsahovat určitý typ komponenty, je možné implementovat nosič jako jednoduché pole, kde mapovací funkcí je prostá indexace komponent pomocí identifikátoru entit. Naopak, pokud je předpokladem, že komponenta bude využívána nízkým počtem entit, nosič může být implementován jako strom, kde klíčem je identifikátor entity. Speciálním typem komponenty je *značka* (*tag*), která neobsahuje žádná data, ale vzniká implicitní informace o tom, zda má entita danou značku přiřazenu, čehož lze využít pro např. rozdělení entit do skupin.

[4] Opět pomocí mechanizmu *template*, nebo obdobných vlastností jiných jazyků.



Obrázek 3.4: Sekvenční diagram přidání komponenty, pokračuje na obr. 3.7.

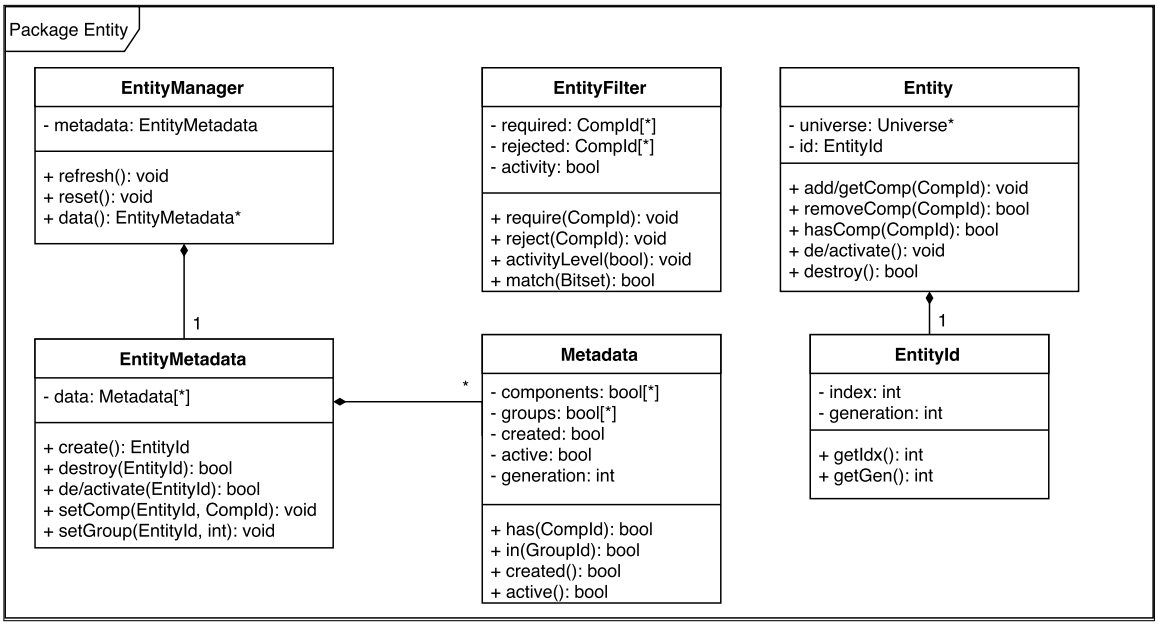
Operace přidání komponenty je rozdělena do dvou částí, první z nich, která pracuje nad **ComponentManager** lze vidět na obr. 3.4. Po úspěšném přidání komponenty následuje modifikace metadat, kterými se zabývá následující část kapitoly o návrhu.

3.3 Reprezentace entit

Entita, jako koncept z *ECS*, je distribuovaný objekt, složený z komponent. Pro každou entitu existuje 1-0..1 mapování na každý typ komponent. Vyvstává tedy požadavek ne jednoznačný identifikátor entit, který by plnil funkci *primárního klíče* v tabulce (obr. 2.12) entit.

Pro účely tohoto entitního systému je identifikátor složen ze dvou čísel – ID a generace. ID je v tomto případě skutečný *primární klíč*, pomocí kterého jsou jednotlivé řádky v tabulce entit indexované [5]. Generační číslo odlišuje různé generace entit, které zabíraly stejný řádek v tabulce (stejné ID) a je inkrementováno při každém smazání entity. Generační čísla identifikátorů, skrz které je přistupováno k systému jsou porovnány s aktuální generací dané entity, čímž je zamezen přístup ke smazaným entitám.

[5] Používá se tedy i při mapování komponent na entity uvnitř nosičů komponent.



Obrázek 3.5: Diagram tříd podsystému správy *entit*.

Podsystém správy entit, jehož diagram tříd lze vidět na obr. 3.5, lze přirovnat k relační databázi. Systém obsahuje jednu tabulku entit, kde každý řádek reprezentuje slot pro entitu. Identifikátor entity je složen z indexu řádku v tabulce a jeho generační hodnoty. Každý řádek obsahuje aktuální číslo generace a množinu *bool* hodnot – metadat. Ilustraci možné konfigurace tabulky metadat lze vidět na obr. 3.6. Výhodou této reprezentace je konstantní složitost přístupu k metadatům dané entity [6].

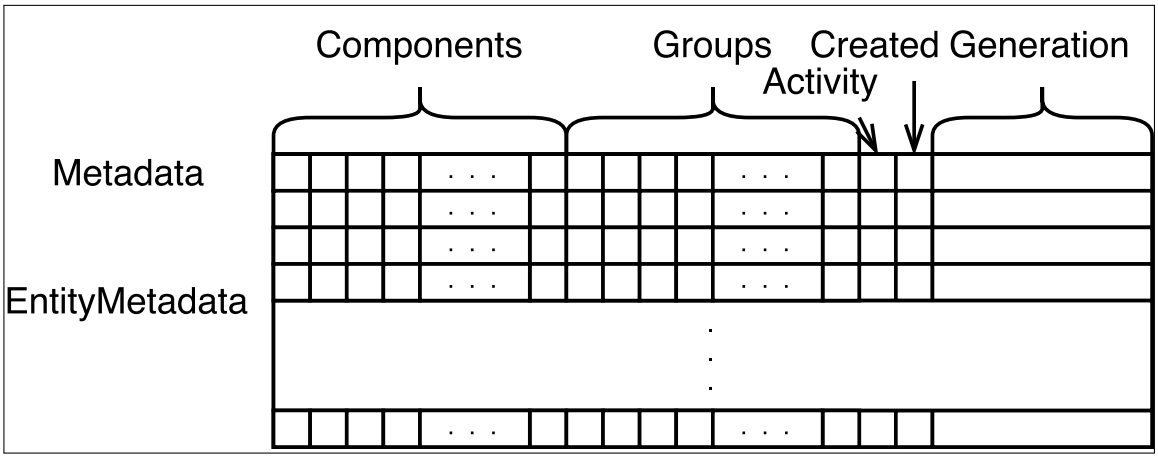
Jelikož je mapováni identifikátorů entit na komponenty již součástí nosičů komponent, tabulka již tuto informaci nemusí obsahovat. Výhodné je však udržování informace o existenci komponent pro jednotlivé entity, které může být použito pro efektivní filtrování entit. Udržování tohoto typu informace je možné díky jednotnému rozhraní pro práci s komponenty (**Universe**). Mezi typy metadat použitých v tomto návrhu patří:

- Aktivita – Entita může být ve stavu aktivní, nebo neaktivní.
- Obsazenost – řádek je použitý, nebo prázdný.
- Přítomnosti každého registrovaného typu komponent.
- Přítomnost entity ve skupinách [7].

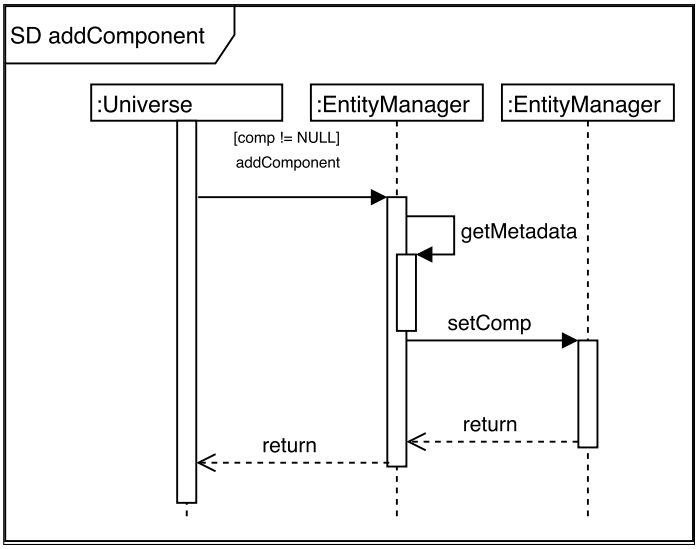
Důležitou součástí *ECS* je možnost pracovat pouze s entitami, které mají požadované komponenty („aspekty“). Z tohoto důvodu je součástí správy entit i možnost filtrování (třída **EntityFilter**). Filtrovat lze pomocí seznamu požadovaných komponent, ale je také možno definovat zakázané komponenty, nebo specifikovat požadovanou úroveň aktivity entit (aktivní/neaktivní).

Poslední částí podsystému správy entit je třída **Entity**, jejíž funkcionalita zjednodušuje práci s *ECS*. Operace na ní provedené jsou pouze přesměrovány na **Universe**, spolu s příslušným identifikátorem entity. Pomocí této třídy lze také *ECS* propojit s okolním *objektově orientovaným* kódem.

[6] Indexace tabulky pomocí *ID* části identifikátoru.
[7] Více o skupinách je součástí části. 3.4 .



Obrázek 3.6: Tabulka metadat, indexy jednotlivých entit jsou implicitní indexy řádků.



Obrázek 3.7: Sekvenční diagram modifikace metadat při přidání komponenty, pokračování obr. 3.4.

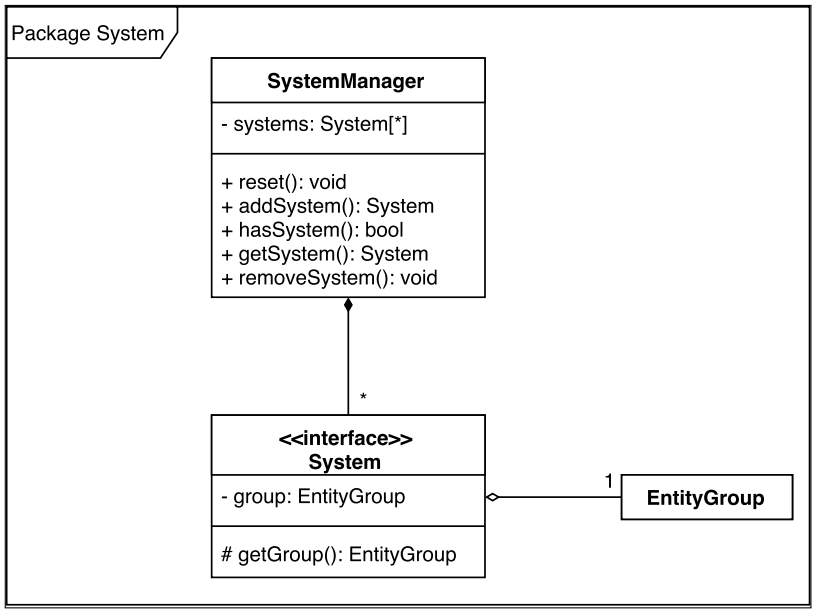
Nevýhodou udržování informací o přítomnosti komponent na dvou místech je, že při každé změně je nutné, aby byla provedena i v metadatech. Příkladem tohoto problému je operace přidání komponenty, jejíž sekvenční diagram lze vidět na obr. 3.4. Pokud je první fáze (přidání komponenty z pohledu **Component Manager**) úspěšná, je nutné upravit metadata.

3.4 Systémy a skupiny

Systém, podle *ECS* paradigmatu, obsahuje pravidla pro výběr vhodných entit a akce, které na nich provádí. Entity jsou filtrováni pomocí množiny požadovaných komponent (aspektů). Vstupem *systému* je tedy seznam entit, které vyhovují filtračním pravidlům a jeho cílem je transformovat dané entity a jejich komponenty pomocí definované akce. Tato myšlenka je základem podsystémy správy *systémů*.

Předpokladem, při návrhu tohoto entitního systému je, že i přeš vysoký počet entit, které se v systému mohou nacházet v jeden okamžik, bude množství skutečně používaných entit [8] nižší. Z tohoto předpokladu vychází myšlenka *skupin*, které jsou plní funkcí vyrovnávací paměti *systémů*. *Skupina* je množinou entit, které splňují požadavky specifikované filtrem. Oproti původnímu návrhu, kdy každý *systém* iteruje (lineárně prochází) nad seznamem všech existujících entit v entitním systému, se při použití skupin již prochází pouze takové entity, které odpovídají požadavkům daného systému.

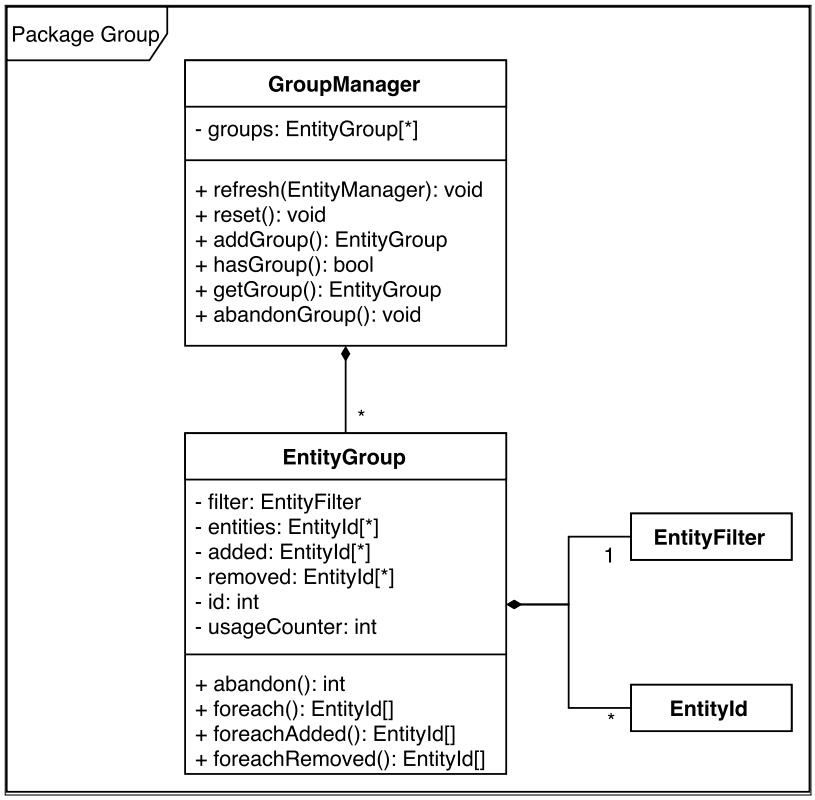
Jednou z nevýhodou použití *skupin* je nutnost skupiny udržovat aktuální, čímž je pověřen podsystém správy *skupin*. Díky potřebě udržování seznamu entitních identifikátorů způsobuje tento systém také vyšší použití paměti. Poslední důležitou vlastností je, že tato funkce předpokládá, že množina všech entit je větší, než množina používaných entit. Pokud tento předpoklad neplatí, potom není vhodné systém *skupin* používat.



Obrázek 3.8: Diagram tříd podsystému správy *systémů*.

Cílem podsystému správy *systému* (obr. 3.8), je údržba systémů a jejich vazba na skupiny. Třída **SystemManager** umožňuje přidávání a odběr nových systémů. *Systémy* jsou definovány uživatelem, ktery vytvoří vlastní třídu, která dědí ze základní třídy **System**. Dále uživatel musí specifikovat požadované vlastnosti entit – vyžadované / zakázané komponenty a úroveň aktivity. Následně je možné nový typ *systému* přidat pomocí manažerské třídy.

[8] Těch, které budou používány jako součást některého ze systémů.



Obrázek 3.9: Diagram tříd podsystému správy *skupin*.

Skupiny (**EntityGroup**) obsahují 3 seřazené seznamy entitních identifikátorů, jejich významy jsou následující – přidané entity, odebrané entity a skutečný seznam všech vyhovujících entit. Důvod pro existenci prvních dvou je primárně umožnit uživateli propojit entitní systém se systémy okolními, které (např. fyzikální simulace) potřebují objekty registrovat.

Důležitou vlastnosti *skupin* je možnost přidávat a odebírat je za běhu aplikace [9]. Tato vlastnost je umožněna skrz podsystém správy *skupin* (3.9). Každá *skupina* obsahuje „počítadlo referencí“, které reprezentuje na kolika různých místech je *skupina* používána. V případě, že počítadlo dosáhne hodnoty 0, je *skupina* odstraněna (nebo je pouze zastaven její běh). Zpráva *skupin* závisí na entitních metadatech, které umožňují efektivnější aktualizace jejich obsahu [10].

3.5 Paralelní přístup

Díky modulárnímu návrhu entitního systému, který byl zatím předveden je implementace základních typů paralelního přístupu velmi přímočará. Mezi základní typy paralelizmu, který entitní systém podporuje jsou paralelizmus na úrovni entit a paralelizmus na úrovni systému.

Paralelizmus na úrovni entit využívá *datového paralelizmu*, kde množina entit, nad kterou systém vykonává akce je rozdělena mezi požadovaný počet paralelních vláken. Tento

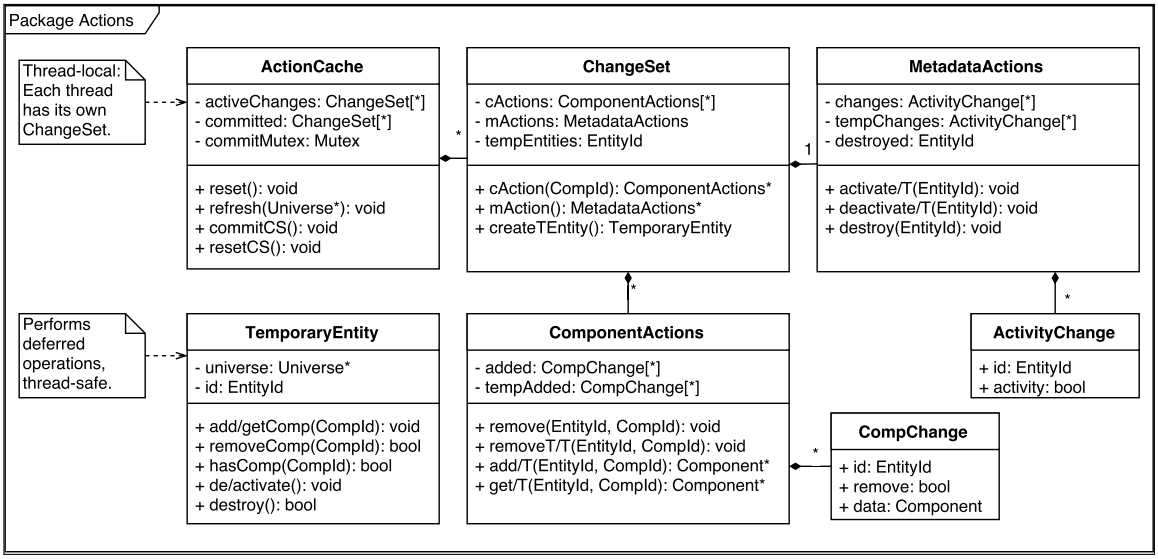
[9] Některé systémy nemusí zpracovávat entity po celý běh aplikace.
[10] Více o tomto problému je součástí implementační části 4.4.

typ paralelizmu je vhodný pouze v případech, kdy akce vykonávají transformace na entitách (a jejich komponentách), bez potřeby informací z ostatních entit.

Paralelizmus na úrovni systémů lze přirovnat k *úkolovému paralelizmu*, kde každé vlákno zpracovává rozdílný systém. Tento způsob má opět omezení – systémy nemohou přistupovat ke stejným komponentám (stejných entit). Tyto dva způsoby paralelizmu lze kombinovat.

V situacích, kdy není možné použít ani jednu z těchto možností, nebo při manipulaci globálního kontextu z několika vláken, např. mazání entit, změna aktivity, je nutné použít jiné řešení. Jedním způsobem, jak tento problém vyřešit, je použití zámků (např. vzájemné vyloučení – „mutex“), které budou tyto metody chránit před problémy souběhu („race condition“). Zámky jsou čistým způsobem, jak implementovat bezpečný paralelizmus, ale vysokém počtu operací zamčení a odemčení je možné ztratit výkon, který byl získán paralelním zpracováním [11]. Kvůli těmto nevýhodám je součástí návrhu také třetí metoda paralelizace – pomocí *množin změn*.

Množiny změn umožňují jednotlivým vláknům odkládat provedení požadovaných operací na pozdější dobu, kdy již bude možné zaručit, že nedojde k problémům souběhu. Aktuální stav světa (entit, komponent apod.), pro každé vlákno, je vytvořen překrytím stavu globálního světa danou *množinou změn*. Pokud vlákno použije operaci přidání komponenty, ke specifikované entitě, skutečné vykonání operace nad globálním kontextem bude odloženo, ale informace o této operaci bude přidána do *množiny změn* aktuálního vlákna.



Obrázek 3.10: Diagram tříd podsystému správy *akcí*.

Základem podsystému *množin změn*, jehož diagram lze vidět na obr. 3.10, jsou třídy **ActionCache** a textbfChangeSet. Úkolem **ActionCache** je zajistit přístup k instanci *množin změn*, která bude pro každé vlákno unikátní. Dále umožňuje také potvrzení (**commitSC**), nebo zrušení (**resetCS**) *množiny změn* aktuálního vlákna.

Zajímavým problémem je odložená tvorba entit a následné provádění operací nad nimi. Pro operace, které jsou provedeny nad *reálnými* entitami [12] je možno specifikovat cílovou entitu pomocí jejího identifikátoru. Jedním možným řešením je zakázat provádění operací

[11] Pokud jsou operace chráněné zámky používány často, je možné paralelní aplikaci redukovat na aplikaci sekvenční.

[12] Takové entity, které jsou již vytvořeny v globálním kontextu.

nad dočasnými entitami, nebo zrušit koncept dočasných entit kompletně a provádět operace vytvoření entity okamžitě [13]. Druhým řešením, které je součástí tohoto návrhu, je přiřazení speciálního typu identifikátoru dočasným entitám, který bude později převeden na identifikátor reálné entity. Operace, jejíž provedení je odloženo na později, mají postfix „**D**“ („deferred“) a operace, které pracují s dočasnými entitami jsou zakončeny „**T**“ („temporary“).

Množiny změn samotné jsou reprezentovány třídou **ChangeSet**, která obsahuje odložené *akce*. *Akce* je možno rozdělit na typy podle změn:

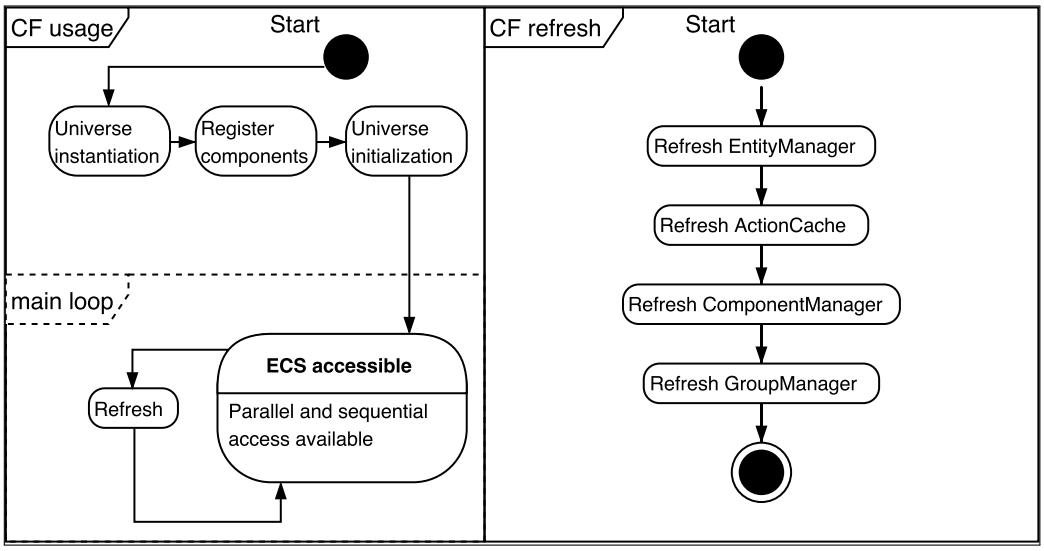
- Změny metadat – rušení entit, změna aktivity entit.
- Změny komponent – přidání/odebrání komponent, změna hodnot komponent.

a podle cílové entity:

- Reálné entity – operace obsahuje identifikátor reálné entity.
- Dočasné entity – operace obsahuje identifikátor dočasné entity.

3.6 Tok řízení

Výše navržený entitní systém se vždy nachází v jedné z následujících fází – *inicializace*, *iterace* nebo *obnova*. Diagram řízení toku, který tyto fáze obsahuje, lze vidět na obr. 3.11.



Obrázek 3.11: Digram toku řízení komponentního systému.

Prvním krokem ve fázi inicializace je konstrukce (vytvoření instance) třídy **Universe**, jejíž součástí je inicializace jednotlivých podsystémů. Potom je již možno registrovat typy komponent, které je potvrzeno závěrečnou inicializací objektu **Universe** [14], čímž se entitní systém přesunuje do fáze *iterace*.

[13] Toto by znamenalo použití zámků na celý podsystém správy entit a metadat.
[14] Po inicializaci již není možno přidávat nové typy komponent.

Fáze *iterace* umožňuje plný přístup k entitnímu systému ze strany uživatele. Kromě práce s entitami a komponentami, je také možné přidávat a odebírat *systémy* a *skupiny*. Obsah jednotlivých *skupin* je v této fázi konstantní, čímž je umožněna práce *systémům*. V okamžiku, kdy již neběží žádné *systémy* je možné přejít do fáze *obnovy*, použitím operace **refresh**.

Úkolem *obnovovací* fáze je posunutí entitního systému do následujícího stavu, ze stavu aktuálního a dokončení operací, které by vedly k porušení konzistence systému. Tohoto cíle je dosaženo postupným voláním operace **refresh** na jednotlivé podsystémy, jejichž pořadí lze vidět na obr. 3.11. Obnova entitního podsystému umožňuje dokončení operací přidání a odebrání *skupin*, u kterých je nutné měnit počet sloupců v tabulce metadat. Následuje aplikace *množin změn*, kdy jsou dokončeny odložené operace. Obnova pokračuje volání **refresh** nad jednotlivými nosiči komponent, kterým je tímto umožněna reorganizace dat. Poslední částí je příprava obsahu *skupin* pro příští fázi *iterace*. Po dokončení obnovy je systém opět uveden do fáze *iterace*.

Kapitola 4

Implementace

Tato kapitola obsahuje popis implementace výše popsaného návrhu entitního systému, založeného na *ECS* paradigmatu, přičemž jsou zde zmíněny pouze zajímavější implementační detaily. V první části je zdůvodněn výběr implementačního jazyka a použitých knihoven. Následují části zaměřené na jednotlivé podsystémy entitního systému – *komponenty*, *entity* a *systémy*. Závěr kapitoly je zaměřen na implementaci paralelního přístupu a *obnovovací* fázi. Výsledným produktem této implementace je knihovna, která umožňuje práci s entitním systémem za využití *ECS* paradigmatu.

Dosavadní návrh entitního systémy lze shrnout do tří částí. První z nich je správa komponent, kde komponenty jsou základní datové bloky, ze kterých se dále skládají entity. Komponenty jsou uchovávány v *nosičích* komponent, které zprostředkovávají mapování entit na jejich komponenty. Druhou částí entitního systému jsou entity samotné. Entita je definována identifikátorem, který jednoznačně určuje danou entitu. Identifikátor je složen ze dvou částí – index a generace. Poslední částí jsou *systémy*, které provádějí akce nad entitami. *Systém* je složen z definice entit, o které má zájem – které komponenty entita má obsahovat – a z akce, kterou nad vybranými entitami provádí. Součástí správy *systémů* jsou také *skupiny*, které operují jako vyrovnávací paměť a obsahují seznam entit pro daný *systém*.

4.1 Implementační nástroje a přenositelnost

Pro implementaci byl zvolen programovací jazyk *C++* [7], ve standardu *C++14* [3]. Jazyk *C++* byl vybrán díky jeho dlouhodobému používání v herním průmyslu, jako primární implementační jazyk mnoha herních *enginů*. Mezi jeho další výhody patří optimalizované překladače, nebo možnosti výběru z několika programovacích paradigmat. Jazyk byl použit ve standardu *C++14*, který kromě ulehčení práce s typy obsahuje také možnosti použití *šablon proměnných* (*variable templates* [4]).

Další důležitou vlastnosti, která byla zvážena při výběru implementačního jazyka, je přenositelnost vytvořené knihovny na různé hardwarové architektury a operační systémy. Díky standardu *C++* je možné psát přenositelný kód, který je následně přeložen na specifické instrukce pro cílovou platformu a výsledné chování aplikace j stejné [1].

[1] Překlad knihovny byl testován za použití překladačů *GCC*, *Clang* a *MSVC*, více ke specifikaci testovacích systémů lze najít v kapitole 5.

Pro implementaci entitního systému byly použity pouze standardní knihovny jazyka *C++*, čímž jsou zmírněny potenciální komplikace při používání výsledné knihovny. Při vývoji bylo také použito *šablonového metaprogramování* [29], které umožňuje práci s typy.

4.2 Komponenty

Komponenty lze definovat jako třídy (nebo struktury), pro která platí určitá omezení. Prvním z nich je požadavek, aby komponenta obsahovala pouze data, která lze kopírovat jako čistou paměť (*POD*), jelikož volání konstruktorů a destruktorů není zaručeno. Komponenta může obsahovat konstruktory, které inicializují data na jejich požadovanou hodnotu, ale vždy musí existovat výchozí konstruktor [2]. Komponenty také mohou specifikovat, který nosič by měl být použit pro jejich uchovávání [3]. Typ nosiče je definován v zanořeném typu s názvem **HolderT**.

Registrace komponent je prováděna pomocí statických šablon proměnných, kde pro každý typ komponent je vytvořena při překladu [4] instance typu **ComponentRegister** (diagram na obr. 3.3). Tento registr obsahuje kromě informací o komponentě také instance jejího nosiče. Díky *šablonovému metaprogramování* je možné přímé mapování komponenty na instanci jejího nosiče za překladu programu, což umožňuje překladači vyšší úroveň optimalizace.

Knihovna obsahuje několik typů základních nosičů komponent. Prvním z nich je implementace využívající standardní mapu (**std::map**), která mapuje identifikátory entit přímo na instance jednotlivých komponent. Tento typ nosiče je výhodný pro komponenty, které nejsou příliš často používané. Dalším krokem je nosič, který obsahuje souvislé pole komponent a mapu, která umožňuje mapování identifikátoru entity na index do pole komponent. Tento nosič je výhodný v případech, kdy komponenty stále nejsou příliš časté, ale jsou zpracovávány v blocích. Poslední typ nosiče je implementován jako čisté pole komponent, kde je mapování prováděno přímou indexací, pomocí identifikátoru entity. Výhodou je konstantní složitost přístupu k jednotlivým komponentám, nevýhodou potom paměťová náročnost v případě, kdy daný komponent neobsahuje každá entita.

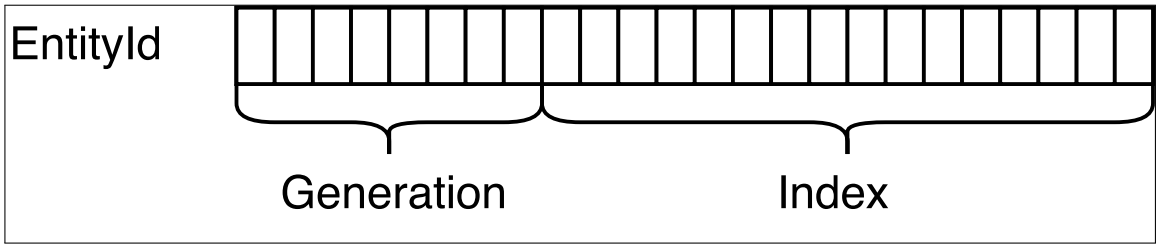
Uživatelům knihovny je umožněna implementace vlastních nosičů komponent, vytvořením vlastní třídy, která dědí ze základní třídy **BaseComponentHolder**. Jelikož informace, zda entita obsahuje daný komponent, je zpravována podsystémem entit, je velmi jednoduché implementovat např. nosič značek [5].

4.3 Správa entit

Entity jsou definované pomocí identifikátorů, složených ze dvou částí – index a generace. Index určuje číslo řádku v tabulce entit a je také používán pro mapování komponent na entity. Generace určuje specifickou entitu, která „obývá“ daný řádek tabulky entit. Tento identifikátor je implementovaný jako třída **EntityId**, která, na rozdíl od návrhu, obsahuje pouze jedno číslo, rozdělené pomocí bitových operací do dvou částí (obr. 4.1). Ve výchozím nastavení knihovny je identifikátor reprezentován 32-bitovým číslem bez znaménka, indexová část zabírá 24 bitů a generace 8 bitů [6]. Validní identifikátor obsahuje indexové číslo,

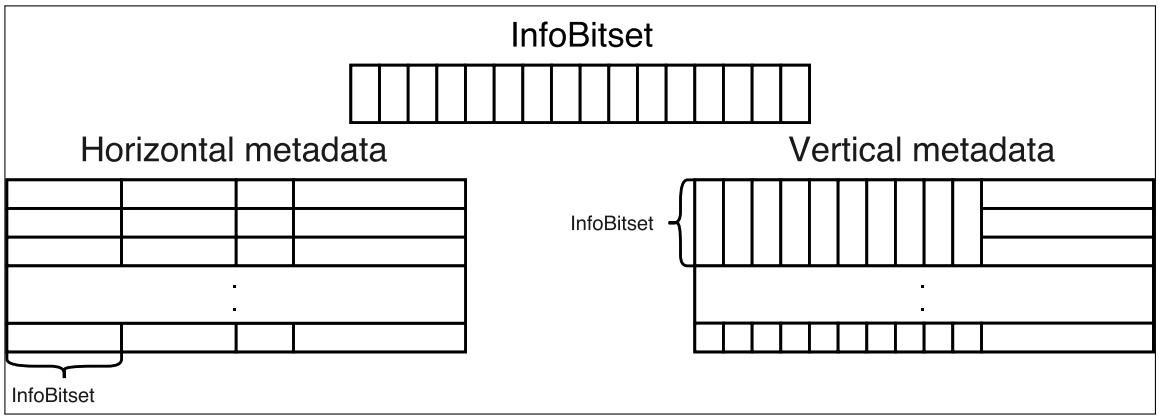
[2] Konstruktor bez parametrů.
[3] Pokud není specifikován, je použit výchozí nosič.
[4] Díky tomu, že jsou registry statické proměnné.
[5] Značka je komponent, který neobsahuje žádná data.
[6] Tato nastavení lze změnit v hlavičkovém souboru.

které není rovné nule. Speciálním případem identifikátoru je identifikátor dočasné entity, jehož generační číslo je rovno maximální hodnotě (255 pro výchozí nastavení).



Obrázek 4.1: Rozdělení identifikátoru entit na generaci a index.

Informace o entitách je uložena v tabulce entit, která obsahuje – aktuální generaci, přítomnost komponentů, přítomnost entity ve skupinách, aktivitu a informaci o tom, zda je daný řádek použitý. Až na generaci lze všechny informace reprezentovat jedním bitem, který je vždy ve stavu '1' (*true*) nebo '0' (*false*). Jelikož standardní množina bitů (**std::bitset**) nezaručuje uložení bitů v souvislé paměti, vznikla za tímto účelem třída **InfoBitset**. Kromě souvislého uložení paměťových bloků, které obsahují jednotlivé bity, má **InfoBitset** další výhodu v tom, že výše zmíněné paměťové bloky jsou uloženy ve třídě samotné a tudíž je možné její instance přesouvat pomocí operací s pamětí (*std::memcpy* atp.).



Obrázek 4.2: Implementace tabulky metadat pomocí množin bitů **InfoBitset**. Tabulka obsahuje následující informace – komponenty, skupiny, aktivitu, použití řádku a generaci. Z této množiny informací jsou všechny, až na generaci, reprezentovány pomocí množin bitů.

Existují dva způsoby, jak organizovat jednotlivé bitové množiny – horizontálně a vertikálně, ilustraci lze vidět na obr. 4.2. Horizontální tabulka metadat obsahuje jednotlivé sloupce v bitech stejné množiny bitů, na rozdíl od vertikální tabulky metadat, kdy pro každý typ metadat (např. každý typ komponent) existuje oddělený sloupec množin bitů.

Mezi výhody horizontálního ukládání metadat patří jednodušší implementace a rychlejší filtrování [7]. Velkou nevýhodou je však problematický paralelní přístup k jednotlivým bitům,

[7] Filtrování je v tomto případě pouze bitová operace *AND*.

jelikož není možné zaručit atomické bitové operace – operace nad jedním bitem ovlivňují celý blok bitové paměti.

Vertikální metadata naopak umožňují paralelní přístup k odděleným sloupcům, díky čemuž je také použita ve finální implementaci této knihovny. Středem tabulky je třída **MetadataGroup**, která umožňuje udržovat skupinu sloupců metadat. Sloupce tabulky metadat jsou rozděleny do tří skupin – komponenty, skupiny a ostatní. Skupina ostatních metadat obsahuje aktivitu entit a využitelnost řádku.

Operace vytvoření entity je realizována přidáním řádku do tabulky entit, nebo, pokud je dostatek volný identifikátorů, použití již dříve vytvořeného řádku. Při operaci mazáni entit, je důležitou součástí inkrementace generačního čísla, které identifikuje jednotlivé entity, které existovaly na stejném řádku. Při recyklaci řádku má nově vytvořená entita stejný index, ale rozdílné generační číslo. Jelikož by při opakovaném vytváření a mazání stejných entit mohlo dojít k rychlému vyčerpání generačních čísel a jejich přetečení [8], je seznam volných řádků implementován jako oboustranná fronta.

4.4 Systémy a skupiny

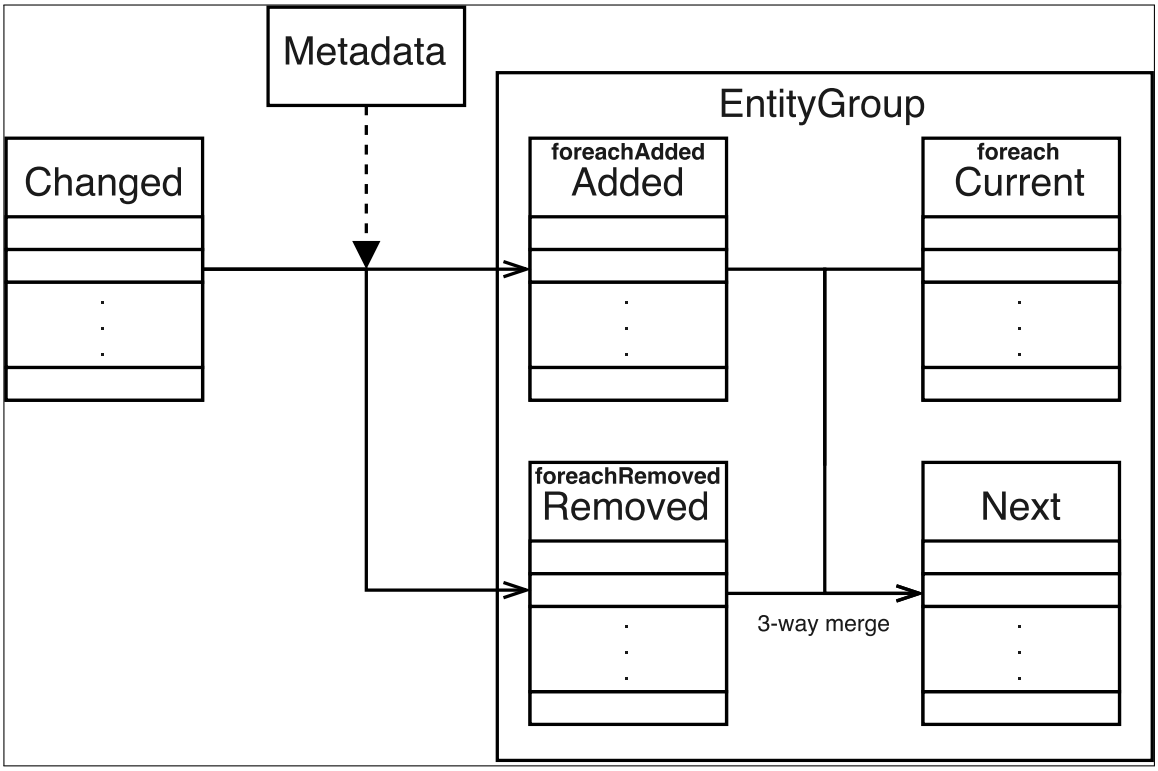
Systémy jsou vytvářeny uživateli knihovny, děděním základní třídy **System**. Následně je nutné *systém* přidat použitím metody **addSystem**, která také zaručí jeho inicializaci a přiřazení *skupiny*. Instance daného *systému* je spravována knihovnou samotnou.

Definice požadovaných entit je přístupná uživateli skrz typové proměnné **Require** a **Reject**. Jejich výchozí hodnota je prázdná, což znamená, že *systém* přijímá všechny (aktivní) entity. Požadovanou hodnotu lze specifikovat skrz pomocné seznamy typů následujícím způsobem:

```
using Require = ent::Require<PositionC, MovementC>;
using Reject = ent::Reject<>;
```

kde **PositionC** a **MovementC** jsou požadované komponenty. *Systémy* dále obsahují pomocné metody, které umožňují přístup k jednotlivým typům entitních seznamů – přidané (**foreachAdded**), odebrané (**foreachRemoved**) a celý seznam (**foreach**).

[8] Není samo o sobě chyba, ale je vhodné se vyhnout příliš vysoké frekvenci opakováni generačních čísel.



Obrázek 4.3: Aktualizace seznamů *skupin*.

Středem *skupin* je třída **EntityGroup**, jejíž úkolem je udržování seznamů přidaných, odebraných a vyhovujících entit. Jelikož všechny operace musí procházet skrz rozhraní knihovny (**Universe**), je možné tvořit seznam entit, které se změnily. Během fáze *obnovení* je seznam použit pro generování nových seznamů přidaných a odebraných entit, pro každou *skupinu*. Operace synchronizace *skupin* je dokončena tří-směrným sloučením seznamů přidaných, odebraných a aktuálních entit. Ilustraci tohoto postupu lze vidět na obr. 4.3.

4.5 Podpora paralelizmu

Knihovna podporuje tři typy paralelního přístupu – systémy, entity a *množiny změn*. Pro paralelní zpracování několika systémů, které nepřistupují ke stejným komponentám, nebylo třeba příliš práce, díky způsobu, kterým je knihovna navržena a využití *vertikální* tabulky metadat. Paralelizací na úrovni entit je myšleno rozdělení *skupiny* entit jednoho *systému* mezi několik vláken, které dané entity zpracovávají odděleně [9]. Jelikož je tabulka metadat implementována pomocí *bitových množin*, které nezaručují bezpečnost operací práce s bity, při přístupu z několika vláken zároveň, bylo třeba implementovat speciální typ *iterátoru* – třída **EntityListParallel**. Úkolem tohoto *iterátoru* je rozdělení seznamu entit do daného množství (podle počtu vláken) podmnožin.

Při implementaci *množin změn* vyvstaly problémy s typovým systémem jazyka *C++*. Akce nad komponenty – přidání, odebrání a změny hodnot – vyžadovaly oddělené seznamy, které je možné uniformě aplikovat v *obnovovací fázi*. Tento problém byl vyřešen využitím

[9] Bez použití operací, které ovlivňují globální kontext – odebírání komponent apod.

šablonového metaprogramování, kde ve *fázi registrace* jsou registrovány také instance třídy **ComponentExtractor**. Tato třída následně operuje jako most mezi třídou **ActionCache** a specifickými seznamy komponent.

4.6 Obnovení konzistence

Cílem *obnovovací fáze* (**refresh**) je uvedení entitního systému konzistentního stavu a dokončení operací, které nemohly být provedeny při jeho aktivním používání. Základní postup obnovy lze vidět na alg. 1. Zajímavou částí je obnova *skupin*, která využívá seznamu změněných entit, díky kterému není třeba procházet všechny entity, ale pouze ty, které byly změněny od poslední obnovy konzistence. Tento seznam je tvořen při normálním používání entitního systému a při obnovovací fázi *správy akcí*, kdy jsou aplikovány *množiny změn*. Seznamy jsou implementovány jako seřazené pole identifikátorů entit, kdy každé vlákno, které pracuje s entitním systémem má tento seznam vlastní. Před předáním seznamu obnovovací funkci *správy skupin*, je nutné ho vytvořit spojením seznamů jednotlivých vláken.

Algoritmus 1: Postup obnovení

```
1:  function REFRESH
2:      Obnova správy entit
3:          Finalizace přidání skupin
4:      Obnova správy akcí
5:          Smazat označené entity
6:          Přidat nové entity
7:          Přidat/odebrat komponenty
8:          Změna aktivity entit
9:      Obnova správy komponent
10:         Obnova jednotlivých nosičů komponent
11:     Obnova správy skupin
12:         Obnova skupin – vyprázdnění seznamů přidaných a odebraných entit
13:         Odebrání nepoužívaných skupin
14:         Kontrola seznamu změněných entit a synchronizace skupin
15:         Dokončení operace synchronizace skupin
16: end function
```

4.7 Možnosti rozšíření

Obsah kapitoly „implementace“ obsahuje stav knihovny, tak, jak byla odevzdána s touto technickou zprávou, i když již umožňuje všechny funkce popsané v kapitole „návrh“, nelze ji považovat za dokončenou. Aktuální stav implementace lze nalézt v *GIT repozitáři* [42], pod jménem „Entropy“. Následují možné směry, jak rozšířit a vylepšit tuto základní implementaci:

- **Vazba na vestavěné skriptovací jazyky** – Umožnit přístup k entitnímu systému skrz některý z často používaných vestavěných skriptovacích jazyků – např. *LUA*.
- **Datově definované entity** – Podpora definice entit a komponent v externích souborech (např. formátu *JSON*) a možnosti serializace komponent.
- **Dynamické komponenty** – Díky způsobu implementace tabulky metadat je teoreticky možné přidávat nové typy komponent i za běhu aplikace. Spojení této vlastnosti s vazbou na skriptovací jazyk by umožnilo případným herním návrhářům větší volnost, bez nutnosti opakovaných úprav a následného překládání zdrojového kódu v jazyce *C++*.
- **Tvorba entit** – Možnosti tvorby entit po blocích a specifikace identifikátoru vytvořené entity.
- **Volitelné použití skupin** – Pro *skupiny*, které obsahují téměř všechny entity je výhodnější iterovat nad všemi entitami.
- **Skupiny pro neaktivní entity** – *Skupiny* již tuto vlastnost obsahují, ale pro tuto chvíli není možné specifikovat požadovanou aktivitu entit.
- **Bezpečnost entit** – Rozdělit entity na bezpečné a nebezpečné, kdy pro nebezpečné entity jsou provedeny kontroly při každém přístupu k entitnímu systému.
- **Paralelizace obnovovací fáze** – Jednotlivé akce, které jsou prováděny součástí obnovovací fáze lze paralelizovat, ale tato vlastnost zatím není implementována.

Kapitola 5

Vyhodnocení

Obsahem této kapitoly je vyhodnocení výsledné implementace z pohledy výkonosti a použitelnosti. První část představuje hardwarové a softwarové konfigurace systémů, na kterých bylo prováděno testování. Následuje popis použitých nástrojů, použitých ve vyhodnocení a jejich výsledků. Poslední část obsahuje porovnání implementace představené v této práci s podobnými volně dostupnými knihovnami.

5.1 Testované sestavy

Součástí vývoje knihovny bylo prováděno její testování na konfiguracích uvedených v tabulce 5.1. Překlad byl prováděn za použití nejvyšší úrovně optimalizace (*-O3*, *Ø2*), použité překladače a jejich verze lze najít v tabulce 5.2.

ID	CPU	Operační paměť
1	Intel Core i7-4710HQ CPU @ 2.50 GHz – 4 cores (8 threads)	8 GB DDR3 @ 1600 MHz
2	Intel Core i5-4670K CPU @ 3.40 GHz – 4 cores (4 threads)	24 GB DDR3 @ 1600 MHz

ID	Operační systém
1	Linux x86_64, Fedora 25 – kernel 4.9.3-200.fc25.x86_64
2	Windows x86_64, Windows 10 Pro – version 1607, OS build 14393.447

Tabulka 5.1: Tabulka s testovanými hardwarovými konfiguracemi.

ID	Překladač	Verze
1	GCC	6.3.1 20161221 (Red Hat 6.3.1-1)
1	Clang	3.9.1
2	Microsoft C/C++ Optimizing Compiler	19.10.24728

Tabulka 5.2: Tabulka s testovanými překladači jazyka *C++*. **ID** obsahuje identifikátor z tab. 5.1.

5.2 Testování knihovny

Důležitou součástí vývoje bylo testování knihovny pomocí metody *unit testing*. Pro tento účel je využito jednoduché knihovny v jazyce *C++* [1], která umožňuje psát testy přímo ve zdrojovém kódu. Kromě testování bylo také prováděno profilování různých částí knihovny, kvůli možným optimalizacím, k čemuž bylo opět využita výše zmíněná knihovna.

Kromě nástrojů zabudovaných přímo do zdrojového kódu byly také použity externí aplikace :

- **GDB – GNU Debugger**
- **Valgrind** – Primárně moduly – *callgrind*, *cachegrind*, *helgrind* a *massif*.
- **ms_print**
- **KCachegrind**

5.3 Výkonnostní testy

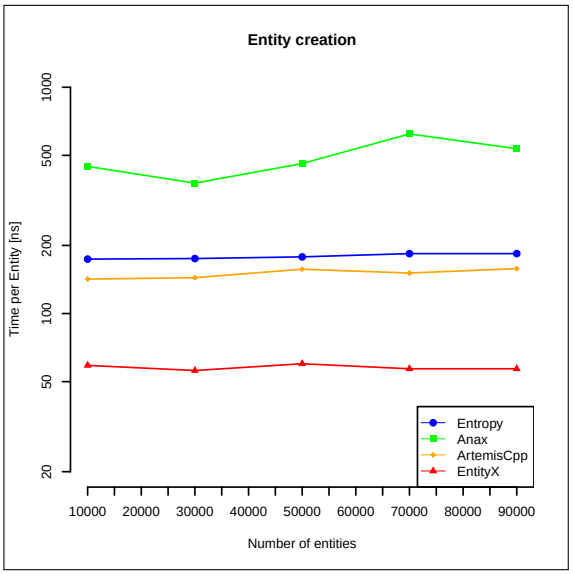
Tato část obsahuje výsledky výkonnostních testů spuštěných ná výše popsané implementaci entitního systému. Hlavním cílem je předvést co knihovna aktuální implementace knihovny zvládá a jak porovnat ji s jinými implementacemi entitních systémů založených na *ECS*. Kromě výkonosti implementace jsou zde také ukázány vlastnosti, které pramení z návrhu systému, jako např. časová složitost jednotlivých operací.

Kromě implementace, který navrhuje tato práce (nazvaná **Entropy**), jsou zde také ukázány jiné *open-source* knihovny, nad kterými jsou taktéž provedeny testy. Mezi testované knihovny patří:

- **Anax** [36] – Jako jediná z ostatních knihoven používá podobný koncept *skupin*, jako **Entropy**.
- **ArtemisCpp** [37] – Implementace knihovny **Artemis** [41] v jazyce *C++*.
- **EntityX** [24] – Obsahuje i pokročilejší funkce, např. systém událostí. Z těchto knihoven je **EntityX** nejvíce dokončená a optimalizovaná.

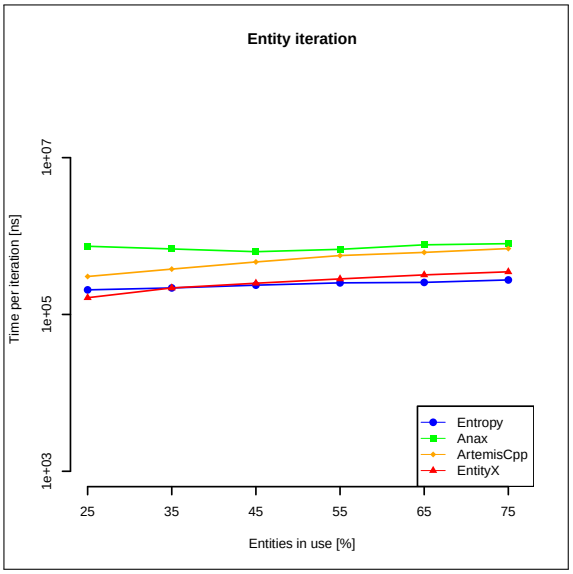
Všechny testy a výsledky, které jsou zobrazeny v této sekci byly provedeny na hardwarové konfiguraci č. *1* 5.1. Zdrojové kódy, knihoven i testovaných bloků, byly přeloženy pomocí překladače *GCC* 5.2, s parametry `-std=c++1z -O3`. Jednotlivé testovací scénáře byly následně spouštěny po uzamčení procesoru do módu nejvyššího výkonu, spolu s příkazem `nice -20`. Výsledná data byla zpracována aplikací *R* [13], jejíž výstupem jsou prezentované grafy.

[1] Tato knihovna byla vytvořena autorem této práce a není jiným způsobem zveřejněna.



Obrázek 5.1: Graf výsledků pro přidání určitého počtu entit.

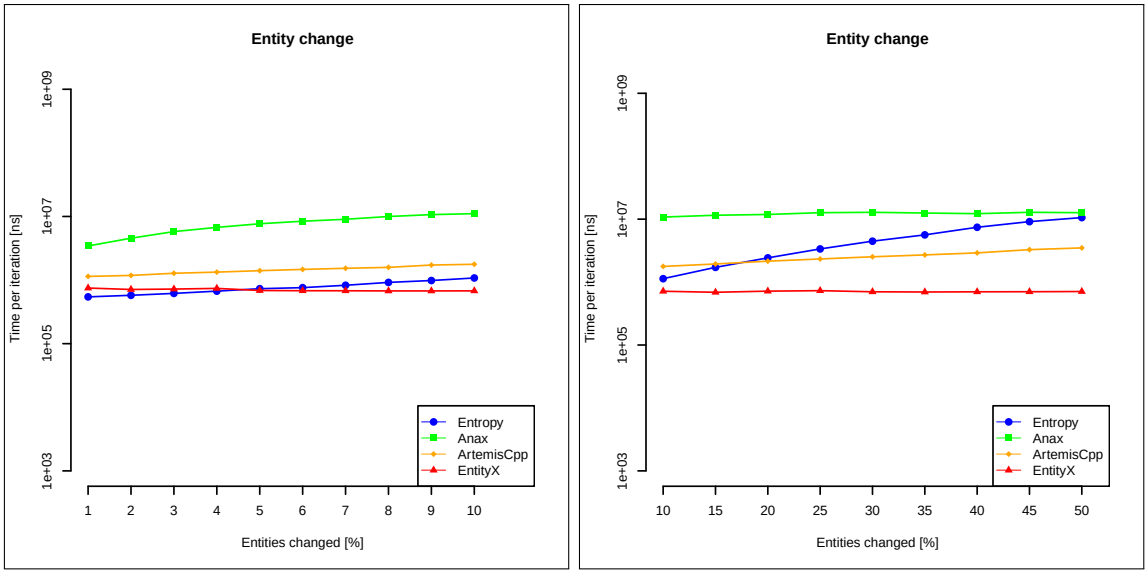
Obsahem prvního scénáře je tvorba určitého počtu entit, výsledný graf lze najít na obr. 5.1. Hodnota na ose Y určuje počet nanosekund, kolik trvá provedení jedné operace – přidání entity a dvou komponent. Z průběhu grafu lze určit, že počet přidaných entit nezvyšuje čas potřebný k přidání dalších entit. Toto umožňuje entitnímu systému obsahovat vysoké množství entit, bez omezení rychlosti vykonávání následujících akcí. Jelikož mají všechny knihovny podobné chování, rozdílem jsou zde pouze konstantní faktory.



Obrázek 5.2: Jednoduchá iterace pohybového *systému*.

Předmětem druhého scénáře je již reálnější případ použití entitního systému – pohyb entit. V tomto případě jsou použity dva typy komponent – pozice a rychlost – kde každá z

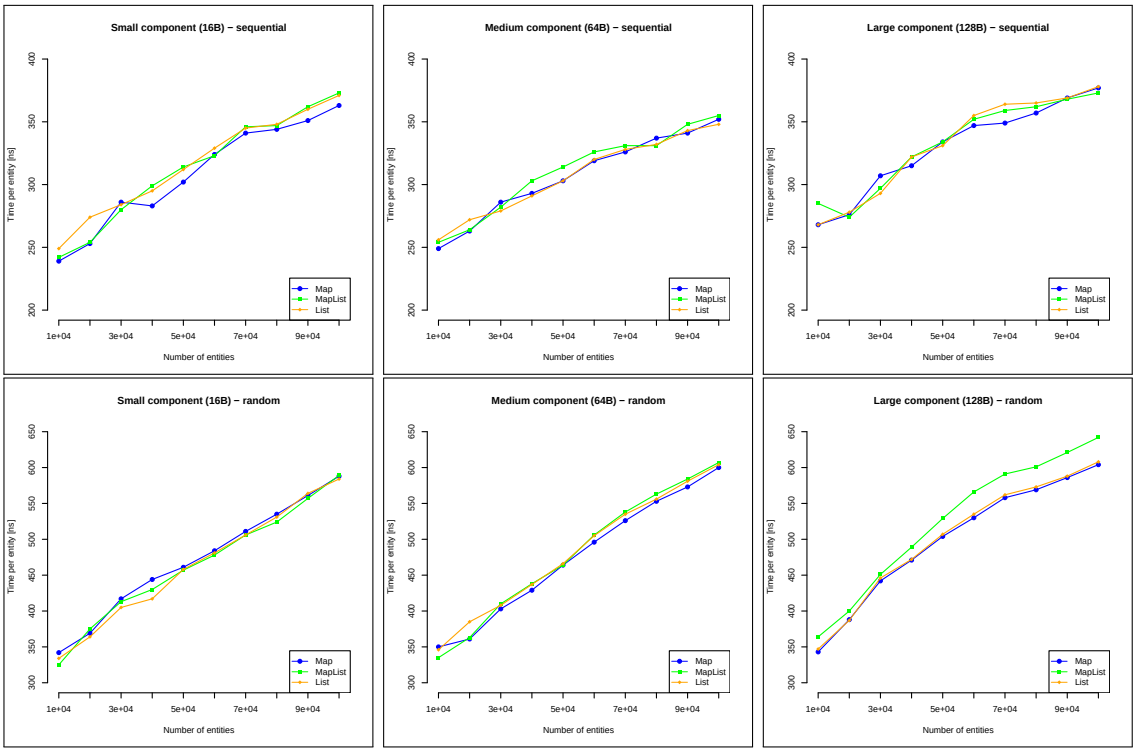
nich obsahuje dvě pole (X a Y). Dále případ obsahuje *systém*, jehož úkolem je iterace nad entitami, které obsahují oba dva typy komponent a následný posun entity o její rychlost. V první části je vytvořeno 10 000 entit, ze kterých pouze určený počet (na ose X) má přiřazeno požadované komponenty. Následuje měřená část, kdy systém prochází vyhovující entity, nad kterými provádí pohybovou akci. Výsledný graf lze najít na obr. 5.2, osa Y obsahuje čas na jednu iteraci systému.



Obrázek 5.3: Iterace s daným procentem měnících se komponent.

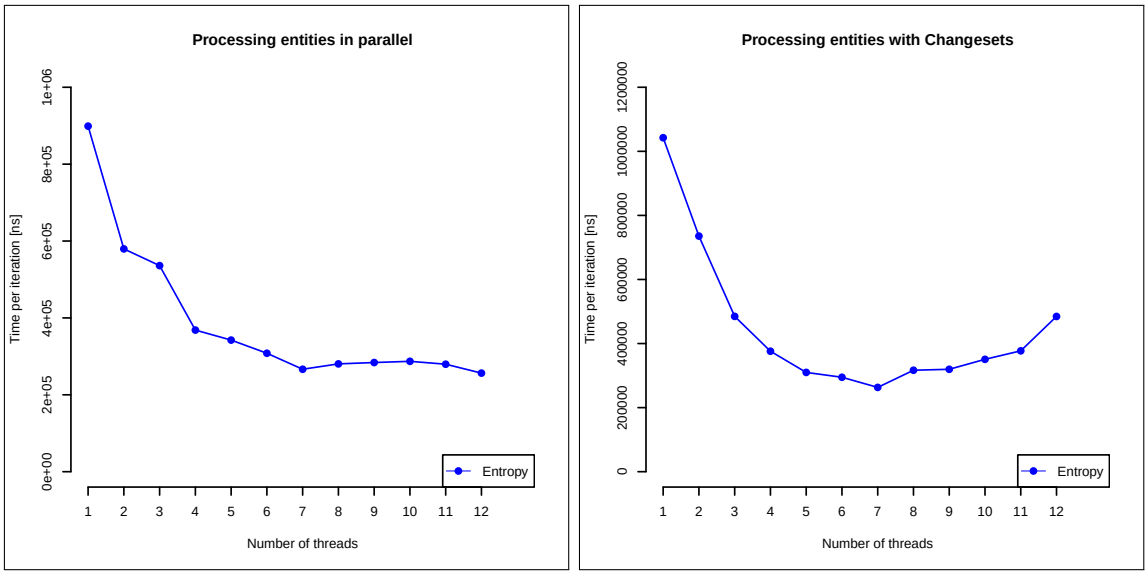
Následující testovací scénář porovnává, jakým způsobem se jednotlivé implementace chovají, v případě, kdy entity mohou měnit svou množinu komponent. Základní strategie je podobná, jako v případě pohybového systému, ale nyní existují dva *systémy*. První z nich je stejný, jako v prvním případě. Druhý iteruje nad všemi entitami a náhodně přidává/odebírá komponenty, čímž se množina entit, nad kterou první systém iteruje, mění. Pravděpodobnost, že *systém* změní entitu, je zobrazena na ose X, osa Y opět obsahuje čas, který průměrně trvá jedna iterace. Výsledné grafy – nižší a vyšší pravděpodobnosti – lze najít na obr. 5.3. V tomto případě je knihovna **Entropy**, výhodná až do určité hranice [2], kdy je režie práce se *skupinami* vyšší, než jejich výkonostní zisk. Knihovny, které nepoužívají *skupiny* musí iterovat nad všemi entitami, čímž jejich průběhy získávají lineární charakter. Mezi možná vylepšení knihovny **Entropy** patří rozdělení seznamu změněných entit do několika seznamů pro každou *skupinu* zvlášť a filtrování pouze podle změněných atributů entit.

[2] Tuto hranici lze posouvat vylepšením implementace a optimalizací.



Obrázek 5.4: Porovnání nosičů komponent pro komponenty různých velikostí.

Dalším testovaným aspektem výsledné knihovny je porovnání různých typů *nosičů komponent*. Test je založen na přidání daného počtu entit (na ose X) a každé entitě je přidán jedna komponenta. Každý nosič je testován se třemi typy komponent – malé (16 bytů), střední (64 bytů) a velké (128 bytů). Následuje cyklus, kdy je přistoupeno k vytvořeným komponentám. Grafickou reprezentaci získaných dat lze vidět na grafech na obr. 5.4, přičemž první řada grafů reprezentuje sekvenční přístup ke komponentám a druhá náhodný přístup.



Obrázek 5.5: Paralelní použití knihovny s daným počtem vláken.

Cílem závěrečného scénáře je otestovat, jak výhodné je použití několika vláken při přístupu k entitnímu systému. Testovaný systém operuje podobným způsobem, jako v případě pohybového systému. V první části je vytvořeno 10 000 entit, kde každé z nich jsou přiřazeny dvě komponenty – pozice a rychlost. Cílem výpočetního *systému* je pohyb všech entit, přičtením rychlosti k jednotlivým složkám komponenty pozice. Výsledné grafy lze vidět na obr. 5.5. V prvním případě je celková množina entit rozdělena mezi jednotlivá vlákna, naproti ve druhém případě jsou použity *množiny změn*. Průběhy potvrzují, že více vláken pomáhá k dřívějšímu dokončení výpočtu. Dále je možné určit počet vláken, od kterého již nedochází k zlepšení – v tomto případě 7-8 vláken [3] – nicméně relativní zlepšení se snižuje s každým přidaným vláknem.

[3] Procesor, na kterém byly tyto testo prováděny, obsahuje 4 jádra a 8 vláken.

Kapitola 6

Závěr

Cílem této bakalářské práce je návrh a následná implementace entitního systému, který umožňuje návrh aplikací za použití *Entity-Component-System* paradigmatu. Důležitou částí bylo studium způsobů, jakým je možno navrhovat hry a jejich vývoj s postupem času, který vedl ke vzniku tvorby entit pomocí kompozice a následně výše zmíněnému paradigmatu. Dále bylo důležitým tématem studium moderní architektury počítačových systémů, primárně hierarchie pamětí, a její využití při psaní efektivnějších aplikací. *Datově orientovaný* návrh, který se tímto tématem zabývá, je základem *ECS* architektury.

Další důležitou částí této práce bylo studium aktuálně existujících implementací a jejich vlastností, které byly ve zkratce popsány i v této zprávě. Následně byly zmíněné znalosti použity při návrhu vlastního entitního systému. Kromě základních vlastností *ECS*, patřilo mezi priority také paralelní přístup, který je u jiných knihoven v mnoha případech nemožný. Před vytvořením výsledného návrhu, tak, jak je popsán v práci samotné, vzniklo několik prototypů, jejichž cílem bylo určit vhodnost návrhu a jeho proveditelnost v jazyce *C++*.

Implementace byla provedena v programovacím jazyce *C++*, mezi jehož výhody patří vysoká efektivnost výsledných aplikací. Při implementaci byly použity pokročilé vlastnosti jazyka, jako jsou *šablony* a *šablonové metaprogramování*, které umožňují vyšší úroveň práce s typy. Dále bylo využito standardní knihovny a menších knihoven, dříve vytvořených autorem této práce.

Výsledkem práce je multiplatformní knihovna **Entropy**, která umožňuje práci s entitním systémem, založeným na *Entity-Component-System* paradigmatu. Hlavním úspěchem je možnost paralelního přístupu, který jiné, volně dostupné, knihovny nepodporují. Při porovnání výsledné implementace se knihovna **Entropy** výkonnostně řadí mezi ty rychlejší a s dalšími optimalizacemi je možné rychlost dále vylepšovat. Aktuální verze knihovny je k dispozici v repositáři *Git* [42] a bude dále autorem vyvíjena. Mezi možné optimalizace patří – např. paralelizace *obnovovací* fáze, k čemuž je již připravena, rozdělení seznamů změněných entit, podle *skupin*, které jednotlivé změny ovlivňují, nebo kompletní konverze filtrování na použití *bitových množin*.

Jak bylo zmíněno výše, autor bude dále pokračovat ve vývoji výsledné knihovny. Mezi plánovaná rozšíření, která jsou hlouběji popsána v části 4.7, patří vazba entitního systému na vestavěny skriptovací jazyk. Tímto bude umožněno herním návrhářům měnit chováni hry, bez nutnosti opětovného překládání celé hry. Další vlastností, která využívá právě vazby na skriptovací jazyk, je možnost přidávat nové typy komponent za běhu aplikace, nebo definovat entity v externích souborech (např. formátu *JSON*).

Literatura

[1] *API thread safety classifications*. [Online; navštíveno 11.04.2017].
URL https://www.ibm.com/support/knowledgecenter/ssw_i5_54/rzahw/rzahwapico.htm

[2] *Avoiding and Identifying False Sharing Among Threads*. [Online; navštíveno 20.04.2017].
URL https://software.intel.com/sites/default/files/m/d/4/1/d/8/3-4-MemMgt_-_Avoiding_and_Identifying_False_Sharing_Among_Threads.pdf

[3] *C++14 Language Extensions*. [Online; navštíveno 27.04.2017].
URL https://isocpp.org/wiki/faq/cpp14-language

[4] *C++14 Variable Templates*. [Online; navštíveno 27.04.2017].
URL https://isocpp.org/wiki/faq/cpp14-language#variable-templates

[5] *Cache Coherence*. [Online; navštíveno 14.04.2017].
URL https://www.d.umn.edu/~gshute/arch/cache-coherence.xhtml

[6] *Data Locality*. [Online; navštíveno 14.04.2017].
URL http://docs.roguewave.com/threadspotter/2011.2/manual_html_linux/manual_html/ch_intro_locality.html

[7] *Domovská stránka ISO C++ standardu*. [Online; navštíveno 27.04.2017].
URL https://isocpp.org/

[8] *Instruction scheduling*. [Online; navštíveno 14.04.2017].
URL https://www.cl.cam.ac.uk/teaching/1617/OptComp/slides/lecture14.pdf

[9] *Introduction to Data-Oriented Design*. [Online; navštíveno 13.04.2017].
URL http://www.dice.se/wp-content/uploads/2014/12/Introduction_to_Data-Oriented_Design.pdf

[10] *Kinds of parallelism*. [Online; navštíveno 20.04.2017].
URL http://www.cs.umd.edu/class/fall2013/cmsc433/lectures/concurrency-basics.pdf

[11] *Pipelining*. [Online; navštíveno 14.04.2017].
URL https://web.cs.iastate.edu/~prabhu/Tutorial/PIPELINE/pipe_title.html

[12] *Task-Based Programming*. [Online; navštíveno 20.04.2017].
URL https://software.intel.com/en-us/node/506100

[13] *The R Project for Statistical Computing*. [Online; navštíveno 28.04.2017].
URL https://www.r-project.org/

[14] *Unity*. [Online; navštíveno 14.04.2017].
URL https://unity3d.com/

[15] *Unity GameObject*. [Online; navštíveno 14.04.2017].
URL https://docs.unity3d.com/ScriptReference/GameObject.html

[16] *Unreal engine 4*. [Online; navštíveno 14.04.2017].
URL https://www.unrealengine.com/

[17] *ARM Architecture*. 2005, [Online; navštíveno 14.04.2017].
URL https://www.scss.tcd.ie/~waldroj/3d1/arm_arm.pdf

[18] *History of Processor Performance*. 2012, [Online; navštíveno 20.04.2017].
URL http://www.cs.columbia.edu/~sedwards/classes/2012/3827-spring/advanced-arch-2011.pdf

[19] *AMD64 Architecture Programmer's Manual*. 2013, [Online; navštíveno 14.04.2017].
URL https://support.amd.com/TechDocs/24592.pdf

[20] *Understanding Component-Entity-Systems*. 2013, [Online; navštíveno 19.04.2017].
URL https://www.gamedev.net/resources/_/technical/game-programming/understanding-component-entity-systems-r3013

[21] *What is an Entity System?* 2014, [Online; navštíveno 19.04.2017].
URL http://entity-systems.wikidot.com/

[22] *Intel® 64 and IA-32 Architectures Software Developer's Manual*. 2016, [Online; navštíveno 14.04.2017].
URL http://www.intel.com/content/dam/www/public/us/en/documents/manuals/64-ia-32-architectures-software-developer-vol-1-manual.pdf

[23] Acton, M.: *Data-Oriented Design and C++*. 2014, [Online; navštíveno 13.04.2017].
URL https://www.youtube.com/watch?v=rX0ItVEVjHc

[24] Alec, T.: *EntityX - A fast, type-safe C++ Entity-Component system*. [Online; navštíveno 27.04.2017].
URL https://github.com/alecthomas/entityx/tree/master

[25] Andrews, J.: *Designing the Framework of a Parallel Game Engine*. 2015, [Online; navštíveno 20.04.2017].
URL https://software.intel.com/en-us/articles/designing-the-framework-of-a-parallel-game-engine

[26] Bendersky, E.: *Dumping a C++ object's memory layout with Clang*. [Online; navštíveno 13.04.2017].
URL http://eli.thegreenplace.net/2012/12/17/dumping-a-c-objects-memory-layout-with-clang

[27] Bilas, S.: *A Data-Driven Game Object System* . 2002, [Online; navštíveno 12.04.2017].
URL
http://scottbilas.com/files/2002/gdc_san_jose/game_objects_slides.pdf

[28] Blow, J.: *New programming language for games*. 2014, [Online; navštíveno 19.04.2017].
URL https://www.youtube.com/watch?v=TH9VCN6UkyQ

[29] Dimov, P.: *Simple C++11 metaprogramming*. [Online; navštíveno 27.04.2017].
URL http://www.pdimov.com/cpp2/simple_cxx11_metaprogramming.html

[30] Fabian, R.: *Component Based Objects*. 2013, [Online; navštíveno 13.04.2017].
URL http://www.dataorienteddesign.com/dodmain/node5.html

[31] Fabian, R.: *Data-Oriented Design*. 2013, [Online; navštíveno 12.04.2017].
URL http://www.dataorienteddesign.com/dodmain/

[32] Fabian, R.: *It is all about the data*. 2013, [Online; navštíveno 20.04.2017].
URL http://www.dataorienteddesign.com/dodmain/node3.html#SECTION00310000000000000000

[33] Gamma, E.: *Design Patterns: Elements of Reusable Object-Oriented Software*. Boston:Addison-Wesley, 1995, ISBN ISBN 0-201-63361-2.

[34] Martin, A.: *Entity Systems are the future of MMOG development*. 2007, [Online; navštíveno 20.04.2017].
URL http://t-machine.org/index.php/2007/11/11/entity-systems-are-the-future-of-mmog-development-part-2/

[35] Marty, M. D. H. M. R.: *Amdahl's Law in the Multicore Era. Cover Feature*, 2008: str. 6, [Online; navštíveno 20.04.2017].
URL http://research.cs.wisc.edu/multifacet/papers/ieeecomputer08_amdahl_multicore.pdf

[36] Miguel, M.: *Anax – An open source C++ entity system*. [Online; navštíveno 27.04.2017].
URL https://github.com/miguelmartin75/anax/tree/master

[37] Nguyen, S.: *C++ port of Artemis Entity System Framework*. [Online; navštíveno 27.04.2017].
URL https://github.com/vinova/Artemis-Cpp/tree/master

[38] Nicholson, B.: *Games: Playing with Threads*. [Online; navštíveno 20.04.2017].
URL http://www2.epcc.ed.ac.uk/downloads/lectures/BenNicholson/BenNicholson.pdf

[39] Norvig, P.: *Approximate timing for various operations on a typical PC*. 2001, [vid. 13.04.2017].
URL http://norvig.com/21-days.html#answers

[40] Nystrom, R.: *Game Programming Patterns*, [Online; navštíveno 12.04.2017].
URL http://gameprogrammingpatterns.com/contents.html

[41] Papari, A.: *Artemis ECS*. [Online; navštíveno 27.04.2017].
URL https://github.com/junkdog/artemis-odb/tree/master

[42] Polášek, T.: *Entropy – Entity-Component-System library*. [Online; navštíveno 28.04.2017].
URL https://github.com/T0mt0mp/Entropy

[43] Romeo, V.: *Analysis of entity encoding techniques, design and implementation of a multithreaded compile-time Entity-Component-System C++14 library*. 2016, [Online; navštíveno 12.04.2017].
URL https://www.researchgate.net/publication/305730566_Analysis_of_entity_encoding_techniques_design_and_implementation_of_a_multithreaded_compile-time_Entity-Component-System_C14_library

[44] Saltares, D.: *Ashley ECS*. [Online; navštíveno 27.04.2017].
URL https://github.com/libgdx/ashley

[45] Shimp, A. L.: *Xbox One: Hardware Analysis and Comparison to Playstation 4*. 2013, [Online; navštíveno 14.04.2017].
URL http://www.anandtech.com/show/6972/xbox-one-hardware-compared-to-playstation-4/2

[46] Trancoso, P.: *How L1 and L2 CPU caches work*. 2014, [Online; navštíveno 14.04.2017].
URL http://www.cs.ucy.ac.cy/courses/EPL605/Fall2014Files/Chapter5-Memory-Cache2.pdf

[47] West, M.: *Evolve Your Hierarchy*, 2007, [Online; navštíveno 12.04.2017].
URL http://cowboyprogramming.com/2007/01/05/evolve-your-heirachy/

Přílohy

Příloha A

Obsah přiloženého paměťového média

Příloha B

Manuál

Příloha C

Použití knihovny