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 vývoj 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 a nevýhod. Následně se práce zabývá důvody, proč tuto metodu návrhu používat a jakým způsobem využití kompozice ovlivňuje výkon aplikací, ve vztahu k hardwarové architektuře moderních počítačů, se zaměřením na paměťovou hierarchii.

V práci je dále navržen entitní systém, který umožňuje dynamickou kompozici entit za běhu aplikace z předem definovaných komponent. Výsledný systém je navržen s ohledem na snadnou paralelizaci vykonávaných akcí a umožňuje uživatelské rozšíření ve směru datových struktur k uchovávání komponent.

Výsledkem této bakalářské práce je multiplatformní knihovna v programovacím jazyce *C++*, pojmenovaná *Entropy*, která vývojářům zpřístupňuje návrh aplikací pomocí *ECS* paradigmatu.

Abstract

The goal of this bachelor thesis is the design and implementation of a library for entity management which allows software development using the *Entity-Component-System* paradigm. Analysis of the current state of software design is presented, including comparison of techniques used for entity representation. Potential advantages of composition are discussed from the hardware point of view, primarily based on 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 to allow for easy parallelization of performed actions and user customizability of component data structures.

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

Klíčová slova

Komponentní systém, Entitní systém, Entity-Component-System, ECS, Kompozice, Data-oriented design, Paralelismus, C++, Návrh her

Keywords

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

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
9. 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*, použitý při 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 způsob návrhu pomocí *Entity-component-system* paradigmatu, který v základu používá princip *kompozice* místo *dědičnosti*.

Jelikož je tato metoda vcelku nová, její širší využití začíná až v posledních letech. Existuje mnoho návrhů, jak by architektura *entitního systému* založeného na kompozici měla vypadat. Nejčastěji je návrh rozdělen do tří celků – *entity*, *komponenty* a *systémy*. Výhodou je striktní oddělení logiky od dat a možnost uložení dat do souvislých datových struktur, čímž je dosaženo efektivnější využití výpočetního hardware.

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

- Možnosti paralelního provádění transformací dat.
- Efektivní využití hardwarových prostředků.
- Jednoduché programovací rozhraní a integrace do existujících projektů.

Mezi další požadavky patří efektivní práce s velkým množstvím *entit*, kde většina z nich nemusí být aktuálně používána a také efektivní řešení komunikace mezi jednotlivými *komponentami* a *entitami*.

Výsledkem této práce je multiplatformní knihovna pro programovací jazyk *C++*, umožňující vývoj aplikací za použití *Entit-Component-System* paradigmatu.

Práce je rozdělena do čtyř logických celků – teoretická část, návrh systému, implementace a vyhodnocení. Kapitola teorie (2) popisuje aktuální stav návrhu software, z pohledu vývoje her a důvody vzniku *ECS* paradigmatu. Nejdříve jsou zde přiblíženy požadavky na *entitní systémy*, následuje porovnání aktuálně používaných způsobů reprezentace *entit* a srovnání jejich výhod a nevýhod. Závěr se zabývá způsoby, jak využít více-jádrové hardwarové konfigurace. Kapitola návrh (3) zahrnuje kompletní návrh *entitního systému*, který je obecný – nezávislý na implementačním jazyce. Úvod obsahuje přehled celého systému, další sekce pokračují podrobnějším popisem jeho částí a rozborem různých způsobů řešení. Následující kapitola – implementace (4) – je věnována implementaci výše zmíněného návrhu v programovacím jazyce *C++*. Závěrečná kapitola (5) shrnuje vlastnosti výsledného systému z pohledu výkonosti a srovnává jej s podobnými volně dostupnými knihovnami.

Kapitola 2

Entitní systémy a jejich využití

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 *ECS* paradigmatu a definice hlavních pojmů – *entita*, *komponenta* a *systém*. Na závěr se věnuje metodám paralelizmu, které budou zvažovány pro následující návrh entitního systému.

2.1 Entitní systémy

Entitní systém je část herního enginu, který zprostředkovává správu *entit* – objektů ve virtuálním světě. Alternativní název pro *entitu*, který se také často používá, je *herní objekt* (*Game Object* [13]). Primární funkcí *entit* je propojení jednotlivých modulů a částí aplikace, které jsou vyvíjeny odděleně – např. herní logika a simulace fyziky.

2.2 Typické problémy entitních systémů

Stále se zvyšující 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* [27, 34], které obsahují zkušenosti a prověřené způsoby návrhu software. Důležitou vlastností správného návrhu, je také modularita – oddělení částí systému – která umožňuje velkému množství programátorů pracovat na jednom projektu. Základem *Entity-Component-System* paradigmatu je striktní oddělení logiky a dat, čímž je umožněna vysoká úroveň modularity.

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. přidání efektu z herní logiky. Přílišná provázanost však většinou znamená zvýšený výskyt programovacích chyb (*bugs*) [34]. Podrobněji se tímto zabývá sekce 2.5.

2.3 Architektura moderních počítačů

Důležitou součástí tvorby her je analýza cílových platforem, pro které je hra určena. Velkou výhodou je v dnešní době podobnost všech různých herních systémů. Stolní počítače,

ale i některé nové herní konzole (*Playstation 4* a *Xbox One* [38]), používají architekturu procesorů *x86* [19, 16]. Stále důležitější platformou jsou také mobilní zařízení, které většinou používají architekturu procesorů *ARM* [14]. Díky podobnosti paměťových hierarchií na těchto platformách je možné implementovat optimalizace, které vylepšují výkon výsledných aplikací na všech cílových systémech.

Moderní procesory dokáží velmi rychle vykonávat jednoduché instrukce a za použití různých mechanizmů (např. *pipelining* [10]) se stále zvyšuje počet instrukcí zpracovaných za jeden cyklus (*Instructions per Cycle* [7]). Problém nastává v případě, kdy procesor operuje s daty, které nejsou k dispozici v jeho registrech, jelikož rozdíl v rychlosti procesorů a přístupové době pamětí stále roste [39]. Kvůli těmto rozdílům existuje hierarchie procesorových vyrovnávacích pamětí (*cache*), jejichž cílem je minimalizace rozdílů rychlostí jednotlivých pamětí.

Právě díky rozdílům rychlostí jednotlivých typů pamětí [33] vznikají nové způsoby, jak navrhovat aplikace, které umožňují efektivně využívat hierarchii procesorových vyrovnávacích pamětí. Jedním z nich je *datově orientovaný* návrh, ze kterého vychází *ECS* paradigma.

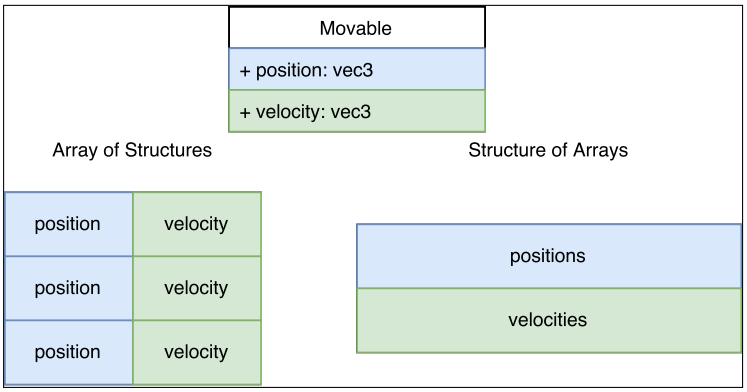
Mezi důležité parametry pro efektivní využití *cache* patří lokalita dat [5] a – v případě více-jádrových systémů – udržování jejich koherence [4]. Lokalita dat je primárně dělena na 2 typy – časová a prostorová. Časovou lokalitou je myšleno opakované používání stejných dat, kdy se při prvním použití data přesunou do vyrovnávací paměti a dále je k nim již přistupováno skrz rychlejší paměť. Načítání dat do *cache* je prováděno v blocích (*cache line*), jejichž velikost je specifická pro každý procesor. Program, který přistupuje k datům, jejichž vzdálenost v paměti není příliš velká (vejdou se do jedné *cache line*) využívá prostorové lokality dat, kdy výsledkem je méně přístupů do pomalejších pamětí. 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 tříd. Základní stavební jednotkou je v tomto případě třída – agregace hodnot a množiny operací nad ní proveditelnou. Toto umožňuje řešení problémů transformací objektů z reálného světa na objekty virtuální, se kterými dokáží lidé pracovat a zároveň jsou interpretovatelné překladači programovacích jazyků. Toto je však problematické pro moderní výpočetní hardware, který je velmi výkonný v případě opakovaného provádění jednoduchých datových transformací, ale má problémy s abstrakcí objektu složeného z více částí. Dalším problémem je využití pouze části dat, která daný objekt obsahuje, čímž jsou tato data zbytečně načtena do vyrovnávací paměti a zabírají cennou kapacitu.

Kvůli výše zmíněným problémům vzniká *datově orientovaný návrh*[25] (*DOD*), jehož cílem (na rozdíl od *OOD*) je analýza dat, se kterými aplikace pracuje a následovné použití jednoduchých transformací. Nevýhodou využití *OOD* je snížení čitelnosti výsledného kódu a složitější transformace návrhu aplikace – který řeší daný problému – ve výsledný program. Základní myšlenkou je oddělení dat a operací nad nimi, čímž je umožněno efektivnějšího využití procesorových vyrovnávacích pamětí.

Základem *DOD* je důkladná analýza aplikačních dat – jejich obor hodnot, transformace, definice vstupních dat, požadovaný výstup atp. Často je využíváno datové struktury typu pole, jejíž vlastnosti umožňují rychlou iteraci. Mezi výhody *DOD* patří také lepší lokalita dat, čímž je zvýšena datová propustnost výsledné aplikace. Díky těmto vlastnostem je *DOD*



Obrázek 2.1: Příklad transformace pole struktur na strukturu polí.

často využíván při návrhu software, který vyžaduje vysokou úroveň optimalizace – v tomto případě při návrhu her[8, 20].

Častým příkladem rozdílů mezi *OOD* a *DOD* je transformace pole struktur (*AOS*) v strukturu polí (*SOA*), jejíž ilustraci lze vidět na obr. 2.1.

2.5 Reprezentace entit

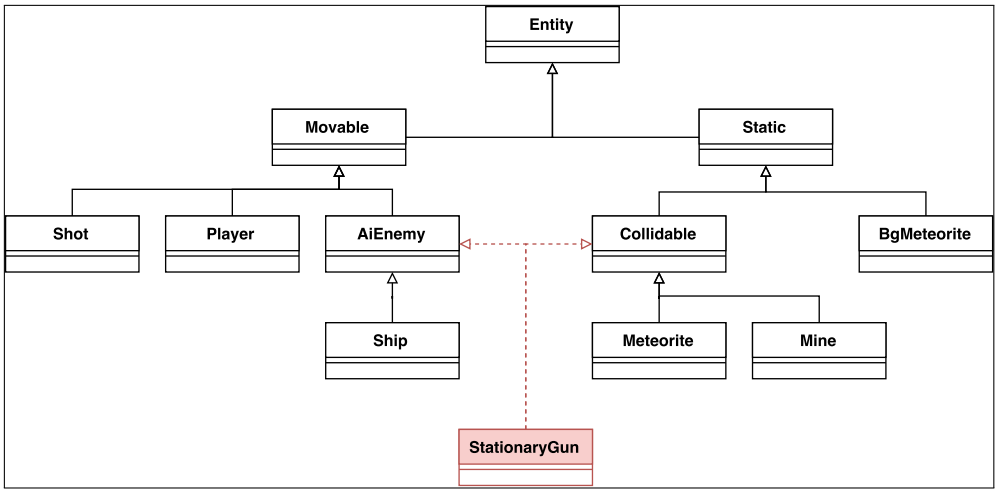
Tato sekce obsahuje rozbor nejpoužívanějších způsobů reprezentace *entit* [40] a jejich *chování*. Pod pojmem *entita* je myšlen objekt ve virtuálním světě. *Chování*, nebo *aspekt*, definuje operace, které *entita* dokáže provést – pohyb, vykreslení, kolize atp.

Pro ilustraci návrhu pomocí jednotlivých metod je použit ukázkový návrh jednoduché 2D „shoot'em up" hry, kde prezentovaný návrh není nutně optimální, ale ukazuje vlastnosti dané metody. Dílčí reprezentace entit 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

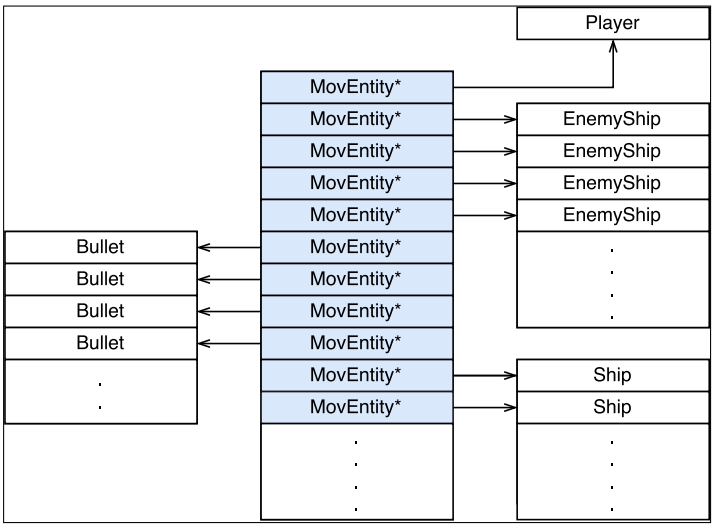
Pod pojmem *objektově orientovaná hierarchie* (*OOH* [34]) je míněn způsob skládání nových typů entit za využití *dědičnosti*. Příklad hierarchie, navržené pro ukázkovou 2D hru, lze vidět na obr. 2.2. V kořenu stromu hierarchie je, v případě *OOH*, bázová třída, která umožňuje uniformní skladování entit. Konkrétní entity, které existují v herním světě, jsou listy ve stromu 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, kdy u prvního z nich je akce specifikována přímo v supertřídě. Druhým je potom použití *polymorfizmu*, kdy děděné třídy specifikují pouze to, že akce by měla existovat, ale definici nechávají na podtřídě.



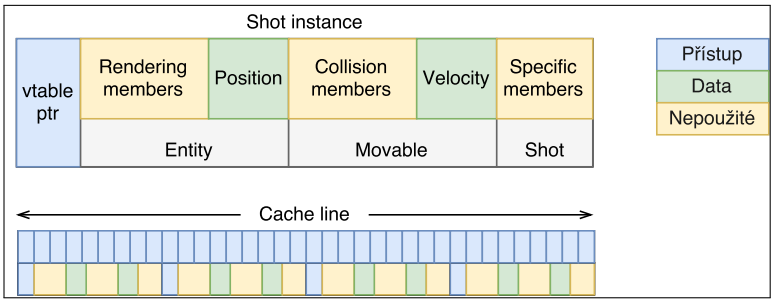
Obrázek 2.2: Příklad objektově orientované hierarchie.

Hierarchii dědičnosti, která je v tomto příkladě použita lze vidět na obr. 2.2. 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 – hráč do nich nemůže „narazit“. V tomto případě je nutné rozdělit nepohyblivé *entity* do dvou podtypů, čímž dochází k duplikaci kódu ve třídách **Meteorite** a **BgMeteorite**. Dalším příkladem problémů s *OOH* je přidání „nepohyblivé zbraně“ – třídy **StationaryGun**. Zbraň potřebuje umělou inteligenci, která ji bude ovládat, ale zároveň je také nepohyblivá. Tyto problémy lze u prezentované hierarchie opravit, ale pro větší projekty to již nemusí být možné.



Obrázek 2.3: Příklad obsahu paměti, při využití *objektově orientované hierarchie*.

Pro příklad využití paměti je využit seznam pohybujících se entit, jejichž pozice musí být s každým snímkem hry aktualizována, aplikováním jejich rychlosti. Ilustraci možné organizace paměti lze vidět na obr. 2.3. Středem je pole ukazatelů, které obsahuje entity, které je potřeba aktualizovat. Instance jednotlivých konkrétních typů jsou uloženy v oddělených polích.



Obrázek 2.4: Využití vyrovnávací paměti při použití *objektově orientované* entitní hierarchie. Uvedený příklad pracuje s objektem „střely“.

Při operaci aktualizace je iterováno přes hlavní pole ukazatelů a na každý objekt je aplikována metoda, která aktualizuje jeho pozici přičtením rychlosti. Postup práce s hlavní pamětí a hierarchií vyrovnávacích pamětí je následující[1]. Prvním krokem je načtení bloku paměti, který obsahuje část pole ukazatelů, do vyrovnávací paměti. Následuje dereference prvního z ukazatelů, která zapříčiní přesun dalšího řádku paměti, který již obsahuje iterované objekty. Ilustrace obsahu paměti *cache* lze vidět na obr. 2.4. Entity kromě požadovaných informací – pozice a rychlost – obsahují také data, která nejsou použita. Výsledkem je neoptimální využití procesorových *cache* [8].

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

Mezi výhody *OOH* patří:

- 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.

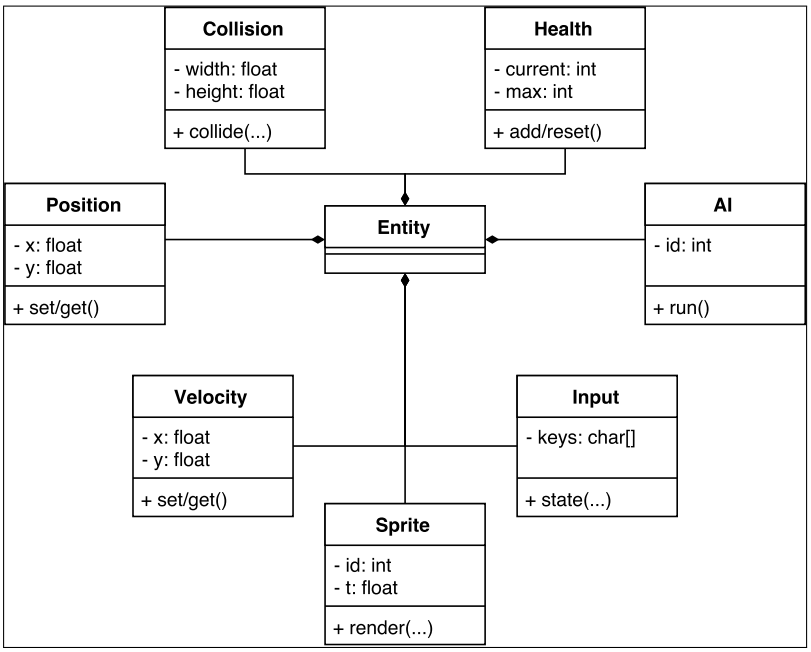
2.5.2 Objektově orientovaná kompozice

Pod pojmem *objektově orientovaná kompozice* (*OOC* [34]) je myšlena tvorba entit z menších částí – komponent – kde komponenty obsahují data a množinu proveditelných akcí. Entita

[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.

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

je při použití *OOC* reprezentována jako kontejner, který obaluje seznam komponent (*kompozice*). Příklad návrhu množiny komponent lze vidět na obr. 2.5, kontejnerem je v tomto případě třída **Entity**.



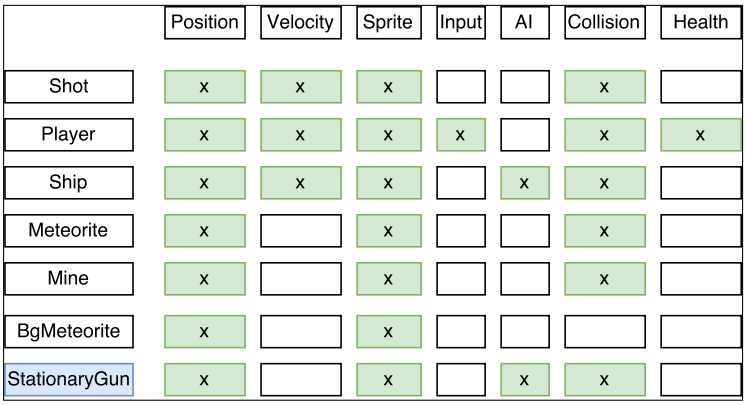
Obrázek 2.5: Příklad objektově orientované kompozice.

Jelikož komponenty obsahují kromě dat také akce, lze množinu akcí, kterou lze nad výslednou entitou provést, definovat jako sjednocení množin akcí jednotlivých vlastněných komponent. Tato vlastnost umožňuje dynamickou tvorbu entit za běhu aplikace, jejichž chování je definováno přítomnými komponentami.

Tato metoda je často používána[3] v návrhu software a je mezikrokem od *objektově orientované hierarchie* k *datově orientované kompozici*.

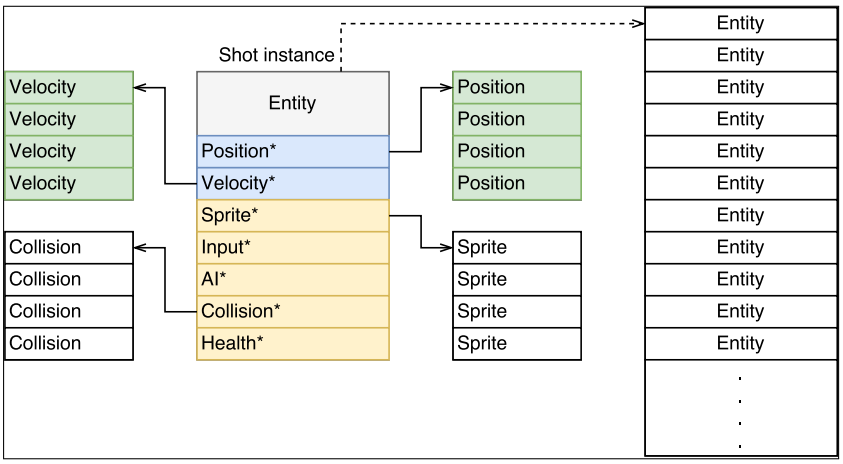
Příkladem využití těchto komponent, pro implementaci stejné množiny konkrétních entit, jako v případě použití *OOH*, lze vidět na obr. 2.6. Oproti využití dědičnosti se zde již lze jednoduše vyhnout problému s umístěním entit do stromu dědičnosti. Přidání typu **StationaryGun** je již také bezproblémové,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

[3] „Composition over inheritance“



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“.

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[4]. Při použití tohoto 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, nevýhodou je využití paměti pro větší množství druhů komponent. Ilustraci této implementace lze vidět na obr. 2.7.



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

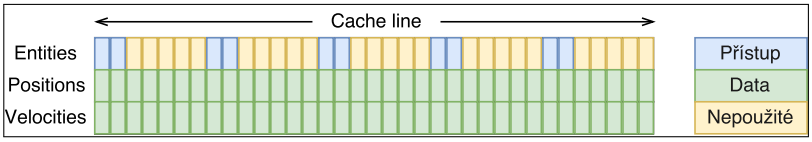
Jako příklad práce s pamětí je opět použita iterace nad seznamem entit, u kterých je nutno provést aktualizaci pozice přičtením jejich rychlosti. V tomto příkladě je použita implementace pomocí statického seznamu ukazatelů, podle obr. 2.7.

Podobně, jako při použití *OOH*, je zde iterováno nad polem entit, který však tentokrát obsahuje již jednotlivé instance typu **Entity**[5]. Využití vyrovnávací paměti, za stejných předpokladů, jako v případě *OOH*, je následující. Nejdříve je načten seznam entit a jejich

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

[5]Předpokládá se, že všechny entity obsahují komponenty typu **Position** a **Velocity**.

ukazatelů na jednotlivé komponenty. Požadovaná operace potřebuje pouze ukazatele na komponenty typu **Position** a **Velocity**, ostatní jsou v tomto případě zbytečné. Následuje přístup k požadovaným komponentám skrz ukazatele v první entitě. Tímto je načten blok paměti, pro každou komponentu, do vyrovnávací paměti. Po provedení první operace jsou již následující komponenty přístupné z vyrovnávací paměti.



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

Komunikace mezi jednotlivými komponenty je při použití *objektově orientované kompozice* problematické. Jelikož jsou jednotlivé komponenty samostatné objekty, nepřístupné z ostatních komponent stejné entity, není mezi nimi možná přímá komunikace. Jedním možným řešením je přidání systému zpráv, který má přístup k celé entitě, včetně jejích komponent. Dílčí komponenty se následně mohou dotazovat na typy zpráv, které je zajímají, nebo je možné využít návrhového vzoru *pozorovatel*. Dalším řešením je umožnit komponentám přímou komunikaci předáním ukazatelů na entitu, nebo na další komponentu, se kterou může daná komponenta komunikovat. Nevýhodou v případě přímé komunikace je bližší vazba výsledných komponent, což může způsobit chyby při změně některé z komunikujících stran.
Mezi výhody *OOC* patří:

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

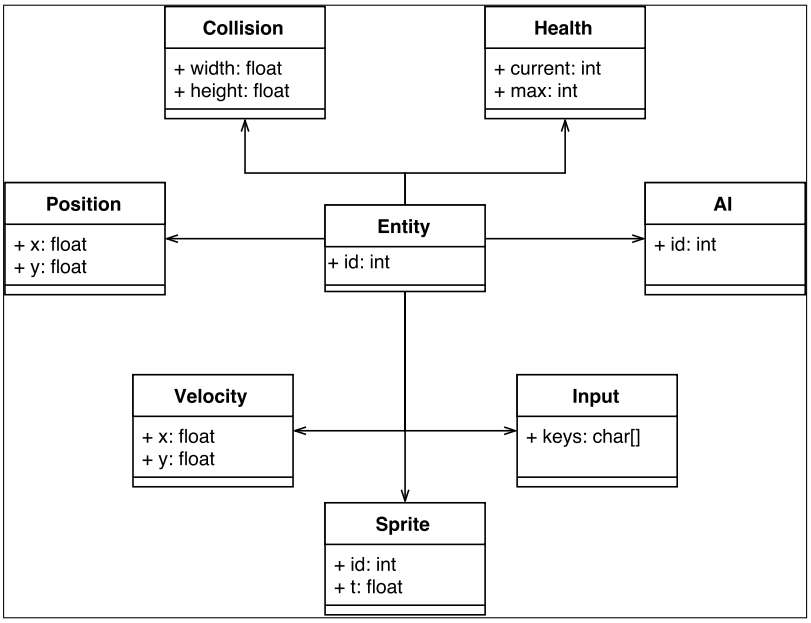
Její nevýhody jsou:

- 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 separaci dat a logiky. Podobně, jako *objektově orientovaná kompozice*, *DOC* rozděluje entity do menších části – komponent – což umožňuje vyšší modularitu výsledného návrhu. Oproti *OOC* však dochází ke kompletní separaci logiky a dat, kdy komponenty již neobsahují kód akcí[6].

Množinu akcí, které 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 potřeba lepší kontrola nad přístupy k jednotlivým komponentám, lze například využít *systémy*, které implementují akce nad komponentami[7]. Tento způsob je využit i v návrhu *entitního systému*, který je výsledkem této práce.

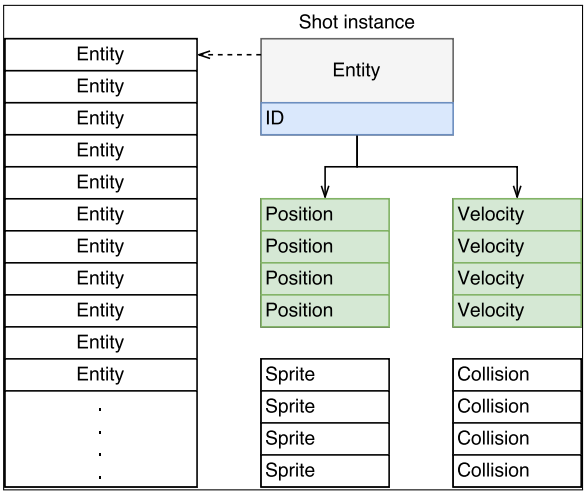


Obrázek 2.9: Příklad datově orientované kompozice a návrh komponent, které tvoří výsledné entity.

Příklad návrhu entity a komponent lze vidět na obr. 2.9. Mezi důležité změny, oproti *OOC*, patří veřejný (*public*) přístup k atributům komponent. Entita je v tomto návrhu reprezentována číslem – *identifikátorem* – skrz který 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.

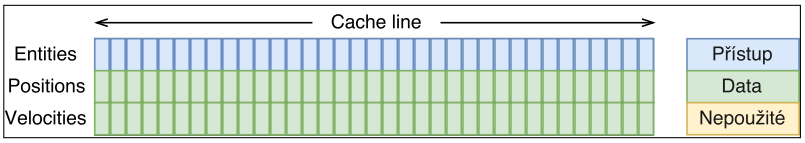
[6] Logikou je v tomto případě myšleno kód, který pracuje na vyšší úrovni, než pouze s daty dané komponenty. Blíže k tomuto tématu v sekci 2.6

[7] Tato metoda je blíže popsána v sekci 2.6



Obrázek 2.10: Obsah paměti u datově orientované kompozice. *Identifikátor* jednoznačně mapuje entity na jednotlivé instance komponent.

Organizaci paměti, se kterou dále tento příklad pracuje, lze vidět na obr. 2.10. Entity jsou uchovávány v homogenním poli. Jednotlivé identifikátory entit je možné mapovat na komponenty přímou indexací polí komponent.



Obrázek 2.11: Využití procesorové vyrovnávací paměti při použití datově orientované kompozice.

Pro ukázku práce s vyrovnávací pamětí je opět použit příklad iterace nad seznamem entit, které se mají posouvat v prostoru. 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[8], je možné po načtení řádku do vyrovnávací paměti pro dané dva typy komponent již následující akce provádět přímo nad pamětí *cache*. Obrázek 2.11 obsahuje ilustraci stavu vyrovnávací paměti, po provedení první operace.

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.

[8] Za předpokladu, že identifikátory všech iterovaných entit nedovolují díry v poli komponent.

Mezi výhody *DOC* patří:

- Efektivní skladování entit, které jsou reprezentovány identifikátorem.
- Lepší využití vyrovnávacích pamětí.
- Možnost definování nových typů entit za běhu, použitím kompozice.
- Odstraněna duplikace kódu a akumulace nechtěného stavu.
- Možnost efektivní implementace akcí, mimo komponenty.
- Opakované provádění jednoduchých akcí, nad seznamy komponent, umožňuje efektivní využití instrukční vyrovnávací paměti.

Její nevýhody jsou:

- Chybí podpora v programovacích jazycích[9].
- Náročná implementace.

Mezi neutrální vlastnosti patří:

- Ztráta abstrakce při práci s komponentami.
- Komponenty jsou čistá data, čímž je umožněna *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 implementaci chování, se nazývají *Entity-Component-System*[10]. Tato část obsahuje popis konceptů, se kterými tento typ entitního systému pracuje a vlastností, které z nich vyplývají.

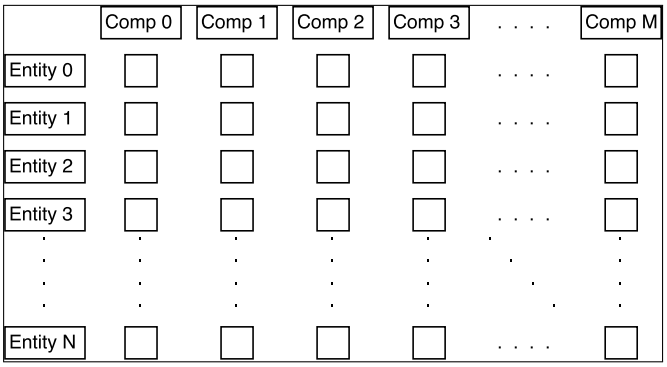
2.6.1 Motivace a koncepty

Entity-Component-System (*ECS* [18, 17]) je návrhové paradigma [28], založené na kompozici a striktní separaci logiky (*systémy*) a dat (*entity*, *komponenty*). *ECS* vychází z principů *datově orientovaného* návrhu, díky čemuž dokáže efektivně využívat moderní výpočetní hardware. Výsledný systém umožňuje vyšší úroveň modularity a dynamické změny aplikace za běhu.

Entita je základním stavebním blokem [28], který lze přirovnat k objektu z *objektově orientovaného* návrhu. Na rozdíl od objektů nejsou však entity tvořeny z předem definovaného vzoru (např. třídy), ale jejich funkce – chování a data – je definována množinou komponent, které daná entita vlastní. 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.

[9] Vzniká programovací jazyk, který bude tento typ kompozice podporovat [23].

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

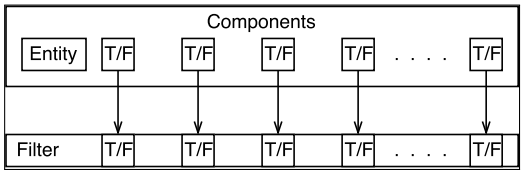


Obrázek 2.12: Entita reprezentována jako řádek v databázi. Primárním klíčem je identifikátor entity.

Komponenty jsou základními nosiči dat v *ECS*. Často používaným názvem pro komponenty, je také „aspekt“ [28], jelikož po přiřazení komponent k entitám mají komponenty další funkci – určují množinu chování entit. Kromě explicitních dat obsažených v komponentách existuje také implicitní informace o tom, zda daná entita obsahuje (má přiřazen) určitý typ komponenty. Každá entita má k sobě přiřazeno 0 a více[11] komponent daného typu, kde přítomnost komponenty znamená, že entita má definovaný daný aspekt. V přirovnání *ECS* k *relační databázi*, lze o komponentách smýšlet jako o sloupcích (obr. 2.12) tabulky entit.

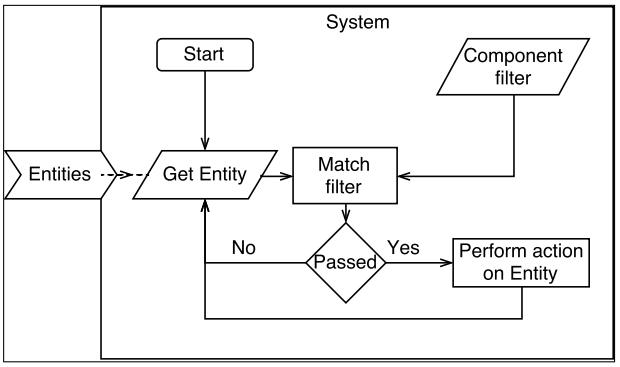
Pokud bychom pokračovali s přirovnáním k *OOP* – entity jsou jednotlivé objekty, komponenty umožňují polymorfizmus – potom *systémy* implementují chování, nebo metody jednotlivých komponent. Pro mnoho různých typů dat, kde každý typ vyžaduje vlastní chování, je *OOP* výhodné, protože je možné data i 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 [28].

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



Obrázek 2.13: Filtr operuje nad entitou a informací o přítomnosti komponent.

[11] Častým omezením, které se v reálných implementacích *ECS* objevuje, je omezení maximálního počtu komponent na 1.



Obrázek 2.14: Vstupem *systému* jsou entity, ze *systém* kterých vybere pouze ty, které mají odpovídající komponenty. Na vybraných entitách provádí požadované akce.

2.6.2 Vlastnosti

Důvodů, proč se *ECS* 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, čímž je vylepšena efektivita využití vyrovnávacích pamětí. Lokalita dat se vztahuje i na instrukce[12] 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“[13] 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. Tímto je umožněna vyšší úrovně paralelizace celého systému, 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 mezi jádra procesoru 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* [37])

[12] *Instrukční cache* [39, 26]

[13] Kód, který se stará o interakce virtuálních objektů ve hře.

2.7.1 EntityX

EntityX [21] je implementace entitního systému v programovacím jazyce *C++*, která silně využívá vlastností standardu *C++11* a *šablonového metaprogramování* [24]. 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 [35] je původně knihovna pro programovací jazyk *Java*, ale existuje i mnoho implementací pro další jazyky, např. *C++* (*ArtemisCpp* [31]). 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.

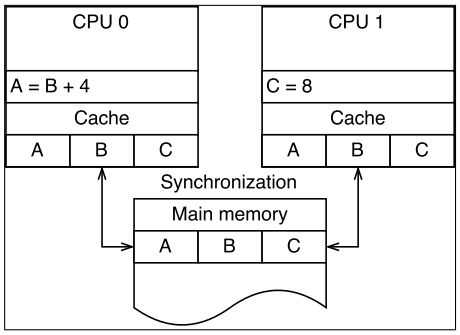
2.7.3 Anax

Kromě základní implementaci vlastností *ECS*, obsahuje knihovna **Anax** [30] 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

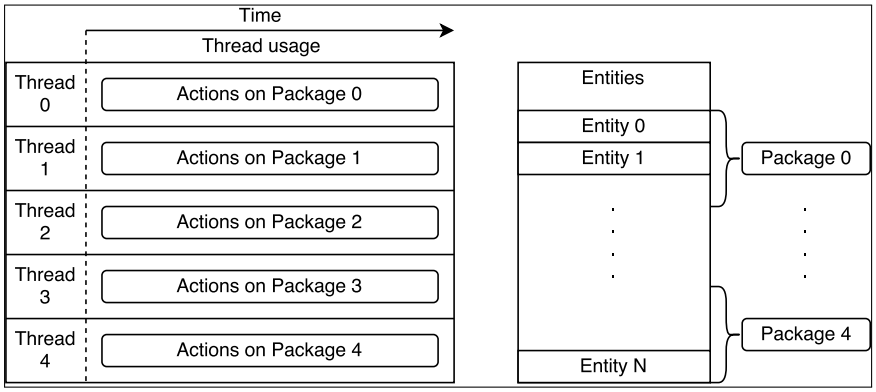
Vývoj moderních procesorů se ubírá směrem zvyšování počtu výpočetních jader [15] a proto i moderní aplikace musí být schopné tato jádra využít. To je zvláště pravda pro vývoj her, které se snaží s hardware vytěžit maximální výkon. Paralelizace aplikací, jejichž části jsou úzce provázané, je obzvláště problematické a proto vznikají nové způsoby jak tento typ software navrhovat – jedním z nich je právě *ECS*.

Paralelizmus lze obecně rozdělit na 2 typy – datový a úkolový [9]. Při použití datového paralelizmu je prováděna jedna akce na různých datech (obdoba *SIMD* u procesorů). Opačným přístupem je paralelizmus úkolový [11], u kterého mohou běžet různé operace nad stejnými (*MISD*), nebo i různými (*MIMD*) daty. 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 [32].



Obrázek 2.15: Udržování koherence vyrovnávací paměti a problém falešného sdílení („false sharing“ [1]) . 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* [4] a skrz to problém, který se nazývá „false sharing“ [1]. 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 [14], 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.



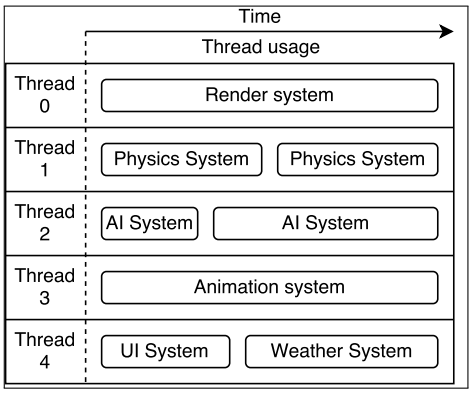
Obrázek 2.16: Paralelizmus uvnitř systémů je příkladem datového paralelizmu [9].

Entity-Component-System paradigma je výhodné z pohledu více-jádrových systémů, díky možnosti téměř [29] dokonalé paralelizace na několika úrovních [22]. První úrovní paralelizace, kterou *ECS* umožňuje, je datový paralelizmus, kdy se uvnitř systému množina všech entit, které odpovídají požadavkům, rozdělí na různá jádra, která mohou dané akce provádět odděleně [15]. Ilustrace principu, jak tento způsob pracuje lze vidět na obr. 2.16.

[14] Paměť je součástí stejného řádku vyrovnávací paměti.

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

Dalším způsobem paralelizace je možnost spouštět jednotlivé systémy na oddělených vláknech (obr. 2.17). Tato metoda je ovšem použitelná pouze pro množinu systémů, kde nemůže nastat situace, kdy několik z nich přistupuje ke stejným komponentám.



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

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 však vhodné – při paralelním přístupu k aktuálnímu stavu herního světa – aby všechny systémy dokázaly přečíst původní data beze změn, lze funkci systémů změnit následujícím způsobem. Systémy mohou místo zápisu nových hodnot do stejných komponent (vstupů) zapsat výsledek do „následujícího stavu“ [32]. Při generování nového stavu je nutné určit okamžik, kde se následující stav stane stavem aktuálním, čímž se celý virtuální svět „posune“ v čase. Výhodou tohoto přístupu je, že několik vláken může přistupovat (čtení i zápis) ke stejným komponentám.

Kapitola 3

Návrh herního entitního systému

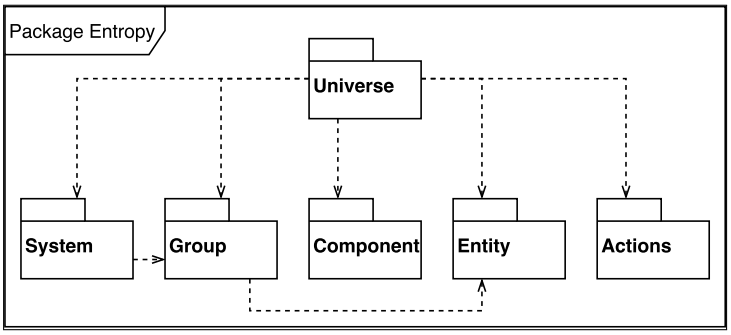
Obsahem této kapitoly je popis návrhu entitního systému založeného na *ECS* paradigmatu. Nejdříve je prezentováno základní rozdělení systému na jednotlivé části a komunikace mezi nimi. Následuje popis návrhu podsystémů – *komponenty*, *entity* a *systémy*. Každá část obsahuje představení návrhu a jeho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

Kromě základních funkcí, které pramení z *ECS* paradigmatu, jsou při návrhu zohledněny také požadavky na umožnění paralelního vykonávání akcí. Navržený systém podporuje paralelizaci na úrovni *entit*, *systému*, ale i formou generování „následujícího stavu“.

Při návrhu entitního systému byly použity poznatky z teoretické části této práce a *open-source* implementace entitních systémů založených na *ECS* paradigmatu – *EntityX* [21], *ArtemisCpp* [31], *Anax* [30], *Artemis* [35] a *Ashley* [37].

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.

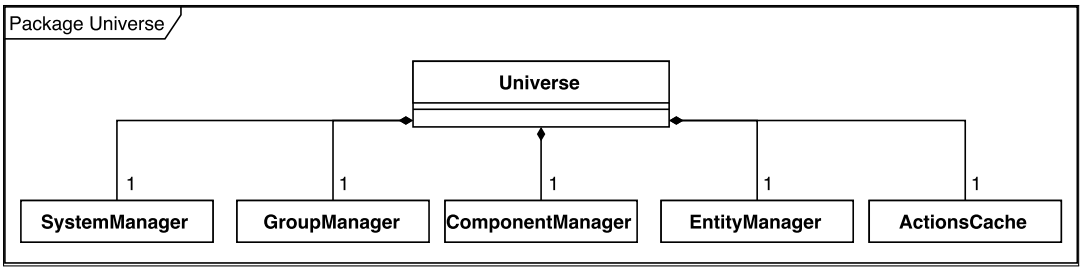


Obrázek 3.1: Rozdělení entitního systému do modulů.

Mezi tyto moduly patří:

- Modul **systémů** – Spravuje registraci *systémů* a jejich vazbu na *skupiny entit*.
- Modul **skupin** – Zajišťuje správu *skupin entit*, nad kterými *systémy* iterují, a udržuje jejich konzistenci.
- Modul **komponent** – Umožňuje registraci *komponent* a správu *nosičů*.
- Modul **entit** – Spravuje *entity* a udržuje jejich metadata – přítomnost *komponent*, aktivitu apod.
- Modul **akcí** – Obsahuje funkcionalitu odložených akcí a generaci „následujícího stavu“.

Ovládání celého entitního systému je zastřešeno třídou **Universe** (obr. 3.2), která umožňuje uživateli přístup k jednotlivým podsystémům. Hlavní funkcí třídy **Universe** je zprostředkování komunikace mezi moduly – např. přidání komponenty má dvě části, skutečná operace nad *nosičem komponent* a úprava metadat (obr. 3.4 a obr. 3.7).



Obrázek 3.2: Třída **Universe** je rozhraním entitního systému.

3.2 Komponenty a jejich nosiče

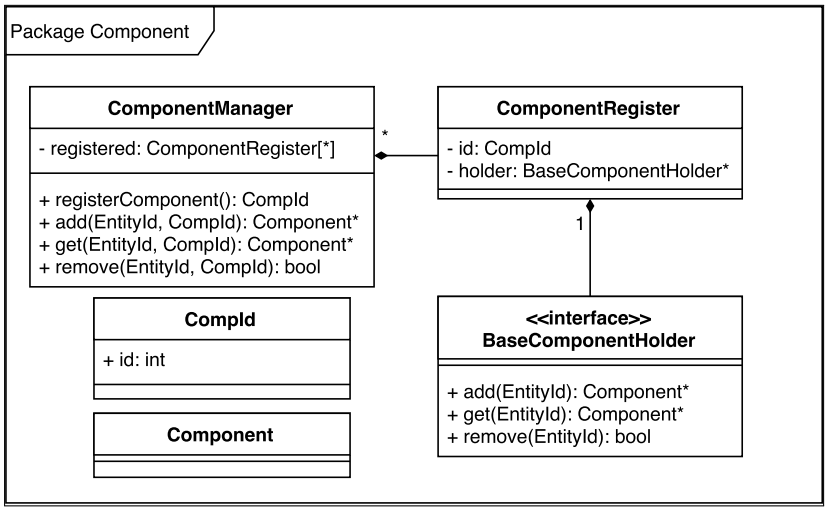
Komponenty jsou definovány jako základní datové jednotky v *ECS*, kde každá entita má k sobě přiřazeno 0-1 komponent daného typu. Komponenty, jak s nimi pracuje tento návrh, mohou být jakékoliv třídy, kdy jedinou požadovanou informací je typ *nosiče*. *Nosič komponent* je datová struktura, která obsahuje instance *komponent*. Každá komponenta má přiřazen unikátní identifikátor – **CompId** – tento identifikátor je neměnný po dobu běhu entitního systému. Komponenty by měly být pasivní datové struktury[1], 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 registraci komponent a jejich následnou asociaci 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**[2], která vytvoří mapování z daného typu na jeho registr (**ComponentRegister**). Kromě identifikátoru, obsahuje registr také *nosič* samotný.

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

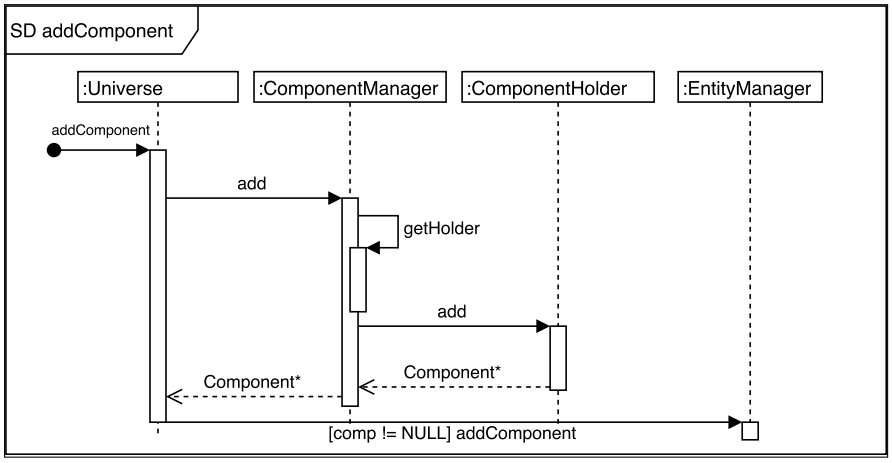
[2] Této metodě je třeba předat typ komponenty, která má být registrována, čehož lze docílit pomocí např. *template* (C++), nebo *generics* (Java)



Obrázek 3.3: Diagram tříd podsystému správy *komponent*.

Každý typ komponent je uložen ve vlastním kontejneru – tzv. nosič komponent. Nosičem je třída, která implementuje všechny požadované operace ze třídy **BaseComponentHolder** – přidání, odebrání a získání komponenty specifické entity. 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.



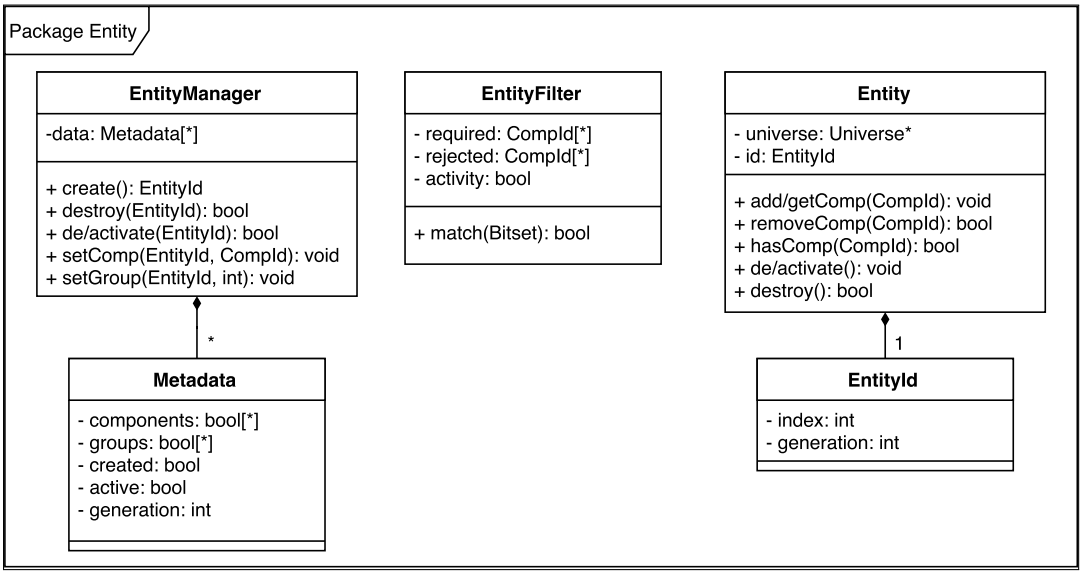
Obrázek 3.4: Sekvenční diagram přidání komponenty, pokračuje na obr. 3.7.

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, kterou se zabývá následující část kapitoly o návrhu.

3.3 Reprezentace entit

Entita, jako koncept z *ECS* paradigmatu, je objekt, složený z paměťově distribuovaných komponent. Pro každou entitu existuje jednoznačné mapování na její komponenty. Vyvstává tedy požadavek na jednoznačný identifikátor entit, který by plnil funkci *primárního klíče* v tabulce entit (obr. 2.12).

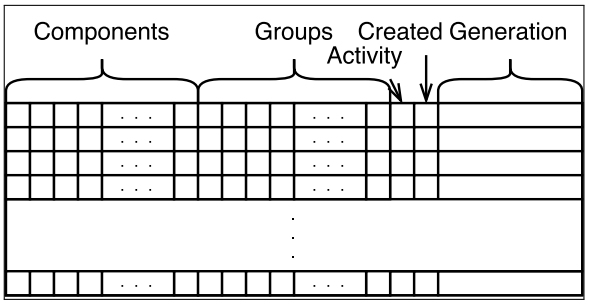
Pro účely tohoto entitního systému je identifikátor složen ze dvou čísel – ID a generace. ID je v tomto případě skutečný *primární klíč* a zároveň plní funkci indexu v tabulce entit. Generační číslo odlišuje různé generace entit, které zabíraly stejný řádek v tabulce (stejné ID) a je inkrementováno při každém smazání entity. Generační čísla identifikátorů, skrz které je přistupováno k entitnímu systému jsou porovnány s aktuální generací dané entity, což umožňuje rozpoznat přístup ke smazaným entitám.



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 (**Metadata**) 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[3].

[3]Indexace tabulky pomocí *ID* části identifikátoru.



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

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

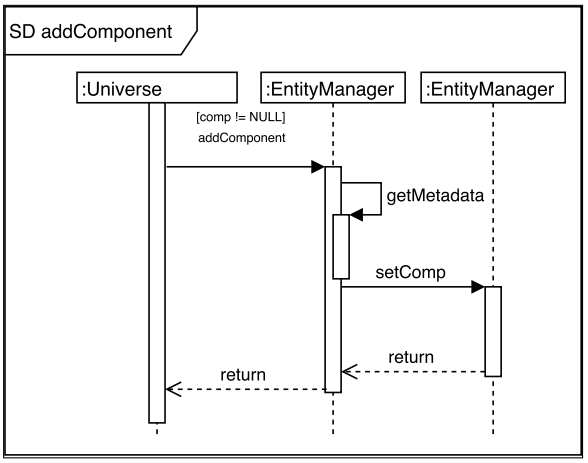
- Aktivita – Entita může být ve stavu aktivní, nebo neaktivní.
- Obsazenost – řádek je použitý, nebo prázdný.
- Přítomnosti každého registrovaného typu komponent.
- Přítomnost entity ve skupinách [4].

Důležitou součástí *ECS* je možnost pracovat pouze s entitami, které mají požadované komponenty („aspekty“). Z tohoto důvodu obsahuje správa entit i možnost filtrování (třída **EntityFilter**), které je prováděno pomocí seznamů komponent, jejichž obsah definuje, které komponenty entita musí, nebo nesmí obsahovat. Komponenty, které nejsou obsaženy ani v jednom seznamu jsou při filtrování ignorovány. Filtr obsahuje také další filtrovacím kriterium, které umožňuje specifikovat požadovanou úroveň aktivity entity – aktivní, nebo neaktivní.

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

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 *správy komponent*) úspěšná, je nutné upravit metadata.

[4] Více o skupinách v části. 3.4 .



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

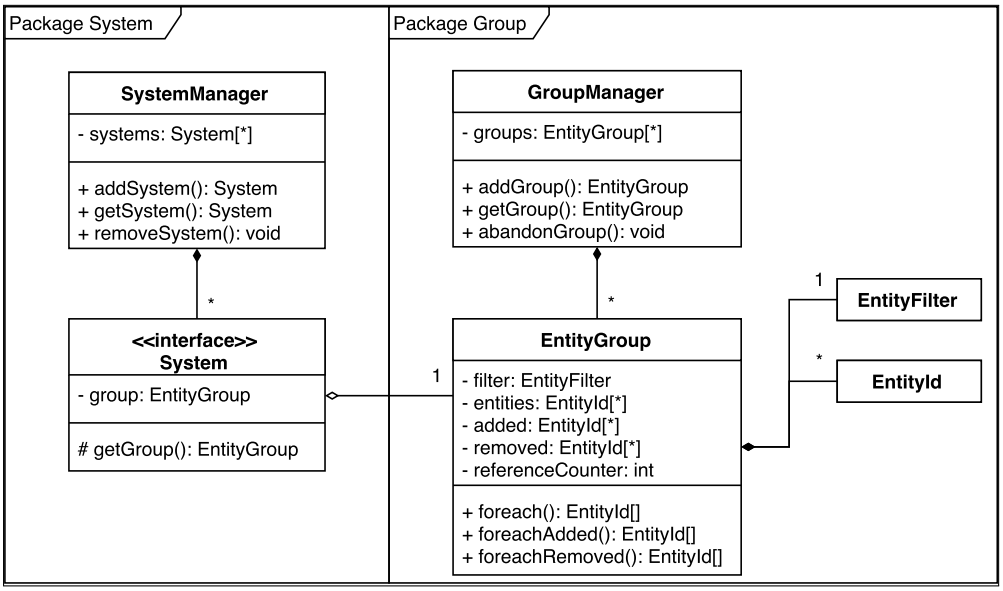
3.4 Systémy a skupiny

Systém, podle *ECS* paradigmatu, obsahuje pravidla pro výběr vhodných entit a akce, které na nich provádí. Entity jsou filtrovány 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 podsystému správy *systémů*.

Předpokladem, při návrhu tohoto entitního systému je, že i přeš vysoký počet entit, které se v entitním systému mohou v jeden okamžik nacházet, bude množství skutečně používaných entit[5] výrazně nižší. Z tohoto předpokladu vychází myšlenka *skupin*, které plní funkci 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 nevýhod použití *skupin* je nutnost udržovat *skupiny* aktuální, čímž je pověřen podsystém *správy skupin*. Díky potřebě udržování seznamu entitních identifikátorů způsobuje tento systém také vyšší spotřebu 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 *skupiny* používat.

[5] Těch, které budou vyhovovat některému ze systémů.



Obrázek 3.8: Diagramy tříd podsystémů správy *systémů* (vlevo) a správy *skupin* (vpravo).

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

Skupiny (**EntityGroup**) obsahují 3 seřazené seznamy entitních identifikátorů. Jejich významy jsou následující – přidané entity, odebrané entity a skutečný seznam všech vyhovujících entit. Primárním důvodem pro existenci prvních dvou je umožnit uživateli propojení entitního systému se zbytkem aplikace[6].

Důležitou vlastností *skupin* je možnost jejich přidávání a odebírání za běhu aplikace[7]. Toto umožňuje podsystém správy *skupin* (3.8 vpravo). 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 zastaven její běh).

3.5 Paralelní přístup

Díky modulárnímu návrhu entitního systému, který byl 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.

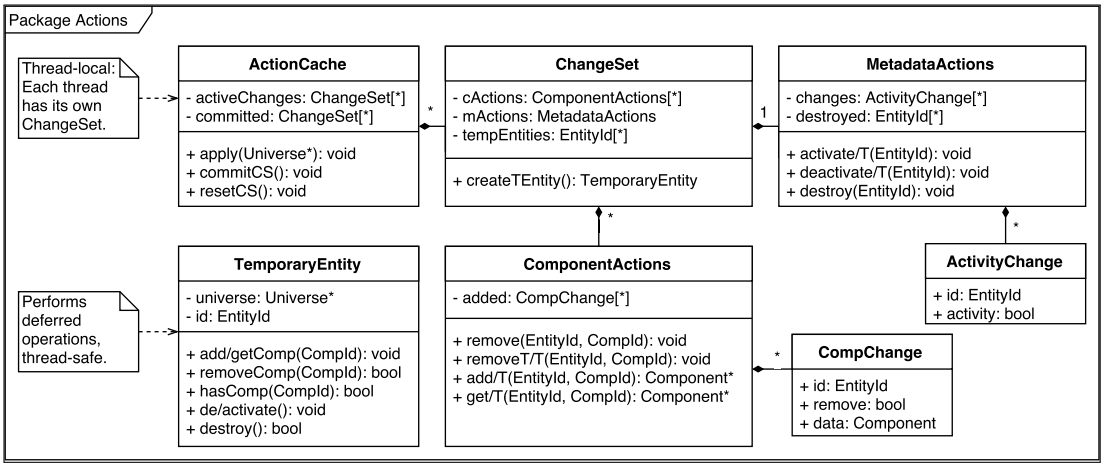
Paralelizmus na úrovni entit využívá *datového paralelizmu*, kde množina entit, nad kterou systém vykonává akce je rozdělena mezi požadovaný počet paralelních vláken, které mohou vykonávat akce odděleně. Tento typ paralelizmu je vhodný pouze v případech, kdy akce vykonávají transformace na entitách (a jejich komponentách), bez potřeby informací z ostatních entit.

[6] Např. registrace fyzikálních těles při simulaci *PhysX*.
[7] Některé systémy nemusí zpracovávat entity po celý běh aplikace.

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 při vysokém počtu operací zamčení a odemčení je možné ztratit výkon, který byl získán paralelním zpracováním[8]. 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, nad danou entitou, skutečné vykonání operace nad globálním kontextem bude odloženo, ale informace o této operaci bude přidána do *množiny změn* aktuálního vlákna.



Obrázek 3.9: Diagram tříd podsystému správy *akcí*.

Základem podsystému *množin změn*, jehož diagram lze vidět na obr. 3.9, jsou třídy **ActionCache** a **ChangeSet**. 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 jejich následné použití při normálních operacích. Pro operace, které jsou provedeny nad „reálnými“ entitami[9] je možno specifikovat cílovou entitu pomocí jejího identifikátoru, čehož není možné dosáhnout u „dočasných“ entit, které zatím nemají přiřazený globálně platný identifikátor. Jedním z možných řešení je zakázat provádění operací nad dočasnými entitami, nebo zrušit koncept dočasných entit kompletně a provádět operaci vytvoření entity okamžitě. Druhým řešením, které je součástí tohoto návrhu, je přiřazení speciálního typu identifikátoru „dočasným“ entitám, který bude později převeden na identifikátor „reálné“ entity.

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

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

Dočasné entity jsou reprezentovány třídou **TemporaryEntity**. Jejich rozhraní je velmi podobné normálním entitám, s tím rozdílem, že dovolují pouze odložené operace. *Množiny změn* samotné jsou reprezentovány třídou **ChangeSet**, která obsahuje odložené *akce*. *Akce* je možno rozdělit podle typu změn:

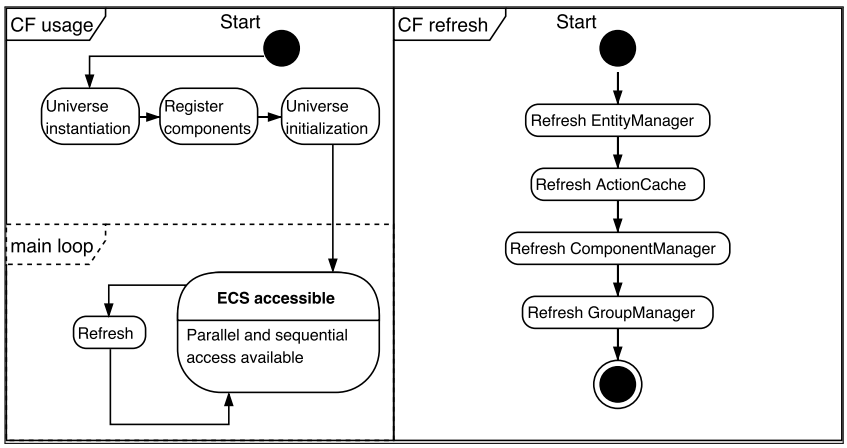
- Změny metadat – rušení entit, změna aktivity entit.
- Změny komponent – přidání/odebrání komponent, změna hodnot komponent.

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. 3.10 vlevo.



Obrázek 3.10: Digram toku řízení entitního systému.

Prvním krokem ve fázi inicializace je konstrukce (vytvoření instance) třídy **Universe**, jejíž součástí je uvedení jednotlivých podsystémů do výchozího stavu. V této fázi je jedinou možnou operací registrace komponent. Tato etapa je završena inicializací entitního systému, kdy je možné vytvoření struktury tabulky entit, jednotlivých nosičů komponent, apod.

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

Úkolem *obnovovací* fáze je posunutí entitního systému ze stavu „aktuálního“ do stavu „následujícího“ a dokončení operací, které by vedly k porušení konzistence. Tohoto cíle je dosaženo postupným voláním operace **refresh** na jednotlivé podsystémy, jejichž pořadí lze vidět na obr. 3.10 vpravo. 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*, čímž je *obnova* dokončena a entitní systém je opět uveden do fáze *iterace*.

Kapitola 4

Implementace

Tato kapitola obsahuje popis implementace výše popsaného návrhu entitního systému, založeného na *ECS* paradigmatu. Výsledná implementace obsahuje kompletní jádro, včetně práce s *entitami*, *komponentami* a *systémy*. Kromě těchto základních vlastností umožňuje také všechny tři typy paralelizace – *entitní*, *systémovou* a pomocí *množin změn* – přičemž tato kapitola obsahuje pouze zajímavější implementační detaily.

Dosavadní návrh entitního systému 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 identifikuje 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 musí, nebo nesmí obsahovat a požadovanou úroveň aktivity – 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++* [6], ve standardu *C++14* [2]. Jazyk *C++* byl vybrán díky dlouhodobému používání v herním průmyslu v roli primárního implementačního jazyka mnoha herních *enginů*. Mezi jeho další výhody patří optimalizované překladače nebo možnosti výběru z několika programovacích paradigmat, které podporuje. Jazyk byl použit ve standardu *C++14*, který kromě ulehčení práce s typy obsahuje také možnosti použití *šablon proměnných* (*variable templates* [3]).

Další důležitou vlastností, 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, přičemž je výsledné chování aplikace stejné[1].

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* [24], které umožňuje vyšší úroveň práce s typy.

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

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. Komponenty mohou také specifikovat, který nosič by měl být použit pro jejich uchovávání. 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[2] instance typu **ComponentRegister** (diagram na obr. 3.3). Tento registr obsahuje kromě informací o komponentě také instanci jejího nosiče. Díky *šablonovému metaprogramování* je možné přímé mapování komponenty na instanci jejího nosiče za překladu programu, což umožňuje překladači vyšší úroveň optimalizace.

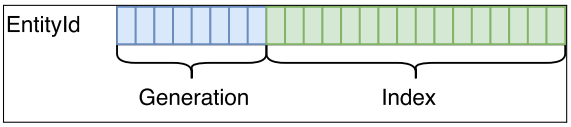
Knihovna obsahuje několik typů základních nosičů komponent. Prvním z nich je implementace využívající standardní mapu (**std::map**), která mapuje identifikátory entit přímo na instance jednotlivých komponent. Tento typ nosiče je výhodný pro komponenty, které nejsou příliš často používané. Dalším typem 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 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[3].

4.3 Správa entit

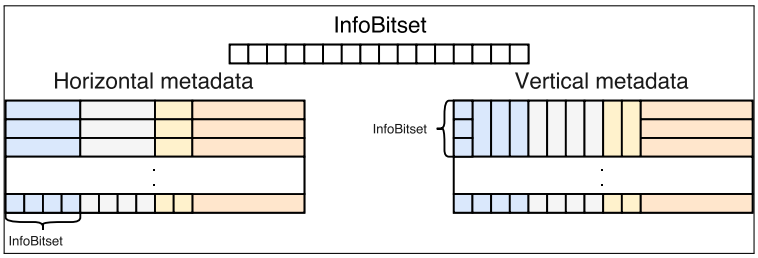
Entity jsou definované pomocí identifikátorů, složených ze dvou částí – index a generace. Index určuje číslo řádku v tabulce entit a je také používán pro mapování komponent na entity. Generace určuje specifickou entitu, která „obývá“ daný řádek tabulky entit. Tento identifikátor je implementovaný jako 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[4]. Validní identifikátor obsahuje indexové číslo, které není rovné nule. Speciálním případem identifikátoru je identifikátor dočasné entity, jehož generační číslo je rovno maximální hodnotě (255 pro výchozí nastavení).

[2] Díky tomu, že jsou registry statické proměnné.
[3] Značka je komponent, který neobsahuje žádná data.
[4] Tato nastavení lze změnit v hlavičkovém souboru.



Obrázek 4.1: Rozdělení identifikátoru entit na generaci a index.

Informace o entitách je uložena v tabulce entit, která obsahuje – aktuální generaci, přítomnost komponent, 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[5]. Velkou nevýhodou je však problematický paralelní přístup k jednotlivým bitům, 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 tato možnost 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.

[5]Filtrování je v tomto případě porovnání dvou čísel.

Jelikož by při opakovaném vytváření a mazání stejných entit mohlo dojít k rychlému vyčerpání generačních čísel a jejich přetečení[6], je seznam volných řádků implementován jako oboustranná fronta – smazané indexy jsou přidány na konec fronty, odběr je prováděn začátku fronty. Tímto je dosažena lepší distribuce generačních čísel a nižší frekvence přetečení jednotlivých čítačů.

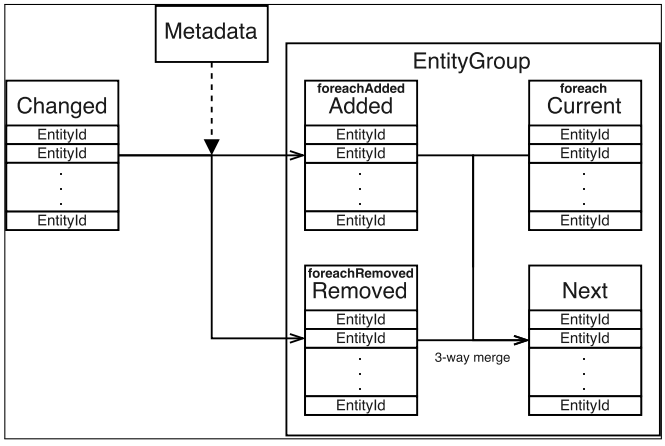
4.4 Systémy a skupiny

Systémy jsou vytvářeny uživateli knihovny, děděním základní třídy **System**. Následně je nutné *systém* přidat použitím metody **addSystem**, která také zaručí jeho inicializaci a přiřazení odpovídající *skupiny*. Doba života instance daného *systému* je spravována entitním systémem, čímž jsou zamezeny problémy při nevalidním smazání používaného *systému* ze strany uživatele knihovny.

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**).



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.

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

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[7]. 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. Normální iterace nad všemi vyhovujícími entitami vypadá následovně:

```
for (auto &e : foreach())
{
    PositionC *p{e.get<PositionC>()};
    MovementC *m{e.get<MovementC>()};
    p->x += m->dX;
    p->y += m->dY;
}
```

Paralelní iterace pro 4 vlákna nad množinou, které patří třetímu vláknu potom může vypadat takto:

```
auto parForeach = foreachP(4);
for (auto &e :parForeach.forThread(2))
{ /* ... */ }
```

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 *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.

Operace, jejichž provedení je odloženo na později, mají postfix „**D**“ („deferred“) a operace, které pracují s „dočasnými“ entitami jsou zakončeny „**T**“ („temporary“).

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 provést spojení seznamů jednotlivých vláken.

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

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. Přestože již umožňuje všechny funkce popsané v kapitole „návrh“, nelze ji považovat za plně dokončenou z důvodu přílišného rozsahu projektu. Aktuální stav implementace lze nalézt v *GIT repozitáři* [36], pod názvem „Entropy“. Následují možné směry, kterými bude její vývoj dále postupovat:

- **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. Dále také možnost tvorby entit po blocích a specifikace identifikátoru vytvořené entity.
- **Dynamické komponenty** – Díky způsobu implementace tabulky metadat je teoreticky možné přidávat nové typy komponent i za běhu aplikace, mimo inicializační fázi. Spojení této vlastnosti s vazbou na skriptovací jazyk by umožnilo případným herním návrhářům větší volnost, bez nutnosti opakovaného překládání zdrojového kódu v jazyce *C++*.
- **Volitelné použití skupin** – Pro *skupiny*, které obsahují téměř všechny entity je výhodnější iterovat nad všemi entitami. Dalším rozšířením skupin je umožnění specifikace požadované úrovně aktivity 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 prozatím není implementována.

Kapitola 5

Vyhodnocení

Tato kapitola obsahuje vyhodnocení výsledné implementace knihovny „Entropy“ a její porovnání vůči ostatním, volně dostupným knihovnám. Mezi testované scénáře patří operace s entitami, výpočetní *systém*, přesun entit mezi *skupinami*, nosiče komponent a paralelizmus.

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`, `\O2`), 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 GB DDR3 @ 1600 MHz
2	Intel Core i5-4670K CPU @ 3.40 GHz – 4 cores	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 Použité nástroje

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**

Pro tvorbu této práce byly také využity následující aplikace:

- **draw.io** – Tvorba diagramů, schémat a ilustrací.
- **R** – Grafy, které jsou použity v této kapitole.

5.3 Výkonnostní testy

Tato část obsahuje výsledky výkonnostních testů spuštěných ná výše popsané implementaci entitního systému. Hlavním cílem je předvést co aktuální implementace knihovny zvládá a jak ji lze porovnat 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, kterou navrhuje tato práce (nazvaná **Entropy**), jsou zde také testovány jiné *open-source* knihovny, mezi které patří:

- **Anax** [30]
- **ArtemisCpp** [31] – Implementace knihovny **Artemis** [35] v jazyce *C++*.
- **EntityX** [21] – Obsahuje i pokročilejší funkce, např. systém událostí. Z těchto knihoven je **EntityX** nejvíce dokončená a optimalizovaná.

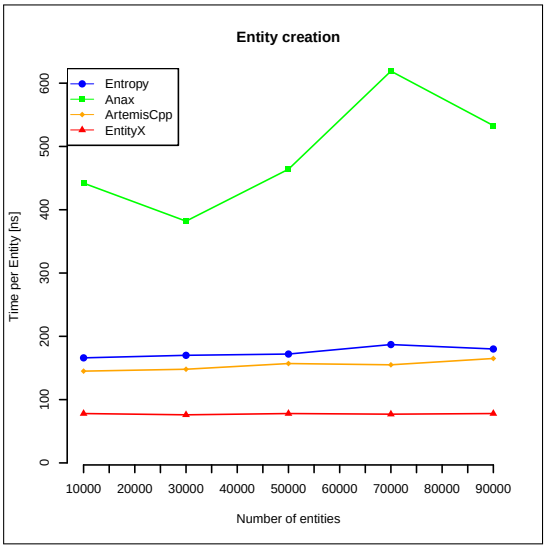
Všechny testy a výsledky, které jsou zobrazeny v této sekci byly provedeny na hardwarové konfiguraci č. 1 z tabulky 5.1. Zdrojové kódy, knihoven i testovaných bloků, byly přeloženy pomocí překladačem *GCC* (tabulka 5.2), s parametry `-std=c++1z -O3`. Jednotlivé testovací scénáře byly následně spouštěny po uzamčení procesoru do módu nejvyššího výkonu, spolu s příkazem `nice -n -20`, který aplikaci spustí s vyšší prioritou. Výsledná data byla zpracována aplikací *R* [12], jejíž výstupem jsou prezentované grafy.

Pojem *výpočet* v následujících sekcích vždy odkazuje na stejnou operaci, která má za úkol simulovat provedení výpočtu nad komponentami. V případě následujících scénářů je *výpočtem* operace nad čísly s plovoucí řádovou čárkou a aplikace matematických funkcí *sinus* a *cosinus*.

Ve všech případech, až na testování jednotlivých nosičů, jsou v testech používány dva typy komponent. Obě komponenty obsahují dva atributy typu `float` a v případě **Entropy** používají nosič typu pole.

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

5.3.1 Tvorba entit

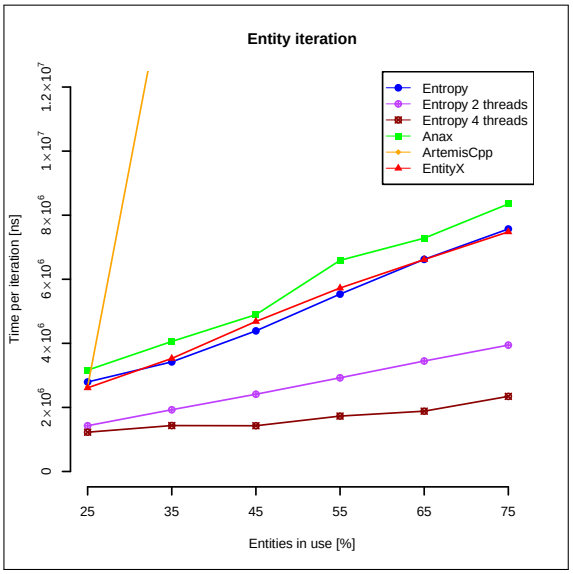


Obrázek 5.1: Graf výsledků pro přidání určitého počtu entit.

Obsahem prvního scénáře je tvorba určitého počtu entit, včetně dvou menších (8B) komponent. Cílem tohoto testu je zjistit, jakou časovou složitost má operace v závislosti na přidaném počtu entit. Výsledný graf lze najít na obr. 5.1, kde horizontální osa obsahuje počet vytvářených entit a vertikální čas vytvoření jedné z nich.

Z grafu lze určit, že časová složitost tvorby entit pro knihovnu **Entropy**, v závislosti na jejich celkovém počtu, je *konstantní*. Toto umožňuje entitnímu systému obsahovat vysoké množství entit, bez omezení rychlosti vykonávání následujících akcí. Rozdíl mezi knihovnou **Entropy** a ostatními knihovnami lze vysvětlit tím, že akce tvorby entity a přidání komponent musí identifikátor entity přidat do seznamu změněných entit.

5.3.2 Výpočetní systém

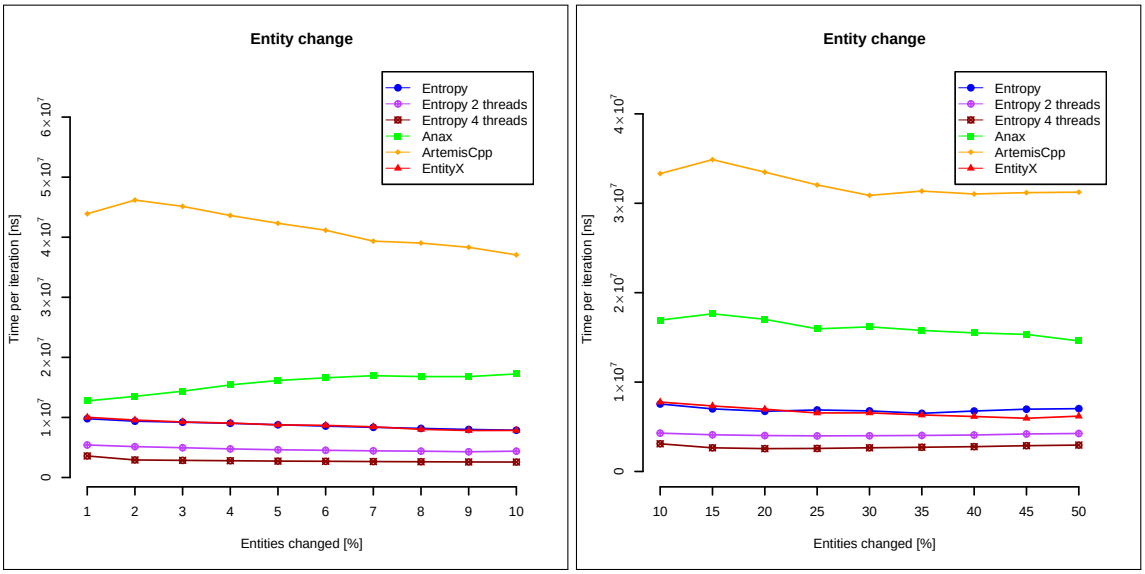


Obrázek 5.2: Jednoduchá iterace pohybového *systému*.

Předmětem druhého scénáře je použití entitního systému k provedení *výpočtu*. V první fázi testu je vytvořen určitý počet entit (V tomto případě 10 000.), následně je náhodně vybráno procento z nich, kterým jsou přidány 2 komponenty. Výsledný graf lze najít na obr. 5.2, horizontální osa obsahuje procentuální zastoupení entit, nad kterými bylo iterováno a vertikální průměrný čas jedné iterace. Graf, kromě jednotlivých knihoven, obsahuje také využití paralelizace knihovny **Entropy**. V tomto případě byl celkový počet iterovaných entit rozdělen mezi dvě, nebo čtyři výpočetní vlákna.

V grafu lze vidět lineární nárůst průměrné doby na jednu iteraci, který odpovídá zvyšujícímu se počtu iterovaných entit. Časový průběh pro knihovnu **ArtemisCpp**, je v tomto případě pravděpodobně způsoben implementací filtrování entit. Zajímavé průběhy mají také paralelní verze knihovny **Entropy**, kde se ze začátku ukazuje nevýhoda rozdělení malého počtu entit mezi více vláken.

5.3.3 Změny obsažených komponent



(a) Nižší procenta změn.

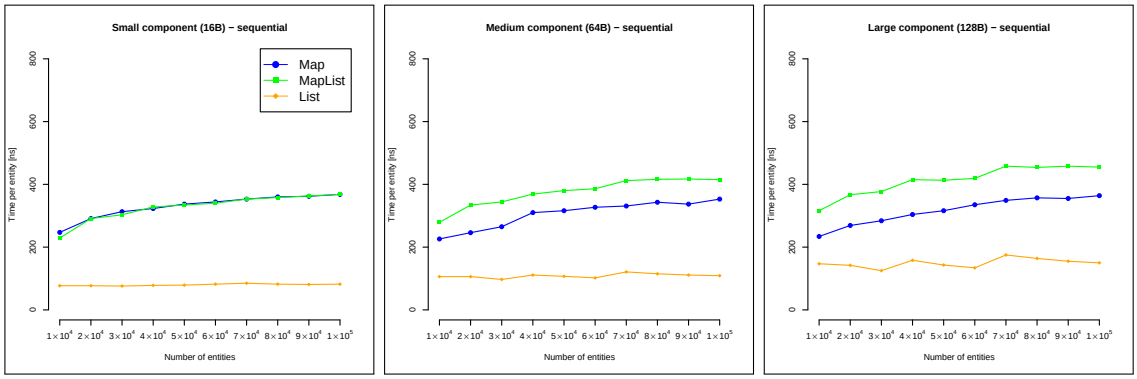
(b) Vyšší procenta změn.

Obrázek 5.3: Iterace nad entitami s daným procentem změn v obsažených komponentách.

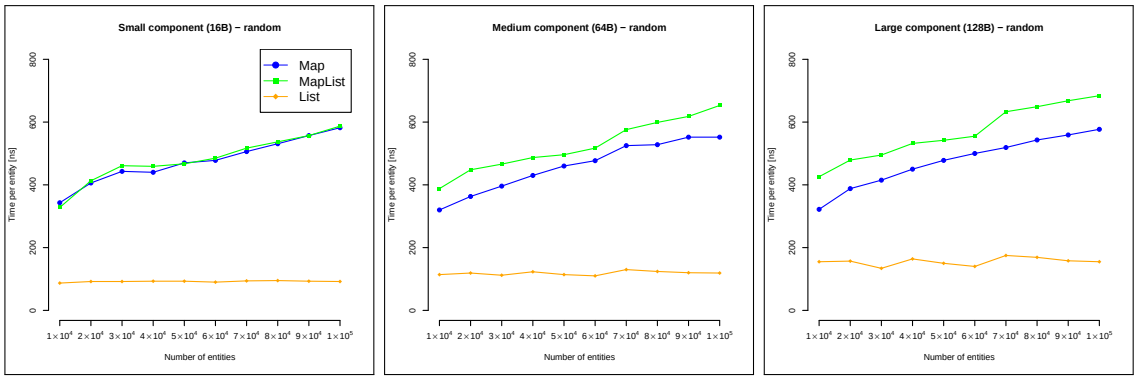
Následující testovací scénář porovnává, jakým způsobem se jednotlivé implementace chovají, v případě, kdy entity mění svou množinu přítomných komponent. V první fázi je opět vytvořen stejný počet entit – 10 000. Na rozdíl od předchozího případu jsou oba komponenty přidány všem entitám. Kromě základního *systému*, který provádí *výpočet*, je zde také *systém*, jehož úkolem je náhodně přidávat a odebírat komponenty. Tímto je simulována změna entit, čímž je vynucena aktualizace *entitních skupin*. Výsledný graf je rozdělený na nižší procenta změn (obr. 5.3a) a vyšší procenta změn (obr. 5.3b). Horizontální osa obsahuje v kolika procentech případů je množina komponent, které entita vlastní, změněna. Vertikální osa opět obsahuje průměrný čas na jednu iteraci obou systémů a následnou *obnovovací fázi*.

Z grafu lze vidět, že je knihovna **Entropy** dokáže udržet stejnou úroveň výkonu i přestože musí obnovovat obsah *entitních skupin*. Lineární charakter výsledné křivky lze zdůvodnit tím, že počet *výpočtů* byl – v případě vyššího počtu změn – snížen. Při využití paralelizace lze vidět předpokládaný nárůst výkonu, který je přibližně lineární v závislosti na počet vláken.

5.3.4 Nosiče komponent



(a) Sekvenční přístup, velikosti 16 B, 64 B a 128 B.



(b) Náhodný přístup, velikosti 16 B, 64 B a 128 B.

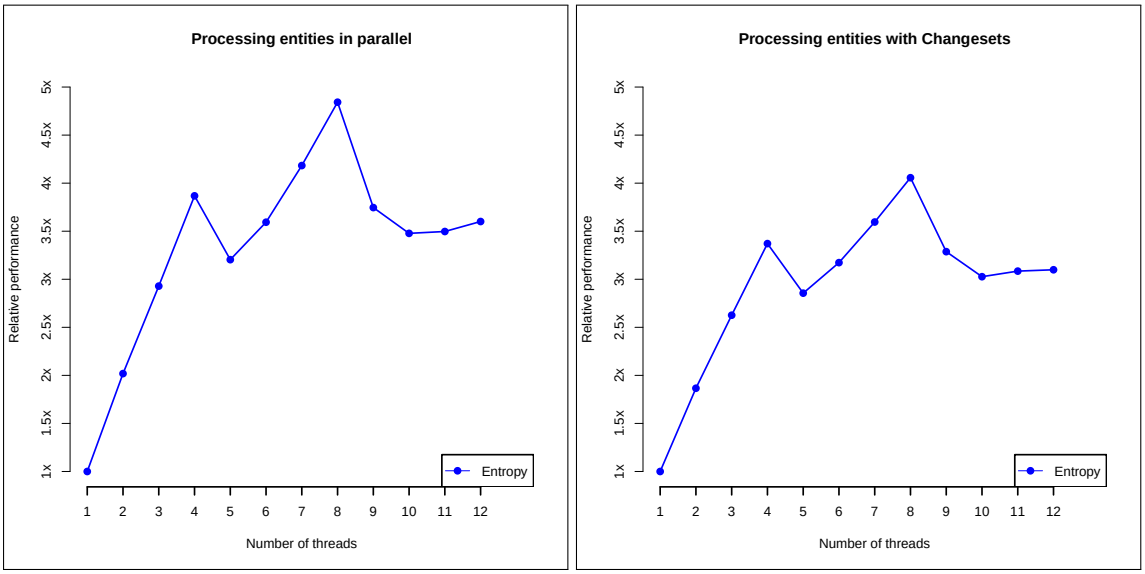
Obrázek 5.4: Porovnání nosičů komponent pro komponenty různých velikostí.

Dalším testovaným aspektem výsledné knihovny je porovnání různých typů *nosičů komponent*. Po vytvoření stejného množství entit, jako v předchozích případech, začíná měřený úsek, kdy je každé entitě přidán daný typ komponenty. Každý nosič je testován se třemi typy komponent – malé (16 bytů), střední (64 bytů) a velké (128 bytů). Tyto velikosti byly zvoleny, kvůli rozměrům řádku vyrovnávací paměti, která je v případě testované soustavy 64 B. Následuje druhý měřený úsek, ve kterém je přistupováno k entitám a jejich komponentám. Poměr operací přidání k operacím přístupu je 1:20, čímž je simulována předpokládaná zátěž entitního systému. K jednotlivým komponentám je přistupováno sekvenčně (obr. 5.4a) a náhodně (obr. 5.4a).

První vlastností, kterou lze z grafů vypozorovat, je na první pohled jednoznačná výhody nosiče typu **List**. Tento výsledek je dosažen implementací tohoto nosiče – jednoduché pole, které je indexováno identifikátorem entit. Za tuto výhodu je zaplaceno vyšší paměťovou náročností, v případech, kdy není pole zaplněno. Zajímavou vlastnosti, kterou lze vidět na průbězích nosiče **List**, jsou „zuby“, které vznikají kvůli alokaci většího bloku paměti.

Pro komponenty nejmenší velikosti jsou nosiče **Map** a **MapList** na stejné úrovni. Důvodem je pravděpodobně lepší datová lokalita nosiče **MapList**, díky které je do vyrovnávací paměti načteno vyšší množství instancí čtených komponent. Tato výhoda je, v případě větších komponent, přebita režií udržování *mapy* i *pole*.

5.3.5 Paralelní zpracování



(a) Entitní paralelizmus.

(b) Množiny změn.

Obrázek 5.5: Relativní urychlení testovaného *výpočtu* při použití daného počtu vláken.

Cílem závěrečného scénáře je otestovat, jak výhodné je použití několika vláken při přístupu k entitnímu systému. Testovaný systém opět zpracovává entity dříve použitým *výpočtem*. V první části je vytvořeno 10 000 entit, kde každé z nich jsou přiřazeny dvě komponenty. V prvním případě (obr. 5.5a) je množina entit rozdělena mezi jednotlivá vlákna. Ve druhém případě (obr. 5.5b) je místo přímého zápisu výsledných hodnot použito *množin změn*, s odloženým vykonáním. Součástí měřeného úseku je také obnova entitního systému. Horizontální osy obsahují počet vláken, který byl při zpravování použit. Vertikální obsahuje informaci o tom, kolikrát rychleji byly jednotlivé iterace dokončeny. Tato hodnota je relativní, vůči základnímu případu, kdy je použito pouze jedno vlákno.

Grafy ukazují téměř lineární škálování s počtem vláken, až do počtu fyzických jader testovaného systému (4 jádra). Následně lze na grafech vidět propad, při přechodu ze čtyř na pět vláken, který lze zdůvodnit využitím maximálního počtu fyzických jader. Dalším možným vysvětlením tohoto propadu je začátek použití technologie. Využití maximálního možného počtu vláken lze na grafu vidět ve formě maxima, při použití 8 vláken. Následně výkon opět klesá, kvůli režii správy jednotlivých vláken.

Graf použití *množin změn* (obr. 5.5b) ukazuje mírnější vzrůst relativní úrovně výkonu, která je způsobena odloženým zpracováním požadovaných akcí. Tuto situaci lze vylepšit dokončením implementace paralelní *obnovovací fáze*.

Škálování s počtem jader velmi závisí na složitosti úlohy, kterou jádra musí zpracovávat. Při jednoduchých operacích je režie na tvorbu vláken a údržba koherence vyrovnávacích pamětí příliš časově náročná na to, aby se paralelizace vyplatila.

Kapitola 6

Závěr

Cílem této bakalářské práce byl návrh a následná implementace entitního systému, který umožňuje vývoj aplikací za použití *Entity-Component-System* paradigmatu. Důležitou částí bylo studium aktuálních problémů při vývoji her a architektury moderních počítačů, díky kterým *ECS* paradigma vzniklo.

Při návrhu byly kromě základních vlastností *ECS* také zohledněny požadavky na paralelní přístup a jednoduchou integraci do existujících projektů. Před vytvořením výsledného návrhu vzniklo několik prototypů, jejichž cílem bylo určit vhodnost návrhu a jeho proveditelnost v jazyce *C++*.

Implementace byla provedena v programovacím jazyce *C++*. Při implementaci byly použity pokročilé vlastnosti jazyka, jako jsou *šablony* a *šablonové metaprogramování*. Dále bylo využito standardní knihovny a menších knihoven, dříve vytvořených autorem této práce. Implementováno bylo samotné jádro entitního systému, které umožňuje práci s *entitami*, *komponentami* a *systémy*. Kromě těchto základních vlastností umožňuje implementace také všechny tři typy paralelizace – *entitní*, *systémová* a pomocí *množin změn*.

Výsledkem práce je multiplatformní knihovna **Entropy**, která umožňuje návrh aplikací za použití *Entity-Component-System* paradigmatu. Hlavním úspěchem je možnost paralelního přístupu, který jiné knihovny nepodporují. Při porovnání výsledné implementace se knihovna **Entropy** výkonnostně řadí mezi ty rychlejší, přičemž je možné ji dále optimalizovat. Mezi možné optimalizace patří – paralelizace *obnovovací* fáze, rozdělení seznamů změněných entit podle *skupin* nebo kompletní konverze filtrování na použití *bitových množin*.

Autor práce bude dále pokračovat ve vývoji výsledné knihovny, jejíž aktuální verze je k dispozici v repositáři *Git* [36]. Mezi plánovaná rozšíření patří vazba entitního systému na vestavěný 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] *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

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

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

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

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

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

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

[8] *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

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

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

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

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

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

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

[15] *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

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

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

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

[19] *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

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

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

[22] 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

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

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

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

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

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

[28] 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/

[29] 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

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

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

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

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

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

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

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

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

[38] 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

[39] 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

[40] 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

Plakát

Komponentní systém pro herní grafický engine

Tomáš Polášek
Vysoké učení technické v Brně – Fakulta informačních technologií

Cíle

- Cílem byl návrh a implementace knihovny pro správu entit, která umožňuje vývoj pomocí *Entity-Component-System* paradigmatu.
- Mezi požadavky patřily:
  1. Možnosti paralelního provádění datových transformací.
  2. Efektivní použití vyrovnávacích pamětí.
  3. Intuitivní programovací rozhraní.
  4. Jednoduchá integrace do existujících projektů.

Důvody

- Obvyklé návrhové prostředky – jako *objektově orientované programování* – jsou pro návrh her příliš rigidní.
- Zvyšující se požadavky na oddělení jednotlivých částí herního enginu.
- Výkonnostní rozdíl mezi přístupovou dobou pamětí a rychlostí vykonávání instrukcí se stále zvyšuje.
- Vzniká *datově orientovaný návrh*, jehož cílem je vyšší efektivita práce s fyzickým hardware – vyrovnávací paměti, pipelining.

Entity-Component-System

- Návrhové paradigma založené na kompozici. Vychází z principů *datově orientovaného návrhu.*
- Striktní separace logiky a dat, čímž umožňuje vyšší úroveň modularity.
- Skládá se ze tří částí – *entity*, *komponenty* a *systémy*.

ID	Comp1	Comp2	Comp3
Entity1	NULL	DATA	DATA
Entity2	DATA	NULL	DATA
Entity3	NULL	NULL	DATA

Entity

- *Entity* reprezentují virtuální objekty uvnitř herního světa.
- Jejich hlavní funkcí je identifikace daného objektu a vazba na komponenty.
- Identifikátor *entity* je primárním klíčem v tabulce *entit*.

Komponenty

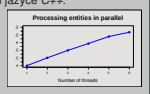
- Základní nosiče dat a zároveň nejmenší datová jednotka, kterou lze *entitám* přiřadit.
- Neobsahují „výkonnou“ logiku, pouze operace pro manipulaci dat.
- Sloupce tabulky *entit* obsahují jednotlivé typy komponent.

Systémy

- *Systémy* implementují datové transformace nad *entitami* a jejich *komponentami.*
- Transformují pouze takové *entity*, které obsahují požadované *komponenty.*

Výsledky

- Výsledkem je multiplatformní knihovna **Entropy**, která umožňuje návrh pomocí *ECS* paradigmatu.
- Implementována v programovacím jazyce *C++.*
- Využívá koncept *skupin*, které umožňují uchovávat seznamy filtrovaných *entit. Systémy* iterují pouze nad *entitami*, které již skutečně obsahují požadované *komponenty.*



Lineární škálování s počtem vláken

- Umožňuje paralelizaci na třech úrovních:
  - *Entity* – Rozdělení množiny *entit* na jednotlivá vlákna, která provádí požadovanou transformaci odděleně.
  - *Systémy* – Běh několika *systémů* zároveň na rozdílných vláknech.
  - *Množiny změn* – Pokud vlákna přistupují ke stejným zdrojům, je možné generovat *množiny změn.* Tím je daná operace odložena na dobu, kdy ji bude možné vykonat bez synchronizace.

Použití

- Knihovna **Entropy** je volně dostupná, pod licencí *MIT*, z níže uvedeného repositáře projektu.
- Příklad jednoduchého *systému*, který implementuje pohyb *entit*, definice *komponenty*:

```
struct PositionC
{
  float x, y;
  flot dX, dY;
}
```

- Definice *systému*:

```
struct MovementS : public System
{
  using Require = Require<PositionC>;
  void doMove();
}
```

- Samotný posun entit:

```
void doMove()
{
  for (auto &e : foreach())
  {
    PositionC *p{e.get<PositionC>()};
    p->x += p->dX;
    p->y += p->dY;
  }
}
```

Kontaktní informace

- GIT projektu: https://github.com/T0mt0mp/Entropy
- E-mail: xpolas34@stud.fit.vutbr.cz

Příloha C

Manuál

Příloha D

Použití knihovny

Co je Entropy

Entropy je knihovna napsaná v programovacím jazyce *C++*, která umožňuje návrh aplikací za použití *Entity-Component-System* paradigmatu. Knihovna je volně k dispozici, pod *MIT* licencí, v repositáři GIT.

Použití knihovny

Požadavky

Pro správnou funkci vyžaduje knihovna následující nástroje:

- Překladač jazyka *C++*, který podporuje standard jazyka *C++14* – mezi testované překladače patří GCC 6.3.1, Clang 3.9.1 a MSVC 19.10
- (volitelné) CMake – alespoň verze 2.8 (3.1 pro překlad porovnání, kvůli knihovně EntityX)

Stažení

Nejjednodušším způsobem, jak získat knihovnu **Entropy**, je klonováním jejího repositáře:

```
mkdir Entropy
git clone https://github.com/T0mt0mp/Entropy Entropy
cd Entropy
git submodule update --init --recursive
```

Překlad testů

Součástí knihovny **Entropy** jsou testy, které lze přeložit pomocí systému *CMake*.

```
mkdir build && cd build
cmake ../ -DCMAKE_BUILD_TYPE=Release
```

nebo pro operační systém Windows:

```
cmake ../ -DCMAKE_BUILD_TYPE=Release \
  -G "Visual Studio 15 2017 Win64"
```

potom lze pro překlad použít:

```
cmake --build . --config Release
```

Výsledkem jsou spustitelné soubory v adresáři `build`, nebo `Release`:

- `Comparison*` – Porovnání různých volně dostupných knihoven.
- `Test*` – Testy knihovny **Entropy**.
- `gameTest` – Testovací implementaci hry za použití knihovny **Entropy**.

Instalace

Po stažení knihovny ji lze nainstalovat do adresáře `/usr/local/include` následující sekvencí příkazů:

```
mkdir build && cd build
cmake ../Entropy/ && make install
```

Alternativně je možné manuálně nakopírovat obsah adresáře `Entropy/include/` (obsahuje složku `Entropy`) do systémového `include`, nebo jiného adresáře, kde ji překladač najde.

Základy

Před jejím použitím je nutno knihovnu přidat do zdrojového kódu, pomocí direktivy `include`:

```
#include <Entropy/Entropy.h>
```

Universe

Středem knihovny **Entropy** je třída **Universe**, která umožňuje komunikaci s entitním systémem. Z této třídy je třeba nejdříve podědit což umožňuje existenci několika oddělených světů:

```
class MyUniverse : public ent::Universe<MyUniverse>
{ };
```

Potom již lze vytvořit instance daného vesmíru:

```
MyUniverse u;
```

Entity

Entity jsou v knihovně **Entropy** jednoduché číselné identifikátory, kterým je možné přiřadit 0-1 *komponent*. *Entity* se mohou nacházet ve dvou stavech – aktivní a neaktivní. Tvorba nových entit je možná skrz třídu **Universe**:

```
using Entity = MyUniverse::EntityT;
Entity e = u.createEntity();
```

Entity je dále možné zničit, použitím metody `destroy`:

```
e.destroy();
```

Informaci, zda je *entita* platná lze získat metodou `valid`. *Entity* se stává nevalidní ve chvíli, kdy je zničena.

Komponenty

Komponenty jsou základními nosiči dat v *ECS*. Nové typy *komponent* lze definovat následovně:

```
struct PositionC
{
    float x, y;
};
```

Aby třída, nebo struktura, mohla být považována za *komponentu*, musí splňovat následující požadavky. Musí obsahovat výchozí konstruktor – konstruktor bez parametrů. Kromě toho může obsahovat neomezený počet dalších konstruktorů, kterým lze předávat parametry při přidání *komponenty*.

Každý typ *komponent* v knihovně **Entropy** má vlastní tzv. *nosič komponent*. Nové typy *nosičů* lze definovat skrz dědění třídy **BaseComponentHolder** a implementaci všech virtuálních metod. Knihovna **Entropy** obsahuje tři typy předdefinovaných nosičů:

- **ComponentHolder** – Výchozí nosič pro komponenty, které nemají specifikovaný jiný. Používá `std::map`. Tento nosič je výhodný v případech, kdy nejsou entity, které obsahují daný typ komponenty nijak seřazené.
- **ComponentHolderMapList** – Používá `std::map`, který mapuje entity do souvislého pole komponent.
- **ComponentHolderList** – Používá pole, do kterého jsou přímo namapovány identifikátory entit. Tento nosič je výhodný pro případy, kdy každá entita obsahuje daný typ komponent.

Specifikaci nosiče komponent lze provést následujícím způsobem:

```
struct PositionC
{
    using HolderT = ent::ComponentHolderList<PositionC>;

    PositionC(float px, float py) :
        x{px}, y{py} { }
    PositionC() :
        x{0.0f}, y{0.0f}
    float x, y;
};
```

Nad entitami lze provádět operace, které ovlivňují jaké komponenty daná entita obsahuje. Mezi tyto operace patří – přidání (`add`), odebrání (`remove`), získání komponenty (`get`) a získání přítomnosti (`has`). Příklad použití těchto metod je následující:

```
Entity e = u.createEntity();
e.add<PositionC>();
e.remove<PositionC>();
e.has<PositionC>();      // -> false
e.add<PositionC>(1.0f, 2.0f);
e.has<PositionC>();      // -> true
e.get<PositionC>->y;     // -> 2.0f
```

Systémy

Systémy umožňují iteraci nad entitami, tedy i nad komponentami, které vyhovují danému filtru. Jelikož komponenty neobsahují žádnou logiku, *systémy* plní funkci tvorby akcí. Pokud bychom chtěli vytvořit systém, který bude iterovat nad entitami s komponenty typu **positionC** a **MovementC**, vypadala by jeho deklarace následovně:

```
class MovementS : public MyUniverse::SystemT
{
    using Require = ent::Require<PositionC, MovementC>;
    using Reject = ent::Reject<>;
public:
    void doMove();
};
```

V tomto případě není `Reject` nutné, jelikož výchozí hodnota obou typů je prázdný typový seznam. Implementace samotné akce by mohla vypadat následovně:

```
void MovementS::doMove()
{
    for (auto &e : foreach())
    {
        PositionC *p{e.get<PositionC>()};
        MovementC *m{e.get<MovementC}()};
        p->x += m->dX;
        p->y += m->dY;
    }
}
```

Děděná třída *System*, obsahuje 3 metody – `foreach`, `foreachAdded` a `foreachRemoved` – díky kterým lze iterovat před entity vyhovující specifikovanému filtru. *Systémy* lze přidat danému vesmíru voláním metody `addSystem<S>`.

Kontrolní tok

Důležitou součástí knihovny **Entropy** je tok kontroly, který postupuje následujícím způsobem:

1. Vytvoření instance třídy **Universe**.
2. Registrace komponent, metodou `registerComponent<C>`. Tento krok je možný pouze před inicializací!
3. Inicializace vesmíru zavoláním metody `init` na dané instanci třídy **Universe**.
4. Následně je již možná práce s entitním systémem – přidávání *systémů*, tvorba entit apod.
5. Kdykoliv lze z předchozí fáze přejít do fáze *obnovy*, kdy je obnovena konzistence celého entitního systému. Po dokončení se systém vrací do kroku 4.

Součástí *obnovovací* fáze je také aktualizace seznamů, nad kterými *systémy* iterují.