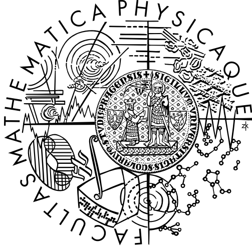


MATEMATICKO-FYZIKÁLNÍ FAKULTA
Univerzita Karlova

BAKALÁŘSKÁ PRÁCE

Kristián Petráš

Lepenka – Modulárny framework umožňujúci tvorbu hier pre platformu .NET

Katedra distribuovaných a spolehlivých systémů

Vedoucí bakalářské práce: Mgr. Filip Kliber
Studijní program: Informatika
Studijní obor: IPP2

Praha 2022

Prohlašuji, že jsem tuto bakalářskou práci vypracoval(a) samostatně a výhradně s použitím citovaných pramenů, literatury a dalších odborných zdrojů. Tato práce nebyla využita k získání jiného nebo stejného titulu.

Beru na vědomí, že se na moji práci vztahují práva a povinnosti vyplývající ze zákona č. 121/2000 Sb., autorského zákona v platném znění, zejména skutečnost, že Univerzita Karlova má právo na uzavření licenční smlouvy o užití této práce jako školního díla podle §60 odst. 1 autorského zákona.

V .... dne ....

Podpis autora

Rád by som sa poďakoval môjmu vedúcemu práce, Mgr. Filipovi Kliberovi, za ochotu, trpezlivosť a cennú spätnú väzbu. Ďalej ďakujem rodine, priateľom a blízkym za podporu pri písaní tejto práce a počas štúdia.

Název práce: Lepenka – Modulárny framework umožňujúci tvorbu hier pre platformu .NET

Autor: Kristián Petráš

Katedra: Katedra distribuovaných a spolehlivých systémů

Vedoucí bakalářské práce: Mgr. Filip Kliber, Katedra distribuovaných a spolehlivých systémů

Abstrakt: Vytváření her pomocí datové architektury Entity Component System vyžaduje změnu přístupu k programování. Současné frameworky .NET předpokládají, že uživatel zná principy ECS. Proto s rostoucím zájmem o tvorbu her pomocí ECS roste i potřeba frameworku, který by umožnil intuitivní osvojení těchto principů. Výsledkem této práce je datově řízený framework s archetypální organizací dat umožňující paralelizaci systémů. Kromě toho framework hojně využívá generování kódu, což uživatele vede k dosud neprozkoumanému deklarativnímu způsobu vytváření her. V kombinaci s nezávislostí na platformě je Lepenka jediným řešením pro platformu .NET, které má výše uvedené vlastnosti. Další důležitou vlastností je opakované použití kódu v různých hrách, které zajišťují moduly umožňující úplnou abstrakci herních systémů a komponent.

Klíčová slova: framework modularita .NET hry

Title: Lepenka - Modular framework for games development on the .NET platform

Author: Kristián Petráš

Department: Department of Distributed and Dependable Systems

Supervisor: Mgr. Filip Kliber, Department of Distributed and Dependable Systems

Abstract: Creating games using the Entity Component System data architecture requires a change in programming approach. Current .NET frameworks assume that the user is familiar with the principles of ECS. Therefore, as the interest in creating games using ECS grows, so does the need for a framework that allows intuitive learning of these principles. The result of this work is a data-driven framework with an archetypal data organization allowing for parallelization of systems. In addition, the framework makes extensive use of code generation, leading the user to a previously unexplored declarative way of creating games. Combined with platform independence, Lepenka provides the only solution for the .NET platform that takes on the aforementioned characteristics. Another important feature is code reusability across games, provided by modules allowing full abstraction of game systems and components.

Keywords: framework modular .NET games

Obsah

Úvod **4**

1 Problematika vývoja hier **6**
1.1 Dekonštrukcia hry . . . 6
1.2 Herný engine . . . 7
1.3 Objektový model . . . 8
1.4 Objektovo komponentový model . . . 11
1.4.1 Problém inicializácie komponentov . . . 13
1.4.2 Problém aktualizácie komponentov . . . 15
1.4.3 Zhrnutie . . . 17
1.5 Entity Component System . . . 17
1.5.1 Motivácia . . . 18
1.5.2 Pamäťový model . . . 19
1.5.3 Archetype . . . 21
1.5.4 Nevýhody . . . 25

2 Lepenka **26**
2.1 Štruktúra projektu . . . 26
2.2 Konfigurácia projektu . . . 27
2.3 Komponent . . . 28
2.4 Systém . . . 29
2.4.1 Globálny systém . . . 30
2.4.2 Inicializačný systém . . . 30
2.4.3 Entitový systém . . . 31
2.4.4 Systém špecifického vlákna . . . 32
2.5 Systémové príkazy . . . 33
2.5.1 `IUniverseCommand` . . . 34
2.5.2 `EntityCommand` . . . 35
2.6 Zdroj . . . 36
2.7 Tvorba modulov . . . 37
2.7.1 Systém . . . 37

2.7.2 Component 39
2.7.3 Schéma XML 40
2.7.4 Integrácia modulu v projekte 41
2.8 Integrácia modulov 42
2.8.1 Prepojenie systémov 42
2.8.2 Komponenty implementované modulom 44
2.8.3 Integrácia modulových komponentov 44

3 Implementácia Lepenky 46
3.1 Architektúra 46
3.2 Vesmír 47
3.3 Komponent 48
3.3.1 Úložisko komponentov 48
3.3.2 Manažér komponentov 48
3.4 Entita 49
3.5 Systémy 49
3.6 Príkaz 50
3.6.1 Príkazy pre vesmír 50
3.6.2 Príkazy pre entitu 50
3.6.3 Integrácia so systémami 52
3.7 Zdroj 52
3.8 Archetyp 53
3.8.1 Identifikácia archetypov 53
3.8.2 Filtrovanie archetypov 54
3.8.3 Registrácia archetypov 55
3.8.4 Konzistentnosť entít 56
3.8.5 Operácie archetypu 57
3.9 Generátor 58
3.9.1 Generácia vesmíru 58
3.9.2 Generácia herných slučiek špecifických vlákien 65
3.9.3 Generácia modulov 65
3.10 Porovnanie ECS frameworkov 66

4 Ukážky frameworku 68
4.1 Konzolový príklad 68
4.2 Raylib 74
4.3 Príklad so sieťovaním 80

Záver 83
4.4 Budúci vývoj 84
4.4.1 Iterátor entít 84

4.4.2 Optimalizácia výkonu a pamäťovej stopy 86
4.4.3 Kompletná konfigurácia v kompilačnom čase 87
4.4.4 Plánovanie systémov 88
4.4.5 Interakcia s modulmi 88
4.4.6 Diagnostika 88
4.5 Benchmark 89

Seznam použité literatury **91**

A Používanie Lepenky **92**
A.1 Tvorba projektu 92
A.2 Tvorba modulov 93

Úvod

Väčšina dnešných hier je tvorená frameworkami, takzvanými hernými enginmi. [1] Tie ponúkajú prívetivé tvorčie prostredie poháňané výkonnými nástrojmi. V praxi sa stretávame s rôznymi prístupmi organizácie dát v herných enginoch. Entity Component System, softvérový architektonický vzor na reprezentáciu objektov herného sveta, sa stáva čoraz populárnejším. V práci si ukážeme, ako Lepenka implementuje spomenutý vzor a aké inovácie v prostredí ECS ponúka. [2]

Koncepty a pravidlá ECS sa dajú vo všeobecnosti jednoducho naučiť. Niektoré aspekty dizajnu ECS sú ortogonálne s princípmi OOP[1]. Ich správne uplatnenie je často v rozpore s intuíciou a vyžaduje si prax. Premostenie z konvenčných prístupov tvorby hier si vyžaduje čas, no používatelia uvádzajú, že akonáhle práca s ECS „klikla“, písanie kódu bolo jednoduchšie a kód bol opätovne použiteľný a škálovateľný. [3]

Naším cieľom je vytvoriť framework umožňujúci intuitívne pochopenie princípov ECS. Abstrakciou komponentov a systémov do modulárnych prvkov prináša Lepenka doteraz nepreskúmaný spôsob tvorby hier. Jazyk C# ponúka nástroje na generáciu dodatočného zdrojového kódu počas kompilácie. Extenzívnym využívaním generácie kódu oslobodzujeme užívateľa od implementačných detailov. Dáta budú organizované spôsobom umožňujúcim rýchlu iteráciu herných entít a paralelizáciu herných systémov. Generátory kódu tak v kombinácii s TPL[2] dovoľujú tvorbu optimalizovanej hernej slučky bez vnášania komplexity do procesu tvorby hry.

V práci si ukážeme, ako Lepenka rieši problém členenia dát, tvorby hernej slučky a organizácie modulov. Moduly slúžia ako diskrétne jednotky kódu použiteľné naprieč projektami. Ako prvé si detailnejšie vysvetlíme rôzne spôsoby organizácie objektov herného sveta. Ukážeme, ako ich úskalia rieši ECS prístup. V ďalšej časti opíšeme architektúru a implementáciu Lepenky. Vysvetlíme, ako využívať nami implementovaný framework rovnako ako aj moduly,

[1] Objektovo orientované programovanie.

[2] Task Parallel Library, kolekcia typov a namespacov pridávajúca paralelizačné schopnosti do programov.

dôležitý koncept frameworku. Zhrnieme si výhody v používaní Lepenky a porovnáme proces tvorby hry aj výkonnosť rôznych prístupov organizácie dát. Predvedieme ukážky využitia frameworku v praxi a na záver preskúmame nedostatky frameworku a ponúkneme jeho možné vylepšenia.

Kapitola 1

Problematika vývoja hier

1.1 Dekonštrukcia hry

V terminológii vývoja hier sa opakovane stretávame s fluidnými definíciami pojmov. Všetci intuitívne tušíme, čo pojem hra znamená, avšak závisiac od kontextu sa vynárajú definície z oblasti hernej teórie, alebo herného dizajnu. Podľa Raphaela Kostera, hlavného dizajnéra Ultimy Online, je hra

> „interaktívny zážitok ktorý ponúka hráčovi sekvenciu vzorov so zvyšujúcou sa náročnosťou, ktoré sa on, či ona naučia a eventuálne vycibria." [4]

Pre nás je dôležitý pohľad na hru z hľadiska jej architektúry. Hrou rozumieme imaginárny matematický model manipulovaný počítačom. Často reprezentuje podmnožinu reálneho sveta, pričom využíva techniky zjednodušenia a aproximácie. Tento model je časová simulácia, v ktorej svet je dynamický a mení sa v závislosti od času. Dôležitou vlastnosťou hier je ich schopnosť interagovať s užívateľom. Vstup od užívateľa je nepredvídateľný, pričom korektný priebeh hry vyžaduje pravidelné aktualizácie systému. Tým sa dostávame ku klasifikácii hry ako mäkkom interaktívnom systéme reálneho času.

Pre systémy reálneho času je dôležitá záruka, že sa určitá činnosť ukončí v známom časovom úseku. Systémy delíme na mäkké, pri ktorých sú záruky len približné a negarantované, a tvrdé, kde prípadne nedodržanie časového limitu spôsobí vážne následky. V našom prípade nie je nebezpečné nedodržať záruku aktualizácie systému alebo vykreslenia snímku, preto radíme systém do mäkkej triedy. [5] Aj pre hru ako mäkký systém je prioritou zaručiť konzistentný beh hry a vykresľovanie snímkov.

Základným stavebným prvkom hry je herný objekt. Herný objekt reprezentuje nezávislú jednotku v hernom svete, pod ktorou si môžeme predstaviť

vozidlo, budovu, hráča, alebo osvetlenie. Na zachytenie kompletnej domény herného sveta vyžadujeme od herných objektov aj schopnosť reprezentovať abstraktnejšie koncepty. Okrem herných objektov s ktorými hráč priamo interaguje rozumieme objektom aj ako scénam, alebo sieťovým prvkom.

Objekty v prepojení s popisom ako medzi sebou komunikujú a pravidlami ako aktualizovať hru kolektívne tvoria model herného sveta. Ten simulujeme pomocou herného cyklu, podobne ako u numerických simulácii, zloženého z opakovaného výpočtu definovaných kalkulácii. Tie upravujú herný svet a reprezentujú jeho stav v diskrétnych časových jednotkách.

Hra teda obsahuje dáta reprezentované v objektoch a logiku, ktorá sa opakovane vykonáva v predurčenom poradí. Tento mentálny model je bežný pre objektovo orientované jazyky, v ktorých je tvorená väčšina hier. [6]

1.2 Herný engine

Pojem "herný engine" vznikol v polovici deväťdesiatych rokov ako referencia na populárnu strieľačku z pohľadu prvej osoby Doom od herného štúdia id Software. [4] Architektúra Doomu rozdeľovala základné komponenty ako vykresľovací a zvukový systém od častí špecifických na hernom svete. To umožnilo vzniku komunity, ktorá modifikovala ich nástroje na tvorbu nových zbraní, vozidiel, alebo dokonca nových hier. Rozhodnutie navrhnúť rozdelenú architektúru vyžadovalo zložitejšiu a časovo náročnejšiu implementáciu, avšak uvoľnilo to zdroje pri návrhu neskorších hier. Ku koncu deväťdesiatych rokov už bolo bežné navrhovať hry s vysoko prispôsobiteľnými časťami, často pomocou integrácie skriptovacieho jazyka do enginu. Id software si pomocou hier ako Unreal získal rozsiahlú používateľskú základňu tvoriacu nový obsah rovnako ako sekundárny zárobok z licencovania komponentov ich herných enginov. [4]

V súčasnosti neexistuje definícia hraníc medzi hrou a herným enginom. Aj samotné hry môžu slúžiť ako engine, no sú značne obmedzené. Čím ďalej abstrahujeme koncepty a oddiaľujeme sa od viazanosti na žáner alebo doménu, tým horšie engine zvláda špecifické úlohy. Je zjavné, že v funkcionality enginu nápomocné pri tvorbe bojových hier nebudú naplno využité v strategických hrách. Vplyv na výkonnosť a schopnosť tvoriť dané hry sa môže výrazne líšiť už aj zmenami v dátových štruktúrach enginu. Všeobecne zamerané enginy ponúkajú možnosť voľnosti v dizajne hry na úkor výkonu a doménovo špecifických funkcionalít. Je ale potrebné vyzdvihnúť, že aj napriek špecializácii na strieľačky z prvej osoby bol Unreal Engine úspešne použitý na populárne tituly ako Tekken 7 alebo Mass Effect. [4] Engine v ktorom sa dá vytvoriť úplne ľubovoľná hra je z vyššie uvedených dôvodov pravdepodobne nevytvoriteľný.

Stále ale nevieme, aké komponenty tvoria engine. V komunite tvorcov

hier neexistuje konsenzus v odpovedi na túto otázku. Z praxe si všímame, že enginy obsahujú grafický vykresľovací systém, systémy určené na fyziku, zvuky a podporu sieťovania. Vytvárajú určitú abstrakciu nad správou pamäti a často podporujú využívanie viacerých vlákien.

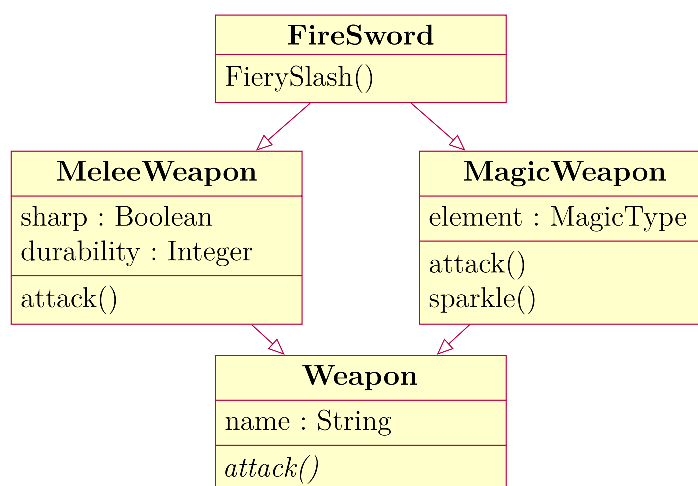
Bežne sa stretávame aj s pojmami herný framework a herná knižnica. Jednoducho povedané, knižnica slúži ako znovupoužiteľná kolekcia kódu na implementáciu špecifickej funkcionality. Framework sa skladá z vhodne zlúčených knižníc, pričom ponúka celistvú kostru pre tvorbu hier. Prechod od frameworku do enginu zvyčajne spočíva v zabalení kostry pod užívateľsky prívetivý editor sveta a graf reprezentujúci objekty herného sveta.

1.3 Objektový model

Jeden zo spôsobov ako modelovať herný svet je definovať explicitnú triedu pre každý typ herného objektu. Programátor vytvára vhodné abstrakcie a buduje strom typov herných objektov. Výhodou tohto prístupu je jednoduchá a zrozumiteľná implementácia. [6] Na druhú stranu tento model prináša mnoho problémov:

Hlboké hierarchie Jeden z veľkých problémov objektovo orientovaného programovania je zložitá navigácia kódu pre hlboké dedičné stromy. Excesívne používanie polymorfizmu znepríjemňuje testovanie a ladenie kódu. Tento model svojim návrhom priamo podporuje tento vzor.

Diamantový problém Návrh herných objektov sa spolieha na schopnosť viacnásobnej dedičnosti objektov.



Obr. 1.1 Diagram hierarchie typov zbraní v hernom svete.

Výpis kódu 1 Nejednoznačné volanie funkcie:

```
public class FireSword : MeleeWeapon, MagicWeapon
{
  public void FierySlash()
  {
    Sparkle();
    Sharp = true;
    // We can't decide whether to call
    // the MeleeWeapon or MagicWeapon class function.
    Attack();
  }
}
```

Už v tomto jednoduchom príklade sa dostávame do situácie, kedy nie je jasné, metódu akého predka máme zavolať. Riešenie sa v rôznych programovacích jazykoch líši, avšak je potrebné vyzdvihnúť, že najbežnejšie riešenie tohto problému spočíva v duplikovaní kódu.

Duplicitný kód Pri tvorbe hry potrebujeme viac herných objektov s podobným správaním. V tomto modeli sú reprezentované unikátnymi triedami. Všetky herné objekty zdieľajú väčšinu logiky a aby sme zabránili duplikovaniu kódu ich zjednotíme pod jedného rodiča.

V tomto bode potrebujeme rozhodnúť, či je ohnivý meč potomkom zbrane na blízko alebo čarovnej zbrane. Pri dedení z jednej z tried potrebujeme na zachovanie funkcionalít skopírovať kód z druhej triedy čím znejasňujeme hierarchiu herných objektov a porušujeme odporúčané princípy programovania.

Výpis kódu 2 Riešenie diamantového problému kopírovaním funkcionality druhej triedy:

```
public class FireSword : MeleeWeapon
{
    public void FierySlash()
    {
        // We can't decide whether to call
        // the MeleeWeapon or MagicWeapon class function.
        Attack();
    }

    // MagicWeapon functionality
    public MagicType Element {get;} = MagicType.Fire

    public void Sparkle() => Console.Writeline("*****");
}
```

Monolitickým riešením je vytvoriť jednotnú triedu, ktorá zahrnie funkcionalitu oboch tried a tým vyriešime problém s viacnásobnou dedičnosťou. Pri komplexnejších hierarchiách sa aplikácia tohto riešenia stáva neudržateľnou.

Výpis kódu 3 Riešenie diamantového problému zjednotením predkov:

```
public class MagicMeleeWeapon : Weapon
{
    // MeleeWeapon functionality
    public int Durability {get; set;}
    public bool Sharp {get; set;}

    // MagicWeapon functionality
    public MagicType Element {get; set;}
    public void Sparkle() => Console.Writeline("*****");
}
```

Neprenositeľnosť Vybudovaný model často nie je flexibilný a obsahuje len zopár prvkov, ktoré je možné využiť v budúcom projekte. Objekty sú z definície viazané na doménu hry a nie je možné ich integrovať do hier iných žánrov.

1.4 Objektovo komponentový model

Pri objektovom modeli je hlavnou príčinou problémov využívanie dedičnosti na pridávanie funkcionality. Enkapsulácia zdieľanej logiky a dát tvorí komponenty, ktoré herný objekt spravuje.

Výpis kódu 4 Základná implementácia objektovo komponentového modelu:

```
public interface IComponent
{
    public void Update();
}

public class GameObject
{
    public List<IComponent> Components {get;} = new();

    public void AddComponent(IComponent component)
    {
        Components.Add(component);
    }

    public void RemoveComponent(IComponent component)
    {
        Components.Remove(component);
    }

    public void UpdateComponents()
    {
        foreach (var component in Components)
        {
            component.Update();
        }
    }
}
```

Herný objekt teraz slúži ako kontainer komponentov s funkciami na inicializáciu a aktualizáciu. Ideálne je objekt a aj komponent plne generický.

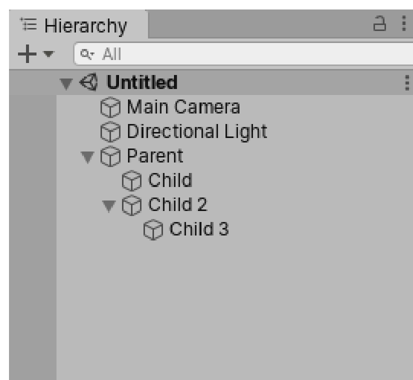
Zmenou v štruktúre herných objektov sme sa oslobodili od všetkých nevýhod objektového modelu. Pri voľnejšej väzbe medzi funkcionalitou a herným objektom sa otvára priestor pre modulárnejšie komponenty, použiteľné naprieč hrami. Umožnili sme vytvárať referencie nie len pomocou vzťahu objekt–objekt, ale aj objekt–komponent a dokonca komponent–komponent.

Výpis kódu 5 Príklad závislosti medzi komponentami:

```
public class HealthComponent : IComponent
{
    public int Health {get; set;}

    public void Update()
    {
        var bleeding = GetObject()
                       .GetComponent<BleedingComponent>();
        if (bleeding is not null)
        {
            Health--;
        }
    }
}

public class BleedingComponent : IComponent
{
    public int Duration {get; set;}

    public void Update()
    {
        if (--Duration == 0)
        {
            GetObject().RemoveComponent<BleedingComponent();
        }
    }
}
```

Manipulácia s komponentami a objektami prebieha pomocou funkcií zistenia objektu, ktorý vlastní daný komponent. Z objektu extrahujeme komponenty cez volanie funkcie získania komponentu. Špecifické rozhranie na komunikáciu medzi komponentami sa samozrejme líši naprieč implementáciami.

Model zvyčajne obsahuje implicitný priestorový komponent a strom herných objektov. Všetky vytvorené herné objekty sú reprezentované vrcholom na strome, čím vytvárame hierarchiu objektov. To umožňuje manipuláciu s objektami, hlavne s priestorovým komponentom aj na relatívnej úrovni. Ako príklad hierarchie si ukážeme rozhranie herného enginu Unity:



Obr. 1.2 Ukážka hierarchie objektov.

Môžeme si povšimnúť, že herný svet reprezentuje strom objektov. Pri dopytovaní sa na pozíciu objektu môžeme uvažovať nie len v absolútnych hodnotách, ale aj relatívnych od otcovského vrcholu. Výhodou tohto prístupu je konceptuálne jednoduchá reprezentácia herného sveta. Prívetivý myšlienkový model napomáha pri tvorbe nástrojov a uvažovaniu nad zmenami v strome.

Herní vývojári komplexnejších projektov v súčasnosti experimentujú z novými modelmi. Aj tento model si so sebou nesie potenciálne problémy. Možnosť referencovať komponenty si vyžaduje spôsob ako získať komponent z herného objektu a operácie na posun po hierarchickom strome. Funkcie získania rodiča a vyhľadávania objektov vytvára cyklické závislosti medzi objektmi. Bobby Anguelov, vývojár hier s dlhoročnou praxou výborne znázorňuje príklad zdôrazňujúci potenciálne problémy modelu. [7]

Majme nasledujúce komponenty:

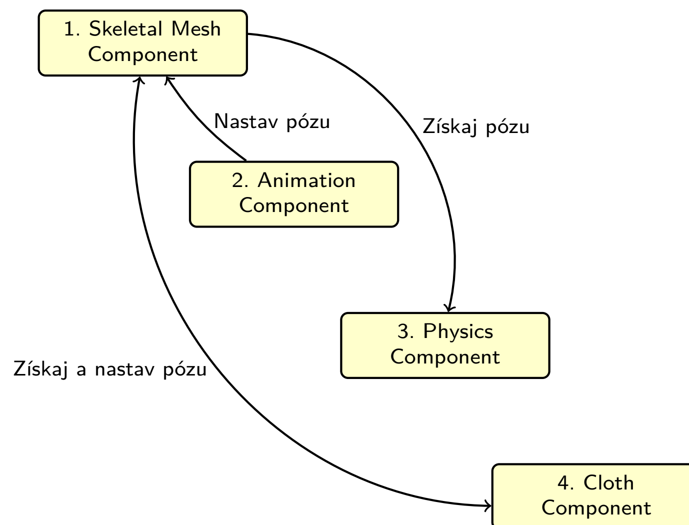
- Skeletal Mesh Component drží informácie o kostre postavy a transformácii kostí
- Animation Component riadi logiku animácie a generuje pózu postavy
- Physics Component aktualizuje fyziku objektov; postáv a látky
- Cloth Component spúšťa simuláciu látky na základe pózy postavy

Tento príklad komponentov spôsobuje dva problémy.

1.4.1 Problém inicializácie komponentov

Komponent kostry zodpovedá za správnu alokáciu štruktúr na vytvorenie pózy postavy a z toho dôvodu ho inicializujeme prvý. Animačný komponent inicializujeme za ním, nastavujúc počiatočnú pózu. Ako ďalší inicializujeme

fyzikálny komponent, ktorý novovytvorenú pózu pripraví na neskoršie transformácie objektu. Nakoniec inicializujeme látkový komponent, ktorého úloha je čítať a modifikovať pózu aj s pridaním látky.



Obr. 1.3 Diagram závislostí inicializácie komponentov.

Pretože existujú závislosti medzi komponentami, musíme modelu zadefinovať, v akom poradí sa budú inicializovať. Jedno z možných riešení je implementovať inicializačné etapy. To znamená, že počas hernej slučky sú preddefinované pasáže, v ktorých sa spúšťa kód. Tým vieme rozdeliť inicializáciu na viac častí, v nich je možné do istej miery definovať poradie komponentov. Toto riešenie nie je udržateľné, eventuálne sa pri komplexnejšom príklade dopracujeme ku ďalšiemu cyklu závislostí.

Aké máme možnosti, ak sa dostaneme ku cyklu aj za inicializácie pomocou etáp? V tomto bode sme nútení vytvoriť nový typ herného objektu, ktorý manuálne inicializuje svoje komponenty. Simulujeme tak objektový model, ktorému sme sa chceli vyhnúť. Prichádzame o výhody explicitného odkazovania sa na dáta, keďže manipulujeme s generickými komponentami. Navyše strácame bezpečnosť umožnením odstránenia komponentu, ktorý je nutnosťou pre daný objekt. Za ďalšiu alternatívu sa ponúka tvorba hybridu medzi generickými komponentami a staticky definovanými komponentami, na úkor čitateľnosti kódu.

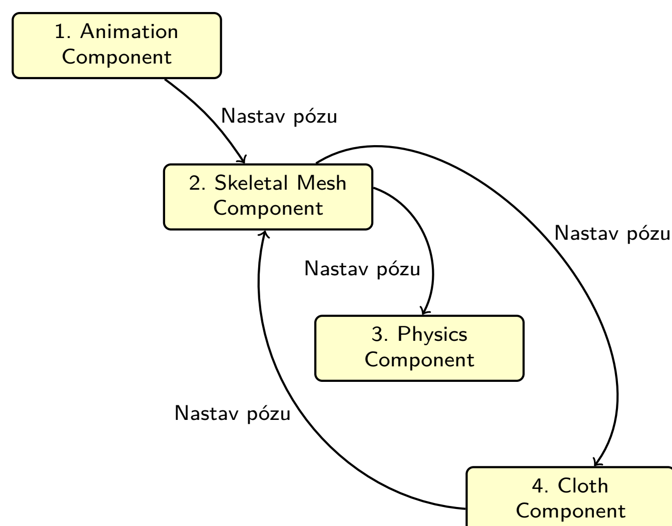
Výpis kódu 6 Explicitná inicializácia komponentov:

```
public class Player : GameObject
{
    // Components internally call
    // GetObject().GetComponent<T>()
    // to get required dependancies.
    public void Construct()
    {
        AddComponent(new SkeletalMeshComponent("Mesh"));
        AddComponent(new AnimationComponent("Animation"));
        AddComponent(new PhysicsComponent("Physics"));
        AddComponent(new ClothComponent("Cloth"));
    }
}
```

1.4.2 Problém aktualizácie komponentov

Poradie aktualizácie komponentov sa líši od poradia ich inicializácie. Ako prvé sa aktualizuje animačný komponent, aktualizujúci pózu kostry. Komponent kostry pri aktualizácii nastavuje fyzikálny komponent. Simulácia látky musí zbehnúť *až po* aktualizácii fyziky. Navyše nastavenie stavu látky je závislé na komponente kostry, no aktualizácia látky upravuje kostru.

Ocitáme sa v situácii, kde potrebujeme zaručiť beh simulácie látky, avšak až po dokončení simulácie fyziky. Navyše, komponent kostry môže počas aktualizácie komponentov vyžadovať viacero volaní nastavenia póz. Pre predstavu môže animačný komponent nastaviť pózu animácie tváre, animácie postavy alebo dokonca pózu procedurálne generovanej animácie. Tieto akcie vytvárajú implicitné závislosti medzi poradím volania komponentov.



Obr. 1.4 Diagram závislostí aktualizácie komponentov.

Tieto aktualizácie potrebujeme naplánovať. Najväčším problémom je, že tieto vzťahy sú skryté. Ako vyriešiť tento problém? Znova manuálne inicializovať komponenty ako v objektovom modeli? Pridať numerickú prioritu každému typu komponentu, čím zaručíme správne poradie behu aktualizácií? Čo ak by sme potrebovali špeciálnu prioritu pre špecifický herný objekt? Príkladom môže byť unikátne správanie špeciálneho typu nepriateľa, kedy jeho komponenty spúšťame v inom poradí. V každom riešení sa vynárajú problémy. Bobby ako príklad uvádza ukážky kódu.[1]

Výpis kódu 7 Explicitná aktualizácia komponentov:

```
public void UpdateComponents() {
    var weapon = GetObject().GetComponent<Weapon>();
    if (weapon is null) return;
    var health = GetObject().GetComponent<Health>();
    if (health is null) return;
    var knowledge = GetObject().GetComponent<Knowledge>();
    if (knowledge is null) return;

    etc...
    // Execute actual logic
```

[1] Ukážky sú upravené do jazyku C#.

1.4.3 Zhrnutie

Hlavnou úlohou objektovo komponentového modelu je zaistiť modulárne správanie medzi komponentami a objektmi a zaviesť znovupoužiteľnosť kódu naprieč projektami. Z dôvodov implicitných závislostí sa idea modelu rozbíja, komponenty zodpovedajú za viac funkcionalít a zakázaním ich sa vraciame k objektovému modelu. Navyše uviesť logiku komponentov do viacvláknového prostredia je náročné. V hernej slučke nevieme lokalizovať dobu kedy budú komponenty navzájom interagovať.

Objektovo komponentový model ponúka príjemnú abstrakciu nad herným svetom, no v praxi zlyháva ponúknuť skutočnú modularitu a umožnením voľného referencovania dát v systéme uvádza množstvo problémov. Bobby, o ktorého sa v priebehu príkladov opierame, taktiež vyzdvihuje, že tieto problémy sú nesmierne časté najmä k blížiacemu sa deadlinu počas vývoja hier.

Aj napriek nevýhodám tento model ponúka dostatočne flexibilnú implementáciu systémov na tvorbu hier. Tieto systémy v dnešnej dobe označujeme pojmom herné enginy. Implementácie tohto modelu sa v praxi samozrejme mierne líšia, avšak väčšina konvenčných herných enginov ako napríklad Unity, Unreal Engine a Cryengine používa práve objektovo komponentový model. [7]

1.5 Entity Component System

Na rozdiel od objektového a objektovo komponentového modelu, Entity Component System, v skratke ECS, opisuje návrhový vzor oddeľujúci dáta od logiky. Spôsob implementácie konceptov vzoru nie je jasne definovaný, vzor stanovuje funkcionality, ktoré má ECS spĺňať. Realizácia vzoru sa líši naprieč frameworkmi.

Štruktúra pozostáva z troch častí:

- Entita reprezentuje generický herný objekt, tak ako v objektovo komponentovom modeli. Rozdielom je štruktúra entity. Entita v ECS slúži len ako identifikátor a neobsahuje žiadnu logiku ani dáta. Implementácie typicky využívajú jednoduchý celočíselný typ ako entitu.

- Komponent je priradený ku danej entite. Priradeniami entita modeluje dáta. Pre entitu z pravidla platí, že môže obsahovať ľubovoľnú množinu unikátnych komponentov. Tie sa dokážu dynamicky pridávať, upravovať a odoberať. Komponentami rozumieme prosté dátové typy bez akejkoľvek logiky. Oproti komponentom v objektovo komponentovom modeli tieto komponenty nie sú priamo komponované hernými objektami, ale riadené frameworkom.

- Systém umožňuje uskutočnenie procesov, ktoré sa spustia na všetkých entitách s vybranými vlastnosťami. V technickom slova zmysle sú to statické funkcie, ktoré transformujú komponenty entít. Množina komponentov entít zodpovedá požiadavke systému, čím filtrujeme výber. Ako príklad môže fyzikálny systém vyžiadať všetky entity majúce hmotnostné, rýchlostné a polohové komponenty a transformovať ich na základe fyzikálnych výpočtov.

V praxi sú princípy ECS voľnejšie, pričom niektoré frameworky neimplementujú systémy a len ponúkajú metódy na dopytovanie entít.

1.5.1 Motivácia

Herní vývojári sa obracajú na ECS najmä z týchto dôvodov: [3]

- ECS typicky podporuje väčšie množstvo objektov oproti iným modelom
- Kód ECS má tendenciu byť viac opakovane použiteľný
- Projekt je jednoduchšie rozšíriť o nové funkcie
- ECS umožňuje dynamickejší štýl kódovania

Zo zdrojov ohľadom ECS sú často zdôrazňované výkonnostné výhody. Pokiaľ nepracujeme na masívnych AAA hrách[2] je hlavným dôvodom použitia ECS organizácia kódu. Výkon je podstatný faktor vývoja hier, ale vhodne organizovaný kód ma vo vývojárskom tíme nevyčísliteľnú hodnotu. Vo veľkom množstve hier nie je obmedzujúcim faktorom výkon ani s OOP alebo inými implementáciami komponentových modelov. [9]

Michele Caini[3] vyzdvihuje tento fakt svojim tvrdením, že „programovanie založené na komponentoch je neuveriteľne výkonným nástrojom, ktorý robí kód flexibilným a urýchľuje iterácie počas vývoja. Toto musí byť váš prvý cieľ, nič iné.“ [9]

Vo svojom blogu opisuje proces prechodu od OOP ku ECS. Prezentuje dva pohľady na herný svet. Doména herného sveta obsahuje objekty od hráčovej postavy cez stromy až po trpaslíkov, všetky zastúpené v zozname. Prvý pohľad dekonštruuje svet vertikálne, čím získame koncepty objektov. V OOP sú tieto koncepty reprezentované triedami. Vodorovný rez rozdelí svet na pozície, textúry, modely, konečné automaty a tak ďalej. Vizualizovať

[2]Pojmom AAA hra označujeme vysoko rozpočtovú počítačovú hru tvorenú spravidla okolo 120 osobami, po dobu okolo 18 mesiacov s rozpočtom približne 30 miliónov dolárov. [8]

[3]Tvorca EnTT, open source C++ ECS knižnice použitej aj pri tvorbe hry Minecraft.

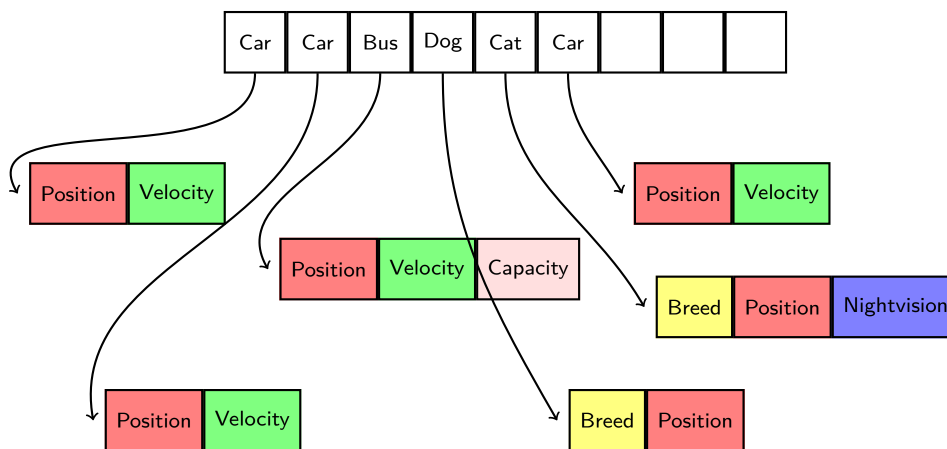
si to vieme ako tabuľku, pri ktorej stĺpce reprezentujú objekty a riadky ich komponenty. Každý riadok je priradený ku danému typu komponentu, čím sa v tabuľke objavujú prázdne miesta.

Klasický OOP sa spolieha na iteráciu všetkých herných objektov, ktorá ich periodicky aktualizuje konkrétnou metódou. Podľa predchádzajúceho príkladu postupne prechádzame stĺpcami herného sveta a celý stĺpec aktualizujeme naraz. ECS obracia problém naruby a iteruje riadkami. Táto iterácia je zabezpečená spúšťaním systémov. Systém teda prechádza po riadku, na ktorých vykonáva svoju špecifickú úlohu.

V zjednodušenom popise si predstavme ECS ako tabuľku, kde stĺpce zastupujú entity, riadky reprezentujú typy komponentov a bunky sú aktuálne existujúce komponenty priradené ku entite. Systémy vykonávajú vhodné výbery v tabuľke a transformujú dáta.

1.5.2 Pamäťový model

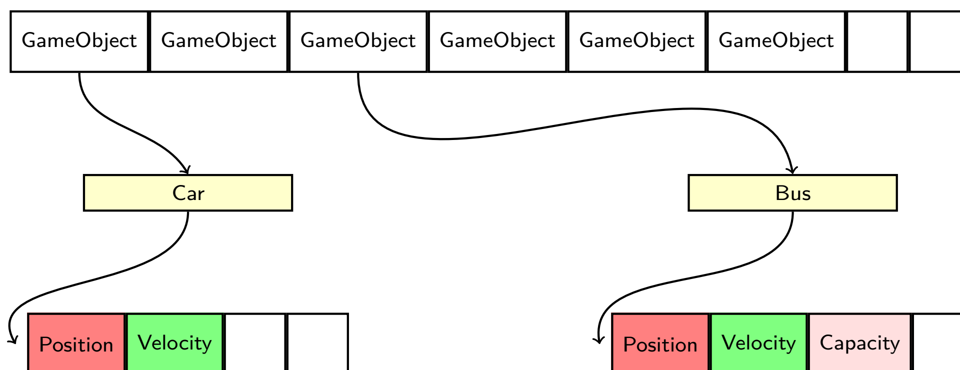
V následujúcom kroku nahliadneme na internú štruktúru objektov spomínaných modelov za účelom pozorovania zmien v modeli pri prechode na ECS. Objektový model ukladá objekty na halde, bez záruky akéhokoľvek usporiadania. Počas hernej slučky iterujeme tieto objekty a aktualizujeme ich. Môžeme si povšimnúť, že objekty nedbajú na cache, taktiež pre manipuláciu aj len s jedným komponentom čítame celý objekt. Náhodnosť usporiadania objektov môže mať následky na výkone hry.



Obr. 1.5 Herný svet v objektovom modeli.

Objektovo komponentový model má rovnaké problémy ako objektový model. Navyše môžeme pozorovať ešte nepriamejší spôsob získania dát. Kom-

ponenty sú reprezentované generickými objektami v poli. Na ich prístup potrebujeme o vrstvu dereferencií viac. Navyše bez záruky známej lokácie komponentu v poli ho musíme prehľadať a zaneprázdniť systém.



Obr. 1.6 Herný svet v objektovo komponentovom modeli.

V ECS nemáme dôvod iterovať všetkými entitami. Každý systém má explicitne špecifikované dáta, na ktorých pracuje. Dáta sú modelované tak, aby systém nemusel skákať v pamäti. Na zabezpečenie tejto vlastnosti frameworky využívajú rôzne dátové štruktúry. Ďalej spomenutý zoznam menuje a vysvetľuje najpopulárnejšie pristúpy naprieč implementáciami. [3]

Archetype ECS založený na archetypoch ukladá entity v tabuľkách, kde komponenty reprezentujú stĺpce a riadky sú zastúpené entitami. Tento spôsob pripomína štruktúru relačných databáz. Databáza v našom prípade obsahuje všetky objekty rozdelene podľa schémy do daných tabuliek, archetypov. Archetypový systém umožňuje konštantne vyhľadávanie a rýchlu iteráciu entít. Nevýhodou je pridávanie a odoberanie komponentov v lineárnom čase.

Sparse set Implementácia pomocou riedkych množín ukladá každý typ komponentu do vlastnej množiny, ktorú indexujeme identifikátorom entity. Každý komponentový typ vytvára jediné pole komponentov. Pomocou vhodných dátových štruktúr existuje mapovanie medzi entitami a komponentami. Implementácie riedkych množín umožňujú rýchle pridanie a odobratie komponentov výmenou za pomalšiu iteráciu entít.

Bitset Bitsetový spôsob organizácie dát ukladá entity v poliach, pričom identifikátor entity slúži ako index. Polia spolupracujú s bitsetom, ten indikuje či entita obsahuje daný komponent. Jeden zo spôsobov implementácie bitsetu je mať pole pre každý typ komponenty so sprevádza-

júcim bitsetom. Mertens opisuje bitsetovú organizáciu dát pomocou dátovej štruktúry hibitset. [4]

Reactive Reaktívny ECS sleduje mutácie entít. Kontroluje kedy sa entita vytvorí, odstráni, alebo sa upraví jej komponent. ECS signalizuje dané zmeny a kombinuje entity do vhodných skupín. Užívateľ reaguje na tieto skupiny a upravuje ich. Interpretácia reaktívneho spôsobu organizácie dát je rôzna naprieč frameworkmi.

1.5.3 Archetype

Tak, ako herné enginy všeobecne, aj tieto prístupy majú svoje výhody a nevýhody. Rozhodnutie ktorý typ ECS využiť závisí od špecifického problému. V našom hernom frameworku využívame práve archetypovú dátovú štruktúru. Z tohto dôvodu si ju na príklade ukážeme. Ako prvé vysvetlíme problém skompaktnenia polí. Majme herný svet s 3 entitami obsahujúce jediný typ komponentu. To znamená, že všetky entity vieme reprezentovať v jednom poli.

```
A a[3];
```

Vo svete sa nachádzajú 3 entity. Každá obsahuje práve jeden komponent a to typu A. Entita je reprezentovaná identifikátorom, v tomto prípade to je index do poľa, preto entita 2 má dáta uložené na indexe 2. Po pridaní nového typu komponenty entitám získavame tento stav sveta:

```
A a[3];
B b[3];
```

Znova platí, že pre entitu 2 sú komponenty v oboch poliach uložené na indexe 2. Polia sú súvislé, kompaktné a vektorizovateľné. Komplikácia vzniká, ak upravíme prvú entitu odobratím komponenty B. Herný svet teda obsahuje následovné polia:

```
A a[3];
B b[2];

// Entity
0: [A B]
1: [A  ]
2: [A B]
```

[4] Hibitset umožňuje hierarchické bitové množiny, umožňuje rýchlu iteráciu v riedkych dátových štruktúrach. Bližšie informácie o dátovej štruktúre vieme nájsť v dokumentácii programovacieho jazyka Rust. `https://docs.rs/hibitset/0.6.3/hibitset/`

Mapovanie pomocou indexov nefunguje, našťastie vieme pridať invariant, ktorý zoradí entity podľa toho, či obsahujú komponentu B. Ostáva nám vytvoriť nové mapovanie entít na index. Vyžadované vlastnosti sú zachované.

```
// Entity
0: [A B]
2: [A B]
1: [A  ]

// Mapovanie (index -> ID entity)
0 -> 0
1 -> 2
2 -> 1
```

Toto riešenie avšak nie je udržateľné. Pridaním ďalšej komponenty sme nútení mať voľné miesta v poliach. S podmienkou držať všetky komponenty daného typu v jednom poli je problém nevyriešiteľný.

```
// Entity
0: [  B  ]
1: [  B C]
2: [A B C]
3: [A B  ]
4: [A    ]
5: [A   C]
6: [    C]
```

Namiesto vytvorenia jedného poľa pre každý komponent vytvorme polia pre každú unikátnu množinu komponentov. Každá entita patrí do takzvaného archetypu, ktorý je definovaný množinou jej komponentov. Entity pomocou archetypov zoradíme následovne.

```
// Archetyp [A]
A a[1];
// Archetyp [B]
B b[1];
// Archetyp [C]
C c[1];
// Archetyp [A, B]
A a[1];
B b[1];
// Archetyp [A, C]
```

```
A a[1];
C c[1];
// Archetyp [B, C]
B b[1];
C c[1];
// Archetyp [A, B, C]
A a[1];
B b[1];
C c[1];
```

Archetypy zachovávajú všetky vlastnosti, ktoré od našich podkladových dátových štruktúr vyžadujeme. V jednoduchšom príklade si predstavme nasledovný herný svet:

```
// Archetyp [A]
A a[2];
// Archetyp [A, B]
A a[2];
B b[2];
// Archetyp [A, C]
A a[2];
C c[2];

// Entity
0: [A]
1: [A]
2: [A B]
3: [A B]
4: [A C]
5: [A C]
```

Vytvorením viacerých polí pre jeden typ komponentu vyžaduje dodatočnú réžiu pri iterácií, avšak umožňuje nám jednoducho filtrovať entity a taktiež paralelizovať výpočty systémov. Ak systém žiada všetky entity obsahujúce určité komponenty, vyhľadávanie nemusí prebiehať na úrovni entít, ale až na úrovni archetypov. V archetypoch využívame princíp SoA[5] kvôli tomu, že systémy sa v praxi často dopytujú len na výber komponentov entity, čo výrazne zefektívni napĺňanie cache line. Na príklade si ukážme výhody archetypového systému.

[5]SoA, alebo Struct of Arrays je dátovo orientovaný vzor pozostávajúci z ortogonálneho rozdelenia dát. Namiesto poľa štruktúr rozdeľujeme vlastnosti do samostatných polí.

```
1. Archetyp [Position, Velocity, Age, Player]
2. Archetyp [Position, Velocity, Age]
3. Archetyp [Position, Age]
4. Archetyp [Weapon, Age, Rarity]
5. Archetyp [Player, Weapon, Rarity]
6. Archetyp [Weapon]

1. Systém [Age]
2. Systém [Position, Velocity]
3. Systém [[ReadOnly]Player, Position]
4. Systém [Weapon, [ReadOnly]Rarity, !Player]
5. Systém [Weapon, [ReadOnly]Rarity, [ReadOnly]Player]
```

Ako prvé si vysvetlíme, ktoré entity budú spracovávané systémami. Systém `[Age]` zachytáva všetky entity z archetypov 1, 2 a 3. Systémy sa spúšťajú na všetkých entitách, ktoré vyhovujú predikátu z filtru. Z rovnakého princípu druhý systém spracováva všetky entity archetypov 1 a 2. Tretí systém sa mapuje len na prvý archetyp. Pri posledných systémoch si všímame rozdiel jedine v komponente hráča. Systém `[Weapon, Color, !Player]` platí len na entity štvrtého archetypu. Analogicky funguje posledný systém. Posledný archetyp nemá systém, ktorý by s ním interagoval.

Každý systém vykonáva vlastnú funkcionalitu. Prvý systém aktualizuje vek danej entity, druhý simuluje pohyb a podobne. Počas hernej slučky potrebujeme tieto systémy spúšťať. Triviálnym riešením je spustiť systémy sekvenčne. Na príklade si môžeme povšimnúť, že systémy vieme rozdeliť na nezávislé časti.

```
// 1. [Age]
Archetyp [1, 2, 3, 4]

// 2. [Position, Velocity], [[ReadOnly]Player, Position]
    -> spoločná závislosť [Position]
Archetyp [1, 2] [1]

// 3. [Weapon, [ReadOnly]Rarity, !Player]
Archetyp [5]

// 4. [Weapon, [ReadOnly]Rarity, [ReadOnly]Player]
Archetyp [4]
```

ECS jasne definuje dátové závislosti systémov, v tomto príklade nevidíme závislosti medzi skupinami. Systémy v skupinách teda môžeme bezstarostne

paralelizovať. Cieľové entity nemajú prienik medzi systémami. Zobrazme si porovnanie behu systémov.

```
Plynutie času:
------------------>

Pôvodné riešenie:
[1][2][3][4][5]

Nové riešenie:
[1]
[2][3]
[4]
[5]
```

To, že môžeme paralelizovať štvrtý a piaty systém je zásluhou archetypov. Ten nám delí entity na 2 samostatné skupiny. Archetypový systém teda slúži nie len na tvorbu kompaktných polí komponentov, ale taktiež dovoľuje efektívnu paralelizáciu systémov. Práve z tohto dôvodu v tejto práci implementujeme archetypový spôsob organizácie dát.

1.5.4 Nevýhody

Entity Component System neslúži ako riešenie všetkých problémov tvorby hier. Písanie kódu v princípoch ECS si vyžaduje masívny posun v spôsobe premýšľania nad hrou, čo spôsobuje kognitívnu zátaž, najmä pre menej skúsených programátorov. Najväčším problémom je absencia priestorovej hierarchie. [7] S entitami je nutné pracovať bez komfortu relatívnych prepojení. Nakoniec, keďže ECS ponúka len usmernenia ako ho implementovať sa rôzne frameworky líšia. Neexistuje preto univerzálny konceptuálny model tvorby hry v ECS.

Kapitola 2

Lepenka

Lepenka stavia na princípoch ECS a ponúka riešenie založené na archetypovom spôsobe organizácie dát. Ponúka predvolené paralelné plánovanie systémov. Užívateľovi stačí vhodne anotovať systémy a framework vygeneruje paralelnú hernú slučku umožnenú za pomoci dátovo orientovanej architektúry systému.

Názov Lepenka plynie z modularity frameworku. Princípy ECS umožňujú oddeliť systémy a komponenty do samostatných častí, modulov. Tie sa môžu vzájomne „lepiť". Prepájaním a dopĺňaním funkcionalít pomocou modulov rastie využitie frameworku.

Framework bol navrhnutý pre tvorbu hier na platforme .NET. Dôvodom implementácie frameworku v jazyku C# nie je len kvôli popularite jazyka pre tvorbu hier.[1] Výhodou je podpora pre generáciu kódu v kompilačnom čase a výkonná paralelná architektúra. Dôležitou súčasťou frameworku je integrácia s generátorom kódu. Generátor slúži ako odporúčaná nadstavba nad ECS a generuje hernú slučku z definovaných systémov, komponentov, modulov a konfigurácie.

V tejto kapitole si vysvetlíme všetky funkcie frameworku. Lepenka implementuje princípy ECS spomenuté v sekcii 1.5. Informácie ako pripraviť projekt využívajúci Lepenku si priblížime v dodatku A.

2.1 Štruktúra projektu

Projekt obsahuje definície komponentov, systémov a dodatočné konfigurácie behu hry. Konvenciou väčších projektov je vytvoriť samostatné priečinky

[1] Jazyk C# sa využíva na tvorbu väčšiny hier pomocou populárnych herných enginov ako napríklad Unity [1], Godot, alebo frameworku MonoGame.

Components a *Systems*. Narozdiel od konvenčných[2] herných frameworkov Lepenka vytvára hernú slučku a inicializuje hru automaticky. Užívateľ nastavuje jedine logiku a správanie behu systémov, ktoré framework vhodne spúšťa. Z toho dôvodu je kód projektu minimálny, bez „boilerplate" kódu. Znovupoužitie systémov je umožnené pomocou modulov. Koncept modulov je bližšie dokumentovaný v kapitole Moduly 2.7.

2.2 Konfigurácia projektu

Odporúčaný spôsob ako začať pracovať s frameworkom je využiť projektovú šablónu. Šablóna obsahuje plne konfigurovaný projekt pripravený na použitie. Šablóna je zdieľaná v repozitári NuGet balíčkov. Po jej inštalácii sa v indexe projektov zobrazia projekty frameworku Lepenka. Bližšie technické informácie o konfigurácii projektu sa nachádzajú v dodatku A. V ďalšej časti sú priblížené konfigurácie predpripravené v šablóne. Tie je nutné vykonať v prípade manuálnej tvorby projektu.

Dôležitou súčasťou využívania frameworku Lepenka je integrácia generátoru kódu do projektu. Platforma .NET ponúka možnosť generácie zdrojového kódu zapojením sa do kompilačného procesu. Lepenka využíva 2 generátory, prvý je určený pre tvorcov modulov a automaticky exportuje metadáta do XML súborov. Druhý je využívaný projektom používajúcim vesmír.

Výpis kódu 8 Projekt využívajúci generáciu kódu:

```
// UniverseGenerator
<ItemGroup>
    <PackageReference Include="Lepenka.Generator"
        Version="0.0.X"
        OutputItemType="Analyzer"
        ReferenceOutputAssembly="false" />
</ItemGroup>

// ModuleGenerator
<ItemGroup>
    <PackageReference Include="Lepenka.Generator.Module"
        Version="0.0.X"
        OutputItemType="Analyzer"
        ReferenceOutputAssembly="false" />
</ItemGroup>
```

[2] Myslíme tým iné ECS frameworky napísané v jazyku C#. Porovnanie fungovania bolo vykonané s 5 najpopulárnejšími frameworkami na platforme GitHub.

Oba balíčky sú zdieľané správcom balíkov NuGet. Pre správne využitie generátorov potrebujeme projekt konfigurovať podľa spomínaného príkladu.

Aby sa korektne vygenerovala herná slučka potrebuje projekt obsahovať súbor `Universe.xml`. Ten obsahuje názov generovaného vesmíru rovnako ako zoznam závislostí na správny chod programu.

Výpis kódu 9 Príklad súboru `Universe.xml`:

```
<?xml version="1.0" encoding="utf-8"?>
<Universe Name="NetworkUniverse">
    <Imports>
        <Using>System.Net</Using>
        <Using>System.Net.Sockets</Using>
    </Imports>
</Universe>
```

Atribút názvu vesmíru určuje v akej triede sa vygeneruje herná slučka. Po správnom nastavení projektového súboru je konvenciou využívať top-level statement verziu funkcie `Main`. Projektový súbor spúšťajúci hru je spravidla následovný:

Výpis kódu 10 Ukážka funkcie `Main`:

```
using ProjectExample;

await new UniverseName()
    .Run();
```

2.3 Komponent

Komponenty mapujú všetky dáta v hre. Z princípov ECS rozumieme entite ako hernému objektu zastrešujúcemu komponenty. V Lepenke sú komponenty reprezentované štruktúrami. Tvorba komponentov a priraďovanie entít prebieha v systémoch. Užívateľ nevytvára štruktúry manuálne, ale pomocou preddefinovaných príkazov. Komponenty by nemali obsahovať žiadnu logiku a mali by byť štruktúrované z primitívnych dátových typov. Písanie štruktúr podľa týchto pravidiel umožní rýchlej iterácii entít v hernej slučke. Výhodou sú aj možnosti jednoduchej serializácie komponentov, čo zjednoduší proces posielania informácii cez sieť. Lepenka avšak neobmedzuje charakter štruktúr a užívateľ má povolené ich modelovať akokoľvek.

Výpis kódu 11 Ukážka komponentov:

```
public struct Position
{
    public int X;
    public int Y;
    public int Z;
}

public struct Enemy
{
    public int Damage {get; set;}
    public int MaxHealth {get; init;}
    public int Health {get; set;}
}
```

2.4 Systém

Všetky procesy v rámci hry sú uskutočnené vďaka systémom. Používateľ importuje alebo vytvára verejné statické funkcie. Tieto funkcie slúžia ako herná logika, riadenie procesov vykresľovania grafiky, prehrávania zvukov alebo posielania informácií cez sieť. Tvorba entít v hernom svete je taktiež vykonávaná pomocou týchto systémov.

Lepenka spústa systémy opakovane každý cyklus hernej slučky. Na dodatočnú špecifikáciu poradia, v akom sa majú systémy spúšťať, využíva framework takzvané etapy. Každý systém je zaradený do práve jednej z etáp, alebo patrí medzi štartovacie systémy.

Výpis kódu 12 Príklad etáp:

```
[Stage]
internal enum Stage {
    BeforeRound,
    Update,
    AfterRound
}
```

Enumeráciu označíme anotáciou Stage, čím frameworku oznámime, do ktorých etáp systémy radíme. Pre hru táto konfigurácia znamená, že po inicializácii hry bude herná slučka opakovať systémy v poradí enumerácie. Ako prvé teda *súčasne* zbehnú všetky systémy etapy `BeforeRound`, pokračujúce systémami etapy `Update` a nakoniec zvyšné systémy etapy `AfterRound`.

Cyklus sa následne opakuje. Každý systém podlieha práve jednej statickej triede. Tá obsahuje statické funkcie reprezentujúce systémy. Tento systém framework naplánuje na spustenie v etape `Update` v každom cykle hernej slučky.

2.4.1 Globálny systém

Ako prvý typ systému radíme takzvaný globálny systém. Ten sa spúšťa vždy práve raz za cyklus hernej slučky. Neviaže sa na žiadnu z entít a ani ich nedokáže priamo ovplyvňovať. Príklad jednoduchého systému zostavíme následovne.

Výpis kódu 13 Príklad systému:

```
public static class GameSystems
{
    [System<Stage>(Stage.Update)]
    public static void ExampleSystem()
    {
        Console.WriteLine("Hello from ExampleSystem!");
    }
}
```

2.4.2 Inicializačný systém

Taktiež je možné vytvoriť inicializačný systém, ktorý sa spúšťa na začiatku hry. V tom prípade nám môže záležať na priorite spustenia systémov. Prioritu voliteľne definujeme v anotácii. Lepenka zozbiera všetky inicializačné systémy a zoradí ich podľa priority tak, že systémy bez explicitnej priority spustí ako posledné.

Výpis kódu 14 Štruktúra inicializačných systémov:

```
public static class GameSystems
{
    [StartupSystem]
    public static void ExampleStartupSystem()
    {
        Console.WriteLine("Startup systems only run once.");
    }

    [StartupSystem(priority: 1)]
    public static void PriorityStartupSystem1()
    {
        Console.WriteLine("This system runs first.");
    }

    [StartupSystem(priority: 2)]
    public static void PriorityStartupSystem2()
    {
        Console.WriteLine("This system runs second.");
    }
}
```

2.4.3 Entitový systém

Posledným typom systémov sú entitové systémy. Transformujeme nimi entity herného sveta. Entity filtrujeme pomocou anotácií. Pre entitový systém platí, že sa spustí práve raz pre každú entitu spĺňajúcu filter daného systému. Entity je možné filtrovať na základe komponentov. Lepenka ponúka filtre `With` a `Without`. Ku komponentom pristupujeme pomocou parametrov systému s použitím kľúčového slova `ref`.

Výpis kódu 15 Príklad entitových systémov:

```
public struct Position
{
    public int X {get; set;}
    public int Y {get; set;}
}

public struct Velocity
{
    public int Value {get; set;}
}

public struct Stunned {}

public static class GameSystems
{
    [System<Stage>(Stage.Update)]
    [With<Position, Velocity>]
    [Without<Stunned>]
    public static void UpdatePosition
        (ref Position pos, ref Velocity vel)
    {
        pos.X += vel.Value;
        pos.Y += vel.Value;
    }

    [System<Stage>(Stage.Update)]
    [With<Stunned>]
    public static void InformStunnedStatus()
    {
        Console.WriteLine("This entity does not move.");
    }
}
```

Pristupovať môžeme len ku komponentom, ktoré sú anotované filtrom `With` a zároveň nie `Without`. Toto riešenie umožňuje flexibilne dopytovať typy entít a špecializovať systémy na základe komponentov.

2.4.4 Systém špecifického vlákna

Všetky systémy je možné delegovať na samostatné vlákno anotáciou `Thread`. Používateľ by mal túto možnosť využiť len ak je to nutné. Lepenka zozbiera systémy daného vlákna a vytvorí špeciálnu hernú slučku pre tieto systémy. Tak ako pri etapách, používateľ vytvorí špeciálnu enumeráciu, na ktorú sa odkazuje v anotácii systému.

Výpis kódu 16 Príklad systémov samostatného vlákna:

```
[ThreadLayout]
internal enum Threads {
    Render
}

public static class GameSystems
{
    [System<Stage>(Stage.Update)]
    public static void MainThreadSystem()
    {
        Console.WriteLine("Runs on the main thread!");
    }

    //Startup System
    [StartupSystem]
    [Thread<Threads>(Threads.Render)]
    public static void RenderInit()
    {
        Graphics.InitWindow();
    }

    //Global System
    [System<Stage>(Stage.BeforeRound)]
    [Thread<Threads>(Threads.Render)]
    public static void RenderUI()
    {
        Graphics.DrawText(10, 10, "UI from other thread");
    }

    //Entity System
    [System<Stage>(Stage.AfterRound)]
    [Thread<Threads>(Threads.Render)]
    [With<Position>]
    public static void RenderEntity(ref Position pos)
    {
        Graphics.DrawCircle(pos.X, pos.Y, Color.Blue);
    }
}
```

2.5 Systémové príkazy

Lepenka ponúka parametre typu `EntityCommand` a `IUniverseCommand`. V prípade použitia týchto parametrov v systéme framework vloží špeciálny typ umožňujúci dodatočnú manipuláciu s hrou. Tieto typy nazývame prí-

kazmi. Medzi funkcionalitu patrí vytváranie nových entít alebo zdrojov, či manipulácia s entitou.

Výpis kódu 17 Vloženie príkazov do systému:

```
[System<Stage>(Stage.Update)]
[With<A>]
public static void
ExampleSystem(IUniverseCommand universeCommand,
              EntityCommand entityCommand)
{
    // Use commands here
}
```

2.5.1 `IUniverseCommand`

Príkaz `IUniverseCommand` je možné vložiť do ľubovoľného systému. Jeho funkcionalita je nasledovná:

`EndUniverse` Po zavolaní tejto funkcie sa dokončí súčasný cyklus hernej slučky a hra sa ukončí.

`CreateEntity` Vytvorí entitu so špecifikovanými parametrami, vyhľadateľná systémami bude až pre následujúcu fázu hernej slučky. Funkcia obsahuje varianty pre rôzne počty komponentov. Parametre musia byť komponenty, teda štruktúry.

`CreateResource` Vytvorí zdroj so špecifikovanými dátami. Tento zdroj je zdieľaný pre všetky systémy a entity herného sveta. 2.6

Výpis kódu 18 Ukážka príkazov v systéme:

```
[StartupSystem]
public static void
Startup(IUniverseCommand command)
{
    var client = NetworkClient.Create("127.0.0.1", 216);

    if (!client.IsConnected)
    {
        command.EndUniverse();
    }

    command.CreateEntity(new Player(), new Health(10));
    command.CreateEntity(new Enemy());
    command.CreateResource(client);
}
```

2.5.2 EntityCommand

Príkaz typu `EntityCommand` je použiteľný len v entitových systémoch. Jeho úlohou je interagovať priamo s entitou a upravovať jej vlastnosti. Po akejkoľvek úprave entity pomocou príkazu `EntityCommand` sa akcia naplánuje, ale kvôli zachovaniu konzistencie entít medzi systémami sa vykoná až po dokončení etapy hernej slučky. Počas jednej etapy môže nastať maximálne jedna transformácia entity pomocou príkazu. Rovnako ako v prípade tvorby entity, príkazy obsahujú varianty metód pre rôzne počty parametrov. Medzi možné úpravy patrí:

`AddComponent` Priradí ďalší typ komponentu ku entite so špecifikovanými dátami.

`RemoveComponent` Odstráni entite špecifikovaný typ komponentu.

`ModifyComponent` Pridá a zároveň odstráni špecifikované množiny komponentov. Kombinácia príkazov `AddComponent` a `RemoveComponent`.

`RemoveEntity` Označí entitu na odstránenie z herného sveta.

Výpis kódu 19 Ukážka entitového príkazu:

```
[System<Stage>(Stage.Update)]
[With<A, B>]
public static void
ExampleSystem(EntityCommand command)
{
    // Only one command action may happen per stage.
    switch (GetRandomChoice())
    {
        case 1:
            command.AddComponent(new C {Value = 10});
            break;
        case 2:
            command.RemoveComponent<A, B>();
            break;
        case 3:
            command.ModifyComponent<C, Remove<B>>
                                    (new C {Value = -1});
            break;
        case 4:
            command.RemoveEntity();
            break;
    }
}
```

2.6 Zdroj

Pri tvorbe hry často vyžadujeme dáta, ktoré nepasujú do definície entít. Jedná sa o zdroje obrazových dát, zvukov, textúr, alebo jednoducho objektov, ktoré sú viazané na hru ako celok a nie na špecifický herný objekt. V Lepenke využívame označenie `Resource` pre ľubovoľný objekt prístupný z akéhokoľvek systému. Tento objekt je zdieľaný medzi všetkými entitami a je obmedzený na práve jeden pre daný typ triedy. Ku zdrojom systémy pristupujú špecifikovaním typu zdroja v parametri funkcie. Pre každý typ zdroja môže existovať nanajvýš jedna jeho inštancia. Vďaka tomu je prístup ku zdrojom priamy a intuitívny.

Výpis kódu 20 Ukážka využitia zdroja v systéme:

```
public class ExampleResource
{
    public int MaxPlayers {get; init;}
}

public static class ExampleSystems
{
    // Resource creation
    [StartupSystem]
    public static void Startup(IUniverseCommand command)
    {
        command.CreateResource(
            new ExampleResource {MaxPlayers = 4}
        );
    }

    // Resource usage
    [System<Stage>(Stage.BeforeRound)]
    public static void PrintPlayerCount(ExampleResource res)
    {
        Console.WriteLine(res.MaxPlayers);
    }
}
```

2.7 Tvorba modulov

Moduly slúžia na vytvorenie samostatných častí programu, ktoré sa pomocou Lepenky vhodne prepájajú. Bežnou úlohou modulov je zabezpečiť funkčnosť rôznych domén projektu, pri hrách si môžeme predstaviť modul na vykresľovanie, zdieľanie dát pomocou siete alebo simuláciu fyziky. Samotná herná logika môže byť taktiež reprezentovaná ako modul, ktorý ich prepája.

Vo všeobecnosti obsahujú množiny systémov, rozhraní, alebo samotných komponentov. Žiadna z týchto častí avšak nie je povinná, no vzhľadom na závislosti medzi týmito konceptami je bežné implementovať ich všetky.

2.7.1 Systém

Jediné závislosti ECS systémov sú vo frameworku priamo definované v parametroch, alebo filtroch, ktoré systém obsahuje. Parametre sú spravidla komponenty, zdroje, alebo príkazy. V prípade komponentov alebo zdrojov systém nutne nepotrebuje špecifický parameter, len rozhranie, s ktorým bude

interagovať. Lepenka prináša možnosť extrahovať systémy do generických funkcií, čím umožníme integráciu systémov do hernej slučky aj z externých knižníc. Ukážme si jednoduchý príklad:

Výpis kódu 21 Porovnanie interných systémov s modulovými systémami:

```
// Example in main project
[System<Stage>(Stage.BeforeRound)]
[With<A, B, C>]
[Without<D>]
public static void Update
    (Resource r,
    ref A a,
    ref C c,
    IUniverseCommand command)
{
    // System implementation
}

// Example in external module
[ExportSystem]
public static void Update<TResource, TA, TB>
    (TResource r,
    ref TA a,
    ref TB b,
    IUniverseCommand command)
    where TResource : IResource
    where TA : IA
    where TB : TB
{
    // Same system implementation
}
```

Všimnime si, že implementácia systému je rovnaká aj pre systém napísaný v moduli. Na použitie funkcie modulu potrebujeme vhodne doplniť typy do funkcie, preto naše komponenty upravíme pridaním vybraného rozhrania. Abstrahovaním systémov do externej knižnice sme taktiež prišli o informácie filtrov a etapy, v ktorej sa systém spustí. Informácie, ako si zadefinujeme systém drží projekt v XML súbore. Predlohu so správnou schémou pre systém generuje modulový generátor.

Výpis kódu 22 XML zápis systému:

```
<EntitySystem Name="Update" Stage="Draw">
    <Parameters>
        <ResourceParameter Type="Resource">
            <Implementing>IResource</Implementing>
        </ResourceParameter>
        <EntityParameter Type="A">
          <Implementing>IA</Implementing>
        </EntityParameter>
        <EntityParameter Type="B">
          <Implementing>IC</Implementing>
        </EntityParameter>
        <UniverseCommand />
    </Parameters>
    <Filters>
        <With Type="A" />
        <With Type="B" />
        <With Type="C" />
        <Without Type="D" />
    </Filters>
</EntitySystem>
```

Okrem entitových systémov analogicky tvoríme aj globálne a štartovacie systémy. Tie nepodporujú entitové typy parametrov ani filtre. Modul prirodzene deklaruje rozhrania, tie z dôvodu prehľadnosti modulových metadát generátor zapisuje do jednoduchého zoznamu.

Výpis kódu 23 XML zápis rozhraní:

```
<Interfaces>
    <Interface>IWindow</Interface>
    <Interface>IBox</Interface>
    <Interface>IText</Interface>
    <Interface>IColor</Interface>
</Interfaces>
```

2.7.2 Component

Moduly umožňujú tvoriť triedy s vlastným správaním použité v hre, alebo aj v iných moduloch. Tieto triedy herná slučka využíva podľa kontextu ako entitu, alebo ako zdroj. Ako príklad si ukážeme vlastný komponent s priradeným systémom. Tento vzor zaručí správne využitie komponentov.

Výpis kódu 24 Ukážka komponentu v moduli:

```
[ExportComponent]
public class Time
{
    public float GetElapsedTime() => // Implementation

    internal void Restart() => // Implementation
}

[ExportStartupSystem]
public static void CreateTimer(IUniverseCommand command)
{
    command.CreateResource(new Time());
}

[ExportSystem]
public static void RestartTimer(Time time)
{
    time.Restart();
}

// XML
<Components>
    <Component>Time</Component>
</Components>
```

Výhodou je modulárne napájanie častí programu. Modul môže slúžiť aj ako fasáda riadiaca iné moduly, alebo ako knižnica univerzálnych systémov na používanie. V projektoch obsahujúcich viacero modulov je potrebné nakonfigurovať poradie, v ktorom sa budú systémy spúšťať. Rovnako ako pri klasických systémoch, XML zápis umožňuje definovať priority systémov.

2.7.3 Schéma XML

Na korektnú registráciu a spustenie systémov potrebuje generátor vesmíru poznať všetky závislosti v module. Obalením predstavených konceptov informáciami o importoch a konfiguračných názvov získavame finálnu štruktúru XML modulových metadát.

Výpis kódu 25 Príklad XML metadát modulu:

```
<?xml version="1.0" encoding="utf-8"?>
<Module Name="ExampleName" Namespace="ExampleNamespace">
  <Imports>
    <Using>System.Diagnostics</Using>
    <Using>Another.Example.Library</Using>
  </Imports>
  <Systems>
    <StartupSystem Name="CreateTimer" Thread="ExampleThread">
      <Parameters>
        <UniverseCommand />
      </Parameters>
    </StartupSystem>
    <GlobalSystem Name="RestartTimer" Stage="Finish">
      <Parameters>
        <ResourceParameter Type="Time" />
      </Parameters>
    </GlobalSystem>
  </Systems>
  <Components>
    <Component>Time</Component>
  </Components>
  <Interfaces />
</Module>
```

2.7.4 Integrácia modulu v projekte

Pre správnu integráciu modulov potrebujeme oznámiť projektu, aké moduly budeme využívať. Na to definujeme referencie dodatočných súborov v projektovej konfigurácii.

Výpis kódu 26 Pridanie modulov do projektu:

```
<ItemGroup>
    <AdditionalFiles Include="Universe.xml" />
    <AdditionalFiles Include="TcpNetworkClient.xml" />
    <AdditionalFiles Include="Timer.xml" />
    <AdditionalFiles Include="Raylib.xml" />
</ItemGroup>
```

2.8 Integrácia modulov

Ukážky v predchádzajúcich kapitolách počítali s implementáciou systémov a komponentov priamo v projekte. V tejto kapitole si priblížime proces integrácie modulov do projektu. Pri integrácii modulov pracujeme s priradenými metadátovými súbormi integrovanými do projektu. Predstavíme si príklad modulu, s ktorým budeme pracovať naprieč kapitolou.

Výpis kódu 27 Ukážka implementácie modulu:

```
[ExportEntitySystem]
public static void SendMessage<TMessage>
    (ExampleServer server,
    ref TMessage message,
    IUniverseCommand command)
    where TMessage : IMessage
{
    // Implementation
}

[ExportInterface]
public interface IMessage
{
    public byte Type {get;}
    public byte Value {get;}
}

[ExportComponent]
public class ExampleServer
{
    // Implementation
}

[ExportStartupSystem]
public static void InitServer(IUniverseCommand command)
{
    // Implementation
}
```

2.8.1 Prepojenie systémov

Po vložení metadátového súboru do cieľového projektu potrebujeme dodefinovať informácie o systémoch. Systémy modulov sú spravidla štyroch typov:

- Inicializačné systémy exportované atribútom `[ExportStartupSystem]`, v XML dátach pod názvom `StartupSystem`.

- Inicializačné entitové systémy exportované atribútom `[ExportEntityStartupSystem]`, v XML dátach pod názvom `EntityStartupSystem`.
- Globálne systémy exportované atribútom `[ExportSystem]`, v XML dátach pod názvom `GlobalSystem`.
- Entitové systémy exportované atribútom `[ExportEntitySystem]`, v XML dátach pod názvom `EntitySystem`.

Informácie špecifikované v metadátach musia zodpovedať konfiguráciám projektu. To znamená, že názvy etáp alebo vlákien sa musia zhodovať s informáciami v metadátach. Všetkým systémom je možné definovať samostatné vlákno v ktorom sa spustia.

```
<StartupSystem Name="Example1" Thread="ExampleThread">
<EntityStartupSystem Name="Example2" Thread="ExampleThread">
<GlobalSystem Name="Example3" Thread="ExampleThread">
<EntitySystem Name="Example4" Thread="ExampleThread">
```

Pre globálne a entitové systémy je potrebné priradiť etapu.

```
<GlobalSystem Name="SendData" Stage="Update">
<EntitySystem Name="DrawPlayer" Stage="Update" Thread="Render">
```

Všetky entitové systémy vyžadujú špecifikovaný filter. Filter umožňuje vyhľadávať entity obsahujúce daný typ komponenty kľúčovým slovom `With`, alebo vyhľadávať entity neobsahujúce typ komponenty pomocou slova `Without`.

```
<EntitySystem ...>
    <Parameters>
        ...
    </Parameters>
    <Filters>
        <With Type="A" />
        <With Type="B" />
        <Without Type="C" />
    </Filters>
</EntitySystem>
```

Parametre všetkých systémov definujeme vnoreným označením značky `Parameters` do ktorej vkladáme parametre v poradí podľa definície systému. Generátor generuje parametre v správnom poradí. Na špecifikáciu systémov preto stačí dodefinovať typy komponentov.

```
<EntitySystem ...>
    <Parameters>
        <ResourceParameter Type="WindowColor">
            <Implementing>IColor</Implementing>
        </ResourceParameter>
        <EntityParameter Type="Box">
            <Implementing>IBox</Implementing>
        </EntityParameter>
        <UniverseCommand />
        <EntityCommand />
    </Parameters>
    <Filters>
        ...
    </Filters>
</EntitySystem>
```

Označenie `Implementing` slúži len ako informácia pre užívateľa, nie je potrebná pre správnu generáciu hernej slučky. Keďže systémy v moduloch definujeme s generickými parametrami, na ich spustenie je povinnosťou užívateľa explicitne špecifikovať typ komponenty. Parametre `EntityParameter` a `EntityCommand` je možné vkladať len do entitových systémov.

2.8.2 Komponenty implementované modulom

Komponenty, ktoré modul exportuje, sú často sprevádzané systémami na jeho inicializáciu a prácu s ním. Komponenty implementujú funkcionalitu potrebnú vo svojich systémoch. Spoliehajú sa na správnu inicializáciu užívateľom a následne sú vkladané do systémov daného modulu. Pri importe modulu aj s metadátami používateľ nepotrebuje vykonať žiadnu úpravu v kóde na umožnenie používania týchto komponentov. Ak ale komponent vyžaduje manipuláciu cez systémy, tak používateľ je povinný dopísať informácie špecifické pre implementáciu. Spravidla je to etapa v ktorej sa systém spúšťa a prípadne iné komponenty ktoré systémy vyžadujú. Moduly taktiež dokážu implementovať komponenty nezávisle na systémoch. V tom prípade s komponentami pracujeme ako s konvenčnými triedami a štruktúrami, vieme ich implementovať vo vlastných systémoch alebo obaliť do vlastných komponentov, či zdrojov.

2.8.3 Integrácia modulových komponentov

Ak chceme využívať systém implementovaný modulom, musíme implementovať rozhrania parametrov systému. V našom príklade je jediný neimplementovaný

parameter typu `IMessage`. V projekte teda potrebujeme vytvoriť komponent implementujúci dané rozhranie. V niektorých prípadoch môže komponent implementovať viacero rozhraní a pasovať tým do viacerých systémov. Po implementácii komponentu je nutné dodefinovať ho v metadátach ku cieľovému systému. Ak máme viacero typov komponentov, ktoré implementujú rozhranie vyžadované systémom, je povolené spúšťať systém na všetky validné typy.

Výpis kódu 28 Implementácia rozhrania modulu:

```
// Project file
using Module;

public struct Message : IMessage
{
    public byte Type {get; set;}
    publit byte Value {get; set;}
}

// System in XML file
// with added project specific attributes
<EntitySystem Name="SendMessage" Stage="Update">
    <Parameters>
        <ResourceParameter Type="ExampleServer" />
        <EntityParameter Type="Message">
          <Implementing>IMessage</Implementing>
        </EntityParameter>
        <UniverseCommand />
    </Parameters>
    <Filters>
        <With Type="Message" />
        <Without Type="SecretMessage" />
    </Filters>
</EntitySystem>
```

Kapitola 3

Implementácia Lepenky

Jedným z hlavných cieľov pri tvorbe Lepenky bolo navrhnúť ECS s intuitívnym API. Základom je jadro, ktoré obsahuje logiku ECS a pomocné triedy na manažovanie archetypov a filtrovanie entít. Framework absolútne abstrahuje komponenty a systémy ako samostatné kombinovateľné jednotky. Užívateľ preto narozdiel od konvenčného písania kódu hernej slučky deklaratívne opisuje architektúru hry. Definuje komponenty, systémy a takzvané fázy, ktoré s vhodnou anotáciou komponujú modulárne časti hry. Používanie Lepenky oslobodzuje používateľa od implementačných detailov a zameriava sa na deklaráciu samotných funkcionalít hry. Abstrakcia je zabezpečená pomocou generátoru zdrojového kódu. Ten slúži ako odporúčaná, ale nepovinná nadstavba nad ECS, ktorá automatizuje napájanie systémov a komponentov a vytvára hlavnú hernú slučku programu. Generátor vytvára kód zaisťujúci správne spúšťanie systémov, vhodné rozloženie systémov naprieč vláknami a spravuje entity a ich komponenty.

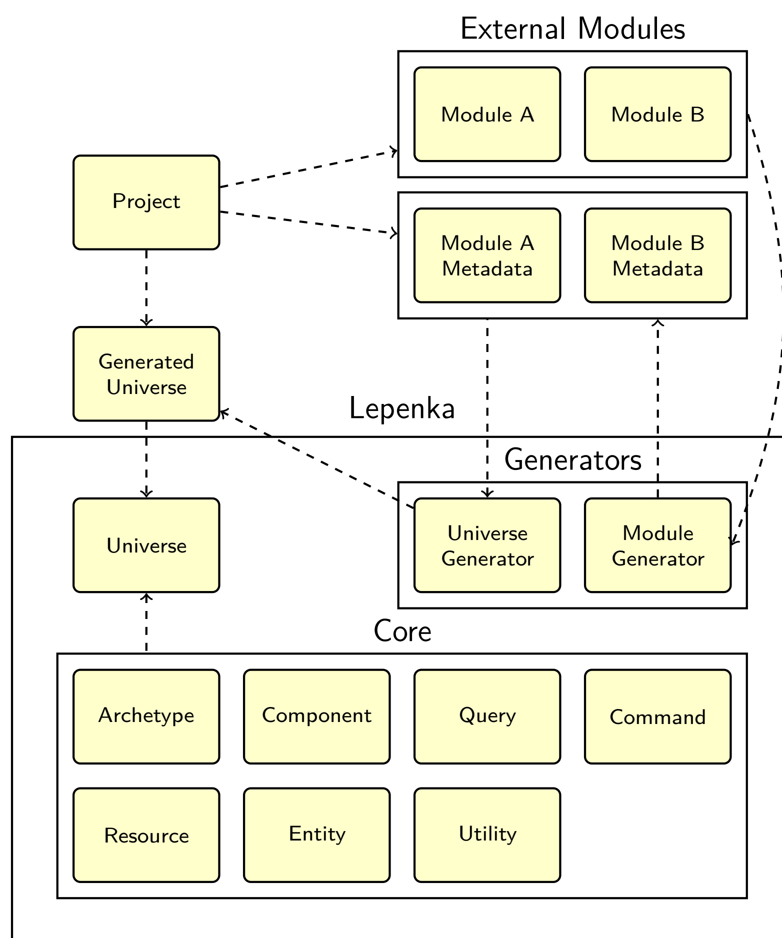
Dôležitým konceptom Lepenky sú modulárne časti ECS, nazývané moduly, ktoré umožňujú import externých systémov a komponentov do projektu. Moduly umožňujú znovupoužiteľnosť herných funkcionalít naprieč projektami. Tvorcovia modulov môžu využiť generátor na extrakciu potrebných metadát z definovaných komponentov alebo systémov v knižniciach.

3.1 Architektúra

Lepenka v jadre obsahuje 7 hlavných logických časti, pričom každá zodpovedá za inú oblasť frameworku. Tieto časti zjednocuje takzvaný *vesmír*, trieda riadiaca hernú slučku. Vesmír slúži ako abstraktná trieda, generátor jej implementáciou vytvára hernú slučku prispôsobenú na systémy danej hry. Pokročilý používateľ má možnosť nevyužiť generátor a implementovať si vlastnú hernú

slučku. Manuálna tvorba hernej slučky nie je doporučená.

Projekt často používa knižnice obsahujúce moduly. Taktiež obsahuje vlastné implementácie hernej logiky v podobe systémov a komponentov. Dodatočne obsahuje konfiguráciu hry a metadáta modulov. Počas kompilácie projektu sa spúšťa generátor. Ten po prehľadaní projektu generuje zdrojový kód s hlavnou slučkou programu. Výstupom je generovaný vesmír ponúkajúci funkciu spustenia hry.



Obr. 3.1 Architektúra frameworku Lepenka.

3.2 Vesmír

Základom herného frameworku Lepenka je trieda `Universe`. Vesmír riadi všetky entity, komponenty a systémy. Na správne fungovanie využíva manažérov špecifických oblastí, ako napríklad komponentov alebo archetypov.

Vesmír obsahuje abstraktnú metódu `Run`. Úlohou metódy `Run` je spustiť hernú slučku so všetkými relevantnými systémami a komponentami. Je dôležité podotknúť, že herná slučka sa buduje na základe generátora kódu. Používateľ taktiež môže nevyužiť generátor a implementovať hernú slučku samostatne. Odporúčaním je avšak generátor využiť. Odteraz budeme ku funkcionalite vesmíru pristupovať z hľadiska vygenerovaného vesmíru. Vesmír obsahuje manažérov, ktorí režírujú vnútorné štruktúry. Herná slučka sa na nich odkazuje a získava od nich herné dáta. V tejto kapitole počítame so znalosťou frameworku Lepenka. Štruktúra generovanej hernej slučky je nasledovná:

1. Inicializuj všetky potrebné dátové štruktúry
2. Spusti štartovacie systémy
3. Spusti entitové štartovacie systémy
4. V cykle opakuj spúšťanie systémov jednotlivých fáz, cyklus ukonči po zachytení signálu ukončenia vesmíru.

3.3 Komponent

Lepenka vyžaduje komponenty definované ako štruktúry. Reprezentácia pomocou štruktúr umožňuje ukladať dáta kompaktne, do takzvaných bazénov.

3.3.1 Úložisko komponentov

Samotné vytvorenie inštancie komponentu prebieha v interných dátových štruktúrach frameworku. Ten riadi všetky procesy vytvárania aj získavania dát. Využíva generickú triedu `ComponentPool<T>` ako úložisko všetkých komponentov v hre. Bazén je implementovaný ako zoznam „chunkov“, teda polí s kompaktne uloženými dátami. Trieda obaľuje funkcionalitu indexovania ale nevie o stave komponentov, ktoré drží. Validitu komponentov riadi vrstva nad komponentami, konkrétne archetyp. `ComponentPool<T>` implementuje `ref` indexér, čo v kombinácií s uložením štruktúr priamo v poli umožňuje rýchlu iteráciu.

3.3.2 Manažér komponentov

Lepenka umožňuje použitie akejkoľvek štruktúry ako komponent bez akejkoľvek registrácie vopred. Pri prvom použití typu komponenty triedy `ComponentManager` priradí unikátny identifikátor použitému typu. Archetypy

využívajú na svoju identifikáciu množiny týchto identifikátorov. Nutná vlastnosť identifikátoru je schopnosť kombinovať komponenty do množín bez straty informácie. Každý identifikátor reprezentuje jeden rád čísla binárnej sústavy, pričom hodnota 1 signalizuje prítomnosť daného komponentu a naopak 0 jeho absenciu. `ComponentManager` obsahuje mapu z typu komponentu na jeho identifikátor a opačne.

3.4 Entita

Entita slúži ako mapovanie medzi komponentami a logickým prvkom reprezentujúcim herný objekt. V implementácii je entita jednoduchá.

Výpis kódu 29 Implementácia entity:

```
public readonly struct Entity : IEquatable<Entity>
{
    internal ArchetypeIdentifier Archetype
        { get; init; }
    public int Index { get; init; }

    //IEquatable implementation
}
```

Každá entita patrí do archetypu, z princípov archetypu v sekcii 1.5.3 sú jej komponenty štruktúrované do samostatných komponentových bazénov. Entita si drží stav o archetype, v ktorom sa nachádza a rovnako obsahuje index. Index slúži na získanie pozície komponentov v komponentových bazénoch. Tieto dátové položky nám umožňujú jednoznačne identifikovať komponenty patriace danej entite.

3.5 Systémy

Implementácia systému pozostáva z vytvorenia statickej metódy v statickej triede. Návratové hodnoty ECS nevyužíva, preto je konvenciou vytvárať systémy bez návratovej hodnoty. Spúšťanie samotných systémov závisí na implementácii hernej slučky. V prípade generovanej hernej slučky sa systémy plánujú podľa ich etapy.

Herná slučka sa správa podľa definície v opise vesmíru v sekcii 3.2. Vesmír iteruje internými dátovými štruktúrami, čím získava entity. Tie vkladá ako parametre entitových systémov. V prípade zvyšných parametrov sa vesmír dopytuje manažéra zdrojov, alebo vytvára relevantné príkazy.

3.6 Príkaz

Príkazy vyhľadávajú správne archetypy a zaručujú správny postup pri manipulácii s entitami. Taktiež ponúkajú varianty metód pre rôzne počty parametrov. Tvorbu a logiku príkazov zastrešuje `ICommandManager`. Obsahuje odkaz na vesmír a dokáže sa dopytovať ku archetypom. Bližšiu implementáciu manažéra príkazov spomenieme v dokumentácii príkazov samotných.

3.6.1 Príkazy pre vesmír

Trieda `IUniverseCommand` umožňuje tvorbu entít, tvorbu alebo vymazanie zdroja a ukončenie vesmíru. Pre zachovanie konzistencie medzi cyklami hernej slučky, efekty príkazov sa uplatnia až pri spustení následujúcej fázy. Implementácie pre funkcie s viacerými parametrami sú obdobné. Následujúce implementácie pochádzajú z triedy `ICommandManager : IUniverseCommand, IEntityModifiable`.

Výpis kódu 30 Ukážka implementácie `IUniverseCommand`:

```
public void EndUniverse()
    => _universe.Running = false;

public void CreateResource<T>(T resource)
    => _universe.ResourceManager.Commit(resource);

public void RemoveResource<T>()
    => _universe.ResourceManager.Remove<T>();

public void CreateEntity<T>(T component) where T : struct
{
    var archetypeIdentifier =
    new ArchetypeIdentifier(_universe.ComponentManager)
        .AddComponent<T>();

    var archetype = _universe.ArchetypeManager
        .GetArchetype(archetypeIdentifier);

    archetype.RegisterEntity(component);
}
```

3.6.2 Príkazy pre entitu

`EntityCommand` ponúka úpravu danej entity v entitovom systéme. Narozdiel od `IUniverseCommand` sa vytvára samostatná inštancia štruktúry pre

každú entitu, na ktorej sa systém spúšťa. Medzi funkcionalitu štruktúry `EntityCommand` patrí pridanie alebo odobratie komponenty. Lepenka vyžaduje práve jedno volanie pre cyklus v systéme. Pre všetky funkcie preto existuje obdobná viac parametrická varianta. Z dôvodov zachovania jednoduchého rozhrania `EntityCommand` deleguje vykonávanie funkcie spomínanej triede `ICommandManager`. Tá pomocou štruktúry `EntityCommand` získa odkaz na entitu a môže vykonávať logiku manipulácie s komponentami.

Výpis kódu 31 Ukážka implementácie `EntityCommand`:

```
...

public void RemoveEntity() => _manager.RemoveEntity(_entity);

public void AddComponent<T>(T component) where T : struct =>
    _manager.AddComponent(_entity, component);

...
```

Toto riešenie skrýva implementačné detaily pred používateľom a umožňuje intuitívnejšie používanie príkazov na úrovni entít. `ICommandManager` má prístup ku všetkým potrebným dátam a vykonáva potrebnú logiku funkcií.

Výpis kódu 32 Implementácia manipulácie s entitami:

```
public void RemoveEntity(Entity entity)
{
    var archetype = _universe.ArchetypeManager
        .GetArchetype(entity.Archetype);
    archetype.RemoveEntity(entity);
}

public void AddComponent<T>(Entity entity, T component)
    where T : struct
{
    var oldArchetype = _universe.ArchetypeManager
        .GetArchetype(entity.Archetype);

    var archetypeIdentifier = entity.Archetype
        .AddComponent<T>();
    var newArchetype = _universe.ArchetypeManager
        .GetArchetype(archetypeIdentifier);

    oldArchetype.CopyEntity(entity, newArchetype, component);
    oldArchetype.RemoveEntity(entity);
}
```

3.6.3 Integrácia so systémami

Vesmír zodpovedá za správne dosadenie príkazových typov do systémov. V prípade `IUniverseCommand` nám stačí dosadzovať do systémov referenciu na `ICommandManager`. Pri `EntityCommand` potrebujeme entitu, na ktorej budeme robiť operácie, preto vždy do systému vkladáme vytvorenie príkazu.

Výpis kódu 33 Vytváranie príkazov vesmírom:

```
protected internal IUniverseCommand CreateUniverseCommand()
    => CommandManager;

protected internal EntityCommand CreateEntityCommand
(Entity.Entity entity)
    => new (entity, CommandManager);
```

3.7 Zdroj

Na úrovni hernej slučky existuje manažér zdrojov, ktorý interne drží slovník s dátami, indexovanými na základe typu zdroja. Tvorba zdrojov prebieha v

systémoch, preferovane v inicializačných, pretože súčasne môže byť definovaný len jeden zdroj daného typu. Počas behu programu pristupujeme ku zdrojom pomocou manažéra zdrojov. Do systémov vkladáme zdroje funkciou `Get<T>()`. Zdroje sú ukladané ako objekty a pri prístupe sú pretypované na požadovaný typ. Charakter zdrojov má sémantiku triedy, avšak aj štruktúra môže slúžiť ako zdroj. V tom prípade nastáva boxovanie pri každom prístupe.

3.8 Archetyp

Lepenka ukladá všetky informácie o entitách, vrátane entít priamo v archetypoch. Neexistuje unifikovaný zoznam všetkých entít, každá entita je priradená práve jednému archetypu, ktorý riadi jej životnosť. Pri operáciach s entitami sa archetypy dopytujú manažéra, ktorý im ponúka referencie na cieľové archetypy. Archetypy nie sú navzájom prepojené, čo umožňuje paralelizovať správu a čistenie archetypových interných dátových štruktúr. Archetyp pozostáva z unikátneho identifikátoru, zoznamu entít a komponentových bazénov.

3.8.1 Identifikácia archetypov

Každý archetyp obsahuje unikátny identifikátor reprezentujúci typy komponentov archetypu. Tento identifikátor implementujeme ako `readonly struct`, s `IEquatable` rozhraním. Interne je implementovaný ako obal na číselnom type `ulong`, pričom každá pozícia binárnej reprezentácie čísla určuje, či daný archetyp obsahuje odpovedajúci typ komponentu. Táto vlastnosť nám umožňuje odvodiť výsledný archetyp po transformácii entity.

```
// Component map
Position  -> 0001
Velocity  -> 0010
PlayerTag -> 0100
Weapon    -> 1000

// Example ArchetypeIdentifier[Position, Player, Velocity]
ArchetypeIdentifier: 0111

// Add Component[Weapon]
ArchetypeIdentifier: 1111

// Remove Component[Position]
ArchetypeIdentifier: 1110
```

Vytvorenie archetypového identifikátora si vyžaduje inštanciu triedy `ComponentManager`. Bez nej nedokáže identifikátor určiť aké hodnoty sa mapujú na dané typy komponentov. Po vytvorení identifikátora je prepojenie s komponentami vytvorené a identifikátor správne reaguje na svoje úpravy.

Výpis kódu 34 Implementácia pridania komponentu identifikátora:

```
internal ArchetypeIdentifier AddComponent<T>()
{
    var newId = Id;
    newId |= _componentManager.GetComponent(typeof(T));
    return new ArchetypeIdentifier(_componentManager)
    {
        Id = newId
    };
}
```

Identifikátor obsahuje analogické funkcie pre odstránenie jedného typu komponentu, alebo aplikácie masky komponentov, ktorá odstráni viacero typov súčasne.

3.8.2 Filtrovanie archetypov

Pri každom spustení systému potrebuje framework vyhľadať všetky relevantné archetypy. Vesmír obsahuje `ArchetypeManager` pomocou ktorého riadi aktívne archetypy v programe. Ten obsahuje mapu identifikátorov na samotné archetypy a zoznam archetypov. Na filtrovanie entít využívame zoznam ako `IEnumerable` a dopytujeme sa pomocou filtračných funkcií.

Výpis kódu 35 Ukážka dopytu validných entít:

```
// System
//With<A, B, C>
//Without<C>
//ref A a, ref B b

var archetypes = CreateQuery()
                    .With<A>()
                    .With<B>()
                    .With<C>()
                    .Without<D>();

foreach (var archetype in archetypes)
{
    var poolA = archetype.GetPool<A>();
    var poolB = archetzpe.GetPool<B>();
    foreach (var entity in archetype)
    {
        RunSystem(
            ref poolA[entity.Index],
            ref poolB[entity.Index]);
    }
}
```

Všimnime si, že na získanie entít často potrebujeme iterovať viacerými archetypmi. Každý archetyp z dopytu určite obsahuje nutné parametre vďaka filtru. Preto môžeme získať komponentové bazény daného archetypu. Archetyp spravuje všetky svoje entity, taktiež implementuje `IEnumerable<Entity>`. Pre iteráciu všetkých validných entít ostáva iterovať daný archetyp a spúšťať relevantné systémy.

3.8.3 Registrácia archetypov

Typový systém C#-u neumožňuje volanie generických funkcií s typom, ktorý nie je známy v kompilačnej dobe. Z toho dôvodu využívame reflexiu na inicializáciu doteraz nepoužitých archetypov. Toto miesto je taktiež jediným miestom, v ktorom momentálna verzia Lepenky využíva reflexiu. Funkcia sa volá pre každý unikátny archetyp práve raz, čím neobmedzuje výkonnostnú stránku frameworku. Typovú informáciu potrebujeme predať generickou funkciou, pretože interne vytvárame generickú triedu komponentových bazénov pre dané typy komponentov.

Výpis kódu 36 Tvorba nového archetypu:

```
private IArchetype RegisterArchetype
    (ArchetypeIdentifier identifier) {
    // Semantically same as:
    // CreateArchetype()
    //      .WithComponent<A>()
    //      .WithComponent<B>()
    //      .Build();

    var builder = new ArchetypeBuilder(_componentManager);
    for (ulong componentType = 1;
        componentType < (ulong)1 << 63;
        componentType <<= 1)
    {
        if (!identifier.HasComponent(componentType))
            continue;

        var type = _componentManager
                    .GetComponent(componentType);
        typeof(ArchetypeBuilder)
            .GetMethod("WithComponent",
            BindingFlags.NonPublic | BindingFlags.Instance)
            ?.MakeGenericMethod(type)
            .Invoke(builder, null);
    }

    var archetype = builder.Build();

    _archetypeMap.Add(identifier, archetype);
    _archetypes.Add(archetype);

    return archetype;
}
```

Vytvoríme staviteľa archetypov a iteráciou typov identifikátora skladáme vhodné typy do konečného archetypu. Ten registrujeme do vnútornej štruktúry archetypového manažéra. Pri opätovnom vyhľadávaní daného archetypu už manažér odkáže na vytvorený archetyp narozdiel od registrácie nového. Táto registrácia je skrytá. Pri dopytovaní na archetyp manažér transparentne vráti vytvorený archetyp bez informovania, či bol práve registrovaný.

3.8.4 Konzistentnosť entít

Počas hernej slučky sa spúšťajú systémy, ktoré z hľadiska úpravy archetypov delíme do nasledujúcich tried.

Statické systémy Tieto systémy nevytvárajú ani neodstraňujú žiadne entity a preto nenastáva zmena štruktúry entít v archetypoch.

Generatívne systémy Počas behu tohto systému možno počítať so zmenou v archetypoch a to pridaním, alebo odobraním entity.

Transformačné systémy Systémy transformujú štruktúru entít a to pridaním alebo odobratím komponentu. Taktiež je možné transformovať entitu kombináciou aditívnych a subtraktívnych operácií.

Predstavené triedy systémov sú v praxi kombinované a nie je raritou pracovať so systémom ktorý patrí do všetkých spomenutých tried. Problém registrácie nových doteraz neregistrovaných archetypov v tomto príklade neberieme do úvahy. Zaujímajú nás len situácie, kedy potrebujeme entitu vytvoriť, odstrániť, alebo preradiť do iného archetypu. Všimnime si, že vytvorenie a odstránenie entity je operácia vykonaná na práve jednom archetype. Pre transformáciu avšak potrebujeme nie len odstrániť entitu zo starého archetypu, ale informovať výsledný archetyp o tvorbe novej entity. Aby sme zachovali konzistentnosť aktívnych entít, všetky zmeny v štruktúre entít sú delegované na po-aktualizačný proces. To znamená, že Lepenka garantuje *uskutočnenie zmien až po zbehnutí všetkých systémov danej fázy hernej slučky.* Nemôže sa teda stať, že bude v rovnakej fáze entita vytvorená a zároveň ku nej bude pristúpené. Na zaistenie tejto vlastnosti obsahujú archetypy vyrovnávaciu pamäť, tá v sebe ukladá všetky entity, ktoré ešte treba odstrániť alebo pridať. Po zbehnutí všetkých systémov si každý archetyp súbežne aktualizuje svoj stav entít a pripraví sa na ďalšiu fázu hernej slučky.

3.8.5 Operácie archetypu

Archetyp umožňuje vytvárať nové entity, pristupovať ku komponentom entít a odstrániť entity. Odstránením sa entita vymaže zo zoznamu živých entít, no jej komponenty ostávajú v komponentových bazénoch nezmenené. Index v bazéne sa uvoľní pre ďalšiu entitu, až jej vytvorením sa prepíšu neplatné dáta. Odstránenie entity sa teda stáva lacnou operáciou.

Výpis kódu 37 Registrácia entity:

```
public void RegisterEntity<T>(T component) where T : struct
{
    var entity = ((IArchetype) this).CreateEntity();

    ref var data = ref GetData<T>(entity);
    data = component;

    ((IArchetype) this).CommitNewEntity(entity);
}
```

Po operáciach registrácie alebo odstránenia entity je archetyp v nestabilnom stave. Na dokončenie zmeny stavu entít je potrebné zavolať funkciu `ExecuteChanges`, ktorá dokončí manipuláciu s entitami a pripraví ich na ďalšie úkony. Táto funkcia je riadená triedou `ArchetypeManager` a je generovaná medzi etapami hernej slučky. `ExecuteChanges` paralelne spustí v každom archetype spracovávanie entít v dočasnej vyrovnávacej pamäti.

3.9 Generátor

Lepenka využíva generátor vesmíru, ten zozbiera informácie o projekte a vystavia hernú slučku na mieru. Herná slučka je implementáciou triedy vesmír, presnejšie funkcia `Run`. Tvorca modulov pre framework Lepenka môže využiť generátor modulov. Ten spracuje anotované časti knižnice a vytvorí metadátový súbor nutný pre správne fungovanie generátoru vesmíru.

3.9.1 Generácia vesmíru

Úlohou generátora je vytvoriť hernú slučku na mieru definovaným systémom a komponentom. Informácie získavame pomocou triedy `UniverseWalker` dediacej z triedy `CSharpSyntaxWalker`. `UniverseWalker` prechádza každý vrchol syntaktického stromu kompilácie a extrahuje dáta vyžadované na správny chod programu.

Výpis kódu 38 Zber informácií z projektových súborov

```
private static (EnumDeclarationSyntax,
                EnumDeclarationSyntax?,
                SystemCollector)
CollectUniverse(GeneratorExecutionContext context)
{
    var collector = new SystemCollector();
    var walker = new UniverseWalker(collector);

    foreach (var node in context
                        .Compilation
                        .SyntaxTrees
                        .Select(st => st.GetRoot()))
    {
        walker.Visit(node);
    }

    return
        (walker.StageEnum
        ?? throw new ArgumentNullException
                        ("No Stages found!"),
        walker.ThreadEnum,
        collector);
}
```

Jedna z dátových štruktúr použitá na zber systémov je `SystemCollector`. Počas čítania zdrojového kódu nevieme ihneď určiť, akého typu bude systém. Keďže anotácie čítame postupne, budujeme takzvaný `TemporarySystem`, ktorý na konci čítania systému vyhodnotí akého typu systém je. `TemporarySystem` zisťuje meno systému a triedu, v ktorej sa nachádza. Ďalej zisťuje či je nastavená priorita, filter, špecifické vlákno alebo etapa. Nakoniec vhodne zozbiera parametre a čaká na predanie inštancii `SystemCollector`-a, tá vyhodnotí správny typ systému a uloží ho v zozname.

`UniverseWalker` pridáva logiku pre isté typy vrcholov syntaktického stromu. Pri prechádzaní definície enumerácie `UniverseWalker` zisťuje, či enumerácia nedefinuje etapy alebo vlákna programu. V prípade zhody si enumeráciu uloží a kontroluje duplicitné definície enumerácií. Okrem toho analyzuje systémy, pri pristúpení na deklaráciu metódy vytvorí špecializovanejšiu inštanciu triedy `MethodWalker` a spustení prechodu extrahovaný systém `SystemCollector` spracuje.

Každý `MethodWalker` obsahuje práve jeden `TemporarySystem`. Vypĺňa ho navštívením anotácií a parametrov funkcií. Pre príklad si uvedieme zisťovanie názvu a priority systému.

Výpis kódu 39 Extrakcia dát systému

```
private static int?
GetStartupSystemPriority(AttributeSyntax node)
{
    var priorityString = node.ArgumentList?
        .Arguments
        .OfType<AttributeArgumentSyntax>()
        .Select(argument => argument.Expression)
        .OfType<LiteralExpressionSyntax>()
        .Select(literal => literal.Token.Text)
        .First();

    return priorityString is null
        ? null : int.Parse(priorityString);
}

private static string GetAttributeName(AttributeSyntax node)
{
    var name = node.Name;
    return name switch
    {
        IdentifierNameSyntax namedIdentifier
            => namedIdentifier.Identifier.Text,
        GenericNameSyntax genericIdentifier
            => genericIdentifier.Identifier.Text,
        _ => throw new ArgumentException("Invalid node Name")
    };
}
```

Podobným spôsobom získame všetky dáta, ktoré systém obsahuje, pričom nepovinné a nenájdené reprezentujeme hodnotou `null`. Po dokončení zbierania vesmíru z projektových systémov a komponentov generátor spracuje priradené moduly. Na získanie informácii z modulov potrebujeme deserializovať XML súbory pribalené ku projektu.

Výpis kódu 40 Kolekcia modulových systémov

```
private static GeneratorUniverse
CollectModules(GeneratorExecutionContext context)
{
    var modulePaths = context.AdditionalFiles
        .Where(module => module.Path
                            .EndsWith(".xml"))
        .Where(module => !module.Path
                            .EndsWith(UniverseFileName))
        .Select(module => module.Path);

    var universePath = context.AdditionalFiles
        .First(module => module.Path
                            .EndsWith(UniverseFileName))
        .Path;

    var universe =
        Deserialize<GeneratorUniverse>(universePath);
    foreach (var modulePath in modulePaths)
    {
        var module = Deserialize<Module>(modulePath);
        universe.Modules.Add(module);
    }

    return universe;
}
```

Trieda `GeneratorUniverse` slúži len ako schéma pre deserializáciu. Vytvára stromovú štruktúru zloženú z modulov, tie obsahujú systémy. Typy týchto systémov vieme jednoznačne určiť už pri deserializácii. Celý proces generácie vesmíru spočíva v týchto krokoch:

Výpis kódu 41 Generácia vesmíru:

```
public void Execute(GeneratorExecutionContext context)
{
    var (stageEnum, threadEnum, systemCollector)
        = CollectUniverse(context);
    var moduleUniverse = CollectModules(context);

    var stages = stageEnum.Members
        .Select(m => m.Identifier.Text);
    var threads = threadEnum?.Members
        .Select(m => m.Identifier.Text);
    var namespaces = moduleUniverse.Modules
        .Select(module => module.Namespace);
    var usingDirectives = moduleUniverse.Modules
        .SelectMany(module => module.Imports).Distinct();
    var universeBuilder
        = new UniverseBuilder(stages,
                              threads,
                              namespaces,
                              usingDirectives,
                              moduleUniverse.Imports);
    var projectNamespace = context.Compilation.AssemblyName
        ?? throw new ArgumentException
                  ("No namespace found.");
    var universeName = moduleUniverse.Name;

    // Fill universeBuilder with systems
    universeBuilder.ImportCollectedUniverse(systemCollector);
    universeBuilder.ImportModules(moduleUniverse);

    universeBuilder.OrderStartupSystems();

    using var stream = new MemoryStream();
    using var writer =
        new IndentedTextWriter(new StreamWriter(stream));

    universeBuilder.Generate(writer,
                              projectNamespace,
                              universeName);

    stream.Position = 0;
    context.AddSource($"{universeName}.g.cs",
        SourceText.From(stream,
                        Encoding.UTF8,
                        canBeEmbedded: true));
}
```

V prvej časti kódu spúšťame kolektory na získanie všetkých systémov. Neskôr si pripravujeme dodatočné informácie, teda názov, etapy, vlákna a podobne. Trieda `UniverseBuilder` importuje systémy a konvertuje ich do uniformnej podoby. Navyše pripravuje dáta v závislosti od ostatných doplnených informácií, napríklad faktu, či bude program využívať samostatné špecifické vlákna. Po zoradení systémov generátor pripravuje finálny kód.

Výpis kódu 42 Generácia kódu vesmíru:

```
internal void Generate(IndentedTextWriter writer,
                        string projectNamespace,
                        string universeName)
{
    GenerateHeader(writer, projectNamespace);
    GenerateUniverseBegin(writer, universeName);

    if (ThreadSystems.Count > 0)
    {
        GenerateRun(writer);
        GenerateThreadClass(writer, universeName);
    }
    else
    {
        GenerateRunSingleThread(writer);
    }
    GenerateUniverseEnd(writer);
}
```

Generácia kódu prebieha vo viacerých krokoch. Ako prvé sa vygeneruje hlavička súboru obsahujúca všetky importy knižníc a definíciu namespace. Generátor pokračuje vytvorením vlastnej triedy dediacej z vesmíru, pričom nový vesmír implementuje hlavnú slučku herného sveta. Funkcia `Run` inicializuje vhodné dátové štruktúry, následne spustí štartovacie systémy v správnom poradí a nakoniec opakovane spúšťa herné behové systémy. Každý systém `UniverseBuildra` implementuje reprezentáciu spustenia v zdrojovom kóde. Generátor využíva preddefinovanú konvenciu názvov, čo umožňuje jednoduchú skladbu hernej slučky. Komponentové bazény nesú názov vo formáte `pool[Meno typu]` a premenné majú rovnako jednotný názov, teda `archetype`, `entity`, `systemTasks` a podobne. Po každom cykle hernej slučky generátor vloží volanie funkcie `ExecuteCommands`.

Výpis kódu 43 Príklad generovanej hernej slučky:

```
public override async Task Run() {
    var systemTasks = new List<Task>();
    // Run Startup Systems:
    systemTasks.Add(Task.Run(
        () => StartupSystem.Startup(
            CreateUniverseCommand())));
    await ExecuteCommands(systemTasks);
    // Run Entity Startup Systems:
    ...
    while (Running) {
        // Stage: BeforeRound
        // System: Update
        systemTasks.Add(Task.Run(
            () => NewPlayerSystem.Update(
                ResourceManager.Get<GameRules>(),
                ResourceManager.Get<GameState>(),
                CreateUniverseCommand())));
        // System: Update
        systemTasks.Add(Task.Run(
            () => NewRoundSystem.Update(
                ResourceManager.Get<GameRules>(),
                ResourceManager.Get<GameState>())));
        await ExecuteCommands(systemTasks);
        // Stage: Update
        // System: Update
        systemTasks.Add(Task.Run(
            () => PrintMessageSystem.Update()));
        // System: Update
        foreach (var archetype in ArchetypeManager
                                    .CreateQuery()
                                    .With<Player>()
                                    .With<Score>()
                                    .ToList()) {
            var poolPlayer = archetype.GetPool<Player>();
            var poolScore = archetype.GetPool<Score>();
            foreach (var entity in archetype) {
                systemTasks.Add(Task.Run(
                    () => ScoreSystem.Update(
                        ref poolPlayer[entity.Index],
                        ref poolScore[entity.Index])));
            }
        }
        await ExecuteCommands(systemTasks);
        //Next phases...
    }
}
```

3.9.2 Generácia herných slučiek špecifických vlákien

Pre hry so špecifickými vláknami a systémami delegovanými na ne vytvára samostatné herné slučky. Priebeh hlavnej slučky sa obohatí o tvorbu vlákien s generovanými funkciami. Tieto vlákna držia odkaz na vesmír a teda vedia pracovať s entitami zdieľane. Systémy naprieč vláknami sa čakajú medzi etapami. Konzistentnosť behu systémov medzi vláknami je zabezpečená pomocou synchronizačnej bariéry.

Výpis kódu 44 Útržok vytvorenia vlákna so samostatnými systémami:

```
//After initialization
using var barrier = new Barrier(2);

var threadRender =
new Thread(new UniverseThread(this, barrier).RunRender);
threadRender.Start();

//Waiting for initialization systems on second thread.
barrier.SignalAndWait();

//Game loop
...
```

Spomínané vlákno obsahuje a spúšťa systémy definované v projekte, alebo modulových metadátach.

3.9.3 Generácia modulov

Generácia modulov prebieha na rovnakých princípoch ako generácia vesmíru.

Výpis kódu 45 Generácia modulov:

```
public void Execute(GeneratorExecutionContext context)
{
    var mainSyntaxTree = context.Compilation.SyntaxTrees
                      .First(x => x.HasCompilationUnitRoot);
    var directory = Path.GetDirectoryName(
                        mainSyntaxTree.FilePath);

    var walker = new ModuleWalker();
    foreach (var node in context
                        .Compilation
                        .SyntaxTrees
                        .Select(st => st.GetRoot()))
    {
        walker.Visit(node);
    }

    foreach (var module in walker)
    {
        Serialize(module,
            Path.Combine(directory, $"{module.Name}.xml"));
    }
}
```

3.10 Porovnanie ECS frameworkov

Lepenka ponúka unikátny pohľad na ECS. Jazyk C# je populárnou voľbou pre tvorbu hier a s rastúcou popularitou ECS prístupu sa ekosystém vývoja hier v posledných rokoch rozšíril o silné nástroje. Keďže ECS prístup ponúka len usmernenia ako modelovať dáta, funkcionalita a vlastnosti frameworkov sa líšia. Ako ukážku porovnania vlastností menujeme Unity DOTS, dátovo orientovaný prístup tvorby hier za pomoci ECS, a 5 najpopulárnejších C# ECS frameworkov na platforme GitHub.

Môžeme si povšimnúť, že z populárnych riešení je Lepenka jediná implementácia ECS spĺňajúca všetky spomenuté požiadavky súčasne. Samozrejme, ostatné frameworky využívajú iné spôsoby organizácie dát, čím ponúkajú odlišné výhody, ako napríklad reaktivitu.

	Archetypový systém	Generácia zdrojového kódu	Paralelizovateľné systémy	Dátovo orientovaná architektúra	Platformová nezávislosť
Unity DOTS[a]	✓	✓	✓	✓	–
Entitas[b]	–	✓	–	–	✓
Svelto.ECS[c]	–	–	✓	✓	✓
Actors[d]	–	–	✓	–	–
DefaultEcs[e]	–	✓	✓	✓	✓
EcsRx[f]	–	–	✓	–	✓
Lepenka	✓	✓	✓	✓	✓

Tabuľka 3.1 Porovnanie implementácií populárnych ECS frameworkov.

[a] `https://unity.com/dots`
[b] `https://github.com/sschmid/Entitas-CSharp`
[c] `https://github.com/sebas77/Svelto.ECS`
[d] `https://github.com/PixeyeHQ/actors.unity`
[e] `https://github.com/Doraku/DefaultEcs`
[f] `https://github.com/EcsRx/ecsrx`

Kapitola 4

Ukážky frameworku

V tejto kapitole si priblížime reálne aplikácie frameworku. V prvom príklade sa zameriame na proces tvorby hry, v druhom vysvetlíme priebeh implementácie modulov. Na záver si predstavíme využitie sieťovania v implementovanom programe.

4.1 Konzolový príklad

Ako základná ukážka funkcionalít frameworku Lepenka slúži konzolový príklad. Vytvorená hra je jednoduchá konzolová aplikácia. Ukážka slúži ako predstavenie procesu tvorby hry s dôrazom na jednoduchosť implementácie hernej logiky. Tvorba hry v princípoch ECS žiada užívateľa o definíciu herných procesov. Definujme si nasledujúce pravidlá:

- Hra pozostáva z preddefinovaného počtu kôl.
- Hra obsahuje maximálny počet hráčov.
- Každé kolo hráči postupne hádžu mincou.
- Ak padne hlava, získavajú bod.
- Po konci kola sa môže do hry pripojiť hráč ak ešte nie je naplnený počet hráčov.
- Získaním preddefinovaného počtu bodov hráč vyhráva.
- Ak nikto nezíska stanovený počet bodov do limitu kôl hry, nevyhráva nikto.

Transformujme spomenuté pravidlá do konceptov Lepenky. Vytvorme si komponenty opisujúce herné prvky. Majme teda komponent hráča, komponent skóre, herné pravidlá a súčasný herný stav.

Výpis kódu 46 Dáta konzolového príkladu:

```
public class GameState
{
    public int CurrentRound { get; set; }
    public int TotalPlayers { get; set; }
    public string? WinningPlayer { get; set; }
}

public class GameRules
{
    public int WinningScore { get; init; }
    public int MaxRounds { get; init; }
    public int MaxPlayers { get; init; }
}

public struct Player
{
    public string Name { get; init; }
}

public struct Score
{
    public int Value { get; set; }
}

[Stage]
internal enum Stage
{
    BeforeRound,
    Update,
    AfterRound,
    GameOverCheck
}
```

Niektoré pravidlá sme zachytili definíciou štruktúry herných dát. Ostáva nám zachytiť zvyšné pravidlá definíciou systémov.

Výpis kódu 47 Inicializácia hry:

```
[StartupSystem]
public static void Startup(IUniverseCommand command)
{
    command.CreateResource(new GameRules
    {
        MaxRounds = 10,
        MaxPlayers = 4,
        WinningScore = 4
    });
    command.CreateResource(new GameState
    {
        CurrentRound = 0,
        TotalPlayers = 2,
        WinningPlayer = null
    });
    command.CreateEntity(
        new Player { Name = "Alice" },
        new Score { Value = 0 });
    command.CreateEntity(
        new Player { Name = "Bob" },
        new Score { Value = 0 }
    );
}
```

Definovali sme obmedzenia na počet kôl a hráčov, taktiež sme vytvorili počiatočných hráčov hry. Ukážeme si, ako prebieha hod mincou.

Výpis kódu 48 Priebeh kola hry:

```
[System<Stage>(Stage.Update)]
[With<Player, Score>]
public static void
UpdateScore(ref Player player, ref Score score)
{
    var scoredAPoint = new Random().NextDouble() > 0.5;
    if (scoredAPoint)
    {
        score.Value++;
        Console.WriteLine($"{player.Name} scored \\
        a point! Their score is: {score.Value}");
    }
    else
    {
        Console.WriteLine($"{player.Name} did not \\
        score a point! Their score is: {score.Value}");
    }
}
```

Vytvorením entitového systému zaručíme spustenie tejto funkcie na všetky entity, ktoré obsahujú komponenty typu hráč a skóre. Preiterovaním entít aktualizujeme ich stav skóre. Po kole následuje kontrola výhercu a potenciálne pridanie hráča.

Výpis kódu 49 Kontrola stavu hry po skončení kola:

```
[System<Stage>(Stage.AfterRound)]
[With<Player, Score>]
public static void
CheckWinner(ref Player player,
            ref Score score,
            GameRules rules,
            GameState state)
{
    if (score.Value == rules.WinningScore)
        state.WinningPlayer = player.Name;
}

[System<Stage>(Stage.AfterRound)]
public static void
AddPlayer(GameRules rules,
          GameState state,
          IUniverseCommand command)
{
    var addNewPlayer = new Random().NextDouble() > 0.5;
    if (!addNewPlayer ||
        state.TotalPlayers >= rules.MaxPlayers)
        return;

    state.TotalPlayers++;
    command.CreateEntity(
        new Player { Name = $"Player {state.TotalPlayers}" },
        new Score { Value = 0 });

    Console.WriteLine($"Player {state.TotalPlayers} joined \\
                                                  the game!");
}
```

Môžeme si povšimnúť, že systémy sú nezávislé a preto ich spúšťame paralelne. Pravidlá hry sú jednoducho transformovateľné do herného frameworku. Ostáva nám implementovať pravidlo ukončenia hry po dosiahnutí počtu kôl alebo ak hráč vyhrá. Nakoniec dodefinujeme logiku pri začiatku kola. Logika je kompletne zachytená a hru môžeme spustiť.

Výpis kódu 50 Ukončenie hry a začiatok kola:

```
[System<Stage>(Stage.GameOverCheck)]
public static void
GameOver(GameRules rules,
         GameState state,
         IUniverseCommand command)
{
    if (state.WinningPlayer is not null)
    {
        Console.WriteLine($"{state.WinningPlayer} won \\
                                     the game!");
        command.EndUniverse();
    }
    else if (state.CurrentRound == rules.MaxRounds)
    {
        Console.WriteLine("Ran out of rounds. Nobody wins!");
        command.EndUniverse();
    }
}

[System<Stage>(Stage.BeforeRound)]
public static void
StartRound(GameRules rules, GameState state)
{
    state.CurrentRound++;
    Console.WriteLine($"Begin round {state.CurrentRound} \\
                                of {rules.MaxRounds}");
}
```

Kompletne sme opísali hernú logiku. Hra obsahuje kolá, v ktorom sa hráči striedajú. Hráči sa počas hry pridávajú do hry, víťazom sa stáva najúspešnejší hráč, alebo nikto, ak sa nenaplní dostatočný počet bodov v definovanom čase. Zmena pravidiel alebo počiatočnej konfigurácie je jednoduchá, systémy sú oddelené a dátové závislosti sú známe. V konfigurácii je definované meno generovaného vesmíru. Na spustenie hry stačí jeden top-level statement a generátor v tandeme s frameworkom vygeneruje funkčnú hru.

Výpis kódu 51 Spustenie hry:

```
//Main function
using ConsoleExample;

await new ConsoleUniverse()
    .Run();
```

Na záver si ukážeme jeden z možných výpisov hry. Z dôvodu náhodného hodu mincou nevieme predpokladať priebeh hry.

```
Begin round 1 of 10
Bob did not score a point! Their score is: 0
Alice scored a point! Their score is: 1
Begin round 2 of 10
Bob scored a point! Their score is: 1
Alice scored a point! Their score is: 2
Begin round 3 of 10
Alice scored a point! Their score is: 3
Bob scored a point! Their score is: 2
Begin round 4 of 10
Bob scored a point! Their score is: 3
Alice did not score a point! Their score is: 3
Begin round 5 of 10
Bob did not score a point! Their score is: 3
Alice did not score a point! Their score is: 3
Begin round 6 of 10
Alice did not score a point! Their score is: 3
Bob did not score a point! Their score is: 3
Begin round 7 of 10
Bob did not score a point! Their score is: 3
Alice did not score a point! Their score is: 3
Begin round 8 of 10
Alice scored a point! Their score is: 4
Bob scored a point! Their score is: 4
Player 3 joined the game!
Bob won the game!
```

4.2 Raylib

Následujúca ukážka využíva moduly na integráciu grafických knižníc do hry. Modul `Raylib` ponúka ECS bindingy na grafickú knižnicu `Raylib_cs`. Ukážka zároveň využíva funkcionalitu modulu `Timer` na získanie trvania medzi cyklami.

Ako prvé si priblížime logiku hry. Hra ktorú sa pokúšame replikovať je známa aj ako Breakout, alebo Arkanoid. Loptičku odrážame od hráčom ovládanej plochy a rozbíjame políčka v druhej časti herného sveta. Pravidlá vieme zhrnúť nasledujúcimi bodmi:

- Ak sa loptička dotkne políčka, tak sa políčko rozbije a loptička odrazí.
- Loptička sa taktiež odrazí po kolízii s hráčom.
- Steny odrážajú loptičku.
- Hráč sa pohybuje horizontálne po obrazovke.

Reflektujme tieto pravidlá nasledujúcimi systémami[1]:

Výpis kódu 52 Systém pohybu hráča:

```
[System<Stage>(Stage.Post)]
[With<Box, Direction, Player>]
public static void MovePlayer(
                ref Box box,
                ref Direction direction,
                TimeModule.StopwatchTimer stopwatch)
{
    box.SimulationX += direction.Horizontal * 2
                        * stopwatch.GetElapsedTime();
}

// MoveBall updates box.SimulationX
// as well as box.SimulationY.
```

Systém `MoveBall` je analogický. Tieto systémy počítajú pohyb loptičky a hráča. Hráčov smer upravuje dodatočný systém kontrolujúci stlačenia kláves.

[1] Zobrazenie systémov je v zjednodušenej podobe.

Výpis kódu 53 Kolízne systémy:

```
[System<Stage>(Stage.Post)]
[With<Box>]
[Without<Player>]
public static void
CheckBox(Ball ball,
        ref Box box,
        EntityCommand command,
        Score score)
{
    if (!CheckCollision(ball, box)) return;

    score.ScoreValue++;
    command.RemoveEntity();

    // Improper collision resolve
    ball.Horizontal *= -1;
    ball.Vertical *= -1;
}

[System<Stage>(Stage.Post)]
[With<Box, Player>]
public static void CheckBox(Ball ball, ref Box box)
{
    if (!CheckCollision(ball, box)) return;

    ball.Horizontal *= -1;
    ball.Vertical *= -1;
}

[System<Stage>(Stage.Post)]
public static void CheckBounds(Ball ball, Window window)
{
    if (ball.SimulationX > window.WindowWidth - ball.Width
        || ball.SimulationX < 0) ball.Horizontal *= -1;
    if (ball.SimulationY > window.WindowHeight - ball.Height
        || ball.SimulationY < 0) ball.Vertical *= -1;
}
```

Všimnime si systémy `CheckBox`. Prvý rozhoduje o entitách, ktoré majú komponentu `Box` a zároveň nie sú hráčmi. Rozumieme tým políčka v hernom svete. Funkcia kontroluje kolíziu s loptičkou a v prípade kolízie políčko odstráni, loptičku odrazí a pripočíta skóre. V druhom systéme sledujeme kolízie loptičky s hráčom. V tom prípade loptičku odrážame. Nakoniec kontrolujeme odraz loptičky od steny.

Inicializačná funkcia vytvorí vhodné entity a zdroje.

Výpis kódu 54 Inicializačné systémy:

```
...
commands.CreateResource(new Window
{
    WindowHeight = 400,
    WindowWidth = 200,
    WindowTitle = "Breakout"
});
for (var i = 0; i < 6; i++)
{
    for (var j = 0; j < 6; j++)
    {
        commands.CreateEntity(
            new Box
            {
                SimulationX = 15 + 30 * j,
                SimulationY = 40 + 20 * i,
                Height = 10,
                Width = 20,
                Color = Color.RED
            }
        );
    }
}
...
```

Okrem týchto entít a zdrojov samozrejme inicializujeme aj loptičku, hráča a podobne. Logiku hry sme kompletne zachytili, jediné, čo nám ostáva je vytvoriť jej grafickú reprezentáciu. Na to využijeme modul `Raylib`. Priblížme si časť implementácie modulu:

Výpis kódu 55 Modul grafickej knižnice:

```
[ExportEntitySystem]
public static void DrawBox<T>(ref T component)
    where T : struct, IBox
{
    Raylib_cs.Raylib
            .DrawRectangle(component.X,
                           component.Y,
                           component.Width,
                           component.Height,
                           component.Color);
}

[ExportSystem]
public static void CloseWindow
    (IUniverseCommand commands)
{
    if (Raylib_cs.Raylib.WindowShouldClose())
    {
        Raylib_cs.Raylib.CloseWindow();
        commands.EndUniverse();
    }
}

[ExportInterface]
public interface IBox
{
    public int X { get; }
    public int Y { get; }
    public int Width { get; }
    public int Height { get; }
    public Color Color { get; }
}
```

Dôležitým konceptom pri moduloch je vytvorenie kontraktu pre komponenty. Ten splníme implementáciou štruktúry s rozhraním `IBox`.

Výpis kódu 56 Implementácia rozhrania modulu:

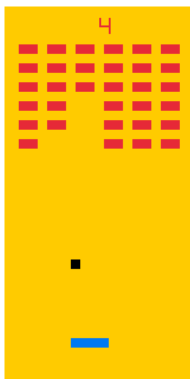
```
public struct Box : Raylib.IBox
{
    public float SimulationX { get; set; }
    public float SimulationY { get; set; }

    public int X => (int)Math.Round(SimulationX);
    public int Y => (int)Math.Round(SimulationY);
    public int Width { get; set; }
    public int Height { get; set; }
    public Color Color { get; set; }
}
```

Dátový model hry je napojený na moduly, ostáva oznámiť frameworku, kedy a ako ich spustiť. Na to nám poslúži metadátový súbor.

Výpis kódu 57 Integrované metadáta:

```
...
<EntitySystem Name="DrawBox" Thread="Render" Stage="Draw">
    <Parameters>
        <EntityParameter Type="Box">
            <Implementing>IBox</Implementing>
        </EntityParameter>
    </Parameters>
    <Filters>
        <With Type="Box" />
    </Filters>
</EntitySystem>
<GlobalSystem Name="DrawText" Thread="Render" Stage="Draw">
    <Parameters>
        <ResourceParameter Type="Score">
            <Implementing>IText</Implementing>
        </ResourceParameter>
    </Parameters>
</GlobalSystem>
...
```

Analogicky opíšeme všetky systémy ktoré chceme spustiť. Po konfigurácii projektu môžeme spustiť hru.



Obr. 4.1 Ukážka hry vytvorenej vo frameworku Lepenka.

4.3 Príklad so sieťovaním

Výhodou ECS prístupu je jednoduchá serializácia dát. Keďže je logika hry kompletne oddelená od jej dát, spôsob získavania komponentov neovplyvňuje systémy. Preto rozdiel, či je entita vytvorená v rámci herného systému, alebo sme ju získali z internetu nie je. Narozdiel od predchádzajúcich príkladov nebudeme opisovať vytvorenie systémov 4.1, alebo integráciu modulov. 4.2 Budeme analyzovať nad možnosťami posielania komponentov, alebo posielania všeobecných dát, taktiež ukážeme možnosť posielania periodických dát pomocou modulu.

Pracovať budeme s triedou `TcpClient`, vhodnou abstrakciou nad posielaním dát pre tento príklad. Modul, ktorý posiela dáta entít by mohol byť implementovaný následovne:

Výpis kódu 58 Ukážka implementácie sieťového modulu:

```
[ExportStartupSystem]
public static void Connect<T>(T resource)
where T : IConnectible, INetworkStream
{
    resource.Client.Connect(resource.IpAddress,
                            resource.Port);
}

[ExportEntitySystem]
public static void
Send<TData>(ref TData entity, INetworkStream client)
    where TData : IEntitySerializable
{
    if (client.Client.Connected)
        client.Client
            .GetStream()
            .Write(entity.SerializedData);
}

[ExportInterface]
public interface INetworkStream
{
    public TcpClient Client { get; }
}

[ExportInterface]
public interface IConnectible
{
    public IPAddress IpAddress { get; }
    public int Port { get; }
}

[ExportInterface]
public interface IEntitySerializable
{
    public ReadOnlySpan<byte> SerializedData { get; }
}
```

V projekte zadefinujeme komponenty, pričom `IEntitySerializable` slúži ako všeobecné označenie pre komponent obsahujúci serializačnú funkciu. Posielanie dát po sieti spočíva vo volaní systému `Send` na všetky zamýšľané komponenty. Výhodou enitových systémov je možnosť filtrovať ich. Ak nechceme serializovať a posielať dáta všetkých entít, vieme označiť štítkovým komponentom len tie pre nás dôležité.

Modul nám umožňuje rozšíriť funkcionalitu frameworku, napríklad pridaním možnosti posielania periodických dát. Implementovaný štartovací systém spustí nový `Task`, ktorý periodicky posiela dáta. Ako príklad slúži nasledujúca funkcia.

Výpis kódu 59 Ukážka implementácie periodického posielania dát pomocou modulu:

```
[ExportStartupSystem]
public static async Task
SendPeriodically(IResourceSerializable resource,
                 INetworkStream client)
{
    while (!resource.Token.IsCancellationRequested)
        await Task.Run(async () =>
        {
            SendAsync(resource.SerializedData,
                      client.Client);
            await Task.Delay(resource.Milliseconds,
                             resource.Token);
        }, resource.Token);
}
```

Ako aj v `Raylib` ukážke sme dosiahli modulárne riešenie, bez nutnosti dodatočnej implementácie logiky používateľom frameworku. Ten sa môže naďalej sústrediť na implementáciu hernej logiky a framework spracováva požiadavky modulov automaticky.

Záver

Lepenka je modulárny framework umožňujúci tvorbu hier pre platformu .NET. Využíva architektúru ECS, moderného spôsobu organizácie dát. Framework splnil všetky svoje stanovené ciele:

- Lepenka vytvára abstrakciu nad tvorbou hry a umožňuje znovupoužiteľnosť kódu naprieč hrami za pomoci modulov.
- Generácia kódu je využívaným konceptom pri frameworku, generovaný kód oslobodzuje užívateľa od implementácie interných štruktúr hernej slučky.
- Rýchla iterácia entít a paralelizácia herných systémov je umožnená organizáciou dát pomocou archetypov.

Z porovnaní ECS pre platformu .NET v tabuľke 3.1 pozorujeme prínos pre ekosystém tvorby hier. Lepenka ako jediná ponúka platfromovo nezavislý, dátovo orientovaný paralelizovateľný ECS framework implementovaný archetypovým systémom. Generáciou kódu zaručuje minimalizáciu písania opakovaného kódu. Tým, že framework abstrahoval systémy do samostatných jednotiek, vieme vylepšovať a optimalizovať každú oblasť hier súčasne. Tvorba hernej slučky pomocou generátora umožňuje množstvo vylepšení frameworku aj bez zásahu do jeho rozhrania. Návrh Lepenky zachováva stabilné rozhranie a zároveň ponúka širokú škálu možných interných úprav.

Využitie novovytvoreného frameworku si nájdu najmä noví záujemcovia o tvorbu hier pomocou ECS. Vstup do systému vytvoreného frameworku je priamy, bez zložitej konfigurácie. Napísať jednoduchý funkčný príklad vieme vo frameworku na zopár riadkov. Z tohto dôvodu Lepenka vyniká aj v tvorbe prototypov.

Aj napriek spomínaným výhodám nemôžeme Lepenku doporučiť do produkčného prostredia. Framework zaostáva oproti ostatným vo funkcionalite. Lepenka je navrhnutá ako exkluzívne archetypový typ ECS, teda neobsahuje reaktívne prvky. Rozhranie je teda jednoduché na používanie, ale v praxi zaostáva o možnosti okamžite reagovať na aktualizáciu, pridanie alebo odobratie

prvkov. Lepenka sa spolieha na užívateľa v implementácii daných funkcionalít. Funkcionalita frameworku avšak rastie s počtom implementovaných modulov.

Zvyšné frameworky vybudovali pevné základy tvorby hier pre .NET. Ich ECS sú optimalizované a ich rozhranie je obľúbené komunitou. Lepenka experimentuje so spôsobom tvorby hier. Aby sa priblížila ku kvalite spomínaných frameworkov si vyžaduje množstvo vylepšení a úprav, ktoré si priblížime.

4.4 Budúci vývoj

Tvorba stabilného a výkonného ECS je komplexný problém, s ktorým sa aj spoločnosti ako Unity Technologies zaoberajú roky. Ich dátovo orientovaný technologický stack využívajúci ECS je v aktívnom vývoji už vyše 5 rokov. [10] Do dnešného dňa je v experimentálnom štádiu a nie je pripravený do produkcie. Prirodzene aj v Lepenke nachádzame priestor na zlepšenie jej funkcionalít.

4.4.1 Iterátor entít

Momentálne neexistuje prívetivá možnosť pre iteráciu entít v systéme. Systém sa spúšťa na úrovni entity, alebo globálne. Keďže užívateľ nemanipuluje s odkazmi na podkladové dátové štruktúry, nemá jednoznačnú možnosť pristúpiť ku iným entitám počas behu systému bez explicitnej manipulácie s hernou slučkou alebo uloženia dát v zdrojoch.

Riešením uchovávajúcim filozofiu Lepenky je implementovať iterátor, ktorý bude vkladaný do systémov rovnako ako komponenty a zdroje. Iterátor definuje navštívené entity, rovnako ako požiadavky systému, teda pomocou filtra a špecifikácie parametrov. Keďže iterátor bude vkladaný do systémov v hernej slučke izolovanej od užívateľa, jediným spôsobom ako definovať entity je pomocou typovej informácie parametra.

Výpis kódu 60 Príklad návrhu entitového iterátoru:

```
public class Entities<TFilter, TParameters>
    : IEnumerable<?>
    where TFilter : IFilter
    where TParameters : IParameters
{}

//One of the variants
public class Filter<TWith, TWithout> : IFilter
    where TWith : IWith
    where TWithout : IWithout
{}

//One of the variants, also containing <T>, <T1, T2, T3>...
public class Parameters<T1, T2> : IParameters
    where T1 : IParameter
    where T2 : IParameter
{}

//One of the variants, also containing <T>, <T1, T2>...
public class With<T1, T2, T3> : IWith
    where T1 : struct
    where T2 : struct
    where T3 : struct
{}

public class Without<T> : IWithout where T : struct
{
    //All constructors are internal to disable
    //object creation. Class should be used
    //by user only as a iterator template.
    internal Without() {}
}

public class Mutable<T> : IParameter where T : struct {}

public class Immutable<T> : IParameter where T : struct {}
```

Toto riešenie ponecháva otvorené otázky, napríklad čo bude enumerovaný typ triedy `Entities`. Jedna z možností je neimplementovať rozhranie `IEnumerable<T>`, ale ponúkať osobitné metódy bližšie špecifikujúcejšie charakter iterátora. Použitie v systéme teda môže vyzerať nasledovne:

Výpis kódu 61 Príklad využitia entitového iterátoru:

```
[System<Stage>(Stage.Draw)]
[With<Box, Player>]
public static void KillCollidingNonPlayerBoxes(
    ref Box box,
    Entities<Filter
                <With<Box>,
                Without<Player>>,
            Parameters
                <Mutable<Box>>
            iterator
)
{
    // Entities implements IEnumerable
    foreach (var otherBox in iterator)
    {
        if (colliding)
            otherBox.removeEntity();
    }

    // Entities implements separate method,
    // might be used if IEnumerable not possible
    foreach (var otherBox in iterator.GetEntities<Box>())
    {
        if (colliding)
            otherBox.removeEntity();
    }
}
```

Systém vie teda jednoducho špecifikovať s akými entitami chce interagovať. Všetky informácie ohľadom interakcií vieme analyzovať už za kompilačnej doby, ponúka sa nám priestor vytvárať rôzne optimalizácie a opatrenia na konzistenciu chodu programu. Interne v hernej slučke ECS vytvára nové queries pomocou ktorých pristupuje ku špecifikovaným entitám. Interpretácii riešenia tohto problému je samozrejme viacero, s potenciálom pre diskusiu. V spomínanom riešení sa opierame o možnosť generácie kódu na mieru. Z definície typu vieme vygenerovať špeciálne triedy spĺňajúce iteračné podmienky už v kompilačnom čase.

4.4.2 Optimalizácia výkonu a pamäťovej stopy

V problematike výkona hier je dôležité, aby zbehol herný cyklus čo najrýchlejšie a preto herné enginy alokujú a inicializujú dáta vopred. Inicializácia hry prebieha práve raz a čas strávený incializáciou je často zanedbateľný.

Komponentové polia pozostávajú z chunkov, teda kompaktných polí naplnených komponentami. Ponúka sa možnosť spojenia chunkov po zbehnutí inicializačných systémov, poprípade označenie trvalých entít. Nastane tak dodatočné skompaktnenie komponentov, čím sa zrýchli iterácia entít počas behu hry.

Zaujímavé by bolo sledovať, v akých prípadoch je rýchlejšie vytvárať samostatné úlohy na aktualizáciu entít pre každý chunk narozdiel od jednej úlohy pre jeden komponentový bazén. Takýchto potenciálnych heuristických optimalizácii sa vo frameworku nachádza viac. Možnosťou je cachovať archetypové queries, alebo aktualizovať archetypy len v prípade pridania alebo odobratia entít. Beh systémov je v súčasnosti paralelizovaný triviálne, neberie ohľad na počet entít a nerozdeľuje prácu do blokov v závislosti od preferovaného počtu vlákien procesora.

Pamäťová stopa hernej slučky pozostáva taktiež z alokácie query iterátorov. Táto alokovaná pamäť je krátkodobá a vďaka vlastnostiam C# garbage collectoru je efektívne spracovávaná. V každom prípade stojí za zmienku a znížením počtu veľmi častých alokácií môžeme vidieť zlepšenie vo výkone hernej slučky.

Posledným problémom z hľadiska optimalizácie sú neefektívne dátové štruktúry a algoritmy. Implementácia určitých častí frameworku je založená na neoptimálnych dátových štruktúrach pre danú funkcionalitu. Na presnejšiu analýzu týchto potenciálnych kritických častí je potrebná bližšia analýza v budúcnosti.

4.4.3 Kompletná konfigurácia v kompilačnom čase

Budujúc na spomínaných optimalizáciách sa predstavuje možnosť presunúť tvorbu archetypov a komponentov do kompilačnej doby. Všetky systémy herného sveta sú známe počas kompilácie. Ak by sa nám podarilo správne analyzovať, aké archetypy plynú z behu systémov, boli by sme schopní vytvoriť kompletnú mapu archetypov vopred. To by výrazne zefektivnilo prácu s entitami. Vyhľadávanie správnych archetypov pre systémy by bolo možné v konštantnom čase, prístup ku entitám by bol priamy, bez dodatočných dereferencií archetypov v manažéri. Komponentový manažér by bol optimalizovaný do kompilačnej doby vďaka informáciám o komponentoch, ktoré systémy používajú. Bolo by možné zoradiť archetypy a optimalizovať prístupové vzory ku dátam. Generátor vie teoreticky analyzovať, aké typy entít systém vytvára, no jednoduchším riešením by bolo pridanie dodatočnej anotácie. Tá by informovala generátor o manipulácii s entitami.

4.4.4 Plánovanie systémov

Systémy sa plánujú podľa etáp ku ktorým sú priradené. Toto riešenie je prehľadné aj pre používateľa a prináša intuitívny mentálny model nad hernou slučkou. Z fungovania frameworku máme zaručené, že všetky systémy jednej etapy zbehnú pred začatím druhej etapy. Problém nastáva v situácii, kedy máme v pláne spustiť dlhotrvajúci systém. Komponenty, ktoré upravuje sú nezávislé od zvyšných dát, poprípade s nimi potrebujeme pracovať až neskôr v hernej slučke. Ideálne by sme systém chceli spustiť na začiatku cyklu v prvej etape a nebrať ho do úvahy, keď čakáme na systémy danej etapy. Plynulo by sa spúšťali systémy neskorších etáp a spomínaný systém by bol oslobodený od princípu etapy. Garanciou by bolo jedine ukončenie do začiatku ďalšieho cyklu. Potenciálne riešenie spočíva vo vhodných anotáciách, tie by dokázali určiť charakter systému. Možnosťou je vytvoriť graf závislostí systémov, no to by vytváralo možnosť cyklickej závislosti. Posun ku anotáciám špecifických závislostí je zatiaľ na zváženie.

4.4.5 Interakcia s modulmi

Využitie modulov vyžaduje úpravu exportovaných metadát. Metadáta nepoznajú návrh projektu a nevedia dosadiť informácie o etape systému, alebo typoch na ktorých sa má spúšťať. Tieto informácie je nutné manuálne doplniť bez možnosti diagnostiky a kontroly, čo pridáva na potenciálnej chybovosti hier. Pridaním schopnosti uložiť etapy a vlákna v metadátach by sme umožnili tvorbu konfiguračnej aplikácie. Tá by umožňovala prepájať systémy, definovať etapy a ponúkala by plne vizuálne skriptovacie prostredie. Interne by vhodne upravovala metadátové súbory ale pre používateľa by to bolo ukryté pod intutívnu vizualizáciu. Dodatočné rozšírenie aplikácie by mohlo spravovať knižnicu modulov a umožniť automatický import žiadaných modulov. Tvorcovia hier by získali elegantný spôsob interakcie s hernými systémami a komponentami.

4.4.6 Diagnostika

Lepenka v súčasnosti neponúka užívateľovi dostatočné chybové hlásenia. Uplatniť pravidlá používania frameworku je obtiažnejšie kvôli spôsobu definície častí hry. Na správnu kontrolu systémov je nutné vytvoriť inkrementálny generátor analyzujúci kód projektu. Generátor by informoval používateľa, či sú systémy vhodne definované, alebo by kontroloval správnosť konfigurácie, poprípade ponúkal nápovedy o potenciálnych vylepšeniach v kóde. Súčasným problémom Lepenky je absencia akejkoľvek kontroly správneho používania frameworku, ten správne funguje len ak je kód validný. Práca s frameworkom

by taktiež bola zlepšená dodatočným upresnením a dodefinovaním výnimiek.

Ďalším problémom je absencia profilovania a debugových správ. Framework momentálne neponúka nástroje na analýzu pamäťovej stopy alebo výkonnosti vytvorených programov. Pridaním vhodných konfigurácii behu programu a nástrojov na vizualizáciu interných dátových štruktúr by sme umožnili jemné dolaďovanie chodu programu.

4.5 Benchmark

Pri testovaní výkonu Lepenky sme ju porovnávali s objektovým modelom. Vytvorili sme si modelový príklad s entitou obsahujúcou tri typy komponentov. V hernej slučke interaguje s troma systémami. Analogický príklad sme vytvorili v objektovom modeli, teda triedu obsahujúcu relevantné dáta s aktualizačnou funkciou simulujúcou rovnakú logiku. Vygenerovaná herná slučka nevytvára optimalizácie ani heuristiky na základe počtu entít. Taktiež nerozdeľuje spúšťanie systémov medzi vlákna po úsekoch, vytvára samostatnú úlohu pre každé spustenie systémov. Výsledkom je mnohonásobné spomalenie oproti objektovému modelu. Po pridaní manuálnych úprav v hernej slučke a aplikácií možných vylepšení z analýzy v sekcii 4.4.2 sa nám podarilo docieliť kompetetívneho výkonu Lepenky.

Benchmark simuluje 1000 cyklov hernej slučke, pričom počet entít sa v testoch mení. Testovanie bolo vykonané s nasledujúcou konfiguráciou:

- Benchmark.NET v0.13.1
- OS Windows 10.0.19044.1826 (21H2)
- Procesor AMD Ryzen 5 3600, 12 logických a 6 fyzických jadier
- .NET SDK verzie 6.0.302

Výsledkom je rýchlejší beh ECS oproti objektovému modelu pri rastúcom počte objektov herného sveta. Objektový model zvláda spracovávať nižší počet objektov rýchlejšie. Túto vlastnosť avšak nehodnotíme ako dôležitú, pretože herný systém je závislý na výkone práve pri vysokom počte herných objektov.

Method	ObjectCount	Mean	Error	StdDev
ObjectModelTest	1000	2.739 ms	0.0360 ms	0.0337 ms
ObjectModelTest	10000	27.307 ms	0.0770 ms	0.0682 ms
ObjectModelTest	100000	398.362 ms	3.2003 ms	2.9936 ms
ECSTest	1000	19.76 ms	0.236 ms	0.221 ms
ECSTest	10000	24.87 ms	0.168 ms	0.157 ms
ECSTest	100000	117.70 ms	0.257 ms	0.240 ms

Tabuľka 4.1 Porovnanie výkonu ECS a objektového modelu.

Je dôležité podotknúť, že tento výsledok bol umožnený až po využití triedy `Partitioner` a zároveň cachovaním archetypov. Dôvodom benchmarku je ukázať potenciál v princípoch ECS a v budúcnosti vývoja frameworku Lepenka.

Seznam použité literatury

[1] Romain Dillet. *Unity CEO says half of all games are built on Unity*. TechCrunch. 2018. URL: https://techcrunch.com/2018/09/05/unity-ceo-says-half-of-all-games-are-built-on-unity/.

[2] trends.google.com. *Google Trends*. 2012. URL: https://trends.google.com/trends/explore?date=all&q=entity%20component%20system.

[3] Sander Mertens. *Entity Component System FAQ*. GitHub. 2022. URL: https://github.com/SanderMertens/ecs-faq.

[4] Jason Gregory. *Game engine architecture*. AK Peters/CRC Press, 2018.

[5] Mathai Joseph et al. *Real-Time Systems: specification, verification, and analysis*. Zv. 62. prentice Hall Englewood Cliffs, 1996.

[6] Michael Dawson. *Beginning C++ Through Game Programming*. Course Technology, a part of Cengage Learning, 2011.

[7] Bobby Anguelov. *Game Engine Entity/Object Models*. Youtube. 2020. URL: https://www.youtube.com/watch?v=jjEsB611kxs.

[8] Briar Lee Mitchell. *Game design essentials*. John Wiley & Sons, 2012.

[9] Michele Caini. *ECS back and forth*. GitHub. 2019. URL: https://skypjack.github.io/2019-02-14-ecs-baf-part-1/.

[10] Unity Technologies. *Data Oriented Technology Stack*. 2022. URL: https://forum.unity.com/forums/data-oriented-technology-stack.147/.

Dodatok A

Používanie Lepenky

Lepenka je zdieľaná pomocou správcu balíkov NuGet. Lepenka momentálne ponúka 4 balíčky a to samotné ECS, generátor modulov, generátor vesmíru a šablónu projektu. Platforma .NET umožňuje jednoduchú výstavbu projektu. Na využitie Lepenky je doporučené využiť šablónu. Šablóna `Lepenka.Templates` obsahuje konfiguráciu projektu s predpripravenou štruktúrou projektu. V šablóne je ukážkový projekt so systémami a komponentami. Príklad obsahuje potrebné súbory ako `Universe.xml` a taktiež správne referencuje knižnice vo svojej konfigurácii.

A.1 Tvorba projektu

Začať tvoriť pomocou šablóny je jednoduché. Využitím .NET CLI si používateľ nainštaluje balíček šablón. Pri tvorbe projektu mu bude umožnené zvoliť si novo nainštalovanú šablónu, ktorá obsahuje ukážky využívania Lepenky.

```
dotnet new --install Lepenka.Templates::0.4.2
```

Ak používateľ plánuje pridať balíčky manuálne, znova je mu to umožnené pomocou NuGet balíku, teda napríklad použitím .NET CLI.

```
dotnet add package Lepenka --version 0.0.5-beta
dotnet add package Lepenka.Universe.Generator --version 0.0.6-beta
```

V prípade manuálneho pridania balíkov je potrebné upraviť projektové konfigurácie podľa nariadení používateľskej dokumentácie.8 Bez správnej konfigurácie projektu generátor nevytvára hernú slučku. Pre správne využitie modulov je potrebné referencovať ich v projekte. 26

A.2 Tvorba modulov

Využitie generátora vyžaduje správnu konfiguráciu projektu ako je spomenuté vo výpise 8. Metadátový súbor je možné vytvoriť aj manuálne, nie je to odporúčané kvôli absencii kontroly správnosti súboru a teda náchylnosti ku chybám v súbore. Používateľ v kóde vytvorí triedu ktorej systémy a komponenty vhodne anotuje.

Po pridaní balíčku a kompilácií sa v projektových priečinkoch vytvoria XML súbory s metadátami pre exportované časti.

```
dotnet add package Lepenka.Module.Generator --version 0.0.4-beta
```