

# **Interaktivní vizualizace evolučních algoritmů pomocí grafického engine**

Bc. David Kučera

---

Diplomová práce  
2025



Univerzita Tomáše Bati ve Zlíně  
Fakulta aplikované informatiky

---



Univerzita Tomáše Bati ve Zlíně  
Fakulta aplikované informatiky  
Ústav informatiky a umělé inteligence

Akademický rok: 2024/2025

# ZADÁNÍ DIPLOMOVÉ PRÁCE

(projektu, uměleckého díla, uměleckého výkonu)

Jméno a příjmení: Bc. David Kučera  
Osobní číslo: A23316  
Studijní program: N0613A140022 Informační technologie  
Specializace: Softwarové inženýrství  
Forma studia: Prezenční  
Téma práce: Interaktivní vizualizace evolučních algoritmů pomocí grafického engine  
Téma práce anglicky: Interactive Visualization of Evolutionary Algorithms Using a Graphics Engine

## Zásady pro vypracování

- Seznamte se s metodami evolučních algoritmů a principy, které jsou klíčové pro vizualizaci jejich optimalizačního procesu.
- Vyberte vhodný grafický engine, který umožní interaktivní vizualizaci evolučních algoritmů.
- Naprogramujte aplikaci, která zobrazí vybraný řez účelové funkce jako plochu s barevně označenými extrémy.
- Implementujte vizualizaci pohybu jedinců vybraných evolučních algoritmů na ploše účelové funkce.
- Upravte vizualizaci tak, aby se pohyb jedinců zobrazoval ve zpomaleném čase, umožňujícím detailní sledování jejich chování.
- Navrhněte uživatelské rozhraní, které studentům umožní snadné experimentování se změnami parametrů algoritmů.
- Vyhodnoťte potenciál aplikace jako vzdělávací a vědecký nástroj, a to zejména v kontextu experimentálního pozorování a vývoje nových verzí algoritmů.

Forma zpracování diplomové práce: **tištěná/elektronická**

**Seznam doporučené literatury:**

1. **Mlýnek, D.** *Umělá inteligence v problémech globální optimalizace*. Ostrava: Vysoká škola báňská – Technická univerzita Ostrava, 2011. 214 s. ISBN 978-80-248-2403-8.
2. **De Jong, K. A.** *Evolutionary Computation: A Unified Approach*. Cambridge, MA: MIT Press, 2006. 256 s. ISBN 978-0-262-04194-2.
3. **Zelinka, I., Oplatková, Z., Šeda, M., Ošmera, P., Včelař, F.** *Evoluční výpočetní techniky, principy a aplikace*. 2009. ISBN 978-80-7300-218-3.
4. **Simon, D.** *Evolutionary optimization algorithms: biologically-inspired and population-based approaches to computer intelligence*. Hoboken, NJ: Wiley, 2013. ISBN 978-0-470-93741-9.
5. **Yang, X. S.** *Recent advances in swarm intelligence and evolutionary computation*. Cham: Springer, 2015. ISBN 978-3-319-13825-1.

Vedoucí diplomové práce: **Ing. Bc. Pavel Vařacha, Ph.D.**  
Ústav informatiky a umělé inteligence

Konzultant diplomové práce: **Ing. Adam Viktorin, Ph.D.**  
Ústav informatiky a umělé inteligence

Datum zadání diplomové práce: **27. října 2024**

Termín odevzdání diplomové práce: **2. června 2025**



L.S.

**doc. Ing. Jiří Vojtěšek, Ph.D. v.r.**  
děkan

**prof. Mgr. Roman Jašek, Ph.D., DBA v.r.**  
ředitel ústavu

Ve Zlíně dne 17. listopadu 2024

# PROHLÁŠENÍ AUTORA DIPLOMOVÉ PRÁCE

## Beru na vědomí, že

- odevzdáním diplomové práce souhlasím se zveřejněním své práce podle zákona č. 111/1998 Sb., v platném znění bez ohledu na výsledek obhajoby;
- diplomová práce bude uložena v elektronické podobě v univerzitním informačním systému a bude dostupná k nahlédnutí;
- jedno vyhotovení diplomové práce v listinné podobě bude ponecháno Univerzitě Tomáše Bati ve Zlíně k uložení;
- na moji diplomovou práci se plně vztahuje zákon č. 121/2000 Sb. o právu autorském, o právech souvisejících s právem autorským a o změně některých zákonů (autorský zákon) ve znění pozdějších právních předpisů, zejm. § 35 odst. 3;
- podle § 60 odst. 1 autorského zákona má Univerzita Tomáše Bati ve Zlíně právo na uzavření licenční smlouvy o užití školního díla v rozsahu § 12 odst. 4 autorského zákona;
- podle § 60 odst. 2 a 3 mohu užít své dílo – diplomovou práci – nebo poskytnout licenci k jejímu využití jen s předchozím písemným souhlasem Univerzity Tomáše Bati ve Zlíně, která je oprávněna v takovém případě ode mne požadovat přiměřený příspěvek na úhradu nákladů, které byly Univerzitou Tomáše Bati ve Zlíně na vytvoření díla vynaloženy (až do jejich skutečné výše);
- pokud bylo k vypracování diplomové práce využito softwaru poskytnutého Univerzitou Tomáše Bati ve Zlíně nebo jinými subjekty pouze ke studijním a výzkumným účelům (tj. k nekomerčnímu využití), nelze výsledky diplomové práce využít ke komerčním účelům;
- pokud je výstupem diplomové práce jakýkoliv softwarový produkt, považují se za součást práce rovněž i zdrojové kódy, popř. soubory, ze kterých se projekt skládá; neodevzdání této součásti může být důvodem k neobhájení práce.

## Prohlašuji, že

- jsem na diplomové práci pracoval(a) samostatně a použitou literaturu jsem řádně citoval(a); v případě publikace výsledků budu uveden(a) jako spoluautor;
- odevzdaná verze diplomové práce a verze elektronická nahraná do IS/STAG jsou obsahově totožné.

Ve Zlíně, dne .....

.....  
podpis autora



## **ABSTRAKT**

Tato práce se zaměřuje na vizualizaci evolučních algoritmů v prostředí Unity. Cílem je vytvořit interaktivní aplikaci, která zobrazuje chování algoritmů, jako jsou genetické algoritmy (GA), evoluční strategie (ES), particle swarm optimization (PSO), diferenciální evoluce (DE) a SOMA.

Aplikace, implementovaná v jazyce C#, umožňuje vizualizovat pohyb jedinců v prostoru vybrané účelové funkce, analyzovat optimalizační proces a experimentovat s parametry. Obsahuje zpomalenou simulaci pro podrobné sledování vývoje populací.

Výsledný nástroj slouží k výuce i výzkumu, propojuje teorii s praxí a otevírá možnosti pro budoucí rozšíření.

**Klíčová slova:** Evoluční algoritmy, genetické algoritmy, PSO, diferenciální evoluce, SOMA, Unity, vizualizace, optimalizace, interaktivní aplikace

## **ABSTRACT**

This thesis focuses on the visualization of evolutionary algorithms in the Unity environment. The aim is to create an interactive application that displays the behavior of algorithms such as Genetic Algorithms (GA), Evolution Strategy (ES), Particle Swarm Optimization (PSO), Differential Evolution (DE), and Self-Organizing Migrating Algorithm (SOMA).

The application, implemented in C#, visualizes the movement of individuals across the objective function landscape, analyzes the optimization process, and allows users to experiment with algorithm parameters. It includes a slowed-down simulation mode for detailed observation of population dynamics.

The resulting tool serves both educational and research purposes, bridging theory with practice and offering opportunities for future expansion.

**Keywords:** Evolutionary algorithms, GA, PSO, DE, SOMA, Unity, visualization, optimization

Rád bych poděkoval panu Ing. Bc. Pavlu Vařachovi, Ph.D. za odborné vedení, cenné rady a vstřícný přístup během zpracování této diplomové práce. Dále děkuji panu Ing. Adamu Viktorínovi, Ph.D. za konzultace, podnětné připomínky a návrhy na rozšíření práce, které přispěly k jejímu zlepšení po obsahové i technické stránce.

Poděkování patří taky celé mojí rodině a kamarádům, kteří mi byli po celou dobu studia nejen oporou, ale také důvodem, díky kterému jsem si studium užil.

# OBSAH

|                                                                         |           |
|-------------------------------------------------------------------------|-----------|
| <b>ÚVOD.....</b>                                                        | <b>11</b> |
| <b>I TEORETICKÁ ČÁST .....</b>                                          | <b>12</b> |
| <b>1 ÚVOD DO EVOLUČNÍCH ALGORITMŮ .....</b>                             | <b>13</b> |
| 1.1 HISTORIE A VÝVOJ EVOLUČNÍCH ALGORITMŮ .....                         | 14        |
| 1.2 PRINCIPY FUNGOVÁNÍ EVOLUČNÍCH ALGORITMŮ .....                       | 15        |
| 1.3 TYPY EVOLUČNÍCH ALGORITMŮ .....                                     | 18        |
| 1.3.1 Genetické algoritmy (GA).....                                     | 18        |
| 1.3.2 Evoluční strategie (ES) .....                                     | 19        |
| 1.3.3 Diferenciální evoluce (DE) .....                                  | 20        |
| 1.3.4 Particle Swarm Optimization (PSO) .....                           | 21        |
| 1.3.5 Self-Organizing Migrating Algorithm (SOMA) .....                  | 22        |
| 1.4 METODIKA TESTOVÁNÍ OPTIMALIZAČNÍCH ALGORITMŮ .....                  | 23        |
| 1.5 HODNOTÍCÍ KRITÉRIA A POSTUP EXPERIMENTU .....                       | 28        |
| <b>2 SROVNÁNÍ A VÝBĚR GRAFICKÉHO ENGINU .....</b>                       | <b>30</b> |
| 2.1 VÝZNAM GRAFICKÝCH ENGINŮ PRO VIZUALIZACI EVOLUČNÍCH ALGORITMŮ ..... | 30        |
| 2.2 VLASTNOSTI GRAFICKÝCH ENGINŮ .....                                  | 30        |
| 2.3 VÝVOJ GRAFICKÝCH ENGINŮ A JEJICH SOUČASNÉ VYUŽITÍ.....              | 32        |
| 2.4 PŘEHLED VYBRANÝCH GRAFICKÝCH ENGINŮ .....                           | 32        |
| 2.4.1 Unity.....                                                        | 32        |
| 2.4.2 Unreal Engine.....                                                | 33        |
| 2.4.3 Godot Engine .....                                                | 34        |
| 2.4.4 CryEngine .....                                                   | 34        |
| 2.5 VÝBĚR NEJVHODNĚJŠÍHO GRAFICKÉHO ENGINU .....                        | 35        |
| <b>II PRAKTICKÁ ČÁST .....</b>                                          | <b>37</b> |
| <b>3 NÁVRH ŘEŠENÍ A IMPLEMENTAČNÍ ARCHITEKTURA .....</b>                | <b>38</b> |
| 3.1 SCÉNOVÁ HIERARCHIE A ROZLOŽENÍ KOMPONENT .....                      | 39        |
| 3.1.1 Struktura projektu.....                                           | 39        |
| 3.1.2 Vizualizační vrstva.....                                          | 41        |
| 3.1.3 Řízení evolučního algoritmu .....                                 | 41        |
| 3.1.4 Kamerový systém.....                                              | 42        |
| 3.2 UŽIVATELSKÉ ROZHRANÍ .....                                          | 42        |
| 3.2.1 Horní lišta (menu bar) .....                                      | 42        |
| 3.2.2 Horní pod-lišta .....                                             | 43        |
| 3.2.3 Levý a pravý ovládací panel .....                                 | 44        |
| 3.2.4 Vizualizační plocha .....                                         | 47        |
| 3.2.5 Fitness graf a spodní panel.....                                  | 48        |
| 3.3 IMPLEMENTACE ALGORITMŮ .....                                        | 49        |
| 3.3.1 Genetický algoritmus .....                                        | 49        |

|          |                                                       |            |
|----------|-------------------------------------------------------|------------|
| 3.3.2    | Evoluční strategie .....                              | 51         |
| 3.3.3    | Diferenciální evoluce .....                           | 52         |
| 3.3.4    | PSO.....                                              | 53         |
| 3.3.5    | SOMA .....                                            | 54         |
| 3.4      | HLAVNÍ KOMPONENTY A JEJICH ÚČEL.....                  | 56         |
| 3.4.1    | Práce s evolucí.....                                  | 58         |
| 3.4.2    | Vizualizace jedinců a jejich evoluce v prostoru ..... | 63         |
| 3.4.3    | Vykreslení optimalizačních funkcí a prostředí.....    | 68         |
| 3.4.4    | Slicing funkce.....                                   | 72         |
| 3.4.5    | Graf průběhu fitness .....                            | 75         |
| 3.4.6    | Režim porovnání .....                                 | 77         |
| 3.4.7    | Podpůrné třídy a import/export dat .....              | 80         |
| <b>4</b> | <b>TESTOVÁNÍ A MODULARITA.....</b>                    | <b>83</b>  |
| 4.1      | TESTOVÁNÍ A LADĚNÍ .....                              | 83         |
| 4.1.2    | Testování evolučních algoritmů .....                  | 84         |
| 4.1.3    | Testování řezů .....                                  | 85         |
| 4.1.4    | Testování uživatelského rozhraní .....                | 85         |
| 4.2      | MODULARITA A ROZŠÍŘITELNOST SYSTÉMU .....             | 86         |
| 4.2.1    | Architektonický přístup.....                          | 87         |
| 4.2.2    | Přidání nového evolučního algoritmu .....             | 87         |
| 4.2.3    | Přidání nové optimalizační funkce .....               | 87         |
| 4.2.4    | Rozšířitelnost a návrh uživatelského rozhraní.....    | 88         |
| <b>5</b> | <b>VYUŽITÍ A DOKUMENTACE.....</b>                     | <b>90</b>  |
| 5.1      | MOŽNOSTI REÁLNÉHO NAsAZENÍ A SCÉNÁŘE POUŽITÍ .....    | 90         |
| 5.1.1    | Využití ve vzdělávání.....                            | 90         |
| 5.1.2    | Prezentace a demonstrace.....                         | 90         |
| 5.1.3    | Budoucí rozšíření pro reálná data.....                | 91         |
| 5.2      | DETAILNÍ DOKUMENTACE UŽIVATELSKÉHO SCÉNÁŘE .....      | 91         |
|          | <b>ZÁVĚR.....</b>                                     | <b>94</b>  |
|          | <b>SEZNAM POUŽITÉ LITERATURY .....</b>                | <b>95</b>  |
|          | <b>SEZNAM OBRÁZKŮ .....</b>                           | <b>99</b>  |
|          | <b>SEZNAM TABULEK.....</b>                            | <b>101</b> |
|          | <b>SEZNAM POUŽITÝCH SYMBOLŮ A ZKRATEK .....</b>       | <b>102</b> |
|          | <b>SEZNAM PŘÍLOH.....</b>                             | <b>103</b> |

## ÚVOD

Evoluční algoritmy představují jednu z nejvýznamnějších metod pro řešení složitých optimalizačních problémů, které nelze efektivně vyřešit tradičními matematickými postupy. Jejich principy, vycházející z biologické evoluce, simulují procesy přirozeného výběru a adaptace. Díky své univerzálnosti nacházejí evoluční algoritmy široké uplatnění v oblastech, jako jsou umělá inteligence, strojové učení, automatizace a průmyslová optimalizace. Tyto algoritmy se vyznačují schopností efektivně prohledávat rozsáhlé prostorové oblasti a přizpůsobovat se měnícím se podmínkám.

Cílem této diplomové práce je navrhnout a implementovat interaktivní aplikaci, která vizualizuje průběh evolučních algoritmů na ploše účelové funkce. Aplikace bude sloužit nejen jako vzdělávací nástroj pro studenty informatiky, ale také jako analytická platforma pro výzkumníky, kteří mohou sledovat chování algoritmů a analyzovat jejich efektivitu při hledání optimálních řešení. Uživatelé budou moci experimentovat s parametry algoritmů, porovnávat jejich výkonnost a sledovat trajektorie konvergence.

Důraz je kladen na dynamickou vizualizaci, která zahrnuje zobrazení pohybu jedinců v prostoru účelové funkce a jejich konvergenci k lokálním i globálním extrémům. Vizualizace bude obsahovat prvky, jako jsou trajektorie pohybu, barevné označení hodnot fitness a zpomalené animace, které umožní detailní analýzu vývoje populací. Aplikace rovněž nabídne možnost sledovat vliv změn parametrů na rychlost konvergence a diverzitu řešení.

K realizaci projektu byl po analýze zvolen grafický engine Unity, který poskytuje výkonné nástroje pro 3D vizualizace a interaktivní ovládání. Skriptování v jazyce C# umožňuje flexibilní implementaci algoritmů a nastavení parametrů. Unity rovněž podporuje multiplatformní export, což umožní zpřístupnění aplikace na různých zařízeních. Modulární architektura aplikace zajistí možnost budoucího rozšíření, například integrací dalších algoritmů nebo účelových funkcí.

Tato práce má za cíl propojit teoretické znalosti s praktickou implementací a poskytnout uživatelům nástroj vhodný pro vzdělávací i výzkumné účely. Díky detailní vizualizaci evolučních algoritmů přispěje k hlubšímu porozumění optimalizačních procesů a nabídne prostor pro další experimentování a rozvoj.

## **I    TEORETICKÁ ČÁST**

## 1 ÚVOD DO EVOLUČNÍCH ALGORITMŮ

Evoluční algoritmy (EA) jsou skupinou optimalizačních metod inspirovaných přírodní evolucí a genetikou. Tyto algoritmy pracují s populací možných řešení, které se v průběhu iterací vylepšují pomocí operátorů, jako je například selekce nejlepších jedinců, křížení (rekombinace) a mutace, podobně jako tomu je v biologické evoluci [1][2][3]. Oproti klasickým gradientním nebo enumerativním metodám nevyžadují evoluční algoritmy, aby cílové funkce byly spojitě ani derivovatelné – stačí vypočítat kvalitu (fitness) daného kandidátního řešení. Díky tomu jsou odolné proti složitým povrchům fitness funkcí a neuváznou v lokálních extrémech tak snadno jako jiné tradiční metody [4][5]. Evoluční algoritmy typicky nemají žádné předpoklady o tvaru optimalizovaného problému, a proto dokážou najít řešení široké škály úloh, aniž by vyžadovaly specifické úpravy problému. Díky tomuto přístupu mají EA velkou výhodu oproti konvenčním optimalizačním přístupům.

Další předností evolučních algoritmů je paralelní prohledávání prostoru možných řešení. Práce s populací (množinou kandidátních řešení) umožňuje prozkoumávat více oblastí řešeného prostoru současně. Tím se zvyšuje šance na nalezení globálního optima i v případech, ve kterých je fitness funkce multimodální (má mnoho lokálních optim) nebo silně nelineární [4]. EA jsou také relativně snadno přenositelné mezi úlohami, stačí jen vhodně zvolit reprezentaci řešení a definovat fitness funkci. Samotné evoluční mechanismy (selekce, mutace, aj.) lze použít univerzálně. To z nich činí tzv. metaheuristiky, tj. obecné postupy použitelné napříč různými problémovými doménami [5].

V praxi našly evoluční algoritmy uplatnění v mnoha oborech. Ve financích se genetické a evoluční algoritmy využívají například pro optimalizaci investičních portfolií, obchodních strategií či pro kalibraci finančních modelů [6]. V robotice se evoluční algoritmy uplatnily při návrhu řídicích systémů robotů a tzv. evoluční robotice, například pro evoluci chůze šestinohých robotů nebo tvaru a chování robotických agentů [7][8]. V bioinformatice pomáhají řešit obtížné úlohy jako je zarovnávání DNA sekvencí, rekonstrukce fylogenetických stromů či predikce struktur proteinů, kde klasické algoritmy selhávají kvůli obrovské kombinatorice řešení. V průmyslové automatizaci a inženýrství jsou evoluční algoritmy využívány pro návrh optimalizovaných systémů, jako je třeba evoluční návrh elektronických obvodů [9], automatizace výroby, rozvrhování a plánování procesů či ladění parametrů regulačních obvodů [10].

## 1.1 Historie a vývoj evolučních algoritmů

První myšlenky využít evoluční principy k řešení technických problémů se objevily již v 50. a 60. letech 20. století. Teoretickým základem je Darwinova evoluční teorie selekce a adaptace druhů, která inspirovala průkopníky v informatice k napodobení těchto mechanismů na počítači. Za jednoho ze zakladatelů oboru se považuje John H. Holland, který v 60. letech formuloval koncept genetických algoritmů a v roce 1975 publikoval monografii *“Adaptation in Natural and Artificial Systems”*. V této monografii poprvé popsal využití Darwinových principů pro automatizované hledání řešení [1]. Hollandův přístup položil základy genetických algoritmů (GA) a inspiroval další výzkum v této oblasti hlavně ve Spojených státech [2].

Nezávisle na tom probíhal v Evropě vývoj tzv. evolučních strategií (ES). Ingo Rechenberg a Hans-Paul Schwefel na Technické univerzitě v Berlíně již v 60. letech experimentovali s evolučními postupy pro inženýrské úlohy, jako je optimalizace tvaru proudnic. Rechenberg obhájil roku 1971 doktorskou práci o evoluční strategii a v roce 1973 publikoval metodiku optimalizace technických systémů pomocí biologické evoluce [11]. Schwefel následně rozvinul tyto myšlenky a v roce 1977 zveřejnil práci o numerické optimalizaci pomocí evolučních strategií [12]. Evoluční strategie kladly důraz na reálnou reprezentaci jedinců a samo-adaptaci mutačních kroků a tím se lišily od Hollandových genetických algoritmů s binárním kódováním. Ve stejnou dobu se rozvíjel také směr evolutionary programming (EP), který založil Lawrence Fogel v Americe. Rozvinul také konečné automaty a soustředil se převážně na mutace bez rekombinace [13]. V 70. a 80. letech si vědci začali všimnout, že různé evoluční přístupy, jako genetické algoritmy, evoluční strategie nebo evoluční programování, mají hodně společného. Postupem času se z těchto podobností vytvořil společný směr, který je dnes známý jako evoluční výpočetní technika.

Další významný milník představuje vznik genetického programování (GP). John R. Koza v roce 1992 demonstroval, že genetický algoritmus může evolvovat celé programy (stromové struktury) namísto jedinců reprezentovaných pevně stanovenými vektory číselných parametrů. Jeho kniha ukázala potenciál evolučních algoritmů pro automatický návrh algoritmů. Tím výrazně rozšířila oblast působnosti evolučních metod [5][7]. Ve druhé polovině 90. let se objevily další bio-inspirované metaheuristiky, které se dnes řadí po bok evolučních algoritmů. Příkladem může být například Particle Swarm Optimization (PSO), inspirovaný chováním hejna ptáků či ryb. PSO navrhl James Kennedy a Russell Eberhart v roce 1995. Jedná se o populační optimalizační algoritmus, ve kterém

“částice” (kandidátní řešení) létají prostorem a vzájemně si sdílejí informaci o nejlepších známých pozicích [14]. Zhruba ve stejné době byla představena také diferenciální evoluce (DE). Tu publikovali Rainer Storn a Kenneth Price jako jednoduchý a účinný algoritmus pro globální optimalizaci v reálném prostoru využívající lineární kombinace rozdílů vektorů [15].

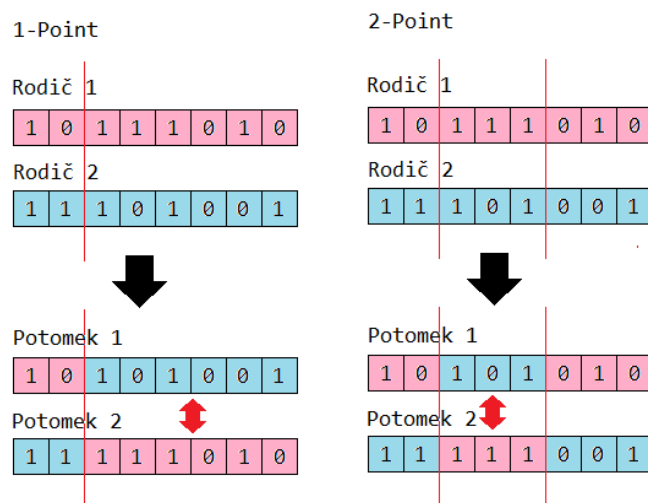
Vývoj evolučních algoritmů pokračoval rychle i po roce 2000. Byly navrženy četné varianty a vylepšení, například samo-adaptivní algoritmy, ko-evoluční algoritmy (soutěž nebo spolupráce více populací) či memetické algoritmy, které kombinují evoluční globální hledání s lokální heuristikou. Za zmínku stojí také český příspěvek v podobě algoritmu SOMA (Self-Organizing Migrating Algorithm), který navrhl Ivan Zelinka okolo roku 2004. SOMA vychází z principů rojové inteligence a využívá kooperativní migraci jedinců za nejlepším členem populace [10]. Evoluční a na evoluci navazující metaheuristiky se staly standardními nástroji v soft computingu. Svědčí o tom vznik specializovaných konferencí (GECCO, IEEE CEC aj.), časopisů a také rostoucí počet reálných aplikací.

## 1.2 Principy fungování evolučních algoritmů

Jak již bylo zmíněno v úvodním textu kapitoly 1, evoluční algoritmy sdílejí obecný postup inspirovaný neo-Darwinistickou evolucí. Každý EA pracuje s populací jedinců, neboli kandidátních řešení, která se v průběhu generací obměňuje a vylepšuje na základě principu přirozeného výběru. Základní cyklus evolučního algoritmu lze popsat následovně [2][4]:

1. **Inicializace populace:** Na počátku je náhodně vygenerována počáteční populace o dané velikosti (např. 50 či 100 jedinců). Každý jedinec představuje jedno možné řešení úlohy, které je obvykle zakódováno jako vektor parametrů, samozřejmě přizpůsobené danému algoritmu. U GA například jako binární řetězec, u ES by to mohl být reálný vektor. Inicializace náhodně pokryje prostor řešení, aby se prohledávání mohlo od začátku ubírat různými směry [5].
2. **Hodnocení (fitness):** Pro každého jedince v populaci se vyhodnotí jeho kvalita pomocí tzv. fitness, neboli účelové funkce. Fitness  $f(x)$  udává, jak dobře dané řešení  $x$  splňuje cíle optimalizační úlohy. Vyšší fitness (nebo naopak nižší hodnota cílové funkce u minimalizačních úloh) znamená lepší řešení. Toto hodnocení umožní porovnat jedince mezi sebou.

3. **Selekce rodičů:** Na základě fitness je zvolena podmnožina jedinců. Tito jedinci se stanou rodiči pro vytvoření nové generace. Selektace simuluje přírodní výběr, takže pravděpodobnost výběru jedince roste s jeho fitness (favorizují se kvalitnější řešení). Běžné selekční metody jsou např. ruletová selekce (proporční k fitness), turnajová selekce (náhodně se vybere skupinka jedinců a z ní postoupí ten nejlepší) či rangová selekce (pravděpodobnost je funkcí pořadí v žebříčku fitness). Cílem je umožnit šíření dobrých řešení do dalších generací, aniž by tím však byla zcela eliminována diverzita.
4. **Rekombinace (křížení):** Ze zvolených rodičů jsou vytvořeni potomci kombinací genetického materiálu rodičů. Tomuto procesu se říká křížení a představuje hlavní operátor prozkoumávání nových oblastí prostoru řešení. U genetických algoritmů je typické jednobodové či dvoubodové křížení u binárních řetězců, případně uniformní křížení, což je náhodná výměna bitů. U reálných vektorů můžeme aplikovat aritmetickou rekombinaci (potomek je např. průměrem rodičů) nebo jiné operace. Některé EA, jako třeba základní evoluční strategie, křížení nepoužívají vůbec nebo jen omezeně, spoléhaly totiž historicky spíše na mutaci. Obecně ale platí, že rekombinace pomáhá urychlit konvergenci [16].



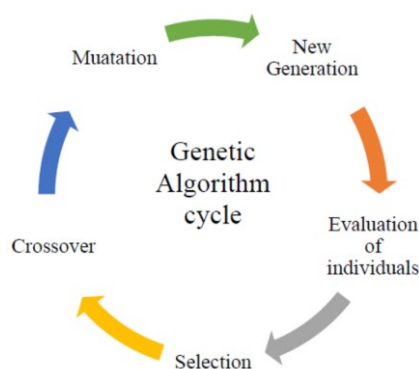
Obrázek 1 Ukázka jednobodového a dvoubodového křížení

(zdroj: Chalupník, 2012)

5. **Mutace:** Potomci vzniklí křížením jsou dále náhodně mutováni, tj. drobně pozměněni. Mutace simuluje biologické mutační jevy a zajišťuje genetickou diverzitu populace. To zabraňuje předčasné konvergenci k suboptimálnímu řešení. Mutační operátor obvykle náhodně mění část genetické informace jedince: u binární

reprezentace invertuje náhodně zvolené bity, u reálných vektorů přičítá malé náhodné hodnoty (třeba z Gaussova rozdělení). Mutace bývá aplikována s velmi nízkou pravděpodobností na každého potomka. Typicky to bývá např. 1 % u bitů, nebo jedna mutace na jedince. Správné nastavení míry mutace je obzvlášť důležité. Příliš vysoká mutace může vést k náhodnému prohledávání a tím rozbití dobrých řešení, naopak příliš nízká zase ke stagnaci populace. Praktické aspekty nastavení a implementace evolučních algoritmů shrnuje Pohlheim[17].

6. **Vznik nové generace:** Z původních rodičů a jejich nových potomků je sestavena nová populace pro další generaci. Opět se můžeme setkat s různými strategiemi náhrady generací. V generačních EA (tzv.  $(\mu, \lambda)$  strategie) rodiče po vytvoření potomků zanikají a celou novou generaci tvoří potomci. Naproti tomu stacionární EA (tzv.  $(\mu + \lambda)$  strategie) vloží potomky do původní populace a nadbytečné jedince, typicky ty nejhorší, odstraní. Často se zavádí také elitismus, tedy nejlépe adaptovaný jedinec generace  $t$  je přímo zkopírován do generace  $t+1$ , aby se neztratil dosud nejlepší nalezený kandidát řešení.
7. **Test ukončení:** Po vytvoření nové generace se opakuje cyklus opět krokem hodnocení. Algoritmus iteruje, dokud není splněna zastavovací podmínka. Tou může být dosažení požadované kvality řešení, nalezení známého optima, nebo prosté vyčerpání daného výpočetního rozpočtu, například maximální počet generací či funkčních hodnocení. Pokud kritérium není splněno, pokračuje EA opět krokem selekce a produkce potomků. Po ukončení algoritmu je výstupem nejlepší nalezené řešení, respektive řešení poslední generace.



Obrázek 2 Schéma evolučního cyklu

(Zdroj: Safouan et al. 2024 [18], obr. 1)

Tento obecný proces může mít mnoho konkrétních variant a vylepšení. Například existují víceobjektové evoluční algoritmy, které hodnotí jedince podle více fitness kritérií a používají specializované selekční metody (např. NSGA-II) k dosažení pareto-optimálních řešení [19]. Dále se v praxi upravuje kódování jedinců podle povahy problému. Kromě binárních a reálných reprezentací lze použít permutační reprezentaci pro úlohy obchodního cestujícího, stromovou strukturu pro genetické programování atd.. Různé typy EA také volí různé poměry mezi křížením a mutací a různé parametry populační velikosti nebo selekčního tlaku. Společným rysem však zůstává iterativní evoluční cyklus, který střídá fáze exploatace (výběr a kombinace dobrých řešení) a explorace (náhodné objevování nových řešení mutacemi).

Evoluční algoritmy jsou stochastické. To znamená, že zahrnují náhodu v iniciaci i v operátorech. To se týká výběrů rodičů, místa křížení, mutačních změn apod. Proto jednotlivé běhy téhož algoritmu mohou dospět k různým řešením a hodnocení výkonnosti EA vyžaduje statistický přístup, o kterém se více dozvíme v kapitole 1.4. Náhodnost zároveň zajišťuje, že algoritmus má šanci prozkoumat libovolnou část prostoru, ale při špatném nastavení může způsobit velký rozptyl kvality výsledků. V praxi se parametry, jako je velikost populace, pravděpodobnost mutace/křížení, počet generací, a další, ladí experimentálně nebo adaptivně během běhu tak, aby algoritmus dosáhl dobré rovnováhy mezi prohledáváním prostoru (diverzitou) a využíváním nejlepších nalezených řešení (konvergencí).

### 1.3 Typy evolučních algoritmů

Pod společný pojem evoluční algoritmy spadá několik hlavních kategorií, které se liší způsobem reprezentace řešení a operátory. Mezi nejvýznamnější patří genetické algoritmy (GA), evoluční strategie (ES), diferenciální evoluce (DE), optimalizace rojem částic (PSO) a SOMA. Níže jsou jednotlivé typy stručně popsány a porovnány z hlediska jejich charakteristik a vhodnosti pro různé úlohy [4][5].

#### 1.3.1 Genetické algoritmy (GA)

Genetické algoritmy představují nejznámější typ evolučních metod, zavedený Hollandem v roce 1975. Klasický GA pracuje s populací fixně dlouhých řetězců znaků. Každý bit řetězce je analogií genu a celý řetězec tvoří chromozom jedince. GA se pak snaží pomocí

genetických operátorů evolučně vylepšovat populaci těchto chromozomů tak, aby maximalizovaly danou fitness funkci.

Typický genetický algoritmus používá jednobodové nebo dvoubodové křížení. Vezmou se dva rodičovské řetězce a prohodí se jejich koncové části za náhodně zvoleným bodem zlomu. Tím vzniknou dva potomci řetězce kombinující genetický materiál rodičů. Mutace u GA probíhá jako náhodné invertování bitů ( $0 \rightarrow 1$  nebo  $1 \rightarrow 0$ ) s malou pravděpodobností. Díky binární reprezentaci jsou GA velmi obecné – jakýkoli problém je třeba zakódovat do binárního vektoru. Tento univerzální rámec byl aplikován na ohromné množství úloh od hledání cest, přes optimalizaci návrhů obvodů, až po evoluci umělých neuronových sítí [20].

GA jsou ideální zejména tam, kde jde řešení přirozeně zakódovat binárně nebo kde je řešený problém kombinatorické povahy jako optimalizace permutací nebo různé návrhové problémy. Varianty GA však existují i pro reálné proměnné a pro mnoho GA byly vyvinuty speciální operátory přizpůsobené konkrétním doménám (např. PMX křížení pro pořadí). Přehled základních principů genetických algoritmů včetně jejich variant je shrnut v práci [21].

Síla genetických algoritmů spočívá v jejich jednoduchosti a robustnosti, protože i s minimální doménovou znalostí často dokáží najít akceptovatelná řešení. Nevýhodou může být pomalejší konvergence v přesné fázi ladění řešení, protože GA spíše široce prozkoumávají, než jemně doladují (proto se někdy kombinují s lokální heuristikou, čímž vzniká memetický algoritmus) [20][22].

### 1.3.2 Evoluční strategie (ES)

Evoluční strategie vznikly původně pro optimalizaci spojitých parametrů inženýrských úloh. Klasická ES používá reálný vektor jako reprezentaci jedince. Jedinec tedy přímo kóduje například sadu reálných parametrů  $x = (x_1, x_2, \dots, x_n)$ . Hlavním evolučním operátorem ES je mutace: každý parametr vektoru je pozměněn o náhodnou hodnotu, typicky z Gaussova rozdělení  $\mathcal{N}(0, \sigma^2)$ . Směrodatná odchylka  $\sigma$  (tzv. mutační krok) může být v ES dynamicky samo-adaptována, protože je často součástí genotypu jedince a vylepšuje se spolu s  $x_i$  [12]. Tím si populace sama nastavuje vhodnou intenzitu mutací (adaptivní strategie).

Rekombinace v původních evolučních strategiích hrála menší roli než u GA, ale existuje – lze provádět např. aritmetické křížení. Potomek je v tomto případě kombinací

rodičů (např. průměr hodnot parametrů, nebo náhodná volba každé složky od jednoho z rodičů). Důraz je ale kladený na mutaci a selekci.

Evoluční strategie se velmi osvědčily pro kontinuální optimalizaci reálných funkcí, zvláště ve větších dimenzích (desítky až stovky parametrů), protože tam poskytují robustní výsledky[12]. Moderní ES zahrnují i pokročilé mechanismy jako CMA-ES (Covariance Matrix Adaptation ES), která adaptuje nejen velikost mutačních kroků, ale i směr. To znamená, že odhaduje kovarianční strukturu v datech, aby mutace byly efektivnější. Oproti GA nepotřebují ES žádné kódování, jelikož pracují přímo v prostoru řešení a mají menší množství voleb (většinou stačí nastavit  $(\mu, \lambda)$  a počáteční  $\sigma$ ). Jsou však méně univerzální pro diskrétní problémy, tam je lepší použít GA nebo kombinatorické heuristiky.

### 1.3.3 Diferenciální evoluce (DE)

Diferenciální evoluce je druh evolučního algoritmu představený Stornem a Pricem (1995) pro globální optimalizaci spojitých funkcí. Patří rovněž do kategorie reálných vektorových algoritmů, ale liší se netradičním mutačním operátorem. Zatímco klasická ES používá gaussovskou mutaci každé souřadnice, DE generuje nového jedince kombinací rozdílů náhodně vybraných vektorů z populace. Konkrétně, jeden ze základních mechanismů DE (tzv. “DE/rand/1/bin”) funguje tak, že se z aktuální populace vyberou tři různí jedinci  $x_a$ ,  $x_b$ ,  $x_c$ . Vytvoří se mutant  $v = x_a + F(x_b - x_c)$ , kde  $F$  je mutační faktor, který se typicky nachází v rozmezí 0.4–1.0. Tento mutant je pak zkombinován s původním jedincem  $x_t$  pomocí křížení: výsledný potomek  $u$  vznikne tak, že pro každou složku buď převezme hodnotu z  $v$  s pravděpodobností  $c_r$  (crossover rate), nebo zůstane hodnota z  $x_t$ . Potomek  $u$  se následně ohodnotí a porovná se jeden na jednoho se svým “cílovým” rodičem  $x_t$ . Pokud má potomek lepší fitness, nahradí rodiče v nové populaci, jinak se rodič ponechá [23].

Tento mechanismus zajišťuje směrovou mutaci využívající rozdíly existujících řešení. Výhodou je, že DE má velmi málo parametrových nastavování (v podstatě stačí jen nastavit velikost populace, faktor  $F$  a křížicí pravděpodobnost  $c_r$ ) a často dobře funguje s defaultními hodnotami. Diferenciální evoluce proslula svou jednoduchostí a účinností, zvláště pro nepříliš hladké a multimodální funkce. Je robustní a vyžaduje jen hodnocení funkce, nikoli její derivaci, stejně jako jiné EA. Empirické studie [4][5] ukazují, že DE patří k nejvýkonnějším algoritmům pro řadu standardních testovacích funkcí, a to i ve srovnání s komplikovanějšími metodami.

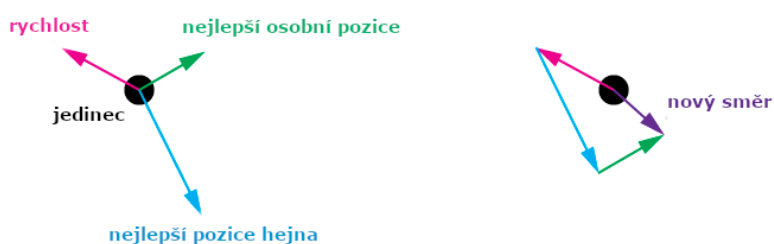
DE je vhodná primárně pro numerické kontinuální problémy. Uplatní se v inženýrských optimalizacích, v ladění parametrů modelů, při učení neuronových sítí apod. Méně se hodí pro diskrétní úlohy, ale existují i úpravy DE pro binární vektory. Ve srovnání s GA a ES má DE často rychlejší konvergenci, ale může být citlivější na volbu parametru  $F$  a na dostatečnou diverzitu populace. Při nevhodném nastavení  $F$  může populace stagnovat (při malém  $F$ ) nebo chaoticky poletovat (při nadměrném  $F$ ). V praxi se proto někdy  $F$  adaptuje, nebo se kombinuje více strategií mutace DE v tzv. self-adaptive DE.

### 1.3.4 Particle Swarm Optimization (PSO)

Optimalizace rojem částic není evoluční algoritmus v užším smyslu, protože nevychází z konceptu přežití potomků, ale často se řadí mezi evoluční a rojové metody díky populaci a biologické inspiraci. PSO navrhli Kennedy a Eberhart inspirováni kolektivním chováním ptáků při hledání potravy. V PSO je udržována populace  $N$  částic a každá má svou pozici  $x_i$  v hledaném prostoru a také vektor rychlosti  $v_i$ . Na rozdíl od GA zde neprobíhá křížení ani mutace. Částice se pohybují spojitým prostorem a postupně konvergují k optimu díky sdílení informací. Každá částice v každé iteraci aktualizuje svou rychlost a polohu podle vzorců:

$$v_i = \omega v_i + c_1 r_1 (p_i^{best} - x_i) + c_2 r_2 (g^{best} - x_i)$$

$$x_i = x_i + v_i$$



Obrázek 3 Aktualizace polohy PSO algoritmu

(zdroj: vlastní)

kde  $p_i^{best}$  je nejlepší pozice, jakou částice  $i$  sama navštívila, a  $g^{best}$  je nejlepší pozice nalezená libovolnou částicí v populaci (globální sdílená informace). Parametry  $c_1$ ,  $c_2$  jsou akcelerační koeficienty a  $r_1$ ,  $r_2$  náhodné hodnoty z  $[0,1]$  generované pro stochastické vážení vlivu.  $\omega$  je tzv. faktor setrvačnosti, který určuje, do jaké míry si částice zachovává svou předchozí rychlost. Laicky řečeno: každá částice létá prostorem a je přitahována jak směrem

ke svému dosud nejúspěšnějšímu známému místu, tak ke globálně nejlepšímu místu nalezenému hejnem, přičemž  $\omega$  tlumí či posiluje její dosavadní směr.

PSO kombinuje kooperativní a kompetitivní prvek. Částice spolu nesoupeří přímo o přežití, ale nepřímo tím, že se snaží být lepší a informují ostatní o svém úspěchu. Výsledkem bývá velmi rychlá konvergence hejna k dobrému řešení, zejména v raných fázích. PSO má minimum parametrů a snadno se implementuje. Je proslulý tím, že na mnoha úlohách dosahuje srovnatelné kvality řešení jako EA, ale často s menším počtem hodnocení. Je však známo, že PSO může snadno uvíznout v lokálním optimu, protože jakmile částice objeví slibnou oblast, celý roj se k ní rychle stáhne a opustí průzkum jinde. Proto existují varianty jako PSO s lokálním sousedstvím (částice sdílí info jen s nejbližšími sousedy, ne globálně) nebo adaptivní  $\omega$ , které udržují diverzitu [24].

PSO, stejně jako všechny již zmíněné algoritmy, se využívá opět hlavně pro spojitě optimalizace a pro některé kombinatorické problémy (existují například discrete PSO varianty). Uplatnil se v úlohách jako ladění vah neuronových sítí, optimalizace řídicích parametrů, návrh struktury výpočetních modelů, aj. [25] [26]. V kontextu evolučních algoritmů se o PSO často uvažuje jako o metodě, která doplňuje EA. Někdy se hybridizuje, nebo se používá PSO pro rychlé nalezení dobrého základu řešení a EA pro jeho doladění.

### 1.3.5 Self-Organizing Migrating Algorithm (SOMA)

SOMA (Self-Organizing Migrating Algorithm) je evoluční/rojový algoritmus původně vyvinutý Ivanem Zelinkou a spoluautory. Patří do širší rodiny swarm-based metod a lze jej charakterizovat jako jednoduchou imitaci sociálního chování skupiny jedinců putujících krajinou za potravou. Na rozdíl od PSO nepoužívá rychlosti, ale operuje pouze s pozicemi jedinců v populaci.

Princip SOMA je následující: v každé iteraci se najde nejlepší jedinec, kterému se říká leader, z aktuální populace podle fitness. Ostatní jedinci, neboli migranti se pak pohybují směrem k leaderovi. Každý migrant se posouvá po spojnici mezi svou aktuální pozicí a pozicí leadera, a to v krocích daných parametrem *Step*. Urazí přitom celkovou dráhu omezenou parametrem *PathLength*. Konkrétně, pokud  $X_L$  je pozice leadera a  $X_M$  pozice migranta, pak migrant bude postupně testovat pozice  $X_M + k(X_L - X_M)$  pro  $k = \text{step}, 2 * \text{step}, 3 * \text{step}, \dots$  dokud nepřekoná vzdálenost *PathLength* [10]. Na každé takové mezipozici se spočítá fitness a pokud je lepší, migrant si ji může zapamatovat jako svůj nový posun.

Po dokončení tažení všech migrantů směrem k leaderovi vznikne nová populace a proběhne nová iterace s novým leaderem.

SOMA tak kombinuje deterministický pohyb za nejlepším jedincem s prvky náhodnosti. Při rovnosti fitness mohou hrát roli náhodné faktory, v některých variantách se pořadí migrace míchá, ale existuje spousta způsobů jak řešit tuto situaci. Výše zmíněné parametry *Step* a *PathLength* ovlivňují granularitu a rozsah hledání. V praxi to znamená, že menší kroky a větší dráha znamenají důkladnější prohledání okolí leadera, zatímco větší kroky a kratší dráha znamenají rychlejší, ale hrubší průzkum. SOMA nepotřebuje operátory křížení ani mutace v klasickém smyslu, protože evoluční aspekt spočívá v migraci za nejlepším řešením (exploatace) a ponechání ostatních jedinců, kteří mohou případně přeskočit přes optimum a hledat dále (explorace, pokud  $PathLength > 1$  umožní jít až za leadera).

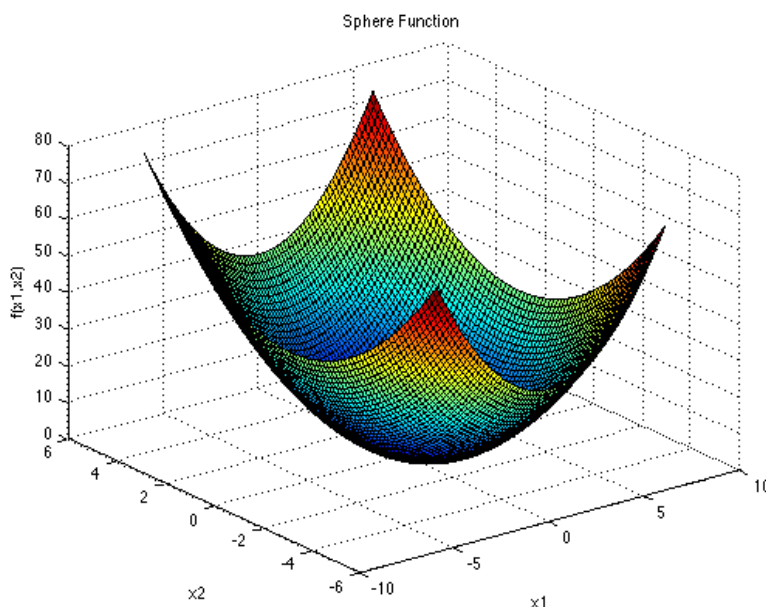
Bylo prokázáno, že SOMA je schopna nalézt kvalitní řešení různých optimalizačních problémů a konkuruje ostatním EA zejména v oblastech, jako je optimalizace řízení chaotických systémů, neuronových sítí či návrh filtrů [10]. Její výhoda tkví v jednoduchosti a v tom, že využívá nejlepší jedince přímo k vedení ostatních, čímž se poněkud liší od genetické metafor (kde dobří jedinci dávají potomky, ale přímo nevedou populaci). SOMA patří mezi algoritmy vhodné pro globální optimalizaci spojitých i nespojitých funkcí; její nastavení je intuitivní a nepotřebuje generátory náhodných mutací. Na druhou stranu při špatném nastavení může dojít k příliš rychlé ztrátě diverzity, tedy že všichni jdou za leaderem úzce nebo naopak k pomalému sbíhání. Výkon SOMA je srovnáván s PSO a DE a v literatuře existují i hybridní varianty (např. kombinace SOMA + DE).

## 1.4 Metodika testování optimalizačních algoritmů

Pro objektivní zhodnocení výkonnosti evolučních algoritmů a dalších metaheuristik je zapotřebí jednotná metodika testování. EA jsou stochastické a jejich výkon se může lišit běh od běhu; navíc různé algoritmy mohou vynikat na různých úlohách. Proto se v oblasti evoluční optimalizace klade důraz na experimentální porovnávání na celé sadě standardních testovacích úloh, doplněné statistickou analýzou výsledků. Základní pilíře metodiky testování jsou: volba vhodných benchmarkových funkcí, definice hodnotících kritérií, provedení dostatečného počtu nezávislých běhů, použití statistických testů k vyhodnocení rozdílů, a nakonec přehledná vizualizace a interpretace výsledků [4][27].

**Benchmarkové funkce:** v literatuře existují standardizované sady testovacích funkcí pro hodnocení globálních optimalizačních algoritmů [28][29]. Tyto funkce představují abstraktní optimalizační úlohy s různou obtížností, protože se mohou lišit v počtu dimenzí, tvaru a počtu extrémů, drsnosti povrchu apod. Mezi klasické benchmark funkce patří např.:

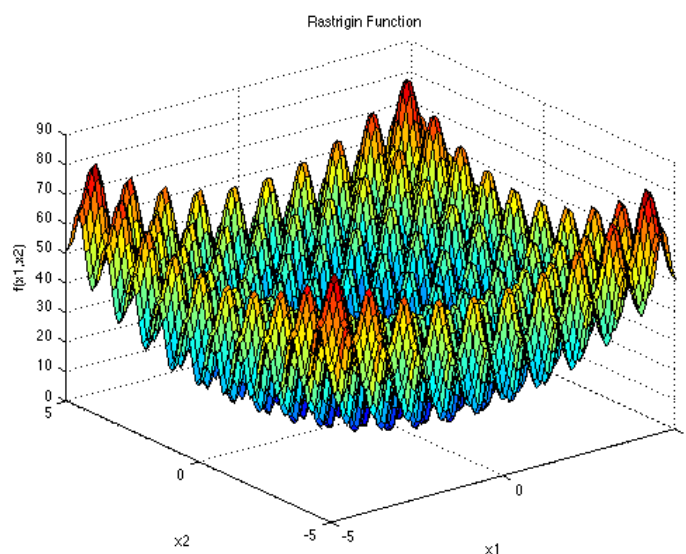
- **Sphere (kulová) funkce:**  $f(x) = \sum_{i=1}^d x_i^2$  je jednoduchá konvexní paraboloidální funkce s jediným globálním minimem v počátku  $(0, \dots, 0)$ , kde  $f_{min} = 0$ . Je unimodální a slouží jako základní test konvergence (snadná úloha).



Obrázek 4 3D povrch funkce Sphere

(zdroj: Surjanovic, 2025)

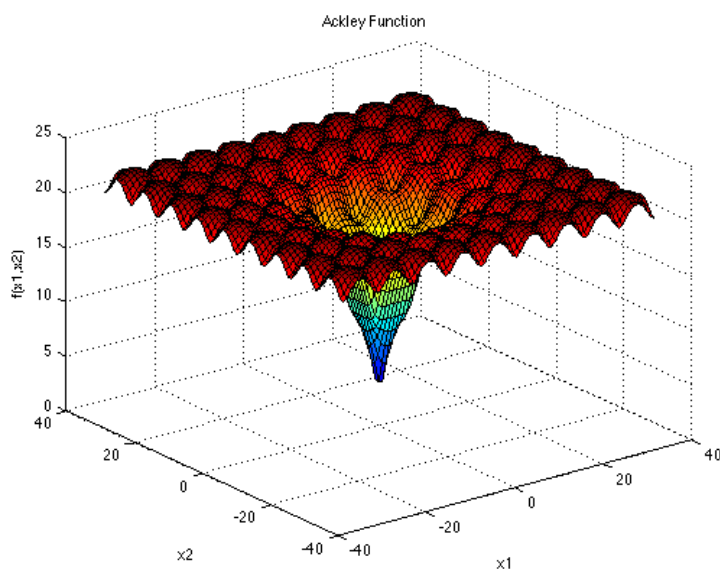
- **Rastriginova funkce:**  $f(x) = 10d + \sum_{i=1}^d [x_i^2 - 10\cos(2\pi x_i)]$ . Má globální minimum 0 v  $(0, \dots, 0)$ , ale obsahuje velký počet periodicky rozmístěných lokálních minim (díky členu  $\cos(2\pi x_i)$ ). Je silně multimodální; testuje schopnost algoritmu unikát mnoha klamným minimům.



Obrázek 5 3D povrch funkce Rastrigin

(zdroj: Surjanovic, 2025)

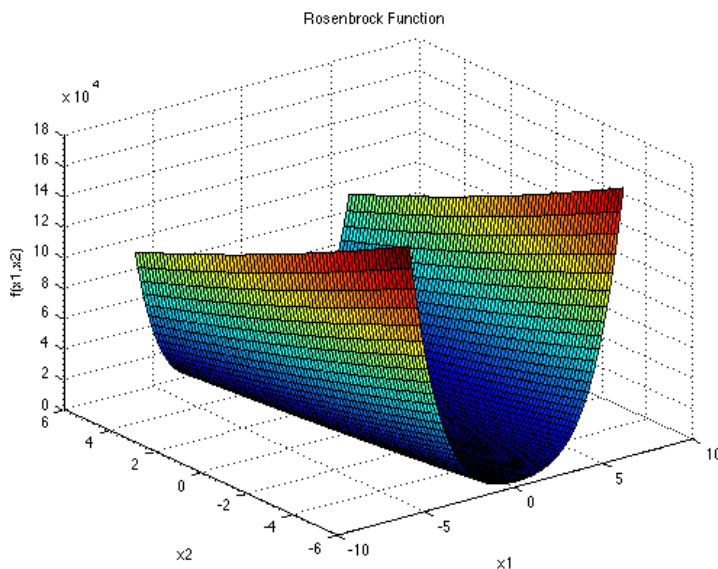
- **Ackleyho funkce:**  $f(x) = -20 \exp \left( -0.2 \sqrt{\frac{1}{d} \sum_{i=1}^d x_i^2} \right) - \exp \left( \frac{1}{d} \sum_{i=1}^d \cos(2\pi x_i) \right) + 20 + e$ . Má tvar plošiny s mnoha oscilacemi a jedním hlubokým kráterem v okolí optima 0 v  $(0, \dots, 0)$  (minimum 0). Kombinuje plošné oblasti a oscilační složku, což prověřuje rovnoměrné prohledávání.



Obrázek 6 3D povrch funkce Ackley

(zdroj: Surjanovic, 2025)

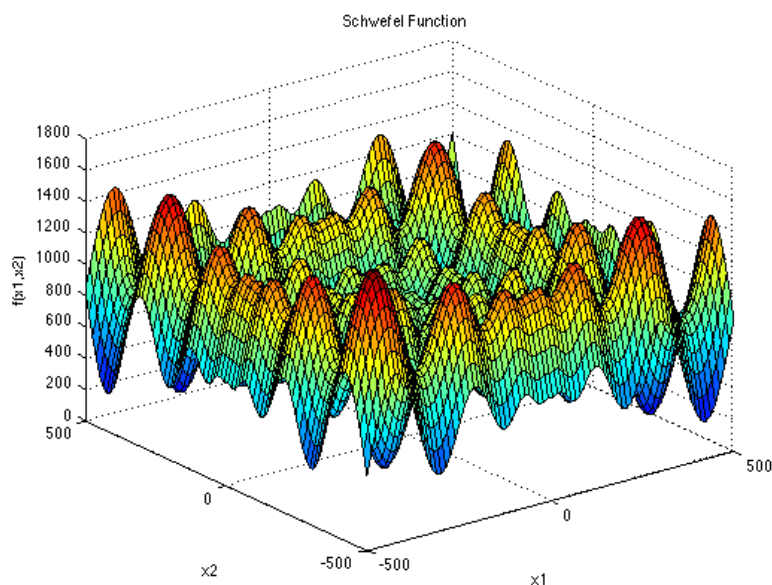
- **Rosenbrockova funkce (Banán):**  $f(x) = \sum_{i=1}^{d-1} [100(x_{i+1} - x_i^2)^2 + (x_i - 1)^2]$ . Pro  $d=2$  tvoří úzké banánovité údolí s globálním minimem 0 v (1,1). Je unimodální, ale velmi špatně podmíněná (úzká, zdlouhavě dosažitelná oblast optima), což testuje schopnost algoritmu jemně doladit řešení.



Obrázek 7 3D povrch funkce Rosenbrock

(zdroj: Surjanovic, 2025)

- **Schwefelova funkce:**  $f(x) = 418.9829d - \sum_{i=1}^d x_i \sin(\sqrt{|x_i|})$ . Je známá svým množstvím lokálních minim a tím, že globální minimum  $f_{min} = 0$  leží neintuitivně v bodě  $x_i \approx 420.9687$  pro všechna  $i$  (nikoli v nule). Představuje velmi obtížnou, multimodální úlohu, která nutí algoritmus prohledávat i odlehlé oblasti prostoru (typicky definována na -500; 500).



Obrázek 8 3D povrch funkce Schwefel

*(zdroj: Surjanovic, 2025)*

V experimentech se obvykle volí větší počet takových funkcí, aby se pokryly různé aspekty obtížnosti: od snadných (Sphere) po těžké (Schwefel), od nízkodimenzionálních po vícerozměrné (často se testuje v dimenzi 10, 30, 50), a zahrnují se jak separabilní funkce, tedy ty, které lze rozdělit na součet jednorozměrných, např. Rastrigin, tak neseparabilní, např. Rosenbrock. Pro algoritmus je žádoucí vykazovat stabilně dobrý výkon napříč všemi těmito funkcemi. V praxi se sestavují i specializované testovací báze, například pro multimodální funkce s proměnnou strukturou. Ty se nazývají CEC Competition Benchmarks. Na obrázcích výše jsou vizualizovány funkce Sphere, Rastrigin, Ackley, Rosenbrock a Schwefel v dvourozměrném prostoru ( $d = 2$ ). Lze vidět rozdíly v charakteru: Sphere má hladký konvexní tvar směřující do jediného minima, Rastrigin vykazuje mnohé periodické “kopečky“, kterým se odborně říká lokální minima, na pozadí kvadratického trendu, Ackley má plochou oblast s mírnými hrboly a poté strmý centrální kráter, Rosenbrock tvoří úzké údolí směřující k minimu a Schwefel má komplexní členitou krajinu s mnoha lokálními extrémy. Právě tyto rozdílné topografie testují schopnost algoritmů. Některé algoritmy si snadno poradí s konvexním tvarem, ale mohou selhat na funkcích s množstvím lokálních minim, a naopak.

## 1.5 Hodnotící kritéria a postup experimentu

Aby bylo možné kvantitativně porovnat algoritmy, definují se metriky, které se v experimentech sledují. Typicky se zaměřujeme na:

- **Nejlepší nalezená hodnota fitness:** po dokončení běhu algoritmu zaznamenáme nejlepší dosaženou fitness. Tuto hodnotu bereme jako výsledek daného běhu. Porovnáním průměrné nejlepší hodnoty mezi algoritmy zjistíme, který dosahuje lepší kvality řešení.
- **Rychlost konvergence:** sleduje se, po kolika generacích či hodnoceních se algoritmus dostane na určitou úroveň fitness. Někdy se měří např. počet hodnocení funkce potřebných k dosažení blízkosti optima. To ukazuje výpočetní efektivitu. Může se také vykreslit konvergenční křivka, tedy graf nejlepší fitness v závislosti na čase, a z ní posoudit rychlost nástupu k dobrému řešení.
- **Spolehlivost/robustnost:** protože algoritmus je náhodný, zajímá nás rozptyl výsledků. Měří se například směrodatná odchylka nejlepší nalezené fitness přes 30 nezávislých běhů. Nízký rozptyl značí, že algoritmus dává konzistentní výsledky, zatímco vysoký rozptyl může znamenat občasné uvíznutí v lokálním optimu. Dále lze sledovat procentuální úspěšnost (kolikrát z 30 běhů algoritmus skutečně dosáhl globálního optimum či cílovou hodnotu).
- **Výpočetní náročnost:** v akademických testech se většinou posuzuje podle počtu výpočtů fitness, protože ty dominují časové složce. Pokud ale jde o reálnou aplikaci, hodnotí se i reálný čas běhu nebo složitost implementace.

Standardní postup je spustit každý porovnávaný algoritmus mnohokrát nezávisle (obvykle 30 nebo 50 běhů) na každé benchmark funkci a zaznamenat uvedené ukazatele. Tak se získá statistický vzorek výsledků pro každý algoritmus na každé funkci.

**Statistická interpretace výsledků:** výsledky z experimentů se zpracují statisticky, aby bylo možné činit závěry s určitou mírou jistoty. Nejprve se typicky spočítá průměrná nejlepší hodnota a směrodatná odchylka pro každý algoritmus a testovací funkci. Tyto hodnoty se často uvádějí v tabulkách. Nicméně aby bylo možné tvrdit, že jeden algoritmus si vedl významně lépe než druhý, je potřeba provést statistický test významnosti. V oblasti evolučních algoritmů se často používají neparametrické testy, protože rozložení chyb nemusí být normální. Oblíbený je Wilcoxonův párový test nebo Friedmanův test s následným

Holmovým post-hoc testem [27], popřípadě t-test pokud jsou data normalizována. Tyto testy porovnají rozdělení výsledků dvou (či více) algoritmů a stanoví, zda rozdíl v jejich výkonech je statisticky významný (např. na hladině významnosti  $\alpha = 0.05$ ). Pokud ano, můžeme říci, že jeden algoritmus překonává druhý pro danou úlohu s danou pravděpodobností. Pokud ne, rozdíl může být přičten náhodné variabilitě.

Kromě testů se někdy používají také grafické nástroje pro analýzu, například krabicové grafy pro rozložení dosažených fitness hodnot, histogramy počtu hodnocení do konvergence, nebo tzv. výkonnostní profily, které ukazují, v kolika procentech případů daný algoritmus dosáhl určité kvality řešení [30]. Tyto vizualizace poskytují intuitivní přehled o spolehlivosti a efektivitě algoritmu.

### **Vizualizace a prezentace výsledků**

Pro lepší pochopení chování algoritmů je vhodné doplnit číselné ukazatele vizuálními prostředky. Běžně se vykreslují již zmíněné konvergenční křivky, tj. grafy vývoje nejlepší (případně průměrné) fitness v populaci v čase. Tyto grafy ukazují, jak rychle jednotlivé algoritmy nalézají dobrá řešení a zda v průběhu času jejich výkon stagnuje, nebo se dále zlepšuje. Dále se používají trajektorie v prostoru (zejména u 2D funkcí). U PSO lze například vykreslit pohyb částic rojem v čase, u GA pohyb celé populace. Pro benchmark funkce se také často uvádí vizuální srovnání polohy optima a nalezeného řešení atd.. V neposlední řadě se výsledky shrnují do přehledných tabulek, seřazené například podle výkonu. V akademických pracích bývá zvykem označit tučně nejlepší dosažené výsledky na každé funkci a hvězdičkou vyznačit, že rozdíl je statisticky signifikantní oproti druhému nejlepšímu.

Je dobré pochopit silné a slabé stránky algoritmů. Zjistíme-li například, že určitý EA funguje skvěle na hladkých konvexních úlohách, ale selhává na multimodálních, můžeme se pokusit ho vylepšit (třeba přidáním mechanismu pro udržení diverzity). Metodické experimenty tak vedou k informovanému výběru algoritmu pro danou reálnou úlohu a k postupnému zlepšování algoritmů jako takových.

## 2 SROVNÁNÍ A VÝBĚR GRAFICKÉHO ENGINU

Grafické enginy představují nástroj pro tvorbu interaktivních vizualizací, simulací, aplikací a her. Jde o softwarové prostředí, které integruje různé funkce, jako je rendering grafiky, fyzikální simulace, skriptování a správa uživatelských vstupů. V kontextu vizualizace evolučních algoritmů hrají svou roli v tom, že umožňují nejen zobrazit proces optimalizace v reálném čase, ale i přidat interaktivní prvky, které uživatelům usnadňují analýzu chování algoritmů.

Grafické enginy pracují na principu zpracování grafických dat prostřednictvím renderovacích pipeline, které převádějí 3D objekty na 2D obraz zobrazený na monitoru. Moderní enginy poskytují podporu pro různé vizuální efekty, stínování, animace a simulace fyzikálních jevů.

### 2.1 Význam grafických enginů pro vizualizaci evolučních algoritmů

Grafické enginy umožňují intuitivní, dynamické a srozumitelné zobrazení komplexních výpočetních procesů. Aby mohl uživatel co nejlépe pochopit chování algoritmů, sledování jejich výkonu a analýzu vývoje populace jedinců v optimalizačním prostoru, je výborná vizuální reprezentace. Díky tomu může uživatel například vidět trajektorii jedinců v reálném čase, včetně jejich sblížování, rozptylu nebo uváznutí v lokálních extrémech. Grafické enginy nám tuto možnost poskytují, a díky 3D pohledu je možné lépe analyzovat složité účelové funkce. Řezem dané účelové funkce pak můžeme sledovat vývoj i ve dvoudimenzionálním zobrazení. Pro lepší vizuální zážitek podporuje mnoho enginů také shadery, díky kterým můžeme pracovat s barevnými mapami a různými efekty pro znázornění fitness krajiny, kde různé barvy mohou ukazovat například výškové rozdíly účelové funkce. Mimo vzdělávání je vizualizace EA vhodná také pro výzkum a experimentální testování. Umožňuje intuitivní analýzu složitých problémů a usnadňuje hledání efektivnějších variant algoritmů prostřednictvím pozorování jejich chování v různých podmínkách.

### 2.2 Vlastnosti grafických enginů

Při výběru grafického enginu je potřeba zaměřit se na vlastnosti, které přímo ovlivňují kvalitu, plynulost a interaktivitu simulací. Pro potřeby této diplomové práce jsou důležité následující vlastnosti:

- **3D rendering a vizualizace účelové funkce:** schopnost kvalitního 3D renderingu pro zobrazení prostoru účelové funkce a trajektorií pohybu jedinců. Engine musí umožnit realistické vykreslování povrchů a světelných efektů, které zpřehlední topologii účelové funkce. Důležitá je podpora efektivního využití grafického procesoru (GPU), optimalizační techniky, jako je Level of Detail (LOD), a schopnost pracovat s velkými datovými sadami.
- **Animace a vizualizace pohybu:** engine by měl podporovat plynulé animace pohybu populací v reálném čase. Důležitá je schopnost vizualizovat trajektorie jedinců, rychlost jejich pohybu a interakce mezi nimi.
- **Interaktivní uživatelské rozhraní:** možnost vytvářet ovládací panely a rozhraní pro interaktivní změnu parametrů algoritmů (např. velikost populace, míra mutace) je dalším aspektem, který by měl engine podporovat. Díky těmto vstupům bude uživatel moci reagovat na různé scénáře a interagovat s jedinci.
- **Podpora skriptování a implementace algoritmů:** grafický engine musí umožňovat integraci vlastních algoritmů, zejména skriptování v programovacím jazyce vhodném pro tyto účely. To umožní propojení vizualizace s implementací evolučních algoritmů.
- **Efektivní správa objektů a výkon:** při vizualizaci stovek až tisíců jedinců je potřeba, aby engine efektivně spravoval objekty, minimalizoval nároky na hardware a podporoval optimalizační techniky, jako je level-of-detail rendering nebo object pooling.
- **Multiplatformní podpora:** schopnost exportovat aplikaci na různé platformy (PC, web, VR) rozšiřuje možnosti prezentace a sdílení výsledků.
- **Dostupnost dokumentace a podpora komunity:** bohatá dokumentace a aktivní komunita pro rychlé řešení problémů a snadnější vývoj.
- **Cena a licenční podmínky:** pro akademické projekty je ideální engine s výhodnými licenčními podmínkami nebo bezplatnou verzí pro nekomerční použití.

## 2.3 Vývoj grafických enginů a jejich současné využití

Vývoj grafických enginů prošel milníky, které formovaly jejich současnou podobu. První grafické enginy vznikly v 80. a 90. letech spolu s rozvojem počítačových her. Tyto rané enginy, například id Tech 1, který poháněl hru *DOOM*, kladly důraz především na 2D vykreslování a jednoduché 3D efekty pomocí spritů. Později, na konci 90. let, se objevily první skutečné 3D enginy, jako byl Unreal Engine, který zavedl pokročilé renderovací techniky a modifikovatelnou architekturu. Tento engine položil základy modernímu přístupu k vývoji grafických enginů, včetně podpory pro shadery, světla a částicové efekty. S příchodem 21. století se rozvoj zrychlil díky rostoucímu výkonu grafických karet. Unity, vydaný v roce 2005, se stal populárním díky své přístupnosti, podpoře multiplatformního exportu a snadné integraci skriptů v jazyce C#. Oproti tomu Unreal Engine 4, uvedený v roce 2014, přinesl špičkovou grafiku a realistické efekty díky pokročilému systému osvětlení a fyziky.

V současnosti jsou grafické enginy používány daleko za hranicemi herního průmyslu. V oblasti vědeckých simulací se používají k vizualizaci složitých fyzikálních a biologických procesů. V průmyslových simulacích pomáhají trénovat AI modely a řídit automatizované systémy. V oblasti výuky a výzkumu umožňují vytváření interaktivních modelů pro prezentaci složitých konceptů, jako jsou evoluční algoritmy.

## 2.4 Přehled vybraných grafických enginů

Tato část poskytuje podrobný přehled vybraných grafických enginů s důrazem na jejich vlastnosti, historii, aktuální funkce a příklady jejich využití ve světě. Informace jsou zaměřeny na aspekty relevantní pro vizualizaci evolučních algoritmů.

### 2.4.1 Unity

Unity je jedním z nejpopulárnějších a nejrozšířenějších grafických enginů, který se stal standardem nejen pro vývoj her, ale i pro simulace, výukové aplikace a vědecký výzkum. Tento engine, postavený na skriptování v jazyce C#, je oblíbený pro svou flexibilitu, rozsáhlé možnosti vizualizace a multiplatformní kompatibilitu. Díky tomu je možné jej snadno integrovat do projektů zaměřených na vizualizaci evolučních algoritmů kde vyžadujeme rychlou iteraci a interaktivní prostředí.

Unity se neustále vyvíjí a jeho nejnovější verze obsahují pokročilé technologie, jako je podpora ray tracingu pro realistické nasvícení scén, Universal Render Pipeline (URP)

umožňující optimalizované vykreslování napříč platformami, a nástroje pro umělou inteligenci, například AI navigace a simulace davů. Součástí Unity je také Asset Store, platforma, která poskytuje tisíce hotových modelů, skriptů a nástrojů, které urychlují vývoj složitých aplikací.

Unity našlo uplatnění i jinde než v herním průmyslu. Například v oblasti medicíny se používá k tréninku lékařů prostřednictvím VR simulací. Z herního prostředí se Unity proslavilo tituly jako “Hollow Knight“, “Monument Valley“ a “Among Us“, které ukazují jeho schopnost podporovat jak 2D, tak 3D projekty.

Unity má také velice širokou komunitu, podporu a rozsáhlou dokumentaci, což usnadňuje vývoj i složitých aplikací. I přes vyšší nároky na optimalizaci u komplexnějších simulací je Unity díky svým vlastnostem ideálním nástrojem pro tvorbu interaktivních vizualizací evolučních algoritmů [31][32].

#### 2.4.2 Unreal Engine

Unreal Engine je považován za průmyslový standard pro tvorbu vysoce kvalitních 3D vizualizací a her. Tento engine se proslavil svým fotorealistickým renderingem a pokročilými simulacemi. Skriptování je možné pomocí vizuálního systému Blueprints nebo jazyka C++, což umožňuje široké spektrum využití od her až po vědecké simulace. Navíc se také neustále vyvíjí a jeho nejnovější verze přináší technologii Lumen pro dynamické globální osvětlení a Nanite, která umožňuje renderovat obrovské množství detailů bez ztráty výkonu. Díky těmto technologiím je Unreal Engine ideální pro realistické zobrazení složitých účelových funkcí a plynulou vizualizaci pohybu populací v rámci evolučních algoritmů.

Unreal Engine, stejně jako Unity, našel uplatnění nejen v herním průmyslu, ve kterém poháněl tituly jako “Fortnite“, “Gears of War“ nebo “Final Fantasy VII Remake“, ale také ve filmovém průmyslu, například při tvorbě virtuálních scén pro seriál “The Mandalorian“. Kromě toho se využívá v automobilovém průmyslu k vytváření simulací a virtuálních showroomů. Další společný prvek s Unity je velice rozsáhlá komunita vývojářů. Přestože jeho křivka učení je strmější a hardwarové nároky vyšší, jeho vizuální kvality, rozsáhlé možnosti skriptování a podpora VR ho činí vhodným nástrojem pro vývoj vizualizací evolučních algoritmů s realistickým zobrazením a vysokou interaktivitou [33].

### 2.4.3 Godot Engine

Godot Engine je open-source grafický engine, který získává stále větší popularitu zejména mezi nezávislými vývojáři a vzdělávacími institucemi. Vyniká svou jednoduchostí a flexibilitou při tvorbě 2D i 3D aplikací. Skriptování probíhá v jazyce GDScript, který svou strukturou připomíná Python a je snadno naučitelný.

Godot Engine je známý svou lehkostí a efektivitou, a proto je skvělou volbou pro menší projekty nebo prototypování algoritmů. Jeho nejnovější verze přináší podporu Vulkan API, která výrazně zvyšuje výkon renderingu. Vizualizační možnosti Godotu jsou dostatečně robustní pro potřeby simulace evolučních algoritmů, zejména díky snadné tvorbě interaktivních uživatelských rozhraní a animací.

I když se Godot zatím nemůže měřit s vizuální kvalitou Unity nebo Unreal Engine, jeho hlavní výhodou je, že je zcela zdarma, bez licenčních poplatků a s otevřeným zdrojovým kódem. To umožňuje plnou kontrolu nad projektem a snadnou integraci vlastních modulů. Godot našel uplatnění zejména v indie herním průmyslu a stvořil hry jako “Cruelty Squad” a “Lumencraft”. Díky svým nízkým nárokům na hardware a otevřenosti se i Godot stal kandidátem, zejména v situacích, kdy je kladen důraz na rychlost vývoje, možnost experimentování a snadnou rozšiřitelnost [34].

### 2.4.4 CryEngine

CryEngine, vyvinutý německou společností Crytek, je známý pro svůj fotorealistický rendering a schopnost simulovat rozsáhlá přírodní prostředí. Tento engine se poprvé proslavil hrou “Far Cry“, která ukázala jeho výjimečné grafické schopnosti. V dalších verzích byl CryEngine dále zdokonalen a stal se silným nástrojem nejen pro herní vývoj, ale také pro vizualizace a simulace v průmyslu.

Technologie CryEnginu se soustředí na vysokou vizuální kvalitu a podporu realistických fyzikálních simulací. Jeho renderingový systém nabízí pokročilé funkce, jako je ray tracing, volumetrická mlha, procedurální generování terénu a dynamické počasí. Tyto prvky jsou vhodné pro tvorbu realistických vizualizací evolučních algoritmů, zejména pokud je nutné simulovat velká prostředí nebo složité povrchy účelových funkcí.

CryEngine je využíván nejen v herním průmyslu, díky kterým vznikly tituly jako “Crysis“, “Ryse: Son of Rome“ a “Hunt: Showdown“, ale také v oblastech mimo herní sféru. Průmyslové simulace, architektonické vizualizace a vojenské výcvikové programy patří

mezi oblasti kde se engine díky své vizuální věrnosti a fyzikálním simulacím prosadil. Například v automobilovém průmyslu byl využit pro tvorbu interaktivních showroomů a vizualizaci prototypů.

Skriptování probíhá primárně pomocí jazyka C++, což poskytuje vývojářům vysokou úroveň kontroly nad algoritmy a optimalizací výpočetních procesů. Engine rovněž obsahuje vizuální skriptovací systém Flowgraph, který umožňuje rychlé prototypování bez nutnosti hlubokých znalostí programování. Přestože je CryEngine vysoce výkonný, jeho nevýhodou je vyšší znalost jeho prostředí a složitější rozhraní ve srovnání s enginy, jako jsou Unity nebo Godot. Přesto jeho vizuální kvalita, podpora velkých otevřených světů a schopnost simulace složitých procesů z něj činí silného kandidáta pro projekty, které vyžadují vizualizaci evolučních algoritmů ve velkých a komplexních prostředích. CryEngine vyniká svou schopností poskytovat špičkovou vizuální kvalitu, avšak vyžaduje výkonný hardware a pečlivou optimalizaci, protože výkon je silně závislý na konfiguraci scény a nastavení grafiky [35].

## 2.5 Výběr nejvhodnějšího grafického engine

Na základě provedené jednoduché komparativní analýzy je jako nejvhodnější engine pro vizualizaci evolučních algoritmů vybrán Unity. Tento výběr je podložen vyváženým poměrem snadné implementace algoritmů, vysokého výkonu, dostatečné vizuální kvality a široké podpory pro tvorbu interaktivních prvků. Unity navíc nabízí rozsáhlou komunitní podporu, bohatou dokumentaci a velký výběr nástrojů a pluginů, které zrychlují vývoj aplikace.

Unreal Engine by byl vhodnou alternativou zejména pro projekty, které kladou důraz na špičkovou vizuální kvalitu, avšak jeho vyšší složitost implementace a náročnější ovládání skriptovacích nástrojů ho činí méně efektivním pro potřeby této práce. Godot je ideální pro prototypování a výukové aplikace, ale jeho výkon je omezený při rozsáhlejších simulacích. CryEngine vyniká vizuálními možnostmi, ale složitá implementace algoritmů ho staví mimo hlavní výběr.

Volba Unity tedy nejlépe odpovídá cílům této diplomové práce, která se soustředí na efektivní vizualizaci evolučních algoritmů s důrazem na interaktivitu, výkon a snadnou rozšiřitelnost aplikace.

Tabulka 1 Srovnání enginů

| Engine    | Grafická kvalita | Snadnost skriptování     | Náročnost na hardware | Vhodnost pro EA vizualizaci |
|-----------|------------------|--------------------------|-----------------------|-----------------------------|
| Unity     | Vysoká (HDRP)    | Vysoká (C#)              | Střední               | Velmi Vysoká                |
| UE        | Velmi vysoká     | Střední (C++/Blueprints) | Vysoká                | Střední                     |
| Godot     | Střední          | Vysoká (GDScript)        | Nízká                 | Vysoká                      |
| CryEngine | Velmi vysoká     | Střední (C++/C#)         | Vysoká                | Sřední                      |

*(zdroj: vlastní)*

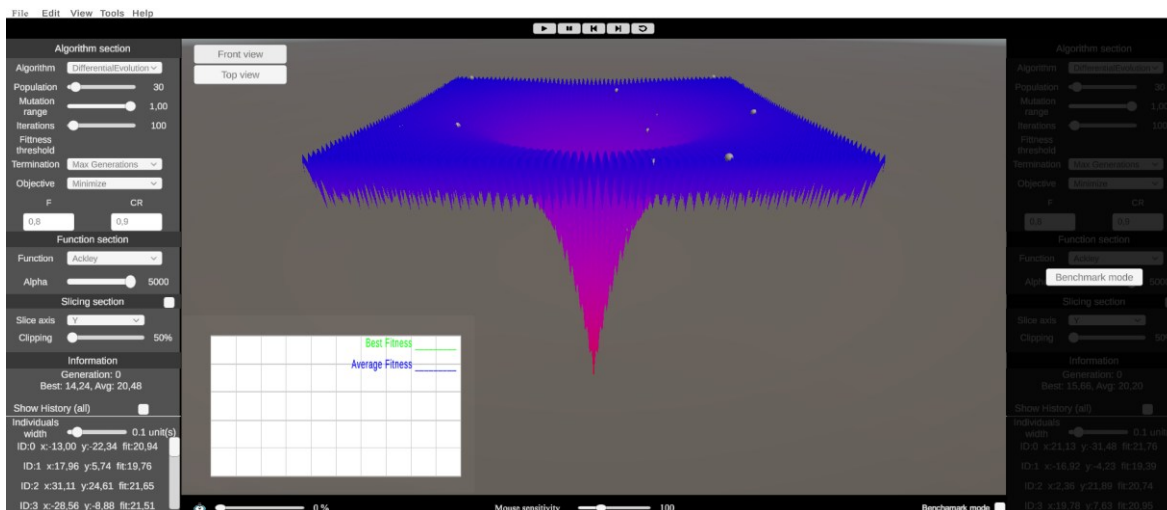
## II PRAKTICKÁ ČÁST

### 3 NÁVRH ŘEŠENÍ A IMPLEMENTAČNÍ ARCHITEKTURA

Cílem praktické části této diplomové práce bylo navrhnout a implementovat vlastní nástroj pro vizualizaci a testování evolučních algoritmů. Kladen byl důraz na přehlednost, interaktivitu a možnost sledování jednotlivých fází optimalizačního procesu v reálném čase. Vytvořený systém by měl uživatelům umožnit nejen spouštění simulací, ale také aktivní ovlivňování parametrů algoritmu, volbu testovacích funkcí, analýzu vývoje hodnot fitness funkce v průběhu času a porovnávání dvou různých algoritmů.

Jedním z hlavních záměrů bylo zpřístupnění evolučních algoritmů i uživatelům bez hlubších programátorských zkušeností. Z tohoto důvodu bylo rozhraní navrženo jako intuitivní a přehledné, s vizuálními prvky připomínajícími klasické desktopové aplikace. Uživatelské rozhraní využívá horní lištu s nabídkami, editační plochu pro vizualizaci a boční panely s ovládacími prvky. Tento přístup zajišťuje nejen komfort při práci, ale i snadnou orientaci v prostředí.

Implementace byla koncipována tak, aby bylo v budoucnu co nejjednodušší aplikaci rozšířit, například o další optimalizační algoritmy, nové typy vizualizací či interaktivní analytické nástroje. Architektura kódu a organizace jednotlivých tříd byly navrženy s důrazem na udržitelnost, přehlednost a oddělení odpovědností.

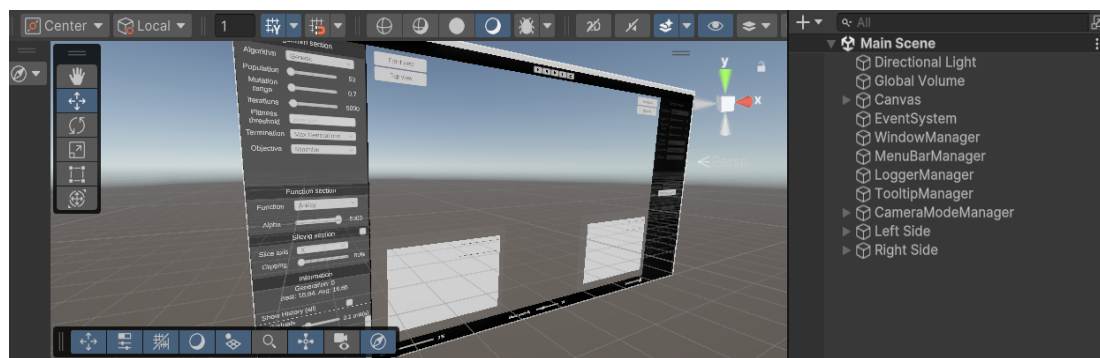


Obrázek 9 Uživatelské prostředí aplikace

(zdroj: vlastní)

### 3.1 Scénová hierarchie a rozložení komponent

Veškeré objekty ve scéně jsou uspořádány pod hlavní entitou Main Scene, která obsahuje základní systémové komponenty jako kamera, světelné zdroje a další. Vizuální rozhraní je sdruženo pod objektem Canvas a skripty pro logiku jsou přidruženy k prázdným GameObjectům.



Obrázek 10 Hierarchie komponent a hlavní layout

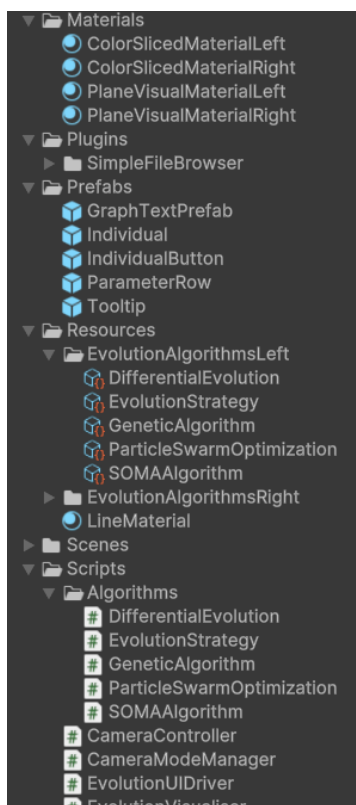
(zdroj: vlastní)

#### 3.1.1 Struktura projektu

Všechny zdrojové soubory a assety jsou uloženy ve složce Assets, která je rozdělena do několika podadresářů podle jejich účelu. Složka Fonts obsahuje fonty, složka Images ukládá ikony a ilustrace používané v uživatelském rozhraní, například tlačítka pro přehrávání a pozastavení simulace. V Materials se nachází materiály pro vykreslení funkčních sítí a řezných rovin (např. ColorSlicedMaterial pro levou a pravou funkci). Externí knihovny a pluginy, jako je SimpleFileBrowser [36] pro import/export JSON, jsou umístěny v adresáři Plugins. Ve složce Resources jsou organizovány dva podadresáře EvolutionAlgorithmsLeft a EvolutionAlgorithmsRight, které obsahují ScriptableObject assety s výchozími parametry evolučních algoritmů pro levou a pravou část benchmark módu. Díky tomu má každá strana své vlastní přednastavené nastavení bez nutnosti ručního kopírování nebo přepínání parametrů v runtime. Také se v této složce nachází materiál pro čaru vyznačující trajektorii. V adresáři Scenes se nachází jediná Unity scéna pojmenovaná MainScene.unity, která slouží jako vstupní bod celé aplikace a spojuje vizualizační vrstvu, ovládací prvky i logiku simulace. Adresář Scripts pak obsahuje veškeré C# skripty od jádra evolučního vizualizéru přes komponenty pro vykreslování a řezání mesh sítí až po řídicí třídy kamer, logování rozhraní pro uživatele.

Poslední složkou, která se v hierarchii nachází, je složka Prefabs, ve které nalezneme následující prefaby:

- Individual.prefab – jednoduchý Sphere GameObject se SphereColliderem a skriptem IndividualInfo, který reprezentuje jednoho jedince v populaci.
- SlicePlaneVisual.prefab – plátno (plane) se speciálním materiálem, které se používá k vizualizaci řezné roviny mesh sítě.
- ParameterRow.prefab – UI řádek složený z Label (TMP\_Text) a InputField (TMP\_InputField), umožňující editaci parametrů vybraného algoritmu.
- IndividualButton.prefab – tlačítko (Button) s textem (TMP\_Text), které se dynamicky vytváří pro každého jedince v seznamu a umožňuje rychlé zobrazení jeho linie (ShowLineage).
- Tooltip.prefab – pozadí s textem, na kterém se zobrazuje pomocný text funkčnosti daného prvku. Velikost je generována podle délky textu.



Obrázek 11 Ukázka struktury projektu

(zdroj: vlastní)

### 3.1.2 Vizualizační vrstva

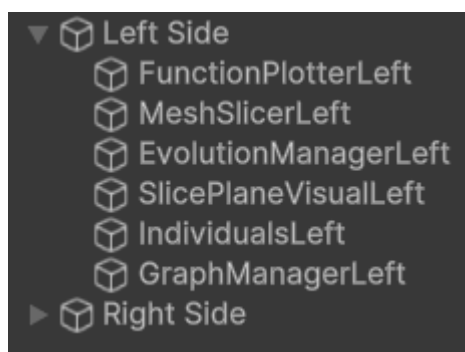
Vizualizační vrstva je rozdělena na dvě části (levá a pravá, kvůli benchmark módu) a slouží k vizualizaci funkce a řešení v prostoru. Tato část je reprezentována objekty:

- FunctionPlotter – spravuje vykreslení sítě v závislosti na parametrech jako hustota mřížky nebo měřítko.
- MeshSlicer – zajišťuje rovinný řez mesh-sítí podle zvolené osy.
- SlicePlaneVisual – vizuální reprezentace roviny řezu nad mesh sítí.
- Individuals – rodičovský objekt pro zobrazení jedinců v každé generaci.
- GraphManager – stará se o správu grafu

Vizualizace je navíc propojena s kamerou, která umožňuje pohyb a rotaci pohledu v prostoru a zajišťuje tak lepší orientaci při sledování vývoje populace.

### 3.1.3 Řízení evolučního algoritmu

Logika samotné simulace je soustředěna do objektu EvolutionManager, jehož hlavní skript EvolutionVisualiser zajišťuje koordinaci všech souvisejících částí. Dále propojuje ovládací prvky (Play, Pause, Reset, Step) a jejich reakce na vstupy, dropdown pro výběr algoritmu a parametry evoluce, jako velikost populace, míra mutace nebo maximální počet generací. Každý jedinec je reprezentován vlastním prefabem a vykreslován v prostoru podle svých parametrů. Vizualizace evolučního procesu je navržena tak, aby uživatel viděl jak vývoj v prostoru, tak i změny ve fitness hodnotách formou grafu.



Obrázek 12 Ukázka vizualizační a aplikační vrstvy

(zdroj: vlastní)

### 3.1.4 Kamerový systém

Pro zajištění pohodlné orientace a porovnávání výsledků je implementován kamerový systém tvořený dvěma hlavními komponentami: *CameraModeManager* a *CameraController*. Zatímco *CameraModeManager* se stará o přepínání mezi režimy „Single“ (jedna kamera) a „Split“ (rozdělená obrazovka), nebo například aktivuje či deaktivuje odpovídající kamery a UI panely (grafy, ovládací tlačítka) pro každou polovinu obrazovky, *CameraController* se stará hlavně o interaktivní ovládání pohledu uživatelem. Uživatel může otočit kamerou tažením myši, posouvat se klávesami WASD/QE, nebo třeba zoomovat kolečkem myši.

Tento dvoustupňový přístup odděluje “kdy” a “kterou” kameru zobrazit (*CameraModeManager*) od “jak” se s kamerou pracuje (*CameraController*). Díky tomu je ovládání přehledné, uživatel může kdykoli přepnout mezi režimy a zároveň si zachovat plnou kontrolu nad prostorem, ve kterém sleduje průběh evoluční simulace.

## 3.2 Uživatelské rozhraní

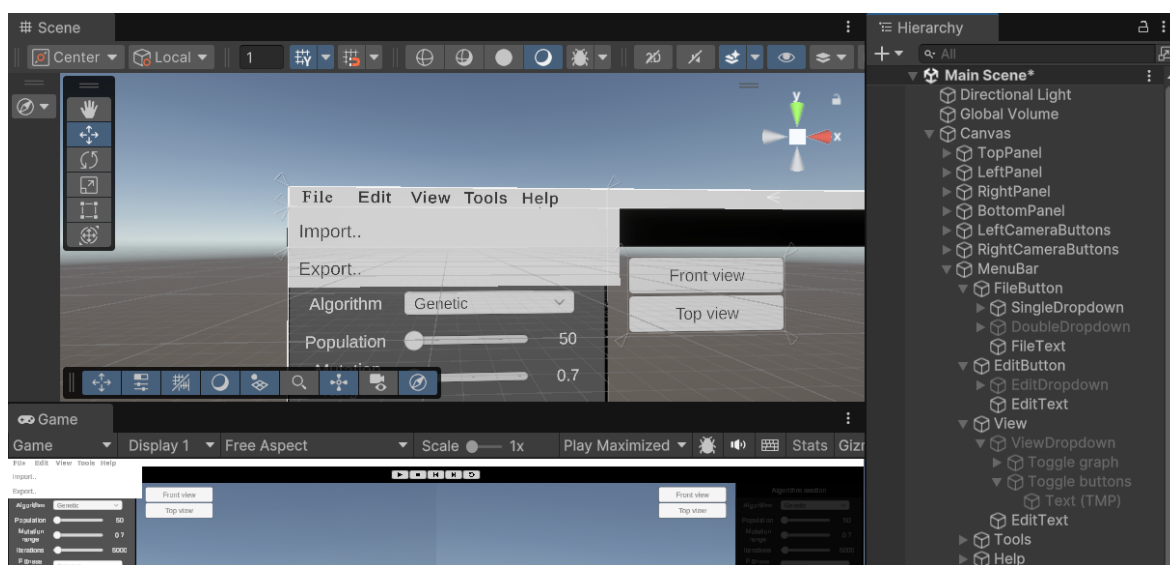
Návrh uživatelského rozhraní vychází z principů známých z běžných editačních nástrojů, aby se uživatel mohl v prostředí snadno orientovat bez nutnosti studovat ovládání. Rozhraní se skládá z několika logicky oddělených sekcí, které byly rozmístěny do samostatných panelů v rámci Canvas objektu. Tyto panely jsou uspořádány do oblastí horní lišty, horní pod-lišty, levého a pravého ovládacího sloupce, vizualizační plochy a spodního panelu se zprávami a doplňkovými nástroji. Všechna tlačítka jsou doplněna o tooltipy, které po najetí kurzorem vysvětlí jejich účel. To přispívá k intuitivnímu ovládání a snižuje riziko chybné manipulace, zejména u méně zkušených uživatelů.

### 3.2.1 Horní lišta (menu bar)

Horní lišta představuje ovládací prvek aplikace, který sdružuje veškeré základní funkce potřebné pro práci s projektem. Je rozdělena do několika hlavních nabídek:

- **File:** obsahuje funkce pro správu projektu, jako jsou import a export, popřípadě, pokud se uživatel nachází v benchmark módu, má možnost si vybrat na kterou stranu chce jaká data importovat.
- **Edit:** umožňuje resetování pohledu (Reset Camera), restartování celé scény (Restart Scene) a vyčištění konzole (Clear log).

- **View:** nabízí možnosti přepínání zobrazení jednotlivých částí rozhraní, tedy zobrazení/skrytí fitness grafu (Toggle Graphs) a zobrazení/skrytí tlačítek pro změnu kamery (Toggle Buttons).
- **Tools:** zahrnuje pokročilé nástroje jako Benchmark mode (spuštění více algoritmů najednou s oddělenými kamerami pro srovnání) a Simulation history (zobrazení historie dráhy všech jedinců v dané generaci).
- **Help:** poskytuje přístup k informacím o ovládání (Controls). Nachází se tam informace pro klávesové zkratky pro tlačítka uvedené výše + ovládání kamery, ovládání simulace a další.



Obrázek 13 Horní lišta s rozbaleným tlačítkem File

(zdroj: vlastní)

Horní lišta je navržena tak, aby zůstala pevně ukotvena v horní části obrazovky a byla dostupná bez ohledu na aktuální zobrazení nebo rozlišení obrazovky.

### 3.2.2 Horní pod-lišta

Bezprostředně pod hlavním menu barem se nachází horní podlišta, která obsahuje sadu pěti tlačítek [37] určených pro rychlé ovládání simulace. Tato tlačítka jsou vizuálně odlišena jednoduchými symboly (trojúhelník, dvojšipka, čtverec atd.) a jsou pevně zarovnána na střed horní části obrazovky.

Konkrétní tlačítka a jejich funkce:

1. ► **Play** – spustí evoluční simulaci (automatický přechod mezi generacemi).

2. **II Pause** – pozastaví simulaci v aktuálním stavu.
3. **⏮ Previous** – vrátí simulaci o jednu generaci zpět (při ručním krokování).
4. **⏭ Next** – přejde na další generaci (při ručním krokování).
5. **■ Stop/Reset** – ukončí běh simulace a obnoví výchozí populaci.



Obrázek 14 Tlačítka pro ovládání simulace

(zdroj: vlastní)

Tato podliška byla navržena tak, aby byla vždy viditelná a poskytovala rychlý přístup k nejčastěji používaným funkcím bez nutnosti rozklikávání hlavního menu. Tlačítka fungují nezávisle na výběru algoritmu nebo funkce a jsou aktivní během celé práce se simulací.

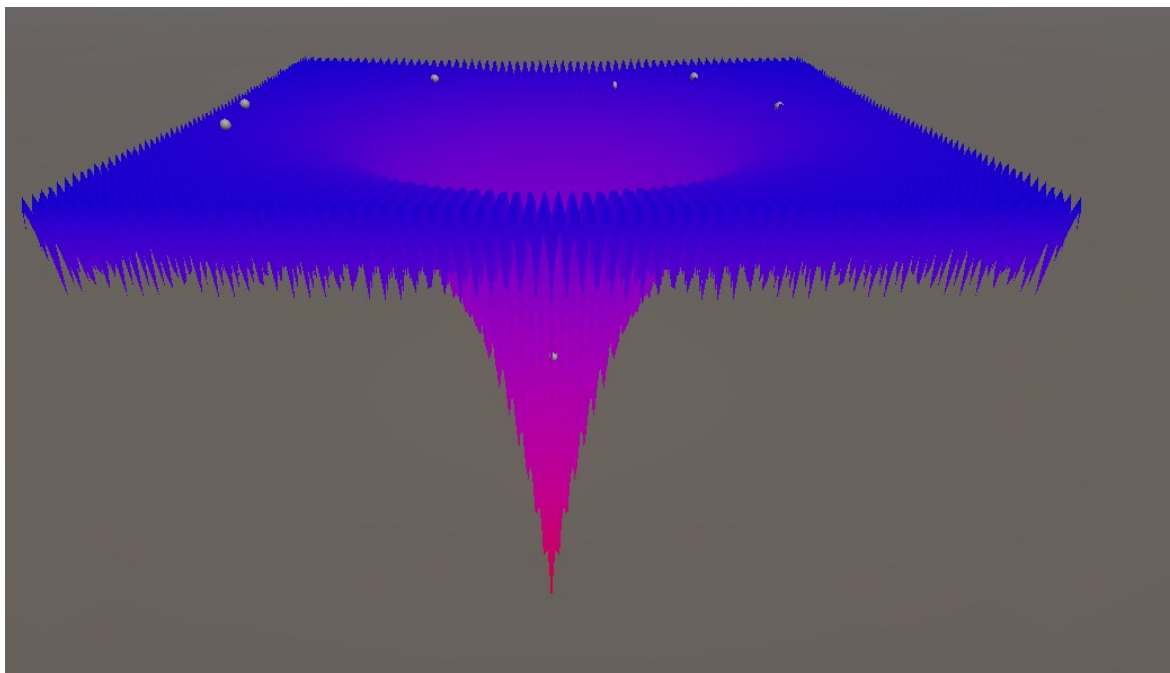
### 3.2.3 Levý a pravý ovládací panel

V těchto částech rozhraní se nachází podpanely s interaktivními prvky pro ovládání evoluční simulace. První panel obsahuje rozbalovací menu pro výběr algoritmu, posuvníky pro nastavení velikosti populace, míru mutace a počet iterací. Uživatel má také možnost si vybrat ukončovací podmínku a případně zadat hodnotu fitness, při které se má simulace ukončit. Pokud této hodnoty simulace nikdy nedosáhne, pokračuje se do maximálního počtu generací. Po výběru konkrétního algoritmu z rozbalovacího seznamu se pod parametry zmíněnými výše zobrazí také příslušné textové pole s popisem pro vybraný algoritmus (např. F a CR u diferenciální evoluce).

Dále se v rozhraní nachází panel s rozbalovacím menu pro výběr funkce. Vzhledem k tomu, že se jedinci někdy při složitějších funkcích “ztrácejí” v meshi, má uživatel možnost funkci zprůhlednit pomocí slideru, nacházejícím se přímo pod rozbalovacím menu. Sílu této funkcionality lze vidět na obrázcích 15 a 16. V prvním případě vidíme pouze zlomek jedinců z populace, kdežto ve druhém případě vidíme všechny (v parametrech nastavena velikost populace jako 30 jedinců v obou případech).

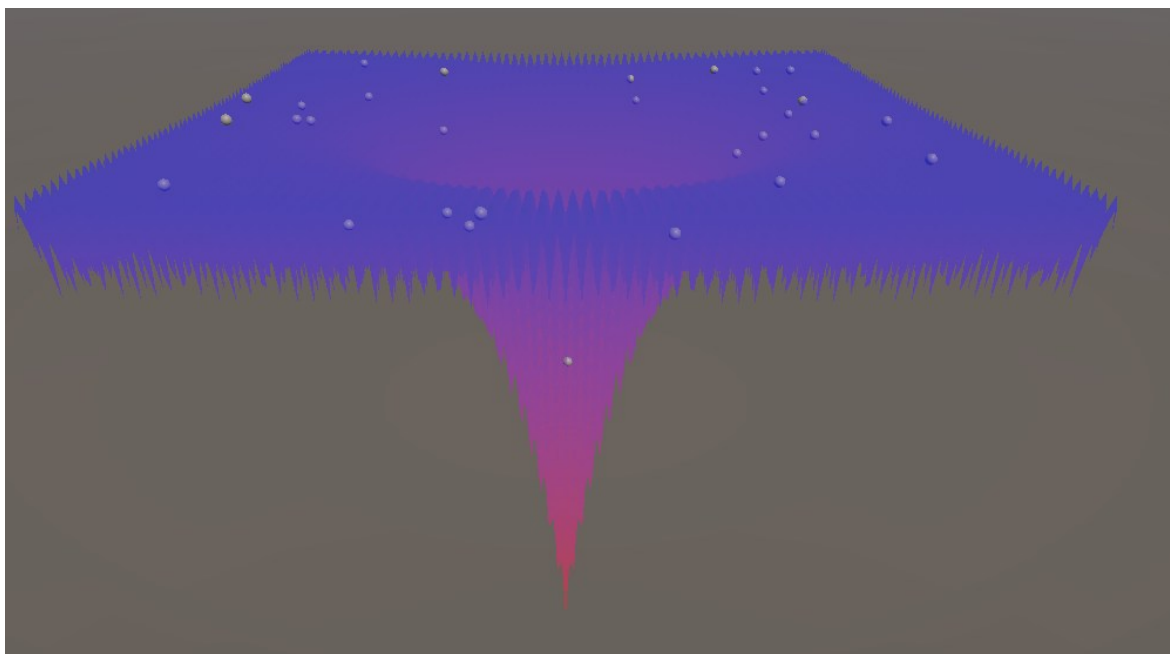
Při spuštění je pravý panel ztmaven a zneprístupněn, jelikož slouží k ovládání druhé evoluce v benchmark módu, který je defaultně vypnutý. Tento mód lze zapnout buďto

stisknutím klávesy B, nebo stisknutím tlačítka „Benchmark mode“. Oba panely jsou totožné, ale každý ovládá svoji polovinu obrazovky.



Obrázek 15 Ukázka funkce s  $\alpha = 1$

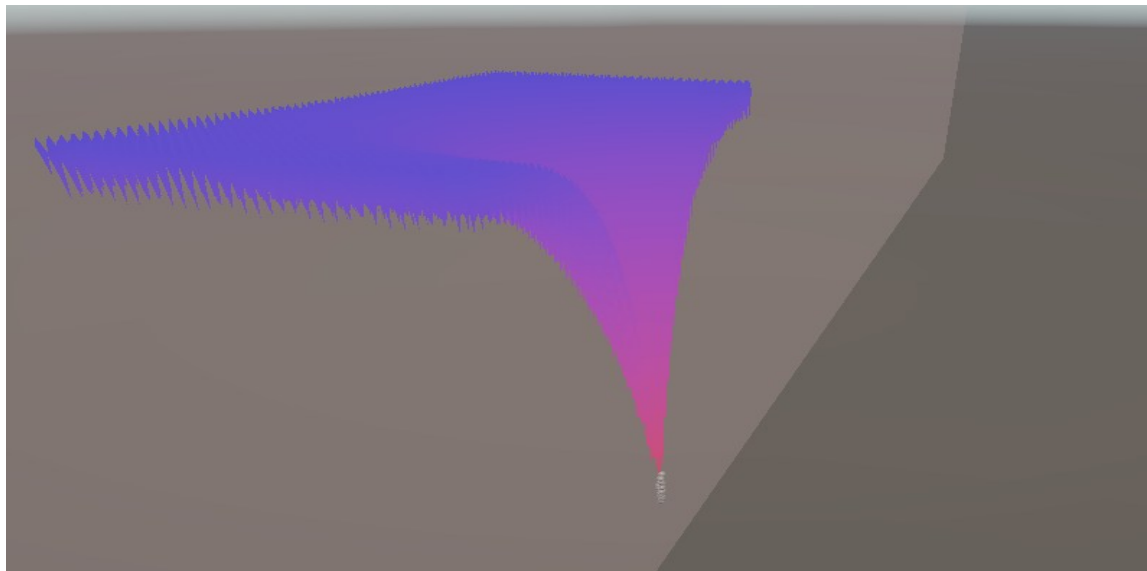
(zdroj: vlastní)



Obrázek 16 Ukázka funkce s  $\alpha = 0.5$

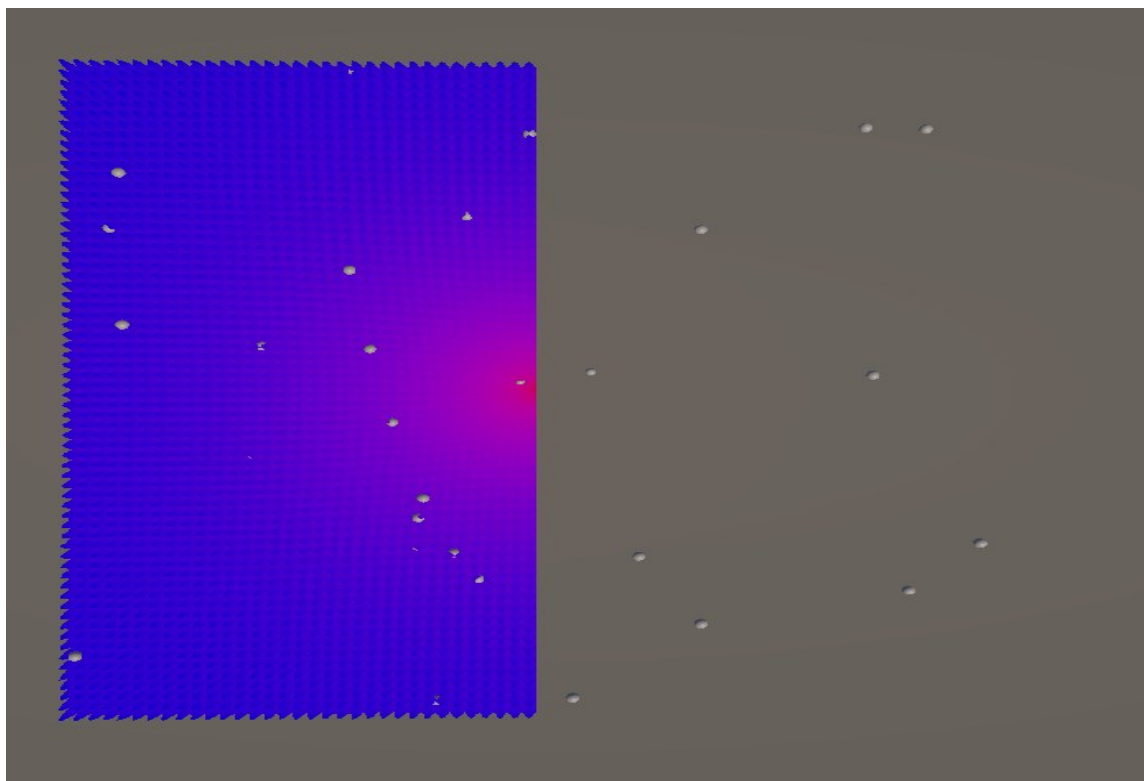
(zdroj: vlastní)

Pod panelem s výběrem funkce pak nalezneme sekci, která umožňuje aktivaci řezu vizualizovanou funkcí pomocí roviny. Uživatel zde může zvolit osu, na které má být rovina aplikována, a také nastavit procentuální průnik funkcí pomocí posuvníku.



Obrázek 17 Řez funkce na X-ose s viditelnou vizualizační rovinou

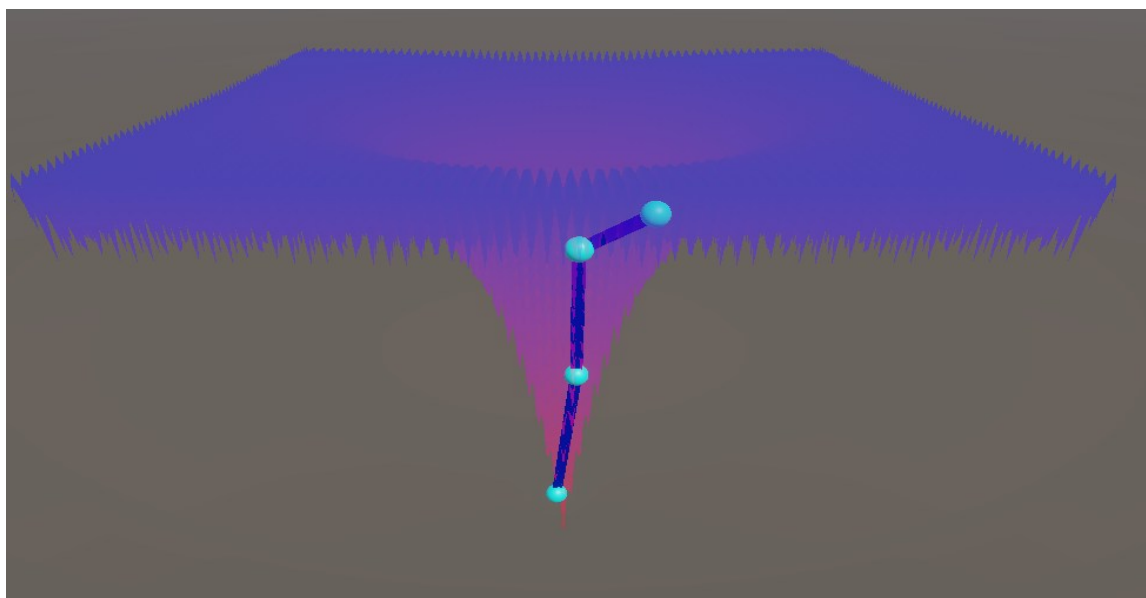
(zdroj: vlastní)



Obrázek 18 Řez funkce při pohledu zezhora

(zdroj: vlastní)

Ve spodní části těchto ovládacích panelů se nachází informace o aktuální generaci, zvoleném algoritmu, jeho statistik a jednotlivých jedincích dané generace - respektive jejich pozici a fitness hodnotu. Pro lepší přehlednost lze měnit velikost jedinců, a také lze na tyto jedince kliknout a vidět vizuálně jejich historii pohybu. Taktéž lze zobrazit historie všech jedinců najednou, odlišené náhodnými barvami.



Obrázek 19 Zobrazení historie pohybu daného jedince

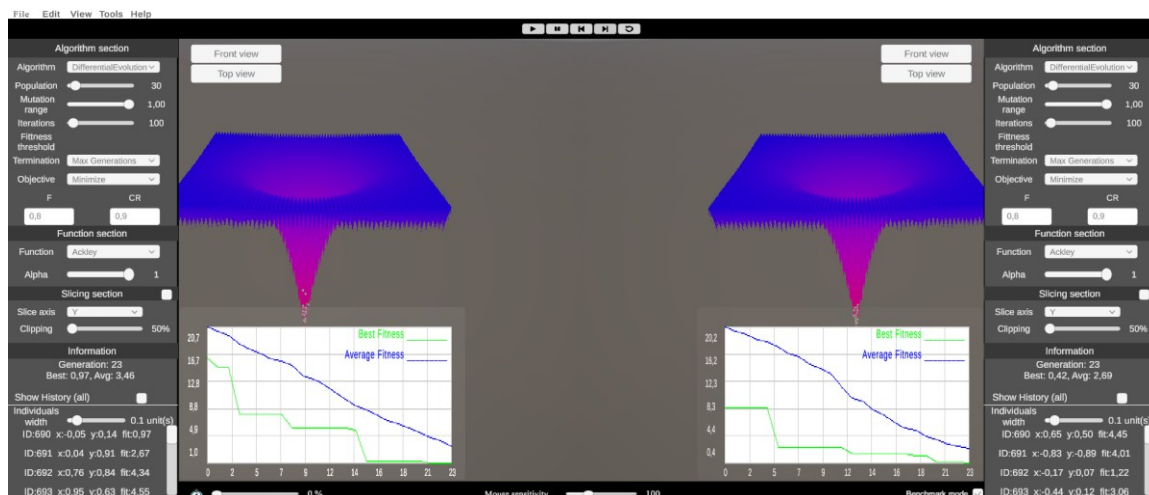
(zdroj: vlastní)

### 3.2.4 Vizualizační plocha

Hlavní část obrazovky je vyhrazena vizualizaci optimalizované funkce a aktuální populace řešení. Trojrozměrná síť cílové funkce je vykreslována skrze *MeshSlicer* a každý jedinec v populaci je reprezentován jako bod. Pohyb kamerou je uživateli umožněn pomocí klasických interakčních gest (otáčení, zoom, posun pomocí W/S/A/D).

Uživateli je umožněno kliknout na libovolného jedince a zobrazit jeho detailní informace, jako jsou souřadnice, hodnota fitness, příslušnost ke generaci a identifikátor. Zároveň se i vizuálně zobrazí jeho dráha od první generace. Tato funkce je užitečná zejména při ladění algoritmu nebo při sledování vývoje konkrétních řešení.

Pokud se uživatel přepne do Benchmark módu, rozdělí se vizualizační plocha na dvě poloviny a zároveň se zpřístupní pravý ovládací panel, který umožní ovládat pravou polovinu.

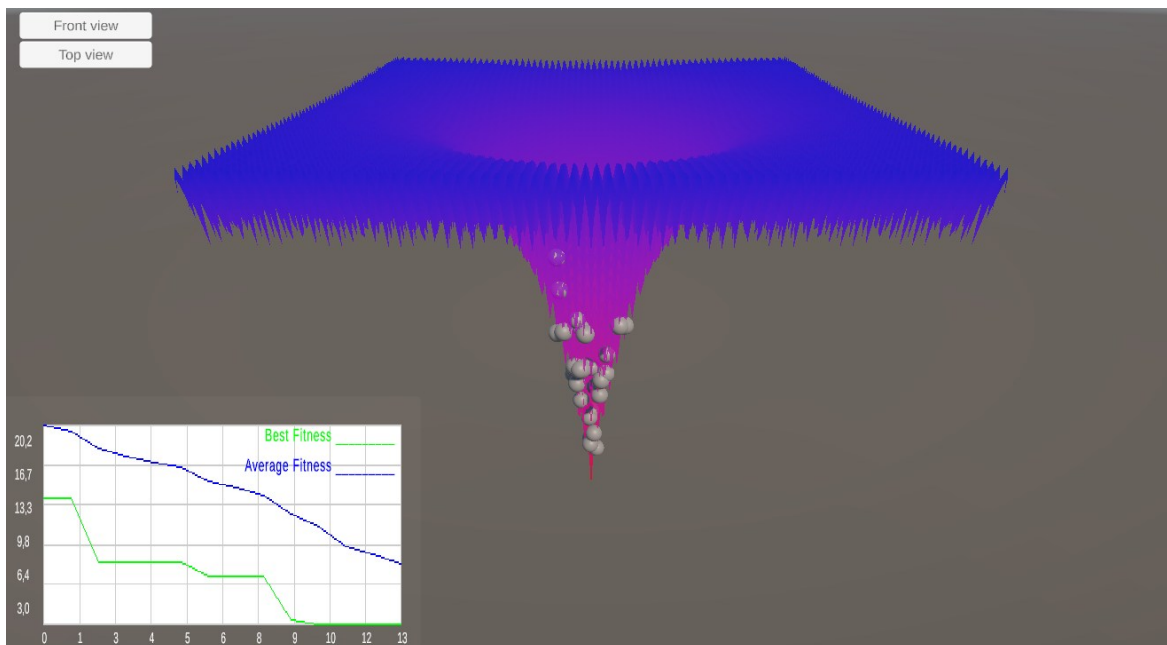


Obrázek 20 Benchmark mode

(zdroj: vlastní)

### 3.2.5 Fitness graf a spodní panel

V dolní části obrazovky je umístěn panel s grafem průběhu fitness funkce v čase. Tento graf slouží jako doplňkový analytický nástroj, který poskytuje zpětnou vazbu o výkonnosti algoritmu v jednotlivých generacích. Uživatel má možnost graf skrýt či zobrazit dle potřeby.



Obrázek 21 Graf a evoluce po 13. generaci

(zdroj: vlastní)

Pod grafem se nachází lišta s posuvníkem pro nastavení rychlosti simulace, posuvníkem pro citlivost myši a zaškrtnuté pole pro zapnutí benchmark módu (rozdělení scény).

### 3.3 Implementace algoritmů

V souboru `IEvolutionAlgorithm.cs` je definováno rozhraní `IEvolutionAlgorithm`, které vyžaduje jedinou metodu `ComputeNextGeneration` přijímající aktuální populaci jedinců, delegát pro vyhodnocení fitness, minimální a maximální hranice domény, cílovou velikost populace, mutační rozsah, příznak, zda minimalizujeme nebo maximalizujeme a log, do kterého se budou zapisovat informace. Implementace pěti algoritmů v podobě `ScriptableObject` tříd na toto rozhraní navazuje jednotným způsobem: kód vždy převezme seznam `List<EvolutionVisualizer.Individual>` a vrátí nový seznam jedinců pro další generaci.

```
public interface IEvolutionAlgorithm {  
    /// <summary>  
    /// Computes the next generation of individuals.  
    /// </summary>  
    6 references  
    List<EvolutionVisualizer.Individual> ComputeNextGeneration(  
        List<EvolutionVisualizer.Individual> currentPopulation,  
        Func<float, float, float> evaluation,  
        float domainMin, float domainMax,  
        int populationSize,  
        float mutationRange,  
        bool maximizeFitness,  
        Logger.LoggerType type  
    );  
}
```

Obrázek 22 IEvolutionAlgorithm interface

(zdroj: vlastní)

#### 3.3.1 Genetický algoritmus

V `GeneticAlgorithm` se zachovává elitismus tak, že první potomek nové generace vznikne přímým zkopírováním nejlepšího jedince předchozí generace (podle `GetBestIndividual`). Ostatní potomci se tvoří v nekonečné smyčce, dokud počet nepřesáhne požadovanou velikost `populationSize`. Pro výběr rodičů je využito turnajové selekce o velikosti tří, která v každém kole náhodně vybere tři jedince a porovná jejich fitness. Vlastní křížení je realizováno blend crossoverem: s pravděpodobností `crossoverRate` se vytvoří vážený průměr dvou rodičů, jinak potomek zdědí celou pozici jednoho z rodičů. Následuje mutace s pravděpodobností `mutationProbability`, kde se obě souřadnice posunou o náhodnou hodnotu v intervalu  $\pm mutationRange$  a poté se znovu oříznou na hranice domény. Každý potomek dostane dočasné ID -1 a posléze, při zpracování ve vizualizéru, skutečné nové ID.

Zároveň se jeho generace zvýší o jedničku a uloží se obě rodičovská ID, což umožňuje pozdější vykreslení rodokmenu.

```
public List<EvolutionVisualizer.Individual> ComputeNextGeneration(
    List<EvolutionVisualizer.Individual> currentPopulation,
    Func<float, float, float> evaluation,
    float domainMin, float domainMax,
    int populationSize,
    float mutationRange,
    bool maximizeFitness,
    Logger.LoggerType type
) {
    var nextGen = new List<EvolutionVisualizer.Individual>();
    Logger.Instance.Add(type, "-----");
    Logger.Instance.Add(type, $"GA");
    Logger.Instance.Add(type, $"Crossover rate: {crossoverRate}");
    Logger.Instance.Add(type, $"Mutation Probability: {mutationProbability}");

    var elite = GetBestIndividual(currentPopulation, maximizeFitness);
    nextGen.Add(new EvolutionVisualizer.Individual(
        elite.Id, elite.Pos, elite.Fitness, elite.Generation + 1,
        elite.Parent1 >= 0 ? elite.Parent1 : elite.Id));

    while (nextGen.Count < populationSize) {
        var p1 = TournamentSelection(currentPopulation, 3, maximizeFitness);
        var p2 = TournamentSelection(currentPopulation, 3, maximizeFitness);

        Vector2 childPos;
        if (UnityEngine.Random.value < crossoverRate) {
            float alpha = UnityEngine.Random.value;
            childPos = alpha * p1.Pos + (1 - alpha) * p2.Pos;
        } else {
            childPos = (UnityEngine.Random.value < 0.5f) ? p1.Pos : p2.Pos;
        }
        if (UnityEngine.Random.value < mutationProbability) {
            float dx = UnityEngine.Random.Range(-mutationRange, mutationRange);
            float dy = UnityEngine.Random.Range(-mutationRange, mutationRange);
            childPos += new Vector2(dx, dy);
        }
        childPos.x = Mathf.Clamp(childPos.x, domainMin, domainMax);
        childPos.y = Mathf.Clamp(childPos.y, domainMin, domainMax);

        float fitness = evaluation(childPos.x, childPos.y);
        nextGen.Add(new EvolutionVisualizer.Individual(
            -1, childPos, fitness, currentPopulation[0].Generation + 1,
            p1.Id, p2.Id));
    }
    return nextGen;
}
```

Obrázek 23 Zdrojový kód genetického algoritmu

(zdroj: vlastní)

### 3.3.2 Evoluční strategie

Zde je využit model  $(\mu, \lambda)$ , kde  $\mu$  je aktuální populace a  $\lambda$  je počet potomků na rodiče, nastavený v parametru *offspringPerParent*. K původní populaci jsou přidávány potomci tak, že ke každému rodiči jsou vygenerovány *offspringPerParent* kopií s náhodnou mutací v rámci  $\pm$ *mutationRange*. Následně je počítána jejich fitness a výsledek je seřazen podle fitness. Z výsledného seznamu je vybráno prvních *populationSize* jedinců pro další generaci, přičemž je u každého ID zachován původní rodič. Tím je docíleno, že se do nové generace zahrnou skutečně nejlepší výsledky mezi rodiči i jejich potomky.

```
public class EvolutionStrategy : ScriptableObject, IEvolutionAlgorithm {
    2 references
    [SerializeField] private int offspringPerParent = 5;

    2 references
    public List<EvolutionVisualizer.Individual> ComputeNextGeneration(
        List<EvolutionVisualizer.Individual> currentPopulation,
        Func<float, float, float> evaluation,
        float domainMin, float domainMax,
        int populationSize,
        float mutationRange,
        bool maximizeFitness,
        Logger.LoggerType type
    ) {
        Logger.Instance.Add(type, "-----");
        Logger.Instance.Add(type, $"ES");
        Logger.Instance.Add(type, $"Offspring per parent: {offspringPerParent}");

        var combined = new List<EvolutionVisualizer.Individual>(currentPopulation);
        foreach (var parent in currentPopulation) {
            for (int j = 0; j < offspringPerParent; j++) {
                float dx = UnityEngine.Random.Range(-mutationRange, mutationRange);
                float dy = UnityEngine.Random.Range(-mutationRange, mutationRange);
                var pos = new Vector2(parent.Pos.x + dx, parent.Pos.y + dy);
                pos.x = Mathf.Clamp(pos.x, domainMin, domainMax);
                pos.y = Mathf.Clamp(pos.y, domainMin, domainMax);
                float fit = evaluation(pos.x, pos.y);
                combined.Add(new EvolutionVisualizer.Individual(
                    -1, pos, fit, parent.Generation + 1, parent.Id));
            }
        }
        combined.Sort((a,b) => maximizeFitness ? b.Fitness.CompareTo(a.Fitness)
                                     : a.Fitness.CompareTo(b.Fitness));
        return combined.GetRange(0, Mathf.Min(populationSize, combined.Count));
    }
}
```

Obrázek 24 Zdrojový kód evoluční strategie

(zdroj: vlastní)

### 3.3.3 Diferenciální evoluce

Zde je implementována varianta DE/rand/1/bin. Pro každého jedince  $i$  jsou nejprve náhodně vybrány tři různé indexy  $a$ ,  $b$ ,  $c$  (nejsou-li totožné s  $i$ ) a vypočítá se donorový vektor  $\text{pop}[a] + F \cdot (\text{pop}[b] - \text{pop}[c])$ . Poté je zkopírována pozice původního jedince do trial vektoru a pro každou souřadnici porovnáno náhodné číslo s parametrem CR; pokud je menší, tato souřadnice je přepsána hodnotou z donora. Trial vektor po oříznutí do  $[dmin, dmax]$  je vyhodnocen a porovnán s fitness původního jedince. Pokud trial zlepší fitness (respektive zhorší, v režimu maximalizace), trial vstoupí do další generace; jinak se zachová pozice  $i$ . Posledním parametrem je rodičovské ID, které je uchováváno pro konzistenci rodokmenu.

```
public class DifferentialEvolution : ScriptableObject, IEvolutionAlgorithm {
    2 references
    [SerializeField] private float F = 0.8f;
    3 references
    [SerializeField] private float CR = 0.9f;

    2 references
    public List<EvolutionVisualizer.Individual> ComputeNextGeneration(
        List<EvolutionVisualizer.Individual> pop,
        Func<float, float, float> eval,
        float dmin, float dmax,
        int size, float mrange, bool maximize,
        Logger.LoggerType type
    ) {
        Logger.Instance.Add(type, "-----");
        Logger.Instance.Add(type, $"DE");
        Logger.Instance.Add(type, $"F: {F}");
        Logger.Instance.Add(type, $"CR: {CR}");

        int n = pop.Count;
        var next = new List<EvolutionVisualizer.Individual>(n);
        for (int i = 0; i < n; i++) {
            int a, b, c;
            do { a = UnityEngine.Random.Range(0, n); } while (a == i);
            do { b = UnityEngine.Random.Range(0, n); } while (b == i || b == a);
            do { c = UnityEngine.Random.Range(0, n); } while (c == i || c == a || c == b);
            var donor = pop[a].Pos + F * (pop[b].Pos - pop[c].Pos);
            var trial = pop[i].Pos;
            if (UnityEngine.Random.value < CR) trial.x = donor.x;
            if (UnityEngine.Random.value < CR) trial.y = donor.y;
            trial.x = Mathf.Clamp(trial.x, dmin, dmax);
            trial.y = Mathf.Clamp(trial.y, dmin, dmax);
            float fitT = eval(trial.x, trial.y);
            var parent = pop[i];
            bool better = maximize ? fitT > parent.Fitness : fitT < parent.Fitness;
            next.Add(new EvolutionVisualizer.Individual(
                -1,
                better ? trial : parent.Pos,
                better ? fitT : parent.Fitness,
                parent.Generation + 1,
                parent.Id));
        }
        return next;
    }
}
```

Obrázek 25 Zdrojový kód diferenciální evoluce

(zdroj: vlastní)

### 3.3.4 PSO

V ParticleSwarmOptimization jsou vedle obvyklého nastavení *inertiaWeight*, *cognitiveCoefficient* a *socialCoefficient* udržovány tři persistentní slovníky jako pole třídy: *velocities*, *personalBestPos* a *personalBestFit*. Při změně algoritmu jsou tyto slovníky vyčištěny, aby resetovaly stav starých běhů. V každé fázi jsou pak hledáni globální nejlepší jedinci podle *maximizeFitness*, aktualizovány pro každý jeho osobní *best*, spočítána nová rychlost jako kombinace setrvačné složky, kognitivní složky směrem k osobnímu *best* a sociální složky směrem k globálnímu *best*. Nová pozice je přičtena k původní, oříznuta do domény a spočtena fitness. ID částice je stejně jako v předchozích algoritmech nastavena na -1 a později přepsána v hlavním skriptu. Rodiče nastaví na ID aktuální částice, takže v *positionHistory* vzniká spojitá trajektorie, která se ve vizualizéru vykreslí pomocí LineRendereru.

```
public class ParticleSwarmOptimization : ScriptableObject, IEvolutionAlgorithm
{
    2 references
    [SerializeField] private float inertiaWeight = 0.729f;
    2 references
    [SerializeField] private float cognitiveCoefficient = 1.49445f;
    2 references
    [SerializeField] private float socialCoefficient = 1.49445f;
    5 references
    private Dictionary<int, Vector2> velocities = new Dictionary<int, Vector2>();
    5 references
    private Dictionary<int, Vector2> personalBestPos = new Dictionary<int, Vector2>();
    3 references
    private Dictionary<int, float> personalBestFit = new Dictionary<int, float>();
}
```

Obrázek 26 Nastavení základních parametrů a vytvoření slovníků PSO

(zdroj: vlastní)

```

public List<EvolutionVisualizer.Individual> ComputeNextGeneration(
    List<EvolutionVisualizer.Individual> pop,
    Func<float, float, float> eval, float dmin, float dmax,
    int size, float mrange, bool maximize,
    Logger.LoggerType type
)
{
    Logger.Instance.Add(type, "-----");
    Logger.Instance.Add(type, $"PSO");
    Logger.Instance.Add(type, $"InertiaWeight: {inertiaWeight}");
    Logger.Instance.Add(type, $"CC: {cognitiveCoefficient}");
    Logger.Instance.Add(type, $"SC: {socialCoefficient}");

    EvolutionVisualizer.Individual global = pop[0];
    foreach (var ind in pop)
    {
        bool better = maximize ? ind.Fitness > global.Fitness : ind.Fitness < global.Fitness;
        if (better) global = ind;
        if (!personalBestPos.ContainsKey(ind.Id))
        {
            personalBestPos[ind.Id] = ind.Pos;
            personalBestFit[ind.Id] = ind.Fitness;
        }
        else if (better)
        {
            personalBestPos[ind.Id] = ind.Pos;
            personalBestFit[ind.Id] = ind.Fitness;
        }
    }
    var next = new List<EvolutionVisualizer.Individual>(pop.Count);
    foreach (var part in pop)
    {
        if (!velocities.ContainsKey(part.Id)) velocities[part.Id] =
            new Vector2(UnityEngine.Random.Range(-mrange, mrange), UnityEngine.Random.Range(-mrange, mrange));
        var vel = velocities[part.Id];
        var cog = cognitiveCoefficient * UnityEngine.Random.value * (personalBestPos[part.Id] - part.Pos);
        var soc = socialCoefficient * UnityEngine.Random.value * (global.Pos - part.Pos);
        var newVel = inertiaWeight * vel + cog + soc;
        var newPos = part.Pos + newVel;
        newPos.x = Mathf.Clamp(newPos.x, dmin, dmax);
        newPos.y = Mathf.Clamp(newPos.y, dmin, dmax);
        var newFit = eval(newPos.x, newPos.y);
        velocities[part.Id] = newVel;
        next.Add(new EvolutionVisualizer.Individual(
            -1, newPos, newFit, part.Generation + 1, part.Id));
    }
    return next;
}

```

Obrázek 27 Zdrojový kód algoritmu PSO

(zdroj: vlastní)

### 3.3.5 SOMA

Algorismus SOMA v této implementaci hledá leadera jako jedince s nejlepší fitness, kterého přímo zkopíruje do nové generace (elitismus). Ostatní jedinci jsou pro každý krok  $t$  od  $stepSize$  do  $pathLength$  posouváni o vektor směru k leaderovi násobený  $t$ , přičemž při každé nové pozici je hodnocena fitness a porovnávána s dosavadním *bestMig*. Pokud nová pozice zlepší fitness, nahradí ji. Po dokončení kroků je přidána „nejlepší migrace“ každého jedince. V případě, že výsledný počet jedinců nedosahuje *size*, jsou náhodně generováni další jedinci. Každý potomek přebírá jako Parent1 ID původního jedince, což umožňuje sledovat migraci

ve vizualizéru, a do nového konstruktoru je předána -1 jako ID, které je pak v hlavním skriptu doplněno.

```
public class SOMAAlgorithm : ScriptableObject, IEvolutionAlgorithm {
    2 references
    [SerializeField] private float pathLength = 3f;
    3 references
    [SerializeField] private float stepSize = 0.11f;

    2 references
    public List<EvolutionVisualizer.Individual> ComputeNextGeneration(
        List<EvolutionVisualizer.Individual> pop,
        Func<float,float,float> eval,float dmin,float dmax,
        int size,float mrange,bool maximize,
        Logger.LoggerType type
    ) {
        Logger.Instance.Add(type,"-----");
        Logger.Instance.Add(type,$"SOMA");
        Logger.Instance.Add(type,$"PathLenhth: {pathLength}");
        Logger.Instance.Add(type,$"Stepsize: {stepSize}");

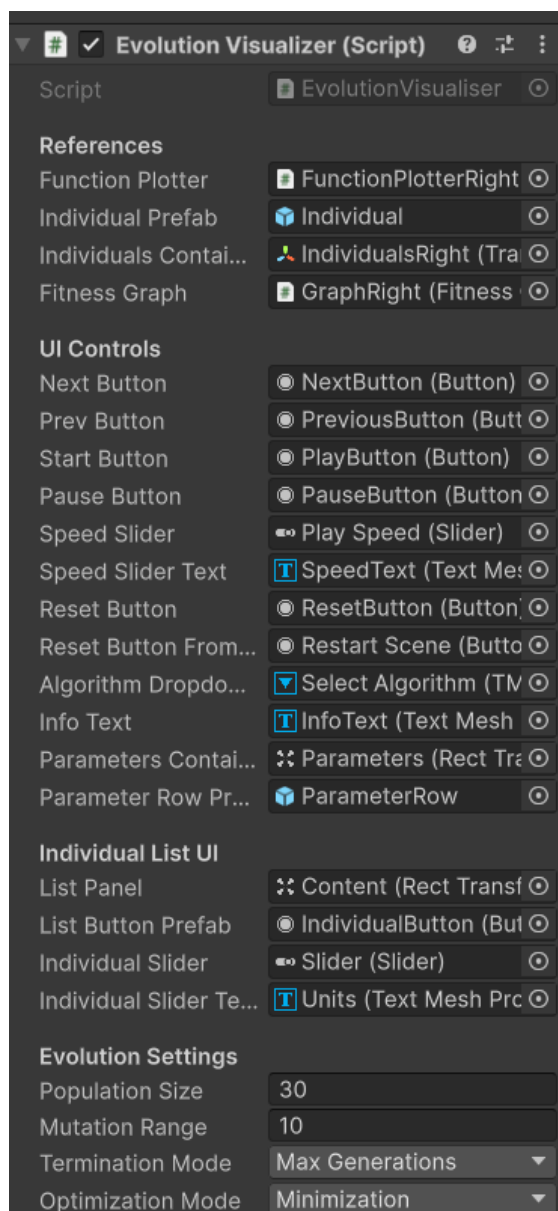
        var best = pop[0];
        foreach(var ind in pop) {
            bool better = maximize ? ind.Fitness>best.Fitness : ind.Fitness<best.Fitness;
            if(better) best = ind;
        }
        var next = new List<EvolutionVisualizer.Individual>(pop.Count);
        next.Add(new EvolutionVisualizer.Individual(
            best.Id,best.Pos,best.Fitness,best.Generation+1,best.Id));
        foreach(var ind in pop) {
            if(ind.Id==best.Id) continue;
            var bestMig = new EvolutionVisualizer.Individual(
                -1,ind.Pos,ind.Fitness,ind.Generation+1,ind.Id);
            for(float t=stepSize; t<=pathLength; t+=stepSize) {
                var dir = (best.Pos-ind.Pos).normalized;
                var pos = ind.Pos + dir*t;
                pos.x = Mathf.Clamp(pos.x,dmin,dmax);
                pos.y = Mathf.Clamp(pos.y,dmin,dmax);
                var f = eval(pos.x,pos.y);
                bool btr = maximize ? f>bestMig.Fitness : f<bestMig.Fitness;
                if(btr) bestMig = new EvolutionVisualizer.Individual(
                    -1,pos,f,ind.Generation+1,ind.Id);
            }
            next.Add(bestMig);
        }
        while(next.Count<size) {
            float x=UnityEngine.Random.Range(dmin,dmax);
            float y=UnityEngine.Random.Range(dmin,dmax);
            float f=eval(x,y);
            next.Add(new EvolutionVisualizer.Individual(-1,new Vector2(x,y),f,pop[0].Generation+1));
        }
        return next;
    }
}
```

Obrázek 28 Zdrojový kód algoritmu SOMA

(zdroj: vlastní)

### 3.4 Hlavní komponenty a jejich účel

Jak již bylo zmíněno v kapitole 3.1.3, jádrem aplikace je třída *EvolutionVisualizer*, která řídí průběh evoluce a propojuje výpočetní část s grafickým rozhraním. Evoluční algoritmy jsou implementovány samostatně jako *ScriptableObject* třídy v souboru *EvolutionAlgorithms.cs*. Dále systém obsahuje komponenty pro vykreslení optimalizačních funkcí, zobrazení průběhu fitness (*FitnessGraphWithGrid*) a komponenty uživatelského rozhraní pro ovládání aplikace (*EvolutionUIDriver*, panel s parametry algoritmů, seznam jedinců aj.).



Obrázek 29 Komponenty pracující s EvolutionVisualizer.cs

(zdroj: vlastní)

EvolutionVisualizer je komponenta Unity (MonoBehaviour), která při spuštění scény inicializuje počáteční populaci, nastaví UI a stará se o iteraci generací. Kód pro inicializaci je uveden v metodě *Start()*. Tam se volá *BindUI()* pro navázání ovládacích prvků uživatelského rozhraní a *InitializeGeneration()* pro vygenerování počáteční generace jedinců. Poté se prvotní populace ihned vykreslí voláním *DisplayCurrentGeneration()* a zobrazí se souhrnné informace v informačním panelu pomocí funkce *UpdateInfoPanel()*.

```
IEnumerator Start()
{
    Logger.Instance.Add(loggerType, "[Start] beginning initialization");
    BindUI();
    Logger.Instance.Add(loggerType, "[Start] UI bound");

    InitializeGeneration();
    Logger.Instance.Add(loggerType, "[Start] generation initialized");

    yield return null;

    DisplayCurrentGeneration();
    Logger.Instance.Add(loggerType, "[Start] displayed generation 0");

    UpdateInfoPanel();
    Logger.Instance.Add(loggerType, "[Start] info panel updated");
}
```

Obrázek 30 Inicializace programu po spuštění

(zdroj: vlastní)

Ve třídě je také vnitřní datová struktura *Individual* pro reprezentaci jedince a uchovává kolekce relevantní pro průběh simulace. Patří sem:

- *generations* – seznam generací, kde každá generace je list objektů *Individual*. Tím je zajištěna historie vývoje populace a možnost navigace zpět na předchozí generace,
- *positionHistory* – slovník mapující ID jedince na seznam jeho pozic v průběhu času. Tento záznam se využívá pro vykreslení trajektorie pohybu jedince napříč generacemi,
- *generationData* – seznam struktur s agregovanými statistikami pro každou generaci (nejlepší dosažená fitness a průměrná fitness dané generace). Tyto údaje slouží k vykreslení grafu fitness a informativnímu zobrazení.

```

9 references
private Dictionary<int, List<Vector3>> positionHistory = new Dictionary<int, List<Vector3>>();
16 references
private List<List<Individual>> generations = new List<List<Individual>>();
5 references
private List<GenerationData> generationData = new List<GenerationData>();

```

Obrázek 31 Kolekce pro uchovávání dat během simulace

(zdroj: vlastní)

Tyto kolekce hrají také velmi důležitou roli při exportu a importu dat, kde se ukládají i společně se jménem funkce a posledním použitým algoritmem.

Jednotlivý jedinec je reprezentován instancí vnitřní třídy *Individual* s poli pro ID, 2D pozici v prostoru hledání (*Vector2 Pos*), hodnotu fitness, číslo generace a případné identifikátory rodičů (*Parent1*, *Parent2*). ID jsou jedinečné v rámci jedné simulace a při vytváření nových jedinců se generují metodou *GetNextId()*.

```

/// <summary>
/// Represents an individual with unique ID, position, fitness, generation, and parent IDs.
/// </summary>
59 references
public class Individual
{
    37 references
    public int Id;
    34 references
    public Vector2 Pos;
    36 references
    public float Fitness;
    15 references
    public int Generation;
    6 references
    public int Parent1 = -1;
    2 references
    public int Parent2 = -1;

    11 references
    public Individual(int id, Vector2 pos, float fitness, int generation, int p1 = -1, int p2 = -1)
    {
        Id = id;
        Pos = pos;
        Fitness = fitness;
        Generation = generation;
        Parent1 = p1;
        Parent2 = p2;
    }
}

```

Obrázek 32 Ukázka třídy Individual

(zdroj: vlastní)

### 3.4.1 Práce s evolucí

Při startu aplikace, hned po zavolání *BindUI()*, je volána také funkce *InitializeGeneration()*, která vygeneruje počáteční populaci pomocí pomocné metody *CreateRandomPopulation(int*

generation). V té se pro každého jedince náhodně zvolí souřadnice  $x$  a  $y$  rovnoměrně z celého definičního oboru aktuálně zvolené optimalizační funkce a spočítá se jeho fitness. Každému jedinci je přiřazeno nové ID a generace 0. Vygenerovaná populace se uloží do seznamu generací a zároveň se zaznamená její nejlepší a průměrná fitness pomocí *RecordGenerationData(first)*.

```

/// <summary>
/// Builds a new population of random individuals within the current function domain.
/// </summary>
/// <param name="generation">The generation index to assign to each new individual.</param>
/// <returns>A list of newly created individuals.</returns>
1 reference
private List<Individual> CreateRandomPopulation(int generation)
{
    var list = new List<Individual>();
    float min = functionPlotter.GetDomainMin();
    float max = functionPlotter.GetDomainMax();
    var func = functionPlotter.GetCurrentFunction();
    for (int i = 0; i < populationSize; i++)
    {
        var x = UnityEngine.Random.Range(min, max);
        var y = UnityEngine.Random.Range(min, max);
        var fit = func(x, y);
        list.Add(new Individual(nextId++, new Vector2(x, y), fit, generation));
    }
    return list;
}

```

Obrázek 33 Ukázka funkce CreateRandomPopulation()

(zdroj: vlastní)

Přes evoluci lze iterovat buď manuálně tlačítky „Další generace“ a „Předchozí generace“, nebo automaticky stiskem tlačítka „Start“.

Metoda *NextGeneration()* ověřuje, zda má evoluce pokračovat (dle ukončovacích podmínek), a poté buď přejde na již vypočtenou další generaci (pokud už byla vytvořena, typicky po stisku „Předchozí“), nebo vygeneruje novou generaci pomocí *ComputeNextGen()*. Při generování nové populace se získá aktuálně vybraný evoluční algoritmus (*GetSelectedAlgorithm()*) a zavolá jeho metodu *ComputeNextGeneration(...)*. Do ní se předává aktuální populace jedinců, zvolená fitness funkce, rozsah domény a parametry, jako velikost populace a rozsah mutace nastavené uživatelem. Výstupem je nový list jedinců neboli potomků, kterým je přiřazeno číslo nové generace a jsou jim případně vygenerována nová unikátní ID.

```

/// <summary>
/// Runs the selected evolution algorithm to compute and return the next generation.
/// </summary>
/// <param name="currentPop">Current population of individuals.</param>
/// <param name="genIdx">Index (number) of the next generation.</param>
/// <returns>A newly created list of individuals for the next generation.</returns>
1 reference
private List<Individual> ComputeNextGen(List<Individual> currentPop, int genIdx)
{
    var algo = GetSelectedAlgorithm();
    var next = algo.ComputeNextGeneration(currentPop,
        functionPlotter.GetCurrentFunction(),
        functionPlotter.GetDomainMin(), functionPlotter.GetDomainMax(),
        populationSize, mutationRange, optimizationMode == OptimizationMode.Maximization, loggerType);
    next.ForEach(i => i.Generation = genIdx);

    AssignGenerationAndIds(next, genIdx);

    return next;
}

```

Obrázek 34 Výpočet nové generace

(zdroj: vlastní)

Nová generace se připojí do seznamu *generations* a zvýší se čítač *currentGen*. Následně se opět volá *RecordGenerationData(next)* pro uložení statistik a provede se vykreslení a aktualizace UI: *DisplayCurrentGeneration()* a *UpdateInfoPanel()*.

```

/// <summary>
/// Clears any existing markers and lines, then spawns a visual marker for each individual
/// in the current generation and updates the fitness graph.
/// </summary>
5 references
private void DisplayCurrentGeneration()
{
    ClearLinesAndMarkers();
    var pop = generations[currentGen];
    ClearChildren(individualsContainer);
    ClearChildren(listPanel);

    foreach (var ind in pop)
    {
        SpawnMarker(ind);
        fitnessGraph.AddFitnessData(currentGen, GetBest(pop), GetAverage(pop));
    }
}

```

Obrázek 35 Zobrazování jedinců v generaci

(zdroj: vlastní)

Nyní, když máme definované funkce *ComputeNextGen()* a *DisplayCurrentGen()*, můžeme nadefinovat funkci, kterou budeme volat při kliknutí na tlačítko „Next“, nebo která bude volána automaticky v *Autoplay()*.

```

/// <summary>
/// Advances to the next generation, computing it if necessary. Stops autoplay if termination conditions are met.
/// </summary>
3 references
private void NextGeneration()
{
    if (CheckTermination()) { StopAutoplay(); return; }
    if (currentGen < generations.Count - 1)
    {
        currentGen++;
    }
    else
    {
        var next = ComputeNextGen(generations[currentGen], currentGen + 1);
        generations.Add(next);
        currentGen++;
        RecordGenerationData(next);
    }
    DisplayCurrentGeneration();
    UpdateInfoPanel();
}

```

Obrázek 36 Funkce počítající další generaci, je-li to nutné

(zdroj: vlastní)

Automatická evoluce: pro režim autoplay slouží metoda *StartAutoplay()*, která spustí coroutine *Autoplay()*. V ní běží nekonečná smyčka volající *NextGeneration()* a následně čekající určitý časový interval. Doba čekání je odvozena od polohy posuvníku, který se nachází ve spodním panelu nalevo, hned vedle obrázku s hodinami. Při vyšší rychlosti je prodleva kratší. Uživatel tak může plynule zrychlovat nebo zpomalovat evoluci. Smyčka se přeruší buď dosažením ukončovací podmínky (viz níže), nebo stiskem tlačítka „Pauza“ (metoda *StopAutoplay()* ukončí coroutine).

```

/// <summary>
/// Starts automatic progression through generations at the current speed.
/// </summary>
2 references
private void StartAutoplay() => autoplay = StartCoroutine(Autoplay());

/// <summary>
/// Stops the automatic generation progression, if it is running.
/// </summary>
3 references
private void StopAutoplay() { if (autoplay != null) StopCoroutine(autoplay); autoplay = null; }

/// <summary>
/// Coroutine that repeatedly advances to the next generation, waiting
/// based on the speed slider value between steps.
/// </summary>
1 reference
private IEnumerator Autoplay()
{
    while (true)
    {
        NextGeneration();
        yield return new WaitForSeconds((1 - speedSlider.value) * maxDelay);
    }
}

```

Obrázek 37 Proces automatické simulace

(zdroj: vlastní)

Krok zpět: metoda *PrevGeneration()* snižuje index *currentGen* o jedničku, pokud to jde, a překreslí příslušnou předchozí generaci. Díky uchovávání historie populací je tak možné prohlížet i starší generace. Důležité je, že pokud uživatel poté opět spustí evoluci dopředu (ručně nebo autoplay), program nepočítá znovu již známé generace, ale jen změní index (dokud nedojde na dosud nejnovější generaci). Teprve z poslední dostupné generace se případně spočte nová.

```
/// <summary>
/// Moves back one generation, if not already at generation zero.
/// </summary>
2 references
private void PrevGeneration()
{
    if (currentGen > 0) { currentGen--; DisplayCurrentGeneration(); UpdateInfoPanel(); }
}
```

Obrázek 38 Kód pro posun generace o krok zpět

(zdroj: vlastní)

Ukončení simulace: *EvolutionVisualizer* podporuje dva typy ukončení. Tím prvním je pevný počet generací: pokud je v nastavení zvoleno ukončení po pevně daném počtu generací, metoda *CheckTermination()* vrátí *true* po dosažení tohoto limitu. Při automatickém režimu to vede k zastavení smyčky. Druhý typ ukončení je prahová hodnota fitness: pokud je zapnutý fitness threshold, metoda *CheckTermination()* v každém kroku spočte nejlepší fitness v aktuální populaci a porovná ji s nastaveným prahem. Pro minimalizační úlohy vrací *true*, jakmile nejlepší (nejnižší) fitness klesne pod tuto hranici (s určitou tolerancí *eps* pro numerickou stabilitu), pro maximalizaci analogicky když fitness stoupne nad stanovený limit.

```

/// <summary>
/// Checks whether the stopping criteria have been met: either max generations reached
/// or fitness threshold achieved (if that mode is selected).
/// </summary>
/// <returns>True if evolution should stop; false otherwise.</returns>
1 reference
private bool CheckTermination()
{
    Logger.Instance.Add(loggerType, $"termination: {terminationMode}, min/max: {optimizationMode}, max-gen: {maxGenerations}");
    if (currentGen >= maxGenerations)
        return true;

    if (terminationMode == TerminationMode.FitnessThreshold)
    {
        float best = (optimizationMode == OptimizationMode.Minimization)
            ? float.MaxValue
            : float.MinValue;

        foreach (var ind in generations[currentGen])
        {
            float fit = ind.Fitness;
            if (optimizationMode == OptimizationMode.Minimization)
                best = Mathf.Min(best, fit);
            else
                best = Mathf.Max(best, fit);
        }

        const float eps = 1e-6f;

        if (optimizationMode == OptimizationMode.Minimization)
            return best <= fitnessThreshold + eps;
        else
            return best >= fitnessThreshold - eps;
    }

    return false;
}

```

Obrázek 39 Ukončení simulace při splnění podmínky

(zdroj: vlastní)

### 3.4.2 Vizualizace jedinců a jejich evoluce v prostoru

Jedním z hlavních cílů aplikace je názorně zobrazit rozmístění jedinců v definičním oboru funkce a jejich evoluci (postupné zlepšování fitness). K tomu slouží kombinace 3D grafu optimalizační funkce a grafických reprezentací jedinců v tomto terénu. Proto je ve třídě *EvolutionVisualizer* definován Prefab objektu jedince (*individualPrefab*), kterým je jednoduchá 3D koule. Při vykreslování generace se pro každého jedince volá *SpawnMarker(ind)*, která instancuje Prefab na správné pozici nad grafem funkce. Určení polohy probíhá tak, že 2D souřadnice jedince (x,y) v doméně funkce se převedou na souřadnice v lokálním souřadném systému vygenerovaného mesh grafu. Výpočty obstarává metoda *WorldPosition(Individual ind)*. Ta normalizuje *x* a *y* na intervalu  $[-0.5, 0.5]$  vzhledem k rozsahu domény a získá příslušnou výšku *z* funkce (*z* = hodnotě funkce, normalizované do  $[0,1]$  a vynásobené svislým měřítkem). Tím získá bod v lokálních souřadnicích meshe, který následně převádí na světové souřadnice voláním *meshFilter.transform.TransformPoint(local)*.

```

/// <summary>
/// Converts an individual's 2D (x,y) function coordinates into a 3D world position
/// by normalizing over the function domain and evaluating the Z-value on the mesh.
/// </summary>
/// <param name="ind">The individual whose world position to compute.</param>
/// <returns>World-space 3D position of the individual marker.</returns>
3 references
private Vector3 WorldPosition(Individual ind)
{
    float min = functionPlotter.GetDomainMin();
    float max = functionPlotter.GetDomainMax();
    float normX = ((ind.Pos.x - min) / (max - min)) - 0.5f;
    float normY = ((ind.Pos.y - min) / (max - min)) - 0.5f;
    float zRaw = functionPlotter.GetCurrentFunction()(ind.Pos.x, ind.Pos.y);
    float z = functionPlotter.verticalScale
        * ((zRaw - meshGenerator.MinZ) / (meshGenerator.MaxZ - meshGenerator.MinZ));
    Vector3 local = new Vector3(normX * functionPlotter.horizontalScale, z, normY * functionPlotter.horizontalScale);
    return functionPlotter.meshFilter.transform.TransformPoint(local);
}

```

Obrázek 40 Výpočet pozice jedince

(zdroj: vlastní)

Výsledný bod by měl ležet na povrchu vykreslené funkce a aby koule jedinců skutečně ležely přímo na povrchu a zároveň také na správné pozici dané funkce, využívá se ještě drobný trik s raycastem: *SpawnMarker* nejprve posune výchozí umístění koule mírně nad povrch a pak spustí paprsek dolů (*Physics.Raycast*). Raycast detekuje přesnou výšku povrchu mesh (díky přidání MeshCollideru na objektu funkce) a umožní dosadit kouli přesně na povrch, tedy na souřadnice dopadu *hit.point*. Tento postup zajišťuje, že značky jedinců nebudou „levitovat“, ani se částečně propadat pod povrch kvůli případným numerickým nepřesnostem.

```

/// <summary>
/// Instantiates a marker prefab at the individual's world position (dropping onto mesh if possible),
/// records its history, and hooks up click info behavior.
/// </summary>
/// <param name="ind">The individual whose position to visualize.</param>
1 reference
private void SpawnMarker(Individual ind)
{
    Vector3 pos;
    int meshLayer = LayerMask.NameToLayer("FunctionMesh");
    int mask = 1 << meshLayer;
    Vector3 approxPos = WorldPosition(ind) + Vector3.up * 1f;

    if (Physics.Raycast(approxPos, Vector3.down, out RaycastHit hit, 20f, mask))
    {
        pos = hit.point;
    }
    else
    {
        pos = WorldPosition(ind);
    }
    RecordHistory(ind.Id, WorldPosition(ind));
    var go = Instantiate(individualPrefab, pos, Quaternion.identity, individualsContainer);
    go.transform.localScale = Vector3.one * individualScale;
    if (go.GetComponent<Collider>() == null)
    {
        go.AddComponent<SphereCollider>();
    }
    var info = go.AddComponent<IndividualInfo>();
    info.SetIndividualData(ind.Id, ind, this);
}

```

Obrázek 41 Tvorba jedince

(zdroj: vlastní)

Každá vytvořená instance jedince dostane kromě grafické reprezentace také komponentu *IndividualInfo*, jejíž kód můžeme vidět na obrázku 42.

```
public class IndividualInfo : MonoBehaviour

    [Header("References")]
    5 references
    private EvolutionVisualizer.Individual individualData;
    4 references
    private int individualId;
    2 references
    private EvolutionVisualizer parentVisualizer;

    [Header("Read-Only Properties")]
    0 references
    public int IndividualId => individualId;
    1 reference
    public float Fitness => individualData.Fitness;
    1 reference
    public float PositionX => individualData.Pos.x;
    1 reference
    public float PositionY => individualData.Pos.y;

    /// <summary>
    /// Initializes this component with individual data and parent visualizer.
    /// </summary>
    /// <param name="id">Identifier for the individual.</param>
    /// <param name="data">Evolutionary data for the individual.</param>
    /// <param name="visualizer">Parent visualizer instance.</param>
    1 reference
    public void SetIndividualData(int id, EvolutionVisualizer.Individual data, EvolutionVisualizer visualizer)
    {
        individualId = id;
        individualData = data;
        parentVisualizer = visualizer;
    }

    /// <summary>
    /// Unity callback when the object is clicked. Triggers lineage display and logs info.
    /// </summary>
    0 references
    void OnMouseDown()
    {
        ShowLineage();
        LogIndividualInfo();
    }

    /// <summary>
    /// Requests the parent visualizer to display this individual's lineage.
    /// </summary>
    1 reference
    private void ShowLineage()
    {
        parentVisualizer?.ShowLineage(individualId);
    }
}
```

Obrázek 42 Ukázka třídy IndividualInfo

(zdroj: vlastní)

Tato pomocná třída si uloží identitu jedince (*individualId* a referenci na datový objekt *Individual*) a její metoda *OnMouseDown()* reaguje na kliknutí myši na daný objekt. Po kliknutí se zavolá *ShowLineage(individualId)*, čímž umožňuje interaktivní zobrazení rodokmenu daného jedince (viz níže). Díky tomu může uživatel v běžícím systému kliknout na libovolného jedince a získat informace o jeho kvalitě a původu.

```

public void ShowLineage(int id, bool all = false)
{
    if (!all)
    {
        ClearLinesAndMarkers();
    }

    if (id >= 0)
    {
        ExtractLineage(id, out var lineIds);
        lineageFilterIds = new HashSet<int>(lineIds);

        if (!all)
        {
            foreach (Transform c in individualsContainer)
            {
                c.gameObject.SetActive(false);
            }
        }

        var obj = new GameObject("LineageMarkers");
        lineageMarkers.Add(obj);

        foreach (var lid in lineageFilterIds)
        {
            if (!positionHistory.TryGetValue(lid, out var hist) || hist.Count == 0)
                continue;

            Vector3 pt = hist[^1];
            var dot = Instantiate(individualPrefab, pt, Quaternion.identity, obj.transform);
            dot.transform.localScale = Vector3.one * individualScale;
            if (dot.TryGetComponent<Collider>(out var col))
                Destroy(col);
            var rend = dot.GetComponent<Renderer>();
            if (rend != null)
            {
                rend.material.color = Color.cyan;
            }
        }

        DrawTrajectory(id);
        DrawLineageLine(id);
    }
    else
    {
        lineageFilterIds = null;
        foreach (Transform c in individualsContainer)
        {
            c.gameObject.SetActive(true);
        }
    }

    if (!all)
    {
        RefreshListPanel();
    }
}

```

Obrázek 43 Ukázka funkce volané pro zobrazení historie

(zdroj: vlastní)

Při aktivaci rodokmenu konkrétního jedince (kliknutím nebo výběrem v seznamu) se vykreslí jeho trajektorie, spojující všechny pozice, které tento jedinec v průběhu evoluce zaujal. Metoda *DrawTrajectory(int id)* vytvoří nový objekt s komponentou *LineRenderer*, do kterého předá pole všech bodů z *positionHistory[id]*. Tím se v 3D scéně zobrazí lomená

čára procházející postupnými pozicemi jedince. Tloušťka čáry (*widthMultiplier*) je nastavena úměrně zvolenému měřítku jedinců, aby vizuálně ladila s velikostí markerů. Uživatel tak na jedno kliknutí vidí, odkud a kam se daný jedinec během evoluce pohyboval.

```

/// <summary>
/// Draws the full movement trajectory of the given individual using a LineRenderer.
/// </summary>
/// <param name="id">Individual ID whose trajectory to draw.</param>
1 reference
private void DrawTrajectory(int id)
{
    var obj = new GameObject($"Traj_{id}");
    trajectoryLine.Add(obj);
    var lr = obj.AddComponent<LineRenderer>();
    var pts = positionHistory[id].ToArray();
    lr.positionCount = pts.Length;
    lr.widthMultiplier = individualScale / 2;
    lr.SetPositions(pts);

    Color randomColor = UnityEngine.Random.ColorHSV();
    lr.material = new Material(Shader.Find("Unlit/Color"));
    lr.material.color = randomColor;
}

```

Obrázek 44 Vykreslovací funkce

(zdroj: vlastní)

Neméně důležité je pochopení genetických vztahů mezi jedinci napříč generacemi. *EvolutionVisualizer* u každého jedince ukládá ID jeho rodičů (pokud algoritmus nepracuje s rodiči ukládá se ID jedince, ze kterého vzniknul). Při výběru konkrétního jedince se volá *ShowLineage(id)*, která zvýrazní jeho linii předků tak, že deaktivuje zobrazení všech ostatních jedinců v aktuální scéně pro přehlednost. Následně pomocí *ExtractLineage(id, out lineage)* získá seznam předků: tato metoda prochází rekurzivně přes *Parent1* odkazy, takže vytvoří řetězec potomek – rodič – prarodič – ... až k zakladateli rodu. (Pozn.: implementace bere v potaz jen první rodičovský odkaz *Parent1*. V případě algoritmů se dvěma rodiči (GA, DE) tedy sleduje rodokmen pouze po jedné linii). Toto zjednodušení bylo zvoleno pro přehlednost vizualizace. Poté obnoví boční seznam jedinců tak, aby zobrazoval pouze vyfiltrované jedince z daného rodokmenu. V horní části listu se navíc objeví speciální tlačítko „Zobrazit všechny“, které umožňuje návrat k zobrazení celé populace.

Rodokmen se zruší buď kliknutím na zmíněné tlačítko nebo opětovným zavoláním *ShowLineage(-1)* (např. klávesou *C* pro zrušení filtru). Implementace pak odstraní zvýrazňující objekty, opět zobrazí všechny jedince a obnoví plný seznam aktuální generace.

```

/// <summary>
/// Builds a list of ancestor IDs by following Parent1 links up the lineage.
/// </summary>
/// <param name="id">Starting individual ID.</param>
/// <param name="lineage">Output list of ancestor IDs, starting with the given ID.</param>
2 references
private void ExtractLineage(int id, out List<int> lineage)
{
    lineage = new List<int>();
    var cur = FindById(id);
    while (cur != null)
    {
        lineage.Add(cur.Id);
        cur = cur.Parent1 >= 0 ? FindById(cur.Parent1) : null;
    }
}

```

Obrázek 45 Funkce postupující rodokmenem a hledající rodiče

(zdroj: vlastní)

### 3.4.3 Vykreslení optimalizačních funkcí a prostředí

K vizuálnímu zobrazení terénu funkce, na níž evoluční algoritmy hledají optimum slouží komponenty *FunctionPlotter*, *OptimizationFunctions*, *MeshGenerator* a volitelně *MeshSlicer*.

Nejmenší třída, *OptimizationFunctions*, poskytuje sadu statických metod reprezentujících známé testovací funkce: Schwefelova, Rastriginova a Ackleyho funkce. Tyto funkce jsou definovány se svými obvyklými vzorci. U každé funkce je v komentáři uveden typický definiční obor. Tyto hranice jsou důležité, protože se používají ve *FunctionPlotteru* pro nastavení domény i v *EvolutionVisualizeru* pro generování náhodné populace a omezování pohybu jedinců. Díky oddělení do samostatné třídy je přidání další testovací funkce snadné, stačí jen doplnit novou statickou metodu a případně její jméno a doménu vložit do přepínače v *FunctionPlotter.SetFunction*.

```

public static class OptimizationFunctions
{
    /// <summary>
    /// Calculates the Schwefel function. Domain: [-500, 500].
    /// </summary>
    /// <param name="x">Input value x.</param>
    /// <param name="y">Input value y.</param>
    /// <returns>Function value computed using the Schwefel formula.</returns>
    1 reference
    public static float Schwefel(float x, float y)
    {
        return 418.9829f * 2 - (x * Mathf.Sin(Mathf.Sqrt(Mathf.Abs(x))) + y * Mathf.Sin(Mathf.Sqrt(Mathf.Abs(y))));
    }

    /// <summary>
    /// Calculates the Rastrigin function. Domain: [-5.12, 5.12].
    /// </summary>
    /// <param name="x">Input value x.</param>
    /// <param name="y">Input value y.</param>
    /// <returns>Function value computed using the Rastrigin formula.</returns>
    1 reference
    public static float Rastrigin(float x, float y)
    {
        return 20 + (x * x - 10 * Mathf.Cos(2 * Mathf.PI * x)) + (y * y - 10 * Mathf.Cos(2 * Mathf.PI * y));
    }

    /// <summary>
    /// Calculates the Ackley function. Domain: [-32.768, 32.768].
    /// </summary>
    /// <param name="x">Input value x.</param>
    /// <param name="y">Input value y.</param>
    /// <returns>Function value computed using the Ackley formula.</returns>
    1 reference
    public static float Ackley(float x, float y)
    {
        float sumSq = x * x + y * y;
        float term1 = -20 * Mathf.Exp(-0.2f * Mathf.Sqrt(0.5f * sumSq));
        float term2 = -Mathf.Exp(0.5f * (Mathf.Cos(2 * Mathf.PI * x) + Mathf.Cos(2 * Mathf.PI * y)));
        return term1 + term2 + Mathf.Exp(1) + 20;
    }
}

```

Obrázek 46 Implementace optimalizačních funkcí

(zdroj: vlastní)

Další třídou, bez které by vykreslování nemohlo fungovat, je třída *FunctionPlotter*. Tato komponenta přebírá starost o vykreslení zvolené optimalizační funkce jako 3D síť a zároveň zajišťuje, aby se veškeré související komponenty (kamera, slicer, evoluční vizualizér) synchronizovaly na nový povrch.

Ve své metodě *Awake()* se nejprve inicializuje seznam položek v rozbalovacím menu *functionDropdown* (Ackley, Rastrigin, Schwefel), vytvoří se instance *MeshGenerator* a nastaví se výchozí parametry pro funkci „Ackley“. V metodě *Start()* se pak přidají posluchače UI prvků, jednorázově se vygeneruje a aplikuje mesh voláním *GenerateAndApplyMesh()*, a nakonec se upraví pozice obou kamer i případného sliceru metodou *AdjustCameraAndSlicer()*. Dropdown je navěšený v *ConfigureDropdown()* tak, že na každou změnu volby volá *OnFunctionSelected(index)*. Tato metoda:

1. Přečte název vybrané funkce.

2. Přenastaví delegát a hranice domény (*SetFunctionParameters(name)*).
3. Přegeneruje mesh (*GenerateAndApplyMesh()*).
4. Přepne kamery i slicer (*AdjustCameraAndSlicer()*).
5. A pokud je připojen *evolutionVisualizer*, zavolá na něj *ResetEvolution()*, aby se zahájila evoluce na novém terénu.

V *GenerateAndApplyMesh()* se volá *meshGenerator.GenerateMesh(currentFunction, gridSize, domainMin, domainMax, horizontalScale, verticalScale, alphaSlider.value)* a výsledek se přiřadí do *meshFilter.mesh*. Díky tomu lze měnit nejen samotnou funkci, ale i průhlednost (parametr *alpha*), sílu vertikálního či horizontálního měřítka a hustotu mřížky.

```

/// <summary>
/// Generates a mesh with the current function and applies it to the MeshFilter.
/// </summary>
3 references
private void GenerateAndApplyMesh()
{
    Mesh mesh = meshGenerator.GenerateMesh(
        currentFunction, gridSize, domainMin, domainMax,
        horizontalScale, verticalScale, alphaSlider.value);
    meshFilter.mesh = mesh;
}

```

Obrázek 47 Generování meshe

(zdroj: vlastní)

Korekce pohledu zajišťují funkce *AdjustCameraAndSlicer()* a *TeleportCameraTopDown()*, které získají světové ohraničení meshe pomocí metody *ComputeWorldBounds(meshFilter)* a předávají ho oběma kamerám (*mainCamera* a *comparisonCamera*) i sliceru (*slicer.UpdateFromExternalBounds(bounds)*). Kombinací těchto metod dokáže *FunctionPlotter*:

- dynamicky překreslovat povrch při změně funkce nebo parametrů (*gridSize*, měřítka, *alfa*),
- synchronizovat s evolučním simulátorem tak, aby se populace vždy generovala na aktuálním terénu,
- automaticky nastavovat obě kamery i slicer na optimální pozici podle velikosti grafu.

Třída MeshGenerator má pouze jediný úkol, a to ten, že zodpovídá za kompletní vytvoření trojrozměrné reprezentace zadané funkce. V zásadě se skládá z jednoho vstupního bodu, metody GenerateMesh(...), která:

1. **Odebírá vzorek hodnot** funkce na pravidelné síti pomocí SampleFunction(...). Ta projde všechny body na čtvercové mřížce o rozměru  $(gridSize+1) \times (gridSize+1)$ , spočítá pro ně hodnotu funkce  $z = func(x,y)$  a zároveň sleduje nejnižší a nejvyšší  $z$  (uloží je do *MinZ* a *MaxZ* vlastností instance).
2. **Vypočte vrcholy** (BuildVertices(...)), převede každé pravidelné  $(x,y)$  na lokální souřadnice  $[-0.5,0.5]$  a na základě normalizované výšky *normZ* vynásobí zvoleným horizontálním a vertikálním měřítkem.
3. **Vygeneruje barvy** pro jednotlivé vrcholy (BuildColors(...)) – lineární interpolací mezi červenou (nejnižší) a modrou (nejvyšší), přičemž průhlednost nastaví podle parametru *alpha*.
4. **Sestaví trojúhelníky** (BuildTriangles(...)) – každou čtvercovou buňku mřížky rozdělí na dva trojúhelníky a zapíše odpovídající indexy do pole.
5. **Sestaví konečný Mesh** (CreateMesh(...)), přiřadí mu vrcholy, trojúhelníky a barvy a nechá Unity dopočítat normály.

```

/// <summary>
/// Main entry point: generates a Mesh from the provided function over a specified domain and resolution.
/// </summary>
1 reference
public Mesh GenerateMesh(
    Func<float, float, float> func,
    int gridSize,
    float domainMin,
    float domainMax,
    float horizontalScale,
    float verticalScale,
    float alpha)
{
    float[] zValues = SampleFunction(func, gridSize, domainMin, domainMax, out float rawMinZ, out float rawMaxZ);
    MinZ = rawMinZ;
    MaxZ = rawMaxZ;

    Vector3[] vertices = BuildVertices(zValues, gridSize, domainMin, domainMax, horizontalScale, verticalScale, rawMinZ, rawMaxZ);
    Color[] colors = BuildColors(zValues.Length, zValues, rawMinZ, rawMaxZ, alpha);

    int[] triangles = BuildTriangles(gridSize);

    return CreateMesh(vertices, triangles, colors);
}

```

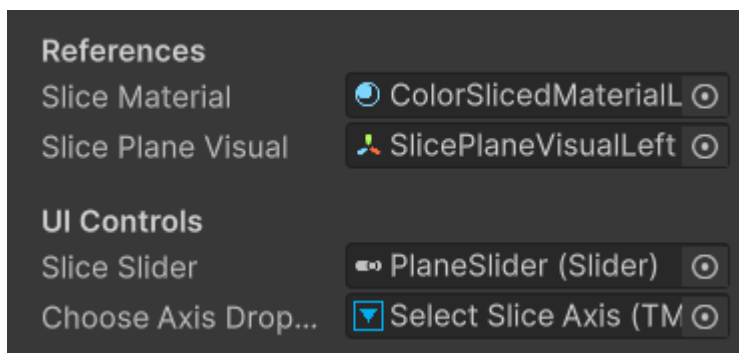
Obrázek 48 Funkce pro generování meshe

(zdroj: vlastní)

### 3.4.4 Slicing funkce

Komponenta MeshSlicer přidává doplňkovou funkci: možnost oříznout zobrazený povrch funkce určitou rovinou, a tím lépe vidět např. průběh jedinců uvnitř údolí, nebo získat 2D řez funkcí. Tato funkcionality je spíše doplňková pro detailní analýzu.

MeshSlicer (skript *MeshSlicer.cs*) využívá speciální materiál/shader, který umí skrýt části mesh na jedné straně zadané roviny (tzv. clipping plane shader). Mezi parametry patří výběr osy řezu (výčtový typ `SliceAxis { X, Y, Z }` pro řez rovinou rovnoběžnou s příslušnou osou) a slider pro nastavení polohy řezu. Dále obsahuje referenci na vizuální reprezentaci roviny, což je poloprůhledný obdélník ve scéně, ukazující kde se řeže.



Obrázek 49 Ukázka referencí pro skript řezné roviny

(zdroj: vlastní)

Hlavní logika tohoto skriptu běží v *Update()*: pokud je aktivní řezací mód, volá se metoda *ApplySlicing(float distance)*, jak je zobrazeno na obrázku 50. Ta přijímá proměnnou která udává kolik funkce chceme ořezat.

```
private void Update()
{
    if (slicingMode)
        ApplySlicing(sliceSlider.value);
}

/// <summary>
/// Applies slicing by updating shader and visual plane position.
/// </summary>
3 references
private void ApplySlicing(float distance)
{
    Vector3 normal =.GetAxisNormal();
    UpdateShader(normal, distance);
    PositionPlane(normal, distance);
}
```

Obrázek 50 Aktualizace řezací plochy

(zdroj: vlastní)

V této metodě se následně získá aktuální normála a společně s předchozím parametrem se volají *UpdateShader()* a *PositionPlane()*. V první zmíněné metodě skrývá shader na základě těchto hodnot část objektu, druhá metoda se stará o to, aby se *slicePlaneVisual* posadil do odpovídající polohy v prostoru a správně ji natočí tak, aby uživatel viděl, kudy řez prochází.

```
/// <summary>
/// Sends slicing parameters to the material shader.
/// </summary>
1 reference
private void UpdateShader(Vector3 normal, float distance)
{
    if (sliceMaterial == null) return;
    sliceMaterial.SetVector("_SlicePlaneNormal", new Vector4(normal.x, normal.y, normal.z, 0));
    sliceMaterial.SetFloat("_SlicePlaneDistance", distance);
}
```

Obrázek 51 Předání argumentů řezné roviny materiálu

(zdroj: vlastní)

Z obrázku 51 lze vidět, že se parametry předávají materiálu, respektive jeho shaderu, který je znázorněn na obrázku 52.

```

Shader "Custom/VertexColorTransparentSliced"
{
    Properties
    {
        _Color ("Color Tint", Color) = (1,1,1,1)
        _SlicePlaneNormal("Slice Plane Normal", Vector) = (0,1,0,0)
        _SlicePlaneDistance("Slice Plane Distance", Float) = 0
    }
    SubShader
    {
        Tags { "Queue"="Transparent" "RenderType"="Transparent" }
        LOD 100

        Blend SrcAlpha OneMinusSrcAlpha

        Pass
        {
            CGPROGRAM
            #pragma vertex vert
            #pragma fragment frag
            #include "UnityCG.cginc"

            float4 _Color;
            float4 _SlicePlaneNormal;
            float _SlicePlaneDistance;

            struct appdata
            {
                float4 vertex : POSITION;
                float4 color : COLOR;
            };

            struct v2f
            {
                float4 pos      : SV_POSITION;
                float4 color     : COLOR;
                float3 worldPos  : TEXCOORD0;
            };

            v2f vert (appdata v)
            {
                v2f o;
                o.pos      = UnityObjectToClipPos(v.vertex);
                o.color     = v.color * _Color;
                o.worldPos = mul(unity_ObjectToWorld, v.vertex).xyz;
                return o;
            }

            fixed4 frag (v2f i) : SV_Target
            {
                // slicing
                float d = dot(i.worldPos, _SlicePlaneNormal.xyz) - _SlicePlaneDistance;
                if (d > 0) clip(-1);

                return i.color;
            }
            ENDCG
        }
    }
}

```

Obrázek 52 Shader pro zobrazení roviny řezu

(zdroj: vlastní)

### 3.4.5 Graf průběhu fitness

Na UI je také panel s grafem fitness v závislosti na generaci (soubor *GraphDisplay.cs*). Ten slouží k monitorování vývoje nejlepší a průměrné hodnoty fitness během evoluce. Graf je implementován vykreslováním do textury (Texture2D), která je zobrazena v komponentě RawImage. Třída obsahuje datovou strukturu *List<DataPoint>* pro uložené body, kde každý bod má číslo generace, best a avg fitness.

```

5 references | 3 references
private struct DataPoint { public int generation;
                           4 references | 4 references
                           public float best, average;
                           1 reference
                           public DataPoint(int g, float b, float a)
                           { generation = g; best = b; average = a; }
}
12 references
private readonly List<DataPoint> dataPoints = new List<DataPoint>();

```

Obrázek 53 Hlavní struktura třídy s grafem

(zdroj: vlastní)

Při inicializaci je navěšen listener na přepínací tlačítko (*toggleButton*) pro schování/zobrazení grafu, případně můžeme klávesou *G* docílit stejného efektu. Hlavní třída *EvolutionVisualizer* přidává data do grafu voláním *fitnessGraph.AddFitnessData(currentGen, best, avg)* vždy, když vykreslí novou generaci. Metoda *AddFitnessData()* v grafu jen přidá bod do listu a zavolá *RefreshGraph()*.

*RefreshGraph()* obstará veškeré překreslení grafu:

- **Přizpůsobí velikost textury:** v *EnsureTexture()* se zjistí, zda rozměry *graphImage.rectTransform* odpovídají velikosti interní Texture2D. Pokud ne, vytvoří se nová Texture2D přesně ve velikosti panelu, aby se graf neroztahoval nekorektně.
- **Vyčistí pozadí:** metoda *ClearTexture()* naplní každý pixel barvou *backgroundColor*, čímž zajistí bílé pozadí před dalším kreslením.
- **Vykreslí mřížku a popisky os:** *DrawGrid()* spočítá, kde mají vést svislé a vodorovné čáry podle *verticalGridCount* a *horizontalGridCount*, a vykreslí je pomocí *DrawLine()*. Následně *DrawXAxisLabels()* a *DrawYAxisLabels()* rozmístí textové štítky generací pod osu X a hodnot fitness pod osu Y.

- **Spojí body fitness do křivek:** pokud jsou v seznamu *points* alespoň 2 prvky, *ComputeDataBounds()* určí *minFitness*, *maxFitness* (s rezervou +1, když jsou stejné) a maximální generaci. Pro každé dva sousední body se pak volá *PointToPixel()* kvůli převodu datových souřadnic na pixely a výsledné body se propojí *DrawLine()* dvěma barvami: *bestLineColor* pro nejlepší fitness a *averageLineColor* pro průměrnou.
- **Aplikuje texturu a aktualizuje legendu:** po dokončení kreslení se zavolá *graphTexture.Apply()* aby se změny projeví, a *graphImage.texture* se na ni přepne. V *UpdateLegend()* se pak do *bestLegendText* a *averageLegendText* vloží aktuální hodnoty a zbarví se odpovídajícími barvami.

Graf se tak při každém volání *RefreshGraph* celý překreslí. Uživatel vidí v grafu dvě křivky: zelenou klesající a modrou klesající (u minimalizace), které se často postupně sbližují. To doplňuje 3D vizualizaci o kvantitativní pohled.

```

/// <summary>
/// Main update routine: prepares texture, draws grid and data, and updates UI.
/// </summary>
1 reference
private void RefreshGraph()
{
    EnsureTexture();
    ClearTexture();
    DrawGridLines();

    if (dataPoints.Count > 1)
    {
        ComputeDataBounds(out float minF, out float maxF, out int maxGen);
        PlotFitnessLines(maxGen, minF, maxF);
        UpdateAxisLabels(maxGen, minF, maxF);
    }
    else
    {
        ClearLabels(xAxisLabelContainer);
        ClearLabels(yAxisLabelContainer);
    }

    ApplyTexture();
    UpdateLegends();
}

```

Obrázek 54 Překreslování grafu

(zdroj: vlastní)

### 3.4.6 Režim porovnání

Dalším zajímavým prvkem je *CameraModeManager* a s ním související režim *Single vs. Split*. Tato část umožňuje, aby aplikace zobrazovala dvě různé simulace vedle sebe současně. Představme si případ, kdy chceme porovnat dva algoritmy na stejné funkci, nebo stejný algoritmus na dvou různých funkcích. Přesně k tomuto účelu slouží split-screen.

*CameraModeManager* (skript *CameraModeManager.cs*) spravuje tři kamery: *mainCamera* pro běžný mód a *leftCamera* a *rightCamera* pro levý a pravý pohled při split režimu. Dále má reference na *CameraControllery*, které zajišťují ovládání dvou pohledů nezávisle. Také obsahuje reference na dva *FunctionPlottery* a jejich grafy fitness. Enum *Mode* určuje aktuální režim (Single/Split) a UI Toggle *splitToggle* je propojen s volbou režimu.

```
public class CameraModeManager : MonoBehaviour
{
    [Header("Cameras & Controllers")]
    2 references
    public Camera mainCamera;
    2 references
    public Camera leftCamera;
    2 references
    public Camera rightCamera;

    // Přidané reference na controllery
    4 references
    public CameraController mainCameraController;
    4 references
    public CameraController leftCameraController;
    4 references
    public CameraController rightCameraController;

    0 references
    public GameObject leftGraph;
    2 references
    public GameObject rightGraph;

    2 references
    public Button rightCamFrontButton;
    2 references
    public Button rightCamUpButton;

    [Header("Plotters (pro Focus)")]
    4 references
    public FunctionPlotter functionPlotterLeft;
    2 references
    public FunctionPlotter functionPlotterRight;

    1 reference
    public MenuDropdownController menuDropdownController;

    6 references | 3 references | 2 references
    public enum Mode { Single, Split }
    3 references
    public Mode currentMode = Mode.Single;
    10 references
    public Toggle splitToggle;
    1 reference
    public GameObject deactivatedPanel;
}
```

Obrázek 55 Komponenty související s rozděleným režimem

(zdroj: vlastní)

V metodě `Start()` se navěsí listener na `splitToggle` (`onValueChanged.AddListener(SetSplitMode)`) a zavolá `ApplyMode()`, aby se podle výchozího nastavení připravily kamery. Je ošetřena i klávesová zkratka `B`, která přepíná režim (simuluje změnu toggle). `SetSplitMode(bool isSplit)` z enum hodnoty nastaví `currentMode` a volá `ApplyMode()`. Ta pak konkrétně nastaví:

- Aktivaci/deaktivaci kamer: v single módu běží pouze `mainCamera`, ve split módu se aktivují `leftCamera` a `rightCamera` a hlavní se vypne.
- UI přizpůsobení: volá `menuDropdownController.SetDropdown(split)`. To je funkce, která zobrazuje ovládací prvky pro dva režimy. Konkrétně řečeno, pokud je uživatel v benchmark módu, jsou v rozbalovacím menu o dvě tlačítka navíc pro export a import pravé části.
- Nastavení viewportů kamer: `mainCamera` dostane rect (0,0,1,1) v single módu, naopak ve split módu se `leftCamera` nastaví na levou půlku obrazovky (0,0 do 0.5,1) a `rightCamera` na pravou polovinu (0.5,0 do 1,1). Tím Unity vykreslí dva pohledy vedle sebe. Dále se ve split módu zapne `rightGraph` panel a aktivují se tlačítka pro změnu pohledu pravé kamery, zatímco v single módu jsou tyto moduly samozřejmě vypnuté.
- Předá informaci o režimu `CameraControllerům` (volá `SetSplitMode(split)` na každém). Kontroler kamery následně upravuje např. rychlost rotace a margin při dvou panelech (aby se orbit ovládal nezávisle v každé polovině).
- Přenastaví kamery na focus jejich funkcí: ve split módu zavolá `FocusOnBounds()` pro `leftCameraController` s `Bounds` levého plotteru a analogicky pro pravý. V single módu použije `mainCameraController` a levý plotter. To zajistí, že obě kamery budou správně namířeny na své grafy funkcí.

```

private void ApplyMode()
{
    bool split = (currentMode == Mode.Split);

    if (splitToggle != null && splitToggle.isOn != split)
        splitToggle.isOn = split;

    mainCamera.gameObject.SetActive(!split);
    leftCamera.gameObject.SetActive(split);
    rightCamera.gameObject.SetActive(split);

    menuDropdownController.SetDropdown(split);
    deactivatedPanel.gameObject.SetActive(!split);
    ...

    if (!split)
    {
        rightGraph.SetActive(false);
        rightCamFrontButton.gameObject.SetActive(false);
        rightCamUpButton.gameObject.SetActive(false);
        mainCamera.rect = new Rect(0, 0, 1, 1);
    }
    else
    {
        rightGraph.SetActive(true);
        rightCamFrontButton.gameObject.SetActive(true);
        rightCamUpButton.gameObject.SetActive(true);
        leftCamera.rect = new Rect(0f, 0f, 0.5f, 1f);
        rightCamera.rect = new Rect(0.5f, 0f, 0.5f, 1f);
    }

    if (mainCameraController != null) mainCameraController.SetSplitMode(split);
    if (leftCameraController != null) leftCameraController.SetSplitMode(split);
    if (rightCameraController != null) rightCameraController.SetSplitMode(split);

    if (split)
    {
        if (leftCameraController != null && functionPlotterLeft != null)
        {
            Bounds lb = CalculateWorldBounds(functionPlotterLeft);
            leftCameraController.FocusOnBounds(lb);
        }
        if (rightCameraController != null && functionPlotterRight != null)
        {
            Bounds rb = CalculateWorldBounds(functionPlotterRight);
            rightCameraController.FocusOnBounds(rb);
        }
    }
    else
    {
        if (mainCameraController != null && functionPlotterLeft != null)
        {
            Bounds mb = CalculateWorldBounds(functionPlotterLeft);
            mainCameraController.FocusOnBounds(mb);
        }
    }
}

```

Obrázek 56 Ukázka metody ApplyMode() ve správci kamer

(zdroj: vlastní)

Celkově to znamená, že program má dvě instance každého skriptu (kromě *CameraModeManager*), které byly doposud zmíněny v této kapitole. Tento „multi“ režim umožňuje například spustit vlevo GA na Ackleyho funkci, vpravo PSO na Ackleyho funkci, a vizuálně je porovnávat vedle sebe. Případně obě se stejným algoritmem na dvou různých funkcích.

### 3.4.7 Podpůrné třídy a import/export dat

V závěrečné části aplikační logiky se nachází pomocné třídy, které spolu utvářejí kompletní uživatelské rozhraní a zajišťují udržitelné ukládání i načítání výsledků simulací. Společně tvoří podpůrnou vrstvu, bez které by byla práce s nástrojem méně pohodlná, méně interaktivní a náchylná k chybám.

První z těchto tříd je statický helper *Export*, jehož úkolem je serializovat a deserializovat kompletní stav evoluční simulace do/z JSON souboru. Při ukládání se vytvoří výměnný datový objekt *AllData*, který obsahuje název algoritmu, název testovací funkce, seznam generací s jednotlivými jedinci a jejich rodiči, statistická data o fitness a naposledy zaznamenané pozice všech jedinců.

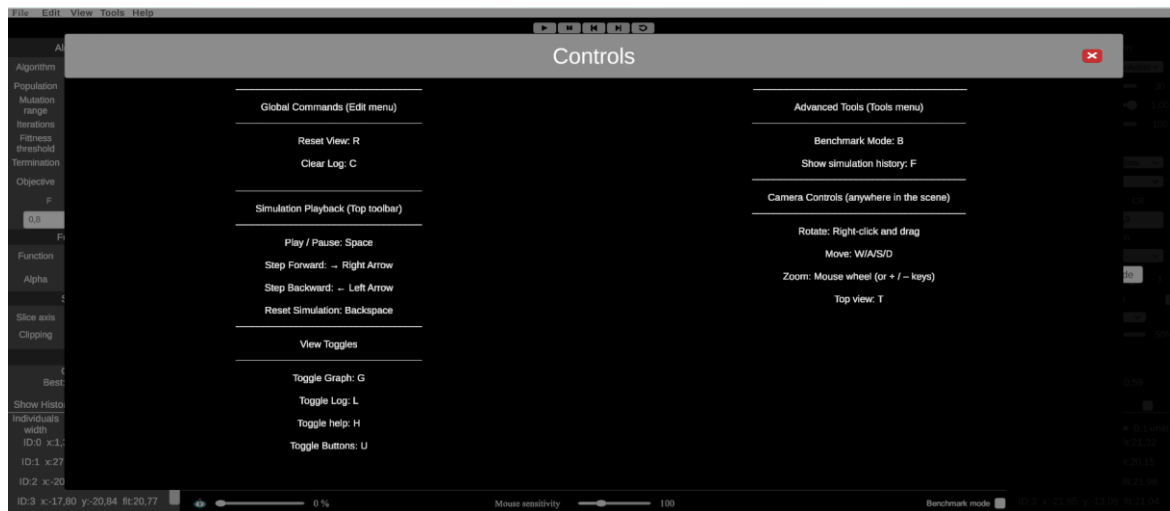
```
[Serializable]
3 references
public class AllData
{
    2 references
    public string AlgorithmName;
    2 references
    public string FunctionName;
    2 references
    public List<GenerationGroup> Generations;
    2 references
    public List<GenerationDataSerializable> GenerationData;
    2 references
    public List<PositionEntry> PositionHistory;
}
```

Obrázek 57 Třída sloužící k ukládání/načítání JSON souborů

(zdroj: vlastní)

Pomocí metody *JsonUtility.ToJson()* se tento DTO zapíše do souboru. Při načítání se načtené JSON převede zpět na příslušné struktury a na základě nich se rekonstruuje historie generací i trajektorie jednotlivců, včetně obnovení správného čítače ID, aby každý nahraný jedinec dostal unikátní identifikátor.

Další třídou je *HelpManager*, která spravuje panel s vysvětlivkami. Této třídě se přiřazují 2 tlačítka, „open“ a „close“. Třída také naslouchá stisku klávesy H, čímž uživatel rychle přepíná viditelnost panelu.

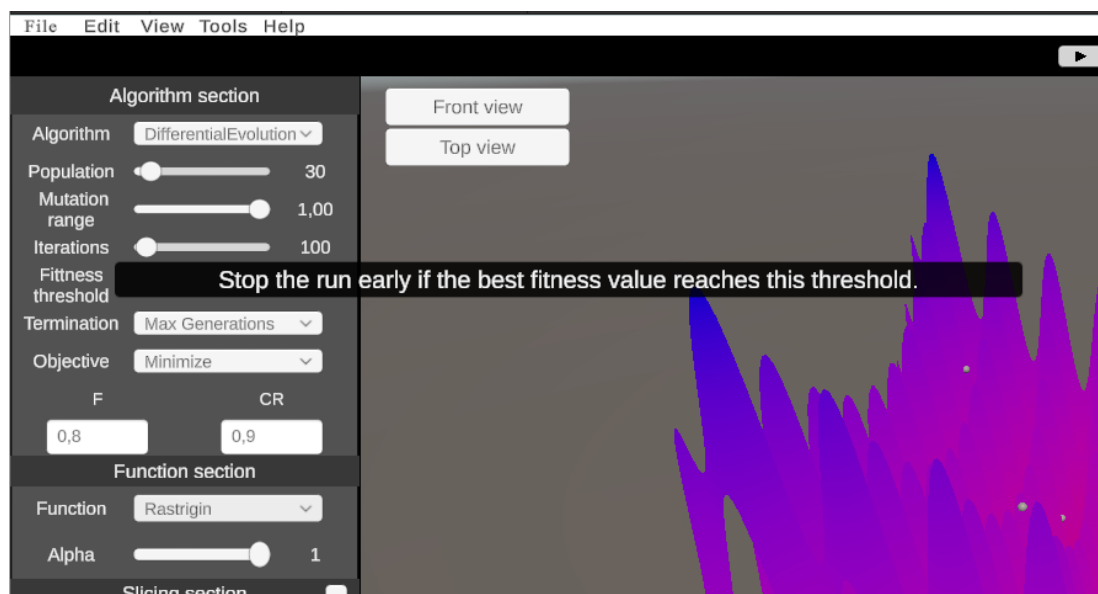


Obrázek 58 Ukázka klávesových zkratk

(zdroj: vlastní)

Třetí komponentou je *MenuDropdownController*, která zajišťuje korektní chování rozbalovacích nabídek v horním menu. Každý takový controller se při *Awake()* zaregistruje do sdíleného seznamu, aby bylo možné řídit viditelnost všech dropdownů najednou. Když uživatel najede myší (*OnPointerEnter*) nebo klikne (*OnPointerClick*), aktivuje se jeho vlastní nabídka a všechny ostatní se skryjí, čímž se zabrání situaci, kdy by se na obrazovce otevřelo více rozbalovacích seznamů současně. Pokud je controller deaktivován, sám se odregistruje, čímž nedojde k úniku paměti ani k nesprávným referencím *MenuDropdownController*.

Dalším prvkem je *TooltipManager*, který poskytuje zobrazení popisků k jakékoli UI komponentě, ať už jde o tlačítko, slider nebo prvek menu. Využívá předlohu *tooltipPrefab*, kterou instancuje do plátna aplikace. Když uživatel najede na prvek, *TooltipTriggerUI* vyvolá metodu *Show(message)*, která s drobným zpožděním zajistí, že se tooltip zobrazí, přizpůsobí svou velikost podle obsahu a v závislosti na poloze kurzoru „flipne“ ukotvení, aby zůstal celý viditelný. Po opuštění prvku se tooltip skryje.



Obrázek 59 Ukázka tooltipu

*(zdroj: vlastní)*

Poslední třída *TooltipTriggerUI* slouží jako jednoduchý spouštěč pro zobrazení tooltipu. Každý UI prvek, kterému chceme tooltip přiřadit, stačí opatřit tímto skriptem a vyplnit vlastnost *tooltipText*. Implementace *IPointerEnterHandler* a *IPointerExitHandler* zajišťuje, že při najetí myši se pomocí singletonu *TooltipManager.Instance.Show* aktivuje zobrazení a při odjetí zase zmizí.

## 4 TESTOVÁNÍ A MODULARITA

Při vývoji aplikace nejde jen o to implementovat jednotlivé komponenty systému, ale také zajistit jejich stabilní a správné fungování v různých scénářích. Vzhledem k tomu, že aplikace kombinuje matematickou logiku, interaktivní uživatelské rozhraní a vizualizaci ve 3D prostoru, bylo testování důležité pro ověření správnosti chování a odstranění potenciálních chyb.

### 4.1 Testování a ladění

Testování probíhalo převážně manuálně v prostředí editoru Unity, přičemž se kombinovaly následující přístupy:

- testování po jednotlivých modulech (modulární testování tříd),
- interaktivní testování v reálném čase,
- vizuální kontrola výsledků ve scéně,
- kontrola výpisů do konzole (Debug.Log).

#### 4.1.1 Testování vizualizace funkcí

První část testování byla zaměřena na správnou interpretaci optimalizačních funkcí a jejich prostorové vizualizace. Nejprve bylo ověřeno, zda jsou hodnoty jednotlivých funkcí počítány správně v rámci definovaného intervalu, například u Ackleyho funkce v rozsahu  $[-32.768; 32.768]$ . Dále bylo potřeba zkontrolovat, že síť (Mesh) reprezentující povrch funkce je generována bez viditelných deformací a že během triangulace nedochází k přehazování vrcholů, což by vedlo ke vzniku nesprávných tvarů v síti. Zvláštní pozornost byla věnována také správnosti barevného gradientu, který by měl odpovídat výškovým hodnotám funkce, tedy aby nízké hodnoty byly znázorněny například tmavšími barvami a vyšší hodnoty světlejšími. Nakonec bylo prověřeno, že při přepnutí optimalizační funkce v uživatelském rozhraní se korektně aktualizuje název funkce a rozsah její domény, aby odpovídal nově vybrané funkci.

**Chyby a úpravy:** v rané verzi docházelo k problémům se zobrazováním funkce, která se vůbec nechtěla zobrazovat. Normalizace hodnot tento problém vyřešila. Zároveň docházelo k chaotickému zobrazování barev u některých funkcí. Problém byl způsoben chybou právě v normalizaci hodnot, která nezohledňovala aktuální minimum. Úpravou výpočtu ve třídě *MeshGenerator* se situace stabilizovala.

#### 4.1.2 Testování evolučních algoritmů

Pro každý z implementovaných algoritmů byl ověřován následující scénář:

1. **Inicializace populace** – správný počet jedinců, pozice uvnitř definičního oboru, fitness odpovídající hodnotě funkce.
2. **Posun mezi generacemi** – po stisknutí tlačítka „Next“ musí být generována nová populace, s odlišnými pozicemi a fitness.
3. **Reakce na změnu funkce** – výpočet musí probíhat pro nově zvolenou funkci a její definiční obor.
4. **Správná implementace logiky** – např. u GA bylo ověřováno, že dochází k mutaci a křížení a se správnými parametry, u DE že jsou vybírání různí jedinci apod.
5. **Konzistence mezi generacemi** – při přechodu zpět na předchozí generaci musí být data zachována beze změny.

**Příklad testovacího postupu u Diferenciální Evoluce:** bylo testováno chování diferenciální evoluce (DE) v situaci, kdy parametr mutace (mutation factor  $F$ ) nabýval extrémních hodnot (např.  $F = 0$  nebo  $F = 2$ ).

Při nastavení  $F = 0$  nedocházelo k žádné skutečné mutaci a nové pozice byly tvořeny pouze rekombinací stávajících jedinců. To vedlo k rychlé konvergenci celé populace do jednoho bodu – potvrzující absenci diverzity a extrémní formu exploitačního chování.

Naopak při vysoké hodnotě  $F$  (např.  $F = 2$ ) se jedinci často dostávali mimo definiční obor. Díky implementované ořezávací funkci (*Mathf.Clamp*) byly však tyto hodnoty korektně vráceny na hranice domény. Tím byla ověřena robustnost algoritmu vůči extrémním hodnotám parametru  $F$  a správné ošetření hraničních podmínek.

**Zachycené chyby během testování algoritmů:** jedna z chyb se vyskytovala v algoritmu PSO. Tam nebyly při přepnutí algoritmu správně resetovány vnitřní datové struktury uchovávající rychlosti částic a jejich osobní nejlepší pozice a fitness (*Dictionary<int, Vector2> velocities, personalBestPos, personalBestFit*). Výsledkem bylo, že při přepnutí algoritmu zpět na PSO zůstávaly ve slovnících staré hodnoty z předchozí simulace, což vedlo k chybnému výpočtu nových pozic. Problém byl vyřešen přidáním metody

*ForceClearState()*, která explicitně maže tyto slovníky při každém přepnutí nebo restartu algoritmu.

Další jednoduchá chyba byla odhalena u diferenciální evoluce (DE). V jedné z předchozích verzí nebyla správně ošetřena možnost, že náhodně vybraní jedinci  $a$ ,  $b$  a  $c$  pro výpočet donorového vektoru mohli být totožní, což vedlo ke generování nulového rozdílu a tím i ke stagnaci vývoje. Chyba byla odstraněna zpřísněním výběru těchto indexů tak, aby vždy platilo  $a \neq b \neq c \neq i$ .

#### 4.1.3 Testování řezů

Při testování funkce řezů povrchů (tzv. slicing) byly ověřovány různé kombinace zobrazení 3D funkcí ve dvou panelech. Hlavním cílem bylo zajistit, aby každá funkce mohla být nezávisle řezána pomocí vlastní roviny a aby výpočet hodnot pro řezy odpovídal skutečnému tvaru funkce.

Na začátku implementace došlo k chybě, kdy byl pro oba grafy použit jeden sdílený materiál, v němž byly hodnoty řezové roviny zapisovány globálně. To způsobovalo, že při aktivaci řezu na jedné straně se zároveň vizuálně ořezávala i druhá funkce, což neodpovídalo očekávanému chování. Problém byl následně vyřešen vytvořením oddělených instancí materiálu pro každý *MeshRenderer*, čímž bylo dosaženo plné nezávislosti řezových parametrů.

Další komplikace nastaly při převodu hodnot mezi normalizovaným rozsahem a reálným definičním oborem u některých funkcí. Pokud byla například optimalizační funkce definována v rozmezí 0 až 1, zobrazovala se řezová rovina nesprávně v oblasti -0.5 až 0.5. Tento problém vznikl kvůli chybnému výpočtu transformace souřadnic v shaderu a také chybou ve výpočtu. Problém byl opraven zavedením správného přepočtu mezi reálnými a normalizovanými souřadnicemi, který zohledňuje aktuální rozsah definičního oboru dané funkce.

#### 4.1.4 Testování uživatelského rozhraní

Další část testování se zaměřila na interaktivní prvky, jejich správné přiřazení a logiku chování. Ověřovaly se následující funkce:

- Dropdown menu algoritmů a funkcí – přepnutí vždy musí spustit novou vizualizaci.

- Tlačítka (Start, Pause, Next, Previous, Reset) – každé musí správně reagovat a mít jasný efekt ve scéně.
- Posuvník rychlosti – plynule mění rychlost simulace.
- Menu bar – otevírá se při kliknutí na dané menu, zůstává responzivní na různých rozlišeních.
- Panel grafu – aktualizace čar, zobrazení legendy, správná osa generací a fitness.

**Odhalené problémy:** jedním z problémů bylo chování tlačítek pro manuální ovládání generací „Next“ a „Previous“ v kombinaci s aktivním automatickým během simulace. Pokud uživatel během automatického přehrávání generací klikl na jedno z těchto tlačítek, docházelo k nežádoucímu chování – například k přeskočení více generací najednou, chybné synchronizaci grafu nebo k přerušení plynulosti simulace. Problém vznikl kvůli tomu, že paralelně běžely dvě logiky, tedy ta automatická a manuální, které nebyly správně odděleny. Řešením bylo pouze upravit řídicí logiku. Další chyba se týkala dropdown menu pro výběr algoritmů. Po přepnutí algoritmu zůstávaly některé komponenty starého výpočtu viditelné (jedinci v prostoru, trajektorie), protože nebyly správně zničeny. Problém byl vyřešen doplněním volání metody *ClearIndividuals()* a úplným resetem stavu vizualizace při každé změně algoritmu nebo funkce.

V panelu s grafem fitness se vyskytla chyba při aktualizaci osy generací. Při opakovaném spuštění simulace od první generace nebyla osa vynulována a zobrazovala nesprávný rozsah. Bylo nutné zavést explicitní reset proměnných sledujících průběh a správné přepočítání rozsahu při každém *ResetEvolution()*.

## 4.2 Modularita a rozšiřitelnost systému

Modularita systému představuje princip, který byl při návrhu celé aplikace důsledně zohledňován. Cílem nebylo vytvořit pouze jednorázovou vizualizaci několika algoritmů, ale vybudovat strukturu, která umožní snadné rozšiřování ať už o nové evoluční strategie, optimalizační funkce, vizualizační prvky, nebo funkce uživatelského rozhraní. Díky tomu může být projekt v budoucnu upraven nejen pro pokročilé uživatele, ale také například pro účely výuky, výzkumu či jako základ pro vlastní experimenty.

#### 4.2.1 Architektonický přístup

Architektura systému je navržena tak, aby klíčové výpočetní komponenty zůstaly co nejvíce oddělené od vizualizační a uživatelské vrstvy. Tento princip je nejvýrazněji patrný v použití rozhraní *IEvolutionAlgorithm*, které sjednocuje způsob, jakým jsou jednotlivé evoluční algoritmy volány, a umožňuje jejich snadnou záměnu bez nutnosti zásahů do hlavního řídicího skriptu. Rovněž předávání optimalizační funkce prostřednictvím delegátu *Func<float, float, float>* zajišťuje, že algoritmy mohou fungovat nad libovolnou funkcí bez závislosti na její konkrétní implementaci.

Přestože některé komponenty (například *EvolutionVisualizer*, *FunctionPlotter* a uživatelské rozhraní) spolupracují velmi úzce a jsou propojeny pomocí přímých referencí, jedná se o záměrné rozhodnutí, které odpovídá specifikům vývoje v Unity. Díky tomuto přístupu bylo možné zachovat srozumitelnou strukturu projektu, jednoduše propojit logiku s vizualizací a zároveň zajistit vysokou interaktivitu celé aplikace. Vzájemná vazba těchto částí tak není na překážku modularitě systému, ale spíše usnadňuje jeho rozšiřování a údržbu při zachování plné kontroly nad chováním simulace.

#### 4.2.2 Přidání nového evolučního algoritmu

Díky použití rozhraní *IEvolutionAlgorithm* je přidání nového algoritmu otázkou implementace jediné veřejné metody: *List<Individual> ComputeNextGeneration(...)*. Stačí vytvořit novou třídu dědící ze *ScriptableObject*, implementovat uvedenou metodu a uložit skript do složky *Evolution Algorithms*. Algoritmus se následně automaticky objeví v nabídce uživatelského rozhraní, protože dropdown načítá všechny objekty s tímto rozhraním automaticky.

#### Ukázkové možnosti pro rozšíření:

- Simulated Annealing,
- NSGA-II (vícekriteriální optimalizace),
- CMA-ES (evoluční strategie s kovariancí),
- nebo neuroevoluční metody.

#### 4.2.3 Přidání nové optimalizační funkce

Přidání nové funkce do systému je ještě jednodušší. Ve statické třídě *OptimizationFunctions* stačí deklarovat novou veřejnou metodu:

```
public static float Rosenbrock(float x, float y) {  
    return 100 * Mathf.Pow((y - x * x), 2) + Mathf.Pow((1 - x), 2);  
}
```

Funkce se poté jednoduše připojí do dropdownu ve *FunctionPlotter*, čímž se stává okamžitě použitelnou v rámci vizualizace i optimalizace. Doménu funkce lze upravit podle potřeby (domainMin, domainMax) a výsledná síť se automaticky regeneruje.

#### 4.2.4 Rozšiřitelnost a návrh uživatelského rozhraní

Uživatelské rozhraní aplikace bylo navrženo s ohledem na přehlednost, intuitivní ovládání a snadné rozšiřování. Základní architektura je postavena na systému samostatných panelů v rámci komponenty Canvas, a ovládacích postranních panelů (viz kapitola 3.2). Toto rozvržení vychází z osvědčených principů používaných v běžných vývojových nástrojích, jako jsou Unity, Visual Studio nebo Blender, a umožňuje uživatelům rychlou orientaci i bez nutnosti rozsáhlého školení.

Pro samotné sestavení uživatelského rozhraní byl zvolen klasický systém Unity UI (uGUI), který nabízí širokou kompatibilitu se stávajícími komponentami Unity, vysokou flexibilitu při návrhu a lepší podporu runtime aplikací oproti novějšímu systému UI Toolkit. Díky tomu bylo možné efektivně integrovat ovládací prvky jako tlačítka, rozevírací seznamy, slidery a textová pole a snadno je spravovat pomocí systému prefabů.

Rozhraní bylo navrženo tak, aby bylo možné bez zásadních zásahů přidávat nové ovládací prvky. Do budoucna by se určitě hodily přidat možnosti checkboxů pro zapínání a vypínání různých vlastností, jako je mutace, a rozšíření horního menu o nové kategorie. Dynamickou správu položek v menu, o které jsme se bavili podrobněji v kapitole 3.4.7, zajišťuje komponenta *MenuDropdownController*, která umožňuje přidávat a spravovat nové položky bez nutnosti měnit základní strukturu UI.

Další rozšíření by mohlo spočívat v umožnění definice vlastních optimalizačních funkcí uživateli, například prostřednictvím textového zadání funkčního výrazu s využitím parseru *ExpressionEvaluator*. Modularitu by rovněž zvýšilo zavedení samostatného rozhraní pro výběrové strategie, například *ISelectionStrategy*, které by genetickému algoritmu umožnilo snadné přepínání mezi různými metodami selekce, jako je turnajový výběr nebo ruletový výběr.

Při návrhu uživatelského rozhraní byl kladen důraz na použitelnost a přístupnost. Byly uplatněny následující principy:

- Minimalistický design – každý panel obsahuje pouze nezbytné ovládací prvky, aby se předešlo zahlcení uživatele nadbytečnými informacemi.
- Vhodné barevné schéma – vizualizace fitness hodnot využívá barevné gradienty pro lepší čitelnost, zatímco samotné UI je laděno do neutrálních tónů, které neodvádějí pozornost od hlavní scény.
- Primární ovládání myši – celé rozhraní je optimalizováno pro ovládání pomocí myši, s možností budoucího rozšíření o klávesové zkratky nebo dotykové ovládání.

Uživatelské rozhraní je rovněž plně responzivní díky využití systémů anchorů a automatického rozvržení v Unity. Jednotlivé panely se tak správně přizpůsobují změnám velikosti okna, což zajišťuje zachování použitelnosti jak na menších obrazovkách, například u notebooků, tak i na větších monitorech s vyšším rozlišením.

## 5 VYUŽITÍ A DOKUMENTACE

Aplikace byla navržena jako univerzální vizualizační nástroj pro evoluční algoritmy, který najde uplatnění jak ve vzdělávání, tak i při vývoji a testování nových metod či při prezentacích zaměřených na umělou inteligenci a optimalizaci.

### 5.1 Možnosti reálného nasazení a scénáře použití

Díky své flexibilní architektuře a názorné vizualizaci je systém vhodný pro široké spektrum scénářů. Může být součástí výuky na školách, nebo může být využit pro experimentální analýzu algoritmů nebo jejich demonstraci v rámci odborných i popularizačních akcí. Následující podkapitoly popisují konkrétní příklady jeho využití v praxi.

#### 5.1.1 Využití ve vzdělávání

Jedním z nejpřirozenějších scénářů nasazení této aplikace je výuka evolučních algoritmů na středních a vysokých školách. V této oblasti nabízí nástroj několik výhod:

- 1) Interaktivita – studenti mohou přímo sledovat, jak se populace vyvíjí, mění rozložení řešení a jak parametry ovlivňují chování algoritmu.
- 2) Okamžitá zpětná vazba – vizualizace reaguje v reálném čase na změny vstupních hodnot, což zvyšuje pochopení probíhajících procesů.
- 3) Srovnání metod – možnost přepínat mezi více algoritmy (např. GA, PSO, DE) a funkcemi a zároveň je sledovat vedle sebe umožňuje vnímat rozdíly v jejich chování bez nutnosti čtení kódu.

Aplikace může být využita například v kurzech jako:

- Umělá inteligence
- Bio-inspirované výpočty
- Algoritmizace a optimalizace
- Základy programování a vizualizace dat

#### 5.1.2 Prezentace a demonstrace

Aplikace je dobře uzpůsobena i pro prezentační a demonstrační účely, například při popularizačních akcích, dnech otevřených dveří nebo jako vizuální doplněk přednášek. Umožňuje jednoduchým způsobem ukázat fungování evolučního výpočtu, způsob hledání

řešení v komplikovaném prostoru a vliv nastavení parametrů na dosažené výsledky. Díky 3D vizualizaci a barevnému značení působí aplikace srozumitelně i pro laické publikum a může sloužit jako atraktivní ukázka principů umělé inteligence.

### **5.1.3 Budoucí rozšíření pro reálná data**

Přestože aplikace v současnosti pracuje s matematickými funkcemi definovanými explicitně, nic nebrání rozšíření systému o podporu reálných dat. Například import výškových dat a hledání optimální cesty (pro robotiku, navigaci), nebo pro optimalizaci parametrů neuronových sítí. Díky tomu může být aplikace použita jako nástroj pro rychlou vizualizaci optimalizačních problémů, které by jinak zůstaly pouze v abstraktní podobě.

## **5.2 Detailní dokumentace uživatelského scénáře**

Aby bylo možné lépe porozumět fungování celé aplikace z pohledu uživatele, je v této kapitole popsán konkrétní scénář použití. Ten ilustruje typický průchod aplikací od jejího spuštění až po analýzu výsledků, přičemž zachycuje jak funkční interakce, tak vizuální a interaktivní odezvu systému. Cílem je demonstrovat dostupné možnosti, logiku ovládání a vazby mezi jednotlivými částmi rozhraní. Tato dokumentace slouží jako jednoduchý souhrn všech informací, které byly prozatím sděleny, do jednoduchého postupu pro uživatele, kteří si práci nečetli, ale chtějí si aplikaci odzkoušet.

### **1) Spuštění aplikace a inicializace**

Po spuštění aplikace dojde k automatickému načtení výchozí benchmarkové funkce (Ackley) a vygenerování její 3D vizualizace pomocí mesh sítě. Současně je inicializována počáteční populace jedinců, kteří jsou rozmístěni náhodně v definičním oboru a vizuálně zobrazení jako barevné body na povrchu funkce. Kamera se automaticky přenastaví na výchozí pohled, který poskytuje celkový přehled scény.

### **2) Volba optimalizační funkce**

Uživatel může v levém panelu zvolit jednu z předdefinovaných funkcí: Ackley, Rastrigin nebo Schwefel. Po výběru se síť okamžitě přegeneruje podle nové funkce a jejího definičního oboru. Zároveň se resetuje evoluce, tedy dojde k vygenerování nové populace a zároveň se vymaže graf. Funkce a její jméno se aktualizují v UI, aby nedošlo k nejasnostem při přepínání.

### **3) Výběr evolučního algoritmu**

Druhý dropdown v levém panelu umožňuje přepnutí mezi algoritmy:

- Genetic Algorithm (GA)
- Differential Evolution (DE)
- Particle Swarm Optimization (PSO)
- Evolution Strategy (ES)
- SOMA

Při změně algoritmu zůstává aktuální generace zachována, ale při výpočtu další generace je již použita nová metoda. Každý algoritmus implementuje stejné rozhraní, což umožňuje jejich snadnou výměnu. Při přepnutí je zajištěn správný reset vnitřního stavu, aby nedocházelo ke zkreslení výsledků.

#### 4) Nastavení parametrů evoluce

Uživatel má k dispozici několik interaktivních ovládacích prvků pro konfiguraci simulace:

- **Populace:** nastavuje počet jedinců ve výpočtu.
- **Mutate / křížení / migrační krok:** parametry specifické pro daný algoritmus.
- **Rychlost simulace:** ovládá interval automatického přepínání generací.
- **Maximální počet generací:** stanovuje ukončovací podmínku. Všechny parametry lze měnit dynamicky a změna se ihned promítne do chování algoritmu.
- **Možnost ukončení:** uživatel si může zvolit, zda chce skončit po dosažení určité hodnoty fitness nebo po dosažení určité generace.
- **Další parametry specifické pro daný algoritmus**

#### 5) Řízení simulace

Simulaci lze řídit pomocí tlačítek:

- **Start** – spustí automatické přehrávání generací,
- **Pause** – pozastaví výpočet,
- **Next / Previous** – přepínání po jedné generaci,
- **Reset** – vynuluje simulaci a vygeneruje novou populaci.

Každý nový krok (ruční i automatický) zapisuje výstupy do grafu průběhu fitness, který zobrazuje průměrnou i nejlepší hodnotu v každé generaci.

## **6) Interakce s jedinci**

Kliknutím na kteréhokoli vizualizovaného jedince se v panelu zobrazí jeho ID, souřadnice a fitness. Zároveň se zvýrazní jeho celá trajektorie od první generace. Zvýraznění představuje barevná křivka, která pomáhá analyzovat, jak se daný jedinec vyvíjel. Tato funkce využívá historii pozic, která je ukládána po každém kroku. Uživatel může zároveň zapnout či vypnout všechny historie všech jedinců pomocí zaškrtačacího tlačítka.

## **7) Režim dělené obrazovky (Split View)**

Aplikace umožňuje aktivaci režimu Benchmark, kdy jsou současně zobrazeny dvě scény s různými simulacemi. Uživatel tak může např. porovnat chování GA a PSO na stejné funkci, nebo stejný algoritmus na různých funkcích. Každý pohled má vlastní kameru, panel, graf a interaktivní kontrolu.

## **8) Řezy a průhlednost funkcí**

Pomocí přepínače lze aktivovat tzv. slicing režim, v němž se do 3D povrchu vykreslí roviny řezu ve směrech X, Y nebo Z. Zároveň si může uživatel změnit průhlednost funkce pro lepší analýzu jedinců.

## **9) Práce s kamerou**

Uživatel se může po scéně pohybovat klávesami W/S/A/D, případně i Q/E, a myší. K dispozici má také 2 tlačítka, díky kterým může změnit pohled kamery z bohu, nebo z vrchu.

## **10) Pozorování výsledků**

Po několika generacích lze ve většině případů pozorovat zřetelnou konvergenci populace – jedinci se shlukují do oblasti globálního minima. Na grafu dochází k poklesu průměrné fitness a stabilizaci nejlepší hodnoty. Každý algoritmus se chová odlišně. PSO může rychle ztrácet diverzitu, zatímco diferenciální evoluce si udržuje rozptyl delší dobu. Pro urychlení experimentování s programem jsou k dispozici klávesové zkratky, zobrazeny na obrázku 58.

## ZÁVĚR

Cílem této diplomové práce bylo navrhnout a implementovat aplikaci pro interaktivní vizualizaci evolučních algoritmů v prostředí Unity. Hlavním požadavkem bylo umožnit uživateli sledovat a porovnávat chování několika známých evolučních metod (genetický algoritmus, evoluční strategie, diferenciální evoluce, PSO a SOMA) na různých optimalizačních funkcích.

V teoretické části práce byly shrnuty základní principy evolučních algoritmů a odůvodněn výběr Unity jako platformy vhodné pro 3D vizualizaci. Praktická část zahrnovala návrh a realizaci aplikace s rozšiřitelnou architekturou. Výpočetní vrstva obsahuje implementace jednotlivých algoritmů, zatímco vizualizační vrstva řeší vykreslování 3D povrchů funkcí, vizualizaci pohybu jednotlivců v prostoru a ovládání uživatelského rozhraní.

Centrální roli hraje třída EvolutionVisualizer, která integruje výpočetní a vizualizační složky a řídí průběh evoluce. Uživatelské rozhraní umožňuje snadné ovládání simulací, včetně spuštění, zastavení, krokování generací a interakce s jednotlivci. Důležitým prvkem je možnost současného sledování dvou různých simulací pomocí režimu rozdělené obrazovky. Kamerový systém podporuje nezávislé ovládání pohledu v obou režimech.

Implementace zahrnuje rovněž export a import průběhu simulace ve formátu JSON, což umožňuje následnou analýzu dat mimo aplikaci. Všechny funkce a algoritmy byly otestovány a byla ověřena jejich správnost a spolehlivost. Vytvořená aplikace splňuje požadavky na přehlednost, názornost a interaktivitu a může sloužit jako výukový i analytický nástroj.

V rámci zpracování této diplomové práce byl využit jazykový model ChatGPT [38] ke kontrole pravopisu, úpravě stylistiky textu a návrhům na zlepšení formální struktury. Veškerý odborný obsah a interpretace výsledků byly zpracovány autorem.

Celkově lze konstatovat, že cíle práce byly splněny. Výsledkem je funkční a rozšiřitelná platforma vhodná pro hlubší studium a názornou prezentaci evolučních algoritmů v praxi.

## SEZNAM POUŽITÉ LITERATURY

- [1] HOLLAND, John H. *Adaptation in Natural and Artificial Systems*. Ann Arbor : University of Michigan Press, 1975. ISBN 978-0-262-58111-0.
- [2] DE JONG, Kenneth A. *Evolutionary Computation : A Unified Approach*. Cambridge, MA : MIT Press, 2006. ISBN 978-0-262-04194-2.
- [3] EIBEN, Agoston E.; SMITH, James E. *Introduction to Evolutionary Computing*. Berlin : Springer, 2003. ISBN 978-3-662-05094-1.
- [4] YANG, Xin-She. *Recent Advances in Swarm Intelligence and Evolutionary Computation*. Cham : Springer, 2015. ISBN 978-3-319-13825-1.
- [5] SIMON, Dan. *Evolutionary Optimization Algorithms*. Hoboken, NJ : Wiley, 2013. ISBN 978-0-470-93741-9.
- [6] ZELINKA, Ivan. *Umělá inteligence v problémech globální optimalizace*. Praha : BEN – technická literatura, 2002. ISBN 80-7300-069-5.
- [7] KOZA, John R. *Genetic Programming – On the Programming of Computers by Means of Natural Selection*. Cambridge : MIT Press, 1992. ISBN 0-262-11170-5.
- [8] BONGARD, Josh C, 2013. Communications of the ACM. Online. *Evolutionary Robotics*. Roč. 56, č. 8. DOI: 10.1145/2493883. Dostupné z: <https://doi.org/10.1145/2493883>. [cit. 2025-03-21].
- [9] THOMPSON, Adrian, 1997. *An evolved circuit, intrinsic in silicon, entwined with physics*. Berlin – Heidelberg: Springer-Verlag, ISBN 3-540-63173-9. DOI: 10.1007/3-540-63173-9\_61. Dostupné z: [https://doi.org/10.1007/3-540-63173-9\\_61](https://doi.org/10.1007/3-540-63173-9_61). [cit. 2025-05-04]
- [10] ZELINKA, Ivan; AHN, Chang Wook, ed, 2006. Studies in Computational Intelligence, ResearchGate. Online. *SOMA – Self-organizing Migrating Algorithm*. In : *Advances in Evolutionary Algorithms : Theory, Design and Practice*. Sv. 18. Berlin : Springer. Dostupné z: [SOMA—Self-organizing Migrating Algorithm | Request PDF](#). [cit. 2025-02-16]
- [11] RECHENBERG, Ingo. *Evolutionsstrategie : Optimierung technischer Systeme nach Prinzipien der biologischen Evolution*. Stuttgart-Bad Cannstatt : Frommann-Holzboog, 1973. ISBN 3-7728-0373-3.
- [12] SCHWEFEL, Hans-Paul. *Numerische Optimierung von Computer-Modellen mittels der Evolutionsstrategie : Mit einer vergleichenden Einführung in die Hill-Climbing- und*

*Zufallsstrategie*. Online. Basel : Birkhäuser Basel, 2019 (1. vydání 1977). eBook ISBN 978-3-0348-5927-1. ISBN 3-7643-0876-1. [cit. 2025-05-02]

[13] FOGEL, David B. *Evolutionary Computation: Toward a New Philosophy of Machine Intelligence*. Piscataway, NJ: IEEE Press, 1995. ISBN 978-0-471-66951-7.

[14] KENNEDY, James; EBERHART, Russell C, 1995. IEEE xplore. Online. *Particle Swarm Optimization*. In: *Proceedings of the IEEE International Conference on Neural Networks*. Dostupné z: <https://doi.org/10.1109/ICNN.1995.488968>. [cit. 2025-05-02]

[15] STORN, Rainer a PRICE, Kenneth, 1997. *Journal of Global Optimization*. Online. *Differential evolution – a simple and efficient heuristic for global optimization over continuous spaces*. ISSN 0925-5001. DOI: 10.1023/A:1008202821328. [cit. 2025-05-04].

[16] CHALUPNÍK, V., 2012. Root.cz. Online. *Biologické algoritmy (3) – Evoluční algoritmy*. Dostupné z <https://www.root.cz/clanky/biologicke-algoritmy-3-evolucni-algoritmy/>. [cit. 2025-07-05]

[17] POHLHEIM, Hartmut, 1999. GEATbx. Online. *Genetic and Evolutionary Algorithms: Principles, Algorithms, Analysis and Implementation*. Dostupné z: <http://www.geatbx.com/>. [cit. 2025-04-26].

[18] SAFOUAN, S., El MOUTAOUAKIL, K. & PATRICIU, A.-M., 2024. ResearchGate. Online. *Fractional Derivative to Symmetrically Extend the Memory of Fuzzy C-Means*. Dostupné z: [https://www.researchgate.net/figure/Genetic-algorithm-cycle\\_fig1\\_384920746](https://www.researchgate.net/figure/Genetic-algorithm-cycle_fig1_384920746) [cit. 2025-04-18]

[19] DEB, Kalyanmoy. *Multi-Objective Optimization Using Evolutionary Algorithms*. Chichester: Wiley, 2001. ISBN 978-0-471-87339-6.

[20] YAO, Xin; LIU, Yong, 1997. *IEEE Transactions on Neural Networks*. Online. *A new evolutionary system for evolving artificial neural networks*. Roč. 8, č. 3, s. 694–713. DOI: 10.1109/72.572107. Dostupné z: <https://doi.org/10.1109/72.572107>. [cit. 2025-05-01]

[21] WHITLEY, Darrell, 1994. *Statistics and Computing*. Online. *A genetic algorithm tutorial*. s. 65–85. ISSN 0960-3174. DOI: 10.1007/BF00175354. Dostupné z: <https://doi.org/10.1007/BF00175354>. [cit. 2025-05-04]

[22] GOLDBERG, David E. *Genetic Algorithms in Search, Optimization, and Machine Learning*. Reading, MA: Addison-Wesley, 1989. ISBN 978-0-201-15767-3.

- [23] HOUCK, Christopher R.; JOINES, Jeffrey A.; KAY, Michael G, 1995. ResearchGate. Online. *A Genetic Algorithm for Function Optimization: A MATLAB Implementation. NCSU-IE TR 95-09*. Dostupné z: [\(PDF\) A Genetic Algorithm for Function Optimization: A MATLAB implementation](#). [cit. 2025-03-22]
- [24] CLERC, Maurice. *Particle Swarm Optimization*. Chichester : Wiley-ISTE, 2006. ISBN 978-1-905209-04-0.
- [25] KENNEDY, James. *Particle swarm optimization*. In: SAMMUT, Claude; WEBB, Geoffrey I., eds. *Encyclopedia of Machine Learning*. Boston: Springer, 2010, s. 760–766. ISBN 0-7803-2768-3.
- [26] Eberhart and Yuhui Shi, *Particle swarm optimization: developments, applications and resources*. Seoul, Korea (South), 2001, pp. 81-86 vol. 1, DOI: 10.1109/CEC.2001.934374.
- [27] DERRAC, Joaquín; GARCÍA, Salvador; MOLINA, Daniel; HERRERA, Francisco, 2011. Swarm and Evolutionary Computation. Online. *A practical tutorial on the use of nonparametric statistical tests as a methodology for comparing evolutionary and swarm intelligence algorithms*. Roč. 1, č. 1, s. 3–18. Dostupné z: <https://doi.org/10.1016/j.swevo.2011.02.002>. [cit. 2025-03-17]
- [28] JAMIL, Momin; YANG, Xin-She, 2013. International Journal of Mathematical Modelling and Numerical Optimisation. Online. *A literature survey of benchmark functions for global optimisation problems*. Roč. 4, č. 2, s. 150–194. Dostupné z: <https://doi.org/10.1504/IJMMNO.2013.055204>. [cit. 2025-03-17].
- [29] SURJANOVIC, S. *Test functions*. Simon Fraser University. Online. Dostupné z <https://www.sfu.ca/~ssurjano/spheref.html>. [cit. 2025-04-24]
- [30] DOLAN, Elizabeth D., MORÉ, Jorge J., 2002. Mathematical Programming. Online. *Benchmarking optimization software with performance profiles*. Roč. 91, č. 2, s. 201–213. Dostupné z: <https://doi.org/10.1007/s101070100263>. [cit. 2025-04-26]
- [31] UNITY DOCS, 2024. *Unity Documentation*. Online. Dostupné z: <https://docs.unity.com>. [cit. 2025-03-17]
- [32] KEVURU GAMES, 2023. Kevuru Games. Online. *Why Choose Unity for Game Development*. Dostupné z: <https://kevurugames.com/blog/unity-what-makes-it-the-best-game-engine>. [cit. 2025-03-17]

- [33] EPIC GAMES DOCS, 2023. Epic Games. Online. *Unreal Engine 5 Documentation*. Dostupné z: <https://dev.epicgames.com/documentation/en-us/unreal-engine/unreal-engine-5-5-documentation>. [cit. 2025-03-17]
- [34] GODOT ENGINE DOCS, 2024. Godot. Online. *Godot Engine Documentation*. Dostupné z: <https://docs.godotengine.org>. [cit. 2025-03-17]
- [35] CRYTEK, 2023. Crytek. Online. *CRYENGINE V Documentation*. Dostupné z: <https://www.cryengine.com/docs/static/engines/cryengine-5/categories/23756816>. [cit. 2025-03-17]
- [36] YASIRKUŮA. GitHub repositor. Online. *UnitySimpleFileBrowser*. Dostupné z: <https://github.com/yasirkula/UnitySimpleFileBrowser>. [cit. 2025-05-11]
- [37] FLATICON [online]. Freepik Company, 2024. Dostupné z: <https://www.flaticon.com/> [cit. 2025-05-21]
- [38] OPENAI. *ChatGPT (GPT-4)* [online]. San Francisco: OpenAI, 2025 [cit. 2025-05-26]. Dostupné z: <https://chat.openai.com>

## SEZNAM OBRÁZKŮ

|                                                                          |    |
|--------------------------------------------------------------------------|----|
| Obrázek 1 Ukázka jednobodového a dvoubodového křížení .....              | 16 |
| Obrázek 2 Schéma evolučního cyklu.....                                   | 17 |
| Obrázek 3 Aktualizace polohy PSO algoritmu.....                          | 21 |
| Obrázek 4 3D povrch funkce Sphere .....                                  | 24 |
| Obrázek 5 3D povrch funkce Rastrigin .....                               | 25 |
| Obrázek 6 3D povrch funkce Ackley .....                                  | 25 |
| Obrázek 7 3D povrch funkce Rosenbrock.....                               | 26 |
| Obrázek 8 3D povrch funkce Schwefel.....                                 | 27 |
| Obrázek 9 Uživatelské prostředí aplikace .....                           | 38 |
| Obrázek 10 Hierarchie komponent a hlavní layout .....                    | 39 |
| Obrázek 11 Ukázka struktury projektu .....                               | 40 |
| Obrázek 12 Ukázka vizualizační a aplikační vrstvy .....                  | 41 |
| Obrázek 13 Horní lišta s rozbaleným tlačítkem File .....                 | 43 |
| Obrázek 14 Tlačítka pro ovládání simulace.....                           | 44 |
| Obrázek 15 Ukázka funkce s $\alpha = 1$ .....                            | 45 |
| Obrázek 16 Ukázka funkce s $\alpha = 0.5$ .....                          | 45 |
| Obrázek 17 Řez funkce na X-ose s viditelnou vizualizační rovinou.....    | 46 |
| Obrázek 18 Řez funkce při pohledu zezhora .....                          | 46 |
| Obrázek 19 Zobrazení historie pohybu daného jedince.....                 | 47 |
| Obrázek 20 Benchmark mode.....                                           | 48 |
| Obrázek 21 Graf a evoluce po 13. generaci.....                           | 48 |
| Obrázek 22 IEvolutionAlgorithm interface .....                           | 49 |
| Obrázek 23 Zdrojový kód genetického algoritmu .....                      | 50 |
| Obrázek 24 Zdrojový kód evoluční strategie.....                          | 51 |
| Obrázek 25 Zdrojový kód diferenciální evoluce .....                      | 52 |
| Obrázek 26 Nastavení základních parametrů a vytvoření slovníků PSO ..... | 53 |
| Obrázek 27 Zdrojový kód algoritmu PSO .....                              | 54 |
| Obrázek 28 Zdrojový kód algoritmu SOMA.....                              | 55 |
| Obrázek 29 Komponenty pracující s EvolutionVisualizer.cs .....           | 56 |
| Obrázek 30 Inicializace programu po spuštění.....                        | 57 |
| Obrázek 31 Kolekce pro uchovávání dat během simulace .....               | 58 |
| Obrázek 32 Ukázka třídy Individual.....                                  | 58 |
| Obrázek 33 Ukázka funkce CreateRandomPopulation().....                   | 59 |
| Obrázek 34 Výpočet nové generace .....                                   | 60 |

|                                                                   |    |
|-------------------------------------------------------------------|----|
| Obrázek 35 Zobrazování jedinců v generaci.....                    | 60 |
| Obrázek 36 Funkce počítající další generaci, je-li to nutné.....  | 61 |
| Obrázek 37 Proces automatické simulace.....                       | 61 |
| Obrázek 38 Kód pro posun generace o krok zpět .....               | 62 |
| Obrázek 39 Ukončení simulace při splnění podmínky .....           | 63 |
| Obrázek 40 Výpočet pozice jedince.....                            | 64 |
| Obrázek 41 Tvorba jedince .....                                   | 64 |
| Obrázek 42 Ukázka třídy IndividualInfo .....                      | 65 |
| Obrázek 43 Ukázka funkce volané pro zobrazení historie .....      | 66 |
| Obrázek 44 Vykreslovací funkce .....                              | 67 |
| Obrázek 45 Funkce postupující rodokmenem a hledající rodiče ..... | 68 |
| Obrázek 46 Implementace optimalizačních funkcí.....               | 69 |
| Obrázek 47 Generování meshe .....                                 | 70 |
| Obrázek 48 Funkce pro generování meshe .....                      | 71 |
| Obrázek 49 Ukázka referencí pro skript řezné roviny .....         | 72 |
| Obrázek 50 Aktualizace řezací plochy.....                         | 72 |
| Obrázek 51 Předání argumentů řezné roviny materiálu.....          | 73 |
| Obrázek 52 Shader pro zobrazení roviny řezu.....                  | 74 |
| Obrázek 53 Hlavní struktura třídy s grafem.....                   | 75 |
| Obrázek 54 Překreslování grafu.....                               | 76 |
| Obrázek 55 Komponenty související s rozděleným režimem.....       | 77 |
| Obrázek 56 Ukázka metody ApplyMode() ve správci kamer.....        | 79 |
| Obrázek 57 Třída sloužící k ukládání/načítání JSON souborů .....  | 80 |
| Obrázek 58 Ukázka klávesových zkratk .....                        | 81 |
| Obrázek 59 Ukázka tooltipu .....                                  | 82 |

**SEZNAM TABULEK**

|                                 |    |
|---------------------------------|----|
| Tabulka 1 Srovnání enginů ..... | 36 |
|---------------------------------|----|

## SEZNAM POUŽITÝCH SYMBOLŮ A ZKRATEK

|         |                                                                                                    |
|---------|----------------------------------------------------------------------------------------------------|
| API     | Application Programming Interface (Programové rozhraní aplikací)                                   |
| CMA-ES  | Covariance Matrix Adaptation Evolution Strategy (Evoluční strategie adaptací kovarianční matice)   |
| CEC     | Congress on Evolutionary Computation (Kongres IEEE o evolučním počítání)                           |
| DE      | Differential Evolution (Diferenciální evoluce)                                                     |
| EA      | Evolutionary Algorithms (Evoluční algoritmy)                                                       |
| ES      | Evolution Strategy (Evoluční strategie)                                                            |
| GA      | Genetic Algorithm (Genetický algoritmus)                                                           |
| GECCO   | Genetic and Evolutionary Computation Conference (Konference o genetických a evolučních výpočtech)  |
| GPU     | Graphics Processing Unit (Grafická výpočetní jednotka)                                             |
| JSON    | JavaScript Object Notation (Formát pro výměnu dat ve struktuře klíč–hodnota)                       |
| LOD     | Level of Detail (Úroveň detailu)                                                                   |
| NSGA-II | Non-dominated Sorting Genetic Algorithm II (Genetický algoritmus pro vícekritériální optimalizaci) |
| PSO     | Particle Swarm Optimization (Optimalizace pomocí částicového roje)                                 |
| SOMA    | Self-Organizing Migrating Algorithm (Samo-organizující se migrační algoritmus)                     |
| UI      | User Interface (Uživatelské rozhraní)                                                              |
| uGUI    | Unity Graphical User Interface (Grafický UI systém v Unity)                                        |
| URP     | Universal Render Pipeline (Univerzální vykreslovací pipeline v Unity)                              |

## SEZNAM PŘÍLOH

Příloha P I: ZIP soubor s aplikací a zdrojovým kódem

## **PŘÍLOHA P I: ZIP SOUBOR S APLIKACÍ A ZDROJOVÝM KÓDEM**

Složka má následující strukturu:

- fulltext (složka)
  - fulltext.pdf – písemná část diplomové práce (tento dokument)
- src (složka)
  - obsahuje celý Unity projekt včetně všech zdrojových kódů
- app (složka)
  - obsahuje sestavenou .exe aplikaci