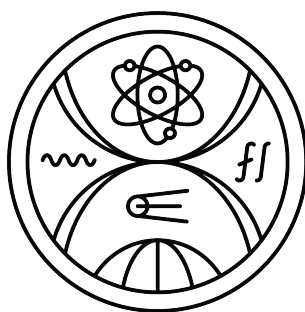


UNIVERZITA KOMENSKÉHO V BRATISLAVE  
FAKULTA MATEMATIKY, FYZIKY A INFORMATIKY

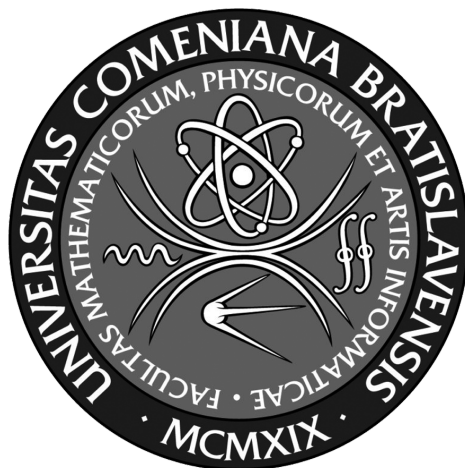


SIMULÁCIA SYSTÉMU AUTONÓMNYCH VOZIDIEL  
S VYUŽITÍM PARALELNÉHO SPRACOVANIA  
*MULTI-AGENT SIMULATION OF AUTONOMOUS  
VEHICLES USING PARALLEL PROCESSING*  
DIPLOMOVÁ PRÁCA

2025  
BC. MATEJ MAGÁT



UNIVERZITA KOMENSKÉHO V BRATISLAVE  
FAKULTA MATEMATIKY, FYZIKY A INFORMATIKY



SIMULÁCIA SYSTÉMU AUTONÓMNYCH VOZIDIEL  
S VYUŽITÍM PARALELNÉHO SPRACOVANIA  
*MULTI-AGENT SIMULATION OF AUTONOMOUS  
VEHICLES USING PARALLEL PROCESSING*  
DIPLOMOVÁ PRÁCA

Študijný program: Aplikovaná informatika  
Študijný odbor: Informatika  
Školiace pracovisko: Katedra informatiky  
Školiteľ: Mgr. Andrej Mihálik, PhD.

Bratislava, 2025  
Bc. Matej Magát





Univerzita Komenského v Bratislave  
Fakulta matematiky, fyziky a informatiky



26556492

## ZADANIE ZÁVEREČNEJ PRÁCE

**Meno a priezvisko študenta:** Bc. Matej Magát  
**Študijný program:** aplikovaná informatika (Jednoodborové štúdium, magisterský II. st., denná forma)  
**Študijný odbor:** informatika  
**Typ záverečnej práce:** diplomová  
**Jazyk záverečnej práce:** slovenský  
**Sekundárny jazyk:** anglický

**Názov:** Simulácia systému autonómnych vozidiel s využitím paralelného spracovania  
*Multi-Agent Simulation of Autonomous Vehicles Using Parallel Processing*

**Anotácia:** Táto práca navrhuje detailný rámec pre realistickú multi-agentovú simuláciu s cieľom podrobného štúdia komplexných procesov koordinácie autonómnych vozidiel v komplexnej sieti ciest. Systém bude využívať pokročilé techniky paralelného spracovania, ako sú OpenCL, CUDA alebo výpočtové shadery, na efektívnu simuláciu správania viacerých vozidiel a centrálného riadiaceho systému.

**Cieľ:** Vyvinúť rámec pre multi-agentnú simuláciu, ktorý predstavuje každé autonómne vozidlo ako agenta a centrálny riadiaci systém ako samostatný proces. Implementovať modely správania agentov, ktoré zahŕňajú plánovanie trasy, vyhýbanie sa prekážkam a komunikáciu s centrálnym riadiacim systémom. Integrovať techniky paralelného spracovania (napr. OpenCL, CUDA, výpočtové shadery) na efektívnu simuláciu správania viacerých agentov. Vyhodnotiť efektívnosť rôznych techník paralelného spracovania pri simulácii viacerých autonómnych vozidiel na rôznych hardvérových platformách. Vyvinúť vizualizačný nástroj na efektívne znázornenie výsledkov simulácie a uľahčenie analýzy.

### Literatúra:

#### Kľúčové

**slová:** Multi-agentná simulácia, Paralelné spracovanie, Koordinácia dopravy

**Vedúci:** Mgr. Andrej Mihálik, PhD.  
**Katedra:** FMFI.KAI - Katedra aplikovanej informatiky  
**Vedúci katedry:** doc. RNDr. Tatiana Jajcayová, PhD.  
**Dátum zadania:** 02.10.2024

**Dátum schválenia:** 22.10.2024

prof. RNDr. Roman Ďurikovič, PhD.  
garant študijného programu

.....  
študent

.....  
vedúci práce





**Čestné prehlásenie:**

Čestne vyhlasujem, že celú diplomovú prácu na tému „Simulácia systému autonómnych vozidiel s využitím paralelného spracovania

*Multi-Agent Simulation of Autonomous Vehicles Using Parallel Processing*“, vrátane všetkých jej príloh a obrázkov, som vypracoval na základe samostatného uvažovania, a to s použitím literatúry uvedenej v priloženom zozname a nástrojov umelej inteligencie. Vyhlasujem, že nástroje umelej inteligencie som použil v súlade s príslušnými právnymi predpismi, akademickými právami a slobodami, etickými a morálnymi zásadami za súčasného dodržania akademickej integrity. Umelá inteligencia bola použitá pri tvorbe základnej kostry textu - formát, zhrnutie danej problematiky - a na opravu niektorých gramatických a vyjadrovacích chýb.



## Podakovanie

Na tomto mieste by som sa rád poďakoval svojmu školiťovi za to, že mi umožnil vypracovať túto diplomovú prácu v skrátrenom a urýchlennom režime, ako aj za jeho trpezlivosť a cenné rady počas celého procesu. Ďalej ďakujem svojej sestre Kataríne Magátovej za pomoc pri tvorbe grafov, ktorá mi ušetrila veľa času a úsilia.

Osobitné poďakovanie patrí aj Matúšovi Rovenskému a sestre Kataríne Magátovej za pomoc pri úprave štylistiky a gramatiky textu.

V neposlednom rade ďakujem všetkým ostatným, ktorí mi akýmkoľvek spôsobom pomohli — či už morálnou podporou, praktickými radami alebo modlitbami.

## Abstrakt

Diplomová práca sa zaoberá návrhom a implementáciou multiagentového systému na plánovanie trasy v prostredí bežného domáceho počítača. Cieľom bolo umožniť efektívne a na kolízie odolné plánovanie pohybu viacerých autonómnych agentov s využitím paralelizácie výpočtov na GPU. Systém pritom počíta so schopnosťou agentov konať samostatne a je doplnený o mechanizmy detekcie a riešenia kolízií. Počas vývoja boli preskúmané viaceré algoritmy a prístupy, vrátane paralelizácie výpočtov na GPU prostredníctvom knižnice SYCL.

Hoci implementácia nepreukázala očakávané zrýchlenie — CPU verzia bola v mnohých testovacích prípadoch výkonnejšia — podarilo sa vytvoriť plne funkčný a stabilný systém, schopný rovnocenne fungovať na CPU aj GPU. Práca predstavuje návrh architektúry systému, detailný popis implementácie a analýzu výsledkov testovania na rôznych scenároch.

Implementácia preukázala svoju funkčnosť, rozšíriteľnosť a spoľahlivosť. Napriek niektorým pevne nastaveným parametrom (napríklad limit 4 GB pamäte), je možné systém prispôbiť zariadeniam s vyššou kapacitou VRAM.

**Kľúčové slová:** GPU, SYCL, plánovanie trasy, paralelné spracovanie, multiagentové systémy

# Abstract

This thesis focuses on the design and implementation of a multi-agent path planning system intended for use on a standard home computer. The goal was to enable efficient and collision-resilient movement planning for multiple autonomous agents by leveraging GPU-based parallel computation. The system accounts for agents' ability to act independently and incorporates mechanisms for collision detection and resolution. During development, several algorithms and approaches were explored, including GPU-based parallelization using the SYCL library.

Although the implementation did not deliver the expected speedup — the CPU version outperformed the GPU version in many test scenarios — a fully functional and stable system was successfully developed, capable of running equivalently on both CPU and GPU. The thesis presents the system architecture, detailed implementation, and analysis of test results across various scenarios.

The implementation demonstrated functionality, extensibility, and reliability. Despite some hardcoded parameters (e.g., a 4 GB memory limit), the system can be adapted for devices with higher VRAM capacity.

**Keywords:** GPU, SYCL, path planning, parallel processing, multi-agent systems



# Obsah

<b>Slovník pojmov - vysvetlivky</b>	<b>1</b>
<b>Úvod</b>	<b>5</b>
<b>1 Súčasný stav – prehľad a výskum</b>	<b>7</b>
1.1 Skúmané technológie . . . . .	7
1.1.1 CUDA . . . . .	7
1.1.2 OpenCL . . . . .	8
1.1.3 GLSL / Vulkan Compute Shaders . . . . .	8
1.1.4 SYCL . . . . .	9
1.1.5 Obmedzenia SYCL na mobilných platformách . . . . .	10
1.2 Vyhľadávacie algoritmy . . . . .	11
1.2.1 A* . . . . .	11
1.2.2 D* . . . . .	11
1.2.3 D* Lite . . . . .	12
1.3 Algoritmy na riešenie konfliktov . . . . .	12
1.3.1 CBS – Conflict-Based Search . . . . .	12
1.3.2 Priority-Based Search (PBS) . . . . .	12
1.3.3 CA* – Cooperative A* . . . . .	13
1.3.4 Mravčí algoritmus (Ant Colony Optimization – ACO) . . . . .	13
1.4 Zhrnutie kapitoly . . . . .	14
<b>2 Analýza súčasných prístupov v literatúre</b>	<b>15</b>
2.1 Presun od centralizovaných k decentralizovaným systémom . . . . .	15
2.2 Optimalizácia trojice robotov . . . . .	17
2.3 Automatizované riadenie premávky autonómnych vozidiel na kontajne- rovom termináli . . . . .	17
2.4 Autonómne vozidlá s pevne definovanými trasami . . . . .	19
2.5 Multi-Agent Path Finding (MAPF) . . . . .	20
<b>3 Cieľ práce</b>	<b>21</b>

<b>4</b>	<b>Návrh experimentov</b>	<b>23</b>
4.1	Pamäťové obmedzenia a motivácia . . . . .	24
4.2	Rozsah testovaných parametrov . . . . .	24
4.3	Testované výpočtové režimy . . . . .	25
4.4	Metodika merania . . . . .	26
4.5	Hardvérové podmienky . . . . .	26
4.6	Predpokladané výsledky a hypotézy . . . . .	26
<b>5</b>	<b>Postup a implementácia</b>	<b>27</b>
5.1	Grafické používateľské rozhranie (GUI) . . . . .	28
5.1.1	Architektúra GUI . . . . .	28
5.1.2	Grafické knižnice - komponenty . . . . .	29
5.1.3	Zásady/Princípy návrhu . . . . .	29
5.1.4	Správa scén a prechodov . . . . .	29
5.1.5	Vlákná a simulácia . . . . .	29
5.1.6	Prepojenie medzi scénami . . . . .	29
5.1.7	Spracovanie užívateľských vstupov . . . . .	30
5.2	Výpočtová logika . . . . .	31
5.2.1	Prehľad výpočtovej časti . . . . .	31
5.2.2	Dátové štruktúry a správa pamäte . . . . .	31
5.2.3	Prístupy – GPU a CPU . . . . .	32
5.3	Grafy . . . . .	33
5.4	Prehodnotenie návrhu . . . . .	39
<b>6</b>	<b>Experimenty a výsledky</b>	<b>45</b>
6.1	Porovnanie výpočtového času CPU a GPU pri hashovaní . . . . .	45
6.2	Výsledky . . . . .	46
6.3	Zhrnutie výsledkov . . . . .	50
	<b>Záver</b>	<b>51</b>
	<b>Príloha A</b>	<b>55</b>
	<b>Príloha B</b>	<b>57</b>

# Slovník pojmov

- **AGV** - skratka z anglického automated guided vehicle - automatické riadenie robotov.
- **AMR** - skratka z anglického autonomous mobile robot
- **Jemnozrnná kontrola** (v kontexte programovania/paralelizácie) - riadenie algoritmu na veľmi nízkej úrovni (low level). Kontrola vlákien alebo pamäťových prístupov umožňuje programátorom riadiť jednotlivé detaily a precízne nastaviť výkon, synchronizáciu, rozdelenia úloh, či správu pamäte medzi výpočtovými jednotkami. V texte sa môže zamieňať so slovným spojením nízkoúrovňová kontrola či kontrola na nízkej úrovni.
- **CUDA core** (tiež stream processor) – jednoduché výpočtové jadro, ktoré je na architektúre CUDA od spoločnosti NVIDIA základnou výpočtovou jednotkou. Každý CUDA core má byť postavaný tak, aby vykonávalo základné aritmetické operácie ako sčítanie, násobenie a iné logické operácie. V porovnaní s ich náprotivkom CPU jadrami sú CUDA jadrá výpočtovo oveľa slabšie. CUDA jadrá majú však výhodu pri paralelizme. Pri používaní grafických kariet mimo okruh NVIDIA sa programátor môže stretnúť s pojmom stream processor. Stream processor je ekvivalentom pojmu CUDA core. Hoci presná terminológia a ich vnútorná stavba či technológia za týmito pojmami sa môže líšiť podľa generácie architektúry a výrobcu. V princípe ide o paralelné jednotky, ktoré sú schopné paralelne vykonávať kernely.
- **Kernel** – predstavuje funkciu (alebo procedúru), ktorá je skompilovaná tak, aby ju vedela vykonať grafická karta alebo iný akcelerátor (pri OpenCL) paralelne, čo je hlavná výhoda pri programovaní kernela. Jej kompilácia sa môže udiť počas kompilácie programu alebo pomocou JIT. Kernel je určený na vykonanie na viacerých výpočtových jadrách GPU súčasne. Každé vlákno (stream mimo CUDA - thread v CUDA) spúšťa ten istý kernel, ale pomocou identifikátora každé vlákno vie pracovať na inej časti dát. Pojem kernel sa spomína v CUDA, OpenCL, GLSL, ale aj v SYCL programovaní.

- **JIT kompilácia (Just-In-Time)** – kompilácia časti programu počas samotného behu programu. Namiesto úplnej kompilácie celého programu je časť zdrojového kódu ponechaná nepreložená alebo čiastočne preložená a preložená je až pri behu programu. To umožňuje, aby sa kód skompiloval na mieru konkrétnym hardvérovým špecifikám alebo určitým podmienkam daného behu. V kontexte počítania na grafických kartách sa JIT používa preto, aby sa vedel využiť plný potenciál špecifickej grafickej karty. Grafické karty sú menej štandardizované ako procesory tej istej architektúry, preto je výhodnejšia JIT kompilácia.
- **Warp** – na GPU architektúrach NVIDIA sa niekoľko vlákien zoskupuje do špeciálnej jednotky. Táto jednotka spravidla obsahuje / združuje 32 vlákien (threads). Všetky vlákna vo warpe by mali vykonávať rovnakú inštrukciu v jednom takte (podobne ako v SIMD modeli), pričom pracujú na rôznych dátach. Ak sa vetvením (napr. `if-else`) dostanú do rôznych ciest, nastáva takzvaná *divergencia warpu*, čo vedie k zníženiu výkonnosti. Pri *divergencii warpu* musí GPU *warp* vykonať každú vetvu zvlášť. Tento jav môže byť výhodný pri optimalizácii paralelných operácií, no predstavuje nevýhodu, keď jednotlivé vlákna potrebujú vykonávať odlišné inštrukcie.
- **CUDA block (thread block)** – skupina vlákien (threads) v CUDA, ktoré spolu tvoria jednu jednotku plánovania. Vlákna v bloku môžu zdieľať lokálne pamäťové zdroje a vykonávať synchronizáciu pomocou príkazu `__syncthreads()`. Synchronizácia nie je, na rozdiel od *warpu*, implicitná, bloky môžu vykonávať rôzne inštrukcie a nie je to neželaný jav. Tento prístup je programátorsky flexibilnejší, napríklad pri simulácii viacerých agentov naraz.
- **Work-item** – pojem používaný v OpenCL alebo SYCL pre jednotlivé vlákno (ekvivalent k `thread` v CUDA, ale samostatnejší ako v CUDA). Každý work-item vykonáva jednu inštanciu kernelu a má svoj vlastný identifikátor, ktorý pri OpenCL získa príkazom `get_global_id()`, zatiaľ čo v SYCL ho dostáva ako argument.
- **Work-group** – zoskupenie viacerých work-item-ov, ktoré môžu medzi sebou zdieľať lokálnu pamäť a synchronizovať sa (napr. cez bariéru `barrier()`). Work-group je približný ekvivalent CUDA `blocku` v rámci OpenCL alebo SYCL. Work-groupy sa vykonávajú nezávisle a ich počet a veľkosť možno definovať pri spustení kernelu.
- **Feromón** – aromatická látka, ktorú živočíchy vedia vycítiť často aj hodiny po vypustení, funguje na značkovanie a komunikáciu.



- **NP-ťažký (NP-hard)** – Trieda problémov, na ktoré neexistuje známy polynomiálny algoritmus, ktorý by ich vyriešil, a ich riešenie je považované za výpočtovo náročné.
- **MILP (Mixed-Integer Linear Programming)** – zmiešané celočíselné lineárne programovanie, programovacia technika, kde cieľom je optimalizácia funkcie (ktorá je lineárna) pri dodržaní lineárnych obmedzení. Niektoré typy premenných sú celočíselné, iné naopak spojité. MILP (Mixed-Integer Linear Programming) sa využíva napríklad v úlohách plánovania trás.
- **LiDAR (Light Detection and Ranging)** – technológia založená na meraní vzdialenosti pomocou odrazených laserových impulzov. Vytvára z meraných vzdialeností body a z bodov vytvára troj-dimenzionálny model okolia.
- **SLAM (Simultaneous Localization and Mapping)** – simultánna lokalizácia a mapovanie. Metodika, pri ktorej robot - agent alebo autonómny systém vytvára mapu prostredia, a zároveň určuje polohu seba samého v mapu, na základe senzorických dát (napríklad zo systému LiDAR alebo z kamery) a algoritmom na odhadovanie pohybu, a štruktúry prostredia.



# Úvod

V súčasnosti patrí simulácia autonómnych vozidiel medzi intenzívne skúmané oblasti. Využívajú sa pritom poznatky z informatiky, umelej inteligencie, robotiky, ako aj paralelného spracovania. S rastúcim počtom autonómnych systémov sa zákonite zvyšuje aj ich zložitosť, čo vytvára tlak na vývojárov, aby hľadali efektívne metódy plánovania, koordinácie a riadenia správania týchto systémov s čo najnižšími časovými a výpočtovými nákladmi.

Vývoj a testovanie takýchto systémov na reálnych prototypoch je často veľmi nákladný a môže byť aj potenciálne nebezpečný pre okolie (napriek existujúcim výnimkám, ako je firma SpaceX). Z tohto dôvodu je výhodné vytvárať na počítačoch modely reálneho sveta – hoci často zjednodušené – ktoré umožňujú overiť návrhy ešte pred ich nasadením v praxi.

Cieľom tejto práce bolo navrhnuť a implementovať simuláciu autonómnych vozidiel, ktorá umožní študovať ich správanie a koordináciu pomocou výpočtov na grafickej karte (GPU) alebo inom akcelerátore schopnom paralelného výpočtu. Sústredili sme sa na vývoj systému, ktorý efektívne využíva paralelizmus výpočtov. Zvažovali sme technológie ako CUDA, OpenCL, SYCL či výpočtové shadery a testovali ich použitie pri paralelnom plánovaní ciest pre desiatky agentov. Výsledky sme následne porovnali s konvenčnými prístupmi využívajúcimi CPU.

V práci sú predstavené technológie a algoritmy, ktoré sme postupne preskúmali. Osobitne sa venujeme dvom skupinám algoritmov – algoritmom na vyhľadávanie trás (napr. A\*, D\*, D\* Lite) a algoritmom na riešenie konfliktov medzi agentmi (napr. CBS, PBS, CA\*), pričom pri každej sme posudzovali ich vhodnosť na paralelné spracovanie. Popisujeme návrh architektúry simulácie, jej implementáciu, experimentálnu časť a vyhodnotenie výsledkov. Súčasťou práce je aj vlastný simulačný nástroj s grafickým rozhraním, ktorý slúži na vizualizáciu výpočtov a úpravu máp.

Práca má za cieľ ukázať možnosti implementácie multi-agentných systémov v paralelnom výpočtovom prostredí a navrhnuť spôsob, ako čo najviac výpočtovej záťaže presunúť z procesora (CPU) na grafickú kartu (GPU) alebo iný akcelerátor.



# Kapitola 1

## Súčasný stav – prehľad a výskum

Táto kapitola predstavuje technológie, ktoré umožňujú efektívnu paralelizáciu výpočtov na GPU a iných akceleratoroch. Následne sa zameriava na algoritmy využívané pri hľadaní ciest v grafe a napokon na stratégie riešenia konfliktov medzi viacerými agentmi.

### 1.1 Skúmané technológie

#### 1.1.1 CUDA

Prvá technológia, ktorú sme skúmali, je rozhranie - API CUDA (Compute Unified Device Architecture). Rozhranie CUDA bolo vyvinuté spoločnosťou NVIDIA ako výpočtová platforma pre ich ekosystém grafických kariet[9]. Rozhranie CUDA umožňuje programovať prostredníctvom rozšírení jazyka C/C++ a niektorých ďalších jazykov priamo jadrá na GPU a využívať ich na paralelné výpočty pomocou kernela. CUDA sa považuje za silný nástroj, pretože poskytuje jemnozrnnú kontrolu nad paralelným výpočtom, alokáciou pamäte a synchronizáciou. Jemnozrnná kontrola poskytuje dostatok možností na jemné optimalizácie kódu.

Rozhranie CUDA sme zvažovali pre jeho podporu masívneho paralelizmu. Masívny paralelizmus sme chceli využiť do algoritmov prehľadávania ciest paralelne (napr. v rámci algoritmu A\*). K nemalým výhodám rozhrania CUDA patrí aj rozsiahla dokumentácia a komunita. Rozhranie CUDA však funguje výhradne na grafických kartách vyrobených firmou NVIDIA, čo obmedzuje jej použitie v heterogénnych prostrediach od iných výrobcov ako AMD či Intel. Na rozdiel od iných sa program pre GPU kompiluje s ostatným programom a nie pomocou JIT.

### 1.1.2 OpenCL

Druhou skúmanou technológiu bolo OpenCL (Open Computing Language). OpenCL je otvorený štandard vydávaný spoločnosťou Khronos s viacerými stabilnými podštandardmi (najčastejšie používané sú napríklad 1.2, 2.0 a najnovší v dobe písania tejto diplomovej práce bol 3.0). OpenCL je určený pre paralelné programovanie naprieč rôznymi typmi hardvéru, vrátane GPU, FPGA, ale aj na CPU či ďalších akceleratoroch. Hlavnou výhodou oproti CUDA je, že OpenCL je multiplatformový a nezávislý od konkrétneho výrobcu, to znamená, že program je ľahko prenositeľný na iné platformy.

Na rozdiel od programovania v CUDA si programátor v OpenCL musí príkazmi sám nastaviť výpočtové prostredie, čo zahŕňa vytvorenie bufferov, kompiláciu kernelov zo stringov pomocou JIT, priradovania kontextov a usporiadania frônt úloh. Toto poskytuje výhodu v podobe úplnej kontroly nad výpočtovým procesom a aj nad kopírovaním dát z pamäte CPU RAM a do pamäte samotného akceleratora a naopak.

OpenCL umožňuje, pri implementácii vyhľadávacích algoritmov, spracovať tisíce paralelných operácií. Paralelné spracovanie v kerneli robí OpenCL ideálny na využitie pri masívnom spracovaní grafov alebo v real-time systémoch. Na druhej strane, jeho absencia vyššej abstrakcie a náročnosť pri ladení predstavujú nevýhody, ktoré treba zohľadniť.

### 1.1.3 GLSL / Vulkan Compute Shaders

GLSL (OpenGL Shading Language) a Vulkan Compute Shaders predstavujú alternatívny prístup k výpočtom na GPU, najmä v rámci grafického a herného vývoja, a hoci sa nezvyknú používať priamo na výpočty osamotene, my sme istý čas uvažovali nad použitím tejto technológie. OpenGL pre kompiláciu svojich kernelov vyžaduje špeciálny nízkoúrovňový jazyk zvaný GLSL. Tento jazyk je používaný na tvorbu kernelov, tu nazývaných ako shader-y. Ďalším vývojom je Vulkan, ktorý je modernejšia a nižšieúrovňová verzia tejto technológie. Obe (OpenGL a Vulkan) používajú shader-y pomocou JIT.

Uvažovali sme využiť compute shadery, ktoré sú väčšinou k dispozícii obom (OpenGL aj Vulkanu). Ich nevýhodou však je, že napriek tomu, že sa dajú použiť a sú najviac prenositeľné na skoro každú platformu, nie sú špeciálne navrhnuté pre vedecké výpočty. V niektorých prípadoch poskytujú výhodu v jednoduchosti nasadenia a integrácie s vizualizačnými systémami (napr. real-time zobrazenie výstupu plánovania ciest), no vo väčšine prípadov sa neodporúča ich využívať ako solídny základ.

Ich nevýhodou je tiež slabšia podpora pokročilých dátových štruktúr a zložitejšia správa synchronizácie medzi vláknami, čo nie je vhodné v prípade, že programátor chce synchronizačné riešenie kolízií.

### 1.1.4 SYCL

SYCL je moderné C++ rozhranie pre heterogénne výpočty. Umožňuje paralelné programovanie pre rôzne zariadenia – CPU, GPU a ďalšie akcelerátory – pomocou jednotného kódu. SYCL, štandardizované organizáciou Khronos Group, má viacero implementácií (ktorý nájdeme [7]). V tejto práci boli použité **oneAPI** (ktorý nájdeme [11]) a **AdaptiveCpp** (predtým **hipSYCL** dostupnej na [1]). SYCL používa „medzistupeň“ medzi úplným JIT zo znakového reťazca z OpenCL a úplnou kompiláciou z CUDA. Program sa skompiluje do LLVM a ten sa kompiluje JIT do SYCL akcelerátora. Keďže sme sa nakoniec rozhodli pre túto možnosť, tak sa pozrieme bližšie na to, čo SYCL ponúka:

#### USM – Jednotná správa pamäte

Unified Shared Memory zjednodušuje prácu s pamäťou tým, že eliminuje potrebu ručne kopírovať dáta medzi hostom a zariadením. Podporuje pamäťové modely pre zariadenie, hosta aj zdieľané rozhranie. Nevýhodou tohto prístupu je veľká spotreba synchronizačného času. V prípade vysokého počtu súbežných vlákien sa oneskorenie spôsobené synchronizáciou môže stať úzkym hrdlom aplikácie.

#### Predalokácia pamäte

Vzhľadom na to, že dynamická alokácia pomocou USM na GPU môže výrazne znižovať výkon, bola skúmaná možnosť predalokácie všetkých dátových štruktúr ako jednorozmerných polí. Každý stream tak pristupuje k svojej časti pamäti na základe identifikátora, čím sa zabezpečuje nezávislosť a paralelizovateľnosť. Tento prístup je výhodný najmä pri plánovaní ciest pre viacerých agentov, kde každý agent operuje vo vlastnej izolovanej časti pamäti. Nakoniec sme sa rozhodli uprednostniť predalokáciu pamäte pred USM prístupom.

#### Synchronizácia

Aj riešenia na grafických kartách používajú viacero synchronizačných techník. Jednou možnosťou je čakanie v work-groups. GPU umožňuje natívnu možnosť čakať na ostatné vlákna v rámci work-group-ov. Druhou možnosťou na GPU sú „spin-locky“, ktoré slúžia na riadenie prístupu ku kritickým sekciám. (Túto druhú možnosť sme nakoniec v tejto práci nevyužili.) Tretím spôsobom je využiť funkciu „**depends\_on()**“, ktorá zabezpečuje explicitnú závislosť medzi jednotlivými úlohami v rámci výpočtovej fronty a umožňuje programátorovi presne určiť poradie výpočtov bez potreby aktívneho čakania.

## Minimalistický štýl kódu

Vzhľadom na obmedzenú podporu komplexných objektovo-orientovaných konštrukcií v niektorých implementáciách SYCL sa často využíva minimalistický, procedurálny štýl implementácie algoritmov, pripomínajúci skôr programátorský jazyk C ako programátorský jazyk C++. Tento prístup minimalizuje režijné náklady spojené s dynamickou alokáciou, polymorfizmom či výnimkami. Okrem toho zabezpečuje vyššiu kompatibilitu medzi jednotlivými verziami SYCL a umožňuje jednoduchšiu analýzu a ladenie výkonu.

### 1.1.5 Obmedzenia SYCL na mobilných platformách

V dokumentácii spoločnosti Qualcomm sa uvádza:

*“With the support of SVM in OpenCL 2.0, the host and devices can share pointers and complex data structures that may contain pointers. [...] This allows developers to write highly interactive parallel code between device and host with minimum synchronization costs.” [10]*

Podľa tejto dokumentácie je OpenCL 2.0 plne podporovaný na grafických jednotkách Adreno A5x, vrátane pokročilých funkcií ako *fine-grained buffer SVM* s podporou atómových operácií. V praxi je však implementácia na mobilných platformách, ako je Android, omnoho zložitejšia a často neúplná.

Moderné rámce ako *SYCL 2020* alebo jeho implementácia *AdaptiveCpp* požadujú širšiu funkcionálnosť než len zdieľanie pointerov pomocou *fine-grained buffer SVM*. Očakávajú komplexný systém správy pamäte (*fine-grained system SVM*), ktorý zahŕňa správu životného cyklu objektov a transparentný prenos dát medzi hostom a zariadením. V dokumentácii *AdaptiveCpp* sa výslovne uvádza, že základná podpora SVM v rámci OpenCL nie je dostatočná:

*“It’s not sufficient for an OpenCL device to support pointer-based SVM; a full memory management system is required for SYCL compliance.” [1]*

Počas experimentov sme skúmali aj možnosť kompilácie aplikácií s použitím SYCL v prostredí **Termux** (terminálový emulátor pre Android). Zistili sme však, že aktuálne verzie systému Android natívne nepodporujú rámec SYCL. Napriek týmto obmedzeniam sme sa rozhodli SYCL v práci použiť, keďže jeho architektúra poskytuje dlhodobý potenciál a kompatibilitu s modernými heterogénnymi výpočtovými platformami.



## 1.2 Vyhľadávacie algoritmy

V tejto sekcii si popíšeme niekoľko prehľadávacích algoritmov, nad ktorými sme počas robenia tejto diplomovej práce uvažovali.

### 1.2.1 A\*

Kombináciou výhod Dijkstrovho algoritmu a heuristického prístupu dostávame algoritmus A\* (čítaj Á star). A\* algoritmus používa funkciu  $f(n) = g(n) + h(n)$  na vybratie vhodného uzla na ďalšie spracovanie. Kde  $g(n)$  predstavuje cestu, ktorú agent musí prejsť do daného bodu ( $n$ ) z počiatku. Heuristický odhad  $h(n)$  je heuristikou odhadujúcou zostávajúcu vzdialenosť do cieľového uzla. Táto heuristika musí mať vlastnosť monotónnej funkcie alebo aspoň funkcie, aby algoritmus našiel vždy optimálne riešenie.

A\* dokáže efektívne obmedziť počet preskúmaných uzlov tým, že vďaka heuristike nerozvíja neperspektívne uzly. Tým, že A\* nerozvíja neperspektívne uzly, má výhodu oproti neinformovaným algoritmom (ako už spomínaný Dijkstrov algoritmus, či prehľadávanie do šírky). Preto sa široko používa v oblasti plánovania trás a je naša prvá a nakoniec aj jediná voľba. A\* vieme ľahko prerobiť, aby zvládol rôzne typy grafov (mriežka, ktorú sme kvôli pamäti GPU použili, ale aj topologický graf) a rôznym metrikám vzdialenosti.

A\* sa nám ukázal ako vhodný najmä v kontexte plánovania trás pre viacerých agentov pomocou work-itemov, kde každý work-item mohol nezávisle na iných počítať prideleného agenta. To je vhodné pre technológie ako OpenCL či SYCL.

### 1.2.2 D\*

Pri dynamickom prostredí sa používa Dynamic A\*(D\*), ktorý sa používa v prostrediach, kde sa môžu prekážky pohybovať. Vychádza z klasického A\* algoritmu, pričom je navrhnutý tak, že namiesto toho, aby sa spustilo plánovanie odznova pri akejkoľvek zmene polohy alebo objavení prekážky, tak D\* využije výpočty, ktoré urobil podobne ako A\* pri prvotnom plánovaní. Týmto spôsobom sa vyhýba prepočítaniu celej trasy a prepočítava iba uzly, ktoré boli zmenou ovplyvnené. Tento prístup znižuje nároky pre svoj výpočet.

Napriek výhodám, ktoré D\* poskytuje, má aj svoje nevýhody. Jeho interná logika je (v porovnaní s klasickým A\*) komplexnejšia a režijná záťaž je vyššia. D\* je teda náročnejší na implementáciu (a to aj v prípade, že programátor má k dispozícii všetky vyššie štruktúry z C++, či iných vyšších jazykov). Kvôli tomu, že sa stále uzly aktualizujú, je obtiažne zistiť, kedy už vzniknutá situácia nemá riešenie. Navyše pamäťové nároky a manažment pre dáta pri absencii vysokej miery abstrakcie sa nehodí na GPU. (Detailnejšie sa touto témou zaoberali v [5].)

### 1.2.3 D\* Lite

Sven Koenig vyvinul však zjednodušenú a efektívnejšiu verziu zvanú D\* Lite. D\* Lite využíva koncept spätnej propagácie hodnôt zo zmenených častí grafu – podobne ako D\*, ale s jednou pomocnou dátovou štruktúrou (tabuľkou) navyše oproti A\*.

Okrem stále pomerne zložitej štruktúry algoritmu, v niektorých prípadoch algoritmus nedokáže správne zvládnuť situácie, kde cesta k cieľu neexistuje. (V takýchto prípadoch sa môže algoritmus správať nepredvídateľne alebo zlyhať. Stretli sme sa s ním v článku [5].)

## 1.3 Algoritmy na riešenie konfliktov

Po vyriešení úlohy samotného nájdenia cesty pre jedného agenta nastupuje výzva koordinácie viacerých agentov. Pri súbežnom plánovaní ciest môže dochádzať ku kolíziám, teda k situáciám, kedy sa dvaja alebo viacerí agenti pokúsia obsadiť ten istý priestor v rovnakom čase. Na riešenie týchto situácií sa využívajú rôzne prístupy, ktoré sa líšia výpočtovou náročnosťou, efektivitou, mierou deterministickosti a vhodnosťou pre paralelizáciu.

### 1.3.1 CBS – Conflict-Based Search

CBS je jeden z najrozšírenejších algoritmov pre riešenie konfliktov medzi agentmi. Využíva dvojúrovňovú architektúru: na vyššej úrovni rozdeľuje plánovanie na jednotlivé podproblémy (constraints), ktoré sú riešené pomocou samostatných A\* vyhľadávaní. Ak sa medzi agentmi vyskytne konflikt, algoritmus vytvorí nové vetvy problému, kde sa pre jednotlivých agentov pridajú obmedzenia, ktoré im zakazujú kolidujúce akcie.

CBS však nie je prirodzene paralelizovateľný. Jeho výpočty sú do značnej miery sekvenčné, pretože každá iterácia závisí od predchádzajúcej. Pri paralelizácii je možné spustiť jednotlivé A\* vyhľadávania pre agentov nezávisle, avšak spracovanie konfliktov zostáva sériové. (Detailnejšie sa touto témou zaoberali v [2].)

### 1.3.2 Priority-Based Search (PBS)

PBS rieši konflikty medzi agentmi tým, že im priraduje pevne stanovené priority. Trasa každého agenta je plánovaná postupne, pričom každý agent musí svoju cestu prispôbiť už naplánovaným cestám agentov s vyššou prioritou. Tento prístup je výpočtovo efektívnejší než CBS, ale často vedie k suboptimálnym riešeniam. Jeho hlavná výhoda spočíva v jednoduchšej implementácii a ľahšej možnosti paralelizácie.

### 1.3.3 CA\* – Cooperative A\*

Kooperatívny A\* je modifikáciou klasického A\* algoritmu, kde každý agent plánuje svoju cestu s vedomím o trajektoriách ostatných. Využíva trojrozmerný plánovací priestor (priestor + čas), v ktorom si agenti rezervujú svoje pozície počas plánovania. Tento algoritmus minimalizuje riziko kolízií, no vyžaduje značnú pamäťovú náročnosť, a preto nie je ideálny pre systémy s obmedzenými prostriedkami.

### 1.3.4 Mravčí algoritmus (Ant Colony Optimization – ACO)

ACO je inšpirovaný mravcami. Mravce pri hľadaní potravy skúmajú cesty (každý mravec sám a paralelne s ostatnými) a zanechávajú na nich stopy z uvoľňovaných feromónov, ktoré vylučujú. Princíp ACO je nasledovný:

1. Mravec (agent) ide a zanecháva na ceste, ktorou kráča, feromónovú stopu.
2. Keď nájdu potravu, tak sa obrátia a idú po tej istej ceste do mraveniska.
3. Keď mravec opäť vyjde z mraveniska, má na výber staré stopy (čím silnejšia, tým väčšia pravdepodobnosť, že ju bude mravec nasledovať) alebo bude skúšať s určitou pravdepodobnosťou novú trasu podľa vlastného uváženia. Ak si vyberie druhú možnosť, tak pokračuje krokom 1.

Týmto spôsobom sa zaistí, že kratšie (rýchlejšie) cesty budú viac frekventované, pretože sa mravec s potravou vráti skôr, čím je väčšia šanca, že ju niektorý iný (poprípade ten istý) prejde znovu, čím sa zvyrazňujú feromónové stopy vedúce najrýchlejšie k cieľu.

Tento algoritmus je schopný reagovať na dynamické zmeny v prostredí a je vhodný pre distribuované výpočty, avšak nájdenie optimálneho riešenia s týmto algoritmom môže byť pomalé. V rámci našej práce sme ho nezaviedli priamo, no bol v istých ohľadoch inšpiráciou pre náš algoritmus. Predstavuje však aj perspektívny smer pre budúce rozšírenia.

## 1.4 Zhrnutie kapitoly

V tejto kapitole sme sa venovali prehľadu technológií a algoritmov využiteľných pri paralelnom plánovaní ciest pre viacerých agentov. V úvode sme predstavili technologické nástroje ako CUDA, OpenCL, Vulkan GLSL a SYCL, ktoré umožňujú efektívne využiť GPU pri paralelizácii algoritmov.

Následne sme detailne popísali vyhľadávacie algoritmy ako A\*, D\*, D\* Lite, ktoré slúžia ako základ pre plánovanie trás. Nadviazali sme popisom algoritmov určených na riešenie konfliktov medzi agentmi (CBS, PBS, CA\* a ďalšie), pričom sme poukázali na ich výhody, nevýhody a ich možnosti paralelizácie.

V neposlednom rade sme spomenuli aj bio-inšpirovaný algoritmus ACO, ktorý napriek svojej výpočtovej náročnosti ponúka zaujímavý spôsob adaptívneho plánovania a decentralizovaného riešenia problémov.

Získané poznatky z tejto kapitoly slúžia ako základ pre návrh vlastného riešenia, ktoré bude podrobne rozobraté v nasledujúcej kapitole.

## Kapitola 2

# Analýza súčasných prístupov v literatúre

V tejto kapitole sme sa zamerali na analýzu relevantných článkov, ktoré sme preštudovali pri tvorbe nášho systému. Zamerali sme sa primárne na články, ktoré pojednávajú o autonómnych mobilných robotoch (AMR), ich koordináciou medzi sebou, plánovaní ich ciest (a vo väčšine sme sa stretli aj s problémom lokalizácie, ktorý síce nepatrí úplne do nášho okruhu, ale bude spomenutý pre úplnosť analýzy daných článkov). Snažili sme sa analyzovať existujúce prístupy. Snažili sme sa zistiť ich výhody, limity a potenciál pre paralelizáciu na GPU (pre našu prácu).


### 2.1 Presun od centralizovaných k decentralizovaným systémom

Článok [3], okrem iného, uvádza historický prehľad vývoja AMR systémov od roku 1953. V raných štádiách vývoja sa pozornosť zameriavala na algoritmy s centrálnou riadiacou jednotkou. Ako išla doba, pozornosť sa čoraz viac začala presúvať k algoritmom, ktoré sa vďaka inteligentným komponentom dokázali centrálnej riadiacej jednotke vyhnúť, a tým vznikli decentralizované architektúry. V článku sa venujú aj tomu, aká dôležitá je presná lokalizácia robotov v priestore. V [3] autori, pri decentralizovaných systémoch, opisujú procesy, ktoré sú dôležité pre riadenie autonómnych vozidiel. Tieto procesy možno rozdeliť do štyroch kategórií:

- **Priradenie úlohy robotovi** predstavuje najväčšiu výzvu, pretože ide o NP-ťažký problém s veľkým množstvom možných riešení. Používané riešenia možno rozdeliť na:

- **Exaktné algoritmy** ako MILP, brute-force alebo branch and bound – sú vhodné najmä pre menšie problémy, keďže pri väčších sa stávajú výpočtovo nepraktické.
  - **Heuristiky a metaheuristiky** – napríklad simulované žihanie, tabu search, genetické algoritmy alebo optimalizácia kolónií mravcov – sú efektívne pri stredne veľkých systémoch, no často trpia uviaznutím v lokálnych minimách.
  - **Alternatívne prístupy**, ako hybridné algoritmy kombinujúce viaceré heuristiky, trhové modely s aukciami medzi robotmi, behaviorálne modely s motiváciou alebo terénne riešenia využívajúce atraktory a odpudzovače. Tieto si však často vyžadujú špecifické podmienky.
- **Lokalizácia robota** - mimo virtuálny svet je potrebné lokalizáciu robotov vykonať presne. Na moderných systémoch v reálnom svete sa z väčšej časti realizuje decentralizovane, to znamená, že každý robot si sám zisťuje polohu. Medzi technológie, ktoré sa používajú na to, aby sa roboty vedeli zorientovať v priestore, patria optické, inerciálne, magnetické a indukčné senzory. Bežne sa využíva magnetická mriežka s presne stanovenými súradnicami, GPS (ak je dostupný výhľad na oblohu), laserové navádzanie, LiDAR v kombinácii s 2D mapou (napr. SLAM) a farebné pásy, ktoré využívajú počítačové videnie. Kvôli nároku na vyššiu spoľahlivosť sa často využíva fúzia senzorov, napríklad pomocou Kalmanovho filtra, či inej fúznej metodiky. Vzhľadom na virtuálny charakter tejto diplomovej práce sme lokalizáciu neriešili.
  - **Detekcia prekážok** - autori uvádzajú ako dôležitý krok identifikáciu objektov v prostredí, na ktoré by mohlo vozidlo naraziť. Považovali za dôležité, aby sa detekcie vykonávali presne a rýchlo, pretože to považovali za kľúčové pre fungovanie systému pri vyšších rýchlostiach alebo pri situáciách, kde museli narábať s vyšším počtom agentov v obmedzenom priestore. Naša architektúra počíta so známymi prekážkami, preto ani túto časť nevyužijeme.
  - **Plánovanie najkratšej trasy** - uvádza sa, že sa využívali grafové algoritmy ako Dijkstra, A\* alebo D\*, či D\* Lite. Článok uvádza, že súčasťou je schopnosť preplánovania v reálnom čase, najmä v prípade zistenia novej prekážky. Ich uvádzané systémy mohli využívať koncept „zón“ – napríklad spomalenie v oranžovej zóne alebo zastavenie v červenej. Konfliktne situácie medzi robotmi možno riešiť napríklad pomocou metódy ORCA (Optimal Reciprocal Collision Avoidance), ktorá operuje vo vektorovom priestore a je kompatibilná aj s pokročilými technikami, ako je hlboké učenie.

Path coordinates of Robots								
	R1			R2			R3	
Time(s)	x	y		x	y		x	y
1	18	44		20	44		22	44
2	18	43		20	43		22	43
3	17	43		20	42		22	42
4	17	42		20	41		22	41
5	16	42		20	40		20	40
6	16	41		20	39		23	40
7	15	41		21	39		24	40
8	15	40		21	38		25	40
9	14	40		21	37		26	40
10	14	39		21	36		27	40
11	13	39		21	35		28	40
12	12	39		21	34		28	39



Obr. 2.1: Ukážka kolízie pri zlej koordinácii robotov, zdroj obrázku [5]

## 2.2 Optimalizácia trojice robotov

V článku [5] sa zameriavali na priradovanie úloh pre trojicu robotov a optimalizáciu tohto procesu, kde na priradenie využívali genetický algoritmus a na plánovanie trasy A\* algoritmus. Formálne definície úloh stanovujú jednoznačné pravidlá:

- každý robot môže vykonávať len jednu úlohu - zvládne odviezť jeden náklad
- každá úloha musí byť vykonaná práve raz a každý robot začína z depa.

Cieľom optimalizácie pre ich výskum bola minimalizácia celkovej spotreby paliva, čo reprezentovali spôsobom, že merali prejdenú vzdialenosť robotov a snažili sa ju minimalizovať. Medzi nevýhody patrí slabá detekcia kolízií (obr. 2.1) a obmedzenie na štvorcovú mriežku (obr. 2.2), čo znižuje flexibilitu algoritmu v reálnych podmienkach, ale poskytuje to možnosť pre ľahkú implementáciu.

Hoci to dávali ako nevýhodu, pri detekcii kolízií sme sa inšpirovali týmto spôsobom detekcie kolízií, tak ako je na obrázku 2.1, tak aj náš algoritmus sme navrhli na podobnom princípe.

## 2.3 Automatizované riadenie premávky autonómnych vozidiel na kontajnerovom termináli

Autori článku [4] rozlišujú medzi silne centralizovanými a slabšie centralizovanými systémami. Riadenie môže byť globálne alebo lokálne – delené na zóny. Zóny môžu byť hierarchicky organizované do tzv. „superoblastí“, ktoré spolu komunikujú iba cez styčné body (obr. 2.3).

Riadenie dopravy v takomto systéme je realizované pomocou semaforov. Existuje

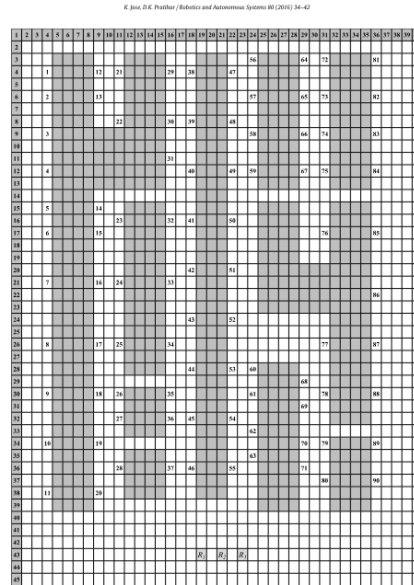


Fig. 5. Planirovanie [11]

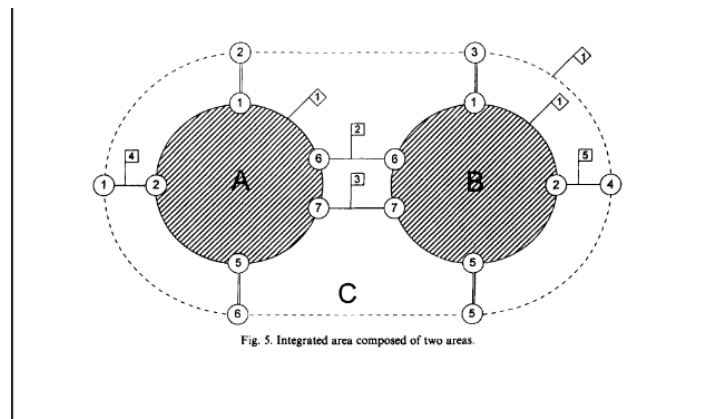
Obr. 2.2: Zjednodušená štvorcová mriežka ako model prostredia, zdroj obrázku [5]

Na obrázku 2.2 je vidieť štvorcová mriežka, v rámci ktorej sa plánujú cesty agentov.

viacero stratégií rozhodovania o prednosti. Medzi najčastejšie používané stratégie rozhodovania o prednosti patria:

- **Náhodné pridelovanie** – prednosť sa priradzuje náhodne bez ohľadu na stav premávky alebo polohu vozidiel.
- **FIFO (First-In-First-Out)** – vstup do oblasti je umožnený podľa poradia príchodu vozidiel. Tento prístup sa dá prirovnať k pravidlu „kto skôr príde, ten skôr ide“.
- **Priorita podľa smeru jazdy** – prednosť má vozidlo, ktoré pokračuje rovnakým smerom ako predchádzajúce vozidlo, čím sa minimalizuje zmena toku.
- **Priorita podľa času príchodu** – systém eviduje presný čas príchodu k zóne a podľa neho rozhodne o poradí.
- **Priorita podľa zaťaženia ďalšej križovatky** – uprednostnené sú vozidlá smerujúce do časti systému, kde je aktuálne najmenší dopravný nápor.
- **Kombinované stratégie** – v praxi sa často využívajú hybridné prístupy, ktoré kombinujú vyššie uvedené metódy podľa aktuálnej situácie v systéme.





Obr. 2.3: Príklad rozdelenia na superoblasti obrázok pochádza z [4]

Na tomto obrázku sú znázornené dve oblasti združené do jednej superoblasti s prístupovými bodmi a semaforom.

## 2.4 Autonómne vozidlá s pevne definovanými trasami

Článok [12] sa zameriava na riadenie autonómnych vozidiel v prostredí, ktoré má pevne definované trasy. Autori navrhovali systém plánovania, ktorý je centralizovaný. Tento systém je vhodný predovšetkým pre prostredia s nízkou hustotou premávky, kde nie je potrebná dynamická adaptácia trás kvôli kolíziám. Plánovanie ciest sa uskutočňuje pomocou Dijkstrovho algoritmu a algoritmu A\*, teda tradičných algoritmov na hľadanie ciest, ktoré sa vyznačujú optimalizáciou dráh z hľadiska vzdialenosti.

Centrálna riadiaca jednotka kontroluje vstup do jednotlivých zón, a to veľmi striktnie. Každý vstup musí byť touto centrálnou jednotkou schválený (musí mať udelené povolenie na vstup). Vozidlá sa v tomto modeli nesprávajú autonómne v zmysle prepočítavania trasy počas jazdy – riadia sa priamočiarou inštrukciou podľa určeného plánu.

Riešenie, ktoré autori navrhli, obsahuje niekoľko významných prvkov:

- **Statické plánovanie trás**, ktoré sa vykonáva pred samotným pohybom robota.
- **Semaforový mechanizmus** - kontrola vstupu do takzvaných kritických sekcií, kde by mohlo dôjsť ku kolízii, napríklad miesta, ako sú križovatky alebo v úzke priestory, kde by potenciálne mohlo dôjsť ku kolíziám.
- **Detekcia cyklov blokovania (takzvaných *deadlockov*)** - pri detekcii systém automaticky preplánuje trasu vozidla, čím sa zabezpečuje obnovenie plynulosti premávky.
- **Priradovanie priorít** - jednotlivým vozidlám na základe faktorov ako naliehavosť úlohy alebo aktuálny stav (napr. vozidlo bez nákladu má nižšiu prioritu ako vozidlo s dôležitou zásielkou).

Autori zohľadňujú skutočnosti, ako sú fyzická šírka a zorné pole vozidla, najmä čo sa týka situácie otáčania vozidla v úzkych uličkách - priestoroch. Tieto problémy autori pomenovali ako takzvané „pink zóny“. Každá „pink zóna“ reprezentuje oblasť, kde by mohlo dôjsť ku kolízii vozidiel so špecifickým tvarom (napr. obdĺžnikové platformy) počas manévrov.

## 2.5 Multi-Agent Path Finding (MAPF)

O problematike plánovania ciest pre viacero autonómnych agentov v spoločnom priestore, teda takzvaný Multi-Agent Path Finding (MAPF), sme sa podrobne dočítali v článku [2]. Dôraz je tu na situácie, kde každému agentovi prislúcha jeden špecifický cieľ. V navrhovanej metóde sa každý agent správa samostatne a nie je prítomná žiadna tímová spolupráca. Takýto pohľad na situáciu odráža scenáre, ktoré sú vo výrobných a logistických systémoch bežným javom. V týchto systémoch vozidlá plnia nezávislé úlohy.

Použitou metódou je **Conflict-Based Search (CBS)**. CBS predstavuje efektívny prístup k riešeniu MAPF úloh. Pri CBS najprv plánujeme trasy pre všetkých agentov nezávisle, napríklad pomocou vyhľadávacieho algoritmu Dijkstra. Následne zistíme prípadné konflikty medzi jednotlivými cestami – napríklad keď by sa dvaja agenti chceli navzájom vymeniť alebo by v rovnaký čas prišli na rovnaké miesto. Ak sa konflikty nenájdu, tak algoritmus môže skončiť a vrátiť trasy.

Tento článok poukazuje na výrazný potenciál algoritmu CBS pre využitie v komplexných prostrediach s vysokou hustotou agentov. Jeho nevýhodou však zostáva sekvenciálny charakter – riešenie konfliktov je vykonávané krok po kroku, čo znižuje možnosti paralelizácie. V kontexte systémov s viacerými procesormi alebo GPU výpočtami by bolo potrebné CBS ďalej modifikovať pre podporu paralelného spracovania konfliktov.

## Kapitola 3

### Ciel' práce

Cieľom tejto diplomovej práce bolo vytvoriť rámec pre simuláciu autonómnych robotov schopných samostatne sa pohybovať po mape s využitím masívneho paralelizmu. Pri tom sme sledovali aj prenositeľnosť na viaceré platformy, nielen v rámci softvéru, ale aj hardvéru. Primárne sme sa zameriavali na využitie grafických kariet pre ich schopnosť vykonávať inštrukcie paralelne vo veľkom počte.



# Kapitola 4

## Návrh experimentov

V tejto časti tejto diplomovej práce si popíšeme návrh experimentov s cieľom overiť výkonnosť navrhutej simulácie pri rôznych podmienkach. Od experimentov požadujeme, aby preverili limity nami navrhnutého systému z hľadiska rôznorodých parametrov, ako je veľkosť mapy, počet agentov, rôzne konštelácie prekážok. V rámci experimentov chceme zároveň porovnať efektivitu jednotlivých implementácií na jednotlivých platformách (GPU a CPU). Budeme sa snažiť dodržať líniu, kde ideme od vysokoúrovňového C++ na CPU až po nízkoúrovňové C (SYCL) na GPU.

Budeme simulovať autonómnych agentov, čo sa ukazuje ako výpočtovo aj pamäťovo náročná úloha, najmä pri použití paralelných techník a pri nutnosti vedieť dopredu veľkosti všetkých používaných štruktúr v rámci programu. Pri algoritmoch vyhľadávania veľkosť dátových štruktúr, ktorých úloha je uchovávať stavy agentov, plánované trasy, mapu, môže veľmi rýchlo narásť. Preto sme museli mať vždy na pamäti maximálnu možnú veľkosť mapy a počet agentov, aby sme sa zmestili do dostupnej pamäte na našom počítači, na ktorom sme testovali náš program.

## 4.1 Pamäťové obmedzenia a motivácia

Platforma SYCL, na ktorej sme postavili náš program, musí rešpektovať limity, ktoré nastavuje samotný hardware. V našom prípade je hlavné pamäťové obmedzenie približne 4 GB pre všetky štruktúry, stack-y (zásobníky), prechodové tabuľky, plánované trasy a pomocné polia na GPU, preto sme tento limit dali ako fixný aj na CPU. S mapou  $N * N$  sú pamäťové nároky síce zložitosti  $O(kn^2)$  pre všetky štruktúry potrebné na výpočet, ale napriek tomu pri rastúcej veľkosti mapy a počte agentov sa tak rýchlo približujeme k tejto hranici. Navyše GPU dokáže synchronizovať len obmedzený počet vlákien (work-item-ov). Preto sa testy budú pohybovať len do určitej hranice, čo sa agentov týka. Maximálny počet agentov sme si teda vopred stanovili na 64. Cieľom testov je sledovať, pri akej kombinácii veľkosti mapy a počtu agentov dochádza k prekonaniu výkonu GPU nad CPU, prípadne či vôbec a ako veľmi.

## 4.2 Rozsah testovaných parametrov

Pre experimentálne testy budeme postupne meniť dva hlavné parametre: budeme skúšať veľkosť mapy a to od veľmi malých rozmerov, napríklad 16 x 16, až po mapy veľkosti 1024 x 1024, prípadne vyššie, až do maxima GPU 4 GB pamäte.

Počet agentov - budeme skúšať parametre v rozsahu od 3 (respektíve 4) po 64 agentov (jeden agent nemá zmysel). Horná hranica zodpovedá bežnému limitu veľkosti work-group pri možnosti synchronizovať GPU paralelizáciu. Tento hardvérový limit platí v OpenCL, Compute Shader-och, a aj v nami zvolenom SYCL, kvôli usporiadaniu jednotlivých výpočtových jednotiek v architektúre.

Ďalšími menšími parametrami budú miesta nakladania a vykladania, poprípade prekážky.

Testovacia matica tak bude obsahovať kombinácie:

- mapy: 16 krát 16, 32 krát 32, 64 krát 64, ..., 1024 krát 1024
- agenti: 3, 4, 7, 8, 15, 16, 31, 32, 63, 64.

Pričom na každých 3 agentov budú dve miesta nakladania a vykladania (minimum 2 z každého). Každá kombinácia bude testovaná vo viacerých režimoch výpočtu.

Nepárne čísla sme navrhli z toho dôvodu, že sme chceli zistiť, či na GPU je výhodnejšie mať počty agentov (v našom prípade work-item-ov) v násobkoch 2 alebo na tom nezáleží.

## 4.3 Testované výpočtové režimy

Aby sme mohli porovnávať výkon jednotlivých výpočtových algoritmov (stratégií), ktoré sme v rámci tejto diplomovej práce implementovali, predpokladáme takéto poradie výkonu (od najpomalšieho po najrýchlejší):

- SYCL funkcie na 1 vlákne (CPU), ktoré síce simuluje princípy paralelizmu, ale reálne je to jedno vlákno, preto predpokladáme, že bude najpomalšie zo všetkých.
- „Obyčajný“ C++ kód (high-level CPU), ktorý je bežným riešením z programátorského prostredia. Predpokladáme, že bude rýchlejší ako SYCL funkcie na 1 vlákne (CPU), pretože využíva vyššie štruktúry (ako mapy), ktoré môžu byť efektívnejšie ako predchádzajúca verzia na tomto zozname. Na druhej strane, SYCL využíva aproximáciu, zatiaľ čo tento variant bude skúšať všetky možnosti.
- Hybridná verzia (GPU A\* + CPU CBS), ktorú sme naprogramovali ako medzistupeň medzi algoritmami na GPU a CPU. Pri hybridnej verzii existuje predpoklad, že bude rýchlejšia ako predchádzajúce dve na tomto zozname, najmä v prípade, keď bude potrebné vyriešiť minimum kolízií.
- SYCL funkcie na viac vlákien (CPU) sa môže ukázať ako efektívna, najmä ak má počítač na procesore viac ako 2 či 4 jadrá. Predpokladáme, že v závislosti od počtu agentov môže byť táto implementácia niekedy rýchlejšia ako verzia čisto na grafickej karte, pretože po výpočte má už výpočty v pamäti.
- SYCL funkcie na GPU je naším favoritom na najrýchlejšiu implementáciu, pretože využíva výpočtové jadrá GPU a natívne synchronizačné mechanizmy. Jednými dvomi faktormi, ktoré môžu znížiť účinnosť tejto implementácie, sú: nižšia taktovacia frekvencia GPU a následné kopírovanie dát z grafickej karty do pamäte procesora.

Každá implementácia bude testovaná s rovnakými vstupmi a výsledky budú zaznamenané do tabuľky vrátane:

- Času (ms), tu budeme rozlišovať medzi časom potrebným na kopírovanie a konvertovanie dát medzi jednotlivými dátovými štruktúrami potrebných na spustenie výpočtu a samotnou dobou potrebnou na uskutočnenie výpočtu.
- Úspešnosti plánovania, a to nielen toho, či sa podarilo nájsť trasu pre všetkých agentov, ale aj veľkosť trasy, pričom C++ implementácia bude slúžiť ako „proof of correctness“.

## 4.4 Metodika merania

Chceme merať čas čo najpresnejšie, a preto sme sa rozhodli, že budeme každý test opakovať viac ako päťkrát a z toho vyrátame priemer. Test bude ukončený úspešným dokončením simulácie. Testy, ktoré by skončili zlyhaním z dôvodu prekročenia pamäťovej kapacity alebo nedostatku výpočtových zdrojov, uvedieme ako hornú hranicu v samostatnej kategórii (ak nejaký taký prípad bude, pretože pokladáme za zbytočné pre 64 agentov robiť väčšie mapy ako 1024 krát 1024). Výsledky z úspešných testov budú vyhodnotené grafmi.

## 4.5 Hardvérové podmienky

Testy budeme tiež skúšať na dvoch operačných systémoch, pričom hlavným testovacím prostredím bude Linux OS (Ubuntu), zatiaľ čo Windows bude vedľajší. Zistili sme totiž, že SYCL na NVIDIA má lepšiu podporu práve na systéme Linux OS. V prípade Windowsu sme využívali SYCL akcelerátor na CPU.

## 4.6 Predpokladané výsledky a hypotézy

Očakávame, že SYCL na GPU dosiahne najvyššiu výkonnosť, avšak hybridná implementácia bude mať najvyššie nároky na synchronizáciu. Hybridný režim môže mať uspokojivé výsledky v pomere rýchlosť verzus optimálne riešenia. Jednovláknové CPU verzie budú slúžiť ako porovnávací vzor, no ich výkon pri veľkých mapách a väčšom počte agentov bude pravdepodobne nedostatočný.



# Kapitola 5

## Postup a implementácia

Táto kapitola opisuje implementáciu nášho riešenia, ktorá je rozdelená do dvoch hlavných častí:

- **Grafické používateľské rozhranie (GUI)** – vizualizácia, zadávanie vstupov a spúšťanie simulácie.
- **Výpočtová logika** – plánovanie ciest agentov, riešenie kolízií a optimalizácia výkonu. Táto časť prešla zmenou logiky riešenia konfliktov počas doladovania finálneho kódu, ale základný kód zostáva rovnaký.

Pri návrhu našej implementácie sme sa inšpirovali poznatkami z analyzovaných článkov v kapitole **Analýza súčasných prístupov v literatúre**. Konkrétne:

- Z článku [3] sme prevzali koncept **rozdelenia úloh** medzi agentov (pričom sme po nejakom čase zaviedli, že agent toto bude riešiť pseudonáhodne), pričom sme sa rozhodli pre jednoduchší prístup pomocou statického rozdelenia na začiatku simulácie. Náročnejšie metódy ako MILP alebo metaheuristiky sme nezaviedli, keďže by boli výpočtovo náročné pre GPU implementáciu v reálnom čase.
- Z článku [5] sme sa zamerali na **kombináciu genetického prístupu pre priraďovanie úloh a A\* pre plánovanie**, ale v našej implementácii sme sa rozhodli použiť iba A\* algoritmus na plánovanie trás kvôli jeho deterministickosti, efektívnosti a jednoduchosti paralelizácie.
- Článok [4] nám poskytol náhľad na **semaforové riadenie križovatiek** – v našej implementácii sme sa rozhodli vytvoriť primitívny model pozostávajúci iba z mriežky, kde v každej bunke vie byť maximálne jeden agent, no pokročilé priority podľa smeru alebo fronty, či vyššie štruktúry boli z dôvodu zložitosti a nutnosti veľkého množstva potrebných synchronizácií, ktoré by spomaľovali výpočet na grafickej karte, vynechané.

- V článku [2] bol predstavený **CBS (Conflict-Based Search)**. Hoci jeho výhody sú zrejmé, kvôli jeho sériovej povahe sme ho zaviedli iba v hybridnej verzii a v C++ verzii, ktorá slúži ako kontrolný a porovnávací výpočet, kde sa CBS rieši na CPU a v hybridnej verzii sa samotné trasy agentov plánujú paralelne pomocou A\* na grafickej karte.

Tieto rozhodnutia boli urobené na základe kompromisu medzi výpočtovou náročnosťou, paralelizovateľnosťou a celkovou zložitou implementácie.

Pre lepšiu modularitu a flexibilitu sme zvolili komponentovo orientovaný návrh. Každá časť systému (grafika, výpočty, správa scén) je implementovaná samostatne a medzi komponentmi prebieha komunikácia cez ukazovatele a referencie.

## 5.1 Grafické používateľské rozhranie (GUI)

GUI slúži prevažne na demonštráciu a vizualizáciu na menších mapách, ale napriek tomu sme ho v tejto sekcii podrobne popísali.

### 5.1.1 Architektúra GUI

GUI bolo naprogramované modulárne. Program má naprogramovaných viacero akcií, ktoré sú združené do jednotlivých scén (okien, pohľadov). Našou základnou triedou je trieda **Scene**. Každá zo scén, ako napríklad hlavné menu, nastavenie parametrov, náhľad mapy a simulácia, je implementovaná ako odvodená trieda. Celkovo sme vytvorili týchto 7 tried:

- **Hlavné menu** – obsahuje možnosti „Nová mapa“ a „Načítať mapu“. Implementované v triede **MainScene**.
- **Nastavenie parametrov** – umožňuje zadanie rozmerov mapy, počtu agentov a iných prvkov cez **GuiValueBox**. Implementované v triede **ParameterScene**.
- **Náhľad a úprava mapy** – vizualizácia mapy, pridávanie a mazanie objektov (agentov, prekážok, loaderov a unloaderov). Realizované v triedach **ViewScene** a **ChangeScene**.
- **Simulácia** – vykresľuje a riadi priebeh simulácie. Implementované v triede **SimulationScene**, ktorá spúšťa výpočtové vlákna a synchronizuje pohyb agentov.
- **Načítanie uložených máp** – umožňuje výber zo zoznamu uložených máp a ich načítanie. Implementované v triede **LoadScene**.

V rámci modulárneho prístupu každá z vyššie uvedených tried dedí a v rámci polymorfizmu implementuje funkciu **DrawControl()**, kde každá trieda má za úlohu vykresliť samu seba a je zodpovedná za prijímanie a usmernenie užívateľských vstupov.

### 5.1.2 Grafické knižnice - komponenty

- **Raylib** ako základná knižnica pre implementáciu grafického rozhrania. Táto knižnica poskytuje tiež základný prístup k získaniu vstupov od používateľa.
- **Raygui** poskytuje rozšírenie prvej knižnice o ovládacie prvky, ako sú tlačidlá, textové polia a posuvníky.

### 5.1.3 Zásady/Princípy návrhu

- **Event-driven programovanie** – kde je všetko riadené vstupmi od používateľa a následnou reakciou na tieto vstupy.
- **Multithreading** – v našej aplikácii využívame jedno vykresľovacie vlákno na kreslenie a event-driven tok informácií, no na výpočty a pohyb agentov využívame vlákna, ktoré sú takmer nezávislé od vykresľovacieho vlákna.

### 5.1.4 Správa scén a prechodov

Prechod medzi scénami funguje na princípe návratu novej inštancie scény z `DrawControl()`. Staré scény sa dealokujú a nové preberajú kontrolu. Výpočtové štruktúry sa prenášajú ako smerníky. Tento prístup sme zvolili, pretože sme chceli jednoduché prepínanie medzi obrazovkami a ľahkú rozšíriteľnosť v prípade nejakého pokračovania našej aplikácie do budúcnosti o nové typy scén.

### 5.1.5 Vlákna a simulácia

Triedu `MemSimulation` sme použili ako medzistupeň medzi grafickou časťou a výpočtovou časťou, aby sme zabezpečili spúšťanie simulácie v oddelenom vlákne od GUI. V prípade, že nebeží žiadna výpočtová úloha a používateľ spustí simuláciu.

Na samotný výpočet sa vytvorí nové vlákno, v ktorom prebehne samotný výpočet simulácie pomocou funkcie `letCompute()`. Výsledky výpočtu sa ukládajú do štruktúry `info` a označujú ako pripravené, čo pri prvom vykresľovanom cykle spôsobí výmenu scén.

Detailný zdrojový kód funkcie `runSimulation()` sa nachádza v Prílohe 6.3.

### 5.1.6 Prepojenie medzi scénami

Každé zavolanie funkcie `DrawControl()` môže, podľa užívateľských akcií, vrátiť referenciu na triedu, ktorá ju polymorfne zavolať, alebo referenciu na úplne novú triedu odvodenú od triedy `Scene`, ktorá má splniť požiadavky užívateľa. Príkladom môže byť situácia, v ktorej po stlačení tlačidla „new map“ v hlavnom menu (teda `MainScene`)

funkcia `DrawControl()` vráti novú inštanciu na triedu `ParameterScene`. V prípade návratu novej triedy program dealokuje starú triedu a pokračuje s novou scénou. Na tomto princípe funguje prepínanie medzi scénami. Niektoré triedy si odovzdávajú výpočtové štruktúry, ktoré sú zodpovedné za výpočty v programe. Tieto štruktúry, ako píšeme v podkapitole 5.2, musia byť kompatibilné s nízkoúrovňovým C, preto sa medzi triedami odovzdávajú ako smerníky. Keď odovzdávajú smerníky na výpočtové štruktúry, každá trieda má povinnosť nastaviť si interné ukazovatele na nulový smerník, aby sa pri ich odstraňovaní neodstránila aj výpočtová štruktúra programu. Zároveň sa týmto prístupom zaručí, že ak program má skončiť, tak dealokuje aj výpočtovú štruktúru. Inšpiráciou pre nás bol aj model, ktorý používa programovací jazyk Rust na správu pamäte [6].

### 5.1.7 Spracovanie užívateľských vstupov

Knižnica Raygui umožňuje niekoľko spôsobov, akými môže užívateľ zadávať vstupy. Náš program prijíma od používateľov vstupy tromi spôsobmi:

- `GuiButton()` – spracováva kliknutia na tlačidlá. Toto je najbežnejší spôsob interakcie v programe.
- `GuiValueBox()` – umožňuje zadávanie hodnôt do textových polí. Možnosť využívaná najmä pri nastavovaní parametrov mapy a simulácie.
- `IsMouseButtonPressed()` – deteguje kliknutie na konkrétnu oblasť v rámci mapy. V triede `ChangeScene` sa táto varianta využíva na označenie políčok, ktoré chceme vymeniť alebo do neho niečo pridať.

Keď sa dokončí výpočet, výsledok sa odovzdá pomocou vnútorných premenných triedy `SimulationScene`, ktorá po nastavení príslušných premenných odovzdá výpočtovú časť triedy `InfoScene`, kde sa zobrazí celkový čas výpočtu a synchronizácie.

Tu uvádzame tabuľku prechodov medzi jednotlivými scénami (podrobnejšie v 5.1):

Tabuľka 5.1: Prechody medzi scénami

Scéna	Možné prechody
MainScene	ParameterScene, LoadScene
ParameterScene	ViewScene, MainScene
ViewScene	ParameterScene, ChangeScene, MainScene
ChangeScene	SimulationScene, ViewScene
SimulationScene	InfoScene, MainScene
InfoScene	SimulationScene
LoadScene	ViewScene, MainScene

Obrázky jednotlivých scén sa nachádzajú v 6.3.

## 5.2 Výpočtová logika

### 5.2.1 Prehľad výpočtovej časti

V tejto časti sa zameriame na výpočtový aspekt aplikácie. Naším hlavným problémom bolo nájdenie optimálnych ciest pre agentov v mape a následné vyriešenie kolízií, ktoré medzi agentmi mohli vzniknúť. Na vyhľadávanie cesty sme použili algoritmus  $A^*$ , ktorý sme rozšírili o správu obmedzení (constraints) a riešenie konfliktov pomocou CBS (Conflict-Based Search) na CPU, a nami navrhnutého algoritmu, inšpirovaného „mravčím algoritmom“ (ant colony alg) . Popíšeme tu viacero výpočtových metód, ktoré sme implementovali, pričom každá metóda sa snaží o rýchle plánovanie ciest s minimálnym výpočtovým nákladom.

Hlavné výpočtové operácie sú implementované paralelne s využitím:

- **$A^*$  algoritmu** - tento algoritmus má implementáciu ako na CPU, tak aj na GPU. Na CPU využíva štruktúry z C++, rovnako aj funkcie, ktoré sú napísané v SYCL, sa volajú rovnako na CPU.
- **CBS (Conflict-Based Search)** - implementovaný iba na CPU, pretože GPU nemá kapacitu na dynamické alokovanie pamäti, čo táto metóda potrebuje na svoj základný beh.
- **Hybridného modelu  $A^*$  + CBS** - volá plánovanie ciest v GPU a kombinuje to s volaním CBS.
- **Riešenia pomocou SYCL** - sem patria tri algoritmy, z ktorých jeden je simulácia paralelizmu pomocou dátových štruktúr, druhý je plne paralelizované riešenie, ktoré je pomocou klasických vlákien v programovacom jazyku C++ a tretím algoritmom je implementácia na GPU.

Pri prvom a druhom riešení sme použili synchronizačné mechanizmy, ktoré boli dostupné na danej architektúre. V prípade CPU sme použili našu implementáciu bariéry a v prípade GPU natívnu implementáciu bariéry. Tieto mechanizmy boli veľmi dôležité pri riešení kolízií.

### 5.2.2 Dátové štruktúry a správa pamäte

Ako už bolo spomenuté, SYCL je síce vyššia úroveň abstrakcie ako OpenCL alebo Vulkan, no stále bolo efektívnejšie použiť C-štýlové štruktúry namiesto C++ tried. Hlavné dátové štruktúry:

- **Struct Agent** – obsahuje informácie o aktuálnej a cieľovej pozícii agenta, smer pohybu a stav čakania.
- **Struct Position** – reprezentuje súradnice bodu v mape.
- **Struct Constraint** – uchováva obmedzenia pre pohyb agentov.
- **Struct MemoryPointers** – obsahuje všetky ukazovatele na dáta (mapu, agentov, cesty, náklady atď.) pre CPU aj GPU výpočty. Táto štruktúra umožňuje efektívny prenos dát medzi CPU a GPU.

### 5.2.3 Prístupy – GPU a CPU

Na dosiahnutie porovnania výkonu rôznych stupňov paralelizácie sme zvolili prístupy (obrázok nižšie 5.3):

- **high-level (vysokoúrovňový) výpočet** - A\* algoritmus sa vykonáva na CPU. Tu sme chceli mať kontrolu „proof of correct“, a zároveň sme chceli porovnanie výkonu C++ s ďalšími výpočtovými modelmi. Riešenie konfliktov sa zabezpečí algoritmom CBS (popísaný v kapitole Súčasný stav a výskum). Tento model je jednovláknový.
- **low-level (nízkoúrovňový, jedno vlákno) výpočet** - A\* sme napísali v SYCL funkciách v takmer čistom C (až na malé drobnosti v syntaxi). Model síce používa SYCL funkcie, ale vykonáva sa na CPU v jednom vlákne, ktoré postupne prechádza všetkých agentov a postupne rieši konflikty, v podstate simuluje paralelizáciu v jednom vlákne. Hoci beží podobne ako **high-level (vysokoúrovňový) výpočet** na jednom vlákne, spôsob riešenia konfliktov je inšpirovaný algoritmi PBS a ACO. CBS sme nepoužili, pretože tento algoritmus už mal simulovať princípy, ktoré sme chceli implementovať na GPU, a kvôli sekvenčnej povahe CBS nie je možné ho implementovať na GPU. Navyše CBS vyžaduje vysokoúrovňové štruktúry a dynamicky alokovanú pamäť, čo sú veci, ktoré sa implementujú ťažko alebo sa nedajú implementovať vôbec.
- **low-level (viac vlákien) výpočet** - A\* sa simultánne počíta na viacerých vláknach, pričom platí, čo agent, to vlákno. Rovnako ako v predchádzajúcich prípadoch, aj tu sa výpočty vykonávajú na CPU. Funkcie sú tie isté ako v modeli **low-level (jedno vlákno) výpočet**. Použili sme tu umelou inteligenciou vygenerovanú a nami trochu upravenú triedu Barrier, ktorá má simulovať bariéru na GPU, zároveň sme použili prístup inšpirovaný algoritmi PBS a ACO.

- **hybridný výpočet** - počiatočné trasy agentov sa počítajú paralelne na GPU pomocou algoritmu A\*. Konflikty medzi agentmi rieši CBS algoritmus bežiaci na CPU. Viacvláknové spracovanie na CPU umožňuje efektívne riešenie konfliktov.
- **výpočet čisto na GPU** - vďaka SYCL bežia na GPU tie isté funkcie ako v modeloch **low-level (jedno vlákno) výpočet** a **low-level (viac vlákien) výpočet**. Pri tomto modeli sme použili natívnu GPU bariéru a, podobne ako pri predchádzajúcich spôsoboch riešenia konfliktov, vychádza z algoritmov PBS a ACO.

V nasledujúcej tabuľke sme zhrnuli vlastnosti algoritmov, ktoré sme navrhli, ich čísla budú 1. Vysokoúrovňový kód (High-level CPU), 2. Nízkoúrovňový kód (Low-level – jedno vlákno)), 3. Nízkoúrovňový kód (Low-level – viac vlákien), 4. Hybridný model, 5. GPU-only:

Tabuľka 5.2: Porovnanie výpočtových prístupov

Model č.	A* vykonávanie	Riešenie konfliktov	Paralelizácia	Poznámky
1	CPU	CBS na CPU	Nie (1 vlákno)	C++ verzia, kontrolný kód
2	CPU	PBS/ACO inšpirácia	Simulovaná	SYCL funkcie, 1 vlákno
3	CPU	PBS/ACO inšpirácia	Áno (1 vlákno / 1 agent)	Synchronizácia cez Barrier triedu
4	GPU	CBS na CPU	Áno (GPU + viac CPU vlákien)	Kombinované výhody CPU/GPU
5	GPU	PBS/ACO inšpirácia	Áno (GPU natívne)	GPU bariéry a atomické operácie

V prípade konfliktu sa uplatňujú pravidlá:

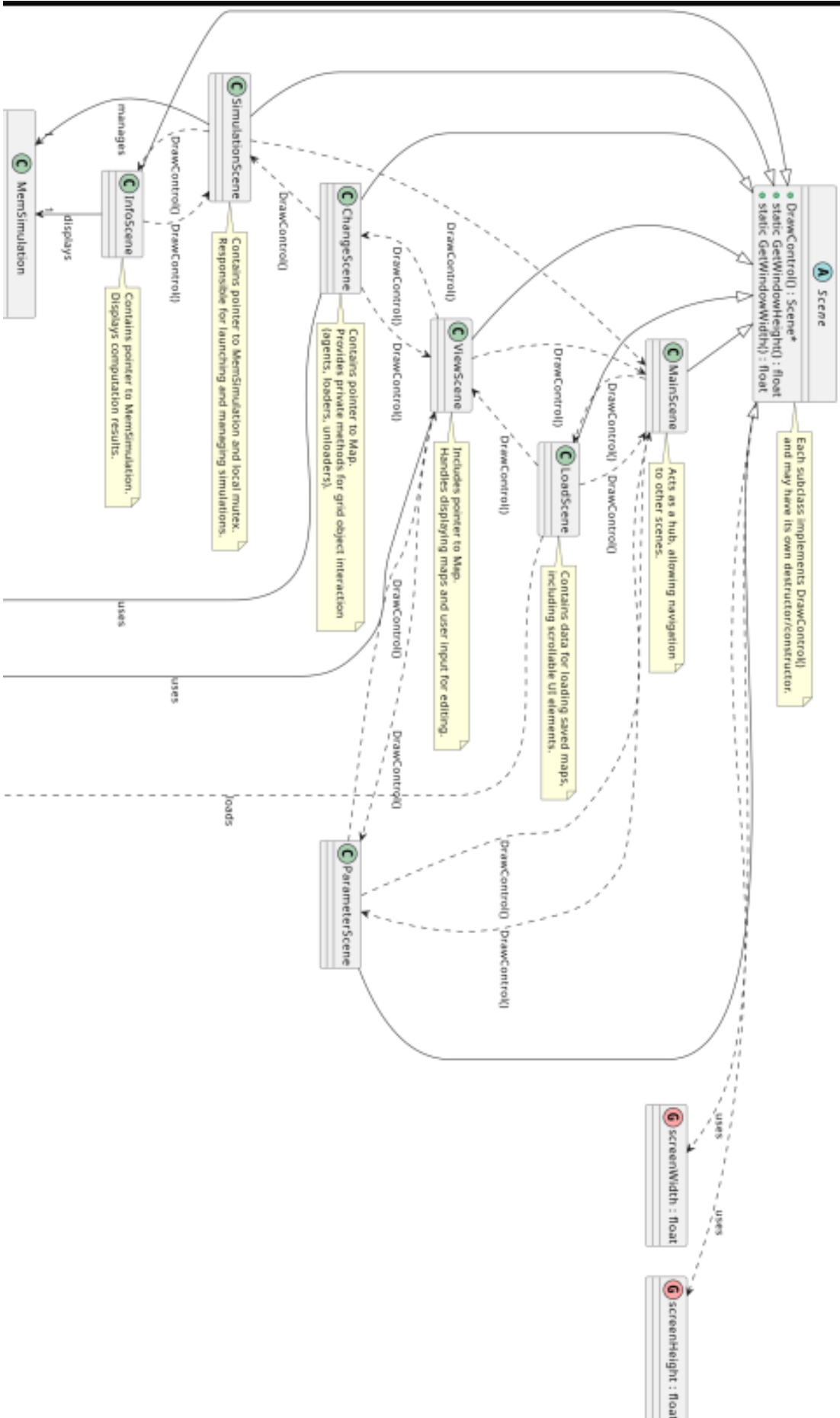
- Agent s vyššou prioritou má prednosť. Priorita sa vypočítava na základe smerovania k vykladaču, dĺžky trasy a ID.
- Ak agent nemôže pokračovať, prepočíta si novú trasu so zapracovaním obmedzení.
- Ak nové riešenie neexistuje, agent hľadá dočasné parkovacie miesto.

## 5.3 Grafy

Pre lepšie pochopenie architektúry a toku výpočtu v nami navrhnutom systéme uvádzame nasledujúce grafy spolu s ich popisom. Na týchto diagramoch ukážeme vizualizáciu nášho návrhu. Tieto diagramy ukazujú dynamiku a kompozíciu tried a logického spracovania výpočtov, ktoré sa dejú počas simulácie autonómnych agentov.

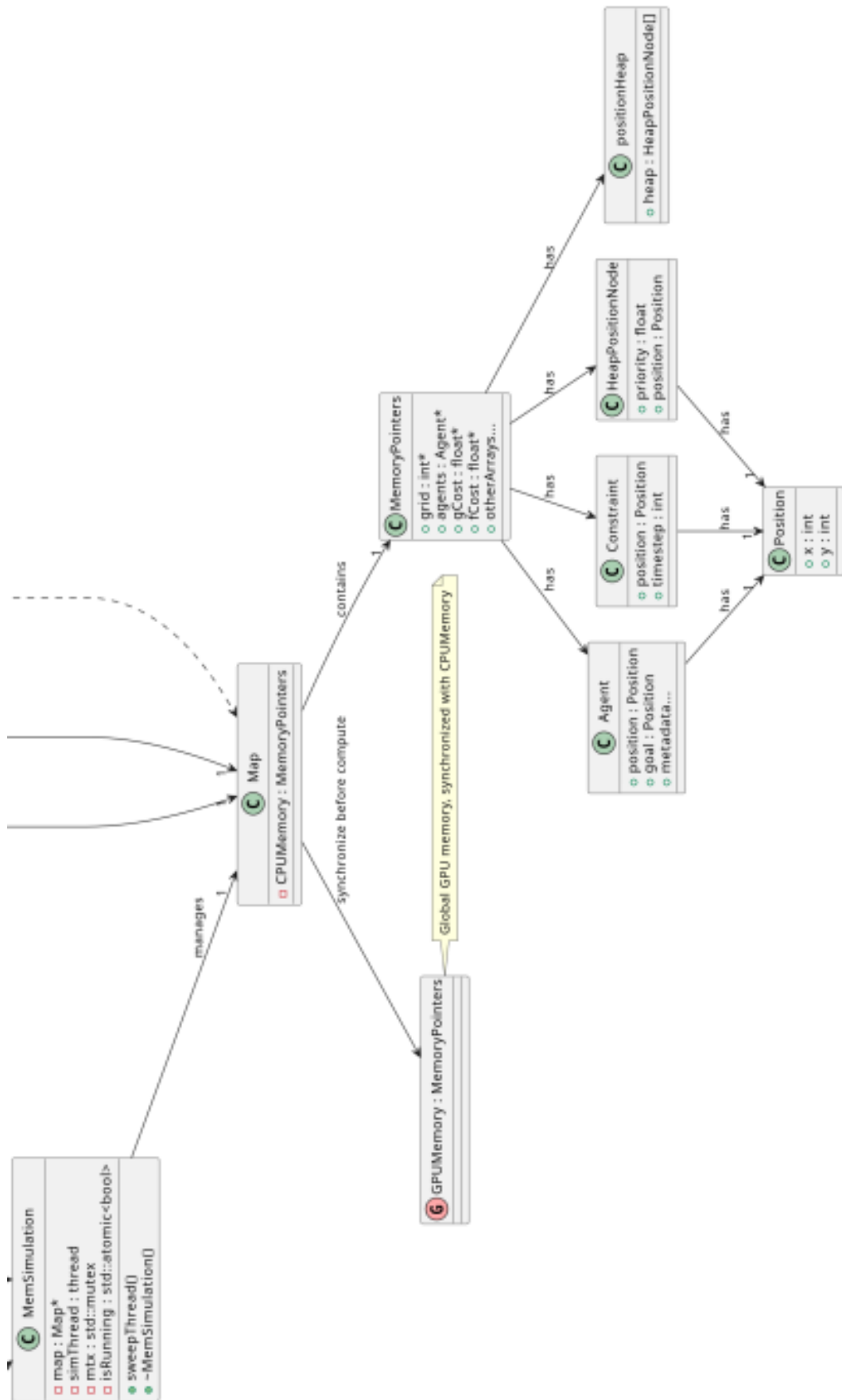
Obrázky 5.1 a 5.2 znázorňujú architektonickú štruktúru previazaností tried (hlavne z GUI), a znázorňuje aj, ktoré triedy z GUI priamo interagujú s výpočtovými štruktúrami.

Triedy sú rozdelené do viacerých hlavných skupín:



Obr. 5.1: Graf tried – GUI





Obr. 5.2: Graf tried – výpočet

- **Triedy (štruktúry) pre reprezentáciu mapy** (na grafe znázornené ako `Map`, `Position`, `Constraint`, ...) zabezpečujú správu „simulovaného sveta“, teda mriežky a manipuláciu s navigačnými bodmi.
- **Štruktúra agentov** (v grafe znázornené `Agent`) je výpočtová štruktúra obsahujúca všetky dáta, ktoré *work-item*, v našom prípade reprezentujúci samostatne plánujúceho agenta, potrebuje o sebe vedieť.
- `SimulationScene` volá implementáciu vyhľadávacích a riešiacich algoritmov.
- **Ostatné GUI komponenty** (`MainScene`, `LoadScene`, `ViewScene`, `ChangeScene` a `InfoScene`) umožňujú vizualizáciu a interaktívne ovládanie simulácie.

Diagram znázorňuje rôzne vzťahy medzi triedami v našom programe:

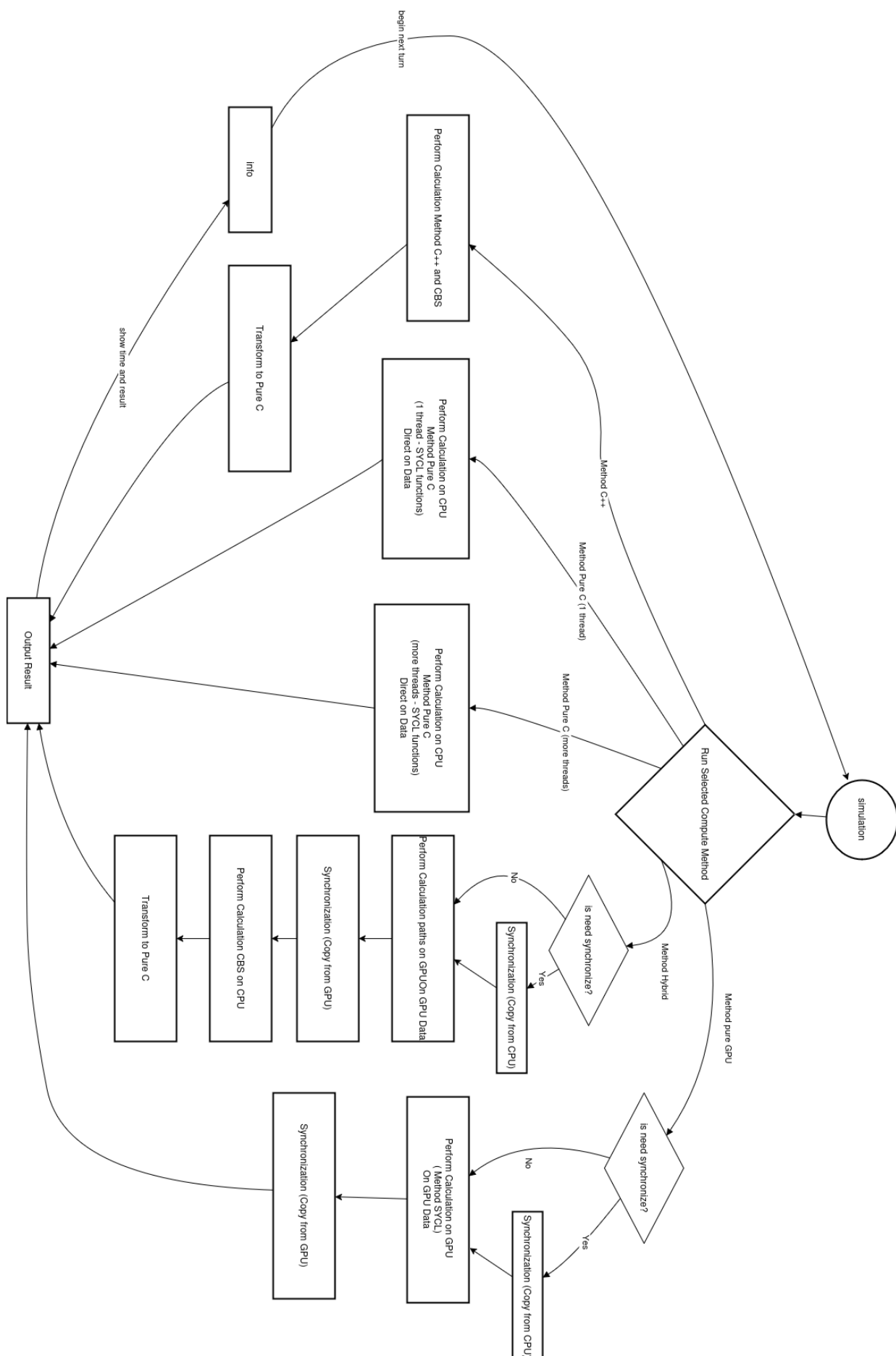
- **Kompozícia** (plné šípky) označuje vlastnícke vzťahy, jediné sporné vlastníctvo je medzi `GPUMemory` a `Map`. `GPUMemory` je v našom kóde síce globálna premenná, ale z praktického zľadiska plne prislúcha k triede `Map`.
- **Dedičnosť** (prázdné šípky) zachytáva generalizáciu a špecializáciu.
- **Asociácie** znázorňujú voľné prepojenia medzi triedami. Triedy pri `DrawControl()` môžu vrátiť seba, ale aj ďalšiu triedu v poradí k nej prislúchajúcom.

Architektúra bola navrhnutá s cieľom umožniť budúce rozšírenia systému.

Obrázok 5.3 znázorňuje procesné kroky v rôznych vetvách (podľa výberu algoritmu), ktoré prebiehajú počas simulácie jednej z týchto vetiev. Rôzne vetvy sa dajú rozdeliť rôzne. Predstavme si najzákladnejšie:

1. **Kopírovanie dát:** - hoci sú na začiatku simulácie všetky štruktúry, ktoré sú potrebné na výpočet, predalokované a skopírované na grafickú kartu, vždy po každom kroku si treba overiť, či sú dáta aktuálne (najmä na GPU, kde kvôli možnosti výpočtov na procesore dochádza k desynchronizácii).
2. **Plánovanie trás:** - paralelné alebo sekvenčné spustenie algoritmu vyhľadávania ( $A^*$ ) pre každého agenta nad úsekom jemu vyhradenej pamäti. Každý agent je plánovaný samostatne, pričom využíva vlastné pamäťové štruktúry na danom úseku pamäte.
3. **Detekcia kolízií a korekcia:** - po plánovaní trás sa vykonáva kontrola kolízií medzi agentmi. V prípade kolízie agent spomalí (zdvojenie pozície), v pôvodnom návrhu sa riešila dynamickým preplánovaním.
4. **Synchronizácia:** - výsledky výpočtu sa synchronizujú, pričom sa zabezpečuje konzistentnosť medzi viacerými agentmi. V GPU verzii sa využívajú natívne synchronizačné mechanizmy (bariéry), v CPU verzii mutexy alebo iné mechanizmy.
5. **Výstupné spracovanie a vizualizácia:** - výsledky výpočtu sa, v prípade výpočtov na GPU, synchronizujú. V GPU verzii sa využívajú natívne synchronizačné mechanizmy (bariéry), v CPU verzii sa výpočty vykonávajú na štruktúrach, ktoré už sú v pamäti. V prípade C++ verzie sa všetky vyššie štruktúry konvertujú na polia a robí sa tzv. „flat“ do jednodimenzionálneho poľa v predalokovanej pamäti.
6. **Kontrola chyby** - keďže GPU, teda ani kernel SYCL, nevie vyhadzovať chyby, použili sme techniku flag-ov, aby sme si signalizovali von z kernela, ktorý agent má problémy. Útržok z kódu 6.3. Keďže sme navrhli program tak, aby generoval mapy, v ktorých agent vždy môže dosiahnuť cieľ, takéto prípady by ani nemali nastať, ale pamätali sme aj na chybové prípady.

Ako bolo spomenuté a možno vidieť na obr. 5.3, výpočet môže bežať paralelne aj sekvenčne podľa výberu.



Obr. 5.3: Znáznornenie toku výpočtu

## 5.4 Prehodnotenie návrhu

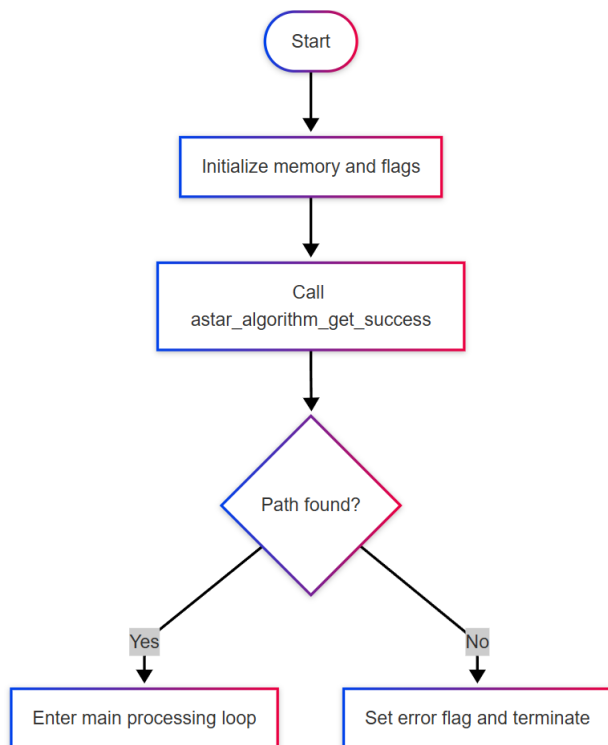
Pri vývoji prvej verzie programu sme narazili na problém, že nami navrhované riešenie kolízií počítalo s tým, že v prípade konfliktu si agent s menšou prioritou musí znovu prerátať cestu a nájsť alternatívu (ako sa určuje priorita, sme si popísali v sekciách vyššie). Túto ideu sme implementovali aj do CBS aj do ostatných algoritmov. Navrhli sme, že ak následná kontrola zistí, že alternatívna cesta z bodu A do miesta B neexistuje, tak agent si mal nájsť „parkovacie miesto“, teda miesto, ktoré bude niekde mimo trasy patriacej agentovi, s ktorým mal kolíziu. Naprogramovali sme to tak, aby prešiel trasu agenta, s ktorým mal kolíziu, a vyskúšal miesta vedľa tejto trasy. Počas implementácie sme si však uvedomili, že toto riešenie má principiálnu chybu v tom, že ak bude riešiť „iba“ jedného agenta (toho, s ktorým mal kolíziu), tak môže zahatať cestu inému. Následne pri riešení tejto (indukovanej) kolízie by potenciálne mohol napláňovať nové „parkovacie miesto“ práve v mieste pôvodnej kolízie. To by spôsobilo, že by sa program cyklil do nekonečna. To bol problém, ktorý by vyžadoval ďalší výskum a urobilo by to SYCL, ale aj C++ algoritmy veľmi zložitými (pozri obrázky 5.4, 5.5a, 5.5b a 5.6) a náchylnými na chyby.

Preto sme prehodnotili náš návrh systému plánovania trás a zvolili sme prístup, ktorý namiesto komplexného dynamického riešenia kolíznych situácií preferuje jednoduché a deterministické správanie agentov. V prípade kolízie agent dočasne spomalí, čo v našej implementácii predstavuje zotrvanie na danej pozícii počas dvoch po sebe idúcich krokov. Tento spôsob modelovania správania agentov umožňuje zachovať konzistentnosť a predvídateľnosť plánovania bez potreby zohľadňovať všetky možné interakcie medzi agentmi v reálnom čase.

Zvolený prístup zároveň eliminuje potrebu navrhovať zložité mechanizmy tzv. „parkovania“, ktoré by si vyžadovali schopnosť agentov dynamicky reagovať na okolie a meniť trasu počas vykonávania plánovania. Hoci ide o zjednodušenie oproti reálnym scenárom autonómnych systémov, umožňuje lepšie analyzovať a testovať správanie algoritmu v paralelnom prostredí a vytvára predpoklady pre jeho budúce rozšírenie o pokročilejšie techniky riešenia kolízií. V rámci zjednodušenej verzie sme už nepotrebovali uchovávať „constraints“, keďže konflikt vyriešime raz a nepotrebuje ho riešiť. (V prvej verzii sme si ich potrebovali pamätať.)

Táto zmena bola zavedená ako súčasť druhej verzie implementácie, pričom aj jej prvý prototyp sa nachádza na [8], v pod adresári „diplomovkaverzia01/diplomovka02“ spolu s ďalšími experimentálnymi backlogmi ako napríklad „D\_star\_backlog“, ktorým sme sa neúspešne pokúšali implementovať *D star lite* grafickej karte aj na C++ verzii. Cieľom bolo nájsť vhodné riešenie, ktoré by bolo robustné a vhodné pre paralelizáciu. Nakoniec sme však zvolili zjednodušenie (popísané vyššie) s „obyčajným“ *A star*. Cieľom bolo dosiahnuť robustnosť základného systému a čo najpodobnejšie algoritmy

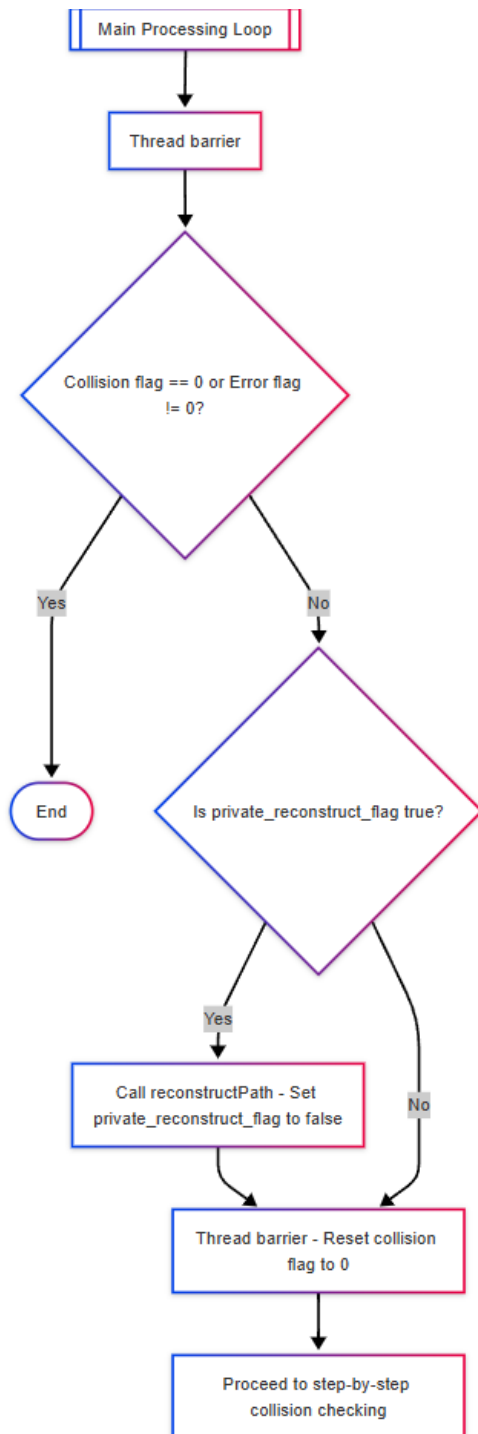
medzi výpočtom na grafickej karte a výpočtom na procesore. Tieto backlogy však môžu poslúžiť ako odrazový mostík pre prípadný ďalší výskum. Finálna verzia sa nachádza v podadresári „diplomovkaverzia01/final01“.



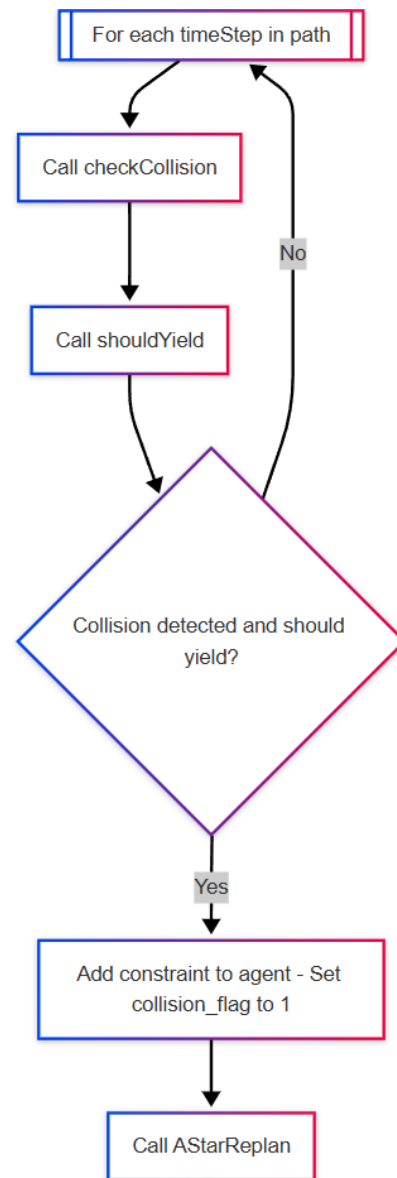
Obr. 5.4: Začiatok algoritmu pre jedného agenta

Na obrázku 5.4 možno vidieť počiatočnú fázu algoritmu, v ktorej sme naprogramovali inicializáciu atomických premenných a každému agentovi plánovanie trasy od svojej polohy k cieľu. V tejto fáze sme ešte nebrali do úvahy možné kolízie ani v prvej verzii nášho programu. Cieľom agenta v tejto časti je nájsť optimálnu cestu pomocou algoritmu A\* bez akýchkoľvek obmedzení. To poskytne prvé cesty, na ktorých sa môžu hľadať kolízie.

Na obrázkoch 5.5a a 5.5b vidieť hlavný cyklus algoritmu, ktorý je kvôli čitateľnosti rozdelený na dve časti. Prvá časť (5.5a) znázorňuje vonkajší cyklus (*while*), v ktorom sa nachádza znovu vytváranie trasy (čiastočne inšpirované ACO algoritmom), synchronizácia a kontrola flagov, zatiaľ čo druhá časť (5.5b) popisuje vnútorný cyklus (*for*) priebeh kontroly kolízií krok po kroku medzi všetkými agentmi. Tento cyklus umožňuje dynamické riešenie konfliktov v reálnom čase a minimalizáciu potreby úplného preplánovania.

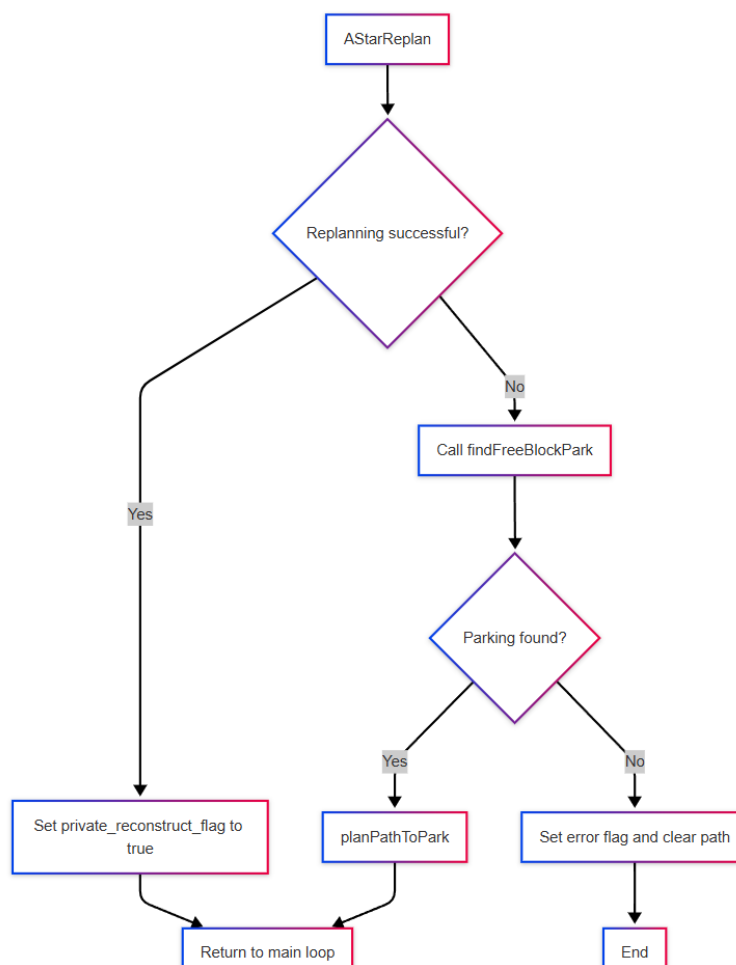


(a) Main Loop – prvá časť



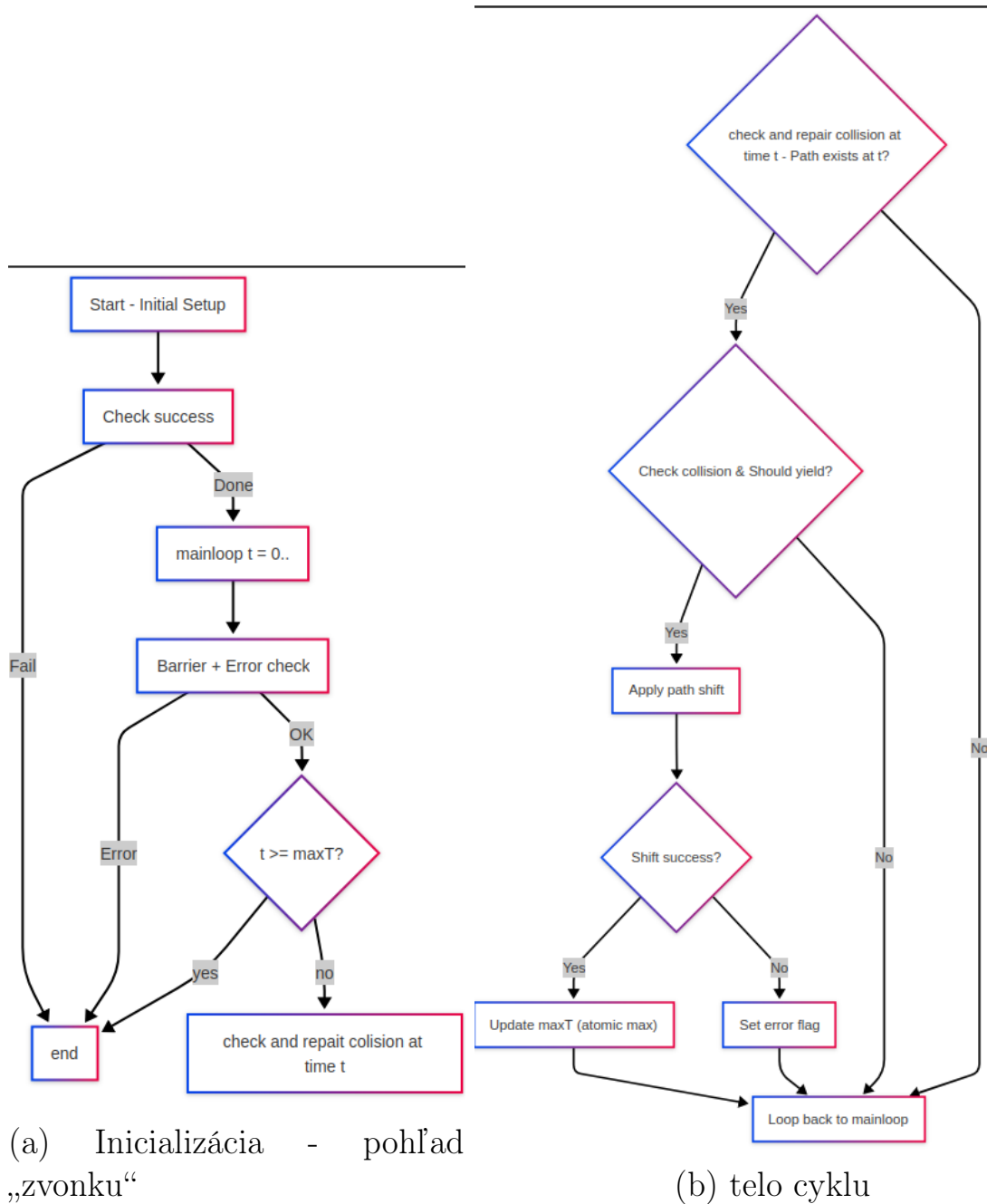
(b) Main Loop – druhá časť





Obr. 5.6: Hľadanie „parkovacieho miesta“ a koniec algoritmu

Obrázok 5.6 predstavuje fázu algoritmu, v ktorej sa algoritmus pokúsil preplánovať cestu a podľa zdaru či nezdaru sa buď vráti k hľadaniu kolízií alebo pristúpi k hľadaniu „bezpečného parkovania“, kde sa hľadá prvé voľné miesto vedľa cesty agenta, ktorému sa má vyhnúť.



Obrázok 5.7a predstavuje pohľad na inicializáciu algoritmu a pohľad „zvonku“ na cyklus, ktorý prechádza všetky pozície a kontroluje kolízie.

Na obrázku 5.7b je vidieť samotné telo cyklu, ktorý má za úlohu rozpoznať kolíziu a „spomaliť“ agenta, čo sa značí duplikovaním pozície.

# Kapitola 6

## Experimenty a výsledky

### 6.1 Porovnanie výpočtového času CPU a GPU pri hashovaní

Aj keď táto experimentálna analýza nepredstavuje priamu súčasť hlavného riešenia práce, bola vykonaná ako podporný test pri štúdiu paralelizácie pomocou SYCL. Naším cieľom bolo získať orientačný pohľad na rozdiel vo výkonnosti medzi výpočtom v C++ na CPU a jeho ekvivalentom spusteným na GPU s využitím SYCL backendu. Testovacím procesom bol *AMD Ryzen 9 5950X 16-Core Processor*, ktorý je skôr serverovým typom procesora so 16 jadrami a 32 hyperthreadovými vláknami a grafickou kartou *NVIDIA GeForce RTX 3060 Ti*.

Implementovali sme jednoduchý „kalibračný program“, ktorý sa nachádza v [8] (v podadresári „sha2“), a ktorý mal vyhľadávať hodnotu  $x$ , pre ktorú platí (v pythonovskom zápise):

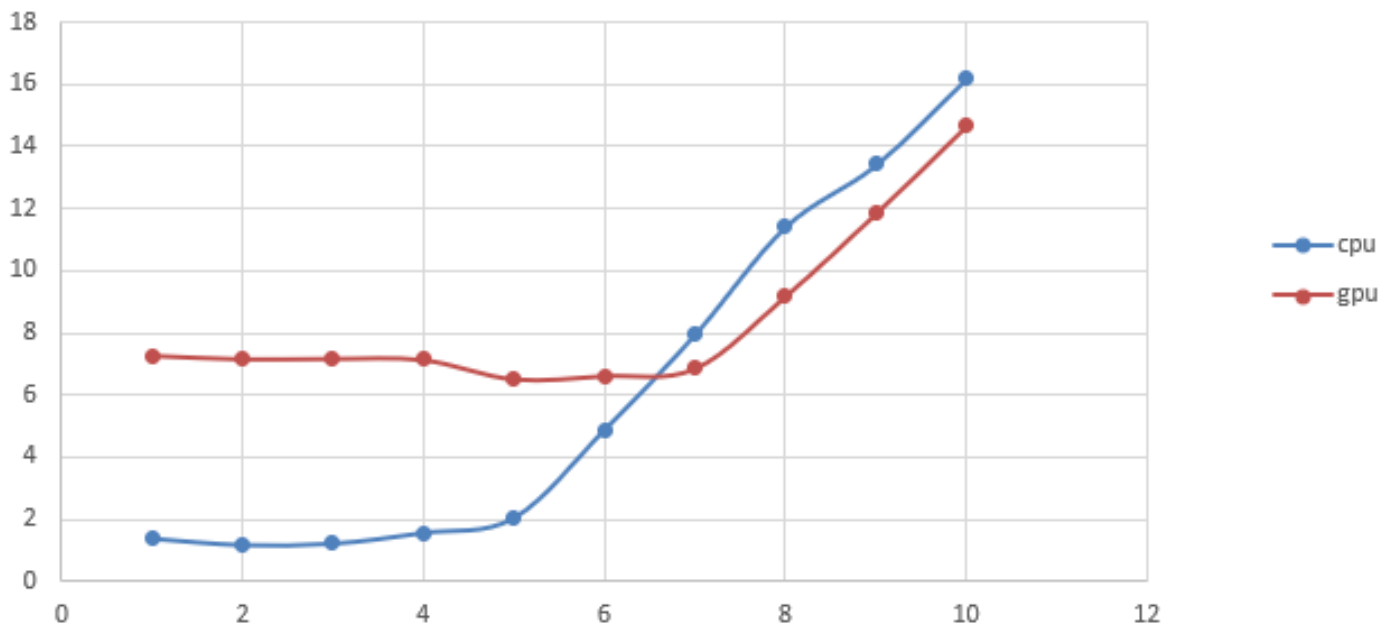
$$\text{SHA256}(\text{"Para"} + \text{str}(x))[:y] == \text{"0"} * y$$

kde  $y \in \{1, 2, \dots, 10\}$  označuje náročnosť úlohy – počet núl na začiatku hashu.

Z porovnania výsledkov vyplynulo, že pri úlohách do hodnoty  $y = 6$  (vrátane) vykazoval CPU nižší čas vykonávania. Tento jav pripisujeme dvom hlavným faktorom:

- **Overheat GPU vykonávania** – pri krátkych úlohách sa GPU výpočtovo neprejaví, lebo zdržanie spôsobené inicializáciou jadier, prenosom dát a synchronizáciou prekryje samotný výpočet.
- **Výhoda okamžitého zastavenia na CPU** – CPU vie úlohu ukončiť hneď po nájdení výsledku, zatiaľ čo GPU vykonáva výpočty paralelne, ale vo fixnej dávke (napr. work-group), takže „dojazd“ výpočtu môže byť dlhší.

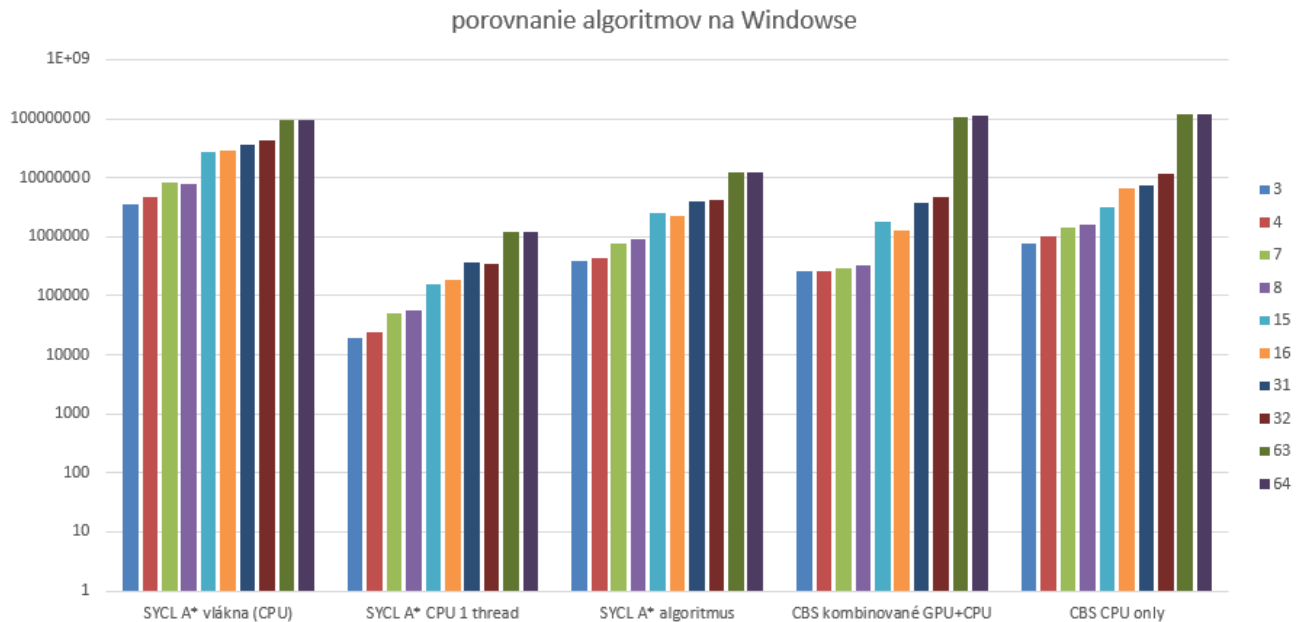
## logaritmovaná škála - porovnanie časov gpu a cpu



Zaujímavosťou je tiež trend mierneho zlepšovania časov GPU pri prvých troch úrovniach náročnosti. Domnievame sa, že tento jav môže súvisieť s **JIT charakterom SYCL**, ktorý pri opakovanom spustení daného kernelu mohol optimalizovať časti kódu v GPU runtime.

Pre vyššie hodnoty  $y$  (napr. 8 až 10) sa už GPU ukazuje ako výrazne výhodnejšie riešenie, aj keď rozdiely medzi platformami sa stále líšia v závislosti od konkrétneho hardvéru a implementácie hashovacej funkcie.

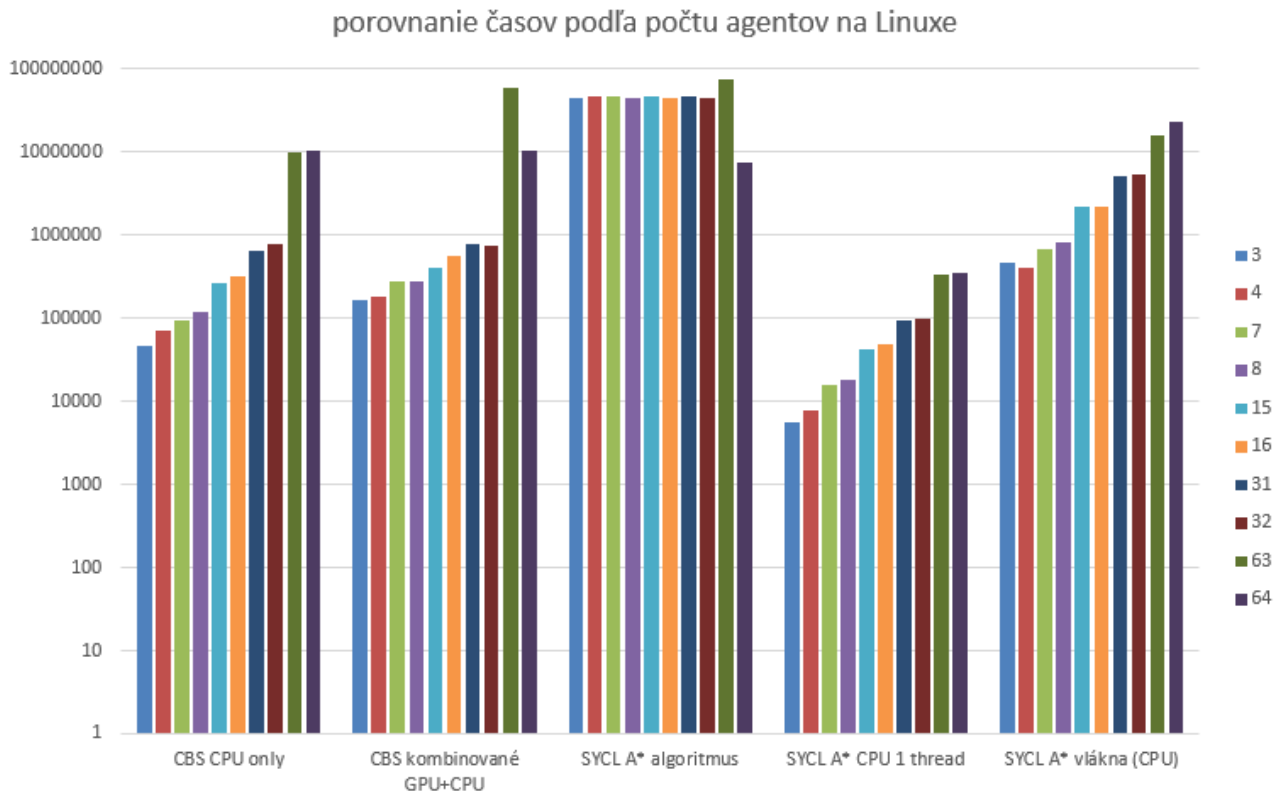
## 6.2 Výsledky



Program sme vyvíjali a prototypovali na platforme Windows a prvé cesty sme urobili na tejto platforme.

Na tomto obrázku je porovnanie jednotlivých algoritmov pre rôzny počet agentov (modrá pre troch agentov - prvý stĺpec vo všetkých kategóriách, červená pre štyroch agentov - druhý stĺpec vo všetkých kategóriách, svetlozelená pre siedmich agentov - tretí stĺpec vo všetkých kategóriách, fialová pre ôsmich agentov - štvrtý stĺpec vo všetkých kategóriách, tyrkysová pre pätnástich agentov - piaty stĺpec vo všetkých kategóriách, oranžová pre šestnástich agentov - šiesty stĺpec vo všetkých kategóriách, tmavomodrá pre tridsaťjeden agentov - siedmy stĺpec vo všetkých kategóriách, tmavočervená pre tridsaťdva agentov - ôsmy stĺpec vo všetkých kategóriách, tmavozelená pre šesťdesiattri agentov - deviaty stĺpec vo všetkých kategóriách a tmavofialová pre šesťdesiatštyri agentov - desiaty stĺpec vo všetkých kategóriách). Výsledky nás prekvapili, pretože sme nečakali, že algoritmus s jedným vláknom (druhý algoritmus zľava), využívajúci SYCL funkcie, bude najrýchlejší algoritmom z päťice, ktorú sme implementovali. Algoritmy dosahujú veľmi podobné výsledky vďaka tomu, že implementácia SYCL na Windowse podporuje paralelizáciu na procesore. Verzie so synchronizáciou (prvá a tretia kategória) sú pomalšie oproti prvému spomínanému algoritmu kvôli potrebe neustálej synchronizácie. Prvý algoritmus je pomalší oproti tretiemu algoritmu kvôli tomu, že prvý algoritmus využíva našu implementáciu bariéry, zatiaľ čo tretí algoritmus využíva natívnu bariéru, ktorá je v balíku SYCL. Na obrázku je vidieť, že hybridná verzia a verzia bežiaca na procesore majú veľmi podobné výsledky.

(Na y-ovej osi grafu sú jednotky uvedené v nanosekundách.)



Naším cieľom bolo skompilovať, spustiť a otestovať program implementovaný pomocou knižnice SYCL na GPU platforme. Očakávali sme, že vďaka paralelnej architektúre GPU dôjde k výraznému zrýchleniu výpočtov. Výsledky meraní však tieto očakávania nenaplnili.

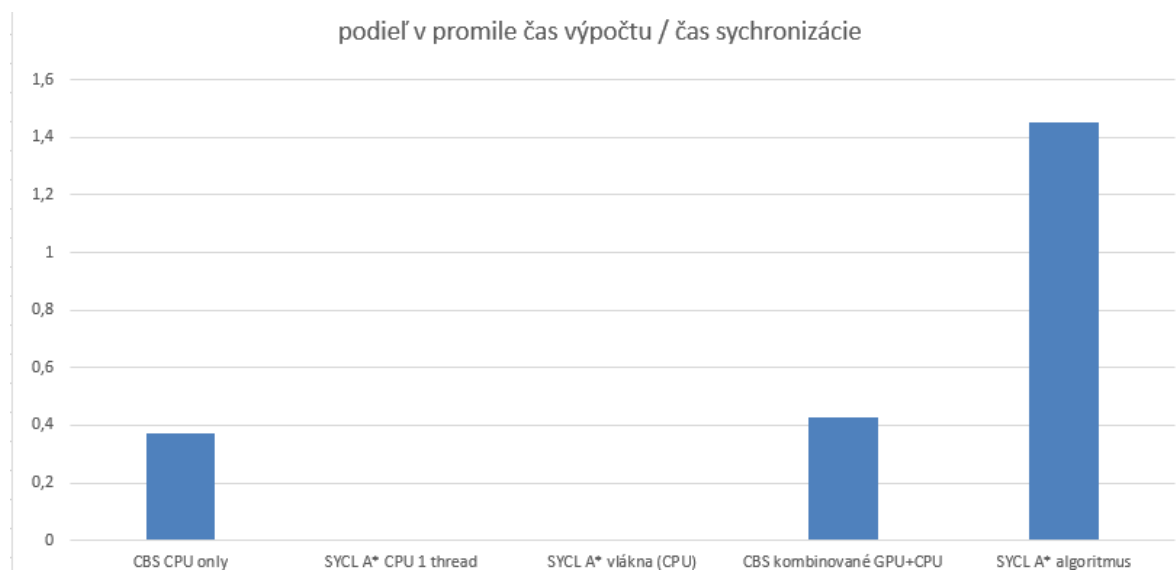
Z grafu je zrejmé, že najrýchlejší bol opäť algoritmus využívajúci jediné vlákno na CPU (štvrtý súbor stĺpcov). Na rozdiel od predchádzajúcich experimentov sa SYCL implementácia algoritmu A\* (tretí súbor stĺpcov) vyznačovala takmer konštantným časom výpočtu pri rôznych počtoch agentov, pričom vo väčšine prípadov bola výrazne pomalšia než ostatné prístupy. Tento rozdiel možno pripísať architektúre testovaného CPU, ktorý disponuje 32 vláknami (vďaka 16 fyzickým jadram a hyperthreadingu), čo umožňuje vysoký stupeň paralelizmu. Naopak, GPU beží na nižšej frekvencii a pri nevhodnom paralelnom rozložení úloh nemusí dosiahnuť optimálny výkon.

Zaujímavým zistením bolo, že v prípade 64 agentov (tmavofialové stĺpce) došlo k zvýšeniu výkonu GPU – v tomto konkrétnom prípade bol čas výpočtu nižší než pri ostatných počtoch agentov. Tento jav možno pripísať efektívnejšiemu využitiu výpočtových jednotiek GPU pri vyššej záťaži, keď sú dostupné výpočtové zdroje plne vyťažené. Okrem toho môže ísť o kombináciu architektonických výhod GPU pri veľkých paralelných úlohách.

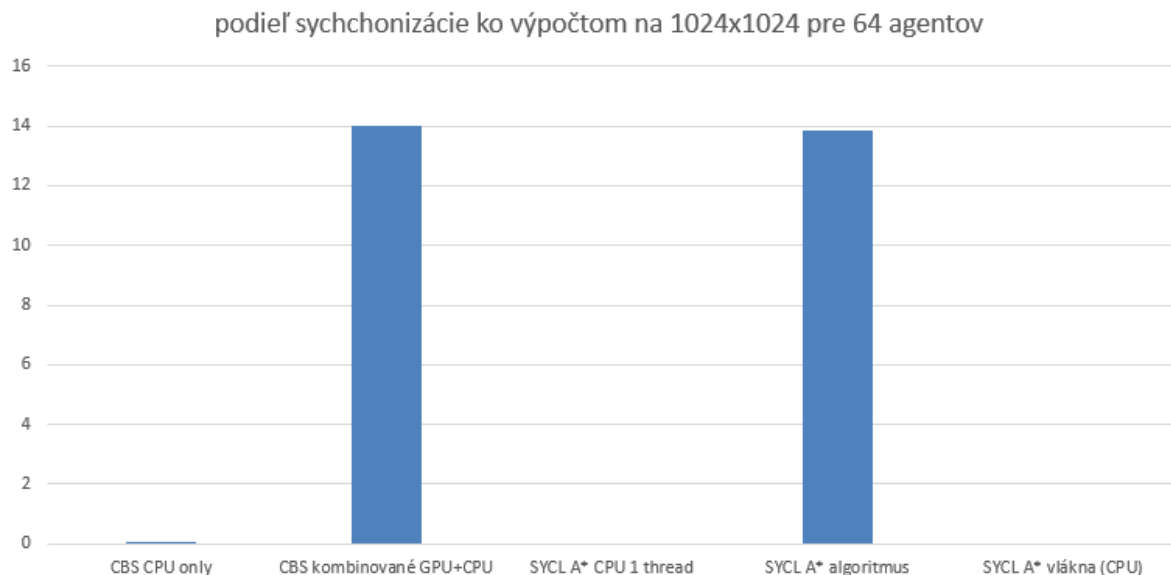
Dôležitým brzdiacim faktorom je však potreba častej synchronizácie medzi výpočtovými vláknami – a to tak na úrovni CPU, ako aj GPU. Problémom je aj overhead spôsobený prenosom dát medzi RAM a globálnou pamäťou GPU (síce len v promile, nižšie - v ďalších grafoch). Táto latencia sa negatívne prejavuje najmä v prípade hybridných riešení (druhý súbor stĺpcov v grafe), kde je nutné často prepínať medzi CPU a GPU výpočtami.

Jednotlivé farby stĺpcov prislúchajú k jednotlivým počtom agentov, tak ako na minulom grafe.

(Na y-ovej osi grafu sú jednotky uvedené v nanosekundách.)



Naše algoritmy majú aj druhú zložku. Okrem samotného výpočtu sme merali aj čas, ktorý zaberie konverzia výsledkov a algoritmu do štruktúr, ktoré majú uchovávať súčasný stav. Na obrázku je vidieť, koľko promile zaberá jednotlivým algoritmom konverzia výsledkov do spoločných dátových štruktúr. Tento obrázok ukazuje synchronizačné časy na malej mape (16 krát 16).



Na tomto obrázku opäť vidíme, koľko promile zaberá synchronizácia pamäti s výsledkami, tentokrát však na veľkej mape (1024 krát 1024).

### 6.3 Zhrnutie výsledkov

Napriek nášmu úsiliu implementovať multiagentový systém na GPU za účelom dosiahnutia akcelerovaného výpočtu sa nám na našom špecifickom hardvéri nepodarilo dosiahnuť lepšie výsledky ako na CPU.

Výsledky nepodporujú hypotézu, že pri dostatočne veľkých dátach sa implementácia na GPU oplatí aj na bežnom spotrebiteľskom hardvéri.



# Záver

V tejto diplomovej práci sme navrhli, implementovali a analyzovali program pre plánovanie trasy viacerých agentov v prostredí domáceho počítača. Program bol testovaný na domácom počítači a využíva A\*-algoritmus doplnený o detekciu a riešenie kolízií, vrátane spomalenia agentov pri zdržaniach spôsobených konfliktmi. Cieľom bolo dosiahnuť efektívny pohyb agentov v zdieľanom priestore s minimálnymi konfliktmi.

Počas návrhu sme preskúmali a implementovali päť rôznych prístupov, pričom sme sa snažili využiť potenciál paralelizácie výpočtov pomocou GPU s použitím knižnice SYCL. Napriek našim očakávaniam sa ukázalo, že GPU verzia v testoch nedosahovala vyšší výkon ako viacvláknové CPU riešenie. Aj napriek tomu sa podarilo vytvoriť plne funkčný, stabilný a multiplatformový systém, ktorý úspešne bežal na CPU aj GPU. Vďaka využitiu SYCL má systém potenciál byť nasadený aj na iných zariadeniach, ktoré túto platformu podporujú.

Implementácia bola testovaná na rôznych scenároch, kde preukázala svoju funkčnosť. Hoci niektoré časti riešenia boli programovo „zadrôtované“, napríklad pevne nastavený limit 4 GB pamäte pre dátové štruktúry, systém je možné upraviť tak, aby využíval väčšie množstvo VRAM.

Napriek našim očakávaniam, že GPU podstatne zrýchli beh algoritmu, sa ukázal opak. Ako najrýchlejší algoritmus sa ukázal jednovláknový aproximačný algoritmus. Algoritmus na GPU bol dokonca pomalší ako všetky algoritmy, ktoré bežali na CPU.

V prípade budúceho rozšírenia by sa dalo uvažovať o alternatívnych algoritmoch ako D\* Lite alebo iné stratégie riešenia kolízií, ktoré by mohli zlepšiť výkonnosť a prispôsobivosť systému v zložitejších prostrediach.



# Literatúra

- [1] AdaptiveCpp Contributors. Adaptivecpp documentation. <https://github.com/AdaptiveCpp/AdaptiveCpp/tree/develop/doc>, 2024. Accessed: 2025-04-29.
- [2] Mustafa Alhassow, Oguz Ata, and Dogu Atilla. Obstacle avoidance capability for multi-target path planning in different styles of search. *Computers, Materials & Continua*, 81:749–771, 10 2024.
- [3] Ankur Bhargava, Mohd Suhaib, and Ajay Singholi. A review of recent advances, techniques, and control algorithms for automated guided vehicle systems. *Journal of the Brazilian Society of Mechanical Sciences and Engineering*, 46, 06 2024.
- [4] J. J. M. Evers and S. A. J. Koppers. Automated guided vehicle traffic control at a container terminal. *Transportation Research Part A: Policy and Practice*, 30(1):21–34, 1996.
- [5] Kelin Jose and Dilip Kumar Pratihari. Task allocation and collision-free path planning of centralized multi-robots system for industrial plant inspection using heuristic methods. *Robotics and Autonomous Systems*, 80:34–42, 2016.
- [6] Shuanglong Kan, Zhe Chen, David Sanan, Shang-Wei Lin, and Yang Liu. An executable operational semantics for rust with the formalization of ownership and borrowing. *arXiv preprint arXiv:1804.07608*, 2018.
- [7] Khronos Group. *SYCL 2020 Specification*, 2020. <https://registry.khronos.org/SYCL/specs/sycl-2020/pdf/sycl-2020.pdf>.
- [8] Matej Magát. Github repozitár kódov, 2024-2025. <https://github.com/MatejMagat305/diplomovka2025>.
- [9] NVIDIA. CUDA Toolkit Documentation, 2023. <https://docs.nvidia.com/cuda/>.
- [10] Qualcomm Technologies, Inc. *Qualcomm® Snapdragon™ Mobile Platform OpenCL General Programming and Optimization*. Qualcomm Technologies, Inc., San Diego, rev. c edition, 2023.

- [11] James Reinders, Ben Ashbaugh, James Brodman, Michael Kinsner, Simon Pennycook, and Xinmin Sodani. *Data Parallel C++: Mastering DPC++ for Programming of Heterogeneous Systems using C++ and SYCL*. Apress, 2021.
- [12] Haining Xiao, Xing Wu, Dejin Qin, and Jingjing Zhai. A collision and deadlock prevention method with traffic sequence optimization strategy for ugn-based agvs. *IEEE Access*, 8:209452–209470, 2020.

# Príloha A: Kód

Listing 1: implementácia SimulationScene

```
void SimulationScene::runSimulation() {  
    if (mem->isRunning) {  
        return;  
    }  
    {  
        std::lock_guard<std::mutex> lock(mem->simMutex);  
        if (mem->isRunning) {  
            return;  
        }  
        mem->isRunning = true;  
    }  
    mem->sweepThread();  
    mem->simThread = new std::thread([this]() {  
        Map* m = mem->map;  
        mem->info = letCompute(mem->method[mem->indexType], m);  
        mem->hasInfo = true;  
    });  
}
```

Tu je kód, ktorý zabezpečuje spustenie výpočtového vlákna.

Listing 2: princíp zapisovania chýb

```
sycl::atomic_ref<int ,  
sycl::memory_order::relaxed ,  
sycl::memory_scope::device ,  
    sycl::access::address_space::global_space>  
err_flag(globalMemory.minSize_maxtimeStep_error[ERROR]);  
....  
err_flag.store(agentID + 1);
```

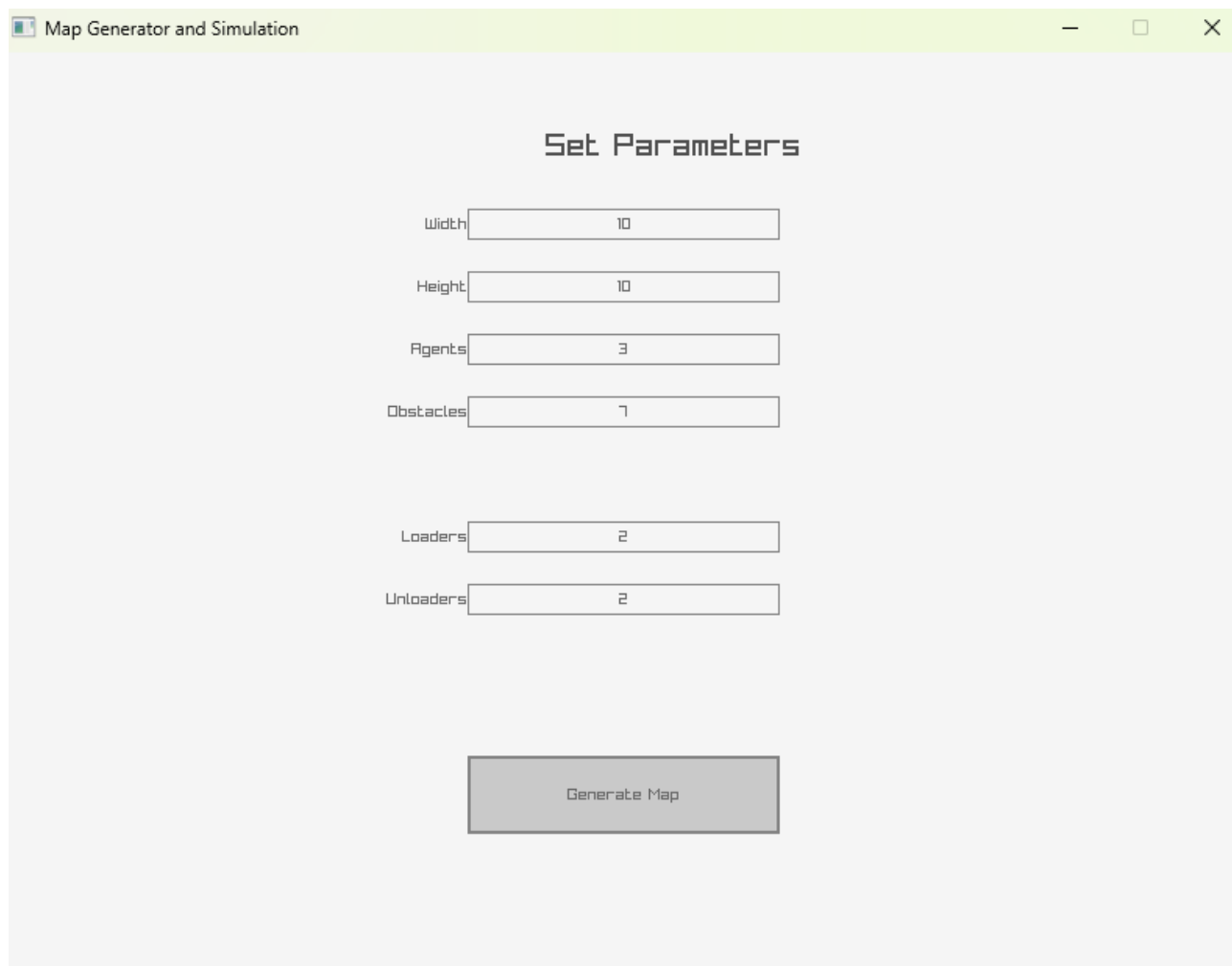
Toto je kód zvnútra kernela. Je predpoklad, že takéto chyby budú vzácne, no dali sme to pre istotu ako atomickú operáciu.

Listing 3: princíp kontroly chýb

```
if (m->CPUMemory.minSize_maxtimeStep_error[ERROR] != 0) {  
    result.error = "Memory_error_in_GPU_computation_in_agent:" +  
    return result;  
}
```

Tu vidíme kontrolu chyby zo strana „bežného“ kódu.

## Príloha B: Obrázky z GUI



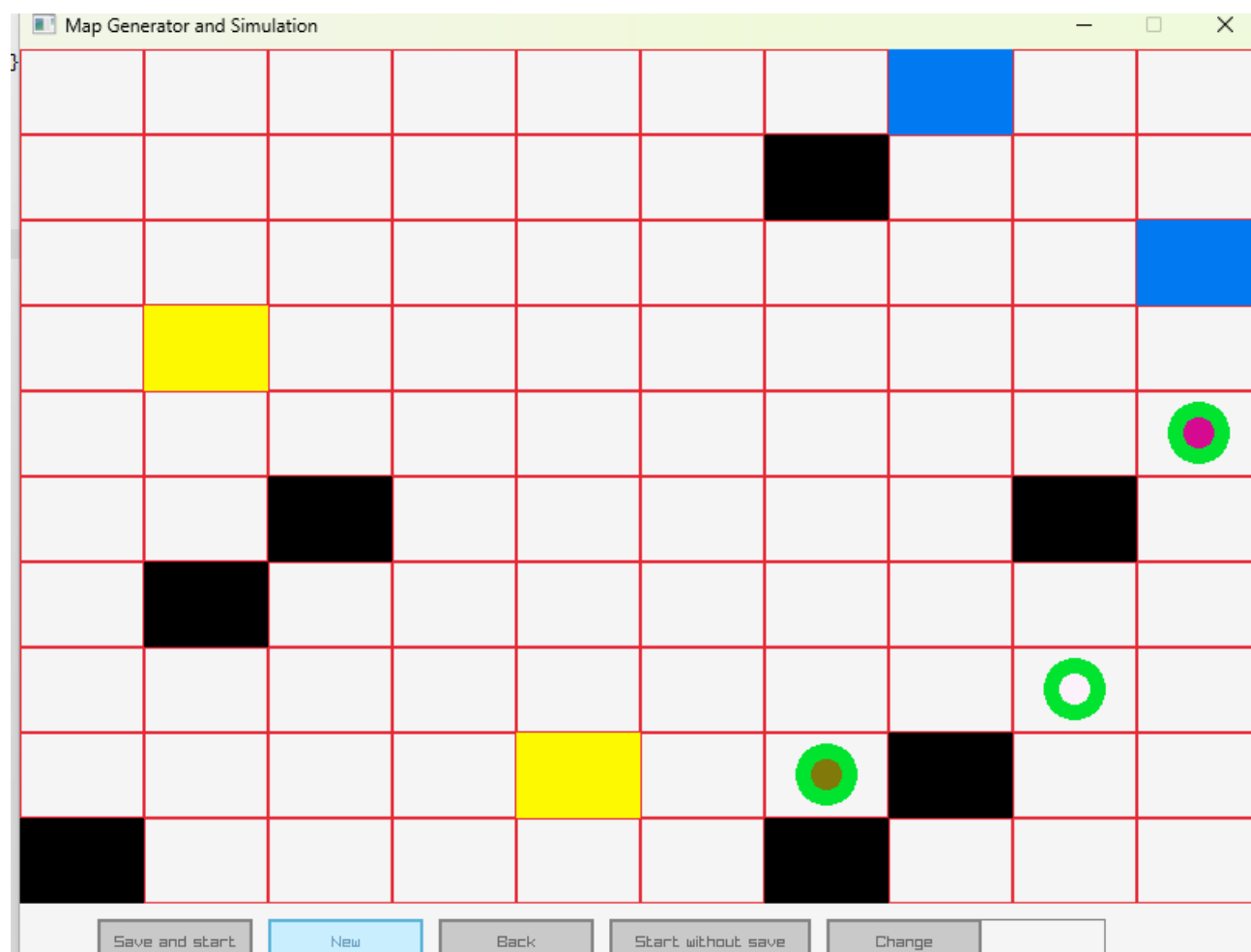
The screenshot shows a software window titled "Map Generator and Simulation". Inside the window, there is a section titled "Set Parameters". This section contains six input fields, each with a label to its left and a numerical value inside the field:

- Width: 10
- Height: 10
- Agents: 3
- Obstacles: 7
- Loaders: 2
- Unloaders: 2

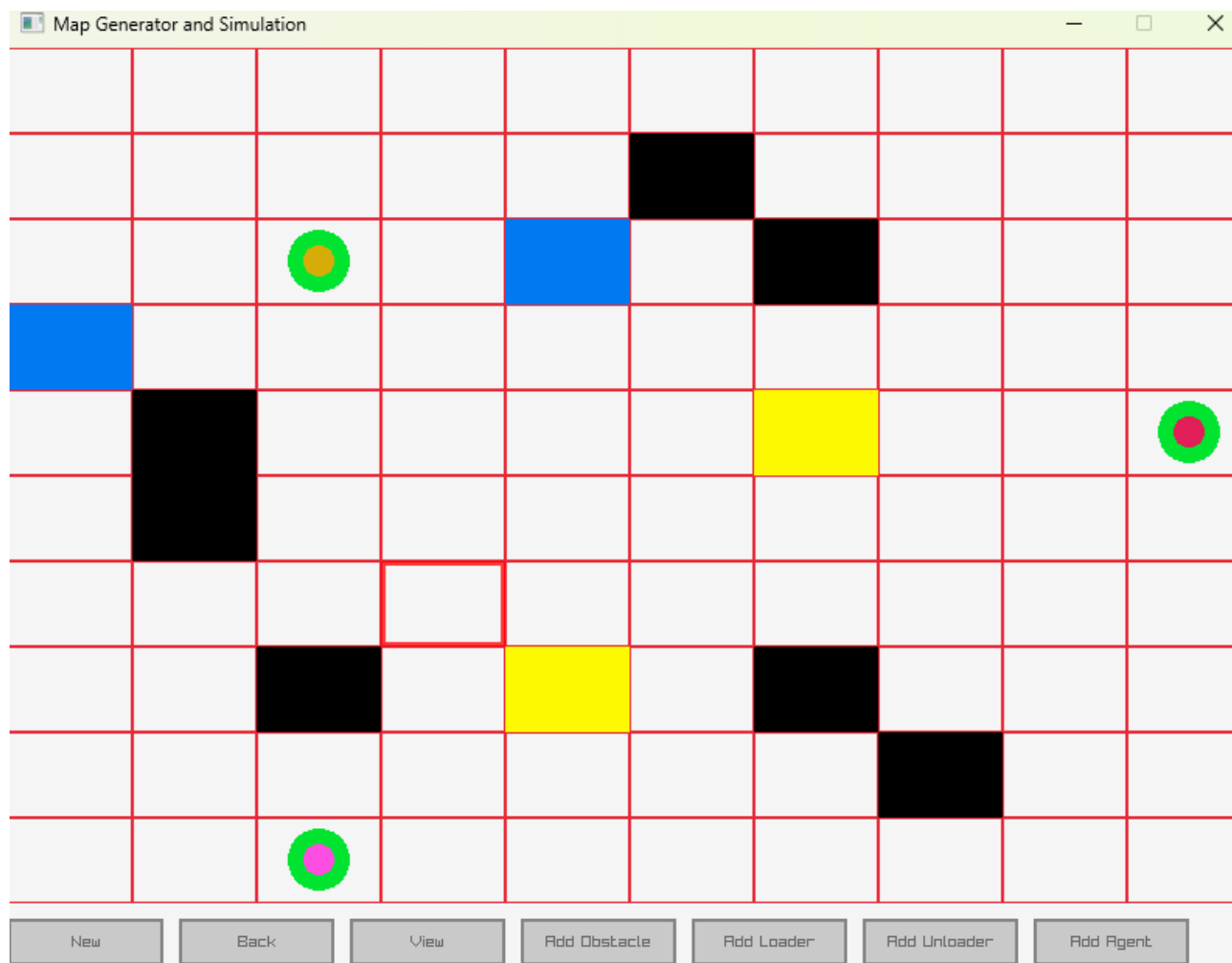
Below these input fields is a large, grey rectangular button labeled "Generate Map".

Obr. 1: Obrázok parameter scény

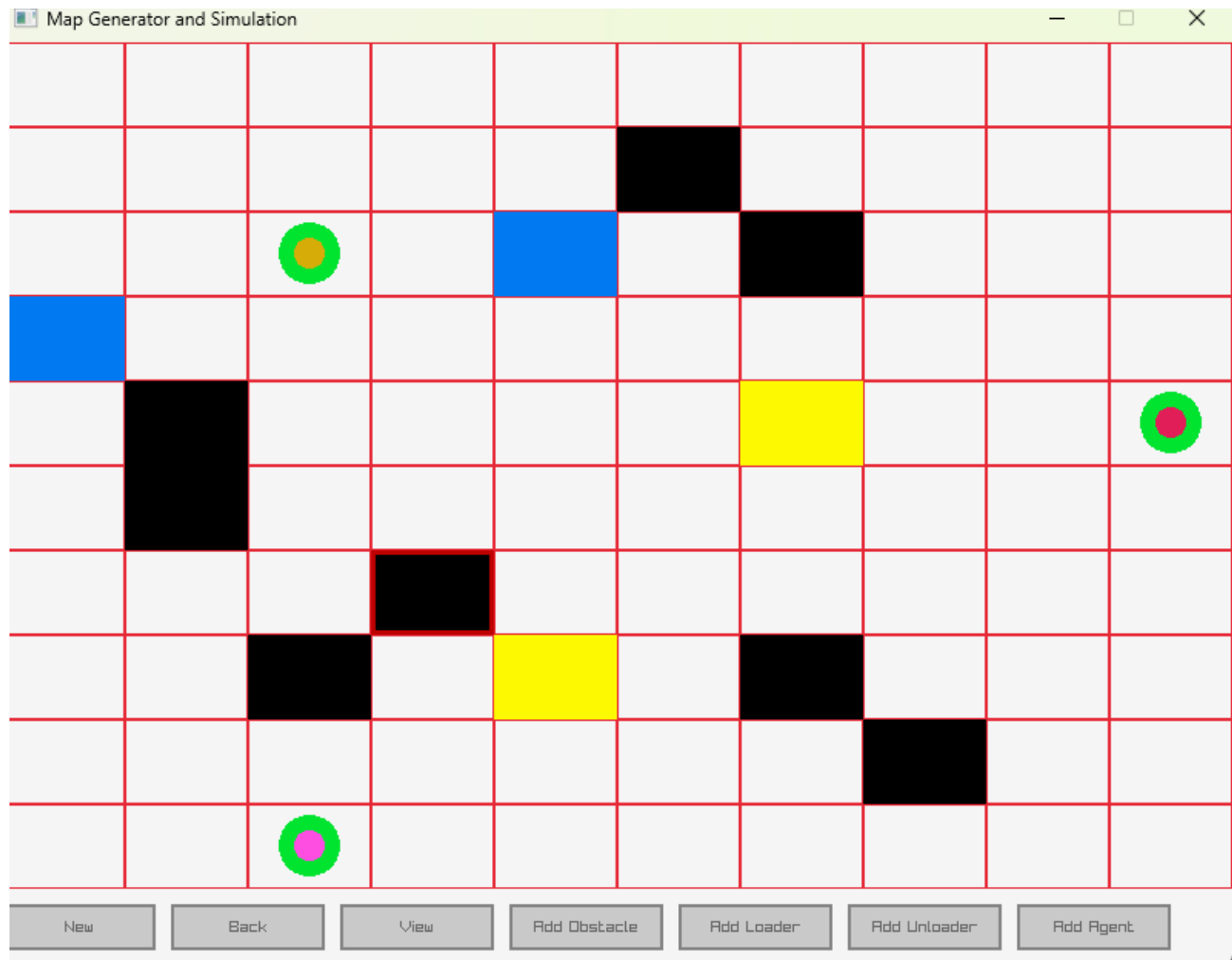




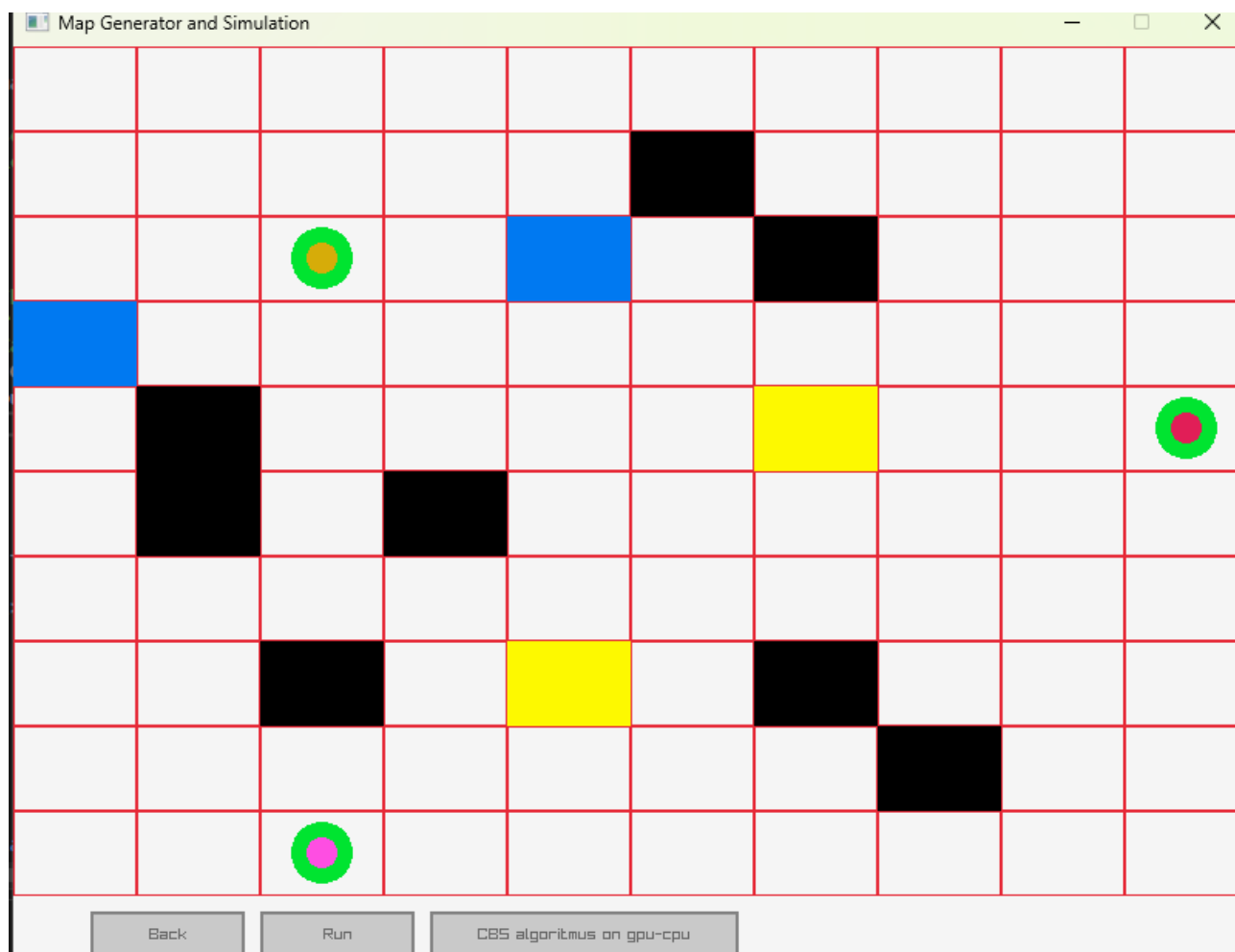
Obr. 2: 1. obrázok view scény



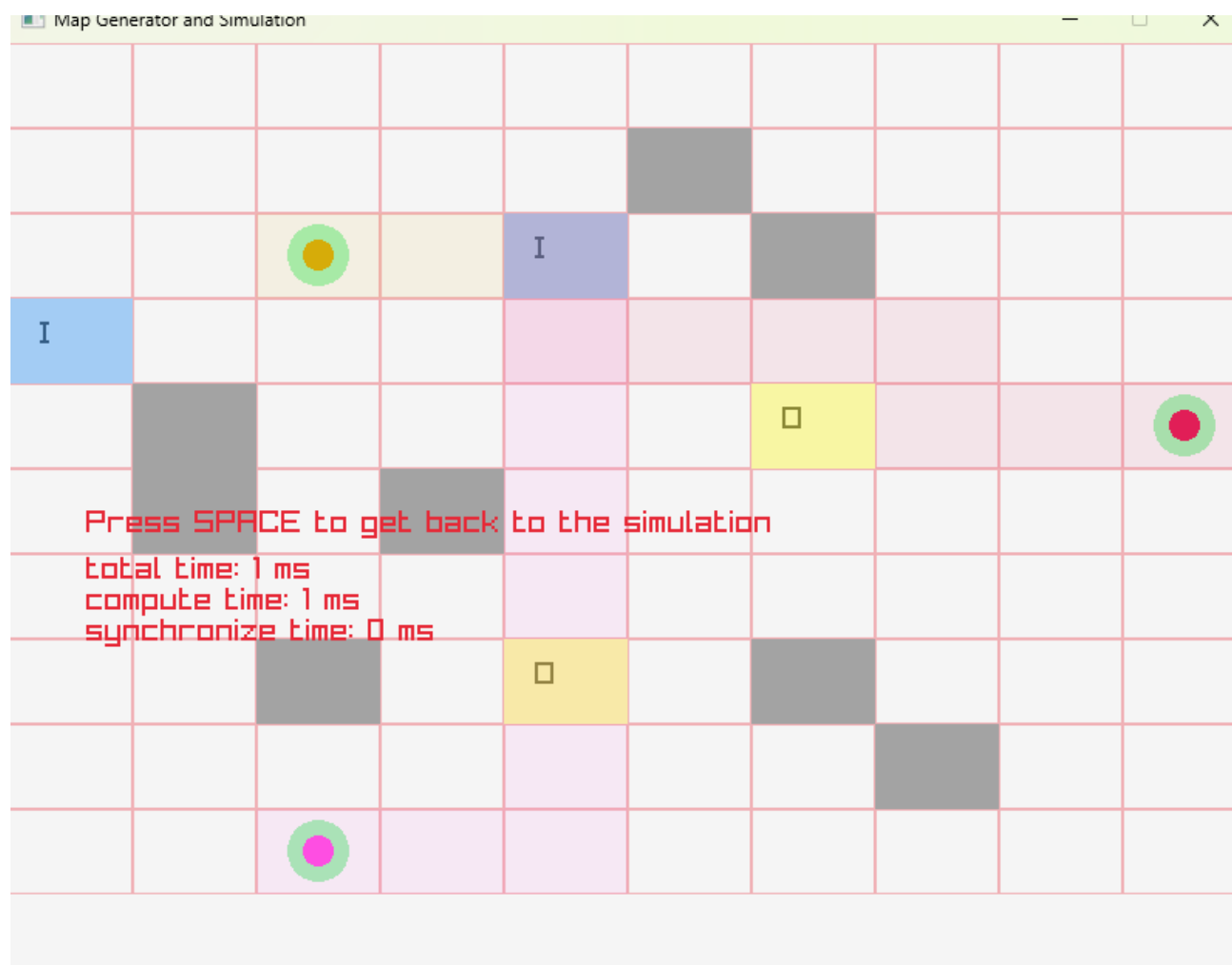
Obr. 3: 1. obrázok change scény s označeným políčka



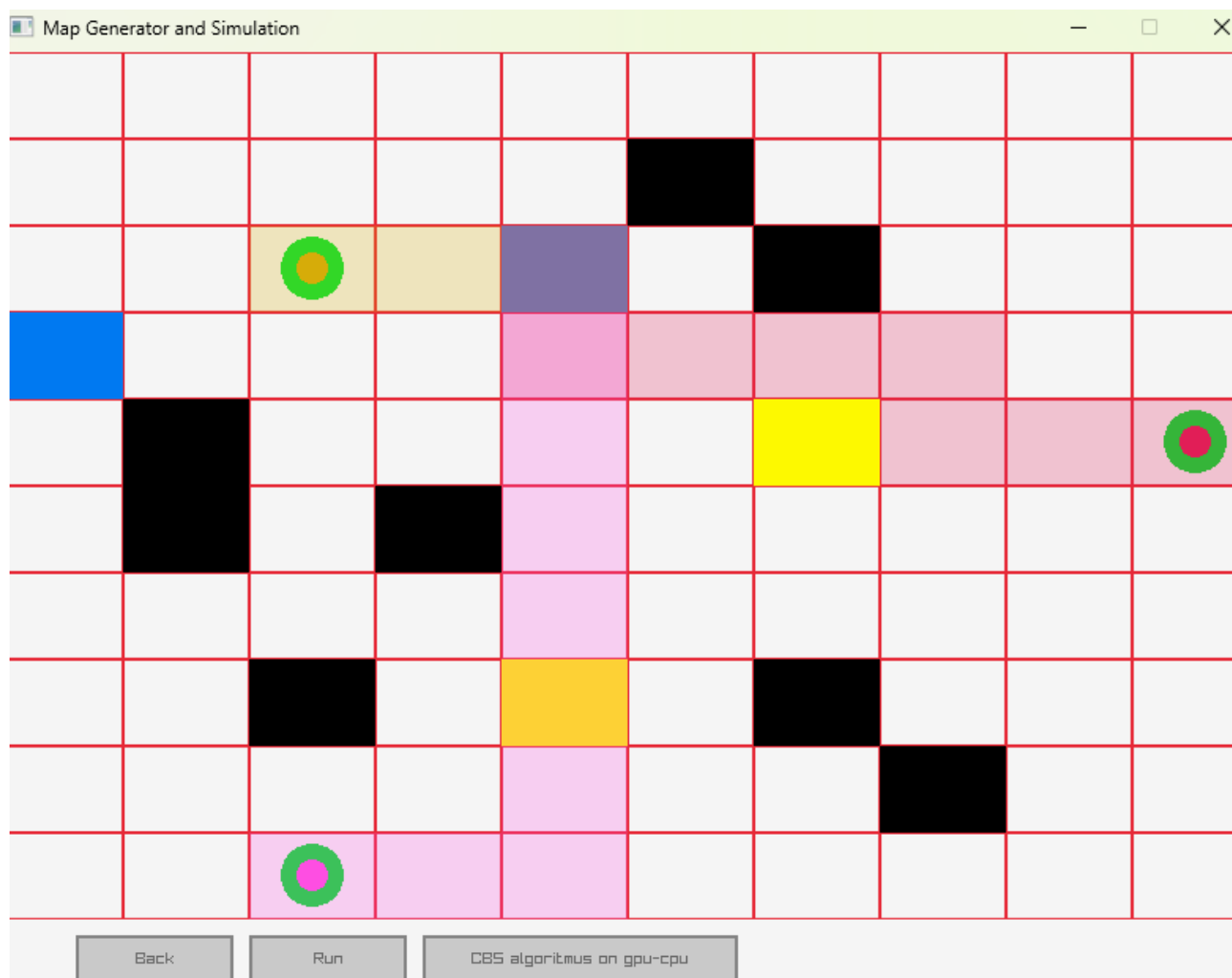
Obr. 4: 2. obrázok change scény s pridaním prekážky



Obr. 5: Obrázok simulation scény - pred simuláciou



Obr. 6: Informácia po výpočte



Obr. 7: Zobrazenie po výpočte