VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
FACULTY OF INFORMATION TECHNOLOGY

ÚSTAV INTELIGENTNÍCH SYSTÉMŮ
DEPARTMENT OF INTELLIGENT SYSTEMS

SIMULACE PROUDĚNÍ TEKUTIN S VYUŽITÍM CELULÁRNÍCH AUTOMATŮ

FLUID DYNAMICS SIMULATION USING CELLULAR AUTOMATA

BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE — **MICHAL REŽŇÁK**
AUTHOR

VEDOUCÍ PRÁCE — **PETR PERINGER, Dr. Ing.**
SUPERVISOR

BRNO 2019

Vysoké učení technické v Brně
Fakulta informačních technologií

Ústav inteligentních systémů (UITS) Akademický rok 2018/2019

Zadání bakalářské práce

Student: **Režňák Michal**
Program: Informační technologie
Název: **Simulace proudění tekutin s využitím celulárních automatů**
Fluid Dynamics Simulation Using Cellular Automata
Kategorie: Modelování a simulace

Zadání:

1. Seznamte se s problematikou modelování proudění tekutin a simulace s využitím celulárních automatů (CA). Prostudujte technologie pro implementaci simulátoru CA v C++ a WebAssembly.
2. Navrhněte simulátor proudění tekutin a sadu minimálně 5 demonstračních modelů pro ověření funkčnosti a výukové účely. Zaměřte se na interaktivní vizualizaci výsledků simulace.
3. Navržený simulátor implementujte v C++ (vhodně zvolte knihovnu pro GUI) a v prostředí WebAssembly. Obě implementace řádně otestujte.
4. Zhodnoťte dosažené výsledky s ohledem na použití ve výuce. Porovnejte efektivitu obou implementací a navrhněte možnosti dalšího vývoje.

Literatura:

- Chopard, B., Droz, M.: *Cellular Automata Modelling of Physical Systems*. Cambridge University Press, 1998.

Pro udělení zápočtu za první semestr je požadováno:

- Splnění prvních dvou bodů zadání.

Podrobné závazné pokyny pro vypracování práce viz http://www.fit.vutbr.cz/info/szz/

Vedoucí práce: **Peringer Petr, Dr. Ing.**
Vedoucí ústavu: Hanáček Petr, doc. Dr. Ing.
Datum zadání: 1. listopadu 2018
Datum odevzdání: 15. května 2019
Datum schválení: 1. listopadu 2018

Abstrakt

Tato bakalářská práce se zabývá vytvořením aplikace pro simulaci proudění tekutin pomocí Lattice gas cellular automata. Použité modely jsou HPP, FHP-I, FHP-II a FHP-III. Program je implementovaný pomocí jazyka C++, tak aby byl spustitelný ve webovém standardu WebAssembly. Součástí práce je porovnání výkonnosti formátů wasm, asm.js a nativního formátu pro desktop (x86_64), kde se zjistilo, že doba načtení aplikace ve webovém prohlížeči je výrazně menší pro wasm formát a provádění aplikace je o 24% rychlejší oproti asm.js a o 50% pomalejší než desktop. Aplikace je vhodná pro studijní účely jako prezentace využití celulárních automatů a poskytuje úvod pro Lattice Boltzmann metodu simulace tekutin.

Abstract

The main objective of this work was to create application for fluid flow simulation using Lattice Gas Cellular Automata. Used simulation models are HPP, FHP-I, FHP-II and FHP-III. The program is implemented in a C++ language in a way that it can run in WebAssembly web standard. Part of the work is comparison between wasm, ams.js formats and native desktop (x86_64). This shows that time for application loading in web browser is much smaller for wasm format than for asm.js and application performance in wasm format is about 24% higher than asm.js but 50% smaller than a native desktop. The application is suitable for educational purpose as a presentation of cellular automata simulation and also as an introduction to the Lattice Boltzmann method for fluid flow simulation.

Klíčová slova

Celulární automaty, LGCA, C++, WebAssembly, ImGui, simulace, proudění tekutin

Keywords

Cellular automata, LGCA, C++, WebAssembly, ImGui, simulation, Fluid dynamics

Citace

REŽŇÁK, Michal. *Simulace proudění tekutin s využitím celulárních automatů.* Brno, 2019. Bakalářská práce. Vysoké učení technické v Brně, Fakulta informačních technologií. Vedoucí práce Petr Peringer, Dr. Ing.

Simulace proudění tekutin s využitím celulárních automatů

Prohlášení

Prohlašuji, že jsem tuto bakalářskou práci vypracoval samostatně pod vedením pana Dr. Ing. Petra Peringera. Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

. . . .
Michal Režňák
14. května 2019

Poděkování

Hlavně bych chtěl poděkovat vedoucímu práce Dr. Ing. Petru Peringerovi, za umožnění práce na velmi zajímavém projektu a ochotu poradit s případnými problémy a nejasnostmi. Dále bych chtěl poděkovat Ing. Pavlu Režňákovi za cenné rady a návrhy pro správnou implementaci aplikace.

Obsah

1 Úvod **2**

2 Teorie celulárních automatů **3**
2.1 Historie 3
2.2 Základní principy CA 8
2.3 Lattice gas cellular automata 12

3 Použité knihovny a nástroje **18**
3.1 Webový standard WebAssembly 18
3.2 JavaScript API WebGL 20
3.3 Grafická knihovna ImGui 21
3.4 Sada nástrojů pro překlad CMake 22
3.5 Překladové a běhové prostředí Docker 23
3.6 Testovací knihovna GTest 24

4 Návrh LGCA simulátoru Caflow **26**
4.1 Graf závislostí komponent 26
4.2 Použité návrhové vzory 26
4.3 Demonstrační modely 31

5 Implementační detaily **32**
5.1 Automatická správa paměti 32
5.2 Struktura simulátoru 32
5.3 Překladové a běhové prostředí Docker 33
5.4 Výsledná aplikace 34

6 Testování aplikace **37**
6.1 Jednotkové testy 37
6.2 Porovnání výkonnosti WebAssembleru 39

7 Závěr **41**

Literatura **42**

A Obsah CD **45**

Kapitola 1

Úvod

Celulární automaty a přidružené simulační techniky jsou vhodné metody pro popis, pochopení a simulaci chování složitých systémů. S příchodem Hry života (Game of Life) se celé odvětví stalo populárním [20] i pro veřejnost, hlavně kvůli ukázce generování komplexních obrazců v simulátoru. Celulární automaty jsou schopny simulovat velké spektrum různých problémů, například existují celulární automaty pro dopravu nebo testování hardwarových zařízení. Tato práce se zabývá Lattice gas cellular automata, které jsou vhodné pro simulaci proudění tekutin.

Lattice gas cellular automata (LGCA) vznikly v roce 1986 s nyní již slavnou publikací [19] autorů Frisch, Hasslacher a Pomeau. Tito autoři ukázali, že druh biliardu, který si zachovává hmotnost a moment při kolizích, vede v makroskopickém měřítku k Navier-Stokesově rovnici, pokud mřížka simulace (například hexagonální ve dvourozměrném prostoru) poskytuje dostatečnou symetrii.

Cílem práce je vytvořit LGCA simulátor s grafickým rozhraním, který pro studijní účely vhodným způsobem zobrazuje výsledek simulace. Simulátor musí být spustitelný ve webovém prohlížeči pomocí WebAssembly. Součástí práce je porovnání výkonnosti wasm formátu s asm.js ve webovém prohlížeči a kvůli existenci wasi formátu se porovnává rychlost běhu wasm formátu s nativním formátem desktopu (`x86_64`).

První kapitola práce je věnována teoretické části, kde jsou popsány celulární automaty, jejich vývoj a praktické využití. V další kapitole jsou rozebrány jednotlivé nástroje a knihovny, které jsou použity při implementaci projektu. U každé knihovny nebo nástroje je krátký popis a důvod použití.

Druhá polovina práce je věnována praktické části. Nachází se zde návrh, implementace a testování výsledné aplikace, která slouží k simulaci proudění tekutin pomocí celulárních automatů, kde byl kladen velký důraz na správnou vizualizaci výsledků. Kapitola s návrhem obsahuje graf závislostí jednotlivých komponent, pět modelů pro demonstraci chování LGCA metody simulace a popis návrhových vzorů, které byly použity při implementaci aplikace. Závěrem kapitoly je popsáno grafické rozhraní a jakým způsobem bylo navrženo. Implementační část obsahuje informace o způsobu implementace, přeložení a možnostech nasazení aplikace do produkce, kde jedna možnost je pomocí nástroje CMake a druhá je vytvoření obrazu pro nástroj Docker. V poslední části jsou zobrazeny výsledky testů aplikace. Byly použity jednotkové testy pro ověření funkčnosti aplikace a testovací sada byla zkontrolována pomocí pokrytí kódu jednotkovými testy.

Poslední částí práce je porovnání výkonnosti aplikace, která byla přeložena do formátu wasm, asm.js a nativního desktopu, a to doba spouštění aplikace ve webovém prohlížeči a rychlost provádění zdrojového kódu.

Kapitola 2

Teorie celulárních automatů

Celulární automaty, často označované zkratkou CA, jsou idealizovaný fyzikální systém, ve kterém je prostor a čas diskrétní a jednotlivé prvky systému jsou schopny nabývat pouze konečného počtu stavů. V této kapitole je probrána historie těchto automatů pro uvedení do kontextu a důvodu proč vlastně vznikly. Část s historií obsahuje i příklady, ve kterých byly použity pro demonstraci funkčnosti a nebo pro popularizaci konceptu. Jsou zde zmíněné i hlavní osobnosti, které se jakýmkoliv způsobem podílely na výzkumu. Druhá část teorie obsahuje upřesnění pravidel, podle kterých obecně celulární automaty fungují a definuje existující varianty automatů. Poslední částí jsou *lattice gas cellular automata*, které jsou hlavním tématem této práce.

2.1 Historie

I když v průběhu historie byly mnohokrát znova vynalezeny pod jiným jménem, tak samotný koncept celulárních automatů se datuje do čtyřicátých let 19. století. V průběhu desetiletích se rozšířily do mnoha různých odvětví využití.

Počátky

Na počátku byl *John von Neumann* [31], který se podílel na designu prvních digitálních počítačů. Před tím, než se začal zajímat o sekvenční počítače, ze kterých vychází ty dnešní, experimentoval s myšlenkou postavit počítače na bázi celulárních automatů s velmi velkým množstvím souběžných výpočtů. Pokoušel se vymyslet systém, který by byl nejenom schopen simulovat chování lidského mozku, ale taky aby obsahoval samořídící mechanismy a mechanismy, které jsou schopny se samy opravovat. Jeho myšlenka byla zbavit se rozdílu mezi daty a procesy. Daný problém řešil v plně diskrétním prostoru tvořeným z buněk, kde každá buňka byla charakterizována svým vnitřním stavem, který se skládal většinou z konečného počtu informačních prvků. Navrhoval, aby se daný systém vyvíjel v diskrétních časových krocích jako jednoduchý automat. Potom, aby se stejně jako v přírodě vyvíjely všechny buňky současně na základě buněk v okolí.

První replikující se celulární automat navržený von Neumannem byl složený ze dvourozměrné čtvercové mřížky a z několika tisíců buněk. Každá tato buňka mohla mít jeden z 29 stavů. Evoluce buňky požadovala stav jí samotné a čtyř nejbližších sousedních buněk. Von Neumann uspěl v hledání diskrétní struktury, která byla schopna replikovat identické struktury jako ona sama. Tento poznatek byl velmi zajímavý, protože se předpokládalo, že stroj je schopen pouze vytvářet objekty menší složitosti než je on sám.

Na práci von Neumanna navázali další, například *E. F. Codd* [14] v roce 1968 a mnohem později *C. G. Langton* [33]. Navrhli mnohem jednodušší celulární automat, který byl taky schopen replikace, ale využíval pouze 8 stavů.

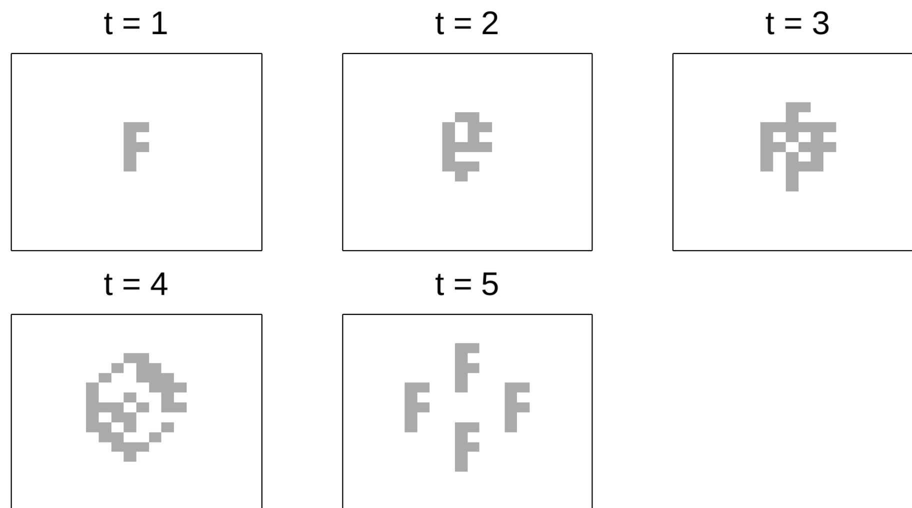
V poslední době je umělá inteligence velmi studovanou doménou. Její smysl je více porozumět skutečnému životu a chování živých tvorů skrz počítačové modely. Celulární automaty byly první pokus v tomto odvětví a věří se, že ještě mnoho jejich částí nebylo prozkoumáno a jsou schopny nabídnout vývoj v tomto odvětví.

Fredkinova hra

Edward Fredkin navrhl pomocí celulárních automatů jednoduchou hru s následnou konfigurací: Každá buňka může nabývat dvou stavů. Buď je *živá* (obsazená), nebo je *mrtvá* (prázdná). Stav všech buněk je aktualizovaný současně. Jako typ okolí je bráno *von Neumannovo*, to znamená, že se berou pouze nejbližší čtyři buňky. Pravidla jsou následující.

- Každá buňka se sudým počtem živých sousedů (0, 2, 4) bude v dalším kole výpočtu mrtvá.
- Každá buňka s lichým počtem živých sousedů (1, 3) bude v dalším kole výpočtu živá.

Model má takovou vlastnost, že po 2^n krocích simulace, kde n závisí na počátečním vzoru, se vytvoří čtyři kopie původního vzoru.



Obrázek 2.1: Jednotlivé stádia Fredkinovy hry od počátku až po 2^n krocích, aby vytvořily čtyři kopie původního tvaru. V tomto příkladu stačily pouze 4 kroky, aby se obrazec zreplikoval.

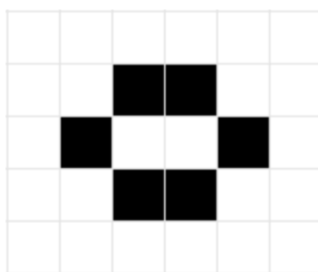
Hra života

V 70. letech minulého století navrhl *John Conway* [15] nyní již proslulou *hru života*. Jeho motivací bylo navrhnout jednoduchá pravidla, která vedla ke komplexnímu chování. Návrhem byla klasická čtvercová mřížka, ve které mohla být každá buňka buď *živá* a nebo *mrtvá*, takže obsahovala pouze dva stavy. Pravidla pro aktualizaci jsou následující:

- Mrtvá buňka, která má v okolí přesně tři živé buňky, přijde zpátky k životu
- Živá buňka, která má v okolí méně než dvě živé buňky, kvůli samotě zemře
- Živá buňka, která má v okolí více než tři živé buňky, kvůli přeplnění zemře

Okolí buňky bylo složeno nejenom ze čtyř nejbližších buněk, ale taky z dalších čtyř buněk ležících na diagonálních osách. Ukázalo se, že *hra života* má až nečekaně bohaté chování. Komplexní struktury vznikají z primitivního chaosu rozložení buněk. Struktury se dělí na stabilní formy, oscilátory, formy schopné prostorového posuvu a chaotické, to je, že nemají předem daný žádný tvar ani periodu. Pokud na pravidelnou strukturu zapůsobí vnější zdroje, například se v okolí struktury objeví jiná živá buňka, ztrácí struktura svoji pravidelnost a dostává se do chaotického stavu, ze kterého se může dostat zpět do stabilního stavu po uplynutí určitého počtu generací.

Stabilní formy jsou takové formy, které v každé generaci mají neustále stejný tvar. To znamená, že nedochází k žádnému generování ani zanikání buněk. Nejjednodušší stabilní formou je blok, kdy stačí pouze čtyři živé buňky po dvou ve sloupcích i řádcích. Okolí buněk musí být prázdné. Potom definovaná pravidla zaručují, že ve všech generacích bude tato struktura nezměněna.



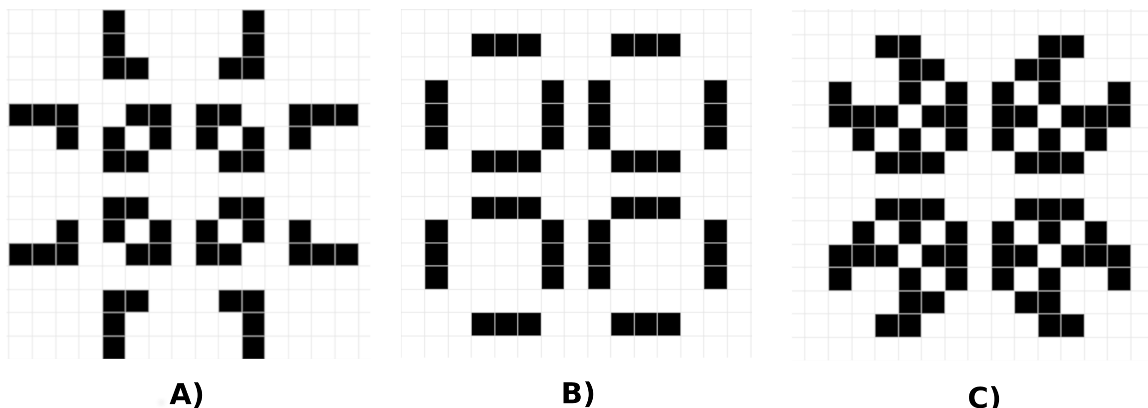
Obrázek 2.2: Stabilní forma *Behive*. Obrazec v každé generaci má stejný vzor.

Oscilátory jsou struktury, u kterých dochází k periodickému střídání stejných forem. To znamená, že začínají v určitém přesně daném stavu. Po stejném počtu kroků, vždy ve stejném pořadí, se vrátí do svojí původní podoby. Nejprimitivnější struktura je takzvaný *blinker*. Obsahuje pouze 3 živé buňky v řadě. Nezáleží jestli ve sloupci nebo v řádku. Protože jedna jeho generace budou právě 3 živé buňky v řádku za sebou a druhá generace 3 živé buňky ve sloupci. To znamená, že má pouze dvě různé generace. Komplexnější, ale stále oscilující strukturou je *pulsar*, která má právě 3 různé generace.

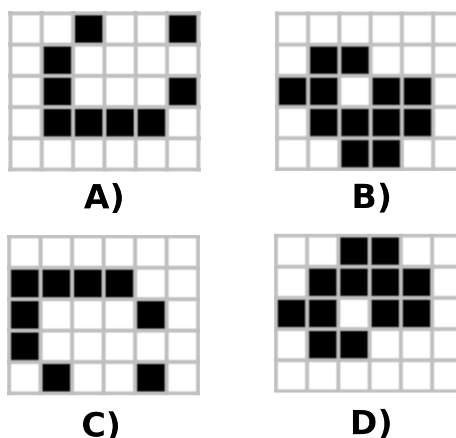
Formy schopné prostorového posuvu jsou takové formy, které stejně jako oscilátory periodicky střídají svou formu. Rozdíl je, že díky tvaru svých forem se neustále posouvají v jednom směru a tím připomínají plavbu vesmírné lodi. Proto se v anglické literatuře označují jako *spaceships*.

Zpracování obrazu pomocí CA

Jako další aspekt využití celulární automatů bylo v 50. letech 19. století zpracování obrazu. Zjistilo se, že model celulárních automatů je ideální pro aplikaci efektů pro úpravu pořízeného snímku. Byl využit hlavní prvek a to, že jednotlivé buňky reprezentovaly pixely ve snímku a v tom případě lze provést výpočet efektů nad všemi pixely současně pomocí primitivních lokálních operací. Byly vyvinuty speciální systémy aplikující logiku celulárních



Obrázek 2.3: Stádia oscilační formy označované jako *Pulsar*. První stádium pulsaru může být stav A, potom přejde do stavu B a následně do stavu C.



Obrázek 2.4: Stádia formy schopné prostorový posuv označované jako *Spaceship*. Přechody jsou seřazené od A po D.

automatů pro redukci šumu, počítání prvků nebo velikosti jednotlivých prvků ze snímků získaných z mikroskopů.

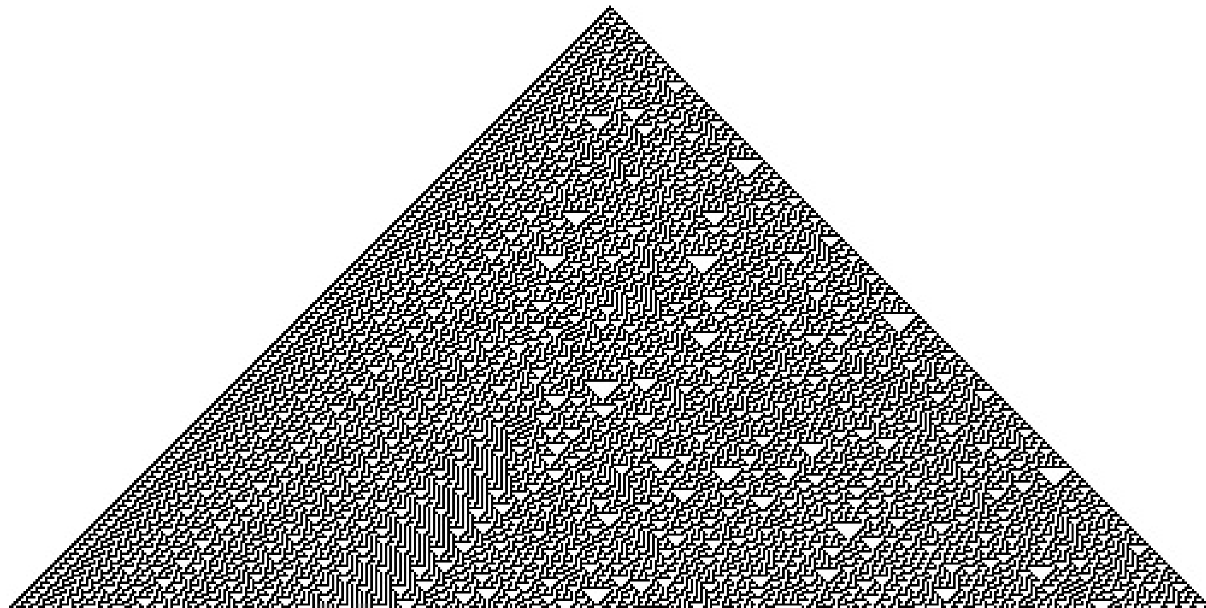
Stephen Wolfram

V 80. letech minulého století se problematice celulárních automatů intenzivně věnoval i *Stephen Wolfram* [42], v dnešní době proslulý hlavně svým programem *Wolfram Alpha* [43]. Ze začátku se podrobně věnoval skupině pravidel pro jednoduché jednorozměrné celulární automaty, nyní označované jako *Wolframova pravidla* [37].

Pravidlo 30

Jeden z příkladů Wolframových pravidel může být *pravidlo 30* [39] představené v roce 1983, které generovalo překvapivě chaotické obrazce z velmi jednoduchých, předem definovaných pravidel. Díky tomuto pravidlu Wolfram věřil, že celulární automaty jsou klíčem k pochopení, jak z jednoduchých pravidel vytvořit něco komplexního, jako je například život

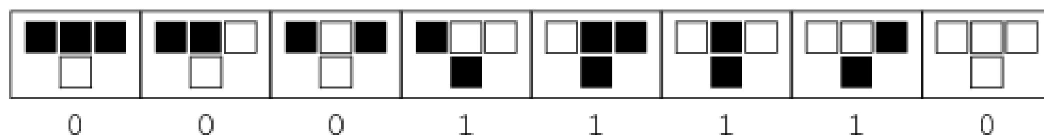
nebo příroda obecně. *Pravidlo 30* lze přímo vidět například mezi plži, kde druh *Homolice síťkovaná* [29] má ulitu pokrytou velmi podobným vzorem, jaký generuje *pravidlo 30.*



Obrázek 2.5: Vygenerovaný obrazec [39] pomocí pravidla 30.

Obrázek 2.6: Ulita Homolice síťkované [29], kde vzor připomíná pravidlo 30.

Čísla Wolframových pravidel jsou určována podle jednotlivých konfigurací daného pravidla. Například vyjádření pravidla 30 je zobrazeno na obrázku 2.7.



Obrázek 2.7: Všechny konfigurace pravidla 30 [39], kde ve vrchním řádku je aktuální buňka a dvě okolní buňky. O řádek níž je výsledný stav buňky.

2.2 Základní principy CA

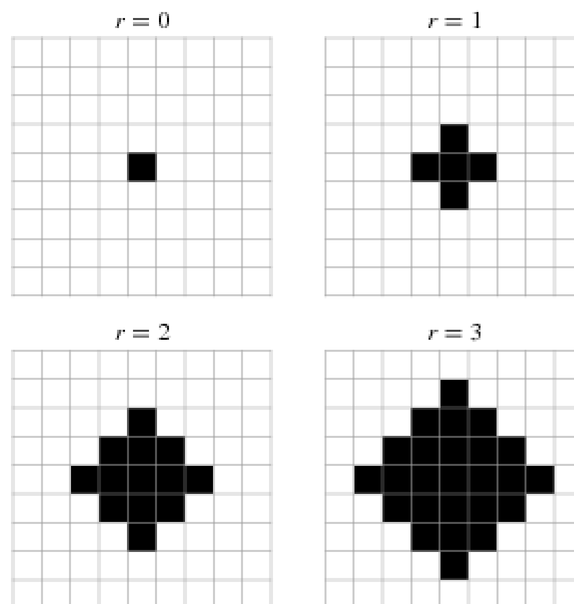
Celulární automaty jsou založené na principu, že existuje pole buněk, které je nejčastěji jedno nebo dvourozměrné, ale jsou definované i automaty ve více rozměrech. Každá buňka má svůj stav z množiny stavů, která je konečná a předem definovaná. Pro každou buňku je definované okolí, které se skládá z určitého počtu sousedních buněk. Okolí je pro každou buňku vždy stejného tvaru. To znamená, že zahrnuje vždy stejné sousední buňky vzhledem k počítané buňce. Dále jsou definována pravidla, která určují, jakým způsobem dojde ke generování další generace. Generace jsou jednotlivé stavy, ve kterých se automat nachází.

Okolí buněk

Existuje mnoho různých druhů okolí buněk a jejich variací. Nejznámější pro dvourozměrný prostor jsou von Neumannovo [40], Mooreovo [38] a Margolusovo [35] okolí.

Von Neumannovo okolí

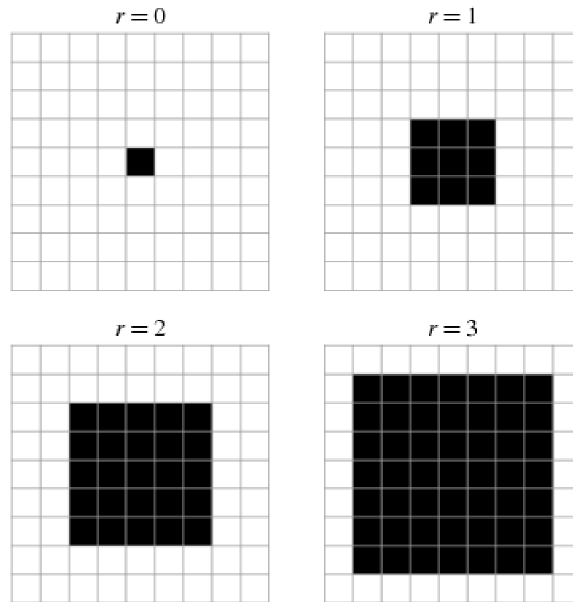
Von Neumannovo okolí je diamantového tvaru na dvourozměrné mřížce. Nejčastěji je používané ve formě, kdy se jako okolí buňky považují pouze její nejbližší sousední buňky ležící na vertikální a horizontální ose. Výsledný obrazec bude potom stejný jako na obrázku 2.8.



Obrázek 2.8: Příklad von Neumannova okolí [40] v konfiguraci, kde r určuje vzdálenost, po kterou se považují buňky za okolí.

Mooreovo okolí

Mooreovo okolí je čtvercového tvaru na dvourozměrné mřížce. Nejčastěji je používáno ve formě, kdy se za okolí buňky považují pouze její nejbližší sousední buňky ležící na vertikální, horizontální a diagonální ose. Výsledný obrazec bude potom stejný jako na obrázku 2.9.



Obrázek 2.9: Příklad *Mooreova okolí* [38] v konfiguraci, kde r určuje vzdálenost, po kterou se považují buňky za okolí.

Margolusovo okolí

Margolusovo okolí je definované na dvourozměrné mřížce. Celá mřížka se rozdělí na čtverce buněk po čtyřech. Pravidla jsou následně aplikována čistě lokálně. Pokud by se aplikovala neustále stejná mřížka v každém kroku, odstranilo by to dynamiku modelu. Proto se v každém kroku hranice mřížky mění.

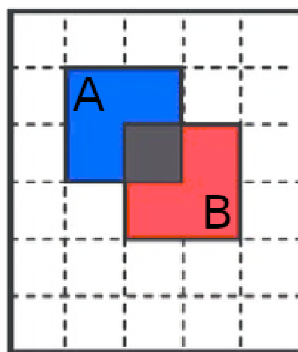
V prvním kroku jsou náhodně vybrány hrany, na kterých je definována hranice okolí podle předem zmíněných pravidel. V následujícím kroku se vytvoří hranice okolí na všech hranách, kde nebyla a odstraní na kterých hranice byla. Tento proces se opakuje celou simulaci. Střídají se proto většinou pouze dvě konfigurace mřížky. Výsledný obrazec bude potom stejný jako na obrázku 2.10.

I když zmíněné okolí má výhodu, že počet buněk uvnitř okolí je velmi malý, existuje i velká nevýhoda a to, že v každém kroku je potřeba mít uloženou informaci, jaké okolí se má aplikovat a musí se v každém kroku definovat, jaké nové okolí se použije v následujícím kroku.

Rozšířením Margolusova okolí do třírozměrného prostoru vznikne *Neckerovo okolí* [36].

Okrajové podmínky

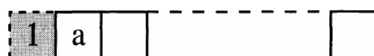
Jelikož je pole buněk konečného rozměru, je potřeba definovat pravidla, podle kterých se budou řešit krajní buňky a jejich okolí. Jedním z řešení je vytvořit speciální pravidla pro okrajové buňky. Toto řešení je ale zbytečně komplikované, místo toho lze jednodušeji vygenerovat potřebné *virtuální sousední buňky* a následně aplikovat stejná pravidla na krajní buňky jako na ostatní.



Obrázek 2.10: Příklad *Margolusova okolí* [5] v konfiguraci, kde sektor A určuje okolí v lichém kroku a sektor B v sudém kroku, uprostřed je buňka, pro kterou se okolí aplikuje.

Fixní okrajové podmínky

Nejjednodušší možností jsou *fixní okrajové podmínky*, kde se nastaví vybraný stav na všechny virtuální buňky. S touto hodnotou se potom počítá ve všech pravidlech. Například pokud zvolíme stav 1, potom všechny virtuální buňky budou obsahovat tento stav. Graficky zobrazené podmínky jsou na obrázku 2.11.



Obrázek 2.11: Fixní okrajové podmínky [11], kde a je buňka pro kterou se aplikují pravidla.

Periodické okrajové podmínky

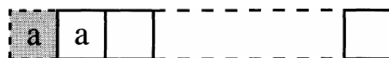
Jedním z dalších komplexnějších možností řešení je vytvořit *periodické okrajové podmínky*. Stav virtuální buňky se získá následovně: Virtuální buňka na levé straně má hodnotu nejpravější buňky na stejném řádku, virtuální buňka na vrchní straně má hodnotu nejspodnější buňky ve stejném sloupci. Opačná pravidla se aplikují pro virtuální buňky na pravé a spodní straně. Například, pokud první buňka obsahuje stav 1 a poslední buňka ve stejném řádku obsahuje stav 2, potom první virtuální buňka před krajní buňkou na začátku řádku bude obsahovat stav 2. Graficky zobrazené podmínky jsou na obrázku 2.12.

Obrázek 2.12: Periodické okrajové podmínky [11], kde a je buňka pro kterou se aplikují pravidla a b je stav virtuální buňky.

Adiabatické okrajové podmínky

Další možností jsou *adiabatické okrajové podmínky*, kde je proložené zrcadlení tak, aby se na kraji pole všechny stavy buněk opakovaly. Například, pokud krajní buňka obsahuje stav

1 a další buňka směrem do středu pole stav 2, bude první virtuální buňka obsahovat stav 1 a druhá virtuální buňka bude obsahovat stav 2. Graficky zobrazené podmínky jsou na obrázku 2.13.



Obrázek 2.13: Adiabatické okrajové podmínky [11], kde *a* je buňka, pro kterou se pravidla aplikují a zároveň stav virtuální buňky.

Reflexní okrajové podmínky

Poslední z nejčastějších řešení jsou *reflexní okrajové podmínky*, kde krajní buňka je brána jako reflexní bod od kterého dochází k opakování stavů buněk. Například, pokud krajní buňka má stav 1 a další buňka směrem do středu pole stav 2, potom bude první virtuální buňka obsahovat stav 2. Graficky zobrazené podmínky jsou na obrázku 2.14.

Obrázek 2.14: Reflexní okrajové podmínky [11], kde *a* je buňka pro kterou se pravidla aplikují a *b* je stav virtuální buňky.

Pravidla přechodů buněk

Pravidla přechodů určují, jakým způsobem a do jakého stavu může přejít buňka při generování další generace. Je to jeden z nejdůležitějších prvků celulárních automatů, který rozlišuje výslednou funkcionalitu simulátoru. Celulární automaty se můžou dělit například do dvou skupin podle uniformnosti. V tom případě můžou být CA buď *uniformní* a nebo *hybridní*.

Hybridní CA

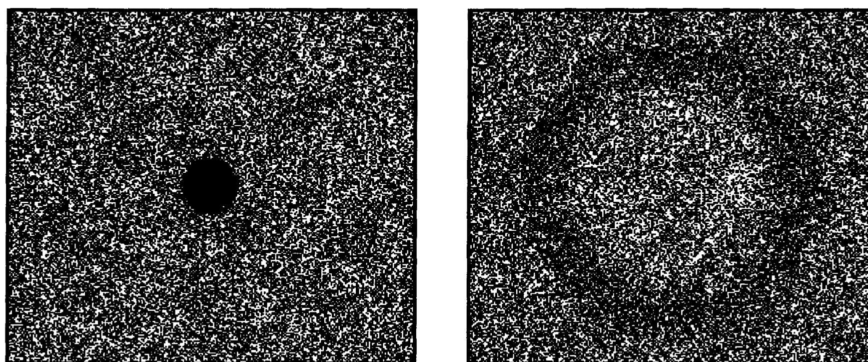
Existují automaty, u kterých nejsou na všechny buňky aplikovány stejná přechodová pravidla. Pokud dojde k výjimečné situaci lze ji ošetřit pouhým pravidlem přechodu na skupinu buněk. Jelikož je tento druh chování v rozporu s hlavní definicí celulárních automatů, nejsou považovány za čisté celulární automaty a v některých případech nejsou mezi celulární automaty řazeny vůbec. Mezi *hybridní celulární automaty* patří například *dopravní celulární automaty*, tato skupina automatů není v této práci probírána.

Uniformní CA

Existují automaty, u kterých jsou na všechny buňky aplikovány stejná přechodová pravidla. To znamená, že nelze definovat výjimečná pravidla při ojedinělé situaci pouze na skupinu buněk. Mezi *uniformní celulární automaty* patří také *lattice gas cellular automata*, které jsou hlavním tématem této práce.

2.3 Lattice gas cellular automata

Lattice gas cellular automata, zkráceně LGCA, jsou celulární automaty, které jsou schopny simulovat proudění tekutin. Nejčastěji se pro simulaci používají pouze dvourozměrné modely, ale jsou definované i třírozměrné. Aplikace pravidel pro přechod mezi stavy je prováděna ve dvou krocích. První krok se nazývá *Propagační fáze* a druhý je *Kolizní fáze*. Každá buňka je složena z určitého počtu částic. Každá částice má přiřazenou hmotnost a rychlost v určitém směru. Součinem hmotnosti a vektoru rychlosti se počítá výsledný moment buňky. Jelikož tento druh modelu obsahuje dvě fáze v jednom kroku, někteří matematici jej neuznávají jako čistý celulární automat. Tato myšlenka je však stále v minoritním zastoupení proto jsou stále řazeny pod celulární automaty.



Obrázek 2.15: Ukázka LGCA [11], který využívá HPP model. Na počátku (levý obrázek) je velká koncentrace částic ve středu obrázku. Na pravém obrázku je zobrazena simulaci v pokročilém stádiu.

Propagační fáze

V propagační fázi dochází k přesunu částic z jejich počáteční pozice v jedné buňce na konečnou pozici v buňce druhé podle pravidel definovaných v simulačním modelu. Neaplikují se kolizní pravidla. Každá částice uvnitř buňky má určený svůj směr pohybu vektorem. V propagační fázi se částice přesune přesně podle daného vektoru.

Kolizní fáze

V kolizní fázi dochází k aplikaci kolizních pravidel na jednotlivé buňky obsahující částice. Například v modelu HPP, pokud v jedné buňce jsou dvě částice, pro které existuje kolizní pravidlo, potom se aplikuje. Pokud částice uvnitř buňky nemají definované pravidlo, potom částice zůstávají na své pozici a žádná kolize se na ně neaplikuje.

Jednorozměrné LGCA

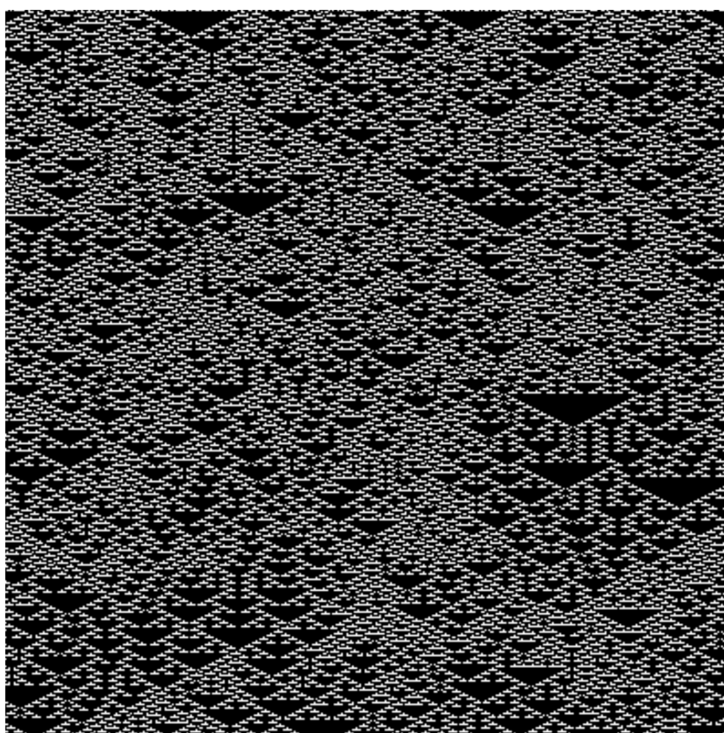
Jednorozměrné automaty mají definovaný pouze jeden rozměr, ve kterém probíhají výpočty. Z tohoto důvodu bývají pravidla a okolí buněk velmi jednoduchá. Nejčastěji se jednotlivé generace automatu zobrazují vedle sebe a výsledné dvourozměrné pole má hledanou vlastnost.

LHCA

Zajímavým modelem pro simulaci proudění tekutin pomocí jednorozměrného prostoru je *Linear Hybrid Cellular Automata* (LHCA). Automat je, jak už z názvu vypovídá, hybridního charakteru a využívá Wolframova pravidla 90 a 150. Hybridní charakter zjednodušeně znamená, že automat využívá více druhů pravidel pro simulaci. Nelze tedy popsat jednou sadou pravidel, ale musí jich být více. Úplný význam je definovaný v kapitole 2.2.

Jedním z využití zmíněného modelu automatu je generátor pseudonáhodných čísel v kryptografii [18]. Další z možností využití je pro testování digitálních obvodů [32] opět pomocí generování pseudonáhodných čísel.

Pomocí jednorozměrných modelů nelze detailně simulovat proudění tekutin, které se děje ve třech rozměrech, proto jim není věnováno hodně pozornosti. Jsou zmíněny pouze pro vytvoření kontextu pro dvou a třírozměrné modely.



Obrázek 2.16: Příklad LHCA simulátoru [41], kde každý řádek odpovídá jedné generaci. Celkově je zobrazeno 400 generací, kde čas plyne z vrchu dolů. To znamená, že první generace je na vrchu obrázku.

Dvourozměrné LGCA

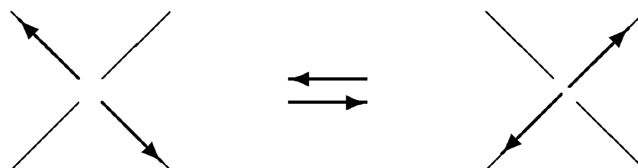
Dvourozměrné modely jsou takové modely, které pracují s dvourozměrnou mřížkou. Modely, které jsou zde uvedeny, používají buď čtvercovou nebo hexagonální mřížku. Čtvercovou používá pouze HPP model. Ostatní používají hexagonální. Jsou opět používány různě komplexní okolí. Každé má výhodu v jiném ohledu, proto nelze říct, které je nejvhodnější. Nejčastěji se v tomto tématu používá okolí, kdy jedna buňka je samostatné okolí pro všechny částice, které obsahuje. Tento systém se používá, protože dovoluje souběžný výpočet kolizní fáze.

Dvourozměrné modely jsou nejpoužívanější forma LGCA [41] modelů. Jsou schopny, stejně jako ostatní druhy modelů, definovat složité chování pomocí jednoduchých pravidel. Výhodou oproti 3D modelům je, že okolí je popsatelné využitím stejného počtu rozměrů jako ve kterých probíhá simulace. Pro 3D modely už není tohle tvrzení pravdivé.

HPP

HPP model je pojmenovaný podle autorů Hardy, de Pazzis a Pomeau. Jedná se o nejjednodušší LGCA model. Pracuje na čtvercové mřížce. Každá buňka obsahuje čtyři částice, které jsou všechny ve stejné vzdálenosti od středu. Pravidla jsou na obrázku 2.17, lze je popsat následovně:

- Pokud buňka obsahuje pouze dvě částice od sebe v 180 stupňovém úhlu, potom se obě částice přemístí o 90 stupňů v kladném směru.



Obrázek 2.17: Grafické znázornění pravidel HPP modelu [41], kde šipka určuje vektor momentu částice. Pozice obsazené částicí jsou znázorněny pomocí šipky, prázdné pozice jsou zobrazeny jako tenká čára.

Jednotlivé částice se nemohou pohybovat v diagonálách. Jelikož pravidlo neobsahuje žádnou pravděpodobnost přechodu z počátečního stavu do následujícího, je model *reverzibilní*. Pravidla lze aplikovat v opačném pořadí a potom se dostaneme do počátečního stavu simulace.

Hlavní nevýhoda modelu je, že nemá *izotropní* vlastnost. To lze vidět na příkladu 2.15. První konfigurace obsahuje izotropní počáteční stav, ale na druhé konfiguraci lze vidět, že nedošlo k šíření vlny ve všech směrech stejně. Řešením pro daný problém je použít jiný druh mřížky. Tento přístup je použitý v následujících modelech.

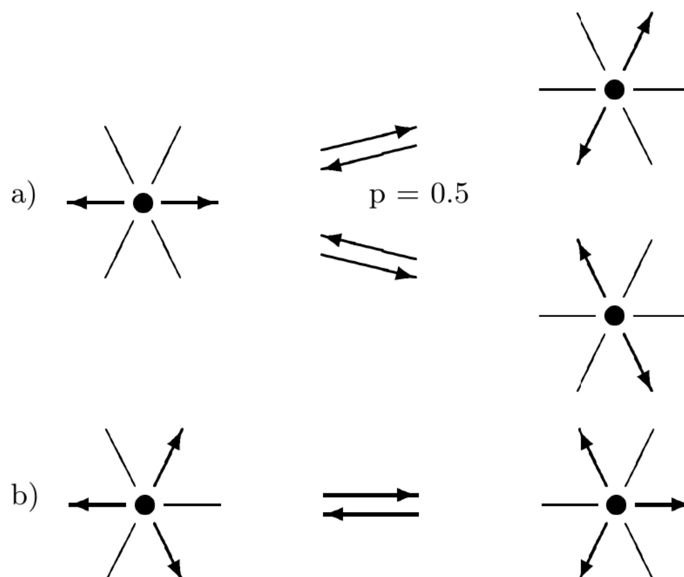
FHP-I

Model je pojmenovaný podle autorů Frisch, Hasslacher a Pomeau. FHP je model definovaný na hexagonální mřížce. Každá buňka obsahuje maximálně šest částic. Pravidla jsou na obrázku 2.18, lze je popsat následovně:

- Pokud buňka obsahuje právě dvě částice od sebe v 180 stupňovém úhlu, potom se obě částice přemístí o 60 stupňů v kladném nebo záporném směru. Směr je určený s 50% pravděpodobností.

- Pokud buňka obsahuje právě tři částice od sebe v 120 stupňovém úhlu, potom se všechny částice přemístí o 60 stupňů v kladném směru.

Jednotlivé buňky se mohou pohybovat v diagonálním směru. Jelikož pravidlo obsahuje pravděpodobnost přechodu z počátečního stavu do následujícího, není model *reverzibilní*. To



Obrázek 2.18: Grafické znázornění pravidel FHP-I modelu [41], kde šipka určuje vektor momentu částice. Pozice obsazené částicí jsou znázorněny pomocí šipky, prázdné pozice jsou zobrazeny jako tenká čára.

znamená, že pravidla nelze aplikovat v opačném pořadí abychom se dostali do počátečního stavu z koncového.

Jednotlivé FHP modely se rozlišují hlavně podle počtu pravidel, které se aplikují v kolizní fázi. Důvod vzniku nových pravidel je, že lze pomocí nich upravit *viskozitu simulované tekutiny* [41]. Platí pravidlo, že čím větší počet pravidel a komplikovanější pravidla jsou, tím má tekutina větší viskozitu. Jelikož FHP-I obsahuje nejmenší možný počet pravidel, je vhodný pro simulaci tekutin s nižší viskozitou.

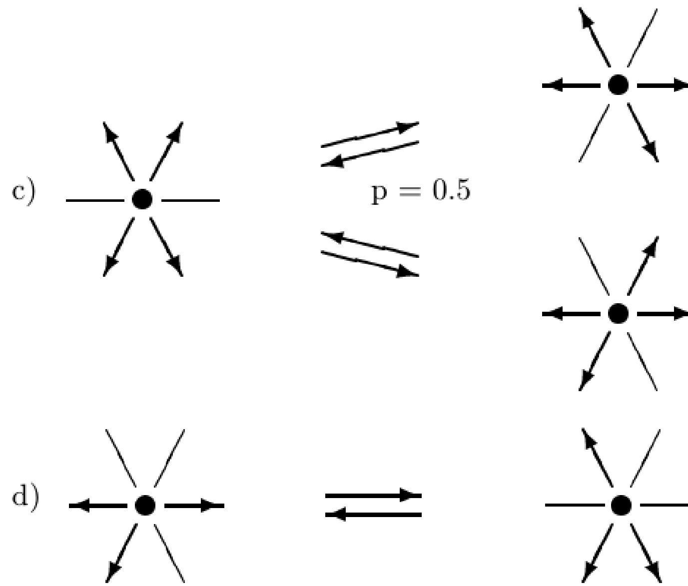
FHP-II

Model druhého řádu je pouze rozšířenou verzí modelu prvního řádu. Jsou definovány komplexnější pravidla pro přechody buněk. Pravidla o které je model rozšířený jsou na obrázku 2.19, lze je popsat následovně:

- Pokud buňka obsahuje právě čtyři částice a volné místa pro částice jsou od sebe v 180 stupňovém úhlu, potom dvě částice obsadí prázdné místa a další dvě částice se rozmístí na prázdné místa tak, aby byly od sebe v 180 stupňovém úhlu. Výběr obsazení je určený s 50% pravděpodobností.

- Pokud buňka obsahuje právě tři částice a z toho jsou dvě ve 180 stupňovém úhlu, potom se tyto částice přesunou o 60 stupňů ve směru, aby nedošlo ke kolizi částic. Osamocená částice zůstane na své pozici.

Jednotlivé buňky se mohou pohybovat v diagonálním směru. Jelikož pravidlo obsahuje pravděpodobnost přechodu z počátečního stavu do následujícího není model *reverzibilní*. Jelikož model obsahuje více pravidel než FHP-I, ale oproti FHP-III neobsahuje *rest částice*, je vhodný pro simulaci tekutin s průměrnou viskozitou.

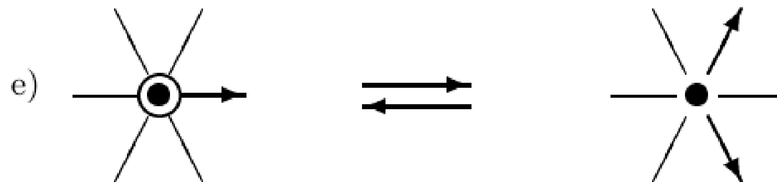


Obrázek 2.19: Grafické znázornění pravidel FHP-II modelu [41], kde šipka určuje vektor momentu částice. Pozice obsazené částicí jsou znázorněny pomocí šipky, prázdné pozice jsou zobrazeny jako tenká čára. Pravidla *a)* a *b)* jsou stejná jako v případu FHP-I na obrázku 2.18.

FHP-III

Model třetího řádu je pouze rozšířenou verzí modelu druhého řádu. Jsou definovány komplexnější pravidla pro kolize částic uvnitř buněk. Buňka obsahuje maximálně sedm částic, kde poslední částice, o kterou je simulátor rozšířený oproti modelu druhého řádu, se nazývá *rest částice*. Částice nemá určený směr ani rychlost, ale má definovanou hmotnost, která je stejná jako ostatní částice. Funguje pouze jako další proměnná v kolizích. Pravidla o která je model rozšířený, jsou na obrázku 2.20, lze je popsat následovně:

- Pokud buňka obsahuje právě jednu částici a jednu rest částici, potom se aktivní částice přesune o 60 stupňů v možném směru a z rest částice se stane aktivní částice v 120 stupňovém úhlu vzhledem k původní částici na nové pozici.



Obrázek 2.20: Grafické znázornění pravidel pro FHP-III model [41], kde šipka určuje vektor momentu částice. Pozice obsazené částicí jsou znázorněny pomocí šipky, prázdné pozice jsou zobrazeny jako tenká čára. Pravidlo *e)* obsahuje *rest částici* znázorněnou pomocí kruhu kolem středu. Pravidla *a)* až *d)* jsou stejná jako v případu FHP-II na obrázku 2.19.

Jednotlivé buňky se mohou pohybovat v diagonálním směru. Jelikož pravidlo obsahuje pravděpodobnost přechodu z počátečního stavu do následujícího není model *reverzibilní*. Model obsahuje oproti FHP-II pravidla pro *rest částici*, proto je vhodnější pro tekutiny s vyšší viskozitou.

Třírozměrné LGCA

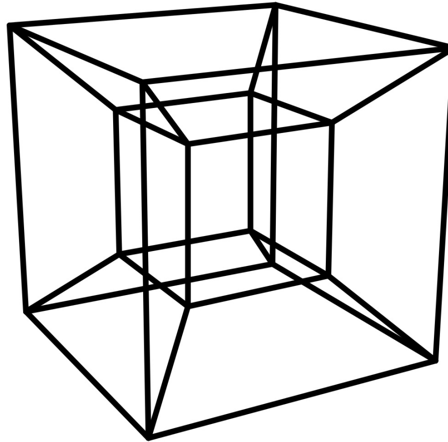
Hydrodynamika je třírozměrný problém a v tom případě je velmi důležité rozšířit dvourozměrné modely o jednu dimenzi, aby byly schopné simulovat složitější jevy. Složitost tohoto úkonu není v problému definovat kolizní pravidla mezi částicemi, ty jsou relativně bez obtíží definovatelné. Problémem jsou silné izotropní vlastnosti trojrozměrného simulačního modelu. Ukázalo se, že neexistuje pravidelná trojrozměrná mřížka s dostatečnou symetrií, aby garantovala, že tensory čtvrtého řádu jsou izotropní.

Proto bylo navrženo, aby se pro simulaci v trojrozměrném prostoru používala čtyřrozměrná mřížka a výsledek se následně převedl zpět do tří rozměrů. Příkladný model aplikující předem zmíněnou teorii je *FCHC*.

FCHC

Model pojmenovaný *Face Centered Hyper Cube* [23], nebo taky označovaný jako *D4Q24*, (první část značí, že se počítá v 4D prostoru a druhá část označuje, že je použito 24 přechodových pravidel) většinou nazývaný pouze FCHC. Model navrhli d'Humieres, Lallemand a Frisch.

Hlavní myšlenkou modelu je právě vytvořit čtyřrozměrnou FCHC mřížku, která je izotropní a následně promítnout výsledek do třírozměrného prostoru. Ve čtyřrozměrném FCHC modelu je možných 24 různých směrů pohybu částice v propagační fázi, které korespondují s jednotlivými stranami nadkrychle. Po promítnutí výsledku výpočtu do 3D prostoru získáme více rychlostní model se dvěma částicemi na každé hraně.



Obrázek 2.21: Zobrazení čtyřrozměrné nadkrychle [16] pro popis okolí buněk v třírozměrných modelech, jako je například FCHC.

Kapitola 3

Použité knihovny a nástroje

V následující kapitole jsou popsány jednotlivé nástroje a knihovny, které byly použity při implementaci. V první části je popsán WebAssembly stantard. Tento standard byl vybrán kvůli jednoduchému nasazení projektu do výuky pomocí webového prohlížeče. Druhá část obsahuje informace o standardu WebGL 2.0. Následně jsou uvedeny knihovny, které byly použity pro vytvoření grafického rozhraní. Zmíněné jsou taky knihovny a nástroje, které byly použity pro překlad aplikace a v poslední části jsou informace o knihovně, vybrané pro jednotkové testování a vytvoření testovací sady. Celkově byla snaha vytvořit program, který je velmi jednoduchý nasadit do produkce a obsahuje co nejmenší počet závislostí.

3.1 Webový standard WebAssembly

Webové služby už delší dobu neposkytují pouze statický obsah a pro silně dynamické webové stránky není rychlost JavaScriptu dostačující. Byla snaha řešit tento problém například pomocí *Just in time* neboli JIT optimalizací. Nejvíce časově náročnou operací v JavaScriptu není samotné provádění programu, ale protože se jedná o interpretovaný jazyk, nejpomalejší částí je překlad z textu do strojového kódu. To zahrnuje vytváření *abstraktního syntaktického stromu* [44]. Samotný překlad je bez optimalizací, které se provádějí až při běhu programu, pokud část programu je prováděna často. *Abstraktní syntaktický strom*, nebo-li AST, je logická reprezentace JavaScriptového programu uložená v paměti. WebAssembly nepotřebuje žádnou z těchto částí. Zdrojový program napsaný ve vysokoúrovňovém jazyku se ještě u vývojáře přeloží do strojového kódu a zároveň dojde ke všem potřebným optimalizacím. Webová stránka si stahuje optimalizovaný kód, který může provádět.

WebAssembly, nebo-li *wasm*, je binární instrukční formát pro virtuální stroje založených na zásobníkovém schématu. Je navržený jako multiplatformní cíl pro překlad z vysokoúrovňových jazyků, jako je například *C, C++ a Rust.* Dočasně omezené jsou jazyky s automatickou správou paměti, která ještě není podporována. WebAssembly podporuje většina předních webových prohlížečů, například *Firefox, Safari, Edge, Chromium a deriváty.*

Standardy předcházející WebAssembly

Před příchodem WebAssembly existovaly pouze řešení, které nebyly ještě rozšířeny a byly specifické pro určitý prohlížeč a jeho výrobce. Technologie založená na stejném principu byla *Google Native Client* (neboli *NaCl*). Tohle řešení bylo ale zbytečně komplikované a proto se nikdy neuchytilo.

Dalším řešením vyvíjeným ve stejné době bylo *asm.js*, kde se vysokoúrovňové jazyky překládaly do optimalizované podmnožiny JavaScriptu. Tohle řešení zase postrádalo potřebný skok ve výkonu, aby se začalo masově používat. Posloužilo jako základní kolekce vlastností, které se implementovaly do WebAssembly. V *červnu 2015* byl oznámený projekt, který spojoval předem zmíněné projekty do jednoho. Tím vzniklo *WebAssembly.*

asm.js

Asm.js je podmnožina JavaScriptu navržená tak, aby výsledný kód byl co nejrychlejší. Posloužil jako základ pro správný návrh WebAssembly. Proto lze stejný kód přeložit jak do asm.js, tak do WebAssembly pomocí nástroje *Emscripten.* Asm.js sloužil jako minimální soubor vlastností, které musel obsahovat WebAssembly před jeho prvním hlavním vydáním.

Rozdíl mezi těmito standardy je hlavně, že asm.js se překládá do JavaScriptu, který lze spustit v téměř jakémkoliv webovém prohlížeči. Pro WebAssembly musí být speciální podpora v prohlížeči. Nevýhodou asm.js je zase rychlost, která oproti webovým aplikacím napsaných v JavaScriptu po optimalizaci není tak výrazná. Asm.js je pouze hodně optimalizovaný JavaScript, ale WebAssembly je celý nový standard, který by měl posunout rychlost aplikací v prohlížeči na téměř nativní rychlost.

NaCl

Google Native Client, zkratkou *NaCl*, je standard pro prohlížeče k vytvoření izolovaných běhových prostředí pro bezpečný a efektivní běh přeložených C a C++ zdrojových souborů nezávisle na uživatelském operačním systému.

Existuje i rozšířená verze *Portable Native Client*, která rozšiřuje technologii o nezávislost na architektuře procesoru. Program, přeložený pomocí *PNaCl*, lze potom spustit na jakémkoliv podporovaném operačním systému.

Hlavní výhodou NaCl (případně PNaCl) je, že aplikace je uzavřená ve svém vlastním běhovém prostředí, potom rychlost s *ahead-of-time* optimalizacemi a přenositelnost. Všechny tyto výhody převzalo WebAssembly z projektu a snaží se je ještě rozvinout. Podpora pro Native Client byla ukončena [9] a postupně i výrobce NaCl Google se snaží přejít na jednotný standard, kterým je nyní WebAssembly.

Chybějící funkce WebAssembleru

Standard WebAssembly je ještě velmi mladý, proto chybí návrhy a implementace některých vlastností, které brání použití tohoto standardu s určitými jazyky. Například není ještě definované, jak bude fungovat *Garbage Collector*, proto jazyky jako *Java* nebo *Python* nemohou být přeložené do webového standardu wasm. Aktuální verze je 1.0, která je označována jako *MVP (Minimum Viable Product)*, značící, že standard je na úrovni použitelnosti, ale ještě nějaké funkce nebudou hotové nebo optimalizované na použitelnou úroveň pro výpočetně náročné programy. Následuje výpis chybějících vlastností, které mají velký vliv na funkcionalitu jazyků ve WebAssembly a jsou jedním z hlavních důvodů proč nejsou některé jazyky ještě použitelné.

- Vlákna
- Obsluha výjimek
- Zřetězené volání

- Import/Export proměnných
- Garbage collector
- Integrace s ECMAScript moduly
- Přístup na DOM

Z pohledu C++ je největším omezením chybějící standard pro *obsluhu výjimek* a *vlákna*. Hlavně chybějící jsou výjimky, protože jsou použity taky v C++ standardní knihovně. Ve WebAssembleru jsou všechny výjimky nahrazeny voláním *assert*, takže pokud při vykonávání kódu dojde k bodu, kde by se měla vytvořit výjimka, tak místo toho se celý program ukončí. Tohle chování není vůbec vhodné, ale zatím neexistuje žádný způsob, jak tomuto chování zabránit. Z tohoto důvodu v celém kódu implementace nebyly použity žádné výjimky. Funkce ze standardní knihovny byly použity jako *noexcept* pokud existovaly, jinak byly použity klasické funkce.

Druhou chybějící vlastností omezující C++ implementaci jsou vlákna. Jelikož celulární automaty jsou navržené způsobem, aby byly spustitelné vícevláknově a souběžně, je toto omezení značnou nevýhodou. Souběžný výpočet by ušetřil velké množství času. Naštěstí tato chybějící vlastnost neomezuje žádným směrem způsob implementace simulátoru.

Pro ostatní programovací jazyky je hlavním omezením chybějící *Garbage Collector*. Velká skupina moderních programovacích jazyků, jako jsou například *Java, Python a Go*, spoléhají na automatizovanou správu paměti. Tento nedostatek je v plánu odstranit hned po vláknech, které mají nyní nejvyšší prioritu [3].

Systémové rozhraní

Díky popularitě a praktické funkčnosti WebAssembly vznikají projekty, které navazují a rozšiřují funkcionalitu. Jedním z nich je *The WebAssembly System Interface* [7], označovaný jako *wasi*. Tento formát rozšiřuje WebAssembly o možnost spustit wasm formát bez nutnosti prohlížeče. Formát je zpětně kompatibilní. To znamená, že wasm formát lze interpretovat v běhovém prostředí pro wasi formát. Jedním z běhových prostředí je *wasmtime* [8], který funguje jako interpret wasi formátu na *desktopu*. Obsahuje i *JS polyfill* [26] pro běh aplikace v prohlížeči.

Hlavními nedostatky wasi formátu jsou, že existuje příliš krátkou dobu, není pořádně otestovaný a podporovaný. Standard a ani implementace není ještě ve finálním stádiu, částečně i proto, že wasm formát, ze kterého tento formát vychází, není taky kompletní. Z těchto důvodů nebyl standard použitý jako cíl překladu aplikace. Nicméně, při úplné specifikaci standardu, by měla jít aplikace bez jakékoliv komplikace přeložit do wasi formátu.

3.2 JavaScript API WebGL

Byla použita verze *WebGL 2.0* [25]. Je to JavaScriptové API pro vykreslování interaktivních 3D a 2D grafických prvků s kompatibilní vrstvou pro webové prohlížeče, bez potřeby rozšíření pro daný prohlížeč. Zmíněné vlastnosti se dosáhlo vytvořením API, které je silně kompatibilní s OpenGL ES 3.0. Webový standard lze použít v *HTML5 elementu <canvas>*. Nejnovější verze WebGL je podporovaná WebAssemblerem. Standard WebGL 2.0 podporují všichni přední výrobci prohlížečů, jako jsou Firefox, Safari, Edge a Chrome.

3.3 Grafická knihovna ImGui

ImGui je grafická knihovna napsaná v C++ s minimálními závislostmi a s nejmenší možnou vrstvou abstrakce. Práce s knihovnou by měla být jednoduchá, ale nesmí dojít k velkému snížení výkonu pouze kvůli vysoké úrovni abstrakce. Využívá *Immediate Mode Graphical User interface model* [28]. Více informací je uvedeno v sekci 3.3. Jsou zajištěny pouze základní grafické prvky s velmi jednoduchým stylem.

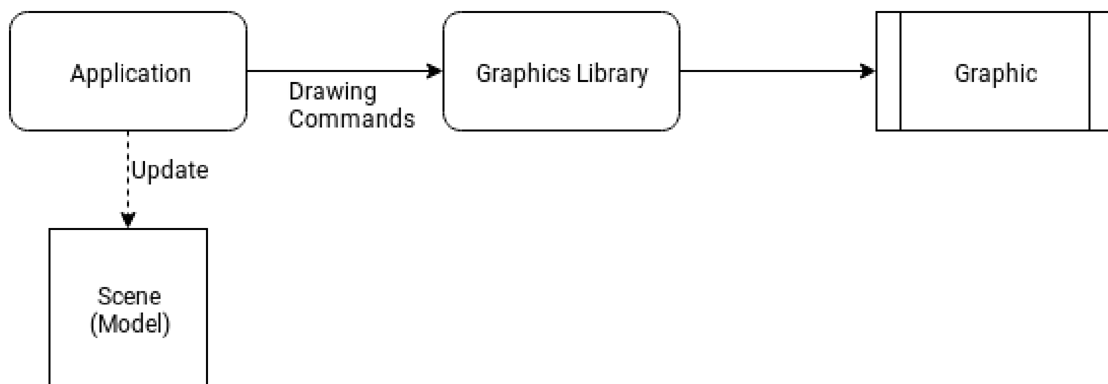
Hlavní myšlenka je, že se vytvoří *vertex buffer objekty (VBO)*, které potom programátor využije až bude potřebovat v jakékoliv části programu. Celá knihovna je *multiplatformní*, má pouze jednu závislost a to *OpenGL* verze specifikované programátorem, kterou využívá pro vykreslování.

Použitý model grafického rozhraní

Knihovna *ImGui* používá model *immediate mode* pro zobrazování a práci s elementy. Jakým způsobem funguje tento model je popsáno v další části. Model nedefinuje jenom vnitřní logiku knihovny, ale i její vnější způsob používání, proto je velmi důležité vědět, jaký model knihovna používá.

Immediate Mode Graphical User Interface

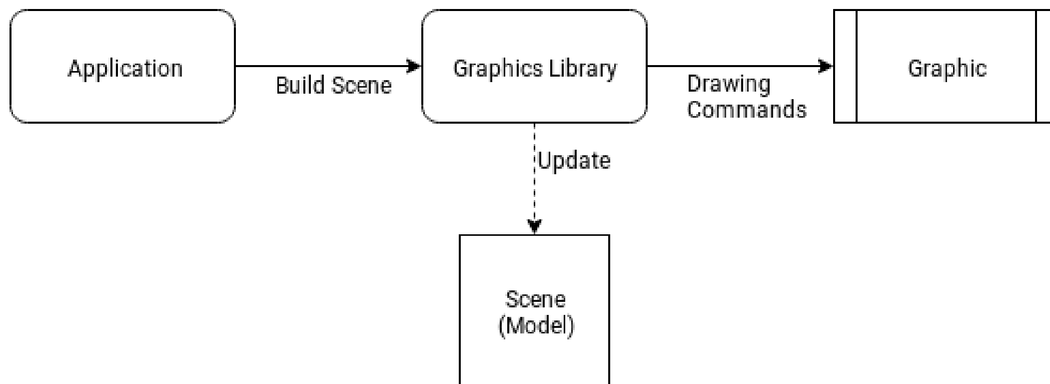
Immediate grafický model, často označovaný pouze jako *IMGUI*, je princip vykreslování používaný hlavně ve hrách, kde je potřeba vykreslit scénu každý snímek. Graficky znázorněný je na obrázku 3.1



Obrázek 3.1: Obrázek logiky funkčnosti grafických knihoven [28] využívajících *Immediate mode*.

Retained Mode Graphical User Interface

Retained mode je nyní nejpoužívanější model [27] pro grafická rozhraní. Používá objekty, které jsou vytvořeny většinou při spuštění aplikace a pouze mění svoje stavy. Každý objekt může obsahovat data a metody reagující na události, které se staly v grafickém rozhraní. Graficky znázorněný je na obrázku 3.2



Obrázek 3.2: Obrázek logiky funkčnosti grafických knihoven [28] využívající *Retained mode*.

Rozdíl modelů pro grafické rozhraní

Hlavní rozdíl je, že v *retained mode* každý objekt může obsahovat data a metody reagující na události. V *immediate mode* je každý objekt pouze grafický prvek ve scéně a logika je řešená zvlášť. Tento přístup má výhody pro jednodušší způsob vykreslování a rychlejší křivku učení. Nevýhodou je, že k vykreslování dochází v každém snímku a grafické prvky jsou kombinovány v implementaci s logickou částí aplikace.

Porovnání ImGui s grafickými knihovnami Qt a Gtk

Je snaha, aby celý program byl co nejvíce *multiplatformní*, proto použití *Gtk* [4] knihovny, která má nulovou podporu ze strany vývojářů pro běh na systémech *Windows*, je vyloučeno jako první. Grafická knihovna *Qt* [6] je jedna z nejrozšířenějších C++ knihoven na multiplatformní grafická rozhraní. Podporuje překlad do WebAssembly a práce s knihovnou není nijak složitá. Nevýhodou použití této knihovny je, že výsledný binární soubor je zbytečně objemný a závisí na dalších sdílených knihovnách. Simulátor je s jednoduchým grafickým rozhraním a pracovní pole je napsáno v čistém *WebGL*, proto použití knihovny Qt je zbytečně složité. Proto z již zmíněných důvodů byla použita knihovna ImGui. Dovoluje vytvořit jednoduché grafické rozhraní, nemá žádné externí závislosti a používá pouze OpenGL. Z těchto důvodů není výsledný binární soubor paměťově náročný a nezávisí na žádných dalších sdílených knihovnách.

3.4 Sada nástrojů pro překlad CMake

CMake je svobodná, multiplatformní kolekce nástrojů navržená pro překlad, testování a distribuci aplikací. Používá se pro kontrolovaný překlad aplikace pomocí jednoduchých konfiguračních souborů nezávislých na platformě nebo na překladači. Hlavní účel CMake nástroje je generování skutečných konfiguračních souborů, pomocí kterých se aplikace překládá. Z toho vyplývá, že CMake samostatně nepřekládá aplikaci, pouze poskytuje jednotnou abstrakci nad různými překladači. CMake je vyvíjený společností *Kitware* jako odpověď na potřebu výkoného, multiplatformního překladového prostředí pro svobodné projekty.

První verze 1.0.0 byla vydána v roce 2000. Od té doby prošel tprojekt velkou sérií změn. Od verze 3.10 vznikl nový standard, který definuje, jakým správným způsobem by se měly psát konfigurační skripty, protože kvůli zpětné kompatibilitě podporuje funkce, které nejsou

vhodné a můžou být i nebezpečné. Tento standard se nazývá *Moderní CMake*. Od verze 3.14, která ještě nebyla vydána, vzniká upravený standard. Ten přináší pouze drobné změny a dokončuje hlavní myšlenku, proč celý nový standard vznikal.

V projektu je CMake použitý přesně kvůli předem zmíněným důvodům. Dovoluje pomocí jedné sady konfiguračních souborů definovat překlad pro všechny platformy. Dalším hlavním důvodem byla podpora platformy *Emscriptem*, kdy pouze s drobnými změnami je projekt možný přeložit pro webové rozhraní. Více informací ohledně správného překladu aplikace je v kapitole 5.

Překlad aplikace pro WebAssembly

Překlad aplikace pomocí nástroje CMake a Emscriptenu je přímočará operace a neliší se nijak oproti klasickému překladu pro cizí platformu. Tento způsob překladu se nazývá *Cross Compile*.

První krok je lokálně nainstalovat *Emscripten SDK* [2], toho lze na *Linuxu* dosáhnou sadou příkazů podle obrázku 3.1. Jako závislosti pro překlad je potřeba nainstalovat *Python, node.js, CMake a Javu*, kde Java není povinná závislost (je potřebná pro Closure Compiler, který minifikuje výsledný soubor).

```
1  # Download of the latest SDK tools
2  git clone https://github.com/emscripten-core/emsdk.git
3
4  # Installation of the latest SDK tools
5  ./emsdk install latest
6
7  # Make the "latest" SDK "active" for the current user
8  ./emsdk activate latest
9
10 # Activate PATH and other environment variables in the current terminal
11 source ./emsdk_env.sh
```

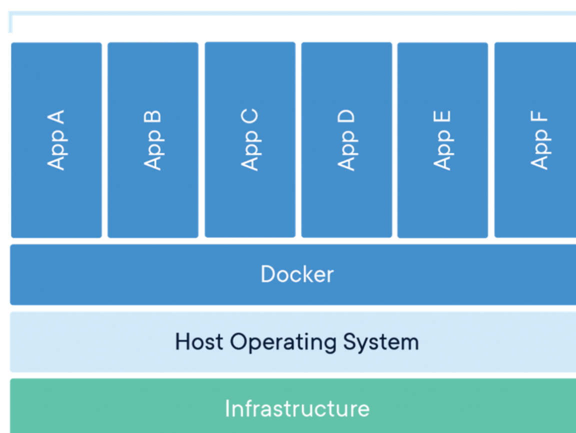
Obrázek 3.1: Postup instalace a aktivace nástroje *Emscripten*.

Druhý krok je překlad aplikace pomocí staženého *Emscripten SDK*. Překlad se neliší od klasického překladu pomocí nástroje CMake, proto zde není popsaný. Jediný rozdílný krok je použití *Emsdk* jako souboru nástrojů pro překlad. Toho se docílí pomocí nastavení následující proměnné:

```
-DCMAKE_TOOLCHAIN_FILE=/<path-to-emsdk-platform>/Emscripten.cmake
```

3.5 Překladové a běhové prostředí Docker

Docker [24] je nástroj pro běh aplikací v *kontejnerech*. Kontejnery jsou vzájemně izolované a každý obsahuje plnohodnotnou aplikaci, potřebné nástroje, knihovny a konfiguraci. Jednotlivé kontejnery jsou běhové instance obrazu aplikace. Pro popis obrazu se používá *Dockerfile*. Kontejner odstraňuje potřebu instalovat vývojové závislosti a má předem definováno, jaké verze závislostí budou přítomny v obrazu. Architektura, jak jsou kontejnery navrženy, lze vidět na obrázku 3.3.



Obrázek 3.3: Architektura [10] použití kontejnerů jako běhové prostředí pro aplikace.

Využití pro servery

Kontejnerizace se používá hlavně na serverech pro *vertikální škálovatelnost* webových aplikací. Spuštěním více instancí jednoho obrazu a vytvořením společného veřejného rozhraní, vznikne možnost zrychlovat službu, bez potřeby existence jednoho výpočetně výkonného stroje. Zatížení se rozloží rovnoměrně na jednotlivé běžící uzly, spuštěním pouze jednoho kontejneru na každém uzlu. Jednotlivé instance mají společné veřejné rozhraní, proto pro klienta vše funguje jako jeden server. Tento systém je použitý při orchestraci kontejnerů pomocí *Kubernetes*.

Využití lokálně

Další možnost využití je spuštění lokálně pouze jednoho kontejneru. Výhodou je, že spuštěný obraz nemá žádnou externí závislost. Není proto potřeba instalovat žádné závislosti. Tento způsob je použitý v projektu pro jednoduché nasazení aplikace. Způsob, jakým se má aplikace spustit, je popsaný v kapitole 5.3.

3.6 Testovací knihovna GTest

Pro jednotkové testování je použita knihovna *GTest* [12] od společnosti *Google*. Je to jedna z nejrozšířenějších testovacích knihoven pro jazyk C++. Existuje i nativní podpora v překladovém systému *CMake* pomocí nástroje *CTest*.

Google testy fungují na principu, kdy pro každou skupinu jednotkových testů pokrývající logický celek, většinou modul, se vytvoří binární spustitelný soubor. Tyto jednotlivé soubory se potom spustí a vytvoří celkovou testovací sadu pro aplikaci. Výhodou je, že lze přeložit a spustit testy pro jednotlivé moduly zvlášť.

Integrace s nástrojem CMake

CMake systém podporuje od verze 3.12 knihovnu *GTest* jako standard pro jednotkové testování C a C++ částí programů. Byla vytvořena pro testy abstraktní vrstva, která se nazývá *CTest*. Tato vrstva si registruje jednotlivé binární soubory obsahující testy. Potom lze jedním příkazem spustit všechny registrované testy současně. CTest, pokud je tak nastavený,

prochází všechny zdrojové soubory a hledá GTest makra označující určitý testovací případ. Výhodou je, že jednotkové testy můžou být napsané v jednom souboru s implementací a v tomto případě lze otestovat i *privátní implementace* přístupné pouze uvnitř jednoho souboru.

Pro povolení nástroje *CTest* a vytvoření cíle pro překlad, je potřeba v konfiguračním souboru CMake povolit testování následující funkcí:

```
enable_testing()
```

Mock testování

Mock objekt je objekt, který svým chováním napodobuje chování skutečného implementovaného objektu. Může obsahovat hlášení, která se budou zobrazovat při volání funkcí nebo může navracet předem definované hodnoty a mnoho dalšího. Způsoby mockování jsou různé a každý se hodí na jinou situaci. Vždy jsou ale součástí určitého způsobu testování. GTesty obsahují část zvanou *GMock*, která obsahuje právě zmíněnou funkcionalitu v provedení jednoduchém na použití. Aplikace používá knihovnu GTest, proto při jednotkovém testování je část GMock přístupná a lze pracovat s mock objekty.

Automatická registrace testů

Google testy se používají pomocí předdefinovaných maker, které popisují jeden případ užití nebo jednu sadu testů. Jsou definovány i makra pro porovnávání výsledku se skutečnou hodnotou. Jednou skupinou maker pro porovnání jsou *expecty* a druhou *asserty*. První skupina pouze selže, ale test pokračuje dále, ve druhé skupině, pokud dojde k chybě, celý případ užití selže.

Pro testování simulátoru byla použita podmnožina možností testovací knihovny. Hlavně byly využity takzvané *test fixtures*. Příklad 3.2 ukazuje, jakým způsobem byla knihovna použita pro testování.

```
1  class Fixture : public ::testing::Test {
2      void SetUp() override {/* Code executed before each test case */}
3      void TearDown() override {/* Code executed after each test case */}
4  };
5
6  TEST_F(Fixture, case_test) {
7      EXPECT_EQ(0, nullptr);
8      EXPECT_STREQ("String", "");
9      ASSERT_NO_THROW(throwing_function());
10 }
```

Obrázek 3.2: Použití v projektu testovacího nástroje *GTest* v jazyku C++.

Kapitola 4

Návrh LGCA simulátoru Caflow

V této kapitole je popsaný postup a výsledek návrhu LGCA simulátoru, který je pojmenovaný Caflow. V první části jsou popsány obecné informace ohledně návrhu. V druhé části jsou specifikovány přístupy, které byly použity a jejich vysvětlení, proč se rozhodlo zrovna daným způsobem.

Aplikace pro LGCA modely vznikla hlavně z důvodu jednoduché prezentace praktické funkcionality simulačního modelu. Při celém návrhu byl kladen důraz na správnou vizualizaci výsledku simulace, jednoduchost použití simulátoru a jednoduchost jeho nasazení v produkci.

Simulátor by měl hlavně výrazně prezentovat hlavní myšlenku celulárních automatů a to, že pouze pomocí jednoduché sady pravidel definovaných na mikroskopické úrovni, lze pozorovat komplexní jevy na makroskopické úrovni. Tohle tvrzení je už matematicky podloženo důkazy, že daná pravidla skutečně implementují vzory, které by byly bez určité úrovně abstrakce velmi složité na popis. Všechny důkazy jsou shrnuty v literatuře [11].

4.1 Graf závislostí komponent

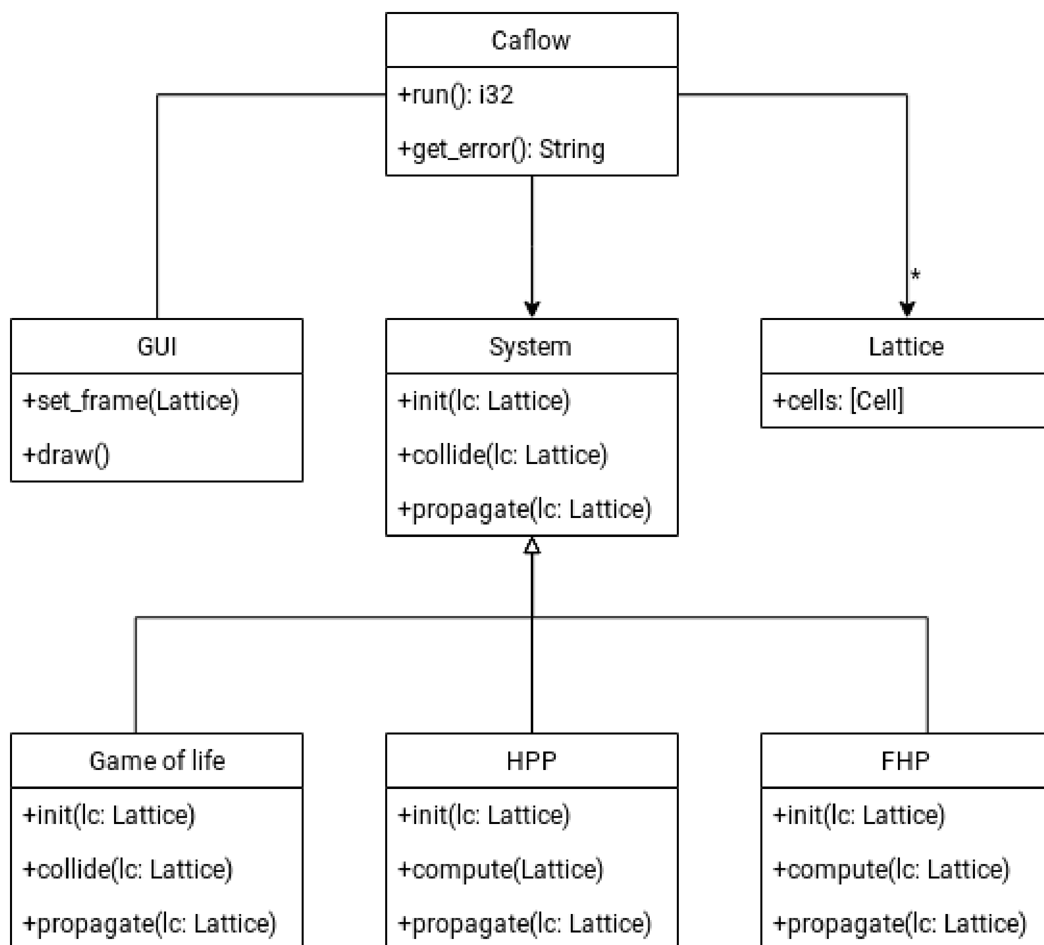
Na obrázku 4.1 lze vidět grafickou reprezentaci závislostí a vztahů mezi hlavními třídami aplikace. Grafická část je reprezentována pouze jako jedna třída. Pomocné třídy, které nejsou nutné pro program jsou vynechány. Třída *Caflow* vlastní všechny třídy potřebné pro simulaci. Třída *System* je pouze virtuální a definuje, jakým způsobem se bude komunikovat s běžícím systémem. Všechny implementované systémy dědí z této třídy. Třída *GUI* zobrazuje výsledek simulace a přeposílá události od uživatele třídě Caflow. Třída *Lattice* obsahuje vypočítaný stav simulace. Caflow obsahuje pole instancí Lattice kvůli historii simulace.

4.2 Použité návrhové vzory

V této sekci jsou popsány všechny důležité nebo zajímavé návrhové vzory, použité v projektu. Některé byly použity pro zjednodušení práce s objekty, jiné zase pro rozdělení logiky objektu a některé pro optimalizaci výkonu aplikace.

Datově orientovaný návrh

Datově orientovaný návrh [17] je způsob návrhu, kdy se upřednostňují samotná data před funkcemi, které mají abstrahovat právě zmíněná data při použití. V objektově orientovaném



Obrázek 4.1: Diagram vztahů hlavních tříd. Grafická část je reprezentována pouze jako jedna třída.

návrhu je kladen důraz hlavně na operace pracující nad daty a jejich zapouzdření. To znamená, že existuje celek, který obsahuje data nepřístupné z okolí a funkce, které pracují právě nad daty a tím vytvoří vnější rozhraní pro komunikaci. Pro větší skupinu dat stejného typu se vytvoří například pole, kde každá instance třídy má definovány svoje metody. Tento vzor lze popsat jako *pole struktur*.

V datově orientovaném návrhu existuje jeden druh třídy, často pojmenovaný jako *Systém*, který operuje nad daty, která nemají definováno rozhraní k jejich přístupu. Jediný kdo může přistupovat na data je právě Systém. To znamená, že zapouzdření a abstrakce je stále zachována, ale je posunuta na jinou úroveň, aby bylo možné aplikovat optimalizace na skupinu prvků. Vzor je velmi vhodný na velké skupiny stejných prvků, se kterými se pracuje většinou lineárně. Při návrhu byl tento vzor použitý pro implementaci pole buněk celulárního automatu, jelikož se buňky prochází při změně stavu vždy lineárně pro kolizi a v předem definovaném okolí pro propagaci. Tento vzor lze popsat jako *struktura polí*.

Příklad pro použití datově orientovaného návrhu je 4.1. Operace je aplikována na každý prvek vždy, ale v jakém pořadí určuje *Systém*, proto operace může běžet paralelně na jednotlivých částech pole a zároveň nedojde k *Cache miss* operacím, protože paměť je lineárně uložená.

```
// Some structure containing all neccessary data
struct Cell {
    // some data
}

class LatticeSystem {
public:
    void collide(std::function<void(Cell&)> fun) {
        for (auto& d: _lattice) {
            fun(d); // Collide particles inside a cell
        }
    }
private:
    std::vector<Cell> _lattice;
}
```

Obrázek 4.1: Použití datově orientovaného návrhu v jazyku C++.

Šablonový vzor CRTP

Curiously recuring template pattern je šablonový návrhový vzor, který je použitý pro odstranění virtuálních metod. Způsob použití je, že se vytvoří šablony s definovanými metodami, které musí být implementované ve třídě, která z ní dědí. Potom se nepoužívá ukazatel na rodičovskou třídu, protože neexistuje, ale používá se *variant* obsahující všechny možné implementace zděděných tříd. Pro přístup na metodu se používá v C++ vzor *Visitor*, který určuje, jaká metoda ze které instance třídy se použije při volání. Jeden ze příkladu použití je na obrázku 4.2.

```
// Base class
template<typename T>
class System {
public:
    // Methods that must be defined in inherited class
    void collide() {
        static_cast<T*>(this)->collide();
    }
};

// Inherited class
class Fhp1System : public System<Fhp1System> {
public:
    void collide() {/* Collide particles */}
};

// Then use case can be something like
using Systems = std::variant<std::monostate, Fhp1System>;

```

```
20 static void collide(Systems systems) {
21     std::visit([](auto&& arg) {
22             using T = std::decay_t<decltype(arg)>;
23             if constexpr (!std::is_same_v<T, std::monostate>)
24                 // Do something with value
25     }, systems);
26 }
```

Obrázek 4.2: Minimální příklad šablonového vzoru *curiously recuring* v jazyku C++.

Velkou nevýhodou při implementaci je chybové hlášení překladače, kdy při zapomenutí implementace zděděné metody dojde k chybě značící chybějící specializaci definované metody ze šablony, které v aktuálním stavu jazyka C++ není přehledné. Tento nedostatek by se měl zlepšit s příchodem standardu *C++20* a nových jazykových vlastností *constraints a concepts*.

Vzor pro instanciaci tříd Builder pattern

Při implementaci konstruktorů jednotlivých tříd byl použit *Builder pattern* [1]. Tento návrhový vzor je vhodný hlavně pro nepovinné parametry a kontrolu vstupních parametrů ještě před vytvořením výsledné instance. Implementace spočívá ve vytvoření další třídy, kde pomocí volání metod nad danou třídou nastavíme parametry, které se nakonec předají třídě, kterou chceme vytvořit. Při porovnání s metodou, kde bychom vytvořili instanci třídy a potom předali pomocí volání metod volitelné parametry, má *Builder pattern* hlavně výhodu, že instance třídy neexistuje dokud nebylo její vytváření ukončeno pomocí metody *build()*. Další výhoda je, že ještě při vytváření instance lze zkontrolovat, že kombinace volitelných parametrů je validní. Po vytvoření instance se už nedají měnit volitelné parametry. Tímto způsobem jsou od sebe odděleny metody, které pouze nastavují stav třídy a ostatní. Příklad implementace je popsán na obrázku 4.3.

```
1  // Forward declaration
2  class CellBuilder;
3
4  // Some base class
5  class Cell {
6      // Builder must be friend to initialize private values
7      friend CellBuilder;
8  public:
9      // Create builder
10     static auto create() -> CellBuilder;
11 private:
12     int _width;
13 };
14
15 // Builder class
16 class CellBuilder {
17 public:
18     // Build unique pointer of a Cell with predefined values
19     auto build_box() -> std::unique_ptr<Cell> {
```

```
20        auto res = std::make_unique<Cell>();
21        res->_width = _width;
22        return std::move(res);
23    }
24
25    // Set result value of a Cell
26    // This parameter is optional
27    auto set_width(int width) -> CellBuilder& {
28        _width = width;
29        return *this;
30    }
31 private:
32    int _width = 0; //< Some value transfered to cell
33 };
34
35 // Return builder
36 inline auto Cell::create() -> CellBuilder {
37    return CellBuilder();
38 }
39
40 // Use case can be something like this
41 int main() {
42    auto cell = Cell::create()
43            .set_width(42)
44            .build_box();
45 }
```

Obrázek 4.3: Příklad implementace vzoru pro instanciaci tříd *Builder pattern* v jazyku C++.

Například chceme vytvořit třídu *Lattice*, která obsahuje parametry *width* a *height*. Parametry můžou být nastaveny na určitou hodnotu a nebo budou použity přednastavené hodnoty. Potom vytvoříme třídu *LatticeBuilder*, která bude obsahovat metody *width()*, *height()* a *build()*. První dvě metody nastaví výsledné nepovinné parametry a poslední metoda vytvoří požadovanou instanci třídy s danými parametry. *Builder* může přesunout všechny hodnoty do jedné instance třídy Lattice nebo může pouze zkopírovat hodnoty. Druhým způsobem funguje jako továrna pro vytváření více podobných instancí.

Zpracování událostí pomocí callback funkcí

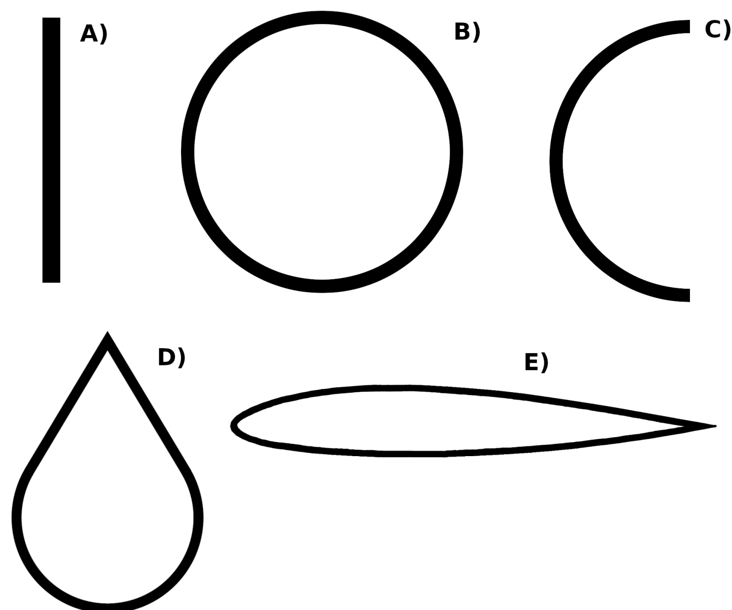
Použitá grafická knihovna ImGui neřeší žádným způsobem reakce na změny a události od uživatele, proto byla použita při implementaci metoda funkce zpětného volání, neboli *callback funkce*, kde každé instanci grafického elementu lze při vytváření předat lambda funkce, které se při nastání určité události provedou. Výhodou popsané metody je, že se odstraní potřeba neustálého předávání stavů elementů v každém cyklu. Nevýhodou je, že metoda je výkonnostně pomalejší než přímé předávání stavu. Z tohoto důvodu není vhodná pro kritické části kódu. Příklad použití lze najít na obrázku 4.4.

```
1  class Window {
2      using ResizeFn = std::function<void(int, int)>;
3  public:
4      void on_resize(ResizeFn&& callback) {
5          _resize = std::move(callback);
6      }
7  private:
8      ResizeFn _resize = nullptr;
9  };
10
11 int main() {
12     auto window = std::make_unique<Window>();
13     window->on_resize([](int width, int height) {
14         // Resize window when called
15     });
16 }
```

Obrázek 4.4: Příklad implementace *callback funkcí* v jazyku C++.

4.3 Demonstrační modely

Součástí programu je pět demonstračních modelů pro zobrazení chování tekutiny při obtékání těles. Různorodost tvarů modelů by měla zajistit vzájemně odlišné chování tekutiny při simulaci. Modely jsou na obrázku 4.2 a jsou seřazeny podle složitosti.



Obrázek 4.2: Vybráné modely pro demonstraci chování LGCA metody při obtékání těles, kde druhy modelů jsou A) přímka, B) kruh, C) půlkruh, D) tvar s nejmenším činitelem odporu a E) tvar křídla.

Kapitola 5

Implementační detaily

V první části této kapitoly je popsán způsob, jakým byla zajištěna správa paměti, dále struktura zdrojových souborů a jejich rozdělení do modulů použitých při implementaci celulárního automatu. V druhé části je popsaný způsob překladu aplikace pomocí CMake nebo obrazu pro nástroj Docker. Poslední část kapitoly obsahuje obrázky grafického rozhraní výsledné aplikace a všechny možnosti vizualizace výsledku simulace.

5.1 Automatická správa paměti

Pro správu paměti jsou použity v projektu *chytré ukazatele*. Každý ukazatel určitým způsobem vlastní danou paměť, která je ve správnou chvíli uvolněna. Nejsou proto potřeba a ani by se neměly vyskytovat volání funkcí *new* nebo *delete*, protože ruční odstraňování paměti bývá velmi komplikované a způsobuje paměťové chyby v programu. Druhy chytrých ukazatelů jsou následující:

- *Vlastnící ukazatel* (`std::unique_ptr`), kdy ukazatel plně vlastní danou paměť. Měl by existovat pouze jeden ukazatel na určitou paměť. Při zániku ukazatele zaniká i paměť pokud nebyla explicitně přiřazena k jinému ukazateli.
- *Ukazatel s počítáním referencí* (`std::shared_ptr`), kdy existuje více ukazatelů na jednu část paměti. Daná část paměti obsahuje čítač ukazatelů, které pracují nad pamětí. Při zvýšení počtu ukazatelů se zvýší i číslo v čítači a naopak. Pokud se čítač vynuluje dojde k automatickému odstranění paměti.
- *Slabý ukazatel* (`std::weak_ptr`), kdy ukazatel nevlastní žádnou část paměti. Lze definovat pouze nad pamětí vlastněnou *ukazatelem s čítačem referencí.* Aby daný ukazatel mohl přistoupit k paměti musí prvně zvýšit čítač vlastníků aby nedošlo k smazání paměti při práci.

5.2 Struktura simulátoru

Implementace projektu je rozdělena do tří částí. Vedle implementace existují ještě externí závislosti umístěné ve složce *external*, které jsou ImGui, GLFW, GTest, Stb a Glad. V kořenovém adresáři je umístěný *Dockerfile* pro zjednodušení nasazení aplikace. Dále jsou zde umístěny soubory *index.html a favicon.ico* pro výsledný běh aplikace ve webovém rozhraní.

Lgca je hlavní část, kde jméno odpovídá složce, ve které se nachází všechny zdrojové soubory daného modulu. Tato část obsahuje veškerou důležitou logiku, implementaci předem zmíněných pravidel simulačních modelů a obsahuje vstupní část celého simulátoru, přes který komunikují vnější zdroje.

Gui část obsahuje grafické rozhraní a všechny prvky, které jakýmkoliv způsobem komunikují s grafickou částí aplikace. V případě potřeby by bylo možné upravit simulátor tak, aby nezávisel na grafické části a tímto způsobem by šlo spustit simulátor na hostitelském stroji, který neobsahuje žádné grafické prvky. Tento přístup by ale nevytvořil žádnou použitelnou výhodu, proto nebyl aplikovaný.

Common část, která je ze všech tří nejméně důležitá. Část obsahuje všechny sdílené části implementace. Zde jsou definovány například vzory pro zachytávání chybového stavu, šablonové třídy pro správu paměti a další.

5.3 Překladové a běhové prostředí Docker

Pro zmíněné účely v kapitole 3.5 je v projektu vytvořený *Dockerfile*, který lze využít pro nasazení aplikace do produkce, nebo lze uvnitř kontejneru vyvíjet.

Na vývojovém nebo produkčním zařízení musí být nainstalovaný *docker* a musí běžet *docker daemon* nebo *podman*, který nepotřebuje žádného běžícího *daemona*. Potom lze v kořenovém adresáři vytvořit obraz aplikace a spustit kontejner aplikace na portu 80, stejně jako na obrázku 5.1.

```
1 # Build docker image from sources in local folder
2 # and tag image with name 'caflow'
3 docker build --tag caflow .
4
5 # Run created container in detached mode locally
6 # with public port 80 mapped to internal port 8000
7 docker run -d -p 80:8000 caflow
```

Obrázek 5.1: Vytvoření a spuštění obrazu aplikace pomocí nástroje Docker.

Pro zjednodušení nasazení do produkce je obraz aplikace už vytvořený na stránce `https://hub.docker.com/`. Docker automaticky prohledává danou stránku pro vytvořené obrazy. Proto stačí pouze jeden příkaz 5.2, který stáhne obraz a spustí kontejner na hostitelském počítači.

```
1 # Run in dettached mode latest version of the container
2 # located at hub.docker.com registry
3 # and map internal port 8000 to external 80
4 docker run -d -p 80:8000 tyrkaen/caflow:latest
```

Obrázek 5.2: Spuštění externího kontejneru pomocí nástroje Docker.

5.4 Výsledná aplikace

Program lze přeložit pro desktop nebo webové prohlížeče. Otestovány byly dva různé cíle překladu. Jeden byl pro `x86_64` architekturu. Druhý překlad potom byl testovaný pro *wasm* formát.

Překlad pro desktop

Překlad programu pro desktop byl testovaný na linuxové distribuci *openSUSE Leap 15.0*. Pro překlad aplikace na jakémkoliv systému je potřeba mít nainstalované následující závislosti:

- *CMake*
- *Ninja*

Překlad pro *Windows* nebo *macOS* nepotřebuje žádné další závislosti. Pro *Linux* se závislosti liší podle protokolu pro compositor a distribuce. Aplikace může být přeložena pro protokol X.org nebo Wayland. Názvy závislostí jsou například:

- `xorg-x11-devel` nebo `wayland-devel` pro openSUSE
- `xorg-dev` nebo `libwayland-dev` pro Ubuntu

Po nainstalování závislostí stačí jenom vygenerovat všechny potřebné konfigurační soubory a spustit překlad. Způsob překladu je na obrázku 5.3.

```
1  # Generate all neccessary config files and run build
2
3  # CMake 3.13 and higher can use commands
4  cmake -G Ninja -B build . && \
5  cmake --build build
6
7  # Older versions of CMake must choose different approach
8  mkdir -p build && \
9  cd build && \
10 cmake -G Ninja .. && \
11 ninja
```

Obrázek 5.3: Překlad programu na desktop pomocí CMake a Ninja.

Pro ověření funkcionality lze spustit testy, které se přeložily zároveň s aplikací. Testy jsou nastavené jako cíl překladu pod názvem *test*, takže pro spuštění je potřeba v adresáři překladu spustit příkaz:

```
ninja test
```

Všechny příkazy, které používají program *ninja*, lze nahradit pomocí programu *make*. *Ninja* je pouze moderní přístup k překladu a například provádí překlad paralelně už v základu, taky funguje jak na systému *Linux* tak *Windows*.

Překlad pro web

Pro web je potřeba nainstalovat *Emscripten*, návod je napsaný v kapitole 3.4. Jinak jsou všechny závislosti stejné jako pro překlad na desktop a probíhá úplně stejným způsobem. Jediný rozdíl je, že se musí nastavit CMake proměnná `CMAKE_TOOLCHAIN_FILE`.

Grafické rozhraní aplikace

Grafické rozhraní aplikace je rozděleno do tří částí. Každá část obsluhuje určitý celek simulace. Při návrhu byl kladen důraz na jednoduchost použití a použitelnost pro všechny velikosti zobrazení pole. Práce s grafickým rozhraním je rozdělena do dvou kroků. V prvním kroku se v bočním panelu nastaví jednotlivé parametry simulace a spustí se výpočet. V druhém kroku se zobrazí první snímek simulace. Pomocí spodní lišty se posouvá v simulaci a ve vrchní části pravého panelu se nastavuje způsob vizualizace výsledku.

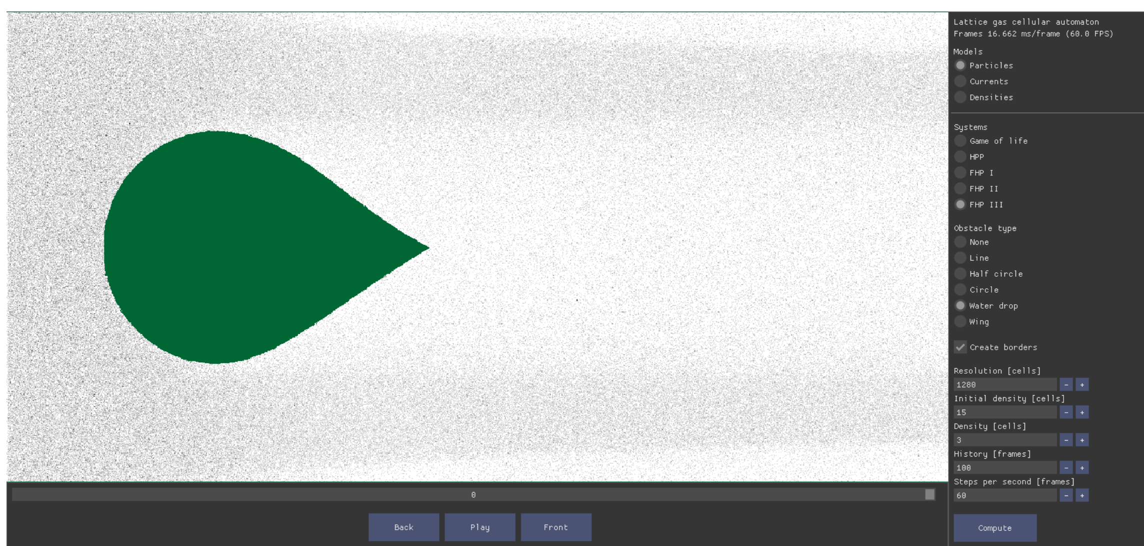
Aplikace obsahuje grafické rozhraní s možností tří různých druhů zobrazení pracovní mřížky. Rozhraní se skládá ze tří částí, kde hlavní část zobrazuje výsledek simulace. Grafické rozhraní je možno spustit i ve webovém prohlížeči.

Spodní panel je složený z posuvníku pro změnu aktuálně zobrazeného výsledku a tlačítek na pozastavení/spuštění nebo posunutí simulace. Hodnota posuvníku je v průběhu simulace stále nulová. Posunem do levé strany se čísla snižují do záporných hodnot a reprezentují, kolikátý snímek zpět od posledního vypočítaného se zobrazuje.

Boční panel obsahuje konfiguraci simulace a jejího zobrazení. V horní části je nastavení zobrazení a ve spodní části je nastavení způsobu simulace. Pro potvrzení nastavení slouží tlačítko na spodní straně panelu. Nastavení zobrazení lze měnit i při běhu simulace.

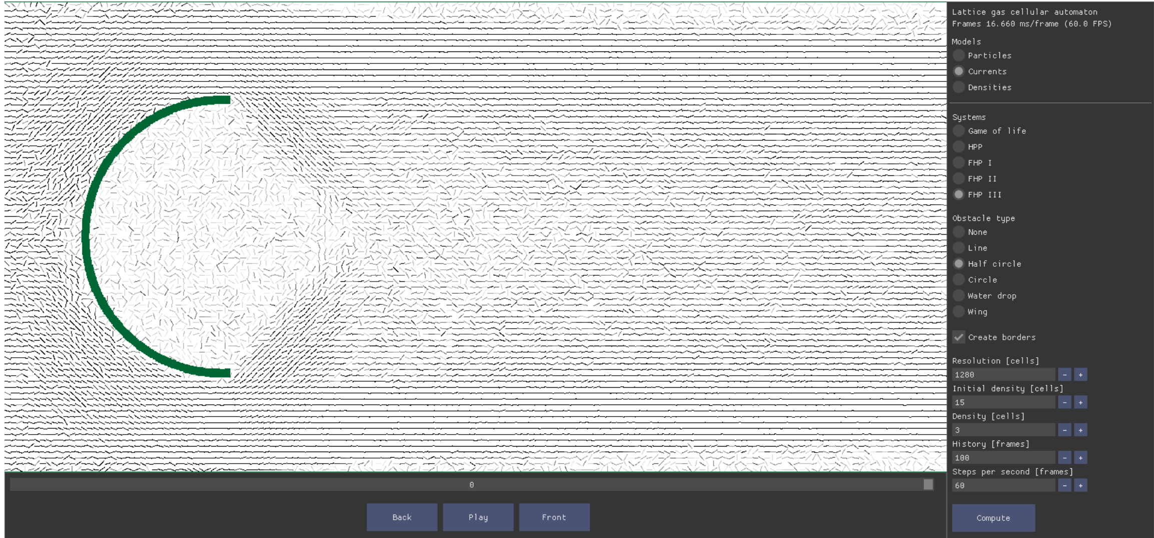
Pracovní mřížka zobrazuje aktuální výsledek simulace vybraný pomocí posuvníku. Lze nastavit tři různé druhy zobrazení výsledku.

Na obrázku 5.1 je zobrazení částic, kde každá buňka je zobrazená pomocí jednoho pixelu. Barva pixelu je určená podle počtu částic uvnitř dané buňky. Pokud buňka neobsahuje žádné částice, potom je bílá. V případě, že obsahuje maximální počet částic, určený modelem simulace, potom je buňka černá. Zbývající stavy jsou pokryty různými odstíny šedé barvy, kde tmavší barva určuje větší počet částic.



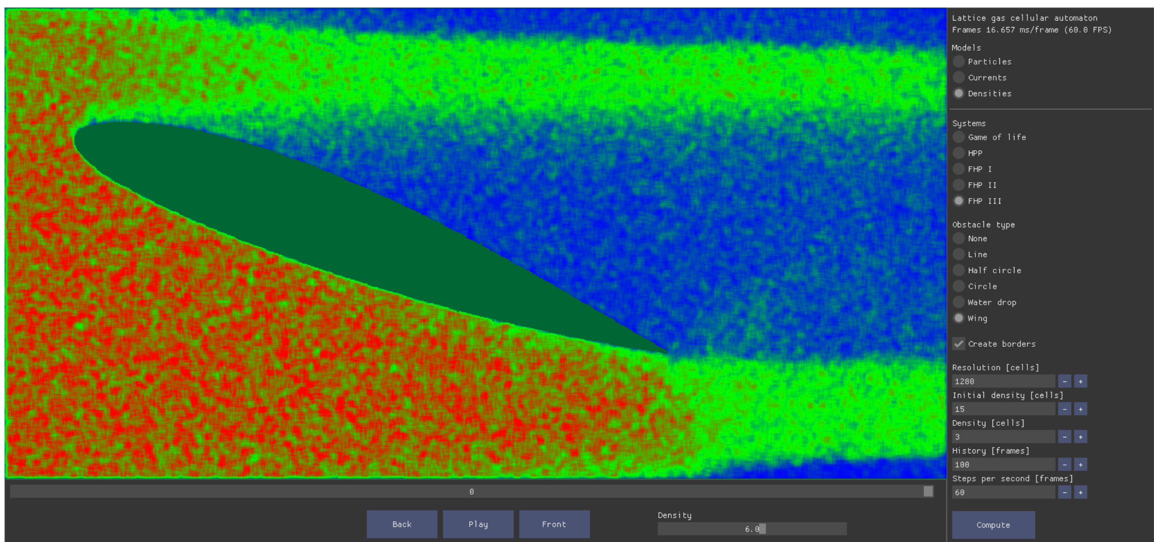
Obrázek 5.1: Výsledný simulátor se zobrazením jednotlivých částic.

Obrázek 5.2 obsahuje grafické znázornění proudění pomocí šipek. Směr šipky je určený pomocí výsledného momentu pole buněk o velikosti 8x8. Moment jedné buňky je vypočítaný z momentů částic uvnitř dané buňky. Barva šipky je určená velikostí výsledného momentu, kde tmavší barva značí větší velikost.



Obrázek 5.2: Výsledný simulátor s aproximovaným zobrazením pohybu částic pomocí šipek.

Poslední možnost zobrazení je na obrázku 5.3, kde je zobrazena hustota zastoupení částic v dané buňce a jejím okolí. Pro každou buňku se vypočítá její barva pomocí počtu částic uvnitř dané buňky a počtu částic uvnitř buněk v nejbližším okolí. Výsledná buňka může mít barvu v odstínu modré, kdy má nejmenší počet částic nebo zelené, kdy má průměrný počet částic a nebo červené s největším počtem částic. Počet částic vytváří plynulý přechod mezi barvami.



Obrázek 5.3: Výsledný simulátor se zobrazením hustoty částic.

Kapitola 6

Testování aplikace

V první části této kapitoly jsou popsány jednotlivé druhy testování, které byly použity pro potvrzení funkčnosti aplikace a taky jsou uvedeny výsledky testů. Pro testování aplikace byly použity jednotkové testy implementované v knihovně *GTest* od společnosti *Google*. Bylo ověřeno, jaké je procentuální pokrytí kódu testovací sadou. Více informací ohledně knihovny je napsáno v kapitole 3.6. V druhé části kapitoly je porovnána výkonnost WebAssembleru s alternativními řešeními.

6.1 Jednotkové testy

Jednotkové testy [30] jsou jedním ze základních způsobů pro testování. Tento druh testování byl zvolen, protože dokáže odhalit hodně nízkoúrovňových chyb a je velmi jednoduchý pro použití. Jednotkové testy mají výhodu, že z jejich spuštění se zjistí procentuální pokrytí zdrojového kódu, to znamená, jaké řádky kódu byly spuštěny a jaké cesty v podmínkách byly pokryty. V ideálním případě jsou jednotlivé testy od sebe vzájemně izolované.

Ve zdrojovém adresáři existují soubory s koncovkou **.test.cpp*, které obsahují všechny implementované jednotkové testy. Tyto soubory byly použity i na vygenerování pokrytí kódu. Každý soubor s testovací sadou odpovídá jinému zdrojovému souboru s implementací, kde oba soubory mají stejné jméno, například zdrojový soubor *gui.cpp*, obsahující implementaci grafického rozhraní, je otestovaný v souboru *gui.test.cpp*, který je ve stejném adresáři.

Výsledek testování

Testy byly spuštěny na linuxové distribuci *openSUSE Leap 15.0*. Výsledek testů lze vidět na obrázku 6.1, který znázorňuje prvních 5 a posledních 8 testů. Na spodní straně obrázku lze vidět, že žádný z testů neselhal. Pro vygenerování výstupu testování byla použita část *CTest* z nástrojové sady *CMake*.

Pokrytí kódu jednotkovými testy

Pokrytí kódu [22] je jeden z možných testovacích způsobů pro vizuální reprezentaci kvality a rozsáhlosti sady jednotkových testů. Výsledek lze vygenerovat jako jednoduchou HTML stránku, která graficky znázorňuje, které části kódu byly provedené, které nebyly a pro podmínky se zobrazí i jaké větve rozhodnutí byly pokryté. Výsledek pro aplikaci je na obrázku 6.2.

```
[14:56] ~/P/s/b/caflow-build (master ⚡ =) ninja test
[0/1] Running tests...
Test project /home/tyrkaen/Projects/school_projects/bachelor/caflow-build
        Start   1: Gui.build_box
  1/122 Test   #1: Gui.build_box ....................   Passed    0.02 sec
        Start   2: Gui.build_raw
  2/122 Test   #2: Gui.build_raw ....................   Passed    0.17 sec
        Start   3: Gui.set_size
  3/122 Test   #3: Gui.set_size .....................   Passed    0.55 sec
        Start   4: Gui.on_start_compute
  4/122 Test   #4: Gui.on_start_compute .............   Passed    0.02 sec
        Start   5: Gui.on_change_system
  5/122 Test   #5: Gui.on_change_system .............   Passed    0.12 sec
        Start 115: Position.assign
115/122 Test #115: Position.assign ..................   Passed    0.02 sec
        Start 116: Position.construct
116/122 Test #116: Position.construct ...............   Passed    0.02 sec
        Start 117: Error.create
117/122 Test #117: Error.create .....................   Passed    0.02 sec
        Start 118: Error.set
118/122 Test #118: Error.set ........................   Passed    0.02 sec
        Start 119: Error.get
119/122 Test #119: Error.get ........................   Passed    0.04 sec
        Start 120: Smart.create_box
120/122 Test #120: Smart.create_box .................   Passed    0.03 sec
        Start 121: Smart.create_raw
121/122 Test #121: Smart.create_raw .................   Passed    0.02 sec
        Start 122: Lgca.create
122/122 Test #122: Lgca.create ......................   Passed    0.01 sec

100% tests passed, 0 tests failed out of 122

Total Test time (real) =  10.83 sec
```

Obrázek 6.1: Výsledek spuštěných testů pomoci *CTest*. Zobrazeno je prvních 5 a posledních 8 výsledků.

LCOV - code coverage report

			Hit	Total	Coverage
Current view:	top level				
Test:	all-merged.info	**Lines:**	1027	1150	89.3 %
Date:	2019-04-25 15:10:23	**Functions:**	106	118	89.8 %

Directory	Line Coverage			Functions	
common		90.3 %	28 / 31	100.0 %	10 / 10
gui		84.0 %	499 / 594	79.2 %	42 / 53
lgca		87.4 %	104 / 119	94.1 %	16 / 17
lgca/system		97.5 %	396 / 406	100.0 %	38 / 38

Generated by: LCOV version 1.0

Obrázek 6.2: Výsledek pokrytí kódu pomocí jednotkových testů.

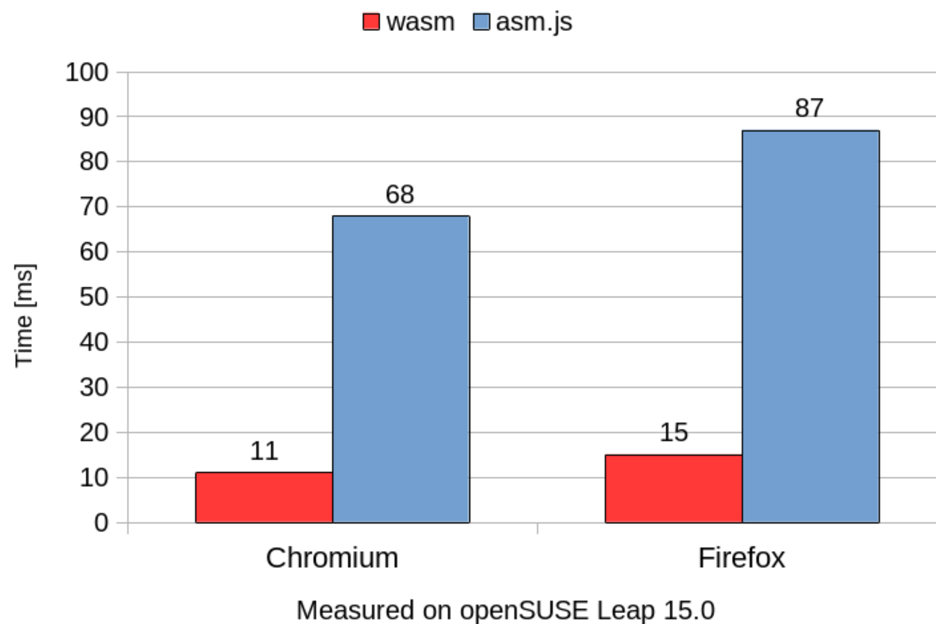
6.2 Porovnání výkonnosti WebAssembleru

V této části je porovnána výkonnost *WebAssembleru* s konkurenčními formáty. Jelikož vzniká nový standard *wasi*, pro běh nativních aplikací bez potřeby prohlížeče, je vhodné porovnat výkonnost *wasm* formátu s `x86_64` architekturou. Vhodnější by bylo porovnat 32bit verzi architektury, ale tento druh architektury se na stolních počítačích vyskytuje méně, než porovnávaný druh a *wasm* standard přejde v blízké budoucnosti na 64bit verzi, proto jsou porovnány nejnovější a nejpoužívanější verze architektur.

Byly porovnány dvě části a to porovnání rychlosti načítání zdrojového souboru a rychlost běhu aplikace. Obě části byli testovány na operačním systému openSUSE Leap 15.0. Testování probíhalo na počítačové sestavě s procesorem `Intel® Core™ i7-8086K` a grafickou kartou `NVIDIA® GeForce® GTX 1060 6GB`.

Porovnání rychlosti spouštění aplikace

Tato část porovnává rychlost načítání zdrojového souboru při otevření stránky [34]. Jelikož binární soubor *wasm* je potřeba pouze stáhnout, je rychlost načtení mnohem větší než pro asm.js, kde je potřeba vytvořit abstraktní syntaktický strom a mnoho dalšího. Výsledek je na obrázku 6.3. Lze vidět, že rozdíl v rychlosti je velmi výrazný. Největší je v prohlížeči *Chromium* od firmy *Google*. Tento prohlížeč slouží jako základ pro ostatní konkurenční prohlížeče, jako jsou například Edge, Brave, Opera a další, proto jejich výkonnost nebyla testována.



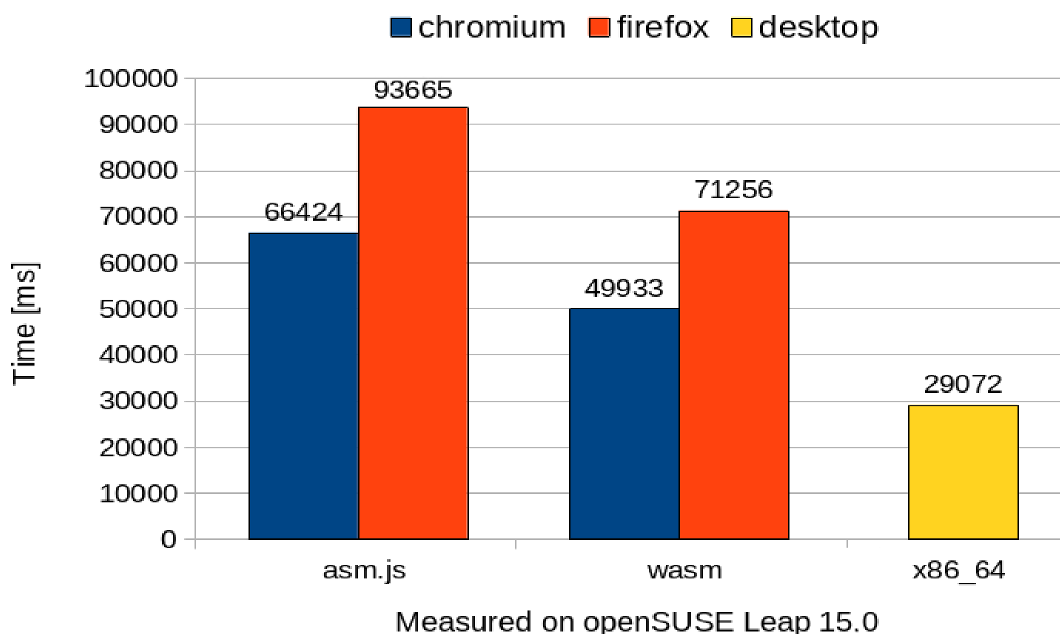
Obrázek 6.3: Porovnání rychlosti načítání *asm.js* a *wasm* formátu v prohlížečích Chromium a Firefox.

Porovnání rychlosti běhu programu

Druhou částí je porovnání rychlosti provádění programu. Porovnán byl desktop s formáty asm.js, wasm ve webových prohlížečích Firefox a Chrome. Testována byla rychlost výpočtu a vykreslení 1000 snímků s rozlišením 1920x960 buněk simulace. Počáteční hustota částic simulace byla nastavena na 2 a hustota generovaných částic byla nastavená na 3. Zobrazovaný režim při testování byl nastaven na zobrazení hustoty částic, který je nejsložitější na výpočet při vykreslení. Jelikož *wasm* je binární soubor, je už optimalizovaný při překladu, oproti tomu *asm.js* je stále JavaScriptový zdrojový soubor, proto optimalizace se provádí až za běhu aplikace.

Výsledek je na obrázku 6.4, kde nižší hodnota znamená rychlejší výpočet simulace. Lze vidět, že rozdíl výkonnosti je opravdu výrazný. Rychlost výpočtu simulace byla na desktopu průměrně o 50% větší než v wasm formátu, který byl o 24% rychlejší než asm.js formát. Rozdíl v rychlosti mezi prohlížeči byl znatelný. Velikost rozdílu rychlosti lze hlavně vidět ve snímkové frekvenci jednotlivých formátů.

Nejvyšší snímkovou frekvenci měl desktop, která byla `32 FPS`. Prohlížeč Chromium měl snímkovou frekvenci `20 FPS` pro formát wasm a `16 FPS` pro asm.js. Nejpomalejší vykreslování bylo v prohlížeči Firefox a to `13 FPS` pro wasm a `11 FPS` pro asm.js. Testováno bylo i zapnutí experimentální funkce *WebRender* [13] v prohlížeči Firefox, která by měla zvýšit rychlost vykreslování. Rozdíl v rychlosti byl téměř neznatelný, proto není zde uveden.



Obrázek 6.4: Porovnání rychlosti provádění *asm.js*, *wasm* formátu a desktopu.

Kapitola 7

Závěr

V této práci bylo cílem vytvořit aplikaci pro simulaci proudění kapalin pomocí LGCA s modely HPP, FHP-I, FHP-II a FHP-III. Aplikace navíc obsahuje Hru života (Game of Life), protože poskytuje zajímavé grafické znázornění celulárních automatů. Součástí programu je 5 modelů pro demonstraci výsledku simulace. Program obsahuje grafické rozhraní pro ovládání simulace a tři různé možnosti zobrazení výsledků. První možnost je zobrazení jednotlivých částic buněk, kde lze vidět aplikovaná pravidla simulace, druhá možnost je zobrazení šipek značících průměrný směr proudění částic v dané oblasti a poslední možnost je zobrazení jednotlivých buněk, kde barva každé buňky je vypočítána z průměru momentů částic dané buňky a osmi okolních. Kvůli rychlosti je mřížka simulace zobrazována pomocí OpenGL (přesněji WebGL 2.0) a nebyla použita žádná pomocná knihovna. Implementace je v jazyku C++ a jsou použity knihovny ImGui, CMake a GTest. Aplikaci lze přeložit pro desktop nebo WebAssembly, kde je potřeba použít nástroj Emscripten. Správná funkčnost programu byla testována pomocí knihovny GTest, byly provedeny jednotkové testy a pokrytí kódu. Pro jednoduchost nasazení aplikace je použit nástroj Docker.

Součástí práce jsou výkonnostní testy, kde byl porovnán formát wasm, asm.js a desktop. Byla porovnána rychlost načítání aplikace při otevření webové stránky, kde načtení wasm formátu je několikrát rychlejší než doba načtení asm.js. Druhý porovnávaný faktor byla rychlost provádění, kde wasm je průměrně o 24% rychlejší než asm.js, ale v průměru o 50% pomalejší než desktop.

Jednou z možností modifikace aplikace je zrychlení simulace, kterého lze dosáhnout paralelním výpočtem dalšího stavu. Pro jednoduchost implementace by šlo použít knihovnu OpenMP. Výpočty jsou nyní prováděny v jednom vláknu, protože WebAssembly v některých webových prohlížečích ještě nepodporuje více vláken. Další možností rozšíření funkcionality aplikace je implementace Lattice Boltzmann metody pro simulaci tekutin. Simulace by potom více odpovídala skutečnému chování tekutiny, ale jelikož práce pojednává o LGCA modelech nebyla tato metoda implementována. Jelikož je *WebAssembly* pouze *32bit* standard, je aplikace omezená s pamětí a není možné pracovat s více jak 2GiB [21] (Číslo je rozdílné pro jednotlivé webové prohlížeče), proto po vytvoření standardu s *64bit* adresací paměti by bylo vhodné přeložit aplikaci pro tento formát. Vyjmenované nedostatky nijak nebrání použití aplikace a je připravená na použití pro demonstraci funkčnosti celulárních automatů pro výukové účely.

Literatura

[1] Design Patterns - Builder Pattern. [Online; navštíveno 8.4.2019].
URL https://www.tutorialspoint.com/design_pattern/builder_pattern.htm

[2] EmSDK - Download and install. [Online; navštíveno 4.4.2019].
URL https://emscripten.org/docs/getting_started/downloads.html

[3] Features to add after the MVP. [Online; navštíveno 18.4.2019].
URL https://webassembly.org/docs/future-features

[4] The GTK project. [Online; navštíveno 4.4.2019].
URL https://www.gtk.org

[5] Margolus neighborhood definition. [Online; navštíveno 20.3.2019].
URL https://www.researchgate.net/figure/Margolus-neighborhood-definition_fig16_47629479

[6] Qt - Cross-platform software for embedded and desktop. [Online; navštíveno 4.4.2019].
URL https://www.qt.io

[7] WASI - The WebAssembly System Interface. [Online; navštíveno 8.4.2019].
URL https://wasi.dev

[8] Wasmtime: a WebAssembly Runtime. [Online; navštíveno 8.4.2019].
URL https://github.com/CraneStation/wasmtime

[9] WebAssembly Migration Guide. [Online; navštíveno 18.4.2019].
URL https://developer.chrome.com/native-client/migration

[10] What is a Container? - A standardized unit of software. [Online; navštíveno 4.4.2019].
URL https://www.docker.com/resources/what-container

[11] Chopard, B.; Droz, M.: *Cellular Automata Modeling of Physical Systems*. Cambridge University Press, 1998, ISBN 0-521-46168-5-hardback.

[12] Civil, G.: Googletest Primer. [Online; navštíveno 4.4.2019].
URL https://github.com/google/googletest/blob/master/googletest/docs/primer.md

[13] Clark, L.: The whole web at maximum FPS: How WebRender gets rid of jank. [Online; navštíveno 6.5.2019].
URL https://hacks.mozilla.org/2017/10/the-whole-web-at-maximum-fps-how-webrender-gets-rid-of-jank

[14] Codd, E. F.: Cellular Automata by E. F. Codd. [Online; navštíveno 4.3.2019].
URL https://archive.org/details/CellularAutomata_201706

[15] Editors, B.: John Horton Conway Biography. [Online; navštíveno 4.3.2019].
URL https://www.biography.com/people/john-horton-conway-111015

[16] English, T.: Understanding the Fourth Dimension From Our 3D Perspective. [Online; navštíveno 20.3.2019].
URL https://interestingengineering.com/understanding-fourth-dimension-3d-perspective

[17] Fabian, R.: Data-Oriented Design. [Online; navštíveno 8.4.2019].
URL http://www.dataorienteddesign.com/dodbook

[18] Fraile Rubio, C.; Hernandez Encinas, L.; White, S.; aj.: The Use of Linear Hybrid Cellular Automata as Pseudo Random Bit Generators in Cryptography. *Neural Parallel Scientific Comp.*, ročník 12, 06 2004: s. 175–192.

[19] Frisch, U.; Hasslacher, B.; Pomeau, Y.: *Lattice-Gas Automata for the Navier-Stokes Equation*. Physical Review Letters, 1986.

[20] Goldenfeld, N.; Kadanoff, L. P.: *Simple Lessons from Complexity*. Science, 1999.

[21] Group, W. C.: Implementation Limitations. 2017, [Online; navštíveno 14.4.2019].
URL https://webassembly.github.io/spec/core/appendix/implementation.html

[22] Hellesoy, A.; Wynne, M.: The Cucumber Book. [Online; navštíveno 13.4.2019].
URL https://www.oreilly.com/library/view/the-cucumber-book/9781941222911/f_0108.html

[23] Hénon, M.: Isometric collision rules for the four-dimensional FCHC lattice gas. *Complex Systems*, ročník 1, 01 1987.

[24] Inc, D.: Docker overview. [Online; navštíveno 8.4.2019].
URL https://docs.docker.com/engine/docker-overview

[25] Jackson, D.: WebGL 2.0 Specification. [Online; navštíveno 4.4.2019].
URL https://www.khronos.org/registry/webgl/specs/latest/2.0

[26] Kantor, I.: Polyfills. [Online; navštíveno 8.4.2019].
URL https://javascript.info/polyfills

[27] Karadžič, B.: Immediate Mode GUI is way to go for GameDev tools. [Online; navštíveno 1.4.2019].
URL https://gist.github.com/bkaradzic/853fd21a15542e0ec96f7268150f1b62

[28] Komppa, J.: Immediate mode GUIs. [Online; navštíveno 1.4.2019].
URL http://lambda-the-ultimate.org/node/4561

[29] Maeda, J.: Wolfram's Rule 30 and the Conus Textile. [Online; navštíveno 20.3.2019].
URL https://howtospeakmachine.com/2018/12/23/wolframs-rule-30-and-the-conus-textile

[30] Osherove, R.: *The Art of Unit Testing: with examples in C Second Edition*. Manning, 2013, ISBN 9781617290893.

[31] Poundstone, W.: John von Neumann American mathematician. [Online; navštíveno 3.3.2019].
URL https://www.britannica.com/biography/John-von-Neumann

[32] Serra, M.; Cattell, K.; Zhang, S.; aj.: One-Dimensional Linear Hybrid Cellular Automata: TheirSynthesis, Properties and Applications to Digital Circuits Testing. [Online; navštíveno 14.4.2019].
URL https://webhome.cs.uvic.ca/~mserra/AttachedFiles/CA_Tutorial.pdf

[33] Stephen, A.: Jun 28 Langton's Loops. [Online; navštíveno 4.3.2019].
URL
https://www.alaricstephen.com/main-featured/2017/6/27/langtons-loops

[34] Trivellato, M.: WebAssembly Load Times and Performance. [Online; navštíveno 14.4.2019].
URL https:
//blogs.unity3d.com/2018/09/17/webassembly-load-times-and-performance

[35] Tyler, T.: The Margolus neighbourhood. [Online; navštíveno 28.3.2019].
URL http://cell-auto.com/neighbourhood/margolus

[36] Tyler, T.: The Necker neighbourhood. [Online; navštíveno 28.3.2019].
URL http://cell-auto.com/neighbourhood/necker/index.html

[37] Vallarta, P.: Wolfram's Rules for Cellular Automata. [Online; navštíveno 17.3.2019].
URL https://rogerhillonline.com/CARules.aspx

[38] Weisstein, E. W.: MathWorld - Moore Neighborhood. [Online; navštíveno 20.3.2019].
URL http://mathworld.wolfram.com/MooreNeighborhood.html

[39] Weisstein, E. W.: Rule 30. [Online; navštíveno 17.3.2019].
URL http://mathworld.wolfram.com/Rule30.html

[40] Weisstein, E. W.: von Neumann Neighborhood. [Online; navštíveno 20.3.2019].
URL http://mathworld.wolfram.com/vonNeumannNeighborhood.html

[41] Wolf-Gladrow, D. A.: *Lattice-Gas Cellular Automata and Lattice Boltzmann Models - An Introduction*. Springer, 2005.

[42] Wolfram, S.: Cellular Automata. [Online; navštíveno 14.4.2019].
URL https://www.stephenwolfram.com/publications/cellular-automata-complexity/pdfs/cellular-automata.pdf

[43] Wolfram, S.: Wolfram Alpha. [Online; navštíveno 4.3.2019].
URL https://www.wolframalpha.com

[44] Zlatkov, A.: How JavaScript works: A comparison with WebAssembly + why in certain cases it's better to use it over JavaScript. [Online; navštíveno 1.4.2019].
URL
https://blog.sessionstack.com/how-javascript-works-a-comparison-with-webassembly-why-in-certain-cases-its-better-to-use-it-d80945172d79

Příloha A

Obsah CD

Data uložená na kompaktním disku jsou rozdělena do následujících skupin:

Zdrojové soubory

Zdrojové soubory jsou uloženy ve složce `sources`. Složka dále obsahuje externí závislosti a soubory `Dockerfile`, `index.html`, `favicon.ico`. Hierarchie složky je popsána v kapitole 5.2.

Text práce

Zdrojové soubory obsahující text práce jsou uloženy ve složce `text`. Součástí složky je soubor caflow.pdf, který obsahuje už přeložený text práce do formátu pdf.

Obraz aplikace

Obraz aplikace pro nástroj Docker je uložený v kořenovém adresáři pod názvem `caflow.tar`. Způsob spuštění aplikace pomocí nástroje docker nebo podman je popsaný v kapitole 5.3.

Přeložená aplikace

Přeložená aplikace je ve složce `binaries`. Složka obsahuje podsložky pro jednotlivé cíle překladu, kde v každé podsložce jsou závislosti potřebné pro běh aplikace ve složce `assets`. Aplikace očekává, že tato složka bude při spuštění vedle binárního souboru, nebo v kořenovém adresáři serveru. Způsob překladu je popsán v kapitole 5.4. Jednotlivé podložky jsou:

- `wasm` – Přeložené soubory pro WebAssembly formát
- `asm_js` – Přeložené soubory pro asm.js formát
- `linux` – Binární soubor spustitelný v operačním systému Linux
- `windows` – Binární soubor spustitelný v operačním systému Windows