

Modulární simulátor buněčných automatů

Bc. Martin Machala

ZADÁNÍ DIPLOMOVÉ PRÁCE

(projektu, uměleckého díla, uměleckého výkonu)

Jméno a příjmení: Bc. Martin Machala
Osobní číslo: A23318
Studijní program: N0613A140022 Informační technologie
Specializace: Softwarové inženýrství
Forma studia: Prezenční
Téma práce: Modulární simulátor buněčných automatů
Téma práce anglicky: Modular Cellular Automata Simulator

Zásady pro vypracování

- Provedte rešerši v oblasti buněčných automatů a existujících řešení.
- Navrhněte architekturu modulárního simulátoru buněčných automatů.
- Vytvořte grafickou aplikaci s implementací buněčného simulátoru.
- Implementujte systém modulárních setů pravidel a buněk.
- Vytvořte ukázkové moduly pro buněčný simulátor.

Seznam doporučené literatury:

1. PECINOVSKÝ, Rudolf. Python: Kompletní příručka jazyka pro verzi 3.11. Grada, 2022. ISBN 978-80-271-3891-3.
2. FITZPATRICK, Martin. *Create GUI Applications with Python & Qt6 (PyQt6 Edition): The hands-on guide to making apps with Python*. 5th Edition. 2022. ISBN 9798829985554.
3. PECINOVSKÝ, Rudolf. Návrhové vzory: [33 vzorových postupů pro objektové programování]. Brno: Computer Press, 2007. ISBN 9788025115824.
4. AMOS, Martyn (ed.). Cellular computing. Series in systems biology. Oxford: Oxford University Press, 2004. ISBN 0195155408.
5. ILACHINSKI, Andrew. Cellular automata: a discrete universe. Singapore: World Scientific, 2001. ISBN 981238183X.
6. HADELER, Karl-Peter a MÜLLER, Johannes. *Cellular Automata: Analysis and Applications*. Springer, 2017. ISBN 978-3-319-53042-0.

Vedoucí diplomové práce: **prof. Ing. Zuzana Komínková Oplatková, Ph.D.**
Ústav informatiky a umělé inteligence

Datum zadání diplomové práce: **27. října 2024**
Termín odevzdání diplomové práce: **2. června 2025**



L.S.

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

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

Ve Zlíně dne 17. listopadu 2024

PROHLÁŠENÍ AUTORA DIPLOMOVÉ PRÁCE

Beru na vědomí, že

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

Prohlašuji, že

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

Ve Zlíně, dne

.....
podpis autora

ABSTRAKT

Tato práce se zabývá návrhem a implementací unikátního modulárního systému pro simulaci buněčných automatů. Velkým přínosem je zaměření na obecnost, snadnou rozšiřitelnost, otevřenost a jednoduchost použití. V rámci obecnosti je systém navržen takovým způsobem, že je schopný podporovat simulaci co nejvíce možných typů buněčných automatů, včetně zajištění podpory pro různé generalizace konceptu buněčných automatů. Rozšiřitelnost simulátoru uživatelé využijí zejména při návrhu a implementaci vlastní sady spustitelných modulů. Samozřejmostí je možnost stažení modulů, možnost spuštění těchto modulů na vlastním zařízení a možnost provádění úprav v těchto modulech. Způsob tvoření těchto modulů je jednoduchý a pochopitelný. Otevřenost systému je podpořena volně dostupným zdrojovým kódem, jeho srozumitelností a volbou vhodné licence pro umožnění volných úprav a rozšiřování uživateli. Jednoduchost použití se projevuje v přehledném a intuitivním uživatelském rozhraní. Práce je doplněna o ukázky fungujících modulů různé složitosti, implementovaných a spustitelných v prezentovaném simulátoru.

Klíčová slova: buněčné automaty, umělá inteligence, CA, AI, simulace, simulátor

ABSTRACT

This thesis deals with the design and implementation of a unique modular system used for simulation of cellular automata. Its major contribution is the focus on generality, easy extensibility, openness and ease of use. In regard to generality, the system is designed in such a way that it can support the simulation of as many types of cellular automata as possible, including the support for different generalisations of the cellular automata concept. The extensibility of the simulator is particularly useful for users in the process of designing and implementing their own executable modules. Of course, it is possible to download modules, to use them on a personal device and to make modifications to these modules. The method of creating these modules is simple and easy to understand. The openness of the system is supported by the freely available source code, its clarity and the choice of a suitable license to allow for free modification and extension of the simulator by its users. The system's ease of use is reflected in a clear and intuitive user interface. The thesis is complemented by a

set of example working modules of varying complexity, implemented and executable in the presented simulator.

Keywords: cellular automata, artificial intelligence, CA, AI, simulation, simulator

Především bych chtěl velice poděkovat vedoucí této práce, prof. Ing. Zuzaně Komínkové Oplatkové Ph.D., za její vstřícnost, odborné vedení a cenné konzultace v průběhu vypracovávání této práce. Dále bych chtěl poděkovat Niekui Goudswaardovi za jeho trpělivost a užitečné připomínky při mnohých diskusích ohledně této práce.

OBSAH

ÚVOD	15
I TEORETICKÁ ČÁST	16
1 BUNĚČNÉ AUTOMATY	17
1.1 KOMPLEXNÍ SYSTÉMY	17
1.2 HISTORIE	17
1.3 PROSTOROVÉ KONFIGURACE	18
1.3.1 Dimenzionalita	18
1.3.2 Tvar buněk	18
1.3.3 Velikost prostoru	19
1.3.4 Topologie prostoru	19
1.4 SOUSEDSTVÍ	19
1.5 VÝBĚR ZE SAD PRAVIDEL	19
1.5.1 Conwayova hra života	19
1.5.2 Langtonův mravenec	20
1.5.3 WireWorld	21
1.6 GENERALIZACE	21
1.6.1 Asynchronicita	21
1.6.2 Spojité buněčné automaty	22
1.6.3 Využití náhodnosti	22
1.6.4 Rozdílná pravidla pro různé buňky	22
1.6.5 Pohyblivé buňky	23
1.6.6 Dynamická struktura mřížky	23
1.7 REÁLNÁ VYUŽITÍ	23
2 SIMULÁTORY BUNĚČNÝCH AUTOMATŮ	24
2.1 MIREK'S CELEBRATION	24
2.2 GOLLY	24
2.3 READY	25
2.4 CELLAB	25

2.5	SHRnutí	26
II	PRAKTICKÁ ČÁST.....	27
3	POŽADAVKY NA SYSTÉM.....	28
3.1	SIMULACE BUNĚČNÝCH AUTOMATŮ	28
3.2	MODULARITA	28
3.3	OBECNOST	28
3.4	KOMPATIBILITA MODULŮ	29
3.5	UŽIVATELSKÉ ROZHRANÍ.....	29
3.6	JEDNODUCHOST.....	29
3.7	MULTIPLATFORMNOST	29
3.8	OTEVŘENOST	29
4	NÁVRH ARCHITEKTURY SYSTÉMU.....	30
4.1	ROZDĚLENÍ POVINNOSTÍ V SIMULACI.....	30
4.2	DEFINICE HIERARCHIE MODULŮ.....	30
4.3	STRUKTURA BUNĚK.....	31
4.4	INSTANCOVÁNÍ BUNĚK	32
4.5	SPRÁVA BUNĚK	33
4.6	STRUKTURA PROSTŘEDÍ.....	34
4.7	KOMPATIBILITA BUNĚK A PROSTŘEDÍ.....	35
4.8	SPRÁVA PROSTŘEDÍ	35
4.9	STRUKTURA VYKRESLOVAČE	36
4.10	KOMPATIBILITA PROSTŘEDÍ A VYKRESLOVAČŮ	37
4.11	SPRÁVA VYKRESLOVAČŮ	37
4.12	SPRÁVA MODULŮ	37
4.13	UŽIVATELSKÉ ROZHRANÍ.....	38
4.14	EXPORTNÍ FUNKCE.....	40
5	VÝBĚR TECHNOLOGIÍ PRO VYTVOŘENÍ SIMULÁTORU	41
5.1	PROGRAMOVACÍ JAZYK	41

5.2	KNIHOVNY	41
5.3	ZVEŘEJNĚNÍ KÓDU	42
6	IMPLEMENTACE NÁVRHU	43
6.1	METADATA.....	43
6.2	BUŇKA	45
6.3	PROSTŘEDÍ	46
6.4	VYKRESLOVAČ	49
6.5	SPOUŠTĚČ BUNĚK	51
6.6	HLEDAČ MODULŮ	53
6.7	UŽIVATELSKÉ ROZHRANÍ.....	57
6.7.1	WelcomeScreen.....	57
6.7.2	MainScreen	62
6.7.3	SimulationLabel	68
6.8	EXPORTNÍ FUNKCE	68
6.9	MAIN	69
7	PŘÍKLADY MODULŮ	71
7.1	VYKRESLOVAČ	71
7.1.1	Simple2DRenderer.....	71
7.2	PROSTŘEDÍ	73
7.2.1	Simple2DEnvironment.....	73
7.2.2	Energy2DEnvironment.....	75
7.3	BALÍČKY BUNĚK.....	78
7.3.1	GoL	78
7.3.2	Pattern	79
7.3.3	RandomWalk.....	80
7.3.4	Virus	81
7.4	WALLS	82
7.4.1	WireWorld	82
7.4.2	GeneticCell	83

ZÁVĚR.....	86
SEZNAM POUŽITÉ LITERATURY	89
SEZNAM OBRÁZKŮ	91
SEZNAM VÝPISŮ	92
SEZNAM POUŽITÝCH SYMBOLŮ A ZKRATEK	93

ÚVOD

Buněčné automaty jsou užitečným nástrojem pro simulaci různých druhů komplexních systémů. Mohou být využity například pro simulaci fyzikálních systémů, ekosystémů či výpočetních systémů.

Existující simulátory buněčných automatů jsou převážně zaměřené na simulaci omezeného počtu druhů buněčných automatů. Často také podporují simulaci pouze minima možných generalizací tohoto konceptu, což výrazně omezuje možná využití těchto simulátorů.

Cílem této práce je navrhnout a vytvořit unikátní simulátor buněčných automatů, který je zaměřen na obecnost, snadnou rozšiřitelnost, otevřenost a jednoduchost použití.

V rámci obecnosti by měl být systém navržen takovým způsobem, aby byl schopný podporovat simulaci co nejvíce možných typů buněčných automatů, včetně zajištění podpory pro různé generalizace tohoto konceptu.

Rozšiřitelnost by měla uživatelům poskytnout možnost návrhu a implementace vlastních modulů. Také by jim měla umožnit využívání modulů stažených z internetu, náhled do jejich implementace a možnost tuto implementaci pozměnit ve vlastních verzích těchto modulů.

Otevřenost simulátoru by se neměla vztahovat pouze na jeho moduly, ale také na zdrojový kód samotného simulátoru. Tento kód by měl být volně dostupný k nahlížení a úpravám. K tomuto účelu je potřeba, aby byl publikován pod vhodnou licencí. V rámci otevřenosti by zdrojový kód simulátoru měl být přehledně napsaný.

Práce je také zaměřena na tvorbu sady ukázkových modulů pro tento simulátor. Uživatelům tak poskytuje náhled na různé druhy modulů a na specifických druhů buněčných automatů, spolu s možností detailního prozkoumání jejich implementací. Tyto implementace poté mohou využít při vytváření svých vlastních sad modulů.

I TEORETICKÁ ČÁST

1 BUNĚČNÉ AUTOMATY

Buněčné automaty jsou systémy reprezentovány kolekcí barevných políček, často nazývaných „buňky“. Tyto buňky existují v N-dimenzionálním prostoru, který je definován jako mřížka specifického tvaru [1].

Chování specifických konfigurací buněčných automatů je definováno sadou pravidel, která určují, jakým způsobem bude docházet k přechodům mezi různými stavy buněk [1].

Systém pracuje v diskrétních časových krocích. Při každém kroku systému je nejdříve vyhodnocován jeho současný stav, a podle něj je za pomoci definovaných pravidel vypočítán následující stav. Počet proveditelných kroků není ničím omezen [1, 2].

Buněčné automaty jsou nejjednodušší reprezentací komplexních systémů. I díky zdánlivě jednoduché sadě pravidel může buněčný automat vykazovat velmi komplexní chování [2]. Jsou multidisciplinárním bodem zájmu, propojujícím elementy z informatiky, biologie, matematiky i inženýrství [3].

1.1 Komplexní systémy

Komplexní systémy představují systémy, které obsahují velké množství částí, které mezi sebou nějakým způsobem interagují, a tím se ovlivňují [2]. Jejich komplexita způsobuje to, že není jednoduché tyto systémy popisovat a modelovat. Chování komplexních systémů totiž může být komplikované a těžce předvídatelné [4]. Tyto komplexní vlastnosti mohou vytvářet nové a nečekané chování, které samostatné části systému nevykazují, takzvané „emergentní chování“ [2].

Komplexní systémy jsou v reálném světě zastoupeny v mnoha podobách. Mezi reprezentanty komplexních systémů mohou například patřit neurony tvořící mozek, počasí, živé organismy tvořené z mnoha buněk, chemické reakce s mnoha komponenty, nebo také systémy tvořené lidmi, jako například politické či sociologické systémy [2, 4].

1.2 Historie

S nápadem buněčných automatů jako první přišel John von Neumann v roce 1948 jako jednoduchý způsob modelování biologické evoluce [2]. Na návrh Stanisława Ulama začal von Neumann reprezentovat buněčné automaty diskrétně, což eventuálně vedlo k von Neuman-

nově vědecké práci popisující dvoudimenzionální buněčný automat schopný seberekplikace. Buňky tohoto automatu nabývaly 29 různých stavů [2, 5]. Tento set pravidel buněčného automatu byl prvním diskretním paralelním výpočetním modelem. Bylo prokázáno, že tento systém byl schopen obecné výpočetní práce, tj. byl „turing complete“ [2].

Dalším významným přispěvatelem k oboru buněčných automatů byl John Horton Conway, který v roce 1970 přišel s vlastní sadou pravidel zvanou „Game of Life“ (hra života). Tato sada pravidel je jedním z nejjednodušších buněčných modelů schopných obecných výpočtů [2]. Hra života se díky kombinaci jednoduchosti svých pravidel se složitostí a ikonicitou svých výstupů stala populárním reprezentantem oboru buněčných automatů [5].

1.3 Prostorové konfigurace

Buněčné automaty mohou existovat v různých prostorových konfiguracích. U těchto konfigurací záleží na vybraném počtu prostorových dimenzí, a také na rozložení buněk v definovaném prostoru. Další důležitou částí je určení velikosti prostoru a jeho topologických vlastností [5].

1.3.1 Dimenzionalita

V ohledu dimenzionality patří mezi nejčastěji používané konfigurace ty, které jsou jednodimenzionální, dvoudimenzionální či trojdimenzionální [2]. Nic ale nebrání využití vyššího počtu dimenzí, mimo možných problémů s vizualizací těchto mnohodimenzionálních buněčných automatů.

1.3.2 Tvar buněk

Nejčastější konfigurace rozložení buněk v prostoru jsou ty, které jsou složeny z kolekce identických opakujících se útvarů, které do sebe zapadají [2]. V případě jednodimenzionálních buněčných automatů to může být řada na sebe navazujících čar, případně jedna řada čtverců. Díky tomu, že je potřeba pouze jedna dimenze, je možné společně se současným stavem jednodimenzionálního automatu jednoduše vizualizovat i stavy předchozí, například skládáním výstupů pod sebe vzestupně podle počtu časových kroků [1, 2]. V případě dvoudimenzionálních buněčných automatů se často pro reprezentaci buněk využívají opakující se vzory tvořené čtverci, trojúhelníky nebo šestiúhelníky [2, 5]. U trojdimenzionálních buněčných automatů se poté nejčastěji používají krychle [2]. Stejně jako u počtu dimenzí není prostorové

rozložení buněk omezeno. Buňky nemusí mít stejný tvar, a ani na sebe nemusejí vizuálně navazovat [5].

1.3.3 Velikost prostoru

Prostor, ve kterém buněčné automaty pracují, má určitou definovanou velikost. Tato velikost může být nekonečná, nebo omezená na specifické rozměry [5].

1.3.4 Topologie prostoru

Tvar mřížky, ve které se buněčné automaty nacházejí, nemusí být lineární. Buněčné automaty mohou fungovat v neeuklidovském prostoru. Prostor na sebe také může různými způsoby navazovat, dvoudimenzionální prostor může například představovat tvar trubice, pokud konec jedné osy navazuje znovu na svůj začátek [2, 5].

1.4 Sousedství

Sousedství definuje, které buňky mohou mezi sebou interagovat [5]. U jednodimenzionálních automatů se nejčastěji dívá na přímé sousedy nalevo a napravo od buňky [6]. U dvoudimenzionálních automatů využívajících čtvercových buněk jsou dva často používané typy sousedství. Prvním tímto typem je Moorovo sousedství, ve kterém spolu sousedí buňky, které jsou vedle sebe vertikálně, horizontálně či diagonálně. Druhým typem je sousedství von Neumannovo, ve kterém na rozdíl od Moorova sousedství buňky diagonálně vedle sebe spolu nesousedí [5]. Tato dvě sousedství je možné generalizovat i pro trojdimenzionální buněčné automaty [7]. Jako u mnoha dalších aspektů buněčných automatů jsou možnosti sousedství neomezené a závislé jenom na implementaci. Buňky mohou například sousedit jenom diagonálně, nebo mohou „dohlédnout“ dále, a sousedit tak s buňkami, které jsou vzdálenější.

1.5 Výběr ze sad pravidel

1.5.1 Conwayova hra života

Conwayova hra života (Conway's Game of Life), pojmenována podle svého autora, Johna Hortona Conwaye, patří mezi nejpopulárnější reprezentanty buněčných automatů [5]. Buňky v tomto systému mohou nabývat dvou různých stavů. Buňka může být buďto „mrtvá“ nebo „živá“. Každý cyklus buňky kontrolují, kolik mají žijících sousedů v Moorově sousedství. Pokud má mrtvá buňka přesně tři živé sousedy, stane se z ní živá buňka. Pokud má živá

buňka méně než dva sousedy nebo více než tři sousedy, je nahrazena mrtvou buňkou. V jiných případech se stav živých a mrtvých buněk nemění [2].

I s těmito jednoduchými pravidly je hra života schopná vytvářet komplikované struktury, které se postupně vyvíjejí. Bylo zdokumentováno mnoho různých struktur, které se v simulacích hry života vyskytují. Mezi tyto struktury patří například „kluzáky“ (glider), které se každé čtyři cykly posunou o jednu buňku diagonálně. Další častou strukturou jsou oscilátory, které cyklicky prochází několika stavy s tím, že se nehne ze svého místa. Velmi častou strukturou ve hře života jsou také „stabilní“ struktury. To jsou struktury, které permanentně zůstávají v jedné konfiguraci, bez jakékoli změny mezi časovými kroky [2].

Jak již bylo zmíněno v předchozí sekci, hra života je schopna provádět obecné výpočty. Je „Turing complete“ [2].

1.5.2 Langtonův mravenec

Langtonův mravenec (Langton's Ant) je typ buněčného automatu který je, podobně jako hra života, schopný provádět obecné výpočty. Byl pojmenován podle svého autora, Christophera Langtona [8].

Na rozdíl od hry života využívá Langtonův mravenec von Neumannova sousedství [8].

V tomto systému existují buňky, které reprezentují „mravence“. Tito mravenci se každý časový krok posunou o jednu buňku. Směr tohoto posunu je určený předchozím směrem kroku a stavem, ve kterém je buňka, která reprezentuje mravence. Informace o tomto stavu v buňce zůstávají i po přesunu mravence. Buňky, které mravence nerepresentují, mohou nabývat dvou různých stavů. První stav mravenci říká, že další krok odbočí doleva. Druhý zase, že mravenec odbočí další krok doprava. Pokud buňka mravence reprezentuje, stav buňky také musí reprezentovat následující směr posunu mravence. Buňka tak může nabývat osmi různých stavů (čtyři možnosti pro směr posunu krát dvě možnosti odbočení). Po vstupu mravence na nové pole se stav odbočení „otáčí“ na druhou stranu [8].

Existuje domněnka která říká, že všechny konfigurace buněk Langtonova mravence s jedním mravencem, které obsahují konečný počet jednoho typu buňky, skončí tak, že mravenec začne cyklicky stavět „dálnici“, tj. mravenec se začne pohybovat pryč od středu simulace, přičemž za sebou tvoří vzor z buněk [8].

1.5.3 WireWorld

WireWorld je buněčný automat, který pomocí svých pravidel simuluje jednoduché „elektrické“ obvody [9].

Byl navržen Brianem Silvermanem v roce 1987 [9].

Stejně jako hra života využívá Moorova sousedství [9].

WireWorld pracuje se čtyřmi různými stavy buněk. Buňky v prvním stavu jsou neaktivní. V čase se nijak nemění. Slouží pouze jako pozadí simulace. Buňky v druhém stavu reprezentují „vodiče“. Vytvořením řady těchto vodičů se dají skládat obvody. Buňka ve třetím stavu reprezentuje přední část „elektronu“. Pokud vodič sousedí s jednou nebo se dvěma předními částmi elektronu, stává se z něj přední část elektronu. Čtvrtým a posledním stavem buněk je stav, kde buňka reprezentuje zadní část elektronu. Pokud je buňka ve stavu přední části elektronu, v dalším časovém kroku se z něj stane zadní část elektronu. Pokud je buňka ve stavu zadní části elektronu, stane se z ní vodič [9].

Díky tomuto systému se signály tvořené elektrony mohou ve vodičích propagovat. Kvůli tomu, že je šíření signálu omezené počtem předních částí elektronu vedle vodiče, je možné vytvářet komplikované obvody. Ve WireWorldu je pomocí vodičů možné vytvářet obvody napodobující funkcionalitu logických hradel, jako je třeba AND, OR nebo XOR. Ve WireWorldu je tedy, stejně jako u předchozích ukázek, možné provádět obecné výpočty [9].

1.6 Generalizace

Koncept buněčných automatů jde do značné míry generalizovat. [2] uvádí několik modelů pro tyto generalizace.

1.6.1 Asynchronicita

Buněčné automaty jsou ve své základní podobě „synchronní“, to znamená, že přechody mezi stavy jsou prováděny pro celou mřížku zároveň. Generalizací tohoto konceptu je provádění přechodů mezi stavy „asynchronně“, tj. různě pro různé buňky [2].

Existuje mnoho způsobů, jak asynchronicity docílit. Mezi příklady asynchronních aktualizací patří třeba aktualizace v náhodném pořadí, aktualizace zleva doprava a shora dolů nebo aktualizace buněk na lichých a sudých pozicích (dle souřadnic) zvlášť [10].

[11] udává, že synchronní buněčné automaty lze plně simulovat v asynchronním prostředí.

1.6.2 Spojité buněčné automaty

U základních buněčných automatů se pracuje s diskrétními hodnotami. Mezi tyto diskrétní hodnoty patří simulační čas, prostor a stavy buněk. Generalizací je potom implementace, která podporuje spojité hodnoty u těchto proměnných [2].

Spojité stav znamená, že se buňky nemusí nacházet pouze v omezeném počtu předdefinovaných stavů, ale prostor těchto stavů je spojitý. Ve hře života buňky například existují ve dvou možných stavech, reprezentovatelnými hodnotami 1 a -1. Ve spojitě implementaci by se buňky mohly nacházet na kterékoliv místě mezi těmito hodnotami [2, 12].

U spojitěho času se pracuje s hodnotou která určuje, kolik času uběhlo mezi simulačními kroky. Čím blíže je tato hodnota k nule, tím blíže je systém k opravdovému spojitému času. Časová délka kroku se využívá jako proměnná při výpočtu stavů buněk. Většinou se vy počítaná hodnota změny stavu násobí velikostí časového kroku. Tím se aproximuje průběh simulace ve spojitém čase [12].

Ve spojitém prostoru se namísto klasického modelu sousedství používá přístup, který do voluje různým buňkám v různých vzdálenostech ovlivňovat změnu stavu buňky, jejíž stav se snažíme vypočítat. Aby tyto změny měly smysl, síla ovlivnění stavu by měla záležet na pozici těchto sousedních buněk. Lze to například řešit tak, že je kolem současné buňky definován „kernel“ který určuje, jakou silou na ni působí okolní buňky. Tento kernel definuje sílu působení sousedních buněk na všech místech v určeném rozsahu [12].

1.6.3 Využití náhodnosti

Všechny příklady buněčných automatů které byly zatím v této práci uvedeny jsou implementovány s plně deterministickými pravidly. Možnou generalizací systému přechodů mezi stavy by bylo přidání možnosti využití stochastického prvku při výpočtu těchto přechodů. Tato stochasticita se může jevit například tak, že buňka má šanci z jednoho stavu přejít na jeden ze čtyřech různých stavů, každý se svou pravděpodobností výskytu [2].

1.6.4 Rozdílná pravidla pro různé buňky

Tento druh generalizace spočívá v tom, že ne všechny buňky v mřížce musí dodržovat stejný set pravidel pro přechod mezi stavy. Buňky v mřížce jsou libovolným způsobem rozdělené do skupin, ve kterých se tato pravidla liší. Buňky se dá do těchto skupin rozdělit různými

způsoby. Jedním z nejjednodušších způsobů je rozdělení těchto buněk náhodně [2].

1.6.5 Pohyblivé buňky

Buňky v klasických implementacích buněčných automatů nejsou schopné pohybu. I když aktivita v průběhu simulace těchto buněčných automatů může vypadat podobně jako pohyb, jedná se pouze o šíření dat v podobě řetězce změn stavů mezi různými buňkami. Generalizace pohyblivých buněk toto omezení odstraňuje, a tím povoluje individuálním buňkám měnit svoji pozici. Pravidla pro pohyb fungují stejným způsobem jako klasická pravidla pro přechody mezi stavy. Buňka například může počítat, kolik má živých sousedů. Při dosažení požadovaného počtu sousedů se posune dopředu. Tato generalizace je užitečná, pokud v systému chceme zachovat určitý počet elementů, který nenarůstá. Aktivní buňky v tomto případě nemusí vytvářet nové buňky, ale jenom se pohybují [2].

1.6.6 Dynamická struktura mřížky

Tato generalizace umožňuje, aby se společně se stavy buněk v systému buněčného automatu také měnila samotná struktura mřížky, na které simulace probíhá. Změna struktury mřížky spočívá v tom, že se s každým krokem podle vybraného pravidla změní, jak jsou na sebe různé pozice v mřížce napojené. Tím se může výrazně změnit dynamika mezi buňkami na této mřížce [2].

1.7 Reálná využití

Buněčné automaty mají mnoho různých reálných využití, především pro různé druhy simulací.

Pomocí buněčných automatů lze například simulovat, jakým způsobem se v populaci mohou šířit a vyvíjet nemoci [13], modelovat chování vozidel v komplexních dopravních situacích [14], nebo například modelovat šíření lesních požárů [15].

2 SIMULÁTORY BUNĚČNÝCH AUTOMATŮ

Existuje řada různých simulátorů zabývajících se simulací buněčných automatů. Tato kapitola se věnuje některým z nejpoužívanějších simulátorů, rozdílům mezi nimi, jejich výhodám a nevýhodám.

2.1 Mirek's Celebration

Simulátor Mirek's Celebration, také známý pod jménem MCell, byl vytvořen Mirkem Wojowiczem v roce 2024 jako moderní nástupce jeho předchozího stejnojmenného simulátoru buněčných automatů, který byl vytvořen v roce 1999 [16].

Aktuální verze programu MCell je open-source webová aplikace zaměřena na simulaci jednodimenzionálních a dvoudimenzionálních buněčných automatů [16, 17].

Buněčné automaty v tomto simulátoru mohou nabývat až 256 různých stavů, reprezentovaných různými barvami [16]. Tento simulátor podporuje pouze čtvercovou mřížku. Buněčné automaty na této mřížce mohou využívat několika různých vybraných typů sousedství. Mřížka má omezenou velikost definovanou používanými pravidly [16, 17]. Tento simulátor nabízí řadu buněčných automatů, se kterými mohou uživatelé experimentovat. Uživatelům také umožňuje definici a importování vlastních sad pravidel. Tento simulátor nepodporuje práci s generalizacemi konceptu buněčných automatů [16].

2.2 Golly

Simulátor Golly vytvořili Andrew Trevorrow, Tom Rokicki a řada přispěvatelů k repozitáři tohoto projektu [18]. Vznikl spojením simulátorů hlife a qlife, které naprogramoval Tom Rokicki [19].

Golly je open-source aplikace, která podporuje běh na různých operačních systémech. Mezi tyto operační systémy patří Windows, Linux, macOS, Android a iOS. Může také pracovat ve webovém prohlížeči [18].

Podporuje simulaci jednodimenzionálních, dvoudimenzionálních a trojdimenzionálních buněčných automatů. Stejně jako u předchozího příkladu mohou buněčné automaty v tomto simulátoru nabývat až 256 různých stavů. Mřížka, na které se buněčné automaty nacházejí ale není omezená pouze na čtverce. Může být například složena z šestiúhelníků [18]. Velikost

mřížky v tomto simulátoru může být neomezená. Velikostně omezené mřížky však obsahují podporu pro různé topologie, jako například topologie válce, torusu, či kleinovy lahve [20]. Podobně jako u předchozího příkladu nabízí Golly mnoho existujících buněčných automatů. Také nabízí možnost vytvářet a importovat vlastní sady pravidel. Obsahuje ale také podporu pro přidávání nové funkcionality pomocí skriptů napsaných v jazycích Lua a Python. Dále také podporuje přehrávání zvuků [18].

2.3 Ready

Ready je simulátor buněčných automatů, který vytvořili Tim Hutton, Robert Munafo, Andrew Trevorrow, Tom Rokicki a Dan Wills, společně s mnoha dalšími programátory, kteří přispěli svůj kód do otevřeného GitHub repozitáře tohoto projektu [21]. S již zmíněným simulátorem Golly tedy sdílí několik svých autorů.

Simulátor Ready je stejně jako předchozí příklady open-source. Neobsahuje však verzi spustitelnou ve webovém prohlížeči [22].

Na rozdíl od předchozích příkladů je tento simulátor zaměřený na simulaci některých z existujících generalizací konceptu buněčných automatů. Snaží se simulovat spojitě buněčné automaty. Stejně jako simulátor Golly simuluje buněčné automaty v jedné, dvou až třech dimenzích [22]. Namísto stavu reprezentovaného celými čísly používá každá buňka (zde známá také jako chemikálie) spojitou hodnotu. Tato spojitá hodnota je s každým krokem simulace postupně ovlivňována hodnotami dalších buněk v okolí, definovaném kernelem [23]. Tento simulátor je také schopný simulovat klasické buněčné automaty stejně jako ostatní příklady v této sekci. Podporuje různé tvary mřížky a různé topologie. Obsahuje vlastní knihovnu existujících příkladů pravidel. Lze také definovat vlastní pravidla pomocí proprietárního souborového formátu založenému na XML [22].

2.4 CellLab

CellLab a jeho webová implementace WebCA, jsou simulátory buněčných automatů, které vytvořili Rudy Rucker a John Walker [24].

Jako jediný z uvedených příkladů není simulátor CellLab open-source. Originální verze tohoto simulátoru byla vydána jako placený výukový balíček od společnosti Autodesk pod jménem „Rudy Rucker’s Cellular Automata Laboratory“. Po ukončení distribuce tohoto soft-

warového balíčku se autoři rozhodli vydat vylepšenou verzi ve formě webové aplikace [24]. Tento simulátor je schopený pracovat s jednodimenzionálními a dvoudimenzionálními buněčnými automaty. Tyto buněčné automaty existují ve čtvercové mřížce. Stejně jako ostatní simulátory obsahuje knihovnu pravidel. Tato pravidla jsou plně open-source. Na rozdíl od ostatních příkladů nejsou pravidla definována pouze jako jednoduché instrukce pro změny stavů dle buněk v sousedství, ale jsou definována pomocí programovacího jazyku JavaScript, případně Java, což umožňuje výrazně rozšířit funkcionalitu těchto buněčných automatů [25]. Díky této změně je simulátor také schopený simulovat některé z generalizací konceptu buněčných automatů, jako jsou spojitě buněčné automaty [24]. Na rozdíl od ostatních zmíněných simulátorů CellLab podporuje změnu způsobu, jakým jsou buňky spouštěny [25].

2.5 Shrnutí

Mezi silné stránky většiny těchto simulátorů patří jejich otevřenost, rozšiřitelnost a dostupnost na mnoha různých platformách. Výhodou jsou také zahrnuté kolekce různých pravidel, které tyto simulátory obsahují.

Hlavní nevýhodou těchto simulátorů je jejich omezenost v rámci práce s generalizacemi buněčných automatů. Další slabou stránkou je také omezená sada prostorových konfigurací, které se v těchto simulátorech dají používat.

Při procesu návrhu a implementace vlastního simulátoru buněčných automatů je tedy důležité vzít tyto body v potaz, zakomponovat do něj a případně také rozšířit nalezené silné stránky a vyhnout se nalezeným problémům.

II PRAKTICKÁ ČÁST

3 POŽADAVKY NA SYSTÉM

Před prvotním návrhem a implementací systému je potřeba stanovit požadavky na jeho funkcionalitu. Tyto požadavky poté při procesu návrhu systému slouží jako šablona, podle které jsou prováděny důležitá rozhodnutí ohledně struktury tohoto systému.

3.1 Simulace buněčných automatů

Systém musí být schopený provádět simulace buněčných automatů v reálném čase. Tyto simulace by měly být ovlivnitelné uživatelskou interakcí.

3.2 Modularita

Simulátor by neměl být omezený na simulaci omezeného počtu typů buněčných automatů. Jeho uživatel by měl být schopený rozšiřovat jeho funkcionalitu pomocí modulů, které si buď to vytvoří sám, nebo je sežene od jiných autorů těchto modulů.

Moduly by měly být schopné do simulace přidávat jak nové typy buněčných automatů, tak i nové konfigurace prostředí, ve kterých tyto buněčné automaty budou existovat. To zahrnuje například nové tvary mřížky, nové druhy topologie, nebo také nové způsoby interakce prostředí s buňkami.

3.3 Obecnost

Systém by měl být schopený simulovat co největší množství druhů buněčných automatů. Simulátor by tedy měl být navržen tím způsobem, aby nabízel podporu pro simulaci co nejvíce generalizací konceptu buněčných automatů. Některé z těchto konceptů jsou popsány v kapitole 1.6.

Mezi tyto generalizace patří využívání různých sad pravidel pro různé buňky (kapitola 1.6.4). To znamená, že moduly, které přidávají sady pravidel pro buněčné automaty, a zároveň fungují ve stejné konfiguraci prostředí, by měly být do jisté míry spolu kompatibilní, aby byly schopné spolu existovat ve stejné simulaci.

Dalším důležitým bodem, který je potřeba v rámci generality dodržet, je to, že simulátor by měl být schopený pracovat s libovolnými prostorovými konfiguracemi (kapitola 1.3), ve kterých mohou být buněčné automaty reprezentovány. Do tohoto konceptu se počítá jak

možnost buněk pracovat v těchto prostorech, tak i schopnost simulátoru tyto prostory v dostatečné míře vizuálně vyobrazit.

3.4 Kompatibilita modulů

Jak již bylo zmíněno v podkapitole 3.3, různé moduly přidávající nové soubory pravidel pro buněčné automaty by měly spolu být kompatibilní. To znamená, že by měl existovat způsob, jak by buňky přidané těmito moduly mohly interagovat bez toho, aby byla narušena funkčnost těchto buněk. Jeden modul by tedy neměl mít přímý přístup k datům jiného modulu.

3.5 Uživatelské rozhraní

Simulátor by měl být schopný dát uživateli možnost vidět, co se v simulaci děje, a zároveň by měl uživateli poskytovat nástroje k tomu, aby mohl se simulací interagovat.

3.6 Jednoduchost

Práce se simulátorem by ze strany uživatele měla být co nejjednodušší. To zahrnuje jednoduchost interakce s uživatelským rozhraním, ale také jednoduchost při instalaci nových modulů, či jednoduchost při vytváření vlastních modulů.

V rámci jednoduchosti při vytváření vlastních modulů by měl systém být dostatečně zdokumentovaný na to, aby bylo vytváření modulů dostatečně jednoduše pochopitelné.

Instalace tohoto simulátoru by neměla být pro uživatele složitá. Ideálně by měl fungovat „out-of-the-box“, s využitím spustitelného souboru.

3.7 Multiplatformnost

Finální program by měl být spustitelný na operačních systémech Windows, Linux a macOS, případně také dalších operačních systémech.

Moduly vytvořené pro tento simulátor by také měly být multiplatformní.

3.8 Otevřenost

Systém by měl být otevřený. To znamená, že jeho kód by měl být volně dostupný pro uživatele, kteří chtějí vytvářet vlastní modifikace tohoto programu, pro ty které zajímá, jak byl tento software naprogramovaný.

4 NÁVRH ARCHITEKTURY SYSTÉMU

Před započítím programování systému je důležité vytvořit solidní návrh architektury systému, podle které se bude proces vývoje řídit. Tento návrh by se měl řídit podle požadavků na systém, které byly stanoveny v předchozí kapitole.

4.1 Rozdělení povinností v simulaci

Proces simulace by měl být dostatečně obecný na to, aby jeho implementace co nejméně omezovala druhy buněčných automatů, které mohou být v tomto systému simulovány. To znamená, že jakékoliv specializované funkce, které jsou důležité pro funkcionalitu pouze omezeného počtu modulů, by neměly být implementovány v samotném simulátoru, ale pouze v těch modulech, kde jsou potřeba. Zároveň ale není ideální, aby každý modul přidávající nové buňky musel tyto specializované funkce implementovat individuálně. Výrazně by to mohlo zkomplikovat proces vytváření nových typů buněk, což by šlo proti požadavku na jednoduchost (kapitola 3.6), a zároveň by mohly nastat problémy s kompatibilitou mezi jednotlivými typy buněk, což by šlo proti požadavku na kompatibilitu modulů (kapitola 3.4). Řešením tohoto problému je vytvoření hierarchie navazujících modulů, které postupně specifikují, jak má simulace fungovat. To zajistí, že implementace různých typů buněčných automatů se mohou zaměřovat pouze na chování těchto automatů, a zároveň to, že pravidla stanovená pro interakci mezi těmito implementacemi mohou být konzistentně dodržována bez nutnosti toho, aby autoři těchto implementací museli tato pravidla detailně znát.

4.2 Definice hierarchie modulů

Hierarchie modulů by měla být navržena tím způsobem, aby co nejvíce snižovala nutnost opakované implementace stejné funkcionality napříč mnoha různými moduly. To znamená, že moduly na nižší úrovni by měly poskytovat nástroje určené k tomu, aby ulehčovaly práci modulům, které jsou na vyšších úrovních. Dalším důležitým důvodem pro definici této hierarchie je vynucení konstantního dodržování pravidel pro kompatibilní definice buněčných automatů.

Všechny tyto kompatibilní buněčné automaty by tedy byly definované tak, aby mohly pracovat pod jedním modulem, který definuje, jakým způsobem tyto buňky mohou interagovat mezi sebou a také jakým způsobem mohou interagovat se samotným prostředím. Tento

modul reprezentující prostředí se potom postará o zprostředkování těchto interakcí, a také o to, aby byly v těchto interakcích dodrženy všechna pravidla tímto prostředím stanovená.

Simulátor má mít podporu pro různé prostorové konfigurace. Pro to, aby tato podpora měla smysl, musí simulátor obsahovat také podporu pro vykreslování těchto různých prostorových konfigurací a buněk v těchto konfiguracích (kapitola 3.3). Různá prostředí a s nimi asociované buněčné automaty mohou, ale nemusí, pracovat se stejnými prostorovými konfiguracemi. Těchto možných konfigurací je také velké množství. Díky tomu není ideální, aby bylo vykreslování řešeno přímo v simulátoru, a mělo by tedy být řešeno vykreslovacími moduly. Vykreslovací moduly tedy řeší problém podpory pro vykreslování velkého množství prostorových konfigurací, a zároveň pomáhají omezit redundantnost v kódu tím, že prostředí, která využívají stejný druh prostorové konfigurace, nemusí vytvářet vlastní implementace, ale mohou využít existujících vykreslovacích modulů. Mimo vykreslování buněk a prostředí, ve kterém se tyto buňky nachází, může tento typ modulu také poskytovat prostředí další vykreslovací funkcionalitu. Pokud chce prostředí například zvýraznit specifické regiony v simulaci, nebo třeba pokud prostředí chce zobrazit text s informacemi ohledně simulace, vykreslovací modul mu k těmto účelům může poskytnout potřebnou sadu nástrojů.

Na nejvyšší úrovni hierarchie modulů je tedy typ modulu, který přidává jednu nebo více definic buněčných automatů. Na střední úrovni je typ modulu představující prostředí, ve kterém se buňky vyskytují. Toto prostředí buňkám poskytuje informace o jejich okolí, a zároveň o věcech, které se v tomto prostředí dějí. Třetím a nejnižším typem modulů v této hierarchii jsou vykreslovače, které uživateli poskytují náhled na to, co se v simulaci děje. Také prostředí poskytují nástroje, které mohou být použity pro vykreslování dalších grafických elementů, které mohou být pro simulaci užitečné.

4.3 Struktura buněk

Buňky v simulaci buněčných automatů fungují jako aktéři, kteří provádějí dva hlavní typy akcí. Prvním typem akce je zjišťování informací o svém okolí. Druhým je poté vykonávání akcí, které toto okolí ovlivňují. V návrhu definovanému v předchozí sekci této kapitoly zprostředkovává získávání informací a vykonávání akcí modul prostředí. Buňka by tedy měla mít přímý kontakt s prostředím, pomocí kterého tyto akce bude provádět. Kvůli tomu, aby prostředí mohlo o buňkách zjišťovat informace, musí s buňkou také mít přímý kontakt. To

znamená, že komunikace mezi buňkou a prostředím bude do jisté míry oboustranná, a bude zajištěna pomocí referencí, které existují na obou stranách této interakce. Informace o stavu buněk by měly být ukládány přímo v datové struktuře těchto buněk. V rámci vynucování dodržování pravidel je část tohoto stavu zpracována prostředím. K této části stavu by buňky neměly mít přímý přístup, ale měly by být spravovány pomocí interakcí s prostředím. Tím se zamezí tomu, že by buňka „ignorovala“ pravidla tím, že by svůj stav měnila „manuálně“. Jako příklad situace, kde tato konfigurace dává smysl, může být uveden pohyb buněk v prostředí, jehož pravidla omezují, jak daleko se buňka může pohnout. Pokud by buňka měla přímý přístup ke své pozici, mohla by tuto pozici změnit na libovolnou hodnotu bez využití funkcí poskytovaných prostředím. Tím by buňka obcházela pravidla daná tímto prostředím. Některé informace o stavu buňky jsou ale pouze využívány v interním procesu rozhodování, a pro dodržování pravidel prostředí nejsou důležitá. Příklad této části stavu může třeba být, když si buňka ukládá informace o počtu sousedů v předchozím kroku pro využití v rozhodování. V tomto případě jsou tyto informace sice získávány z prostředí, ale jejich zpracování a uložení závisí pouze na tom, co buňka zrovna potřebuje.

Tento problém duality typů informací může být vyřešen pomocí rozdělení datové struktury buňky na dvě části. První část je pro všechny typy buněk stejná. Obsahuje prostor, do kterého může modul prostředí ukládat arbitrární data důležitá pro jeho funkcionalitu. Druhá část buňky poté obsahuje proměnné, které jsou definované buňkou samotnou. Tato část také obsahuje funkční část kódu buňky, která je periodicky volána simulátorem. V této části jsou volány funkce poskytované prostředím. Tyto dvě části buňky nemají kvůli dodržování pravidel přímý kontakt. Znamená to, že pokud chce buňka zjistit informace ohledně části svého stavu spravované prostředím, musí se k těmto datům pomocí prostředí dostat. Na rozdíl od části spravované prostředím nemá prostředí přímý přístup do interní části datové struktury buňky. Proměnné v této části závisí na definici buňky, a kvůli tomu se mezi různými typy buněk liší.

4.4 Instancování buněk

To, že simulátor musí podporovat různé prostorové konfigurace (kapitola 1.3) znamená, že také musí podporovat konfigurace nacházející se v prostředí, které je alespoň v jednom směru nekonečné. Fyzicky ale není možné, aby jakékoliv zařízení, na kterém simulátor poběží, mohlo nekonečný počet buněk simulovat. Kvůli tomu je potřeba, aby buňky byly reprezen-

továny instancemi, které budou dynamicky vytvářeny a mazány. Kvůli tomu se v simulaci budou nacházet místa, která se dají považovat za „prázdná“. To znamená, že tyto buňky nemají k sobě připojenou žádnou instanci třídy reprezentující buňku. Simulátor tedy musí zajistit, aby buňky mohly být schopné vytvářet nové instance buněk pomocí prostředí. Funkce některých existujících buněčných automatů také musí být do jisté míry upravena, aby byly schopny s prázdnými místy pracovat.

4.5 Správa buněk

Pro zajištění správné funkcionality simulátoru je potřeba, aby buňky byly nějakým způsobem spravovány a periodicky spouštěny. Tato funkcionality je pro všechny buňky stejná. Nezávisí tedy na specifickém modulu. To znamená, že by měla být implementována přímo v simulátoru.

Kvůli podpoře asynchronicity, u které bylo potvrzeno, že buňky mohou být spouštěny individuálně bez ztráty možnosti simulace buněčných automatů, které využívají synchronního spouštění (kapitola 1.6.1), je individuální spouštění kódu každé z buněk ideálním přístupem ke spouštění.

O tuto implementaci se může starat interní modul, který je schopný buňky vytvářet, mazat, organizovat a spouštět. Možný způsob organizace těchto buněk je dle pořadí, ve kterém jsou spouštěny. Buňky by tedy měly být organizovány v listu, do kterého jsou v průběhu simulace přidávány, a ze kterého jsou eventuálně odebírány. Buňky mohou vytvářet další buňky. Aby se zabránilo mnohonásobnému řetězovému vytváření nových buněk je potřeba, aby se nově přidané buňky v listu řadily před buňkou, která se aktuálně zpracovává. Tím nebude možné, aby byl nově vytvořený potomek spuštěn ve stejném cyklu, jako jeho rodič, a nemohl vytvořit další potomky, kteří by znovu byli spuštěni před dokončením současného cyklu. Při mazání buňky z listu je potřeba zajistit, aby nedošlo k pokusu tuto buňku spustit.

Spouštěcí modul by také měl být schopný poskytovat informace ohledně buněk modulům pro simulaci prostředí a modulům pro vykreslování simulace. Měl by také poskytovat informace o tom, která buňka je aktuálně aktivní. Zároveň by modulům prostředí měl poskytovat základní informace o běhu simulace důležité pro jeho běh.

Tento modul by některé ze svých funkcí měl poskytovat modulům prostředí a uživatelskému rozhraní. Prostředí by měly být poskytovány funkce, které umožňují přidávání no-

vých buněk a odebírání buněk existujících. Uživatelské rozhraní by potom mělo mít přístup k funkcím, které ovládají průběh simulace. Především funkce zajišťující spouštění této simulaci.

Nové buňky jsou do simulace přidávány především pomocí funkcí prostředí. Dalším způsobem přidávání buněk do simulace je pomocí uživatelského rozhraní.

Při výběru konfigurace simulace si uživatel přímo nevybírá, které buňky simulátor načte. Výběr načtených buněk záleží na výběru modulu prostředí. Simulátor potom podle vybraného prostředí zjistí, které buňky jsou s tímto prostředím kompatibilní, a podle toho jsou poté načteny.

4.6 Struktura prostředí

Jak již bylo zmíněno v předchozích sekcích, moduly prostředí zprostředkovávají interakce mezi různými buňkami, a mezi buňkami a sebou. Zároveň se také starají o ty části stavu buněk, u kterých nelze plně důvěřovat implementacím těchto buněk, čímž se zajišťuje, že buňky budou dodržovat pravidla stanovená v prostředích.

Prostředí poskytuje buňkám funkce, které mohou být volány za jejich běhu. Tyto funkce buňkám dávají informace o sobě a jejich okolnímu prostředí, a zároveň jim umožňují provádět v tomto prostředí akce. Všechny akce buněk, které toto okolí ovlivňují, musí být prováděny přes funkce poskytované prostředím. Typy akcí, které jsou buňkám prostředím poskytovány, jsou definovány v těchto prostředích. Stejná pravidla platí také pro funkce, které buňkám poskytují informace o jejich okolí. Musí být poskytovány prostředím, ve kterém jsou také definovány. Definice funkcí v prostředí zajišťuje, že buněčné automaty definované pro toto prostředí nemohou porušovat pravidla stanovená tímto prostředím. Například, pokud jsou buňky v simulaci omezené na Moorovo sousedství, prostředí může zajistit, že buňky nebudou schopny zjistit žádné další informace o buňkách, které se vyskytují mimo toto sousedství. Zároveň může zajistit, že buňka nemůže mimo rozsah tohoto okolí provádět žádné akce definované tímto prostředím.

Zatímco buňky volají funkce prostředí, prostředí je schopné upravovat část stavu buňky nacházející se v datové části struktury buňky. Tato část je dostupná pouze pro prostředí, a obsahuje data, která jsou tímto prostředím zpracovávána. Funkční část struktury buňky do datové části nemá přístup, to znamená, že data uložená do datové části buňky nemohou být

funkční částí čteny ani upravovány. Toto rozdělení buňky poskytuje prostředím místo, do kterého mohou bezpečně zapisovat potřebné údaje o těch částech stavu buňky, které spravují, bez risku toho, že by do nich mohly buňky zasahovat. Pokud chceme, aby buňky měly přístup k některým nebo všem uloženým datům v jejich datové části, je potřeba, aby tyto interakce byly zprostředkovávány prostředím. Výhodou tohoto přístupu je, že prostředí získává nad možnými pokusy o změnu kontrolu, a může je tedy v rámci zachování svých pravidel ovlivňovat.

Na rozdíl od volného přístupu k datovým částem buněk, by prostředí nemělo mít přístup k funkční části těchto buněk. Tato část buněk by měla zůstat pod kontrolou spouštěcího modulu. Tato část buňky obsahuje stavové informace, které nejsou pro prostředí a dodržování pravidel relevantní. Ovlivňují pouze interní rozhodovací proces buněk.

Všechny aktivní buňky v simulaci by měly obsahovat referenci na aktivní instanci prostředí. Tato reference zajistí, že buňky mohou volat funkce svého prostředí. Instance prostředí by zároveň měla být schopná získat referenci na datovou část všech aktivních buněk v simulaci za pomoci spouštěcího modulu.

Prostředí také obsahuje referenci na aktivní instanci vykreslovacího modulu. Vykreslovač je s prostředím v podobném vztahu jako prostředí s buňkou. Vykreslovač poskytuje prostředí funkce, které slouží k ovlivňování toho, co je uživateli vykreslováno.

4.7 Kompatibilita buněk a prostředí

Každý typ buňky musí být definován pro práci ve specifickém prostředí. To zajišťuje, že buňky jsou s vybraným prostředím kompatibilní, a tak jsou schopny využívat funkcí poskytovaných vybraným prostředím. Moduly přidávající buňky by měly specifikovat, pro jaké prostředí jsou tyto buňky určeny. Díky tomu bude simulátor schopný při simulaci specifického prostředí vyřadit ty buňky, které s tímto prostředím nejsou kompatibilní. Nemělo by tak nastávat to, že se buňka bude snažit zavolat funkci prostředí, která v současně aktivním prostředí neexistuje.

4.8 Správa prostředí

Při simulaci může být aktivní pouze jedno prostředí. Typ prostředí, které bude simulováno, je vybírán uživatelem. Podle výběru prostředí jsou také načítány buňky, které jsou s tímto

prostředím kompatibilní. Na začátku simulace je vytvořena instance vybraného prostředí, která je udržována do konce této simulace. Instance buněk obsahují referenci na toto aktivní prostředí. V průběhu simulace nelze tuto aktivní instanci nahradit instancí jiného prostředí.

4.9 Struktura vykreslovače

Vykreslovací moduly slouží k tomu, aby byl simulátor schopný poskytnout obraz, který uživateli ukazuje, co se v simulaci aktuálně děje. Dávají simulátoru schopnost vykreslení současné prostorové konfigurace, a tím uživateli umožňují práci s různými prostorovými konfiguracemi.

Uživatelské rozhraní simulátoru by mělo být univerzálně použitelné pro všechny konfigurace modulů, což znamená, že kompatibilita vykreslovače a uživatelského rozhraní musí být zajištěna ze strany vykreslovače. Řešením tohoto problému je sjednocení způsobu, kterým vykreslovače reprezentují své grafické výstupy. Vykreslovací moduly by tedy měly svůj grafický výstup poskytovat jako list bajtů reprezentujících barvy pixelů tohoto výstupu, což zajistí, že výstup těchto modulů může být zpracován na straně simulátoru bez toho, aby byla potřeba využívat knihovny kompatibilní s těmi, které byly využity při generaci samotného výstupu ve vykreslovačích. Vykreslovače by také měly v rámci kompatibility s velkým množstvím různých zařízení podporovat různá výstupní rozlišení, a také způsob, jak tato výstupní rozlišení měnit za běhu.

Každý vykreslovač musí obsahovat dvě funkce. První funkce slouží pro změnu požadovaného výstupního rozlišení, a měla by být pro všechny implementace vykreslovačů stejná. Druhá funkce slouží k získání grafického výstupu pro poskytnutý list buněk.

Základní implementace buněk je generická, neobsahuje tedy nic, co by vykreslovačům říkalo, kde v prostoru se buňka vyskytuje. Vykreslovače jsou ale schopné číst datovou část buněk. To znamená, že pokud buňka obsahuje požadovaná data, může být vykreslovačem vykreslena. Vykreslovače přímo neupravují data buněk se kterými pracují, to znamená, že pro kompatibilitu s vykreslovačem musí tato data do datové struktury buněk zapisovat moduly prostředí.

Vykreslovače také mohou obsahovat další funkce, které závisí na jejich implementaci. Tyto funkce jsou dostupné pro prostředí využívající těchto vykreslovačů. Tyto funkce prostředím umožňují ovlivňovat to, co se uživateli v simulaci zobrazuje. Příkladem může být

například funkce, která změní barvu vybrané obdélníkové sekce v simulaci, což může například reprezentovat zónu, ve které mají buňky procentuální šanci na zmizení.

4.10 Kompatibilita prostředí a vykreslovačů

Podobě jako jsou buňky postavené na různých modulech prostředí, tak i vykreslovače jsou postavené na modulech nižší úrovně. Prostředí jsou postavené na vykreslovačích. Jak již bylo zmíněno v předchozí sekci, vykreslovače definují, jakým způsobem jsou v datové části buněk ukládána data o jejich pozici v prostoru, který je simulován. Vykreslovače také mohou prostředím poskytovat sadu funkcí pro úpravu toho, co je uživateli zobrazováno. Z těchto důvodů jsou moduly prostředí závislé na specifické implementaci vykreslovače, na kterých jsou postaveny, stejně jako buňky postavené na specifických prostředích.

4.11 Správa vykreslovačů

V rámci správy fungují vykreslovací moduly podobně jako moduly prostředí. To také znamená, že by měly být spravovány podobným způsobem. Stejně jako u prostředí musí v jedné instanci simulace existovat pouze jedna instance vykreslovače, která zůstává do konce simulace aktivní. Na rozdíl od prostředí ale vykreslovač není přímo vybírán uživatelem. Vykreslovač je vybrán automaticky podle toho, které prostředí bylo uživatelem vybráno. Pokud vykreslovač pro toto prostředí neexistuje, neměl by být uživatelem volitelný.

4.12 Správa modulů

Jak již bylo v této kapitole zmíněno, existují tři hlavní typy modulů, které uživatelé mohou tvořit. Vyšší z těchto typů modulů, tedy prostředí a buňky, jsou závislé na modulech relativně nižších, tedy vykreslovačích a prostředích. Je tedy nutné, aby simulátor byl schopen vyhledávat dostupné moduly, a aby při procesu výběru sady modulů byly tyto závislosti respektovány. Toto by mohlo být vyřešeno rozdělením modulů na dvě propojené části. První půlkou modulu by byla jeho funkční část, reprezentována souborem v programovacím jazyce, který definuje, jakým způsobem se tento modul chová. Druhou půlkou tohoto modulu by byl soubor s metadaty. To znamená soubor s daty, který poskytuje informace o samotném modulu. Tento soubor musí být nějakým způsobem propojený se souborem definujícím implementaci tohoto modulu. Pro minimální implementaci je třeba, aby soubor s metadaty definoval, o jaký typ modulu se jedná, aby obsahoval cestu k funkční části tohoto modulu,

aby obsahoval hodnotu, podle které je možné tento modul identifikovat a v případě vyšších modulů také aby obsahoval identifikátor modulu, na kterém je tento modul závislý. V případě buněk by také tento soubor měl obsahovat list všech uživatelsky dostupných typů buněk v tomto simulátoru. Soubor s metadaty také může obsahovat další informace ohledně modulu, které mohou být v simulátoru zobrazeny, jako je například popis modulu, jméno autora nebo obrázek reprezentující tento modul.

Díky tomuto souboru s metadaty je možné, aby simulátor jednoduše vyhledával dostupné moduly a korektně je propojoval dle jejich závislostí. Umožňuje simulátoru také zjišťovat, zda některým modulům chybí jejich závislosti, a v tomto případě uživateli nepovolit tyto moduly načíst.

Aby program nemusel procházet celý souborový systém, bylo by vhodné, aby moduly, které načítá, byly umístěny v jedné složce. Kvůli rozdělení modulů do dvou souborů, společně s možností přidání obrázků, může ale znamenat, že po přidání dostatečného množství modulů, bude obsah této složky nepřehledný, a mohou nastat problémy s jeho správou. Bylo by tedy vhodné, aby obsah této složky mohl být rozdělen do několika podadresářů. Řešením je povolení rekurzivního hledání v jedné specifikované složce, která obsahuje nainstalované moduly pro tento simulátor. To zajistí, že program neprohledává celý souborový systém, ale obsah tohoto souboru není nepřehledný.

Načítání těchto modulů je proces, který je nezávislý na vybrané konfiguraci modulů v simulaci. Měl by tedy být řešen interním modulem plnící tento úkol. Tento modul by tedy měl být schopný rekurzivního hledání souborů s metadaty modulů ve vybraném adresáři, a propojení těchto souborů s funkčními částmi modulů a případně dalšími potřebnými soubory. Měl by také být schopný tyto moduly nějakým způsobem dynamicky načítat tak, aby jejich kód mohl být využitý v samotném simulátoru.

4.13 Uživatelské rozhraní

Aby program mohl být plnohodnotně používán uživateli, je potřeba, aby obsahoval uživatelské rozhraní. Základní uživatelské rozhraní by mělo být pro všechny moduly stejné, funguje tedy jako samostatný modul, který je implementovaný přímo v simulátoru. Toto rozhraní by mělo uživateli poskytnout možnost interakce se simulátorem, a také vizuální zpětnou vazbu ohledně toho, co se v simulaci děje. Mimo práci se simulací by také mělo uživatelské roz-

hraní uživateli umožňovat výběr požadované konfigurace modulů, podle které bude simulace sestavena. Tyto dvě odpovědnosti mohou být rozděleny do dvou oken, která jsou na sebe napojená.

První z těchto oken slouží jako vstup do aplikace. Může obsahovat uvítací zprávu pro uživatele. Hlavním cílem tohoto okna je ale umožnit výběr požadované konfigurace modulů. Pro tento účel okno obsahuje list dostupných modulů načtených pomocí modulu pro načítání modulů zmíněného v předchozí podkapitole. Může také obsahovat prostor pro výpis dalších informací získaných z metadat modulů, jako je například jméno autora nebo popis modulu. Po výběru požadované konfigurace modulů okno uživateli poskytne možnost přejít do simulace, reprezentované přechodem mezi prvním a druhým oknem.

Druhé okno pak reprezentuje samotnou simulaci. Uživateli by mělo poskytnout možnost interakce se simulací a vizuální zpětnou vazbu. Obsahuje tedy kontrolní prvky, které uživateli umožňují simulaci spustit, zastavit či krokovat. Tyto kontrolní prvky jsou propojeny s modulem pro správu běhu simulace. Dále by uživatelské rozhraní mělo být standardizovaným způsobem propojeno s vykreslovacími moduly. Tyto moduly mají jednotnou funkci pro vykreslení toho, co se v simulaci děje. Uživatelské rozhraní tak může tuto funkci volat pokaždé, když potřebuje aktualizovat to, co uživateli zobrazuje. V tomto okně je vizuální odezva reprezentována obrázkem, který byl vygenerován vykreslovacím modulem. Všechny vykreslovací moduly by měly obrázky vracet ve stejném formátu, jejich zobrazování by tedy mělo být jednoduché. Při změně velikosti tohoto okna se může změnit rozlišení tohoto náhledu. Správnou změnu velikosti výstupů zprostředkovává další funkce, která je standardizována napříč vykreslovači. Je také potřeba zajistit, aby uživatel mohl do simulace přidávat nové buňky a mazat buňky existující. Okno tedy obsahuje sekci s výpisem všech načtených sad buněk kompatibilních s aktuálním prostředím, ze kterého uživatel může vybírat buňky. Aby ale přidávání buněk mohlo jednoduše fungovat, je potřeba, aby uživatel mohl interagovat s náhledem simulace pomocí myši. Pro rozšíření funkcionality si uživatel může vybrat, zda interaguje s modulem vykreslovače, či s modulem prostředí. Kliknutí na specifický pixel náhledu musí také být překonvertováno na specifickou pozici v simulaci. Toto je prováděno vykreslovačem, protože tento modul spravuje, jakým způsobem je souřadnicový systém simulace spravován. Pro další interakci s těmito moduly by také bylo vhodné, aby program obsahoval způsob, jakým může přenést vybrané funkce z těchto modulů do uživatelského

rozhraní bez přímé závislosti na specifických implementacích těchto modulů. Tento systém je blíže popsán v následující podkapitole.

4.14 Exportní funkce

V současném návrhu nemá uživatel velkou míru kontroly nad tím, jakým způsobem se chová simulace. Řešením tohoto problému je umožnění nižším modulům, tedy vykreslovačům a modulům prostředí, umožnit definovat funkce, které jsou určeny pro „export“ do uživatelského rozhraní. To znamená, že pokud používané moduly definují tyto exportní funkce, jsou uživateli zobrazovány ve formě interaktivních elementů, jako například tlačítek, posuvníků či políček pro zadání čísel. Uživatelé pak mohou tyto ovládací elementy používat pro ovládání těchto modulů, což uživateli poskytuje novou úroveň kontroly nad těmito moduly. Příkladem použití může být například vykreslovač, který mění barvu pozadí dle zadané hodnoty, či prostředí, které po zmáčknutí tlačítka změní svá pravidla.

Tyto funkce je potřeba definovat mezi všemi moduly konzistentním způsobem. Je tedy potřeba vytvořit další interní modul, který poskytuje ostatním modulům nástroje pro vytváření vlastních exportních funkcí standardizovaným způsobem. Exportní funkce by tedy měly být datové struktury, které definují potřebné parametry pro tyto funkce. Mezi tyto parametry patří typ ovládacího prvku, který funkci reprezentuje, text, který uživateli naznačuje, co ovládací prvek dělá, reference na funkci, kterou ovládací prvek volá, a případné další argumenty relevantní k vybranému typu elementu, jako například rozmezí hodnot pro posuvník.

5 VÝBĚR TECHNOLOGIÍ PRO VYTVOŘENÍ SIMULÁTORU

Pro implementaci návrhu projektu je třeba vybrat vhodnou sadu nástrojů. Tato sada nástrojů by měla umožňovat jednoduchý proces vývoje jak samotného simulátoru tak externích modulů vytvářených uživateli tohoto simulátoru.

5.1 Programovací jazyk

Jako programovací jazyk pro tento projekt byl vybrán Python. Oproti ostatním jazykům má Python několik výhod, které jsou pro tento projekt velmi relevantní.

Python je interpretovaný jazyk, který je podporován mnoha existujícími operačními systémy. Výrazně tedy pomáhá s požadavkem na multiplatformnost projektu. Další výhodou v tom, že je Python interpretovaný je to, že podporuje jednoduchý systém pro dynamické načítání kódu z externích souborů, což velmi napomáhá s požadavkem na modularitu. Díky tomu moduly mohou být také psány v tomto jazyce, což uživatelům zjednodušuje proces vývoje těchto modulů, a pomáhá s požadavkem na jednoduchost.

Jednoduchost čtení a úpravy programů napsaných v Pythonu pomáhá s požadavkem na otevřenost systému. Funkcionalita programu je díky tomu pro uživatele přístupnější.

Výhodou je také velké množství dostupných knihoven a materiálů ohledně tohoto jazyka, které jsou užitečné jak při vývoji simulátoru tak při vývoji externích modulů pro tento simulátor.

Způsob toho, jak Python pracuje s datovými typy, je užitečný pro implementaci některých složitěji propojených částí navržené architektury tohoto simulátoru.

Hlavní nevýhodou tohoto jazyka je především rychlost, s jakou pracuje. Při vývoji je tedy potřeba brát ohledy na optimalizace běhu aplikace, aby rychlost jazyka projekt ovlivňovala co nejméně.

Díky všem těmto výhodám je Python ideální jazyk pro implementaci tohoto simulátoru.

5.2 Knihovny

Implementace některých součástí projektu by byla v čistém jazyce Python komplikovaná. V těchto případech je vhodné využít relevantní knihovny, které tento proces zjednoduší.

Při procesu vývoje nastala nutnost vytvořit uživatelské rozhraní. Pro tento účel byla vy-

brána knihovna PyQt6. Tato knihovna byla vybrána především díky její jednoduchosti, a díky její podpoře mnoha operačních systémů.

Pro ukládání metadat byly vybrány soubory ve formátu JSON. V Pythonu je možné s těmito soubory jednoduše pracovat pomocí zabudované knihovny json. Důvodem pro výběr tohoto formátu je především jeho jednoduchá čitelnost ze strany uživatele. Z velké části jsou jeho data zapsána v čitelném formátu. Mohou být tedy uživateli jednoduše čteny a upravovány.

Pro vytvoření některých ukázkových modulů bylo potřeba vytvářet vizuální výstup ve formě obrázku. Na vytváření těchto obrázků byla vybrána knihovna Pillow. Tato knihovna obsahuje sadu nástrojů pro vytváření a úpravu obrázků, spolu s nástroji na konverzi těchto obrázků mezi různými formáty. Stejně jako PyQt6 je tato knihovna multiplatformní.

Prohledávání externích modulů a jejich dynamické načítání bylo vyřešeno pomocí interní knihovny importlib. Tato knihovna je schopná z načteného zdrojového souboru zjistit, jaké datové struktury tento soubor obsahuje, a může vytvářet funkční reference na tyto datové struktury, které mohou být použity ve spuštěném programu.

Zabudovaná knihovna abc byla využita pro umožnění práce s abstraktními třídami v jazyce Python. Díky tomu bylo možné správně vytvořit generické verze externích modulů a v nich zaznačit, které funkce je potřeba implementovat.

Z různých dalších důvodů byly při procesu vývoje simulátoru využity další zabudované knihovny jazyka Python. Tyto knihovny jsou: sys, os, enum, io a math.

5.3 Zveřejnění kódu

Pro zajištění otevřenosti kódu byl zdrojový kód tohoto simulátoru a jeho modulů publikován pod MIT licencí na platformě GitHub. Zdrojový kód tedy pod touto licencí lze nalézt na adrese: <https://github.com/m-machala/OpenCellLab>

Mimo platformu GitHub je také zdrojový kód dostupný jako elektronická příloha odevzdána společně s touto prací.

6 IMPLEMENTACE NÁVRHU

Finální implementace simulátoru buněčných automatů se odvíjí od návrhu vytvořeného v kapitole 4, s využitím nástrojů sepsaných v kapitole 5. Stejně jako návrh architektury se tato implementace dělí na několik spolupracujících modulů. Externí moduly jsou reprezentovány generickými implementacemi těchto modulů které jsou určené jako základ pro plné implementace těchto modulů. Ukázkové implementace externích modulů jsou blíže popsány v kapitole 7.

6.1 Metadata

Jak již bylo zmíněno v předchozí kapitole, soubory s metadaty jsou ukládány ve formátu JSON. Specifické rozložení těchto souborů se liší dle toho, jaký typ modulu reprezentují. Aby bylo možné s těmito soubory jednoduše pracovat, některé části těchto souborů jsou stejné napříč všemi druhy externích modulů.

Soubor s metadaty pro balíček buněk má oproti souborům pro další typy externích modulů komplikovanější strukturu. Obsahuje typ balíčku s hodnotou „cell“, jméno balíčku, jméno autora či autorů, popis balíčku, relativní cestu k obrázku reprezentující tento balíček, relativní cestu k funkční části balíčku, jméno třídy prostředí, na kterém jsou buňky v tomto balíčku závislé a list buněk, které tento modul zpřístupňuje uživateli. Každá buňka v tomto listu obsahuje název třídy ve funkční části modulu reprezentující tuto buňku, jméno buňky, popis buňky a relativní cestu k obrázku, který tuto buňku reprezentuje. Ne všechna z těchto dat jsou však pro vytvoření funkčního modulu potřeba. Minimální soubor s metadaty pro balíček buněk potřebuje definovat typ balíčku, jméno balíčku, jméno třídy prostředí a list buněk s alespoň jednou buňkou. Tato buňka pak musí mít jméno třídy a jméno buňky.

Struktura metadata souboru pro prostředí je jednodušší než ta pro balíček buněk. Obsahuje typ balíčku s hodnotou „environment“, jméno balíčku, jméno autora či autorů, popis balíčku, relativní cestu k obrázku reprezentující prostředí, relativní cestu k funkční části modulu, jméno třídy reprezentující tento modul a jméno třídy vykreslovače, na kterém je toto prostředí závislé. Minimální verze tohoto souboru použitelná v simulátoru pouze obsahuje typ modulu, jméno modulu, cestu k modulu, třídu modulu a třídu vykreslovače.

V případě vykreslovače je struktura jeho metadat velmi podobná metadatům prostředí. Obsahuje typ balíčku s hodnotou „renderer“, jméno balíčku, jméno autora či autorů balíčku,

```
1 {
2   "package type": "cell",
3   "package name": "PACKAGE",
4   "author name": "AUTHOR",
5   "package description": "DESCRIPTION",
6   "package image path": "PATH",
7   "package path": "PATH",
8   "environment class": "CLASS",
9   "cell types":
10  [
11    {
12      "cell class": "CLASS",
13      "cell name": "CELL",
14      "cell description": "DESCRIPTION",
15      "cell image path": "PATH"
16    }
17  ]
18 }
```

Výpis 6.1 Ukázková struktura JSON souboru pro balíček buněk

```
1 {
2   "package type": "environment",
3   "package name": "PACKAGE",
4   "author name": "AUTHOR",
5   "package description": "DESCRIPTION",
6   "package image path": "PATH",
7   "package path": "PATH",
8   "package class": "CLASS",
9   "renderer class": "CLASS"
10 }
```

Výpis 6.2 Ukázková struktura JSON souboru pro prostředí

popis balíčku, relativní cestu k obrázku reprezentujícímu modul, relativní cestu k funkční části modulu a jméno třídy reprezentující tento modul. Vykreslovače nejsou přímo závislé na žádných dalších typech externích modulů, nepotřebují proto referenci na závislosti. Stejně jako další soubory s metadaty má i tento soubor minimální přijatelnou strukturu. Obsahuje typ balíčku, jméno balíčku, cestu k balíčku a třídu balíčku.

```
1 {
2   "package type": "renderer",
3   "package name": "PACKAGE",
4   "author name": "AUTHOR",
5   "package description": "DESCRIPTION",
6   "package image path": "PATH",
7   "package path": "PATH",
8   "package class": "CLASS"
9 }
```

Výpis 6.3 Ukázková struktura JSON souboru pro vykreslovač

6.2 Buňka

Buňky, jako jeden z externích modulů, mají v samotné implementaci simulátoru pouze generickou implementaci. V simulátoru slouží k přidání nových typů buněk, které mohou být přidány do simulace. Tato implementace je rozdělena na dva soubory.

Prvním z těchto souborů je `Cell.py`. Tento soubor obsahuje definici třídy `Cell`. Tato třída slouží jako základ všech buněk vytvořených pro simulátor.

Třída `Cell` obsahuje inicializační funkci, která po vytvoření instance této třídy vytvoří slovník pod jménem `cellData` a proměnnou pro uložení reference na instanci rozhodovací části buňky, pod jménem `cellBrain`. Tato reference je při vytvoření instance třídy buňky inicializační funkci předána jako vstupní proměnná.

Slovník `cellData` slouží jako prostor, kde mohou být ukládány části stavu buňky ovládané modulem prostředí. Tato datová struktura umožňuje prostředí do instance buňky ukládat libovolná data potřebná pro správný běh simulace. Tato data nemohou být přímo měněna samotnou buňkou.

Mimo funkci pro inicializaci obsahuje třída `Cell` také funkci pro spuštění kódu v pro-

měnné `cellBrain`, což dalším částem kódu ulehčuje proces spouštění funkční části buněk.

Proměnná `cellBrain` ukládá referenci na instanci třídy, která je odvozená třídy vyskytující se v druhém souboru tvořícímu generickou implementaci buňky, soubor `CellBrain.py`. Tato třída se jmenuje `CellBrain`. Stejně jako třída `Cell` obsahuje inicializační funkci. V tomto případě tato funkce vytváří proměnnou, která obsahuje referenci na aktivní instanci prostředí pod jménem `_environment`. Mimo to obsahuje abstraktní funkci `run`, která musí být pro vytvoření odvozené třídy implementována. Tato funkce je spouštěna spouštěcím modulem pomocí funkce `execute` třídy `Cell`.

Reference na instanci prostředí slouží k tomu, aby buňka mohla s prostředím interagovat. Implementace prostředí obsahují funkce, které mohou být buňkami volány. Tyto funkce zprostředkovávají interakce mezi buňkami a jejich okolím. To znamená, že jakékoliv akce, které buňka provádí, mimo ty, které pouze ovlivňují její interní stav (interní proměnné ve třídě `CellBrain`), musí být prováděny přes funkce poskytované prostředím, na kterém je modul buňky závislý.

Pro vytvoření nového balíčku buněk je potřeba vytvořit nový Python skript, který obsahuje definice tříd odvozených od třídy `CellBrain`. Třída `Cell` je pro všechny buňky stejná. Dalším krokem vytvoření balíčku buněk je vytvoření odpovídajícího metadata souboru definovaného v předchozí části této kapitoly. Jako názvy tříd pro buňky se v tomto případě využívají názvy definované v tomto souboru.

6.3 Prostředí

Prostředí se řadí mezi externí moduly. V simulátoru mají tedy jenom generickou implementaci, se kterou uživatelé dále pracují.

Tato implementace se nachází v souboru `Environment.py`, který definuje abstraktní třídu `Environment`. Tato třída slouží jako šablona, ze které jsou následně vytvářeny plné implementace externích modulů prostředí. Obsahuje dvě předdefinované funkce a devět funkcí bez funkční definice, které jsou volány z dalších tříd v případě událostí, které mohou být pro prostředí důležité. Tyto nedefinované funkce mohou, ale nemusí, být autorem modulu definovány při vývoji modulu prostředí. Pokud implementovány nejsou, obsahují pouze klíčové slovo `pass`, které neprovádí žádné akce.

První z předdefinovaných funkcí je funkce pro inicializaci této třídy. Jako vstupní proměnné

nou přijímá referenci na vykreslovač, jehož funkce mohou být následně z prostředí volány. Tato reference se ukládá pod jménem `_renderer`. Dále tato funkce vytváří list pro ukládání funkcí, které budou za běhu simulátoru vyexportovány do uživatelského rozhraní. Tento list se ukládá pod jménem `_exportFunctions`. Tyto funkce jsou v uživatelském rozhraní reprezentovány ovládacími prvky, se kterými uživatel může interagovat. Detailnější popis implementace se nachází v kapitole 6.8.

Druhou předdefinovanou funkcí je `_setExecutor`. Tato funkce by pro správnou funkci prostředí měla být zavolána ihned po vytvoření instance prostředí z lokace, kde tato instance byla vytvořena. Slouží k inicializaci proměnné `_cellExecutor`, která obsahuje referenci na aktivní instanci spouštěče buněk. Tato instance je také vstupní proměnnou pro tuto funkci. Nastavení této proměnné nemůže být provedeno ihned při vytvoření nové instance prostředí kvůli cyklické struktuře referencí mezi prostředím a spouštěčem buněk. Stejně jako prostředí potřebuje referenci na spouštěč, tak spouštěč potřebuje referenci na prostředí.

Funkce `_cellsCycled` slouží k provedení akcí po tom, co spouštěč dokončí jeden celý spouštěcí cyklus buněk. Je tedy volána pokaždé když všechny aktivní buňky v simulaci jednou provedou svoje akce. Je tedy vhodná k provádění akcí, které se v prostředí stávají jednou za tik simulace, například třeba změna typu sousedství, podle kterých jsou vyhodnocovány akce buněk.

Podobně také funguje funkce `_cellSwitched`, která je volána vždy před tím, než je spuštěn kód aktuálně vyhodnocované buňky. Reference na aktuálně aktivní buňku je ve spouštěči v tuto dobu již nastavená. Prostředí s ní tedy může jednoduše pracovat. Tato funkce je tedy vhodná pro akce, které je potřeba provádět před spuštěním každé buňky. Může být například využita pro změnu některého atributu buňky před jejím spuštěním, jako je třeba její energie nebo pozice.

Funkce `_executorClearedCells` je volána vždy, když spouštěcí modul smaže všechny buňky aktuálně aktivní v simulaci. Tato funkce není volána, pokud v simulaci existuje pouze jediná buňka, která je smazána normálním způsobem. Tato funkce může být využita například tak, že po smazání všech aktivních buněk se vybrané proměnné v prostředí resetují. Díky tomu může simulace po smazání všech buněk začít od začátku stejně jako po prvním spuštění.

Zbytek funkcí v generické implementaci prostředí slouží ke zpracování uživatelských in-

terakcí s náhledem na simulaci v uživatelském rozhraní provedených pomocí myši. Tyto funkce se dělí do dvou kategorií.

Funkce v první kategorii zpracovávají kliknutí na náhled simulace. Tyto funkce jsou `_primaryClick`, `_secondaryClick` a `_tertiaryClick`. Jsou zavolány pokaždé když uživatel klikne myší na náhled simulace. V současné implementaci je primární kliknutí namapované na levé tlačítko, sekundární kliknutí na pravé tlačítko a terciární kliknutí na tlačítko kolečka myši. Všechny tyto funkce přijímají jednu vstupní proměnnou, `data`, která obsahuje reprezentaci pozice buňky, na kterou uživatel kliknul. Tato data jsou specifikovaná ve vykreslovači. Dle implementace vykreslovače se může obsah této proměnné výrazně lišit. Při implementaci těchto funkcí je tedy potřeba konzultovat dokumentaci vykreslovače. Tyto funkce jsou vhodné například pro implementaci přidávání buněk uživatelem do simulace a odebrání buněk uživatelem ze simulace.

Druhá kategorie funkcí slouží ke zpracování kombinace kliknutí a tažení myši. Stejně jako u předchozí kategorie existují ve třech variantách, `_primaryDrag`, `_secondaryDrag` a `_tertiaryDrag`. Jsou zavolány v tom případě, kdy uživatel drží tlačítko myši a nad náhledem simulace touto myší hýbe. Stejně jako u první kategorie jsou funkce namapované na levé tlačítko, pravé tlačítko a tlačítko kolečka myši. Jejich vstupní proměnné se ale liší. Tyto funkce obsahují dvě vstupní proměnné, `originalData` a `newData`, které prostředí dávají informace o tom, kde myš byla při předchozím zavolání těchto funkcí a kde je aktuálně. Tyto proměnné fungují stejně, jako u funkcí z první kategorie. Dávají prostředí informace o pozici ve formátu definovanému ve vykreslovači. Jsou také vhodné pro přidávání buněk do simulace a odebrání buněk ze simulace, s přidanou výhodou toho, že uživatel nemusí klikat na každou buňku individuálně.

Důležitou součástí procesu vytvoření nové implementace prostředí je přidání funkcí, které mohou být využívány buňkami pro interakce s jejich okolím. Pomocí těchto funkcí buňky provádějí veškeré akce mimo těch, které ovlivňují pouze jejich rozhodovací část. Tyto funkce mohou například buňce dát informace o tom, kolik má aktivních sousedů, mohou buňkám povolit vytvářet další buňky nebo třeba mohou buňkám umožňovat pohyb po jejich prostředí. Pro zjištění, která buňka je aktuálně aktivní, a tedy zavolala některou z těchto funkcí, nebo pro přidávání či odebrání buněk slouží interní proměnná `_cellExecutor` odkazující se na spouštěč buněk. Implementace tohoto modulu a jeho funkcí je blíže popsána v sekci 6.5 této

kapitoly.

Při vytváření nového modulu prostředí je třeba vytvořit nový Python skript s implementací třídy `Environment` spolu s odpovídajícím souborem s metadaty. Při vytvoření této implementace není třeba implementovat žádné z předdefinovaných funkcí. Je však doporučeno využítí funkcí pro uživatelskou interakci v rámci umožnění vytváření a mazání buněk ze strany uživatele. Jméno této vytvořené třídy musí být zaznamenáno v souboru s metadaty.

6.4 Vykreslovač

Vykreslovače jsou třetím a posledním externím modulem, který v simulátoru obsahuje pouze generickou implementaci. Slouží k vykreslování toho, co se v simulaci aktuálně děje.

Tato generická implementace je reprezentována abstraktní třídou `Renderer` nacházející se v souboru `Renderer.py`. Tato třída obsahuje funkce potřebné k vytvoření plné implementace modulu pro vykreslování. Generická implementace obsahuje dvě předdefinované funkce, dvě funkce které musí být definovány uživatelem vytvářejícím plnou implementaci a šest funkcí které uživatelem mohou ale nemusí být implementovány. Neimplementované verze těchto šesti funkcí obsahují pouze klíčové slovo `pass`, které neprovádí žádné akce.

První předdefinovanou funkcí je funkce, která instanci této třídy inicializuje. Tato funkce přijímá dvě vstupní proměnné, `outputResolutionW` a `outputResolutionH`. Jak již jejich jména napovídají, slouží k definici výstupního rozlišení pro obrázky vytvořené touto třídou. Proměnná `outputResolutionW` určuje šířku výstupního obrázku v pixelech, proměnná `outputResolutionH` pak určuje výšku tohoto obrázku v pixelech. Při vytváření instance je očekáváno, že hodnoty těchto proměnných budou vždy celá pozitivní čísla. Inicializační funkce následně tyto proměnné posílá do druhé předdefinované funkce. Mimo tuto akci inicializační funkce také vytváří interní proměnnou `_exportFunctions`, která slouží pro ukládání funkcí, které budou uživateli zpřístupněny přes uživatelské rozhraní v podobě interaktivních elementů. Detailnější popis tohoto procesu se nachází v kapitole 6.8.

Druhou předdefinovanou funkcí je funkce `_setOutputResolution`. Tato funkce slouží k nastavení interních proměnných `outputResolutionW` a `outputResolutionH` které definují šířku a výšku výstupních obrázků. Tato funkce je volána při inicializaci třídy, a následně po každé změně velikosti náhledu na simulaci v uživatelském rozhraní.

První z funkcí, které musí být autorem modulu definovány, je funkce `render`. Je to hlavní

funkce vykreslovacích modulů, která jako vstupní proměnnou přijímá list aktivních buněk. Jako výstup je očekáván list bajtů, který definuje vizuální výstup tohoto modulu. Kvůli tomu, že funkce přijímá list buněk je nutné, aby všechny buňky kompatibilní s tímto modulem měly ve svých externích částech stavu data, která určují, kde se v simulaci nacházejí. Nastavení těchto dat musí být provedeno modulem prostředí při vytváření nových instancí buněk. Díky těmto proměnným vykreslovač určuje, v jakém typu prostoru se buňky nachází. Mimo informací o pozici buněk mohou vykreslovací moduly od buněk požadovat další proměnné, které specifikují, jakým způsobem jsou buňky vykresleny. Může se mezi ně počítat například barva, nebo třeba bitmapa, která bude na místě buněk tohoto typu vykreslena.

Druhou z těchto funkcí je funkce `convertFromImageCoordinates`, která slouží k překladi z dvoudimenzionálního souřadnicového systému výstupního obrázku, reprezentováno dvěma vstupními proměnnými `xCoordinate` a `yCoordinate`, do souřadnicového systému reprezentujícího pozici v prostoru simulace. Tyto výstupní souřadnice mohou být formátovány libovolným způsobem. Funkce `convertFromImageCoordinates` je volána pokaždé, když uživatel v uživatelském rozhraní kliknutím interaguje s náhledem na simulaci generovaným vykreslovačem v kontextu interakce s prostředím. Při uživatelské interakci myši zavolá modul uživatelského rozhraní tuto funkci. Dle toho, jestli uživatel interaguje s vykreslovačem či s prostředím, jsou souřadnice zaslány do odpovídající třídy funkci, která odpovídá tlačítku myši, které bylo zmáčknuto. Pokud se jedná o prostředí, jsou zaslány konvertované souřadnice. Pokud se jedná o vykreslovač, souřadnice konvertovány nejsou.

Zbývá skupina šesti neimplementovaných funkcí slouží k obsluze uživatelských interakcí se simulací provedených myši.

Tyto funkce se dělí na dvě skupiny. První slouží ke zpracování kliknutí, a řadí se do ní funkce `_primaryClick`, `_secondaryClick` a `_tertiaryClick`. Tyto funkce jsou volány pokaždé, když uživatel zmáčkne buďto levé tlačítko, pravé tlačítko nebo tlačítko kolečka na myši. Jak již bylo zmíněno, přijímají vstupní proměnnou, data, která obsahuje data vygenerovaná funkcí `convertFromImageCoordinates`. Tato data obsahují informace o pozici v simulaci, se kterou uživatel interaguje. Vhodným využitím pro tyto funkce je například pro ovládání zvýraznění vybrané buňky v simulaci.

Funkce druhé skupiny fungují velmi podobně jako funkce ze skupiny první. Patří mezi ně funkce `_primaryDrag`, `_secondaryDrag` a `_tertiaryDrag`. Tyto funkce jsou volány v pří-

padě, že uživatel drží tlačítko myši a bez puštění tohoto tlačítka s myší pohybuje. Tyto funkce přijímají dvě proměnné, `originalData` a `newData`, které funkci dávají informace o pozici kurzoru před provedením současného pohybu a po provedení současného pohybu. Stejně jako u první skupiny jsou tato data získána z funkce `convertFromImageCoordinates`. Vhodné využití těchto funkcí je například pro pohyb náhledu v prostoru simulace.

Mimo vykreslování aktuálního stavu simulace může vykreslovací modul také obsahovat funkce, které mohou být využívány modulem prostředí. Umožňují tomuto modulu ovlivňovat, jakým způsobem jsou uživateli v simulátoru zobrazovány různé informace. Příklad pro použití těchto funkcí může například být změna barvy pozadí dle aktuálního stavu prostředí, nebo třeba zobrazování statistik ohledně simulace v podobě textu či grafů.

Pro vytvoření nového modulu vykreslovače je nejdříve potřeba vytvořit nový Python skript, který obsahuje implementaci třídy `Renderer`. K tomuto souboru je také potřeba vytvořit odpovídající soubor s metadaty. Další součástí tohoto procesu je vytvoření validních implementací funkcí `render` a `convertFromImageCoordinates`. Následně je možné do implementace vložit funkce, které budou exportovány do uživatelského rozhraní, funkce pro modul prostředí a implementace funkcí pro zpracování uživatelských interakcí s myší. Jméno třídy implementace je zaznamenáno v souboru s metadaty.

6.5 Spouštěč buněk

Tento modul slouží ke správě a spouštění buněk, které jsou aktuálně aktivní v simulaci.

Jeho implementace obsahuje šest funkcí. Většina z těchto funkcí slouží k umožnění ovládní simulace těm modulům, které mají ke spouštěči přístup.

Spouštěč se nachází v souboru `CellExecutor.py` pod třídou nazvanou `CellExecutor`. Jeho instance je vytvářena v modulu pro uživatelské rozhraní.

Inicializační funkce tohoto modulu jako vstupní proměnné přijímá referenci na aktivní instanci prostředí a list buněk, které budou na začátku vloženy do simulace. Tento list nemusí obsahovat žádné buňky. Při inicializaci je nejdříve list buněk uložen do proměnné `cellList`. Následně je vytvořena proměnná s referencí na aktuálně zpracovávanou buňku. Tato proměnná je inicializována na prázdnou hodnotu `None`. Dále je reference na instanci prostředí uložena do proměnné `_environment`. Jako poslední krok této funkce je vytvoření proměnné `selectedCellBrainReference`, do které je ukládána reference na třídu odvozenou od třídy

CellBrain, která byla uživatelem vybrána v uživatelském rozhraní. Tato reference se využívá pro vytváření nových instancí buněk při uživatelské interakci s náhledem na simulaci v uživatelském rozhraní. Při inicializaci je nastavena na prázdnou hodnotu None.

Pro nastavení proměnné `selectedCellBrainReference` z uživatelského rozhraní slouží funkce `selectCellBrainReference`. Tato funkce pouze mění hodnotu této proměnné na vybranou referenci na třídu odvozenou od třídy `CellBrain` dodanou pomocí vstupní proměnné.

Jednou z nejdůležitějších funkcí pro třídu `CellExecutor` je `_cycleCells`. Nepřijímá žádné vstupní proměnné. Tato funkce slouží k postupnému spouštění funkčních částí všech buněk aktivních v simulaci. Nejdříve vytváří kopii listu buněk `cellList`. Tato kopie obsahuje stejné reference na buňky, které jsou v originálním listu. Úprava jedné buňky v tomto listu se tedy propíše i do druhého listu. Tento list slouží k zachování indexů při procházení listu buněk. Může se totiž stát, že uprostřed cyklu je v simulaci potřeba smazat buňku, která tento cyklus již byla spuštěna. Což by způsobilo posun indexů v listu buněk. V tomto případě se buňky odstraňují pouze z originálního listu, a kopie slouží pouze pro cyklus. Tento cyklus se ve funkci vyskytuje ihned po vytvoření kopie. Využívá počet prvků této kopie ve smyčce `for`. První krok této smyčky je získání reference na aktuálně aktivní buňku z kopie listu na aktuálním indexu, uložení reference na tuto buňku do proměnné `currentCell` a kontrola validity této buňky. Může se totiž stát, že buňka byla při simulaci smazána, a reference je tedy nevalidní. Následně je volána funkce `_cellSwitched` z aktivní instance prostředí, která prostředí informuje o změně aktivní buňky. Následuje další kontrola validity buňky, protože po zavolání této funkce se mohlo stát, že na rozhodnutí prostředí byla tato buňka smazána. Finálním krokem cyklu je zavolání funkce `execute` v aktuálně spouštěné buňce. Po dokončení tohoto cyklu je zavolána funkce prostředí `_cellsCycled` která jej informuje o tom, že byl dokončen plný cyklus spouštění buněk.

Pro přidávání nových buněk do simulace slouží funkce `addCell`. Tato funkce jako vstupní proměnnou přijímá referenci na buňku, která má do simulace být přidána. Nejdříve pomocí `for` cyklu, který prochází všechny položky listu `cellList`, kontroluje, zda v simulaci již neexistuje instance buňky která je stejná jako ta instance, kterou se funkce snaží přidat. Tato kontrola probíhá ve dvou krocích. Nejdříve se kontroluje, zda je reference mezi přidávanou buňkou a buňkou z listu stejná. Následně se kontroluje, jestli je instance třídy `cellBrain`

stejná mezi přidávanou buňkou a buňkou v listu. Pokud se najde shoda, funkce končí a buňka není přidána do simulace. Pokud se buňka v simulaci nenachází, nastává proces výběru pozice, na kterou bude nová buňka vložena. V případě, že aktuálně není žádná buňka aktivní, tedy když uživatel manuálně vkládá buňky do simulace, se buňka vkládá na začátek listu `cellList`. Pokud však existuje aktivní buňka, nová buňka je vložena do listu `cellList` před tuto aktivní buňku. Tím je zajištěno, že nově vytvořené buňka nebude nikdy spuštěna v tom stejném cyklu ve kterém byla vytvořena. Po zjištění indexu je buňka finálně vložena do listu `cellList` a funkce končí.

Odebírání buněk je řešeno pomocí funkce `removeCell`. Tato funkce jako vstupní proměnnou přijímá referenci na buňku. Nejdříve testuje, zda se tato reference vyskytuje v proměnné `currentCell`. Pokud ano, je tato proměnná nastavena na hodnotu `None`. Následně se tato funkce snaží najít a vymazat vstupní buňku z listu `cellList`. Pokud zde existuje, je z listu odstraněna. Jinak se nic nestane.

Finální funkcí v této třídě je funkce `clearCells`. Tato funkce slouží ke smazání všech buněk aktuálně aktivních v simulaci. Nastaví proměnnou `currentCell` na `None`, smaže obsah listu `cellList` a nakonec volá funkci `_executorClearedCells` z aktivní instance prostředí.

6.6 Hledač modulů

Hledač modulů slouží k procházení souborového systému, hledání souborů s metadaty, validaci a filtrování těchto souborů, hledání přidružených zdrojových souborů a získávání referencí na relevantní třídy v těchto souborech.

Implementace hledače se nachází v souboru `ModuleFinder`, který se dělí na osm funkcí. Na rozdíl od ostatních modulů tento modul neobsahuje definici třídy. Nemá tedy žádné interní proměnné, a může být využíván bez vytvoření instance třídy.

První z funkcí tohoto modulu je funkce `findAllJSONs`. Tato funkce jako vstupní proměnnou přijímá cestu v souborovém systému vedoucí k adresáři. Nejdříve tato funkce vytváří list, který slouží k ukládání všech nalezených souborů. Následně využívá smyčku `for` pro procházení všech souborů nalezených ve vybraném adresáři, jeho podadresáři a dalších rekurzivně nalezených podadresářích. Tyto soubory jsou získány pomocí funkce `walk` ze zabudované knihovny `os`. U všech nalezených souborů je kontrolována jejich přípona. Pokud

po převedení do malých písmen obsahují příponu „.json“, je jméno těchto souborů připojeno k adresám adresářů, kde se nacházejí, tyto plné cesty jsou následně ukládány do již zmíněného listu. Na konci tato funkce vrací referenci na tento list.

Funkce `findPackageJSONs` navazuje na předchozí funkci. Stejně jako `findAllJSONs` tato funkce přijímá vstupní proměnnou v podobě cesty k adresáři, který bude prohledáván. Prvním krokem této funkce je rekurzivní získání cest ke všem JSON souborům v tomto adresáři. Je to provedeno zavoláním funkce `findAllJSONs` se vstupní cestou jako vstupem pro tuto funkci. Tyto cesty jsou uloženy do listu. Je také vytvořen další list, do kterého bude ukládán obsah zpracovaných a validovaných souborů ve formě slovníků. Funkce pomocí for cyklu postupně prochází všechny nalezené cesty. Soubory nalezené pod těmito cestami se snaží otevřít, a pomocí funkce `load` z knihovny `json` převádí jejich obsah na slovník použitelný v kódu. Je provedena kontrola, zda obsah tohoto slovníku odpovídá požadované struktuře pro soubory s metadaty. Tato kontrola je provedena funkcí `validatePackageJSON`, která je detailněji popsána níže. Pokud je soubor s metadaty validní, je získána cesta k tomuto souboru, která je upravena tak, aby odkazovala na složku, ve které se soubor nachází, a k této cestě připojuje relativní cestu ke spustitelnému souboru uvedenou v souboru s metadaty. Tím je vytvořena plná cesta k tomuto souboru. Relativní cesta v získaném slovníku je touto plnou cestou nahrazena. Práce s cestou je prováděna funkcemi `dirname` a `join` v modulu `path` třídy `os`. Ve finálních krocích cyklu je slovník přidán do listu a soubor je uzavřen. Na svém konci funkce vrací tento list.

Již zmíněná funkce `validatePackageJSONs` slouží ke zjištění toho, zda je daný JSON soubor validním souborem obsahujícím metadata pro tento simulátor. Na začátku této funkce je vytvořena proměnná, která bude na konci funkcí vrácena. Tato proměnná je na začátku nastavena na hodnotu `False`, reprezentující to, že soubor validací neprošel. Po nastavení této proměnné funkce kontroluje, zda slovník obsahuje klíč `packageType`, zda se pod tímto klíčem nachází hodnota `cell`, `environment` či `renderer`, zda obsahuje klíče `package name` a `package path` a zda tyto hodnoty nejsou prázdné. Pokud jsou všechny tyto podmínky splněny, funkce pokračuje do další fáze validace. Následně se funkce dělí na dvě cesty. V případě, že typ balíčku není `cell`, probíhá kontrola existence klíče `package class` a toho, zda není jeho hodnota prázdná. Jestli tímto krokem projde, je výstupní proměnná nastavena na hodnotu `True`. Pokud však typ balíčku je `cell`, je způsob kontroly komplikovanější. Jako

první probíhá kontrola, zda ve slovníku existuje klíč `cell types`. Pokud touto kontrolou slovník projde, výstupní proměnná je nastavena na hodnotu `True`, a následují dvě další kontroly. První kontroluje, zda je velikost listu pod klíčem `cell types` menší nebo rovna nule. Pokud ano, hodnota je znovu nastavena na `False`. Jinak probíhá finální kontrola ve formě smyčky typu `for`, která prochází veškerý obsah tohoto listu. Zde se kontroluje, zda každý slovník v tomto listu obsahuje klíče `cell name` a `cell class` a zda tyto hodnoty nejsou prázdné. Pokud je jediná položka tohoto listu nevalidní, je výstupní proměnná nastavena na hodnotu `False`. Na konci je tato proměnná vrácena.

Další funkcí tohoto modulu je funkce `filterJSONsByType`. Tato funkce slouží k hledání modulů specifického typu. Přijímá dvě vstupní proměnné. První z nich je list slovníků reprezentující dříve načtené a validované soubory s metadaty. Druhá vstupní proměnná je text reprezentující typ modulu, který se snažíme v tomto listu hledat. Funkce na začátku vytvoří list, do kterého budou ukládány reference na slovníky s metadaty požadovaného typu. Funkce pomocí smyčky typu `for` prochází všechny slovníky ve vstupním listu. Nejdříve slovník vkládá do validační funkce `validatePackageJSON`, a pokud validací projde, kontroluje, zda se hodnota pod klíčem `package type` rovná vstupní hodnotě specifikující typ balíčku. Pokud tomu tak je, tento slovník je uložen do listu. Tento list je na konci funkce vrácen.

Pro další filtrování listu metadat slouží funkce `removeJSONsWithoutDependencies`. Tato funkce prochází list slovníků reprezentující metadata modulů, které filtruje tak, aby odstranil všechny moduly, ke kterým nemá modul, na kterém jsou pro svoji funkčnost závislé. Vstupní proměnnou pro tuto funkci je tento list slovníků. Jako první funkce třikrát zavolá funkci `filterJSONsByType` popsanou v předchozí sekci. Vstupní list je tak validován, a validní položky jsou rozděleny do třech skupin dle typu balíčku, který reprezentují. Funkce zbytek svých operací provádí ve třech hlavních krocích. V prvním kroku se pracuje s listem vykreslovačů. Je vytvořen list jmen tříd vykreslovačů, který je pomocí smyčky `for` naplněn daty z listu metadat vykreslovačů. Druhý krok je práce s daty o prostředích. Je vytvořen list pro jména tříd prostředí a list pro ukládání metadat validních prostředí. Pro každou položku probíhá kontrola toho, zda v ní existuje klíč `renderer class`, a zda tento klíč obsahuje nějakou hodnotu. Následně funkce zjišťuje, zda tato položka existuje v listu jmen tříd vykreslovačů. Pokud všechny tyto požadavky splňuje, jméno třídy a slovník s metadaty jsou uloženy do odpovídajících listů. Ve finálním kroku se pracuje s balíčky buněk. Je vytvořen

list na validní metadata těchto balíčků. Následně je procházen list se všemi metadaty o balíčcích buněk smyčkou `for`. Funkce kontroluje, zda slovník obsahuje klíč `environment class`, a zda hodnota pod tímto klíčem není prázdná. Následně kontroluje, zda se tato hodnota vyskytuje v listu jmen tříd prostředí. Pokud tyto kontroly úspěšně projdou, jsou metadata uložena do listu. Všechny tři vytvořené listy s validními metadaty jsou ve formě tuple se třemi položkami na konci funkce vráceny.

Finální blok třech funkcí slouží k extrakci dat ze zdrojových skriptů a dynamické načítání relevantních tříd z těchto skriptů pro použití v simulátoru. Tyto funkce jsou `loadRenderer`, `loadEnvironment` a `loadCellPack`. Funkce pro načítání vykreslovačů a prostředí jsou stejné. Z toho důvodu funkce `loadEnvironment` pouze volá funkci `loadRenderer`. Rozdělení do dvou funkcí je pouze pro zpřehlednění částí kódu kde jsou tyto funkce volány. Funkce přijímá vstupní proměnnou slovník s metadaty ohledně relevantního modulu, který je načítán. První krok této funkce je zjištění informací o skriptu na lokaci uvedené ve vstupním slovníku. Zjišťování těchto informací je prováděno funkcí `spec_from_file_location` z modulu `util` knihovny `importlib`. Tato funkce jako vstupní proměnné přijímá textový řetězec, který je použitý jako jméno dynamicky načítaného modulu, a textový řetězec reprezentující cestu k relevantnímu souboru. Oba vstupní textové řetězce jsou získány ze slovníku s metadaty. Pro první se používá jméno balíčku dostupné pod klíčem `package name`. Druhý zase používá cestu k balíčku dostupnou pod klíčem `package path`. V případě, že soubor s modulem neexistuje nebo neobsahuje validní data reprezentující Python skript, tato funkce končí, a je vrácena hodnota `None`. V případě že jsou informace o souboru úspěšně získány, nastává dynamické načtení tohoto modulu pomocí funkce `module_from_spec` z modulu `util` knihovny `importlib`. Finální část funkce je „zabalena“ do bloku `try-except`. Nejdříve se funkce snaží spustit nově načtený modul funkcí `exec_module` z modulu `loader` knihovny `importlib`, čímž načte všechny datové struktury a funkce v tomto modulu. Následně se pomocí zabudované funkce `getattr` snaží získat referenci na třídu s požadovaným modulem. První vstupní proměnnou této funkce je samotná reference na modul. Druhou proměnnou je jméno požadované třídy získané ze slovníku s metadaty pod klíčem `package class`. Pokud proces proběhne bez problémů, funkce tuto vrátí tuto referenci na třídu. V případě že nastane problém, program vyvolá `exception`, a funkce vrátí hodnotu `None`.

Funkce `loadCellPack` funguje velmi podobně jako předchozí funkce. Musí však praco-

vat s tím, že balíček buněk může obsahovat několik různých načítaných tříd. Prvním z rozdílů je hned na začátku funkce. Je potřeba vytvořit prázdný list, do kterého se tyto reference na načítané třídy budou vkládat. Další změnou je úprava všech návratových funkcí takovým způsobem, aby vždy vracely tento list. Poslední změnou je vložení volání funkce `getattr` do smyčky `for` pro každou položku listu ve slovníku s metadaty pod klíčem `cell types`. Vstup s klíčem `package class` ze slovníku s metadaty byl nahrazen klíčem `cell class` ze slovníku pro individuální typ buňky. Po každém volání funkce `getattr` je získaná hodnota uložena do listu vytvořeného na začátku této funkce spolu s metadaty pro tento typ buňky pomocí tuple.

6.7 Uživatelské rozhraní

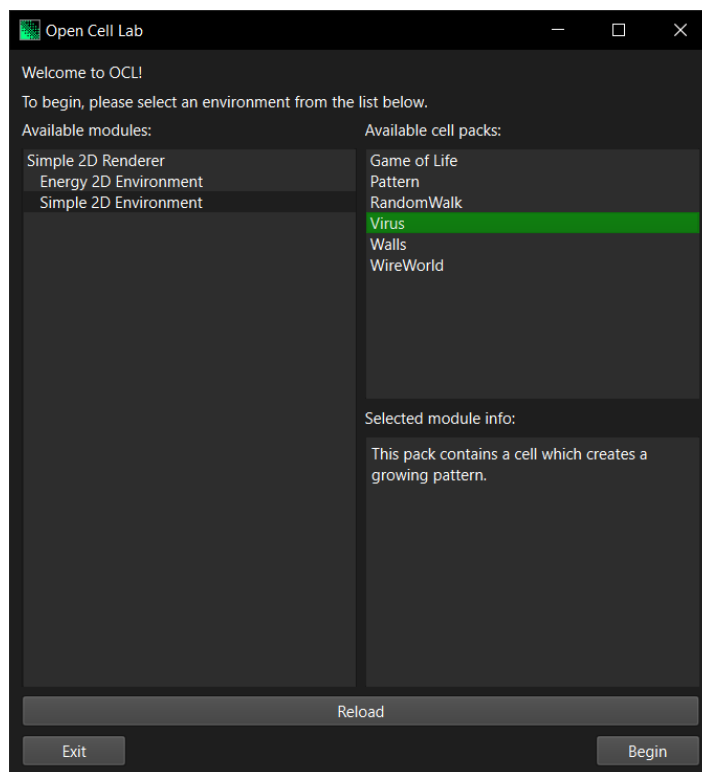
Definice uživatelského rozhraní se nachází v souboru `UserInterface.py`. Dělí se na dvě třídy. První reprezentuje uvítací obrazovku, na které jsou uživateli zobrazovány nainstalované moduly které může spustit. Druhý reprezentuje hlavní obrazovku, na které uživatel vidí náhled na simulaci, a kde může s touto simulací může interagovat. Samotná okna a prvky v těchto oknech jsou definovány pomocí knihovny `PyQt6`.

6.7.1 WelcomeScreen

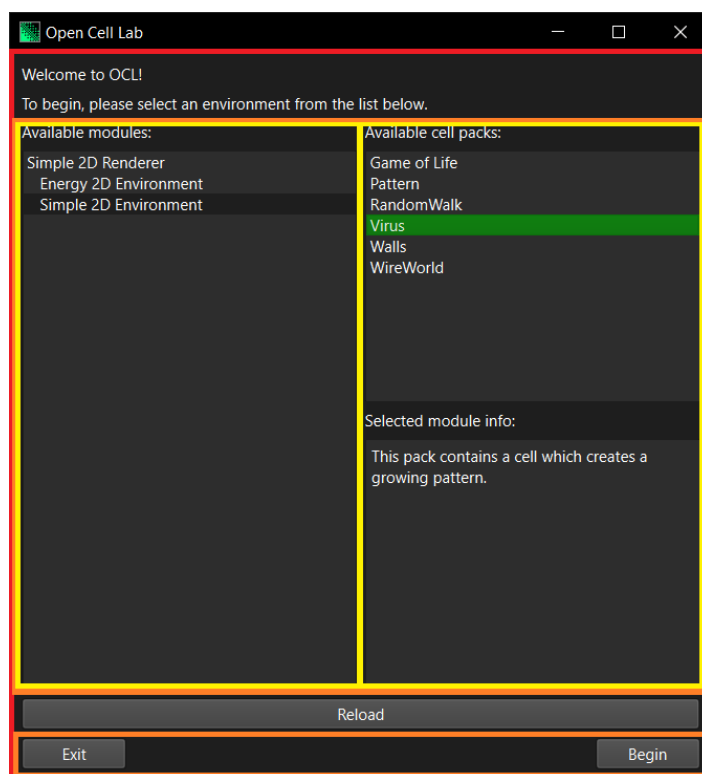
Uvítací obrazovka, viditelná na obrázku 6.1, je reprezentovaná třídou `WelcomeScreen`. Prvky této obrazovky jsou rozděleny do několika layoutů, které definují, jakým způsobem jsou tyto prvky uspořádány. Rozložení těchto layoutů lze vidět na obrázku 6.2.

Všechny elementy této obrazovky se nacházejí ve vertikálním layoutu nebo v dalších layoutech vnořených do tohoto layoutu. Vnější layout je na ilustraci zvýrazněn červenou barvou. Prvním elementem je zde textové pole s nápisem „Welcome to OCL!“, který slouží jako jednoduchý pozdrav, který se uživateli zobrazí při každém spuštění aplikace. Pod tímto textovým polem je druhé textové pole s instrukcemi ohledně výběru konfigurace s textem „To begin, please select an environment from the list below.“. Následující prvek v tomto layoutu je další, horizontální, layout.

Tento vnitřní layout, na ilustraci zvýrazněn oranžovou barvou v sobě obsahuje dva vertikální layouty, označené barvou žlutou. První z nich obsahuje textové pole s nápisem „Available modules:“ a list, ve kterém jsou vypsané vykreslovače a jim přiřazené moduly prostředí. Obsah tohoto listu je načten při prvním spuštění aplikace. Uživatel může na položky tohoto



Obrázek 6.1 Uvítací obrazovka simulátoru (vlastní tvorba)



Obrázek 6.2 Rozložení uvítací obrazovky simulátoru (vlastní tvorba)

listu klikat, čímž je vybírá. Druhý vertikální layout obsahuje pole s textem „Available cell packs:“. Pod tímto textem je list, který slouží ke zobrazení aktuálně dostupných balíčků buněk pro prostředí, které je aktuálně vybrané v předchozím listu. Pokud není nic vybrané, nebo je vybrán vykreslovač, je tento list prázdný. Stejně jako u předchozího listu může uživatel vybírat prvky i z tohoto listu. Pod tímto listem je další textové pole s textem „Selected module info:“, pod kterým je jiný typ textového pole se stejnou barvou pozadí jako předchozí listy, a s možností výběru a kopírování textu ze strany uživatele. V případě, že je vybrán jakýkoliv modul, jak už vykreslovač či prostředí z prvního listu, tak balíček buněk z druhého listu, tak se v tomto textovém poli zobrazuje popis tohoto balíčku získaný z metadat. Pokud popis neobsahuje, nebo není vybrán žádný balíček, pole zůstává prázdné.

Po vnitřních layoutech pokračuje obsah layoutu vnějšího. Jsou následovány tlačítkem s textem „Reload“, které slouží v případě změn ve složce obsahující externí moduly. Tímto tlačítkem jsou znovu načteny a zobrazeny.

Finálním elementem vnějšího layoutu je další horizontální layout, který je také na obrázku označený oranžovou barvou. Tento layout obsahuje dvě tlačítka. První z nich je tlačítko „Exit“, které po zmáčknutí ukončí aplikaci. Druhé je tlačítko „Begin“. Toto tlačítko je zmáčknutelné pouze tehdy, kdy je v levém listu s vykreslovači a prostředími vybráno prostředí. Jinak je tlačítko zašedlé, a nelze zmáchnout. Pokud je zmáchnuté, program přechází na hlavní obrazovku se simulací, která využívá vybrané moduly.

V kódu je toto okno definováno jako třída odvozená od třídy `QMainWindow`. Již zmíněné prvky okna jsou definovány v inicializační funkci této třídy. Layouty jsou instance tříd `QHBoxLayout` a `QVBoxLayout`. Textové pole jsou instancemi `QLabel` a `QTextEdit`. Listy jsou `QListWidget` a tlačítka `QPushButton`. Kvůli tomu, že layout nemůže být centrálním prvkem okna, je venkovní layout vložen do instance třídy `QWidget`. Všechny zmíněné třídy pocházejí z modulu `QtWidgets` knihovny `PyQt6`.

Mimo instancování těchto tříd obstarává inicializační funkce další věci, které jsou potřebné pro správný běh programu. Vytváří proměnné, které slouží k uchovávání referencí na metadata aktuálně zvolených modulů. Inicializační funkce také propojuje interaktivní prvky s funkcemi, které jsou volány po provedení relevantních interakcí. Na konci této funkce je volána jiná funkce, která obstarává naplnění prvků okna daty o modulech.

Tato třída obsahuje deset funkcí, které buďto slouží ke zpracování interakcí s interaktiv-

ními prvky, nebo jako asistence při tomto procesu.

Populaci listu modulů zajišťuje funkce `populateModuleList`. Nejdříve pomocí kombinace funkcí `abspath` a `dirname` modulu `path` zabudované knihovny `os` zjistí, v jaké složce se aktuálně spouštěný skript nachází. Následně je pomocí funkce `join` stejného modulu k cestě přidáno jméno podadresáře `packages`, ve kterém se nacházejí externí moduly. Po získání cesty jsou pomocí funkce `findPackageJSONs` modulu `ModuleFinder` v tomto adresáři a ve všech jeho podadresářích nalezeny a načteny všechny soubory s metadaty externích modulů. Tato data jsou uloženy do proměnné. Poté je vytvořena nová prázdná instance listu jako proměnná instance, určená pro ukládání referencí na tato data v požadovaném pořadí. Aktuální obsah listu, který se zobrazuje uživateli, je vymazán. Pomocí funkce `filterJSONsByType` modulu `ModuleFinder` je list nalezených modulů vyfiltrován na list vykreslovačů a prostředí. Zbytek funkce se nachází ve smyčce `for`, která prochází všechna nalezená data o vykreslovačích. Nejdříve je do seřazeného listu vložena reference na tato data. Následně je jméno tohoto modulu vloženo, nalezeno pod klíčem `package name` vloženo do prvku pro zobrazení listu uživateli. Jméno třídy balíčku pod klíčem `package class` je uloženo do dočasné proměnné. Po získání těchto dat následuje další smyčka `for` vnořená do předchozího cyklu. Tentokrát slouží pro procházení metadat prostředí. Probíhá kontrola názvu třídy vykreslovače a prvku `renderer class` prostředí. Pokud jsou shodné, jsou data vykreslovače přidána do listu modulů, a jméno tohoto prostředí, klíč `package name` je uloženo do listu pro zobrazení uživateli. Před jeho jménem jsou přidány čtyři mezery z toho důvodu, aby byly vykreslovače a prostředí od sebe v listu vizuálně odděleny.

V případě změny výběru položky ve vizuálním listu modulů je potřeba provést několik kroků. To je zajištěno funkcí `moduleSelectionChanged`. Nejdříve je volána funkce `unselectCellPack`, popsání níže. Pokračuje získání indexu nově vybrané položky pomocí vstupní proměnné. Tímto indexem je z listu modulů získána reference na nově vybraný modul. V případě, že data tohoto modulu obsahují klíč `package description`, je tento popis vložen do textového pole s informacemi. Pokud ne, je obsah tohoto pole smazán. Následuje kontrola toho, zda se jedná o modul prostředí. Pokud ano, je proměnná pro aktuálně vybrané prostředí nastavena na tento modul. Protože každé prostředí má svůj vykreslovač v listu modulů někde nad sebou, je tento list procházen od aktuálního indexu nahoru, první nalezená metadata vykreslovače jsou uložena do proměnné pro aktuálně vybraný vykreslovač. Finálně

je povoleno zmáčknutí tlačítka na spuštění simulace, a je zavolána funkce pro vyplnění listů buněk. V případě, že se jedná o výběr vykreslovače jsou proměnné na aktuálně vybrané prostředí, vykreslovač a buňky nastavené na hodnotu None. Tlačítko je vypnuté a vizuální list buněk je vymazán.

Vyplnění listů buněk má na starost funkce `populateCellPackList`. Jako vstupní proměnnou přijímá metadata pro vybrané prostředí. Jako první je prováděna kontrola, zda se pod klíčem `package type` nachází hodnota `environment`. Pokud tomu tak není, funkce končí. Následuje vymazání vizuálního listu buněk a vytvoření nové prázdné instance korespondujícího listu na metadata buněk ve stejném pořadí. Následuje získání cesty k aktuálně spouštěného souboru, které je prováděno stejným způsobem jako ve funkci `populateModuleList`. Stejným způsobem je také získán list modulů, který je vyfiltrován pouze na moduly s buňkami. List balíčků buněk je následně pomocí `for` cyklu procházen. Při tomto procesu se kontroluje, zda hodnota pod klíčem `environment class` je stejná jako hodnota `package class` vstupních metadat prostředí. Pokud ano, jméno balíčku buněk je přidáno do vizuálního listu, a metadata jsou přidána do listu buněk.

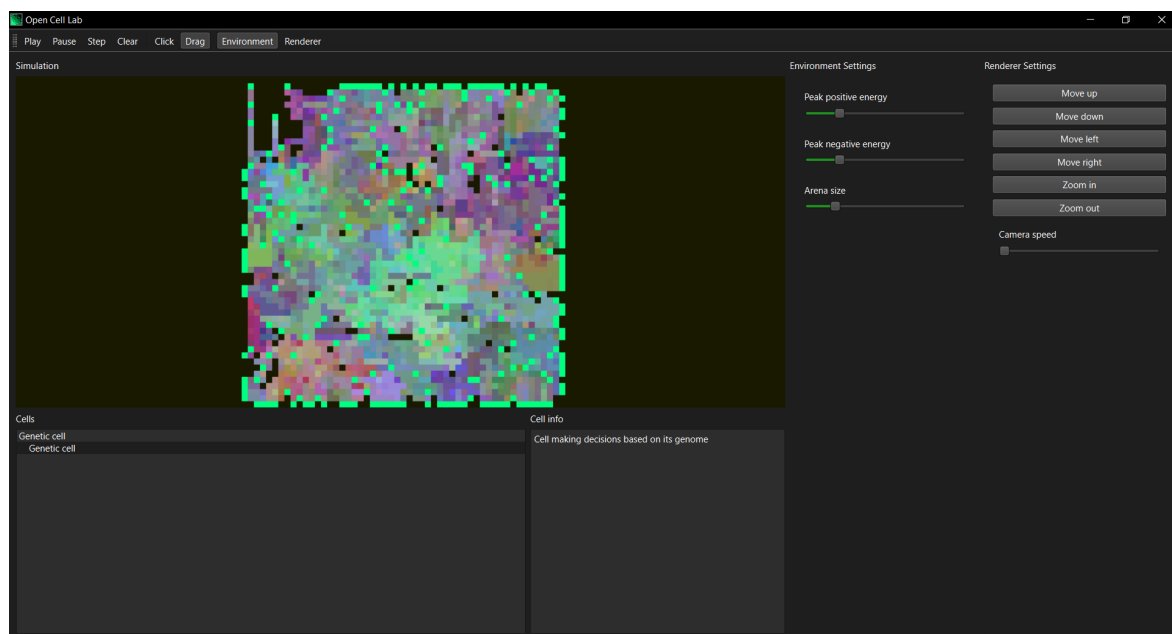
Interakci s vizuálním listem buněk zpracovává funkce `cellPackSelectionChanged`. Nejdříve je ze vstupní proměnné zjištěn index položky, na kterou uživatel kliknul. Následně jsou získána metadata, která se nachází na listu buněk na tomto indexu. Je kontrolováno, zda tato metadata obsahují klíč `package description`. Pokud ano, hodnota na tomto klíči se vypíše uživateli do textového pole. Jinak se obsah textového pole smaže.

Nastavení textu je zjednodušeno funkcí `setModuleInfoText`, která pouze volá funkci relevantní interní proměnné pro změnu textu.

Mezi další funkce, které zjednodušují práci ostatním funkcím, se řadí `unselectModule` a `unselectCellPack`. Tyto dvě funkce pouze ruší výběr ve vizuálních listech externích modulů.

Kliknutí na tlačítko pro znovunačtení externích modulů řeší funkce `reloadClicked`. Tato funkce zruší výběr položek z listů externích modulů, maže všechny prvky v těchto listech, maže obsah textového pole s informacemi o vybraném modulu, volá funkci pro populaci listu modulů a vypíná tlačítko na spuštění simulace. Tato funkce je volána na konci inicializační funkce.

Tlačítko pro vypnutí programu je propojeno s funkcí `exitClicked`. Tato funkce zavře



Obrázek 6.3 Hlavní obrazovka simulátoru (vlastní tvorba)

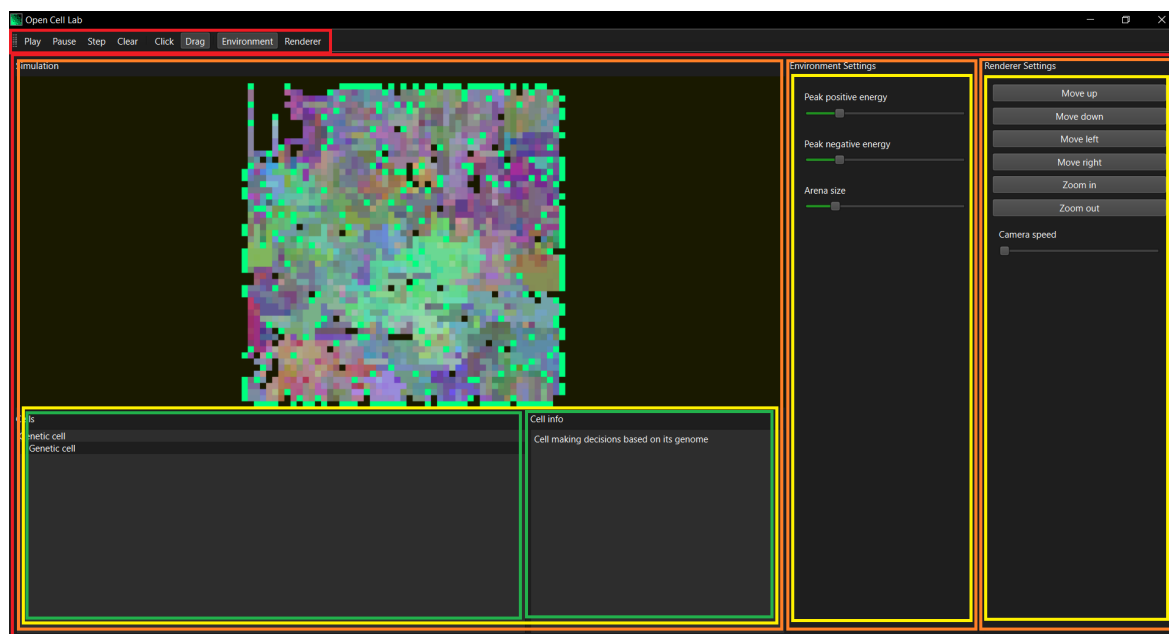
okno aplikace, čímž ukončí její běh.

Zapnutí simulace pomocí tlačítka je řešeno funkcí `beginClicked`. Tato funkce vytvoří novou instanci třídy `MainScreen`, do kterého jako vstupní proměnné pošle svoje interní funkce s vybraným vykreslovačem, prostředím a listy buněk. Tuto instanci nastaví jako svůj centrální prvek, a zvětší svoji velikost. S tímto přebírá veškerou kontrolu nad oknem instance třídy `MainScreen`.

6.7.2 MainScreen

Hlavní obrazovka, viditelná na obrázku 6.3, je reprezentovaná třídou `MainScreen`. Stejně jako uvítací obrazovka je rozdělena na několik layoutů. Mimo tyto layouts také obsahuje toolbar s tlačítky. Rozložení těchto prvků lze vidět na obrázku 6.4.

Toolbar se vyskytuje na vrchu okna. Obsahuje tlačítka na správu běhu simulace a na definici typu uživatelské interakce s náhledem na simulaci. Pro ovládání simulace slouží tlačítka „Play“, které spustí automatický průběh simulace, „Pause“, které pozastaví automatický průběh simulace, „Step“, které provede pouze jeden krok simulace a „Clear“, které smaže všechny buňky aktuálně aktivní v simulaci. Tlačítka pro definici interakce se dělí na dvě skupiny. První dvě tlačítka, „Click“ a „Drag“ definují, zda uživatel chce interagovat pouze klikáním, nebo zda chce interagovat kliknutím a tažením. Druhá dvě tlačítka, „Environment“ a „Renderer“ definují, zda uživatel interaguje s prostředím či s vykreslovačem.



Obrázek 6.4 Rozložení hlavní obrazovky (vlastní tvorba)

Zbytek okna je v jednom velkém horizontálním layoutu. Tento layout obsahuje tři vertikální layouty. Jeden pro simulaci a výběr buněk, jeden pro exportní funkce z prostředí a jeden pro exportní funkce z vykreslovače. Tyto layouty jsou mezi sebou v poměru 3:1:1.

Simulační layout obsahuje textové pole s textem „Simulation“, místo pro bitmapu sloužící jako náhled do simulace a horizontální layout pro list buněk a informace o vybraném balíčku.

Tento layout obsahuje dva další vertikální layouty v poměru 3:2. První z nich obsahuje textové pole s textem „Cells“ a interaktivní list, do kterého se vypisují aktuálně načtené balíčky buněk a buňky v těchto balíčcích. Druhý obsahuje text „Cell info“ a větší textové pole do kterého se vypisují informace o aktuálně vybrané buňce či balíčku buněk.

Oba layouty pro exportní funkce fungují stejně. Obsahují textové pole s nápisem „Environment Settings“ či „Renderer Settings“. Následně obsahují další vertikální layout, který je při vytvoření okna dynamicky naplněn dle exportních funkcí definovaných v modulu prostředí či ve vykreslovacím modulu. Zde se mohou objevovat tlačítka, posuvníky, checkboxy, radiobuttony či boxy pro zadávání čísel.

V kódu je tato třída také odvozena od třídy `QMainWindow`. Obsahuje stejné typy prvků jako uvítací obrazovka. Mimo tyto prvky je toolbar instancí třídy `QToolBar`. Tlačítka tohoto toolbaru jsou instancemi `QAction`. Tlačítka pro výběr způsobu a cíle interakce jsou zařazena do skupiny definované třídou `QActionGroup`. Místo pro bitmapu reprezentující si-

mulaci je definováno třídou `SimulationLabel`, která je definovaná v kódu tohoto modulu, odvozená od třídy `QLabel` a popsána níže. Všechny zmíněné třídy mimo `SimulationLabel`, `QActionGroup` a `QAction` pocházejí z modulu `QtWidgets` knihovny `PyQt6`. `QActionGroup` a `QAction` pocházejí z modulu `QtGui` stejné knihovny.

Třída definující hlavní okno má, stejně jako třída uvítacího okna, několik funkcí, které definují různé reakce na uživatelské interakce, nebo tento proces ulehčují.

Inicializační funkce je především určena k vytvoření instancí prvků okna a propojování interaktivních elementů s funkcemi, které na tyto interakce reagují. Venkovní layout je vložen do instance třídy `QWidget`. Mimo definice těchto prvků je na začátku inicializační funkce vytvořena instance třídy `QTimer` modulu `QtCore` knihovny `PyQt6`. Tento prvek slouží jako časovač, který s určitou periodou spouští danou funkci. V uživatelském prostředí slouží k tomu, aby automaticky prováděl kroky simulace. Na svém konci volá inicializační funkce funkci `reload`, popsanou níže.

Pro načtení a inicializaci externích modulů slouží funkce `reload`. Jako vstupní proměnnou přijímá metadata pro vykreslovač, prostředí a list buněk. Tato data jsou získána z uvítací obrazovky, předána inicializační funkci při vytvoření instance této třídy a následně předána funkci `reload`. Tato funkce nejdříve vytvoří novou interní proměnnou pro ukládání nových instancí časovačů. Tyto časovače slouží pro práci s exportními funkcemi, a jsou popsány níže. Následně jsou pomocí funkcí `loadRenderer` a `loadEnvironment` z modulu `moduleFinder` získány instancovatelné reference na třídy reprezentující načtené moduly vykreslovače a prostředí. V případě, že dojde při načítání těchto modulů k chybě, je volána funkce `loadingFailed`, popsána níže. Funkce je následně ukončena. V případě úspěšného získání referencí jsou instance těchto prvků vytvořeny. Nejdříve je vytvořena instance vykreslovače se vstupními proměnnými udávajícími šířku a výšku plochy pro bitmapy, získané z tohoto prvku. Následně je vytvořena instance prostředí s vykreslovačem jako vstupní proměnnou. Po vytvoření těchto instancí se vytváří instance třídy `CellExecutor`, která přijímá jako vstupní proměnnou prostředí a prázdný list buněk. Finálním inicializačním krokem je zavolání funkce `_setExecutor` v prostředí, která dokončí „cyklus“ propojení těchto tříd. Následně probíhá vyplnění listu volitelných buněk a jejich balíčků. Je vytvořena prázdná instance listu, do kterého budou ukládány instancovatelné reference na třídy buněk. Je také vyčištěn interaktivní list v uživatelském rozhraní. Buňky jsou procházeny balíček po balíčku

pomocí smyčky `for`. Nejdříve je pomocí funkce `loadCellPack` modulu `moduleFinder` získán list obsahující instancovatelné reference buněk aktuálně zpracovávaného modulu. Pokud je list prázdný, balíček je přeskočen. Následně jsou do listu buněk vložena data reprezentující balíček buněk, se kterým pracujeme. Je vložen ve formě tuple se dvěma položkami. První obsahuje hodnotu `None` a druhá metadata pro tento balíček. Hodnota `None` je zde vložena z toho důvodu, aby byla zachována struktura, ve které jsou získávány instance tříd buněk. Jméno balíčku, získané z metadat pod klíčem `package name`, je vloženo do interaktivního listu. Finálně jsou nalezené instance buněk procházeny smyčkou `for`, ve které se reference na třídy spolu s metadaty získanými z funkce `loadCellPack` ukládají do listu buněk, a jména buněk z metadat pod klíčem `cellName` se ukládají do interaktivního listu. Pro vizuální oddělení od balíčků buněk jsou před jméno vloženy čtyři mezery. Finální část funkce `reload` obsahuje načítání exportních funkcí pro vykreslovač a prostředí. V obou případech je vytvořena nová interní proměnná slovníku obsahujícího skupiny radio tlačítek. Následně se pomocí smyčky `for` prochází list exportních funkcí definovaný v dané třídě. Pro tyto prvky listu je volána funkce `buildExportElement` popsána níže. Následně je prvek přidán do relevantní části uživatelského rozhraní.

V případě, že dojde k chybě v načítání modulů, je volána funkce `loadingFailed`. Tato funkce do konzole vypisuje chybovou hlášku.

Uživatelská interakce s listem obsahujícím moduly a buňky je zpracována za pomoci funkce `cellListSelectionChanged`. V této funkci je nejdříve pomocí vstupní proměnné zjištěn index prvku, který uživatel vybral. Následně je pomocí tohoto indexu získána reference na tento balíček či buňku. Pokud obsahuje popis, který je reprezentován buďto klíčem `package description` nebo `cellDescription` je vložen do sousedního textového pole. Dle toho je na konci je volána funkce `selectCellBrainReference` ze spouštěcího modulu s prvním prvkem tuple jako vstupem. Funkce tedy dostane hodnotu `None`, pokud se jedná o modul, a referenci na třídu, pokud se jedná o buňku.

Sestavení interaktivních prvků, které reprezentují exportní funkce, se provádí ve funkci `buildExportElement`. Tato funkce obsahuje dvě vstupní proměnné. První reprezentuje instanci třídy reprezentující exportní funkci, popsanou v kapitole 6.8, a druhá je logická hodnota uvádějící, zda se jedná o funkce pro vykreslovač či prostředí. Na začátku probíhá kontrola, zda namísto instance exportní funkce neobdržela tato funkce textový řetězec. Pokud

tomu tak je, vrací instanci třídy `QLabel` s tímto řetězcem. Jinak je z instance exportní funkce získán typ elementu, který je následně porovnáván s existujícími typy elementů. Tyto elementy jsou definovány v modulu `ExportFunctions`. Pokud je to element typu `BUTTON`, je vytvořena instance třídy `QPushButton` se vstupním textem, a zmáčknutí je navázáno na zavolání asociované funkce. U elementu typu `REPEATINGBUTTON` se tato instance také vytváří, je však navázána na časovač s vybraným intervalem, který je zapnutý při kliknutí na tlačítko a vypnutý při puštění tohoto tlačítka. Aktivace tohoto časovače je navázána na asociovanou funkci. `RADIOBUTTON` vytvoří instanci třídy `QRadioButton` se vstupním textem. Stejně jako u klasického tlačítka je kliknutí na toto tlačítko propojeno se zavoláním asociované funkce. Toto tlačítko je spojeno s dalšími radio tlačítky. V extra proměnných je definováno jméno skupiny, do které toto tlačítko patří. Pokud tato skupina není ve slovníku skupin, je vytvořena instance `QButtonGroup`, která je do něj pod tímto jménem přidána. Pokud ve slovníku je, je tato reference ze slovníku získána, a nové tlačítko je ke skupině pouze připojené. Prohledává se pouze slovník spojený danou skupinou exportních funkcí, tedy dle toho, jestli se jedná o vykreslovač nebo o prostředí. První tlačítko každé skupiny je automaticky zmáčknuté. `CHECKBOX` vytvoří novou instanci `QCheckBox` se vstupním textem, který po kliknutí zavolá přidruženou funkci. Tato tlačítka nejsou defaultně zakřížkovaná. `SLIDER` vytvoří instanci `QWidget`, do které je vložen `QLabel` se vstupním textem a `QSlider`. Minimální, maximální a startovní hodnota jsou nastaveny ze vstupních proměnných. Změna hodnoty je navázána na přidruženou funkci. `SPINBOX` funguje stejně jako slider, jedinou změnou je nahrazení třídy za `QSpinBox`. Všechny tyto elementy jsou v případě vykreslovače také napojeny na funkci `updateSimulationView`. To zajišťuje, že po aktivaci funkcí propojených s těmito elementy se vždy aktivuje překreslení náhledu na simulaci.

Aktivace časovače definovaného v inicializaci řeší funkce `simulationTimerTriggered`. Tato funkce volá funkci `_cycleCells` ze spouštěče a funkci `updateSimulationView`.

Tlačítko Step je řešeno funkcí `stepClicked`. Vypíná simulační časovač a následně zavolá funkci `simulationTimerTriggered`.

Vymazání buněk tlačítkem „Clear“ je prováděno funkcí `clearClicked`. Zastaví časovač, zavolá funkci `clearCells` ze spouštěče a finálně volá funkci `updateSimulationView`.

Při změně velikosti okna je volána funkce `resizeEvent`. Tato funkce je v základní třídě již definována, ale tato třída tuto definici upravuje. Nejdříve je zavolána rodičovská funkce

resizeEvent. Následně je ve vykreslovači zavolána funkce `_setOutputResolution` určená pro aktualizaci na nové rozlišení náhledu na simulaci. Finálně je zavolána funkce `updateSimulationView`.

Samotná funkce `updateSimulationView` volá funkci `setPixmap`, pocházející z odvozené třídy `simulationImageLabel`. Pro získání vstupní proměnné je volána funkce `render` z vykreslovače s aktuálním listem buněk ze spouštěče jako vstupem. Následně jsou bajty z této funkce převedeny funkcí `fromData` na třídu `QImage`, ze které je funkcí `fromImage` získána instance třídy `QPixmap`.

Pro zjištění toho, kterému modulu mají být poslány uživatelské interakce myši, slouží funkce `determineReceiver`. Tato funkce zjistí, které z výběrových tlačítek je aktivované, a dle toho vrátí referenci na aktivní instanci vykreslovače či prostředí.

Proces konvertování souřadnic je prováděn za pomoci funkce `processCoordinates`. Tato funkce přijímá jako vstupní proměnné dvoudimenzionální souřadnice, a dle toho, zda je aktivovaná interakce s prostředím či vykreslovačem jsou touto funkcí vráceny buďto souřadnice konvertované funkcí `convertFromImageCoordinates` z vykreslovače, nebo pouze čisté souřadnice.

Kliknutí uživatele na náhled simulace pomocí myši řeší funkce `imageLeftClicked`, `imageRightClicked` a `imageMiddleClicked`. Nejdříve je získána reference na příjemce této interakce pomocí funkce `determineReceiver`. Jsou získané vstupní souřadnice zpracovány funkcí `processCoordinates`. Pokud jeden z těchto procesů selže, funkce končí. Tato funkce je volána i na začátku kliknutí a tažení, je tedy vytvořena interní proměnná, která je nastavena na zpracované souřadnice. Pokud se nejedná o kliknutí a tažení, je zavolána funkce `_primaryClick`, `_secondaryClick` či `_tertiaryClick` aktuálně ovládaného modulu se zpracovanými souřadnicemi jako se vstupní proměnnou. Na konci je zavolána funkce `updateSimulationView`.

Kliknutí a tažení je zpracováno funkcemi `imageLeftDragged`, `imageRightDragged` a `imageMiddleDragged`. Funkce nejdříve zjistí, zda je aktivní mód tažení. Pokud není, funkce končí. Poté zjistí příjemce, a zpracuje souřadnice stejně, jako funkce pro kliknutí. Následně je volána funkce `_primaryDrag`, `_secondaryDrag` či `_tertiaryDrag` příjemce. Jako vstupní proměnné pro tuto funkci jsou poskytovány předchozí souřadnice a aktuální souřadnice. Aktuální souřadnice jsou po zavolání této funkce zkopírovány do proměnné pro ukládání před-

chozích souřadnic. Na konec je volána funkce `updateSimulationView`.

Poslední funkcí této třídy je `showEvent`. Tato funkce je také přepsanou verzí funkce již definované touto třídou. Na začátku je volána rodičovská verze této funkce. Následně je pomocí funkce `_setOutputResolution` vykreslovače nastaveno rozlišení. Finálně je zavolána funkce `updateSimulationView`. Tato funkce je volána až po zobrazení a inicializaci okna. Získává tedy správnou velikost náhledu na simulaci, kterou používá jako vstupní proměnnou pro volanou funkci.

6.7.3 SimulationLabel

Finální třídou definovanou v tomto modulu je třída `SimulationLabel`, která je odvozená od třídy `QLabel`. Slouží jako redefinice této třídy určená pro jednodušší práci s uživatelskými interakcemi a pro zobrazování obrázků.

Tato upravená verze třídy definuje šest signálů, které vysílá při uživatelských interakcích myši.

V redefinici inicializační funkce vytváří tři nové proměnné, které obsahují logickou hodnotu definující, zda uživatel aktuálně tahá myši se zmáčknutým tlačítkem.

Redefinovaná funkce `sizeHint` pro zjištění velikosti byla upravena tak, aby vracela hodnotu výšky a šířky 100 v případě, že jsou tyto hodnoty menší nebo rovny nule. Jinak standardně vrací výšku a šířku.

Funkce `mousePressEvent` byla redefinována tak, aby vyslala relevantní signál pro zmáčknuté tlačítko myši. Volá jak funkci pro kliknutí, tak funkci pro tažení. Také nastavuje přidruženou proměnnou tažení na `True`.

Při pohybu myši je volána redefinovaná funkce `mouseMoveEvent`. V případě, že je některá z proměnných pro tažení nastavena na `True`, vysílá se relevantní signál pro tažení s tímto tlačítkem.

Poslední redefinovanou funkcí je `mouseReleaseEvent`, která nastavuje proměnné pro tažení na `False` v případě, že je jejich přidružené tlačítko uživatelem puštěno.

6.8 Exportní funkce

Modul reprezentující exportní funkce se nachází v souboru `ExportFunctions.py`, a dělí se na dvě třídy.

První z těchto tříd je `ControlElement`, která je odvozená od zabudované třídy `Enum`. Obsahuje pouze sadu očíslovaných proměnných, které definují, jakým elementem bude exportní třída v uživatelském rozhraní reprezentována. Tyto hodnoty jsou `BUTTON`, `REPEATINGBUTTON`, `RADIOBUTTON`, `CHECKBOX`, `SLIDER` a `SPINBOX`. Každá z nich reprezentuje jeden element z knihovny `PyQt6`. Jejich interpretace je popsána v předchozí podkapitole.

Druhá třída je `ExportFunctions`. Tato třída funguje pouze jako datová struktura pro strukturovaný způsob ukládání exportních funkcí. Obsahuje tedy pouze inicializační funkci. Tato funkce jako vstupní proměnné přijímá referenci na funkci, jméno funkce, typ elementu definovaný třídou `ControlElement` a také list s dalšími argumenty relevantními pro exportní funkce. V případě, že funkce argumenty nepotřebuje, je tato proměnná defaultně nastavena na prázdný list. V případě, že chceme předat pouze textový řetězec, proměnná `controlElement` slouží jako místo, kde je vložen. Všechny tyto vstupní proměnné jsou nastaveny jako interní proměnné, ke kterým je zvenčí přístup.

Další argumenty se liší podle toho, jakou funkci chceme definovat. Funkce reprezentované jako `text`, `BUTTON` a `CHECKBOX` žádné další proměnné nemají. `REPEATINGBUTTON` obsahuje pouze časovou prodlevu mezi aktivacemi tlačítka v milisekundách. `RADIOBUTTON` obsahuje jméno skupiny tlačítek. `SLIDER` a `SPINBOX` přijímají dolní hranici hodnot, horní hranici hodnot a startovní hodnotu.

6.9 Main

Hlavní vstupní bod do simulátoru se nachází v souboru `main.py`. Tento skript slouží k vytvoření instance okna aplikace.

Na začátku skriptu je vytvořena instance třídy `QApplication` modulu `QtWidgets` knihovny `PyQt6`. Následně na využití funkce `setStyle` z této instance pro nastavení témat aplikace na `Fusion`. Následuje vytvoření instance třídy `WelcomeScreen` modulu `UserInterface`. Je vytvořena instance třídy `QIcon` modulu `QtGui` knihovny `PyQt6`. Pomocí modulu `path` zabudované třídy `os` je získána absolutní cesta ke složce, ve které se nacházejí ikony aplikace. Získání cesty je provedeno kombinací funkcí `abspath`, `dirname` a `join`. K této třídě jsou pomocí funkce `join` připojeny cesty k obrázkům různé velikosti. Tyto kombinace fungují jako vstupy do funkce `addFile` z instance `QIcon`. Tato ikona je následně vložena do aplikace pomocí funkce `setWindowIcon` instance třídy `QApplication`. Jméno aplikace, „Open

Cell Lab“, je nastaveno pomocí funkce `setWindowTitle` v instanci třídy `WelcomeScreen`. Finálně je okno zobrazeno pomocí funkce `show`. Pokud je okno zavřeno, volá se funkce `exit` interní třídy `sys`. Tím skončí běh celé aplikace.

7 PŘÍKLADY MODULŮ

K simulátoru je také potřeba vytvořit řadu modulů, které slouží jako funkční ukázka toho, co je v něm možné vytvořit a také toho, jakým způsobem se správně moduly tvoří. Tato kapitola popisuje, k čemu tyto moduly slouží, a jakým způsobem byly implementovány.

7.1 Vykreslovač

Pro tento simulátor byl prozatím vytvořen pouze jeden vykreslovač.

7.1.1 Simple2DRenderer

Tento vykreslovač slouží k vykreslování buněk ve dvoudimenzionálním prostoru. Buňky reprezentuje jednobarevnými čtverci. Je definován v souborech `Simple2DRenderer.py` a `Simple2DRendererMetadata.json`.

Obsahuje upravenou definici inicializační funkce. Tato funkce nejdříve volá originální inicializační funkci. Následně nastavuje interní proměnné pro měřítko zobrazované sekce prostoru, rychlost pohybu „kamery“ a pozici této kamery. Následně definuje exportní funkce a vkládá je do listu na tyto funkce. Finálně nastavuje barvu pozadí na černou.

Tento vykreslovač požaduje, aby buňky, které zpracovává, obsahovaly ve svých externích datech klíče `xPosition` a `yPosition`, pod kterými jsou číselná data udávající aktuální souřadnice, na kterých se buňka nachází. Dále je také potřeba, aby byl na klíči `color` tuple se třemi pozitivními celočíselnými hodnotami mezi 0 až 255 udávající barvu buňky. Tyto hodnoty reprezentují červenou, zelenou a modrou barevnou složku této barvy.

Hlavní sekci tohoto modulu je funkce `render`. Tato funkce nejdříve získá výšku a šířku výstupního obrazu. Pomocí těchto údajů, společně s pozicí kamery a měřítkem, zjišťuje levou, pravou, horní a spodní mezi, které spolu určují, která sekce plochy simulace bude vykreslena. Pomocí funkce `new` modulu `Image` knihovny `PIL` je vytvořena nová instance obrázku. Mezi vstupní proměnné patří výška a šířka tohoto obrázku společně s barevným prostorem, nastaveným na `RGB`, a barvou pozadí. Z této instance je pomocí funkce `Draw` modulu `ImageDraw` stejné knihovny vytvořena nová instance objektu pro práci s tímto obrázkem. V případě, že došlo k chybě, funkce vrátí hodnotu `None`. Jinak následuje `for` cyklus pro všechny buňky v listu, který byl této funkci předán. V této smyčce nejdříve probíhá kontrola toho, zda se aktuální buňka nachází v definovaných mezích. Pokud ano, je zjištěna

pozice levého horního a pravého dolního rohu této buňky, omezená na meze plochy, která je vykreslována. Podle těchto hodnot, společně s barevným údajem z dat buňky, je v obrázku funkcí `rectangle` instance třídy `ImageDraw` vykreslen obdélník. Finálně je pomocí funkce `BytesIO` zabudované třídy `io` vytvořen buffer, do kterého je obrázek uložen ve formě bajtů uložen funkcí `save` instance třídy `Image`. Obsah bufferu funkce vrací.

Nastavení barvy pozadí je prováděno funkcí `setBackgroundcolor`. Vstupní data jsou tři čísla ve formě tuple. První reprezentuje červenou barvu, druhé zelenou a třetí modrou. Jejich hodnoty se pohybují mezi 0 a 255. Tato funkce je využitelná prostředím.

Implementace funkce `convertFromImageCoordinates` konvertuje mezi souřadnicemi výstupního obrázku a souřadnicemi simulace za pomoci vstupních souřadnic, měřítka, pozice kamery a velikosti obrázku v pixelech.

Tato třída implementuje pouze jednu funkci na uživatelskou interakci za pomoci myši, a to je `_primaryDrag`. Tato funkce odečítá od originální pozice novou pozici, a výsledek následně dělí měřítkem. Získané hodnoty přičítá k aktuální pozici kamery.

Zbylé funkce v této třídě reprezentují exportní funkce.

Skupina funkcí, `_moveUp`, `_moveDown`, `_moveLeft` a `_moveRight` slouží k pohybu s kamerou bez využití interakcí s myší. Všechny tyto funkce dělí rychlost pohybu měřítkem, a tuto hodnotu následně přičítají k některé ze souřadnic nebo ji od těchto souřadnic odčítají. V UI jsou reprezentovány tlačítka, které se při držení spouští opakovaně s periodou deseti milisekund. Jejich zobrazovaná jména jsou „Move up“, „Move down“, „Move left“ a „Move right“.

Funkce `_setMoveSpeed` je reprezentována sliderem, který se pohybuje mezi hodnotami 1 až 25, a začíná na hodnotě 1. Hodnota získaná z tohoto slideru je uložena do proměnné pro rychlost pohybu kamery. U slideru se nachází text „Camera speed“.

Měřítka se mění exportními funkcemi `_zoomIn` a `_zoomOut`. Funkce pro přiblížení násobí měřítko dvěma. Při oddalování je měřítko děleno dvěma. V případě, že by mělo jít pod hodnotu 1, reprezentující jednu buňku na pixel, je nastaveno zpátky na jedna. Tyto funkce jsou reprezentovány klasickými tlačítky s textem „Zoom in“ a „Zoom out“.

7.2 Prostředí

7.2.1 Simple2DEnvironment

Prostředí Simple2DEnvironment je založeno na vykreslovači Simple2DRenderer. Toto prostředí obsahuje základní sadu nástrojů pro buňky, které jim umožňují vytvářet nové buňky, zjišťovat informace o polích v simulaci, zjišťovat, kolik kroků simulace od začátku proběhlo, a mazat sebe sama, i s případným nahrazením sebe jinou buňkou. Je definováno soubory Simple2DEnvironment.py a jeho metadaty.

V inicializační funkci je nejdříve volána originální inicializační funkce rodičovské třídy. Následně je také vytvořena proměnná pro počítání provedených kroků od začátku simulace. Finálně je vytvořen slovník, do kterého jsou za běhu simulace ukládány reference na aktivní buňky s jejich souřadnicemi jako klíči. Tento slovník zrychluje proces zjišťování informací o specifických polích v simulaci, například v případech, kdy je potřeba zjistit, zda je pole, kde chceme vytvořit novou buňku, prázdné.

Pro zjednodušení práce s tímto slovníkem byla vytvořena funkce `_updateCellMap`, která přijímá dvoudimenzionální souřadnice a případně referenci na buňku, která je v její absenci nahrazena hodnotou `None`. Pokud tato hodnota není `None`, je do slovníku na klíč určený souřadnicemi tato reference vložena. Jinak je z tohoto místa a z celého slovníku položka na této pozici odstraněna.

Pomocná funkce `_checkForCellAbsolute` slouží ke zjišťování toho, zda na specifických absolutních souřadnicích, získaných ze vstupních proměnných, existuje buňka. Používá ke své funkci zmíněný slovník buněk. Pokud se na místě buňka nachází, vrací `True`, jinak vrací `False`.

Další pomocná funkce je `_spawnCell`. Tato funkce vytvoří novou buňku na daných souřadnicích, pokud se zde již nějaká buňka nenachází. Její vstupní proměnné tedy jsou souřadnice a reference na instanci třídy odvozené od třídy `CellBrain`. Nejdříve pomocí předchozí pomocné funkce zjišťuje, zda je požadované místo volné. Pokud není, funkce končí. Jinak je vytvořena nová instance třídy `Cell` se vstupní instancí interní části jako vstupem do inicializační funkce. Ve slovníku s daty buňky jsou vytvořeny klíče pro pozici buňky, získané ze vstupních proměnných. Je také vytvořen klíč pro barvu buňky. Pokud třída která definuje vstupní instanci interní části buňky obsahuje proměnnou `COLOR`, je nastavena jako hodnota pod relevantním klíčem. Jinak je nastavena defaultní bílá barva. Na konci jsou volány funkce

`addCell` ze spouštěče buněk s novou buňkou jako vstupem a `_updateCellMap` se souřadnicemi a touto buňkou jako vstupem.

Vytváření buněk ze strany uživatele potřebuje extra kroky. Řeší to funkce `_addUserCell`. Tato funkce přijímá data o pozici pole na které uživatel kliknul vygenerovaná vykreslovačem. Nejdříve tato data rozbálí na jejich komponenty. Následně ze spouštěče získává referenci na uživatelem aktuálně vybraný typ buňky. Pokud tato buňka neexistuje, nic se nestane. Jinak je vytvořena nová instance této interní struktury buňky, která je spolu se souřadnicemi předána předchozí funkci.

Finální pomocná funkce, `_userRemoveCell`, funguje stejným způsobem jako předchozí funkce, s tím rozdílem, že volá funkci `removeCell` spouštěče buněk, a buňku odstraňuje ze slovníku buněk.

Následující funkce jsou určeny k tomu, aby byly volány z instancí buněk při spouštění jejich funkční části.

Funkce `checkForCell` slouží ke zjišťování toho, zda existuje na specifickém místě v prostoru simulace buňka. Vstupní souřadnice této funkce jsou relativní k pozici aktuálně spouštěné buňky. Funkce nejdříve kontroluje, zda aktuální buňka existuje. Pokud ano, bere její pozici, ke které přičítá vstupní souřadnice. Tyto hodnoty jsou následně zasílány do funkce `_checkForCellsAbsolute`, jejíž odpověď následně vrací jako svůj vlastní výstup.

Podobně také funguje funkce `checkIfCellTypeEqual`. Tato funkce má navíc vstupní proměnnou reprezentující referenci na třídu odvozenou od třídy `CellBrain`. Snaží se zjistit, zda na daných souřadnicích existuje buňka této třídy. Pozici buňky získá stejně jako předchozí funkce. Pokud se zde ale buňka nachází, je ještě pomocí zabudované funkce `isInstance` kontrolováno, zda je tato funkce hledanou instancí vstupní třídy. Funkce vrací `True`, pokud se zde buňka hledaného typu nachází. Pokud se zde buňka daného typu nenachází, nebo zde není buňka žádná, vrací `False`.

Pro vytváření nových buněk ze strany buňky slouží funkce `spawnCell`. Tato funkce volá funkci `_spawnCell` s tím, že vstupní souřadnice relativní k pozici aktuálně aktivní funkce převádí na souřadnice absolutní. Také na začátku probíhá kontrola toho, zda současná buňka existuje.

Mazání aktuální buňky řeší funkce `deleteCurrentCell`. Zjistí, zda současná buňka existuje, a jaké má souřadnice. Následně je volána funkce `_updateCellMap` bez interní části

buňky, což smaže aktuální buňku z mapy buněk. Finálně je volána funkce `removeCell` ze spouštěče buněk s aktuální buňkou jako vstupní proměnnou. Tím se buňka odstraní ze simulace.

Nahrazení buňky jinou buňkou je řešeno funkcí `deleteCurrentSpawnNewCell`. Zde jsou získány souřadnice aktivní buňky, a jsou volány funkce `deleteCurrentCell` a `_spawnCell`.

Poslední funkcí pro buňky je `getCurrentStepNumber`, která pouze vrací číslo aktuálního kroku simulace.

Finální skupina funkcí je určena pro volání z jiných funkcí.

Reakci na dokončení cyklu buněk zpracovává funkce `_cellsCycled`. Při zavolání navýší počítadlo cyklů simulace o jeden.

Pokud spouštěč smaže všechny buňky, je volána funkce `_executorClearedCells`. Tato funkce vytvoří novou instanci mapy buněk, která nahradí původní instanci.

Funkce pro zpracování kliknutí a kliknutí + tažení levým tlačítkem myši, `_primaryClick` a `_primaryDrag`, pouze volají funkci `_addUserCell`. Jako vstupní proměnnou používají nově získaná data o pozici.

Stejně funkce s pravým tlačítkem, `_secondaryClick` a `_secondaryDrag`, volají funkci `_userRemoveCell`.

7.2.2 Energy2DEnvironment

Prostředí `Energy2DEnvironment` funguje velmi podobně jako `Simple2DEnvironment` s tím rozdílem, že do simulace přidává systém energie. Různé akce stojí různé množství energie, kterou buňky postupem času dostávají a ztrácejí. Buňky se také nově mohou pohybovat, a posílat mezi sebou zprávy. Toto prostředí se nachází v souboru `Energy2DEnvironment.py` a v přidruženém souboru s metadaty.

V této sekci budou popsány pouze ty části implementace prostředí, které jsou odlišné od `Simple2DEnvironment`. Velké množství kódu je z tohoto prostředí zkopírováno.

Hned první rozdíl se nachází v inicializační funkci. Jsou zde vytvářeny nové proměnné. Dvě z nich určuje maximální úroveň pozitivní a negativní úrovně, které mohou buňkám za běhu simulace být předávány. Kvůli vylepšení výkonu je plocha, ve které se buňky mohou vyskytovat, omezena. Velikost čtvercové plochy, kde buňky přežijí, je definována proměnnou. Tato verze prostředí také definuje tři exportní funkce pro ovládání těchto proměnných.

Exportní funkce pro nastavení maximálních hodnot jsou reprezentovány slidery v rozsahu od 0 do 50 s textem „Peak positive energy“ a „Peak negative energy“. Funkce napojené na tyto slidery jsou `_setTopEnergy` a `_setBottomEnergy`. Obě tyto funkce vezmou vstupní hodnotu, vydělí ji stovkou, a novou hodnotu zapíšou do relevantní proměnné.

Funkce pro změnu velikosti plochy simulace je také reprezentována sliderem, tentokrát v rozsahu od 10 do 100. Napojená funkce je `_setArenaSize`. Získanou hodnotu pouze ukládá do proměnné.

Energie, která je buňkám každý cyklus předávána, se postupem času mění. Tuto hodnotu počítá funkce `_getEnvironmentEnergyLevel`. Hodnota osciluje podle periodické funkce sinus, se vstupní hodnotou vypočítanou z počtu kroků vynásobeného konstantou π a vydělenou třiceti. Výsledek této funkce je následně vynásoben maximální vrchní nebo spodní energií dle toho, jestli je pozitivní nebo negativní. Tuto hodnotu funkce vrací.

Pomocná funkce pro přidávání nových buněk, `_addCell`, přidává do buněk ke standardním proměnným také hodnotu energie těchto buněk, defaultně 0.5, a list určený na přijímání zpráv.

Rozsah působení některých funkcí byl omezen na von Neumannovo sousedství. Funkce `_determineDirection` konvertuje souřadnice na jednu ze čtyřech možných pozic v tomto sousedství. Vytvoří nové relativní souřadnice s nulovými hodnotami. K jedné z těchto souřadnic je přičtena jednička dle toho, která z absolutních originálních vstupních souřadnic je větší. K jedničce je přiřazeno znaménko této souřadnice pomocí funkce `copysign` zabudované třídy `math`.

Z funkcí z originální implementace toto zasahuje funkci pro přidávání nových buněk ze strany buňky, `spawnCell`.

K některým existujícím funkcím byla přidána „cena“, tj. kolik energie buňku stojí použití této funkce. Zjišťování informací o buňce na dané pozici funkcemi `checkForCell` a `checkIfCellTypeEqual` stojí 0.025 energie za každé políčko mezi aktuálně aktivní buňkou a cílovou pozicí. Počet políček je získán součtem relativních vstupních hodnot. Vytváření nových buněk stojí 0.5 energie, což je stejné množství energie se kterou nově vytvořené buňky začínají.

Některé funkce se nyní řadí do speciální kategorie. V každém cyklu může každá buňka použít pouze jednu funkci z této kategorie. Z originálních funkcí toto platí pouze pro funkci

spawnCell.

Funkce kombinující smazání aktuální buňky a její nahrazení novou buňkou v tomto prostředí neexistuje.

Reakce na dokončení cyklu, funkce `_cellsCycled` nyní navíc mění barvu pozadí vykreslovače jeho funkcí `setBackgroundColor`. Získaná barevná hodnota je založena na aktuální úrovni energie v prostředí z funkce `_getEnvironmentEnergyLevel`. Mění se ze žluté, reprezentující pozitivní energii, na modrou, reprezentující negativní energii.

Prostředí nově také reaguje na změnu aktivní buňky. Reakce je definována ve funkci `_cellSwitched`. Při této změně k buňce přičítá aktuální úroveň energie a kontroluje, zda tato energie není menší nebo rovná nule, nebo větší než jedna. Také kontroluje, zda se buňka nachází v mezích simulace. Pokud jednu z těchto podmínek nesplňuje, je ze simulace odstraněna.

Mezi originální funkce pro buňky tohoto prostředí se řadí funkce `getEnergyLevel`, která vrací úroveň energie aktuálně aktivní buňky.

Další z těchto funkcí je `move`, která přijímá relativní souřadnice, a pohybuje aktuálně aktivní buňkou. Tyto souřadnice jsou omezeny na von Neumannovo sousedství, jsou tedy zpracovány funkcí `_determineDirection`. Tato funkce stojí 0.1 energie a patří do skupiny funkcí, ze kterých může buňka za cyklus zavolat pouze jednu. Překonvertované relativní souřadnice jsou přičteny k aktuální pozici buňky, pokud se na nové pozici již nějaká buňka nenachází, což je zjištěno funkcí `_checkForCellAbsolute`. Pokud se posunout může, je pozice v datech buňky upravena. V listu buněk je tato buňka z originální pozice funkcí `_updateCellMap` smazána. Následně je stejnou funkcí přidána na nové souřadnice.

Energii je možné mezi sousedními buňkami předávat. Předávání energie zprostředkovává funkce `giveEnergy`. Tato akce stojí pouze to, co buňka chce předat svému sousedovi. Vstupní proměnné jsou relativní souřadnice a množství energie, kterou chce buňka předat. Funkce relativní souřadnice překonvertuje na von Neumannovo sousedství funkcí `_determineDirection`. Tuto hodnotu sčítá s absolutní pozicí aktivní buňky. Následně zjišťuje, zda na cílové pozici existuje buňka. Pokud ano, je vstupní energie, omezená na rozsah energie, který zdrojová buňka má, předána nalezenému sousedovi. Tato funkce patří do dříve zmíněné omezené skupiny funkcí.

Buňky mohou taky využít svůj „tah“ na generování energie. Funkce `rest` aktuální buňce

přidá 0.1 energie. Patří do skupiny omezených funkcí.

Jak již bylo zmíněno, buňky obsahují „schránku“ na zprávy. Pomocí funkce `sendMessage` může buňka svému sousedovi poslat zprávu s libovolnými daty. Vstupními proměnnými jsou relativní souřadnice a zpráva, která bude sousedovi předána. Zaslání zprávy stojí 0.05 energie. Stejně jako u ostatních funkcí probíhá konverze souřadnic na von Neumannovo sousedství, spolu s kontrolou, zda soused existuje. Pokud jsou tyto podmínky splněny, do listu na zprávy cílové buňky je tato zpráva vložena. Zasílání zpráv je omezeno jako některé další funkce.

Buňka může kontrolovat počet zpráv ve své „schránce“ funkcí `getMessageCount`. Tato funkce není nijak omezená.

Čtení přijatých zpráv je prováděno funkcí `getTopMessage`. Tato funkce vrátí první, tedy nejstarší, zprávu ze „schránky“. Tuto zprávu pak ze schránky odstraňuje. Tato funkce také není nijak omezená.

Poslední novou funkcí je `changeColor`. Tato funkce mění, jakým způsobem je buňka ve náhledu na simulaci vykreslována. Vstupní proměnná je tuple o třech hodnotách pro červený, zelený a modrý barevný kanál. Tyto hodnoty jsou mezi čísly 0 až 255.

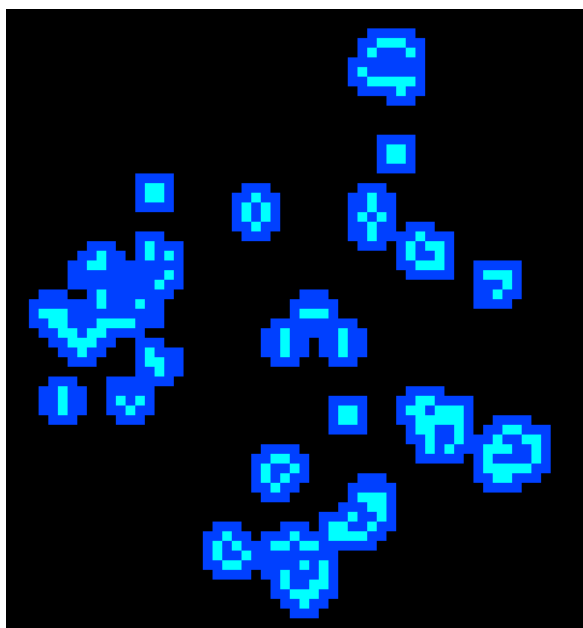
7.3 Balíčky buněk

Většina balíčků buněk, které byly pro tento simulátor vyvinuty, je postavena v závislosti na prostředí `Simple2DEnvironment`.

7.3.1 GoL

Balíček GoL simuluje hru života dle definice v kapitole 1.5.1 upravenou pro práci v prostředí s asynchronním spouštěním buněk a s prázdnými místy v mřížce. Obsahuje definice dvou typů buněk, „živá“ buňka a „mrtvá“ buňka.

Živá buňka je reprezentována třídou `AliveCell`. Její barva je (0, 255, 255), což je odstín azurové. Má upravenou inicializační funkci tak, že spolu s voláním rodičovského inicializátoru vytváří proměnnou pro ukládání počtu živých sousedů, a tuto proměnnou nastavuje na 2. Ve funkci `run` buňka získává funkcí `getCurrentStepNumber` prostředí aktuální krok, který modulem omezuje na rozsah 0-2. Dle aktuálního kroku se provede jedna ze třech možných akcí. V prvním kroku prochází všechny svoje sousedy v Moorově sousedství. Na každém



Obrázek 7.1 Modul GoL v simulaci (vlastní tvorba)

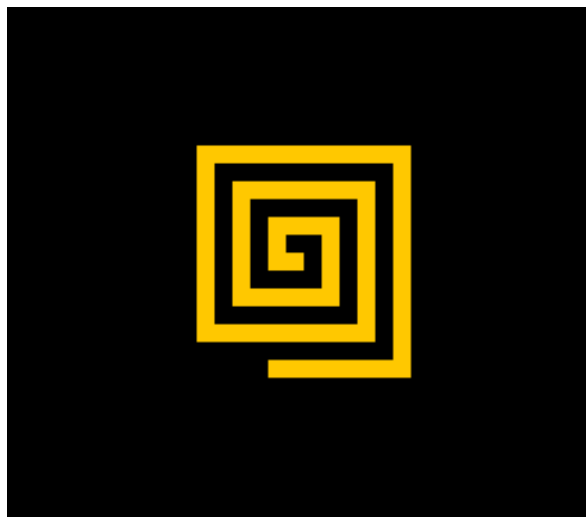
z těchto míst se snaží funkcí `spawnCell` vytvořit novou instanci buňky třídy `DeadCell`. V druhém kroku také prochází všechny své sousedy, v tomto případě ale pomocí funkce `checkIfCellTypeEqual` počítá, kolik sousedů typu `AliveCell` má. Ve třetím kroku tuto hodnotu kontroluje. Pokud je počet sousedů nižší než dva a vyšší než tři, buňka nahrazuje sebe instancí třídy `DeadCell` funkcí `deleteCurrentSpawnNewCell`.

Mrtvá buňka, `DeadCell`, funguje velmi podobně. Její barva je (0, 64, 255), což je odstín modré. V prvním kroku nedělá nic, existuje pouze pro synchronizaci se živými buňkami. V druhém kroku také počítá, kolik má žijících sousedů. Ve třetím kroku je buňka nahrazena instancí `DeadCell`, pokud byl počet sousedů roven třem. Pokud buňka neměla žádné žijící sousedy, je funkcí `deleteCurrentCell` smazána. Tím je zajištěno, že se plocha simulace konstantně nezaplňuje mrtvými buňkami.

7.3.2 Pattern

`Pattern` obsahuje pouze jeden typ buňky, `Pattern`. Pokud může, tato buňka vytvoří jednoho souseda, kterému předá data o pozici v řetězu buněk. Postupným vytvářením nových buněk se tvoří vzor rostoucí spirály.

Tato buňka definuje svoji barvu jako (255, 200, 0), tedy odstín oranžové. Upravený inicializátor této buňky přijímá tři dobrovolné proměnné. První z nich je maximální délka strany spirály, ve které nová buňka je. Druhá reprezentuje aktuální pozici této buňky v této straně



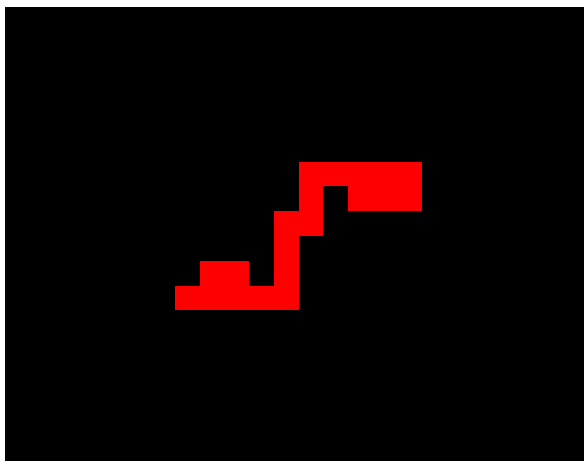
Obrázek 7.2 Modul Pattern v simulaci (vlastní tvorba)

spirály. Poslední reprezentuje směr, ve kterém má buňka vytvořit nového potomka. Tyto hodnoty jsou defaultně nastaveny na 1, 1 a 0. Inicializátor také definuje proměnnou `moved`, která určuje, zda již byl vytvořen potomek. Ve funkci `run` nejdříve probíhá kontrola, zda již byl potomek vytvořený, pokud ano, funkce končí. Následně je vytvořen list s relativními souřadnicemi čtyř stran Moorova sousedství v pořadí doleva, dolů, doprava, nahoru. Pokud je hodnota maximální délky strany stejná jako aktuální pozice buňky v této straně, je pozice vynulována a maximální délce přičtena jednička. K nové straně pohybu je také přičtena jednička, výsledek je ale před uložením do proměnné modulován čtyřmi, aby zůstal v požadovaném rozsahu. V každém případě je následně k pozici přičtena jednička, a je vytvořena nová instance buňky `Pattern` s novými hodnotami. Finálně je v originálním směru pohybu pomocí funkce `spawnCell` vytvořena nová buňka s touto instancí.

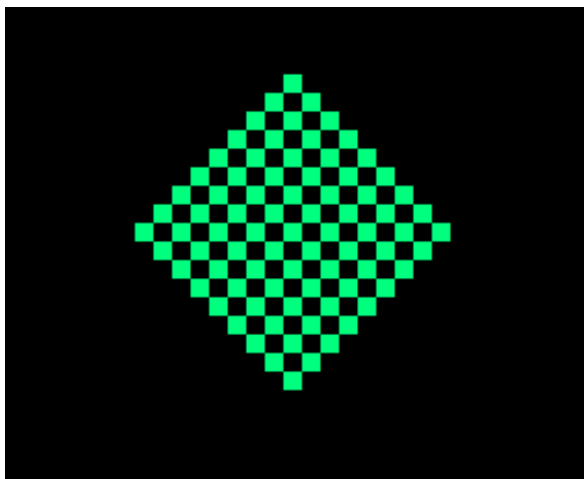
7.3.3 RandomWalk

V balíčku `RandomWalk` se také nachází pouze jedna buňka, `RandomWalk`. Tato buňka vytváří novou buňku stejného typu v náhodném směru, mimo ten směr, ze kterého pochází. Tento proces pokračuje, dokud řetěz buněk nenarazí do jiné buňky nebo sám do sebe.

Tato buňka je červená, s barevným odstínem (255, 0, 0). Její upravený inicializátor přidává proměnnou která udává, zda již bylo provedeno rozšíření buňky. Přidává taky proměnnou s předchozím směrem rozšíření, získanou jako dobrovolná proměnná, která je defaultně nastavena na 0. Funkce `run` nejdříve testuje, zda již buňka vytvořila potomka. Pokud ano, funkce končí. Následně si náhodně vybere nový směr, který je jiný než předchozí obdržený



Obrázek 7.3 Cesta nalezena modulem RandomWalk (vlastní tvorba)



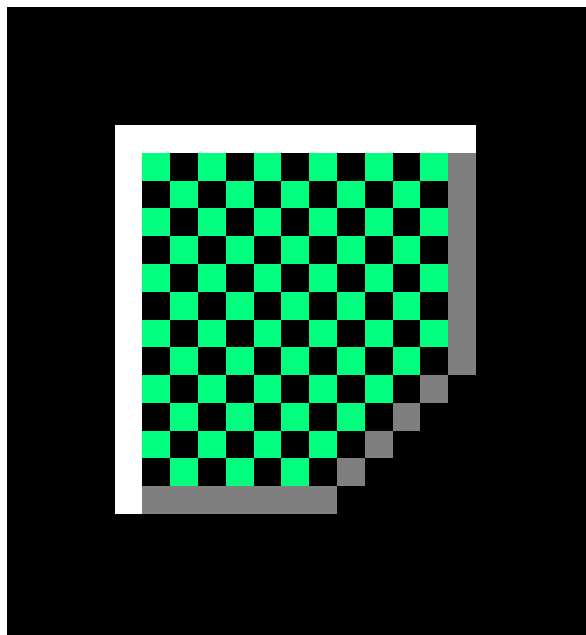
Obrázek 7.4 Modul Virus po osmi krocích (vlastní tvorba)

směr. Směry jsou stejně jako u buňky Pattern reprezentovány listem pozic. Je vytvořena nová instance buňky RandomWalk s hodnotou reprezentující stranu ke které byla přičtena dvojka a která byla modulována čtyřmi jako vstupní proměnnou. Buňka je vytvořena funkcí `spawnCell`.

7.3.4 Virus

Stejně jako předchozí dva příklady obsahuje balíček Virus pouze jednu buňku, Virus. Tato buňka se v absenci překážek nekontrolovatelně šíří prostorem simulace.

Její barva je (0, 255, 127), odstín zelené s nádechem modré. Inicializátor nemění. Ve funkci `run` vytváří čtyři nové potomky třídy Virus, které se snaží vložit do svého, Moorova, sousedství funkcí `spawnCell`. Následně je tato buňka funkcí `deleteCurrentCell` smazána.



Obrázek 7.5 Stěny z modulu Wall blokující buňky z modulu Virus (vlastní tvorba)

7.4 Walls

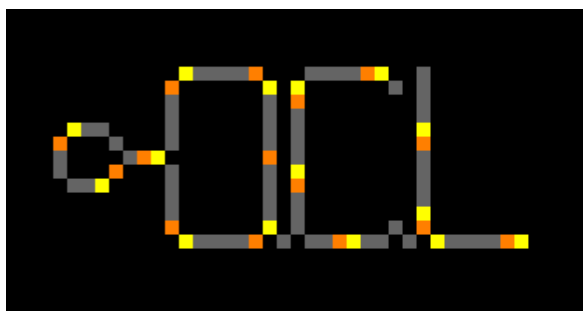
Walls přidává dva typy buněk, které slouží k blokování ostatních buněk. Oba typy buněk fungují stejným způsobem, liší se pouze v barvě. Prvním typem je buňka `WhiteWall`, která má barvu (255, 255, 255), tedy bílou. Druhým typem je `GrayWall`, s barvou (127, 127, 127), tedy šedá barva přesně mezi černou a bílou. Funkce `run` pouze obsahuje klíčové slovo `pass`, funkce tedy nic nedělá a buňky se svým prostředím neinteragují.

7.4.1 WireWorld

Buňky v balíčku `WireWorld` implementují buněčný automat `WireWorld` dle definice v 1.5.3 upravený pro práci v prostředí s asynchronním spouštěním buněk. Obsahuje tři typy buněk, reprezentující přední část elektronu, zadní část elektronu a vodič, kterým se elektron pohybuje. Stejně jako u implementace hry života se chování těchto buněk mění dle toho, kolikrát cyklus simulace aktuálně probíhá. Tentokrát se ale dělí pouze na dva kroky.

Přední část elektronu definuje třída `Head`. Její barva je (255, 255, 0), tedy žlutá. Funkce `run` získává aktuální počet uběhlých kroků simulace funkcí `getCurrentStepNumber`, který následně moduluje číslem dva. V případě, že se toto číslo rovná jedné, je buňka funkcí `deleteCurrentSpawnNewCell` nahrazena nově vytvořenou instancí buňky `Tail`.

Zadní část elektronu, třída `Tail`, má barvu (255, 127, 0), tedy odstín oranžové. Funguje



Obrázek 7.6 Obvod ve tvaru zkratky jména simulátoru (OCL) tvořený buňkami z modulu WireWorld (vlastní tvorba)

stejně jako třída Head s tím rozdílem, že se mění na instanci buňky Wire.

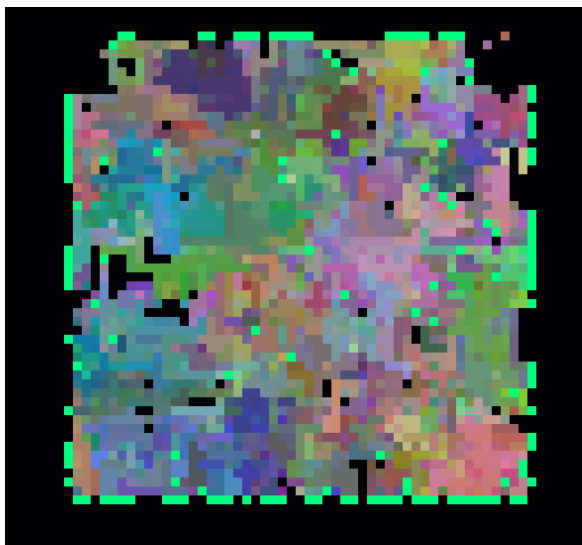
Vodič, reprezentován třídou Wire, s barvou (100, 100, 100), což je šedá barva tmavšího odstínu, je komplikovanější než předchozí třídy tohoto balíčku. Obsahuje upravenou inicializační funkci, která vytváří proměnnou pro ukládání počtu sousedů typu Head. Ve funkci run získává a upravuje aktuální počet kroků stejně jako předchozí typy buněk. Pokud se upravená hodnota rovna nule, je počet sousedů nastavený na nulu, a následně se postupně kontrolují všechny sousední buňky ve von Neumannově sousedství. Pro zjištění, zda je sousední buňka typu Head se využívá funkce `checkIfCellTypeEqual`. Pokud je buňka typu Head, hodnota proměnné sledující počet sousedů je navýšena. Pokud se upravená hodnota kroku simulace rovná jedné, probíhá kontrola, zda je počet sousedů menší než jeden nebo vyšší než dva. Pokud tomu tak je, buňka nic nedělá. Jinak se funkcí `deleteCurrentSpawnNewCell` mění na buňku typu Head.

7.4.2 GeneticCell

Aktuálně jediným balíčkem pro prostředí `Energy2DEnvironment` je balíček `GeneticCell`, který obsahuje pouze jeden stejnojmenný typ buňky. Tato buňka ve své interní části ukrývá genom, tedy list čísel, dle kterého vybírá, které akce bude provádět. Při vytváření nových buněk může dojít k mutacím. Buňky také mají možnost předávat svůj genom ostatním buňkám, čímž se při vytváření potomků jejich genomy kombinují. Díky této funkcionalitě se buňky postupně „vyvíjejí“, a dokážou lépe ustát podmínky nastolené tímto prostředím.

Základní barva buňky `GeneticCell` je (0, 255, 127), což je odstín zelené s modrým komponentem. Barva se ale po vytvoření buňky mění. Inicializační funkce této buňky je výrazně rozšířená. Má dvě nepovinné vstupní proměnné, reprezentující dva možné vstupní genomy pro vytvoření buňky. Nejdříve je nastavena délka genomu na 15 prvků, a šance na mutaci na

5%. Následně je vytvořen prázdný list reprezentující nový genom. Pokud obě vstupní hodnoty obsahují genomy, probíhá proces rekombinace a mutace. Pro každý prvek z patnácti se nejdříve kontroluje, zda jsou oba genomy dostatečně dlouhé na to, aby z nich mohla být vybrána hodnota. Pokud ne, nebo pokud je náhodně vybrána hodnota menší než šance na mutaci, je vygenerována náhodná hodnota od jedné do tisíce, která je využita jako hodnota genu. Jinak je náhodně vybrán jeden gen na aktuální pozici ze vstupních genomů, omezeny na rozsah od jedné do tisíce. Pokud alespoň jedna ze vstupních hodnot nebyla dodána, je celý genom vytvořen z náhodných hodnot od jedné do tisíce. Inicializátor také vytváří proměnnou s defaultní hodnotou 200, která slouží pro ukládání čísel získaných z některých možných operací, a také logickou proměnnou která sleduje, zda již byla změněna barva buňky. Na začátku funkce `run` nejdříve probíhá kontrola, zda již byla nastavena barva buňky. Pokud nebyla, je vygenerována z genomu. Genom je rozdělen na tři části. Hodnoty těchto částí jsou sečteny, a následně jsou normalizovány tak, aby se nacházely mezi hodnotami 0 až 255. První třetina genomu reprezentuje červený kanál, druhá zelený a třetí modrý. Získaná barva je nastavena pomocí funkce `changeColor`. Následně je logická proměnná reprezentující změnu barvy nastavena na `True`. V hlavní části této funkce jsou pomocí smyčky `while` procházeny všechny geny v genomu. Pomocí funkce `modulo` je každý gen omezen na hodnoty 0 až 13. Každá z těchto hodnot reprezentuje jednu akci, kterou buňka může provést. Následující gen, nebo v případě posledního genu první gen, je uložen, a také pomocí modula převeden na jednu pozici v Moorově sousedství a je zvlášť uložen. Tato hodnota se využívá pouze u některých akcí. Do proměnné je také pomocí funkce `getEnergyLevel` uložen aktuální stav energie buňky. Následující krok je rozhodnutý hodnotou získanou z genomu. Při hodnotě 0 se nestane nic. Při hodnotě 1 se volá funkce `rest`. Při hodnotě 2 je vytvořena nová buňka, pokud je interní proměnná větší než 100. Pokud má buňka zprávy, tak vrchní sebere, a použije ji jako jeden ze vstupních genomů, jinak používá dvě kopie svého genomu. Strana, na které je buňka vytvořena, je vybrána dle následujícího genu. Pokud je pokus o vytvoření buňky proveden, je následující gen přeskočen. Při hodnotě 3 se následující hodnota přičte k interní proměnné, a následný gen se přeskočí. Při hodnotě 4 se hodnota odečte, a gen je také přeskočen. U hodnoty 5 se k interní hodnotě přičte současná energie vynásobená deseti. U hodnoty 6 se odečte. 7 ji vynuluje. U 8 se buňka pokusí posunout směrem vybraným následujícím genem, který je poté přeskočen. U 9 se buňka zničí. Hodnota 10 volá funkci `checkForCell` na pozici získanou z následujícího genu. Pokud se zde nachází buňka, je interní proměnná



Obrázek 7.7 Buňky z modulu GeneticCell s různými genomy (vlastní tvorba)

zvýšena o 100. Následující gen je přeskočen. U hodnoty 11 buňka na pozici danou dalším genem pošle svůj genom, pokud má interní hodnota hodnotu více než 100. V tomto případě je přeskočen následující gen. U poslední hodnoty, 12, dává buňka svému sousedovi energii, pokud je interní proměnná vyšší než 0. V tomto případě na pozici vybrané z následujícího genu posílá hodnotu jedna vydělenou interní proměnnou. V případě, že je tato hodnota menší než jedna, je nahrazena jedničkou. I zde je následující gen přeskočen.

ZÁVĚR

Cílem práce bylo vytvoření unikátního simulátoru buněčných automatů, který uživatelům nabízí rozšiřitelnost, otevřenost a obecnost. Dalším cílem bylo také vytvoření ukázkových modulů pro tento simulátor.

V teoretické části práce poskytuje potřebný kontext ohledně buněčných automatů, jejich historie, některých existujících sad pravidel a existujícího softwaru určeného pro simulaci buněčných automatů. Součástí popisu buněčných automatů je také uvedení různých generalizací tohoto konceptu.

Praktická část následně popisuje proces návrhu architektury vlastního simulátoru, výběr potřebných nástrojů a také samotnou implementaci tohoto simulátoru.

Simulátor byl rozdělen do řady interních a externích modulů, které spolupracují pro vytvoření funkční simulace. Interní moduly poskytují obecnou funkcionalitu potřebnou pro práci všech externích modulů. Jsou v tedy v simulátoru přímo zabudovány. Patří mezi ně například modul definující uživatelské rozhraní, či modul pro správu a spouštění buněk. Externí moduly dodávají funkcionalitu potřebnou pouze pro specifické konfigurace buněčných automatů. V simulátoru tedy nejsou implementované, ale musí být dodány. Dělí se na tři hlavní typy. Vykreslovače definují souřadnicový systém simulace, ve kterém následně vykreslují aktivní buňky. Prostředí definují pravidla simulace a interakcí mezi buňkami a jejich okolím. Buňky využívají prostředí pro zjišťování informací o svém okolí a provádění akcí. Tímto způsobem je zajištěno, že se buňky musí chovat podle pravidel stanovených prostředím.

Projekt je psán v jazyce Python, spolu s knihovnou PyQt6 a dalšími vybranými knihovnami zabudovanými v Pythonu. Při vývoji ukázkových modulů byla také využita knihovna Pillow. Zdrojový kód projektu byl publikován pod MIT licencí na platformě GitHub, kde je dostupný pod url: <https://github.com/m-machala/OpenCellLab>. Zároveň byla jeho kopie také odevzdána jako elektronická příloha společně s touto prací.

Finální verze projektu obsahuje deset příkladů externích modulů. Dělí se na jeden vykreslovač, dvě prostředí a sedm balíčků buněk.

Výstupem této práce je funkční simulátor, který je otevřený, rozšiřitelný a dostatečně obecný na to, aby mohl podporovat velké množství typů buněčných automatů. Díky tomu může být využitý jako nástroj pro simulaci mnoha různých druhů komplexních systémů,

jako je například chování jedinců ve velkých skupinách, dynamika tekutin nebo třeba šíření lesních požárů.

Výstupem jsou také ukázky modulů pro tento simulátor, které jsou distribuovány spolu s ním. Ty uživatelům poskytují různé buněčné automaty, se kterými mohou experimentovat, a zároveň uživatelům dávají možnost pročíst a upravit jejich zdrojový kód, díky čemuž slouží také jako odrazový můstek pro tvoření vlastních modulů.

Požadavky stanovené na tuto práci byly úspěšně naplněny, společně se všemi body zadání.

SEZNAM POUŽITÉ LITERATURY

- [1] WEISSTEIN, Eric W., c1999-2025. *Cellular Automaton*. Online. Dostupné z: <https://mathworld.wolfram.com/CellularAutomaton.html>. [cit. 2025-03-17].
- [2] ILACHINSKI, Andrew, 2001. *Cellular automata: a discrete universe*. Singapore: World Scientific. 1. vyd. ISBN 981-02-4623-4.
- [3] AMOS, Martyn, 2004. *Fundamentals of Physics*. Oxford: Oxford University Press. 1. vyd. ISBN 0195155408.
- [4] NORTHROP, Robert B., 2011. *Introduction to complexity and complex systems*. Boca Raton: CRC Press. ISBN 9781439839010.
- [5] HADELER, Karl-Peter a MÜLLER, Johannes, 2017. *Cellular Automata: Analysis and Applications*. Springer. 1. vyd. ISBN 978-3-319-53042-0.
- [6] WEISSTEIN, Eric W., c1999-2025. *Elementary Cellular Automaton*. Online. Dostupné z: <https://mathworld.wolfram.com/ElementaryCellularAutomaton.html>. [cit. 2025-03-24].
- [7] ZAITSEV, Dmitry A., 2017. A generalized neighborhood for cellular automata. Online. *Theoretical Computer Science*. 2017. 666, s. 21–35. ISSN 0304-3975. Dostupné z: <https://doi.org/10.1016/j.tcs.2016.11.002>. Dostupné z: <https://www.sciencedirect.com/science/article/pii/S0304397516305916>. [cit. 2025-03-24].
- [8] GAJARDO, Anahi; MOREIRA, Andres a GOLES, Eric, 2003. Complexity of Langton's Ant. Dostupné z: <https://arxiv.org/abs/nlin/0306022>.
- [9] PEGG JR., Edd, c1999-2025. *WireWorld*. Online. Dostupné z: <https://mathworld.wolfram.com/WireWorld.html>. [cit. 2025-04-01].
- [10] FATÈS, Nazim A., 2018. *Statistical Methods*. In: *Encyclopedia of Complexity and Systems Science*. Springer, s. 21. ISBN 978-3-642-27737-5. Dostupné z: <https://inria.hal.science/hal-01653675v1>.
- [11] BABURIN, Ivan; COOK, Matthew; GRÖTSCHLA, Florian; PLESNER, Andreas a WATTENHOFER, Roger, 2025. Universality Frontier for Asynchronous Cellular Automata. Dostupné z: <https://arxiv.org/abs/2502.05989>.
- [12] BACKER, Lena, 2023. *Continuous Cellular Automata*. Online. Dostupné z: <https://hegl.mathi.uni-heidelberg.de/continuous-cellular-automata/>. [cit. 2025-04-01].
- [13] RODRÍGUEZ PUENTES, Rafael; PÉREZ BETANCOURT, Yadian a MUFETI, Kauna, 09 2015. Cellular Automata And Its Applications In Modeling And Simulating The Evolution Of Diseases.
- [14] CHECHINA, Antonina; CHURBANOVA, Natalia; TRAPEZNIKOVA, Marina; ERMAKOV, Alexey; GERMAN, Mihail et al., 01 2022. Cellular automata in the application to the modeling of traffic flows. *Keldysh Institute Preprints*. 01 2022. s. 1–12. Dostupné z: 10.20948/prepr-2022-98.

- [15] KARAFYLLIDIS, Ioannis a THANAILAKIS, Adonios, 1997. A model for predicting forest fire spreading using cellular automata. *Ecological Modelling*. 1997. 99, č. 1, s. 87–97. Dostupné z: 10.1016/S0304-3800(96)01942-4. Dostupné z: <https://www.dpi.inpe.br/gilberto/cursos/st-society-2013/Kara1997.pdf>.
- [16] WOJOWICZ, Mirek, 2024. *Continuous Cellular Automata*. Online. Dostupné z: <https://mcell.ca/pages/mcell.html>. [cit. 2025-04-21].
- [17] WOJOWICZ, Mirek, 2024. *Supported CA Families*. Online. Dostupné z: <https://mcell.ca/pages/families.html>. [cit. 2025-04-28].
- [18] TREVORROW, Andrew a ROKICKI, Tom, 2025. *Golly*. Online. Dostupné z: <https://golly.sourceforge.io/>. [cit. 2025-04-21].
- [19] TREVORROW, Andrew a ROKICKI, Tom, c2025. *Credits*. Online. Dostupné z: <https://golly.sourceforge.io/Help/credits.html>. [cit. 2025-04-28].
- [20] TREVORROW, Andrew a ROKICKI, Tom, c2025. *Bounded*. Online. Dostupné z: <https://golly.sourceforge.io/Help/bounded.html>. [cit. 2025-04-28].
- [21] HUTTON, Tim, 2024. *Credits*. Online. Dostupné z: <https://gollygang.github.io/ready/Help/credits.html>. [cit. 2025-05-04].
- [22] HUTTON, Tim, 2024. *ready*. Online. Dostupné z: <https://github.com/GollyGang/ready>. [cit. 2025-04-21].
- [23] HUTTON, Tim, 2024. *What is reaction-diffusion?* Online. Dostupné z: <https://gollygang.github.io/ready/Help/introduction.html>. [cit. 2025-05-04].
- [24] RUCKER, Rudy a WALKER, John, 2024. *CellLab*. Online. Dostupné z: <https://www.fourmilab.ch/cellab/>. [cit. 2025-05-04].
- [25] RUCKER, Rudy a WALKER, John, 2024. *CellLab*. Online. Dostupné z: <https://www.fourmilab.ch/cellab/manual/webca.html>. [cit. 2025-05-04].

SEZNAM OBRÁZKŮ

Obrázek 6.1	Uvítací obrazovka simulátoru (vlastní tvorba)	58
Obrázek 6.2	Rozložení uvítací obrazovky simulátoru (vlastní tvorba)	58
Obrázek 6.3	Hlavní obrazovka simulátoru (vlastní tvorba)	62
Obrázek 6.4	Rozložení hlavní obrazovky (vlastní tvorba)	63
Obrázek 7.1	Modul GoL v simulaci (vlastní tvorba)	79
Obrázek 7.2	Modul Pattern v simulaci (vlastní tvorba)	80
Obrázek 7.3	Cesta nalezena modulem RandomWalk (vlastní tvorba)	81
Obrázek 7.4	Modul Virus po osmi krocích (vlastní tvorba)	81
Obrázek 7.5	Stěny z modulu Wall blokující buňky z modulu Virus (vlastní tvorba) ...	82
Obrázek 7.6	Obvod ve tvaru zkratky jména simulátoru (OCL) tvořený buňkami z modulu WireWorld (vlastní tvorba)	83
Obrázek 7.7	Buňky z modulu GeneticCell s různými genomy (vlastní tvorba)	85

SEZNAM VÝPISŮ

Výpis 6.1	Ukázková struktura JSON souboru pro balíček buněk	44
Výpis 6.2	Ukázková struktura JSON souboru pro prostředí	44
Výpis 6.3	Ukázková struktura JSON souboru pro vykreslovač	45

SEZNAM POUŽITÝCH SYMBOLŮ A ZKRATEK

AI	Artificial Intelligence
CA	Cellular Automata
GoL	Game of Life
JSON	JavaScript Object Notation
OCL	Open Cell Lab
RGB	Red Green Blue
UI	User Interface