Warianty gry w życie Conwaya

(Conway's Game of Life variants)

Marcin Rogala

Praca inżynierska

Promotor: dr hab. Jan Otop

Uniwersytet Wrocławski
Wydział Matematyki i Informatyki
Instytut Informatyki

25 grudnia 2020

Streszczenie

Przedstawiona praca opisuje implementację projektu, umożliwiającego tworzenie i wizualizację wariantów gry w życie Conwaya opartych na planszy zbudowanej na podstawie diagramów Woronoja. Części składowe projektu to generator diagramów Woronoja oparty na algorytmie S. Fortunea, wizualizacja gry przy zastosowaniu API OpenGl oraz ewaluator języka opisu gry.

This paper describes the project, which alowes, creation and visualization of Conway's Game of Life variants, on a a board made of Voronoi diagrams. Project consist of Voronoi diagrams generator using S. Fortune's algoritm, game visualisation using OpenGl API and game description language evaluator.

Spis treści

Rozdział 1.

Wprowadzenie

1.1. Gra w życie

Gra w życie, została zaprezentowana w 1970 roku przez Johna Conwaya na łamach miesięcznika *Scientific American* [1]. Jest to gra bez graczy, co oznacza, że jej ewolucja zależy tylko od stanu początkowego bez późniejszych ingerencji człowieka. Planszą jest dwu wymiarowa siatka kwadratowych komórek, z których każda ma 8 sąsiadów. Każda komórka może być żywa lub martwa. W standardowej grze obowiązują tylko trzy reguły:

- martwa komórka rodzi jeśli ma dokładnie 3 żywych sąsiadów,
- żywa komórka umiera z zatłoczenia, jeśli ma więcej niż 3 żywych sąsiadów,
- żywa komórka umiera z samotności jeśli ma mniej niż 2 żywych sąsiadów.

Przedstawione przez Conwaya proste zasady pozwoliły zachować równowagę pomiędzy rozrostem i zanikaniem struktur.

Gra w życie ma wiele ciekawych właściwości. Jedną z nich jest równoważność maszynie Turinga, co oznacza, że ma takie same możliwości obliczeniowe jak komputer z nieskończoną pamięcią i brakiem ograniczeń czasowych. Przy użyciu gry w życie możliwa jest konstrukcja bramek logicznych oraz implementacja różnych systemów komputerowych. Gra może przebiegać chaotycznie lub według jednego z ustalonych wzorców:

- stabilny – pozostają niezmienne bez względu na kolejne przekształcenia,
- oscylatory – zmieniają się cyklicznie, regularnie wracają do poprzednich stanów,
- statki – oscylatory zmieniające swoją pozycję na planszy.

Gra Conwaya jest przykładem automatu komórkowego, które często wykorzystywane są do przeprowadzania symulacji komputerowych. Max Brenner w swoim artykule [4] przedstawił symulację rozprzestrzeniania się wirusa COVID-19, co bardzo dobrze pokazuje możliwości, jakie dają nam modyfikacje gry w życie oraz bardziej rozbudowane automaty komórkowe.

1.2. Opis projektu

Przedstawiona praca opisuje projekt pozwalający na definiowanie zasad gry w życie, używając prostego języka. Gra odbywa się na planszy będącej diagramem Woronoja, co istotnie narusza jedną z właściwości automatów komórkowych, mianowicie komórki automatu różnią się od siebie. Pozwala to wprowadzić element nieregularności oraz upodabnia grę do prawdziwego życia, w którym elementy symulacji mają istotne różnice wynikające se swojej budowy lub środowiska. Weźmy za przykład ludzi, którzy mają różną liczbę osób w swoim otoczeniu, co wpływa na wiarygodność symulacji przeprowadzonych na standardowej siatce kwadratów.

Pierwszą częścią pracy jest przedstawienie algorytmu Fortunea pozwalającego na generowanie diagramów Woronoja. Algorytm przyjmuje na wejściu zbiór punktów oraz opierając się na technice zamiatania, generuje na ich podstawie diagram. Algorytm Fortunea rozpatruje względem współrzędnej y następujące wydarzenia:

- site event - miotła napotyka kolejny punkt z wejściowego zbioru,
- circle event - miotła napotyka wydarzenie będące niwelacją jednej z paraboli wykorzystywanej do wyznaczania krawędzi diagramu.

Po rozpatrzeniu wszystkich wydarzeń wynikiem działania algorytmu są krawędzie oraz graf sąsiedztwa pól diagramu Woronoja.

Drugą częścią pracy jest opis gramatyki oraz mechanizmu prasowania i ewaluacji języka opisu gry. Parser tworzonych przez użytkowników programów wykorzystuje bibliotekę PEGTL, która opiera się na gramatycę PEG oraz parsowaniu z góry na dół. Wynikiem parsowania jest abstrakcyjne drzewo rozkładu języka, które jest następnie ewaluowane do odpowiednich funkcji opisujących grę w życie.

Kolejna część to opis wykorzystania API OpenGl do przeprowadzenia wizualizacji gry.

Pracę uzupełnia opis istniejących narzędzi i algorytmów do przeprowadzania symulacji z użyciem gry w życie.

Rozdział 2.

Diagramy Woronoja

Rozdział opisuje podział płaszczyzny zwany diagramem Woronoja, oraz algorytm Fortunea służący do szybkiego generowania diagramów. Uzupełnieniem rozdziału jest krótkie przedstawienie alternatywnych algorytmów.

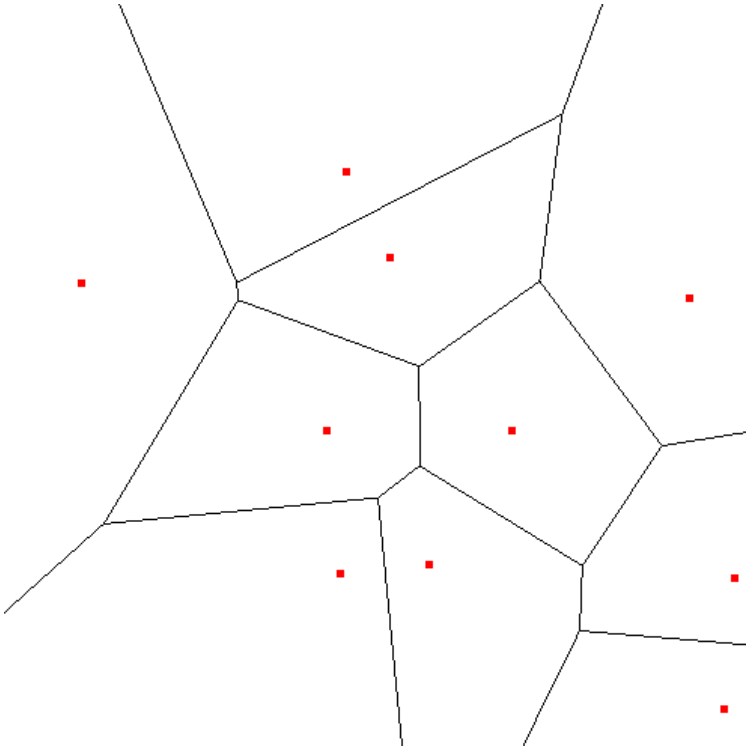
2.1. Definicja

Niech S będzie skończonym zbiorem parami różnych dwuwymiarowych punktów o współrzędnych rzeczywistych. Punkty zbioru S, nazywane będą centrami.

Definicja 1. *Polem diagramu Woronoja* odpowiadającemu punktowi $p \in S$ oznaczamy $Vor_S(p) = \{x \quad | \quad x \in \mathbb{R}^2 \wedge \forall p' \in S \quad dist(x,p) \leq dist(x,p')\}$, gdzie $dist$ oznacza odległość euklidesową dwóch punktów.

Definicja 2. *Diagram Woronoja*, to zbiór $\{Vor_S(p) \quad | \quad p \in S\}$.

Przykład 2.1. *Przykładowy diagram Woronoja z zaznaczonymi* 10 *centrami.*



2.2. Algorytm Fortunea

2.2.1. Opis algorytmu

Jednym ze sposobów generowania diagramów Woronoja jest algorytm zaproponowany przez Johna Fortunea oparty na technice zamiatania [7].

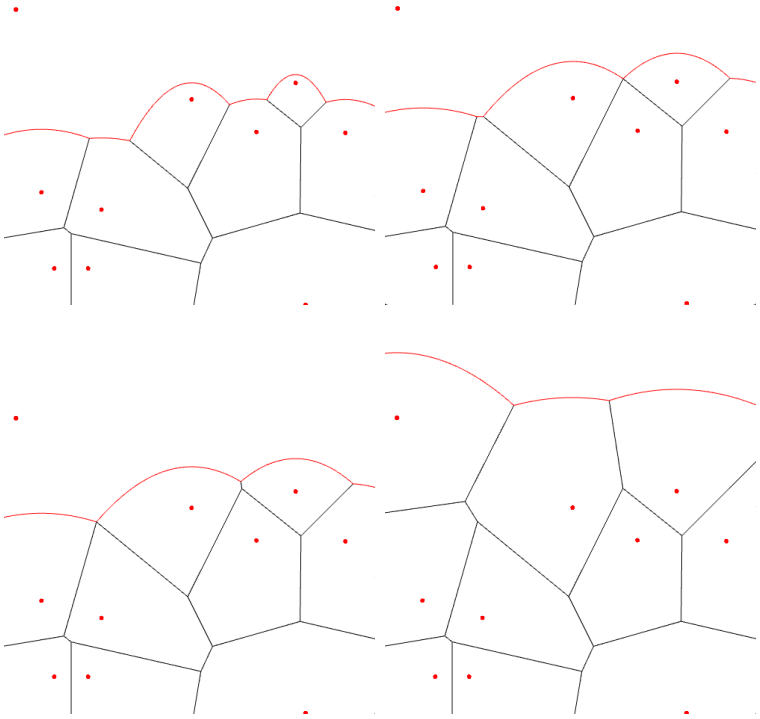
Definicja 3. *Miotłą* nazywamy poziomą linię poruszającą się od dołu do góry planszy. Miotła napotyka wydarzenia, w kolejności ich występowania. Wszystkie wydarzenia poniżej miotły zostały już rozpatrzone przez algorytm, natomiast te powyżej miotły zostaną rozpatrzone, gdy miotła dojdzie do ich wysokości.

Najważniejszą strukturą algorytmu jest towarzysząca miotle linia brzegowa. Składa się ona z parabol, rozpinających się nad rozpatrzonymi przez miotłę punktami oraz nieukończonych krawędzi diagramu. W czasie trwania algorytmu parabole rozszerzają się, a ich przecięcia wyznaczają krawędzie. Parabola, która tworzy pole z centrum $p = (x_p, y_p)$ jest zbiorem $\{x \quad | \quad x \in \mathbb{R}^2 \wedge dist(p, x) = dist(p_s, x)\}$, gdzie przez p_s oznaczamy punkt na miotle znajdujący się najbliżej x.

Poruszająca się przez planszę miotła napotyka dwa rodzaje wydarzeń.

- site event – dotarcie do nowego puntu ze zbioru S
- circle event – zniwelowanie jednej z parabol w linii brzegowej

Przykład 3.1. *Przykładowe kroki podczas trwania algorytmu.*



Każda parabole, zdefiniowana jest przez miotłę i jeden z punktów, będących centrami pól diagramu. Punkty leżące na paraboli są w równej odległości od centrum pola które je definiuje oraz najbliższego punktu na miotle.

Wyznaczymy wzór $f(x)$ parabol znajdujących się w linii brzegowej. Niech y_s oznacza wysokość, na której znajduje się miotła, a (x_c, y_c) będzie centrum pola, które definiuje parabolę.

$$
\begin{aligned}
dist((x_c, y_c), (x, f(x))) &= dist((x, y_s)(x, f(x))) \\
\sqrt{(x_c - x)^2 + (y_c - f(x))^2} &= \sqrt{(x - x)^2 + (y_s - f(x))^2} \\
(x_c - x)^2 + (y_c - f(x))^2 &= (y_s - f(x))^2 \\
(x_c - x)^2 &= (y_s - f(x))^2 - (y_c - f(x))^2 \\
(x_c - x)^2 &= 2f(x)y_c - 2f(x)y_s + y_s^2 - y_c^2 \\
(x_c - x)^2 + (y_c^2 - y_s^2) &= 2f(x)(y_c - y_s) \\
f(x) &= \frac{(x - x_c)^2}{2(y_c - y_s)} + \frac{y_c^2 - y_s^2}{2(y_c - y_s)} \\
f(x) &= \frac{(x - x_c)^2}{2(y_c - y_s)} + \frac{y_c + y_s}{2}
\end{aligned}
$$

Korzystając z otrzymanego wzoru, możemy w łatwy sposób wyliczać między innymi przecięcia parabol z innymi obiektami w linii brzegowej.

Site event – obsługa

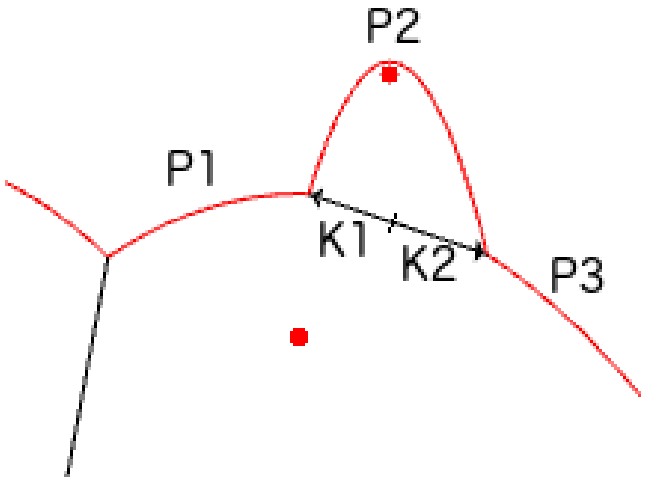
Dodawanie nowego centrum do algorytmu, oznacza uwzględnienie go w wynikowym diagramie. Wszystkie punkty, które znajdują się pod linią miotły, zostały już uwzględnione w działaniu algorytmu. Napotykając nowe centrum, musimy dodać do linii brzegowej nową parabolę, dzieląc jedną z istniejących na dwie części. W tym celu przeszukujemy linię brzegową, poszukując paraboli leżącej bezpośrednio pod nowym centrum. To przeszukiwanie będzie miało kluczowy wpływ na złożoność algorytmu.

Dodając nowe pole, zmieniamy w linii brzegowej jedną parabolę (leżącą pod nowym centrum) na następujący ciąg elementów.

$$P1, K1, P2, K2, P3 \tag{2.1}$$

Gdzie $P1, P3$ to kopie rozdzielanej paraboli. $P2$ to nowa parabola utworzona przez dodawane centrum i miotłę. $K1, K2$ to nowe krawędzie rozszerzające się w przeciwnych kierunkach. Punktem startowym dodawanych krawędzi jest punkt na rozdzielanej paraboli znajdujący się pod nowym centrum.

Przykład 3.2. *Zmienione elementy linii brzegowej po obsłużeniu nowego centrum.*



Pozostaje sprawdzić, czy parabole, które dodaliśmy do linii brzegowej, zostaną kiedyś zniwelowane i czy konieczne jest dodanie odpowiednich wydarzeń. Parabola zostanie zniwelowana przez swoich sąsiadów, jeśli zachodzą dwa warunki:

- parabola nie jest skrajnie lewym lub skrajnie prawym elementem linii brzegowej,

- krawędzie (półproste) będące sąsiadami paraboli, przecinają się.

Aby dodać nowe wydarzenie polegające na ściśnięciu paraboli przez jej sąsiadów, musimy poznać jego współrzędną y. Weźmy ciąg elementów linii brzegowej:

$$P1, K1, P2, K2, P3 \tag{2.2}$$

Punkt $p_i = (x_i, y_i)$ będący punktem przecięcia krawędzi $K1$ i $K2$ jest też miejscem przecięcia parabol $P1$ i $P3$, co czyni y_i wysokością, na której $P2$ zostanie ściśnięte przez $P1$ i $P3$. W tym momencie p_i będzie równo odległe od centrów parabol $P1, P2, P3$. Centra te będą leżały na okręgu o środku p_i, skąd wzięła się nazwa wydarzenia.

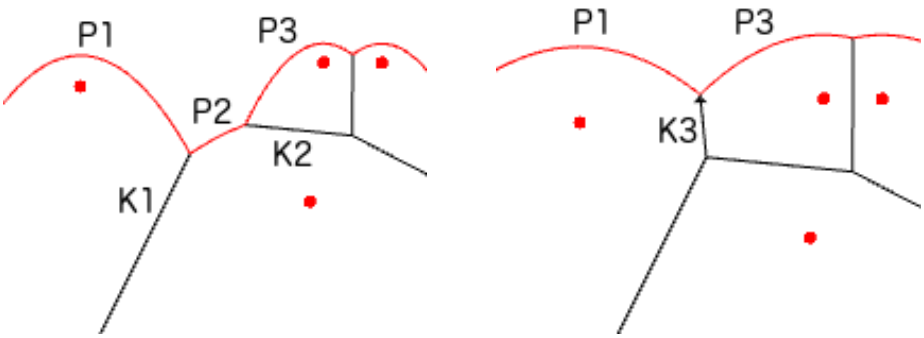
Circle event – obsługa

Rozpatrzmy następujący ciąg elementów w linii brzegowej:

$$P1, K1, P2, K2, P3 \tag{2.3}$$

Załóżmy, że parabola $P2$ zostaje ściśnięta i musimy usunąć ją z linii brzegowej. Leżące po jej bokach krawędzie $K1, K2$ nie będą już rosły. Powinny zostać usunięte i oznaczone jako gotowe krawędzie diagramu. Pozostaje dodać nową krawędź pomiędzy parabolami $P1$ i $P3$ oraz podobnie jak podczas obsługi nowego centrum sprawdzić, czy konieczne jest dodanie wydarzeń niwelacji $P1$ i $P3$.

Przykład 3.3. *Obsłużone wydarzenie, usunięto parabolę $P2$ oraz krawędzie $K1, K2$. W ich miejsce dodana została krawędź $K3$.*



2.2.2. Implementacja

Pseudokod

Poniższy pseudokod pokazuje, bardzo uproszczony sposób rozpatrywania wydarzeń. Wydarzenia trzymane są na stosie, zaimplementowanym przy użyciu *priority_queue*

ze standardowej biblioteki języka $C++$.

Algorithm 1: Pseudokod algorytmu

```
Kolejka wydarzeń eventQueue zawiera wszystkie wydarzenia typu site event.
while eventQueue nie jest pusta do
    currentEvent = eventQueue.top();
    if currentEvent jest typu site event then
        handleSiteEvent(currentEvent);
    else
        handleCircleEvent(currentEvent);
    end
end
```

Linia brzegowa

Linia brzegowa – *Beachline*, jest najważniejszą strukturą całego algorytmu. Zawiera ona listę struktur *BeachlineField*, przechowującą elementy znajdujące się w linii brzegowej. Wszystkie funkcje, których złożoność nie została określona, działają ze stałą złożonością. Najważniejsze funkcje *Beachline* to:

- *findParabolaForNewSite* – funkcja wyszukująca parabolę z linii brzegowej, która znajduje się bezpośrednio pod nowym centrum. Funkcja oblicza przecięcia parabol, z ich sąsiadami co pozwala poznać przedział, na jakim się one znajdują. Złożoność tej funkcji to $O(n)$, gdzie n to rozmiar linii brzegowej.
- *handleSiteEvent* – funkcja wykonująca niezbędne operacje opisane w Site event – obsługa. Złożoność tej funkcji to $O(n)$, gdzie n to rozmiar linii brzegowej.
- *handleCircleEvent* – funkcja wykonująca niezbędne operacje opisane w Circle event – obsługa.
- *addCircleEvents* – funkcja dodająca niezbędne wydarzenia typu *circle event*, dla nowych elementów linii brzegowej.

Beachline zawiera dużo funkcji pomocniczych, pomagających obsługiwać wydarzenia oraz wykonywać niezbędne obliczenia na parabolach i krawędziach. Niektóre z tych funkcji to:

- *addNewField* – funkcja dodająca nowy element do linii brzegowej,
- *removeField* – funkcja usuwająca element z linii brzegowej,
- *parabolaHalfEdgeIntersection* – funkcja wyznaczająca punkt przecięcia paraboli z półprostą,
- *edgesIntersection* – funkcja wyznaczająca przecięcie półprostych,

- *completePolygons* – funkcja czyszcząca linię brzegową po zakończonej pracy oraz oznaczająca ostatnie krawędzie jako zakończone. Złożoność tej funkcji to $O(n)$, gdzie n to rozmiar linii brzegowej.

Beachline przechowuje także wynikowe krawędzie algorytmu oraz uzyskany graf połączeń pól diagramu.

Pozostałe struktury

- *Event* – przechowuje informacje dotyczące wydarzeń przechowywanych w *eventQueue*,
- *Parabola* – reprezentuje parabolę przechowywaną w linii brzegowej,
- *HalfEdge* – reprezentuję nieukończoną krawędź (półprostą), która rozszerza się w trakcie trwania algorytmu,
- *Edge* – reprezentuje ukończona krawędź.

2.2.3. Złożoność

Obsługa każdego wydarzenia typu *site event* dodaje dwie nowe krawędzie oraz dwie nowe parabole do linii brzegowej. Załóżmy, że każda parabola ma przypisane wydarzenie typu *circle event*. Obsługa wydarzenia *circle event* zmniejsza liczbę parabol oraz krawędzi. Oznacza to, że dla n pól diagramu w linii brzegowej będzie $O(n)$ elementów oraz w *eventQueue* będzie $O(n)$ wydarzeń. Ze względu, na złożoność wyszukiwania parabol w linii brzegowej – $O(n)$, złożoność całego algorytmu to $O(n^2)$. Możliwe jest osiągnięcie złożoności $O(nlog(n)$, przez zastosowanie zbalansowanego drzewa binarnego do przechowywania elementów linii brzegowej.

2.3. Inne algorytmy

2.3.1. Algorytm dziel i zwyciężaj

Innym podejściem do generowania diagramów Woronoja jest zastosowanie techniki dziel i zwyciężaj. Podczas każdego kroku algorytmu dzielimy zbiór, z którego generujemy diagram na dwa równe, rozłączne podzbiory. Po rekurencyjnym wywołaniu algorytmu i wygenerowaniu diagramu dla mniejszych zbiorów wynikowe diagramy są ze sobą łączone. Złożoność takiego algorytmu to $O(nlogn)$.

2.3.2. Algorytm inkrementacyjny

W tym podejściu do gotowego diagramu Woronoja dodajmemy kolejno nowe punkty, aktualizując istniejący diagram. Początkowo dla jednego punktu diagramem jest ciała płaszczyzna. Złożoność algorytmu to $O(n^2)$.

2.3.3. Triangulacja Delone

Definicja 4. Triangulacja Delone zbioru P oznaczana jako $DP(P)$ to taka triangulacja, że żaden punkt $p \in P$ nie leży w środku okręgu opisanego na dowolnym trójkącie należącym do $DP(P)$.

Algorytmy wyznaczające triangulację Delone działają w złożoności $O(nlogn)$. Korzystając z wyznaczonej triangulacji, można w łatwy sposób uzyskać diagram Woronoja, łącząc punkty okręgów opisanych na trójkątach.

Rozdział 3.

Język opisu gry

Standardową grę w życie można opisać w bardzo prosty sposób. Potrzebne są nam informacje o tym, kiedy komórka pozostaje żywa i kiedy ożywa, będąc wcześniej martwa. Możemy z tego wywnioskować, kiedy komórka umiera, co daje nam komplet niezbędnych reguł. Zasady przedstawiamy według następującej konwencji:

- na początku podajemy liczby żywych sąsiadów, dla których komórka pozostaje żywa,
- następnie po ukośniku podajemy liczby żywych sąsiadów, dla których komórka ożywa, będąc wcześniej martwa.

Podane liczby są mniejsze lub równe 8, przez co nie ma potrzeby w żaden sposób ich oddzielać.

Przykład 4.1. 23/3 *to opis zasad gry w życie zaproponowanej przez Conwaya.*

Niestety, nieregularność diagramów Woronoja oraz bardziej rozbudowany stan komórek nie pozwala na opisywanie gry w tak prosty sposób. W omawianym projekcie użytkownik definiuje zasady, przy użyciu prostego języka, którego składnia przypomina składnię języka *C*. Takie rozwiązanie pozwala na wykonywanie bardziej skomplikowanych obliczeń podczas przechodzenia między stanami oraz obsługę stanów będących krotkami liczb rzeczywistych, co pozwala na przechowywanie większej ilości informacji o komórce. Użytkownik definiuje także sposób w jaki komórka będzie wyświetlana. Korzystając z bieżącego stanu, określamy jej kolorowanie w postaci *RGB*.

Do zdefiniowania zasad, użytkownik powinien napisać trzy programy.

- program *INIT* – określa początkowe parametry gry. Wywoływany jest tylko raz, przed rozpoczęciem symulacji,
- program *TRANSITION* – określa w jaki sposób komórka zmienia swój stan. Wywoływany dla wszystkich komórek podczas każdego kroku gry,

- program *COLOR* – określa kolor danego pola, korzystając z jego stanu. Wywoływany dla wszystkich komórek podczas każdego kroku gry, po wykonaniu programu *TRANSITION*.

Symulacja gry w życie przebiega w następujący sposób.

Algorithm 2: Pseudokod algorytmu

```
Ustaw początkowe parametry, wykonując program INIT
while licznik kroków jest mniejszy od limitu kroków do
    for każda komórka do
        TRANSITION
        COLOR
    end
end
```

Przykład 4.2. *Przykładowy program INIT ustawiający kolor początkowy na biały oraz limit kroków na* 1000*.*

```
initialColor(1.0, 1.0, 1.0);
stepsLimit(1000);
```

Przykład 4.3. *Przykładowy program TRANSITION ożywiający komórkę, jeśli ma więcej niż jednego żywego sąsiada.*

```
if(count(0, 1) > 1)
{
   newState[0] = 1;
}
```

Przykład 4.4. *Przykładowy program COLOR kolorujący żywe komórki na szaro.*

```
if(state[0] == 1)
{
   color[0] = 0.5;
   color[1] = 0.5.
   color[2] = 0.5.
}
```

3.1. Dane początkowe

Dane do programu przekazujemy w pliku o rozszerzeniu *.csv*. Pierwszy wiersz pliku zawiera niebędące liczbami nazwy kolumn. Kolejne wiersze zawierają początkowe informacje o komórkach. Pierwsze dwie kolumny każdego wiersza, opisują współrzędne punktów służących do generowania diagramu Woronoja. Współrzędne muszą zawierać się w przedziale $[-1, 1]$. Kolejne kolumny zawierające dowolne liczby

rzeczywiste opisują początkowy stan komórek. Liczba podanych kolumn definiuje rozmiar stanu komórek. Liczby rzeczywiste w pliku wejściowym powinny być zapisane w formacie angielskim – część ułamkowa oddzielona kropką.

Przykład 4.5. *Przykładowy plik wejściowy opisujący współrzędne punktów oraz stan początkowy zawierający jedną liczbę.*

x	*y*	*alive*
0.150832	*-0.972647*	*1*
-0.0799638	*-0.631274*	*0*
0.535321	*-0.442343*	*0*
0.307639	*-0.247769*	*0*
0.704164	*0.757753*	*1*
-0.983095	*-0.128225*	*0*
-0.310227	*0.320775*	*0*
-0.998757	*0.349289*	*1*
0.155918	*-0.674355*	*1*
0.240373	*0.8879*	*0*

3.2. Konstrukcje językowe

3.2.1. Komentarze

Użytkownik pisząc programy, może umieścić w nich komentarze. Blok komentarza zaczyna się i kończy znakiem #.

Przykład 4.6. *# Tekst komentarza #*

3.2.2. Zmienne

Zmienne przechowują liczby typu rzeczywistego. Nazwy zmiennych tworzone są, tak jak w języku C. Użytkownik nie deklaruje zmiennych, a zmienna bez przypisanej wartości ma domyślnie wartość 0.

3.2.3. Atrybuty

Atrybuty to krotki przechowujące informacje o kolorze i stanie komórek. Dostęp do wartości atrybutów wykonujemy przez operator [] do którego podajemy liczby naturalne z przedziału $[0, n)$, gdzie n to rozmiar atrybutu. Podczas wykonywania napisanych przez użytkownika programów mamy dostęp do następujących atrybutów:

- *color* - krotka o rozmiarze 3, do której zapisywany jest kolor komórki (RGB), wyznaczony ze stanu w programie, *COLOR*.

- *state* – krotka z której odczytujemy aktualny stan komórki,
- *newState* – krotka do której zapisujemy wyliczony nowy stan komórki.

Użytkownik nie może wykorzystywać innych atrybutów. Rozmiar *state* i *newState* zależy od dostarczonych do programu danych początkowych.

3.2.4. Wywołanie funkcji

Wywołanie funkcji ma następującą formę:

$$nazwaFunkcji(argument1, [argument2, \dots]) \tag{3.1}$$

Argumentami funkcji mogą być liczby rzeczywiste lub naturalne. Wywołanie funkcji może być osobną instrukcją programu (wtedy wartość zwracana przez funkcję jest ignorowana) lub częścią wyrażenia. Podczas pisania programów, użytkownik ma do dyspozycji przygotowane wcześniej następujące funkcje:

- *count(index, value)* – zwraca liczbę sąsiadów komórki, których stan spełnia *state[index] == value*.
- *random(l, r)* – zwraca losową liczbę rzeczywistą z przedziału $[l, r]$. Funkcja korzysta z rozkładu jednostajnego.
- *stepsLimit(limit)* – ustawia limit kroków do wykonania. Wartość -1 oznacza nieskończoność.
- *initialColor(r, g, b)* – ustawia początkowy kolor komórek.
- *printEvery(value)* – ustawia co ile kroków wygląd planszy jest aktualizowany.
- *skipState(index, value)* – przerywa wykonywanie programu *TRANSITION* jeśli zachodzi *state[index] == value*. Dla tego pola pominięte zostanie też wykonanie programu *COLOR*.

Jeśli wywołanie funkcji nie jest częścią wyrażenia, musi być zakończone średnikiem.

3.2.5. Wyrażenia

W prezentowanym języku dostępne są operatory binarne: $+, -, *, /, \%, <, >, <=, >=, ==, \&\&, ||$. Operatory logiczne zwracają wartości 1 i 0, oznaczające odpowiednio prawdę i fałsz. Wyrażeniami mogą być następujące konstrukcje:

- liczba rzeczywista lub naturalna,

- wywołanie funkcji,
- zmienna lub dostęp do wartości atrybutu,
- operator binarny.

Argumentami operatora binarnego mogą być dowolne wyrażenia. Jeśli argumentem operatora binarnego jest inny operator binarny, taki argument musi znajdować się w nawiasach.

Przykład 4.7. *Przykładowe poprawne i błędne wyrażenia:*

- $varName$ – *poprawne wyrażenie,*
- $1 + varName$ – *poprawne wyrażenie,*
- $1.0 + (state[1] + varName)$ – *poprawne wyrażenie,*
- $1 + 2 * 4$ – *błędne wyrażenie,* $1 + 2$ *lub* $2 * 4$ *powinno znajdować się w nawiasie.*

3.3. Przypisania

Wyrażenia możemy przypisywać do zmiennych oraz atrybutów *color* i *newState*. Przypisanie musi być zakończone średnikiem.

Przykład 4.8. *Przykładowe poprawne przypisania:*

- $varName = (1 + varName2) * (10 - varName\_3);$,
- $color[2] = 0.5;$.

3.4. Instrukcje warunkowe

Warunkiem instrukcji if, może być dowolne wyrażenie. Blok instrukcji, które zostaną wykonane, jeśli warunek zostanie spełniony, może składać się tylko z instrukcji przypisania i musi być otoczony nawiasami $\{, \}$. Warunek jest spełniony, jeśli wartość wyrażenia jest różna od 0.

Przykład 4.9. *Przykładowa poprawna instrukcja warunkowa:*

```
if((condVar + 1) == 0)
{
   someVar = 2.0;
   newState[0] = 0;
}
```

3.5. Opis głównych programów

3.5.1. *INIT*

Program *INIT* jest miejscem, gdzie przy użyciu funkcji *printEvery, stepsLimit, initialColor* ustawiane są początkowe parametry symulacji.

3.5.2. *TRANSITION*

Program *TRANSITION* służy do definiowania zasad zmiany stanów komórek. Aktualny stan odczytujemy ze zmiennej *state*, która jest automatycznie ustawiana na wartości stanu każdej komórki. Wyliczony stan, który będzie aktualny w następnym kroku symulacji zapisujemy do zmiennej *newState*. Domyślnie wartości zmiennej *newState* są takie same jak te w zmiennej *state*.

3.5.3. *COLOR*

Program *COLOR* służy do wyliczania koloru komórek. Korzystając ze zmiennej *state*, wyliczany jest kolor w postaci RGB, który musi zostać zapisany do zmiennej *color*.

Rozdział 4.

Parsowanie i ewaluacja programów

4.1. Wykorzystanie biblioteki PEGTL

PEGTL to biblioteka pozwalająca na pisanie parserów, dostępna pod licencją MIT. Pozwala ona na proste zdefiniowanie gramatyki języka opisu zasad gry. Przetworzony program przechowywany jest w postaci abstrakcyjnego drzewa rozbioru, wykorzystując dostarczoną w tym celu strukturę.

Funkcje parsujące oraz definicja gramatyki znajdują się w plikach *Parsing.h* oraz *Parsing.cpp*.

4.2. Gramatyka

Biblioteka PEGTL pozwala na tworzenie gramatyk przy użyciu prostych reguł:

- one – dopasowuje pojedynczy znak,
- string – dopasowuje ciąg znaków,
- seq – dopasowuje ciąg zasad,
- star – dopasowuje dowolnie długi ciąg jednej reguły,
- plus – dopasowuje jedno lub więcej wystąpienie reguły,
- opt – opcjonalne dopasowanie reguły,
- sor – wybór pierwszej pasującej reguły w kolejności od lewej.

Wszystkie pozostałe dostępne reguły są kombinacjami podanych wyżej. Po zdefiniowaniu gramatyki, jako wejście parsera przekazywany jest ciąg znaków wczytany

z plików w katalogu *programs*. Z wczytanych programów usuwane są wszystkie białe znaki, co powoduje, że użytkownik może z nich dowolnie korzystać podczas pisania programów.

Przykład 4.10. *Reguły definiujące liczby.*

```
struct positiveInteger : plus < digit > {};

struct integer : seq <
   opt<one<'-'>>,
   positiveInteger
> {};

struct floatingPoint : seq <
   opt < one < '-' > >,
   plus < digit >,
   one < '.' >,
   plus < digit >
> {};
```

- Dodatnia liczba całkowita zdefiniowana jest przez niepusty ciąg cyfr.
- Liczba całkowita zdefiniowana jest jako dodatnia liczba całkowita poprzedzona opcjonalnym minusem.
- Liczba rzeczywista zdefiniowana jest jako opcjonalny minus, po którym następuje niepusty ciąg cyfr, kropka oraz ponownie niepusty ciąg cyfr.

4.3. Klasy Instruction i Expression

Abstrakcyjna klasa *Expression* jest podstawą struktury wyrażeń zdefiniowanych w pliku *Evaluation.h*. *Expression* zawiera wirtualną metodę *calculate*, która jest implementowana przez klasy dziedziczące. Pozwala to na łatwe obliczanie wartości rozbudowanych wyrażeń zawartych w programach.

Przykład 4.11. *Klasa dziedzicząca po Expression reprezentująca liczbę rzeczy rzeczywistą.*

```
class Number : public Expression {
    float number;
public:
    Number(float num) : number(num) { }
    float calculate()
    {
        return number;
    }
    ~Number() { }
};
```

Abstrakcyjna klasa *Instruction*, zawiera wirtualną metod *evaluate*, która jest odpowiedzialna za wykonanie danej instrukcji. Wszystkie klasy dziedziczące po *Instruction* implementują *evaluate*.

Przykład 4.12. *Klasa dziedzicząca po Instruction reprezentująca blok instrukcji do wykonania. Block wykorzystywana jest między innymi w klasie reprezentującej instrukcję warunkową.*

```
class Block : public Instruction {
    std::vector<Instruction*> block;
public:
    Block(std::vector<Instruction*> b) : block(b) { }
    void evaluate()
    {
        for(auto ins: block)
            ins->evaluate();
    }
    ~Block() { }
};
```

Struktura klas dziedziczących po *Expression* i *Instruction* pozwala na budowanie programów, które zawierają w sobie zasady w jaki sposób mają być wykonane. Dzięki takiemu rozwiązaniu, interpretowanie drzewa zwróconego przez bibliotekę *PEGTL* jest wykonywane tylko raz na początku symulacji.

4.4. Ewaluacja programów

Po uzyskaniu abstrakcyjnych drzew rozbiorów programów *INIT, TRANSITION, COLOR* zostaną one przetłumaczone na drzewo klas dziedziczących po klasach *Expression* i *Instruction*. Przetłumaczone programy są wywoływane w strukturze *Game*, która odpowiedzialna jest za wykonywanie symulacji. Progra, *INIT* ewaluowany jest jednokrotnie, przed rozpoczęciem symulacji. *COLOR* i *TRANSITION* ewaluowane są wielokrotnie, ze atrybutami ustawionymi na dane odpowiadające aktualnie przetwarzanej komórce diagramu.

Wyrażenia ewaluowane są rekurencyjnie, w sposób gorliwy. Oznacza to, że lewa strona wyrażenia jest wyliczana w całości, przed wyliczeniem prawej strony. Nazwy użytych w programach zmiennych przechowywane są w kontenerze *variable* będącym *unoredered_map*, ze standardowej biblioteki języka *C++*. Wszystkie przypisane do zmiennych wartości są czyszczone po wykonaniu każdego programu *COLOR* i *TRANSITION*.

Rozdział 5.

Wizualizacja

Pierwszym podejściem do wyświetlania symulacji było użycie Legacy OpenGL, który pozwala na wyświetlanie prostych kształtów na ekranie, bez użycia shaderów (programów uruchamianych na karcie graficznej, które opisują w jaki sposób na ekranie wyświetlany jest obraz) i dodatkowych struktur do przesyłania danych. Niestety takie rozwiązanie wiąże się z koniecznością przekazywania kompletu informacji o wyglądzie planszy po każdym kroku symulacji. Jest to bardzo nieefektywne, ponieważ podczas trwania programu kolory wielu pól zmieniają się rzadko, a krawędzie wielokątów pozostają niezmienne. Fakt ten, można wykorzystać stosując Vertex Buffer Object – VBO, który pozwala na przesłanie danych raz oraz późniejsze aktualizowanie ich jeśli zajdzie taka potrzeba.

Wyświetlanie obrazu podzielone jest na dwie części: wyświetlanie krawędzi wielokątów oraz kolorowanie wielokątów. Dane do wyświetlania ładowane są w funkcji *createBuffers* struktury *Board*, a za aktualizowanie wyglądu planszy, po wykonaniu kroku symulacji, odpowiedzialna jest funkcja *print*.

5.1. Shadery

Shadery podzielone są na dwa rodzaje:

- Vertex shader – obsługuję wyświetlanie wierzchołków danej figury, określając ich położenie oraz kolor.

- Fragment shader – opisuje w jaki sposób wyświetlane jest pole pomiędzy wierzchołkami. Kolor który przekazywany jest na wejściu fragment shadera jest interpolacją koloru wierzchołków figury. W celu uzyskania jednolitego koloru pola, kolor wszystkich wierzchołków musi być taki sam.

5.2. Wyświetlanie krawędzi

Krawędzie przekazywane są jako dwa punkty, będące ich końcami. Po zakończeniu generowania diagramu, krawędzie nie są nigdy modyfikowane, dlatego wyświetlane są tylko raz na początku wizualizacji. Za wygląd krawędzi odpowiadają dwa proste shadery.

5.2.1. Vertex shader

Podstawowy shader ustalający pozycję końców krawędzi.

```
layout(location = 0) in vec4 position;
void main()
{
   gl_Position = position;
}
```

5.2.2. Fragment shader

Shader wyświetlający ciało krawędzi w czarnym kolorze, ignorując kolor wierzchołków.

```
precision mediump float;
out mediump vec4 color1;
void main()
{
   color1 = vec4(0.0, 0.0, 0.0, 1.0);
}
```

5.3. Kolorowanie pól diagramu

Pola diagramów będące wielokątami wypukłymi podzielone zostały na trójkąty. Wierzchołkami każdego z trójkątów są dwa końce jednej z krawędzi oraz punkt będący centrum danego pola. Wielokąt przekazywany jest do VBO jako zbiór trójkątów. Każdy trójkąt reprezentowany jest przez trzy punkty razem z ich kolorami. Podczas trwania symulacji kolory pól ulegają zmianie. W takim przypadku w VBO aktualizowany jest kolor tylko zmienionych wielokątów, z wykorzystaniem funkcji *glBufferSubData*.

5.3.1. Vertex shader

Shader ustalający pozycję wierzchołków trójkąta oraz ustawiający ich kolor.

```
layout(location = 0) in vec4 position;
layout(location = 1) in vec3 colorIn1;
precision mediump float;
out mediump vec4 color2;
void main()
{
   gl_Position = position;
   color2 = vec4(colorIn1, 1.0);
}
```

5.3.2. Fragment shader

Shader wyświetlający ciało trójkąta w kolorze będącym interpolacją kolorów wierzchołków.

```
precision mediump float;
in mediump vec4 color2;
out mediump vec4 color3;
void main()
{
   color3 = color2;
}
```

Rozdział 6.

Przykładowe symulacje

6.1. Pulsujące pierścienie i pechowe pola

6.1.1. Stan

Stan opisany jest przez dwie liczby.

- state[0] – opisuje, czy komórka jest żywa,
- state[1] – licznik, mówiący od kiedy komórka powinna być żywa.

6.1.2. TRANSITION

```
cond1 = count(0, 1.0) > 0;
if(cond1 && (random(0.0, 1.0) > 0.5))
{
    newState[0] = 1;
}
if(cond1)
{
    newState[1] = state[1] + 1;
}
if(state[1] > 20)
{
    newState[1] = 0;
    newState[0] = 0;
}
```

Program sprawdza, czy pole ma żywego sąsiada, w takim przypadku startuje licznik oraz z prawdopodobieństwem 1/2 ożywia dane pole. Jeśli licznik przekroczy 20, pole umiera i licznik się zeruje.

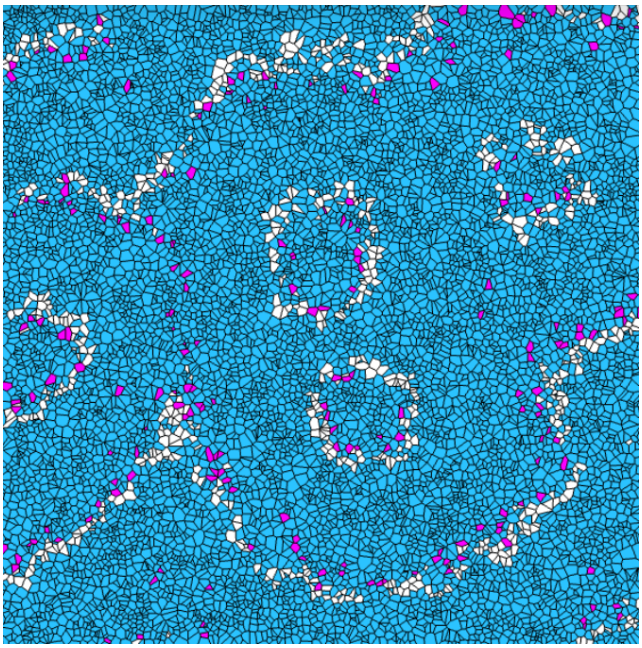
6.1.3. COLOR

```
if(state[0] == 0)
{
    # przypadek 1. #
    color[0] = 1.0;
    color[1] = 1.0;
}
if(state[0])
{
    # przypadek 2. #
    color[0] = 0.1;
    color[1] = 0.8;
}
if((state[1] > 1) && (state[0] == 0))
{
    # przypadek 3. #
    color[0] = 0.9;
    color[1] = 0.0;
}
```

Program *COLOR*, rozpatruje trzy przypadki:

- pole jest martwe – kolor biały,
- pole jest żywe – kolor niebieski,
- pole jest martwe, ale od więcej niż jednego kroku powinno być żywe – kolor fioletowy.

Przykład 4.13. *Przykładowy krok symulacji dla* 10000 *komórek. Na początku żywe komórki ustawione zostały w* 4 *różnych pozycjach.*



6.2. Zalewanie przeszkód

6.2.1. Stan

Stan opisany jest przez jedną liczbę – *state[0]*.

- state[0] ma wartość 1 – czy komórka jest żywa,
- state[0] ma wartość 0 komórka jest martwa.
- state[0] ma wartość 2 komórka jest przeszkodą.

6.2.2. INIT

Program *INIT* wywołuje funkcję *initialColor*, ustawiając kolor początkowy na biały.

6.2.3. TRANSITION

```
skipState(0, 1);
if(count(0, 1) && (state[0] < 2))
{
   newState[0] = 1;
}
```

Program sprawdza, czy pole nie jest przeszkodą i ma żywego sąsiada, w takim przypadku pole ożywa. Jeśli pole jest żywe, pomijane jest wywołanie programu *COLOR*.

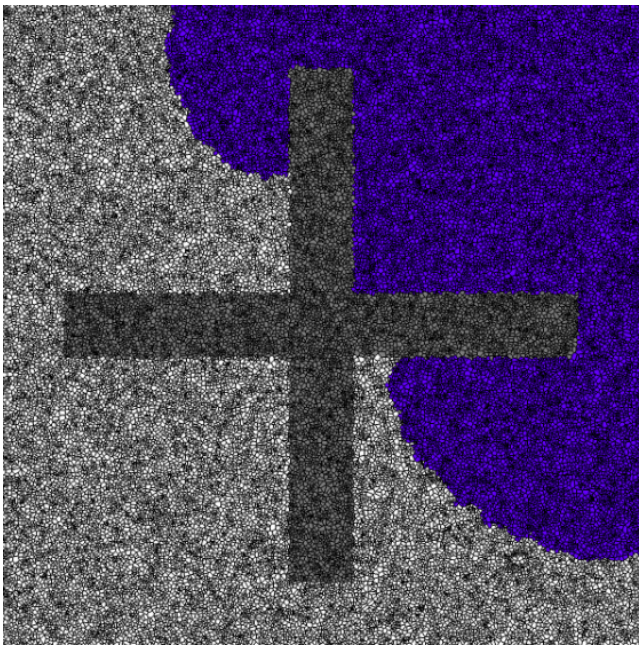
6.2.4. COLOR

```
if(state[0] == 1)
{
   color[0] = 0.0;
   color[1] = 0.0;
}
if(state[0] == 2)
{
   color[0] = 0.5;
   color[1] = 0.5;
   color[2] = 0.5;
}
```

Program *COLOR*, rozpatruje trzy przypadki:

- pole jest martwe – kolor biały,
- pole jest żywe – kolor niebieski,
- pole jest przeszkodą – kolor szary.

Przykład 4.14. *Przykładowy krok symulacji dla* 50000 *komórek. Przeszkody ułożone zostały w krzyż.*



6.3. Przydatne techniki

Kolejny przykład demonstruje użycie kilku technik w programie *TRANSITION*.

Opis stanu:

- state[0], state[2] – liczby naturalne przechowujące informację o stanie pola,
- state[1] – losowa liczba opisująca właściwość pola,
- state[3] – koniunkcja state[0] i state[2], przydatna do użycia zasady włączeń i wyłączeń.

6.3.1. COLOR

Program *COLOR* rozpatruje trzy przypadki, przyporządkowując kolory: czerwony, pomarańczowy i żółty.

```
if((state[0] > 0) && (state[2] == 0))
{
    color[0] = 1.0;
    color[1] = 0.0;
    color[2] = 0.0;
}
if((state[0] > 0) && (state[2] == 1))
{
    color[0] = 1.0;
    color[1] = 0.5;
    color[2] = 0.0;
}
if((state[0] < 1) && (state[2] == 1))
{
    color[0] = 1.0;
    color[1] = 1.0;
    color[2] = 0.0;
}
```

6.3.2. TRANSITION

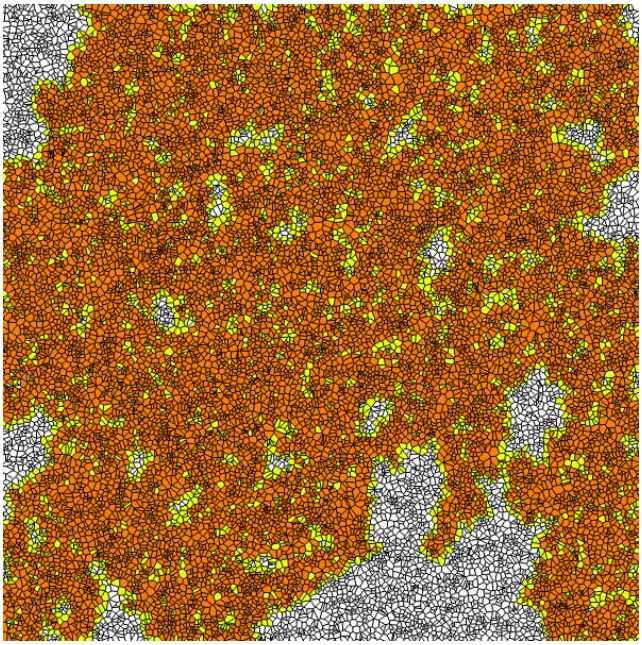
```
const1 = 0.5;
const2 = 0.1;
const3 = 1.5;

state1 = count(0, 1);
state3 = count(3, 1);
dif = state1 - state3;

if((state1 > 0) && (random(0, 1) > const2))
{
    newState[2] = 1;
}
ppd = 0.7;
if(state[1] > const1)
{
    ppd = 0.2;
}
cond = ((dif + 1.0) * random(0.0, 1.0)) > const3;
if(cond && (state[0] == 0))
{
    newState[0] = 1;
}
if((state[0] == 1) && (random(0, 1) > const2))
{
    newState[2] = 1;
    newState[3] = 1;
}
```

Przydatne techniki:

- Deklaracja stałych wartości na początku programu.
- Użycie dodatkowej wartości w stanie do trzymania koniunkcji wartości.
- Upraszczanie wyrażeń przez przypisywanie ich części do zmiennych.
- Losowanie właściwości pola przez podawanie losowych początkowych wartości stanu (podanych w pliku wejściowym).

Przedstawiona symulacja jest bardzo podatna na zmiany stałych wartości w programie. Najciekawsze wizualnie efekty uzyskane są przez powyższe ustawienie zmiennych.

Przykład 4.15. *Przykładowy krok symulacji dla* 20000 *komórek. Pola koloru pomarańczowego rozprzestrzeniają się w losowych kierunkach.*



Rozdział 7.

Inne narzędzia

Gra w życie zyskała dużą popularność. Od czasu jej publikacji powstało wiele narzędzi pozwalających przeprowadzać symulację różnych jej wariantów. Znaleźć można zarówno bardzo proste, interaktywne narzędzia pozwalające na przedstawienie podstawowych zasad działania gry oraz zaprezentowanie zachodzących zjawisk takich jak oscylatory, jak i zaawansowane programy naukowe, które wykonują tysiące kroków na sekundę i pozwalają w prawie dowolny sposób modyfikować zasady gry.

7.1. PlayGameOfLife

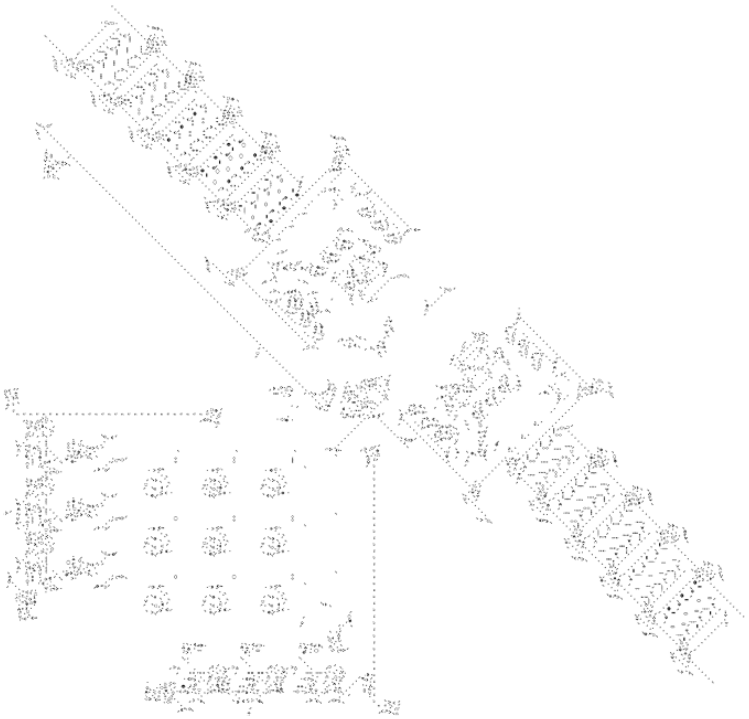
Jest to narzędzie symulujące podstawową grę w życie przedstawioną przez Conwaya. Dane wprowadzane są przez użytkownika ręcznie, zaznaczając żywe pola na interaktywnej planszy. Symulację przeprowadzamy online, pod adresem `https://playgameoflife.com/`. Dodatkowe możliwości:

- modyfikacja planszy w trakcie trwania symulacji
- zatrzymywanie symulacji i wykonywanie pojedynczych kroków
- zmiana odstępu czasowego pomiędzy krokami

7.2. Golly

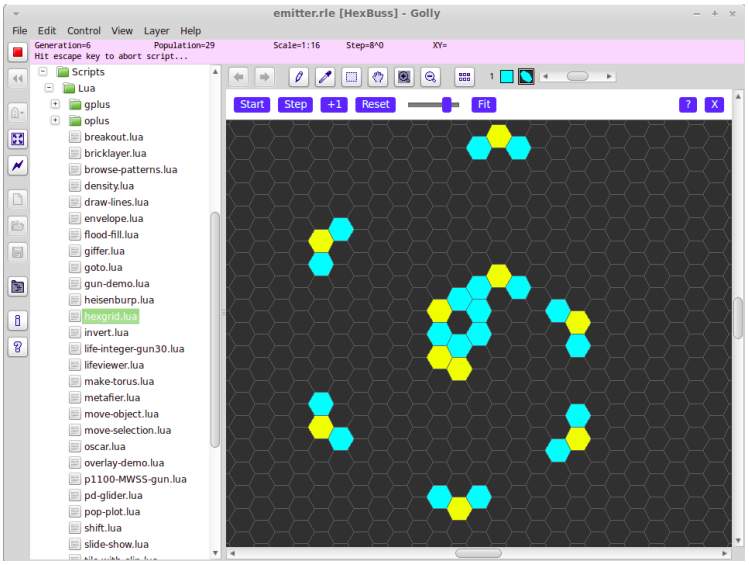
Golly [2], to narzędzie stworzone do symulacji automatów komórkowych. Skrypty obsługujące symulację są pisane w językach *Python* lub *Lua*. Dzięki zastosowaniu algorytmu *hashlife* [3] Golly jest w stanie przetwarzać duże struktury składające się z milionów komórek. Przykładami pokazującymi możliwości tego narzędzia są zaimplementowane przy jego użyciu systemu komputerowe wykorzystujące równoważność gry w życie z maszyną Turinga [5].

Przykład 4.16. *Maszyna Turinga stworzona przez Paula Rendella.*



Golly dostarcza także dużą bibliotekę przykładowych gier w życie, prezentujących jej najciekawsze zachowania.

Przykład 4.17. *Jeden z przykładów zaprezentowanych przez twórców – gra w życie odbywająca się na planszy złożonej z sześciokątów.*



7.2.1. Algorytm hashlife

Algorytm *hashlife* reprezentuje planszę jako drzewo, którego każdy wierzchołek ma czterech synów. Początkowo plansza dzielona jest na cztery równe kwadratowe części, następnie procedura powtarzana jest dla czterokrotnie mniejszych części planszy. Rozpoznając powtarzające się fragmenty w otrzymanym drzewie, niektóre krawędzie są przepinane w taki sposób, żeby występujące na planszy wzory ułożeń komórek występowały w strukturze tylko raz. Pozwala to zaoszczędzić wykorzystywaną przez algorytm pamięć. Wynik przekształcenia każdego wierzchołka wyliczany jest jednorazowo, a następnie zapamiętywany. Algorytm wykorzystuje ważną własność automatów podobnych do gry w życie. Nawet pozornie losowe struktury automatów komórkowych, mogą zamienić w się zbiór oscylatorów i niezmiennych fragmentów. Istotnymi minusami, są wykorzystywana przez algorytm pamięć oraz początkowy czas działania, niezbędny do zgromadzenia danych o symulacji.

Rozdział 8.

Podsumowanie

8.1. Woronoja

Pierwszą wyzwaniem podczas pracy nad projektem była odpowiednia implementacja algorytmu *Fortunea*, generującego diagramy Woronoja. Niezbędna geometria wymaga wykonywania obliczeń na liczbach rzeczywistych oraz rozpatrzenie wielu przypadków brzegowych. Koncepcja wszystkich obliczeń jest bardzo prosta ale liczba obsługiwanych podczas trwania algorytmu krawędzi i parabol sprawia, że nawet drobny błąd obliczeniowy, może prowadzić do całkowicie zniekształconego diagramu. Obecna wersja implementacji pozwala na bezproblemowe generowanie diagramu posiadającego więcej niż 100000 pól w czasie kilku sekund.

8.2. Wizualizacja

Podczas rozbudowywania programu natknąłem się na kolejny duży problem. Pierwsza wersja wizualizacji wykorzystująca *legacy OpenGl* okazała się być zbyt wolna. Symulacja prostej gry na planszy zawierającej kilka tysięcy pól była niepłynna, a oczekiwanie na dalsze kroki zbyt długie. Problem udało się rozwiązać wykorzystując współczesne techniki dostępne w OpenGl. Przesyłanie informacji o wyglądzie planszy tylko w momencie, gdy jest to konieczne pozwoliło osiągnąć płynność nawet dużych symulacji. W tym miejscu pozostaje potencjał do poprawy. Wykorzystując *Vertex Buffer Array*, możliwe jest zredukowanie redundancji przesyłanych danych, co powinno dodatkowo przyśpieszyć wizualizację.

8.3. Język opisu gry

Grę w życie zaprezentowaną przez Conwaya opisują trzy liczby. Taka prosta reprezentacja zasad, utrudniona jest przez nieregularność planszy, dlatego w przed-

stawionym projekcie, zasady implementowane są w prostym języku opisu gry. Parsowanie przygotowanych przez użytkownika programów, odbywa się przy użyciu biblioteki *PEGTL*. Wynikiem parsowania jest abstrakcyjne drzewo składni języka złożone z dostarczonych struktur. Odpowiednie interpretowanie wynikowego drzewa pozwala na wykonanie programów zawierających reguły gry. Niestety, takie rozwiązanie okazało się być zbyt wolne. Cały zysk, otrzymany przez poprawę wizualizacji, został wykorzystany na interpretowanie programów.

Proces wykonywania symulacji został przyśpieszony, tłumacząc otrzymane podczas parsowania drzewa, na wewnętrzną reprezentację będącą drzewem opierającym się na dwóch abstrakcyjnych klasach *Expression* i *Instruction* oraz klasach po niech dziedziczących. Takie rozwiązanie pozwala na jednorazowe rozpatrywanie wielu przypadków, niezbędnych do interpretacji programów napisanych przez użytkownika. Po konwersji do nowej reprezentacji, wszystkie informacje potrzebne do wykonania instrukcji, są już dostępne w wierzchołkach drzewa.

Język opisu gry jest też częścią projektu, w której są największe możliwości rozwoju. Dodając nowe reguły do gramatyki parsowanego języka oraz nowe klasy do struktury reprezentującej zinterpretowane drzewo, można w łatwy sposób dodać nowe funkcjonalności. Jednym z przykładów funkcjonalności niewprowadzonych do języka są stałe wartości definiowane w programie *INIT* lub dodatkowe funkcje matematyczne. Oczywiście, możliwości jest więcej i późniejsze rozwijanie zależne jest od potrzeb użytkowników.

Bibliografia

[1] Martin Gardner. Mathematical games - the fantastic combinations of john conway's new solitaire game life. *Scientific American*, 223:120–123, 1970.

[2] Golly contributors. Golly, 2020. [Online; accessed 08-January-2021].

[3] R.Wm. GOSPER. Exploiting regularities in large cellular spaces. *Physica D*, 10:75–80, 1984.

[4] Max Brenner. Simulating covid-19 with cellular automata, 2020. [Online; accessed 29-August-2020].

[5] Paul Rendell. Conway's game life turing machine, 2001. [Online; accessed 12-January-2021].

[6] Wikipedia contributors. Diagram woronoja, 2020. [Online; accessed 17-August-2020].

[7] Wikipedia contributors. Sweep line algorithm, 2020. [Online; accessed 17-August-2020].