Elementarni stanični automati

Lukšić, Hrvoje

SVEUČILIŠTE U ZAGREBU
FAKULTET ORGANIZACIJE I INFORMATIKE
VARAŽDIN

Hrvoje Lukšić

ELEMENTARNI STANIČNI AUTOMATI

ZAVRŠNI RAD

Varaždin, 2023.

SVEUČILIŠTE U ZAGREBU

FAKULTET ORGANIZACIJE I INFORMATIKE

V A R A Ž D I N

Hrvoje Lukšić

Matični broj: 35918/07–R

Studij: Informacijski i poslovni sustavi, Razvoj programskih sustava

ELEMENTARNI STANIČNI AUTOMATI

ZAVRŠNI RAD

Mentor :

Izv. prof. dr. sc. Mario Konecki

Varaždin, lipanj 2023.

Hrvoje Lukšić

Izjava o izvornosti

Izjavljujem da je moj završni rad izvorni rezultat mojeg rada te da se u izradi istoga nisam koristio drugim izvorima osim onima koji su u njemu navedeni. Za izradu rada su korištene etički prikladne i prihvatljive metode i tehnike rada.

Autor potvrdio prihvaćanjem odredbi u sustavu FOI-radovi

Sažetak

Elementarni stanični automati najjednostavnija su klasa jednodimenzionalnih staničnih automata u kojima stanje ćelije (1 ili 0) u budućim generacijama ovisi samo o vlastitom i stanju njenih susjednih ćelija. Unatoč vrlo jednostavnom skupu pravila, pokazuju zanimljiva ponašanja i u nekim slučajevima omogućuju čak i univerzalno računanje, odnosno čine Turingov stroj. [1] U ovom radu proučit ćemo njihovo ponašanje razvojem programa koji simulira i crta bilo koji elementarni stanični automat u proizvoljnoj veličini i rezoluciji s raznim izborima početnih stanja i izračunati vremensku složenost korištenog algoritma.

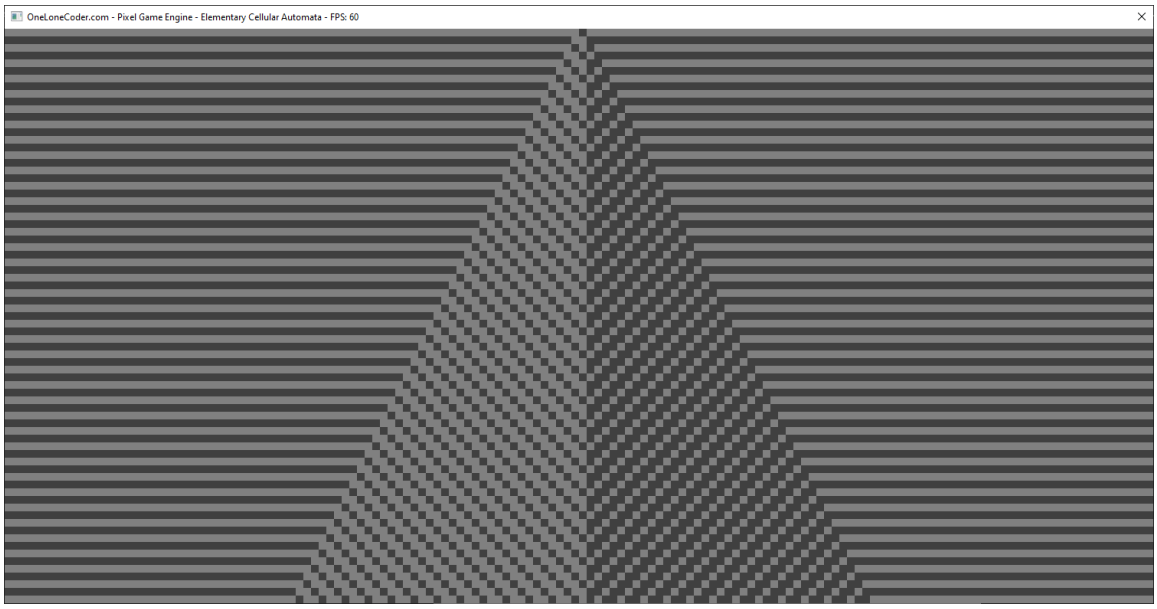
Ključne riječi: Teorija automata, stanični automat, elementarni stanični automat, evolucija

Sadržaj

1. Uvod

Elementarni stanični automati jednostavni su, ali fascinantni računski modeli koji rade na **jednodimenzionalnom** skupu ćelija. Oni su definirani **skupom ishoda** koji određuju stanje pojedine ćelije u budućoj generaciji automata s obzirom na vlastito i stanje njenih susjednih ćelija. Unatoč jednostavnosti, mogu ostvariti neobično složena ponašanja te su zbog toga vrlo dobro istraženo područje računalne znanosti.

U elementarnim staničnim automatima, svaka ćelija u **polju** (engl. *array*) može biti **živa** ili **mrtva** (u slučaju našeg programa, bit će obojana svjetlije ili tamnije). Evolucija automata događa se u **koracima**, odnosno svaka nova generacija se ispisuje na ekran ispod prethodne stvarajući nakon određenog broja koraka potpunu sliku njegova ponašanja.



Slika 1: Prvih 75 generacija pravila 57, Izvor: vlastita slika

2. Metode i tehnike rada

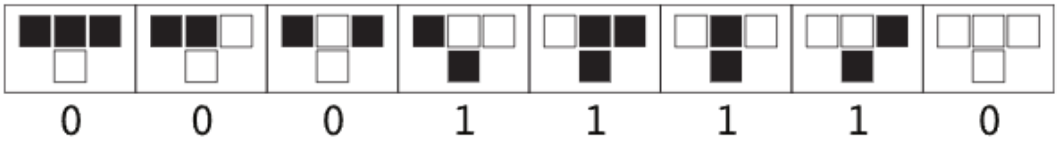
Pri razradi teme korišteni su pretežito izvori s *Wolfram MathWorld*a [2] i knjige *A New Kind of Science* [3] čiji je autor Stephen Wolfram jedan od pionira u istraživanju staničnih automata. Za razvoj programskog rješenja korišteni su sljedeći alati:

- Razvojno okružje *Visual Studio 2022*
- Programski jezik C++20
- *Git* i *Github* [4]
- Grafička biblioteka *olcPixelGameEngine* [5]

3. Razrada teme

3.1. Sustav imenovanja

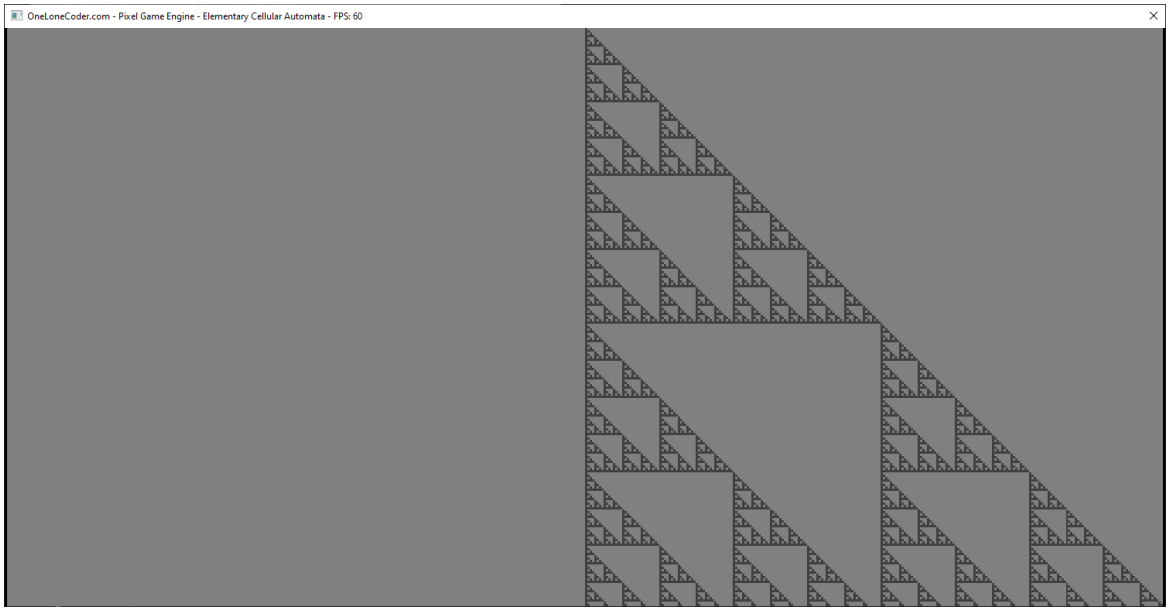
Pravilo koje određuje automat temelji se na $2 * 2 * 2 = 8$ mogućih početnih binarnih stanja tri ćelija koje se uzimaju u obzir. Budući da svako od osam stanja ima definiran ishod (srednja ćelija je u sljedećoj generaciji živa ili mrtva), pravila se indeksiraju pomoću 8-bitnog binarnog broja koji se prevodi u dekadski, dakle postoji ukupno $2^8 = 256$ elementarnih staničnih automata koji se nazivaju po uzorku *pravilo* [0-255]. [1]



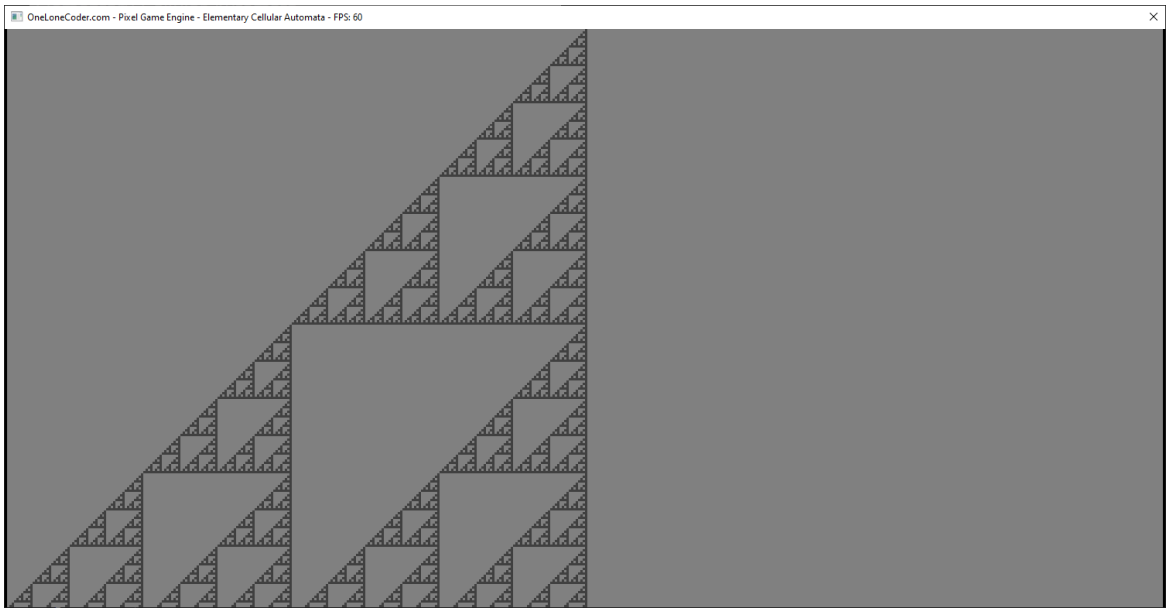
Slika 2: Grafički prikaz ishoda pravila 30 (00011110), Izvor: `https://mathworld.wolfram.com/ElementaryCellularAutomaton.html`

3.2. Zrcalna i komplementarna pravila

Od ukupno 256 pravila, mnoga su trivijalno **ekvivalentna** nakon jednostavnih transformacija. Zrcalna pravila obrnuta su po osi *y*, a 64 pravila su **akiralna** odnosno identična svom zrcalnom pravilu. [1] Pravilu 60 je primjerice zrcalno pravilo 102. [6]

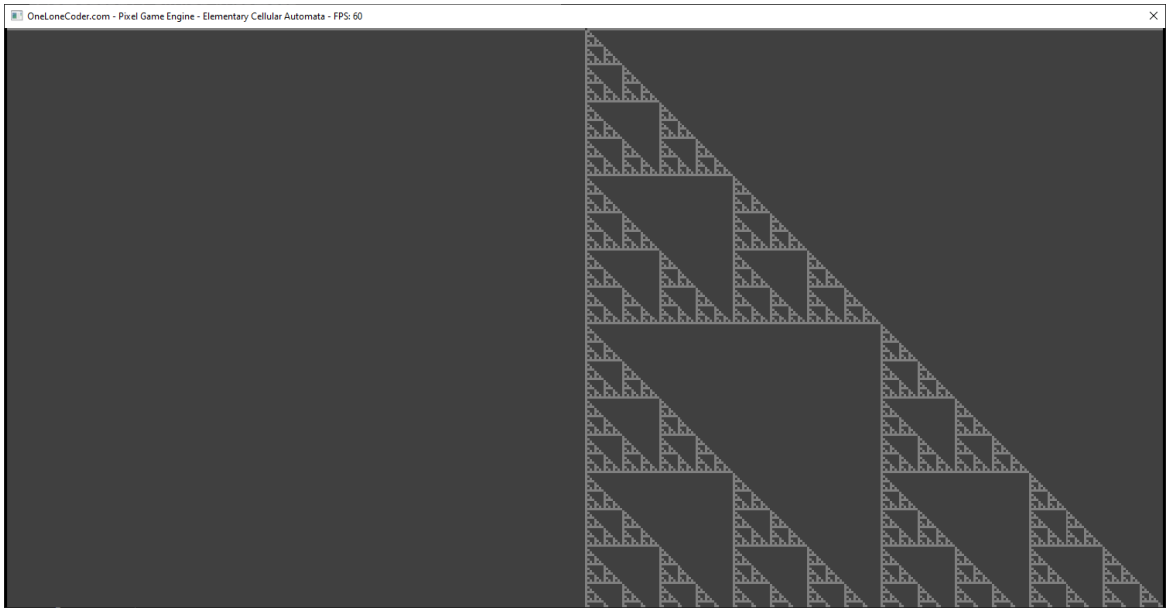


Slika 3: Pravilo 60 iz središnje početne ćelije, Izvor: vlastita slika



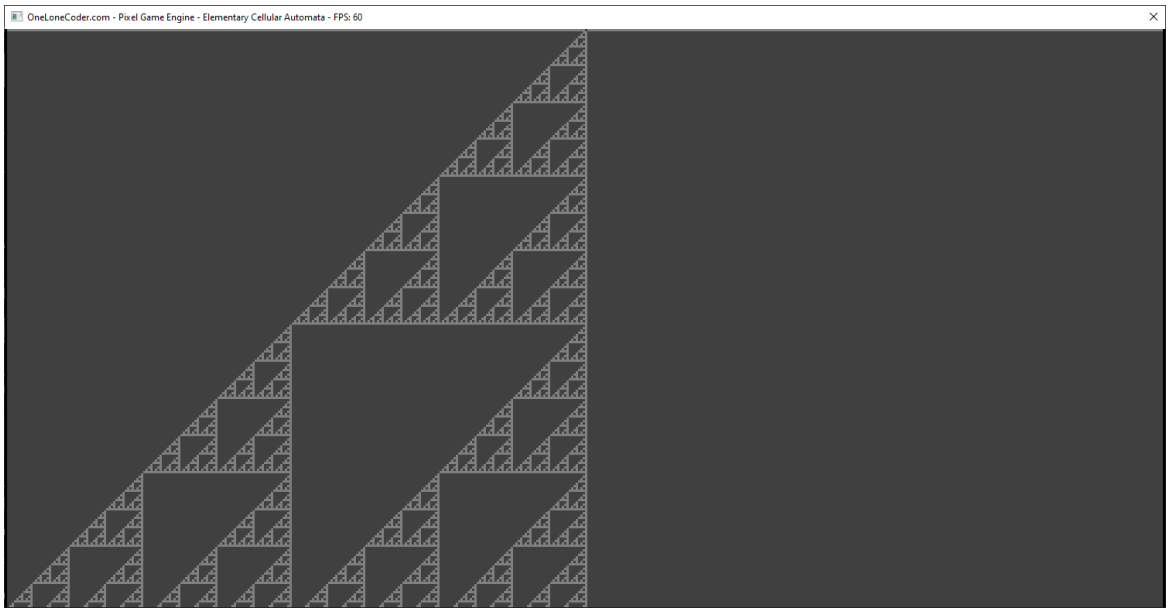
Slika 4: Pravilo 102 iz središnje početne ćelije, Izvor: vlastita slika

Komplementarnim pravilima su žive i mrtve stanice zamijenjene (imaju učinak inverzije boja), a 16 pravila je identično svojim komplementima. Komplement pravila 60 je pravilo 195.



Slika 5: Pravilo 195 iz središnje početne ćelije, Izvor: vlastita slika

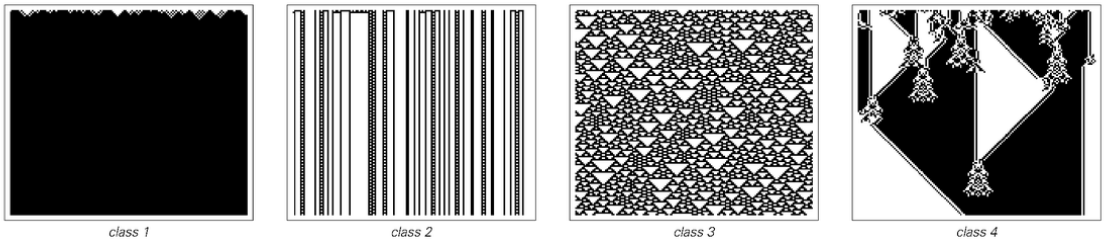
Konačno, postoje i **zrcalni komplementi** kao rezultat primjene obje transformacije, a 16 pravila je identično svojim zrcalnim komplementima. Zrcalni komplement pravila 60 je pravilo 153.



Slika 6: Pravilo 153 iz središnje početne ćelije, Izvor: vlastita slika

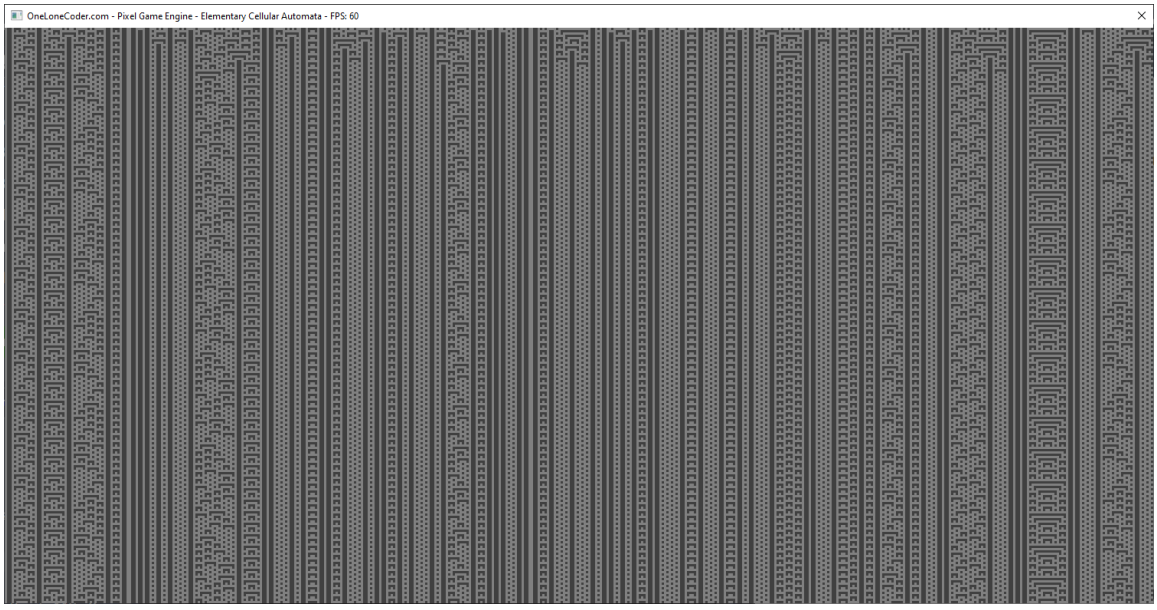
3.3. Klase elementarnih staničnih automata

Iako svaki automat ima različit skup ishoda s različitim svojstvima stvarajući bar donekle drugačije uzorke, većina ih zapravo prilično jasno spada u četiri temeljno različite **kategorije** (**klase**) s obzirom na karakteristike uzorka (ponajviše složenost). Podjelu je uočio i definirao Stephen Wolfram 1983. godine, na samom početku istraživanja elementarnih staničnih automata. [3]



Slika 7: Primjeri automata svake klase, Izvor: [3], stranica 231

Zanimljivost je što čak i automati nižih klasa mogu stvoriti složene uzorke koje ćemo proučiti u programu korištenjem raznih opcija za početno stanje nulte generacije koje su opisane u poglavlju 3.3. ili prelijevanjem jednog pravila u drugo što je opisano u poglavlju 3.3.5.



Slika 8: Pravilo 73 (klasa II) iz nasumičnog početnog stanja, Izvor: vlastita slika

3.3.1. Klasa I

Automati prve i najjednostavnije klase iz gotovo svih početnih uvjeta degeneriraju u **jednolično** i **stabilno** krajnje stanje nakon svega nekoliko iteracija zbog čega nisu posebno zanimljivi u istraživanju. Neka od pravila ove klase su pravila 0, 32, 160 i 232.

3.3.2. Klasa II

Automati druge klase stvaraju **periodične** ili **kvazi-periodične** uzorke. Iako nisu homogeni kao kod prve klase, ne pokazuju nikakve interakcije, složene strukture ili složeno ponašanje. Mogu proizvesti zanimljivije i složenije uzorke nego automati prve klase, no relativno su jednostavni i predvidljivi. Kod ovih automata nasumično početno stanje igra veliku ulogu u rezultatu u usporedbi s jednom živom ćelijom. Primjeri su pravila 4, 108, 218 i 250.

3.3.3. Klasa III

Za razliku od automata prve dvije klase koji brzo degeneriraju u homogeno ili ponavljajuće stanje, automate treće klase karakterizira **nepredvidljivo**, **nasumično** ponašanje. Oni se nikada ne svode na stabilan uzorak već zauvijek održavaju visoku razinu aktivnosti. Stvaraju zamršene uzorke koji se obično sastoje od donekle pravilnih objekata manjih razmjera (često od trokuta). [3]

Zbog velike razine kaotičnosti, automati ove klase imaju potencijalnu primjenu u kriptografiji, pa se na primjer pravilo 30 koristi kao generator pseudonasumičnih brojeva u programskom jeziku *Wolfram*. [7]

3.3.4. Klasa IV

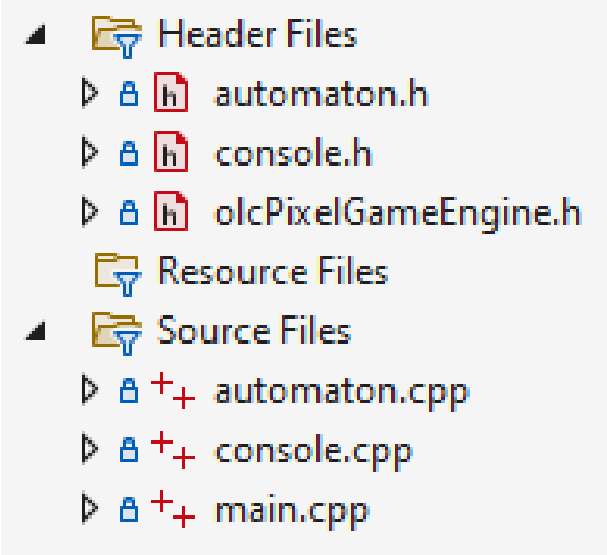
Četvrta klasa je najsloženija i najzanimljivija. Automati ove klase **organiziraju** se iz kaotičnosti nasumičnog početnog stanja stvarajući jednostavne lokalizirane strukture koje imaju jedinstvenu sposobnost kretanja i **složene međusobne interakcije**. [3] Zbog tih interakcija sveukupna je razina aktivnosti ove klase automata veća nego kod druge klase, ali nešto manja nego kod treće klase, dok je razina složenosti veća od obje.

Štoviše, **pravilo 110** je dokazano **univerzalan Turingov stroj**, odnosno sposobno je za izvođenje bilo kojeg programa (engl. *universal computation*). [8] Ostala pravila četvrte klase također bi mogla imati ista svojstva, no takvo što još nije dokazano.

3.4. Programsko rješenje

3.4.1. Struktura i logika programa

Program je podijeljen na datoteku *main.cpp* u kojoj se nalazi jezgrena klasa, datoteke zaglavlja *console.h* i *automaton.h* u kojima su deklarirane istoimene klase i konačno datoteke *console.cpp* i *automaton.cpp* u kojima se nalazi njihova implementacija. Ukupno se sastoji od **tri klase** opisane u sljedećim potpoglavljima. Grafička biblioteka nalazi se u datoteci *olcPixelGameEngine.h*.



Slika 9: *Solution explorer*, Izvor: vlastita slika

3.4.1.1. Klasa *Simulator*

Simulator je **nadležna** klasa programa i predstavlja jezgru (engl. *game engine*) koja pokreće grafiku. Nasljeđuje klasu *PixelGameEngine* iz istoimene biblioteke i nadređuje (engl. *override*) osnovne metode jezgre potrebne za izvođenje programa. U konstruktoru postavljamo ime programa ispisano na naslovnoj traci prozora:

```
Simulator()
{
    sAppName = "Elementary␣Cellular␣Automata";
}
```

Za korištenje jezgre obavezno je nadrediti metode *OnUserCreate* i *OnUserUpdate*. *OnUserCreate* izvodi se jedanput prilikom pokretanja jezgre, a služi za **inicijalizaciju** varijabli i postavljanje početnog stanja jezgre, dok *OnUserUpdate* služi kao **glavna petlja** i izvodi se pri svakoj sličici (engl. *frame*). U njoj se odvija logika programa, hvatanje korisničkih unosa i slično:

```
bool OnUserCreate() override
{
    Clear(DEAD_COLOR);
    automaton = new Automaton(this);
    console = new Console(this, automaton);
    runAutomaton = false;
    ConsoleCaptureStdOut(true);
    srand(unsigned int(time(NULL)));
    return true;
}

bool OnUserUpdate(float fElapsedTime) override
{
    if (runAutomaton)
        automaton->Run();
    CaptureShortcuts();
    return true;
}
```

Neobavezna metoda *OnConsoleCommand* poziva se pri svakom **unosu teksta** u konzolu, odnosno nakon pritiska tipke *Enter*. Služi za obavljanje logike nad unosom koji se nalazi u argumentu *text*. U našem slučaju ovdje se poziva metoda za parsiranje unosa iz klase *Console*.

```
bool OnConsoleCommand(const std::string& text) override
{
    console->ParseInput(text);
    return true;
}
```

Zadnja od osnovnih jezgrenih metoda, neobavezna *OnUserDestroy* poziva se jedanput pri **gašenju** jezgre, odnosno pri izlasku iz programa i služi za oslobađanje rezerviranih resursa, u ovom slučaju objekata automata i konzole.

```
bool OnUserDestroy() override
{
    delete automaton, console;
    return true;
}
```

Metoda *CaptureShortcuts* hvata korisničke unose pomoću jezgrene metode *GetKey* i izvodi akcije za pojedinu pritisnutu tipku. To su **prečice** za pokretanje, pauziranje i resetiranje automata i otvaranje ili zatvaranje konzole:

```
void CaptureShortcuts()
{
    if (GetKey(olc::Key::TAB).bPressed)
        ConsoleShow(olc::Key::TAB, true);

    if (GetKey(olc::Key::CTRL).bPressed)
        runAutomaton = !runAutomaton;

    if (GetKey(olc::Key::R).bPressed && !IsConsoleShowing())
        automaton->Reset();
}
```

Rezolucija prozora i veličina piksela mogu se postaviti **argumentima** naredbenog retka, a prosljeđuju se prilikom inicijalizacije jezgre u funkciji *main* pozivom metode *Construct*, nakon čega se jezgra pokreće pozivom metode *Start*.

Provjera broja argumenata i postavljanje zadanih vrijednosti obavljaju se u funkcijama *CheckArguments* i *GetArguments*, a validaciju vrijednosti obavlja jezgra na način da metoda *Start* vrati *false* u slučaju da je prozor neuspješno stvoren (na primjer ako proslijedimo negativne ili vrijednosti krivog tipa):

```
void CheckArguments(int argc)
{
    if (argc != 4 && argc != 1)
    {
        std::cout << "Provide␣0␣or␣3␣arguments!\n";
        exit(0);
    }
}

auto GetArguments(int argc, char* argv[])
{
    std::array<int, 3> args = { 500, 250, 3 };
    if (argc == 4)
        args = { atoi(argv[1]), atoi(argv[2]), atoi(argv[3]) };
    return args;
}

int main(int argc, char* argv[])
{
    CheckArguments(argc);
    auto args = GetArguments(argc, argv);
    Simulator sim;

    if (sim.Construct(args[0], args[1], args[2], args[2], false, true))
        sim.Start();
    else
        std::cout << "Invalid␣argument(s)!\n";

    return 0;
}
```

3.4.1.2. Klasa *Automaton*

Klasa *Automaton* od najbitnijih atributa sadrži **logiku evolucije** i **crtanja** staničnog automata, trenutnu i sljedeću generaciju u obliku vektora *boolean* vrijednosti širine jednake početnoj širini prozora, broj pravila i početno stanje automata.

Javna metoda *Run* poziva se u *OnUserUpdate*, a u njoj se poziva nekoliko privatnih metoda koje obavljaju crtanje, evoluciju i zamjenu generacija staničnog automata što predstavlja srž funkcionalnosti programa. Svrha *if else* bloka je omogućavanje glatkog listanja (engl. *scroll*) generacija nakon što automat dostigne dno prozora:

```
void Automaton::Run()
{
    GenerateNextGeneration();
    currentGeneration = nextGeneration;

    if (int(row) < pge->ScreenHeight() - 1)
        row++;
    else
        // scroll by drawing screen one pixel higher and new row at the bottom
        pge->DrawSprite(olc::vi2d(0, -1), pge->GetDrawTarget());

    DrawCurrentGeneration();
}
```

Metoda *DrawCurrentGeneration* pomoću jezgrene metode *Draw* piksele ispunjava različitim bojama ovisno o njihovom stanju, čije konstante su definirane u *automaton.h*:

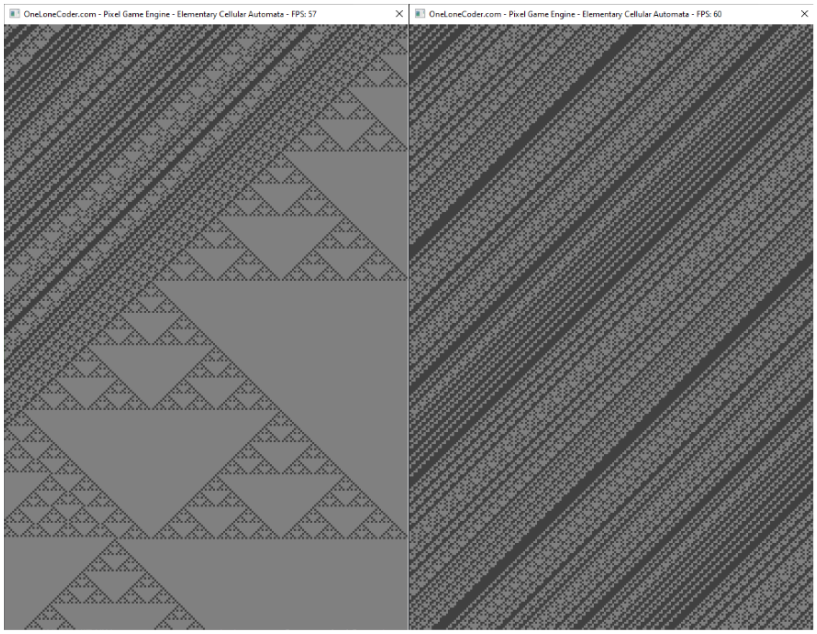
```
void Automaton::DrawCurrentGeneration()
{
    for (unsigned int i = 0; i < width; ++i)
        currentGeneration[i] ? pge->Draw(i, row, LIVE_COLOR)
                             : pge->Draw(i, row, DEAD_COLOR);
}
```

Činjenica da se cijela evolucija odvija u svega nekoliko linija koda naglašava **jednostavnost** elementarnih staničnih automata. Provodi ju metoda *GenerateNextGeneration* i u sljedeću generaciju zapisuje rezultate.

```
void Automaton::GenerateNextGeneration()
{
    for (unsigned int i = 0; i < width; ++i)
    {
        bool left, right, center;

        if (wrap)
        {
            left = currentGeneration[(i + (width - 1)) % width];
            right = currentGeneration[(i + 1) % width];
            center = currentGeneration[i];
        }
        else
        {
            left = !i ? false : currentGeneration[i - 1];
            right = i == width - 1 ? false : currentGeneration[i + 1];
            center = currentGeneration[i];
        }
        nextGeneration[i] = GetNextState(left, center, right);
    }
}
```

Ovdje je omogućen i izbor **omatanja** (engl. *wrapping*) automata oko rubova simulacije. Ako je omatanje uključeno, kod evolucije rubnih ćelija u obzir se uzimaju i "susjedne" ćelije nasuprotnog ruba, a u suprotnom se rubovi smatraju mrtvim ćelijama.



Slika 10: Pravilo 154, lijevo: bez omatanja, desno: s omatanjem, Izvor: vlastita slika

Metoda *GetNextState* kao argumente prima stanja tri ćelija koje se uzimaju u obzir prilikom izračuna evolucije, a vraća **rezultirajuće stanje** ćelije na indeksu *i* pomoću bitovne logike koja je **univerzalna** za sva pravila:

```
bool Automaton::GetNextState(bool left, bool center, bool right)
{
    unsigned int ruleIndex = left * 4 + center * 2 + right;
    return (rule >> ruleIndex) & 1;
}
```

Uzmimo za primjer **pravilo 30** i stanje relevantnih ćelija **110** (točan rezultat prema slici 2 je 0). Postupak računanja rezultata je sljedeći:

1. *ruleIndex* sadrži binarno stanje triju ćelija u obliku dekadskog broja koji predstavlja **indeks** traženog ishoda u skupu pravila iz slike 2 gledajući s desna prema lijevo,

   ```
   ruleIndex = 1 * 4 + 1 * 2 + 0; // -> 4 + 2 + 0 = 6;
   ```

2. bitovni **pomak** pravila (30, 00011110) udesno za 6 bitova pomiče traženi ishod na poziciju najmanje značajnog bita,

   ```
   00011110 >> 6; // 00011110    00000000
                  //   |_________________^
   ```

3. bitovni *i* najmanje značajnog bita vraća njegovu **vrijednost** koja je traženi ishod.

   ```
   00000000 & 00000001; // 0
   ```

Ostale metode klase *Automaton* zbog jednostavnosti ne zahtijevaju pobliže objašnjenje, a obavljaju postavljanje pravila, početnog stanja i resetiranje automata na početno stanje.

3.4.1.3. Klasa *Console*

Klasa *Console* sadrži **logiku naredbenog retka** (konzole) programa i mape s parovima naredbi i njihovih funkcija. Obavlja **parsiranje** i **prepoznavanje** korisničkog unosa, poziva odgovarajuće metode koje mijenjaju stanje automata i ispisuje informacije presretanjem teksta unesenog u *std::cout* što omogućuje grafička biblioteka. Sljedeći blok koda radi jasnoće sadrži definicije tipova korištene u ostatku klase i primjer mapiranja *lambda* funkcija na neke od naredbi.

```
// definicije tipova
using CStringRef = const std::string&;
using CmdNoArgMap = std::unordered_map<std::string, std::function<void(void)>>;
using CmdArgMap = std::unordered_map<std::string, std::function<void(CStringRef)>>;
using CmdNoArgMapIt = CmdNoArgMap::const_iterator;
using CmdArgMapIt = CmdArgMap::const_iterator;

...

// deklaracija mapa naredbi
private:
    CmdNoArgMap commandsNoArg;
    CmdArgMap commandsWithArg;

...

// primjer inicijalizacije mapa
commandsNoArg["clear"] = [&]() { this->pge->ConsoleClear(); };
commandsWithArg["setstate"] = [&](CStringRef arg) { SetAutomatonRule(arg); };
```

Javna metoda *ParseInput* poziva se u *OnUserUpdate*, a poziva metode za **tokenizaciju** unosa u konzolu i **identifikaciju** dobivenih *tokena* (riječi):

```
void Console::ParseInput(CStringRef text)
{
    auto tokens = Tokenize(text);
    if (tokens.empty())
        return;

    tokens.size() > 1 ? IdentifyCommand(tokens[0], tokens[1]) :
                        IdentifyCommand(tokens[0], "␣");
}
```

Metoda *Tokenize* pomoću *std::istringstream* obrezuje i razbija unos te vraća **vektor tokena**:

```
static auto Tokenize(CStringRef text)
{
    std::vector<std::string> tokens;
    std::istringstream iss(text);
    std::copy(std::istream_iterator<std::string>(iss),
              std::istream_iterator<std::string>(),
              std::back_inserter(tokens));
    return tokens;
}
```

Prvi token vektora smatra se naredbom, a drugi argumentom. Daljni tokeni se ignoriraju jer ne postoje naredbe s dva ili više argumenata. Naredba i argument prosljeđuju se metodi *IdentifyCommand* koja **pretražuje mape** s naredbama i poziva odgovarajuće funkcije:

```
void Console::IdentifyCommand(CStringRef command, CStringRef argument)
{
    CmdNoArgMapIt noArgIt = commandsNoArg.find(command);
    CmdArgMapIt argIt = commandsWithArg.find(command);

    if (noArgIt != commandsNoArg.end())
    {
        noArgIt->second();
        return;
    }
    if (argIt != commandsWithArg.end())
    {
        argIt->second(argument);
        return;
    }
    std::cout << "Invalid␣command:␣'" << command << "'.␣Type␣'help'␣for␣a␣list␣of␣
        commands.\n\n";
}
```

Provjera argumenata za naredbe kojima je to potrebno obavlja se u metodama *SetAutomatonState* i *SetAutomatonRule*:

```
void Console::SetAutomatonState(CStringRef state)
{
    if (state == "center")
        automaton->SetState(States::CENTER);
    else if (state == "left")
        automaton->SetState(States::LEFT);
    else if (state == "right")
        automaton->SetState(States::RIGHT);
    else if (state == "random")
        automaton->SetState(States::RANDOM);
    else
    {
        std::cout << "Invalid␣state:␣'" + state + "'.␣Valid␣states:␣left,␣right,␣
            center,␣random.\n\n";
        return;
    }
    std::cout << "State␣set.\n\n";
}

void Console::SetAutomatonRule(CStringRef argument)
{
    for (const char& c : argument)
    {
        if (!isdigit(c))
        {
            "Invalid␣rule:␣'" + argument + "'.␣Value␣must␣be␣in␣range␣[0,255].\n\n";
            return;
        }
    }
    int rule = std::stoi(argument);
    if (rule < 0 || rule > 255)
    {
        "Invalid␣rule:␣'" + argument + "'.␣Value␣must␣be␣in␣range␣[0,255].\n\n";
        return;
    }
    std::cout << "Rule␣set␣to␣" << rule << ".\n\n";
    automaton->SetRule(rule);
}
```

3.4.2. Korištenje programa

Prilikom pokretanja programa mogu se postaviti **rezolucija** prozora i **veličina** piksela pomoću tri argumenta naredbenog retka, a zadana rezolucija je 500×250 s pikselima širine i visine 3 piksela ekrana. Prava veličina zadanog prozora je dakle 1500×750 piksela i primjenjuje se kada nisu dani argumenti.

```
// pokretanje programa u zadanoj rezoluciji
>ElementaryCellularAutomata.exe

// pokretanje programa u rezoluciji 300x150x5
>ElementaryCellularAutomata.exe 300 150 5
```

Za kontrolu programa služi **konzola** koja se otvara i zatvara tipom *TAB* i nekoliko **prečica** koje rade bez njenog otvaranja. Postoje sljedeće konzolne naredbe:

- *help* - prikazuje popis naredbi i prečica
- *setrule* - postavlja pravilo automata, prima vrijednosti od 0 do 255
- *getrule* - ispisuje trenutno pravilo
- *setstate* - postavlja početno stanje automata, prima vrijednosti *left* (najlijevija živa stanica), *right* (najdesnija živa stanica), *center* (središnja živa stanica) ili *random* (nasumično)
- *reset* - briše sliku i vraća automat u zadnje postavljeno početno stanje
- *clear* - briše povijest konzole
- *wrap* - uključuje i isključuje omatanje automata oko rubova simulacije

Također postoje tri prečice:

- *TAB* - otvara/zatvara konzolu
- *CTRL* - pokreće/pauzira automat
- *R* - prečica na naredbu *reset*

Slika 11: Ispis naredbe *help*, Izvor: vlastita slika

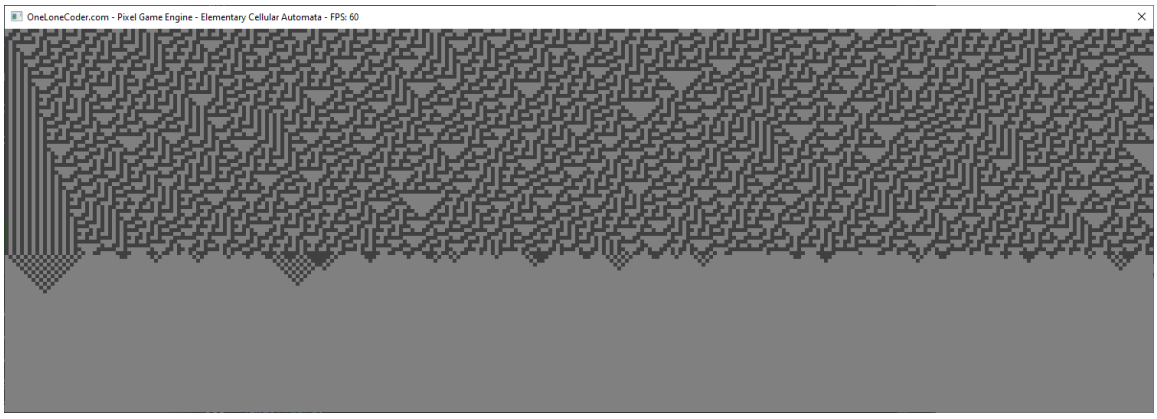
3.5. Rezultati

U ovom poglavlju nalaze se **primjeri** pravila svake klase, komplementarnih i zrcalnih pravila, učinka početnog stanja na uzorke i ostalih pronađenih zanimljivih ponašanja, kao i prikaz nekoliko različitih postavki rezolucije i veličine piksela.

3.5.1. Pravilo 160 - Klasa I

Pravilo 160 iz jednostavnih početnih stanja (jedne žive ćelije) ne proizvodi nikakvu sliku, a razna druga stanja nakon malog broja evolucija rezultiraju mrtvim automatom, odnosno ne ostane niti jedna živa ćelija.

Slika 12: Pravilo 160 iz nasumičnog početnog stanja, Izvor: vlastita slika

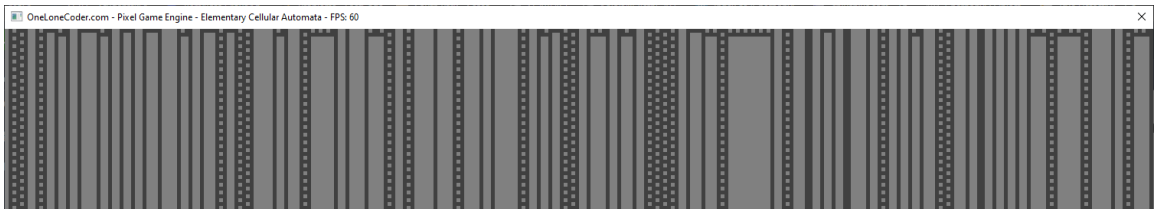


Slika 13: Preljev pravila 30 iz nasumičnog početnog stanja u pravilo 160, Izvor: vlastita slika

3.5.2. Pravilo 108 - Klasa II

Na pravilo 108 uvelike utječe početno stanje. Iz jednostavnih početnih stanja stvara samo liniju, dok iz nasumičnog početnog stanja dobivamo nešto zanimljiviji stabilan uzorak što je karakteristično za ovu klasu automata.

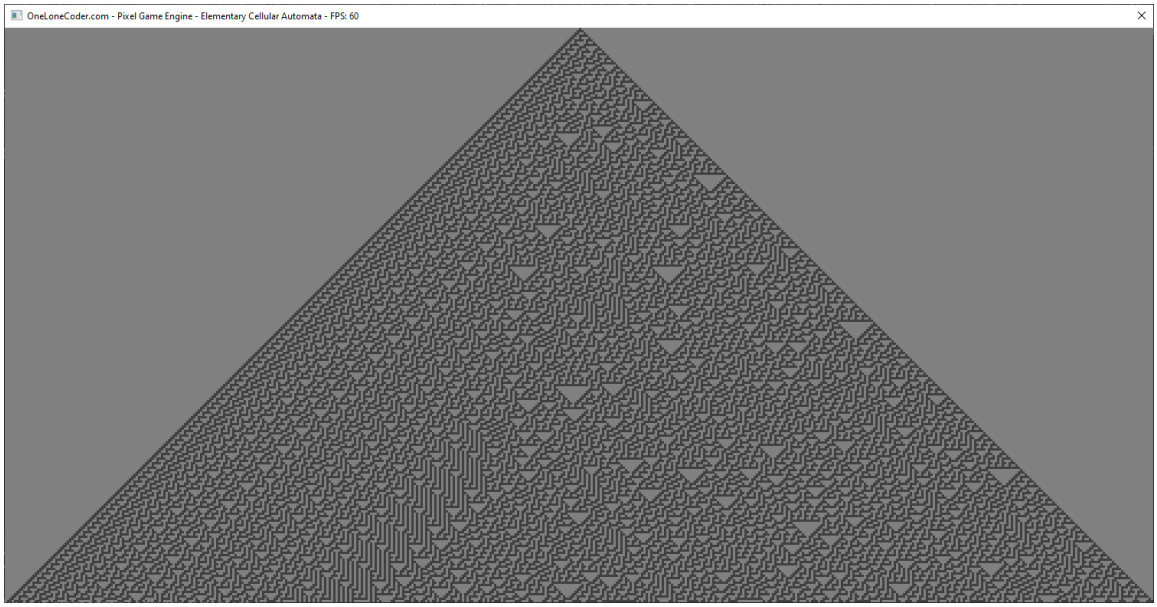
Slika 14: Pravilo 108 iz središnje početne ćelije, Izvor: vlastita slika



Slika 15: Pravilo 108 iz nasumičnog početnog stanja, Izvor: vlastita slika

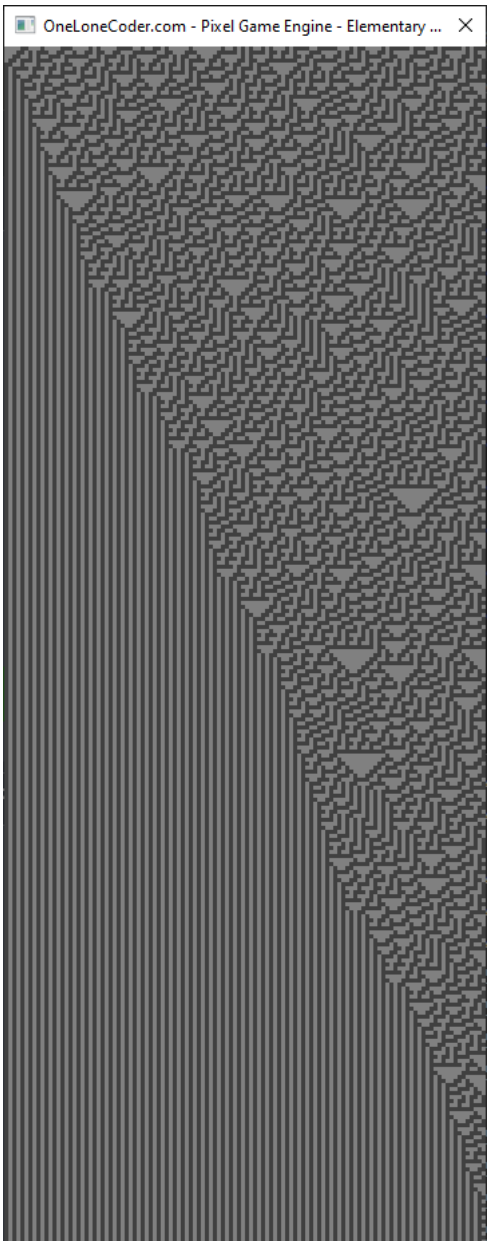
3.5.3. Pravilo 30 - Klasa III

Pravilo 30 dobar je primjer kaotičnog automata i utjecaja omatanja na rezultat. S jednostavnim početnim stanjem vidljiva je podjela na stabilan uzorak lijeve polovice i kaotičnost desne polovice automata, a vide se i trokutaste strukture karakteristične za mnoge složenije elementarne stanične automate.



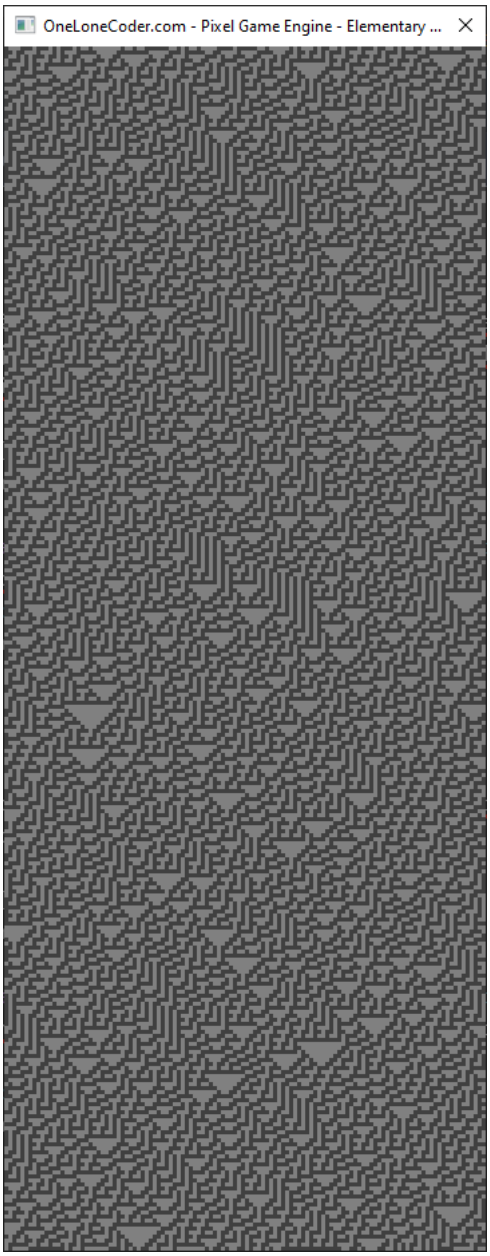
Slika 16: Pravilo 30 iz središnje početne ćelije, Izvor: vlastita slika

Zanimljivo je što bez uključenog omatanja uzorak paralelnih linija koji se pojavi kada živa ćelija dotakne lijevi rub simulacije eventualno svede automat u stabilno stanje bez obzira na početno stanje.



Slika 17: Pravilo 30 iz nasumičnog početnog stanja bez omatanja, Izvor: vlastita slika

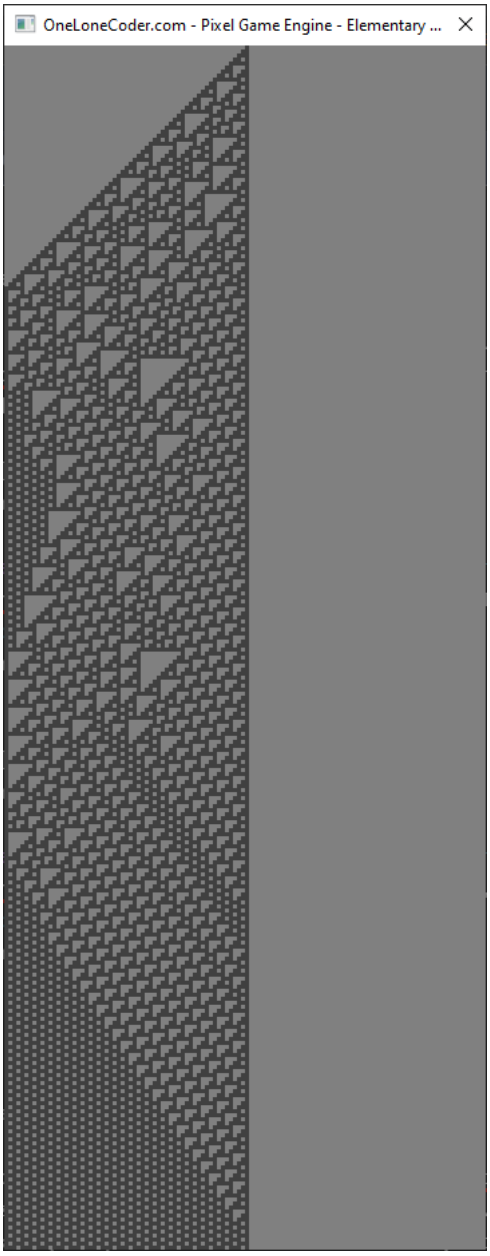
Ta pojava može se spriječiti uključivanjem omatanja naredbom *wrap*, čime dobivamo beskonačan nasumičan uzorak.



Slika 18: Pravilo 30 iz nasumičnog početnog stanja s omatanjem, Izvor: vlastita slika

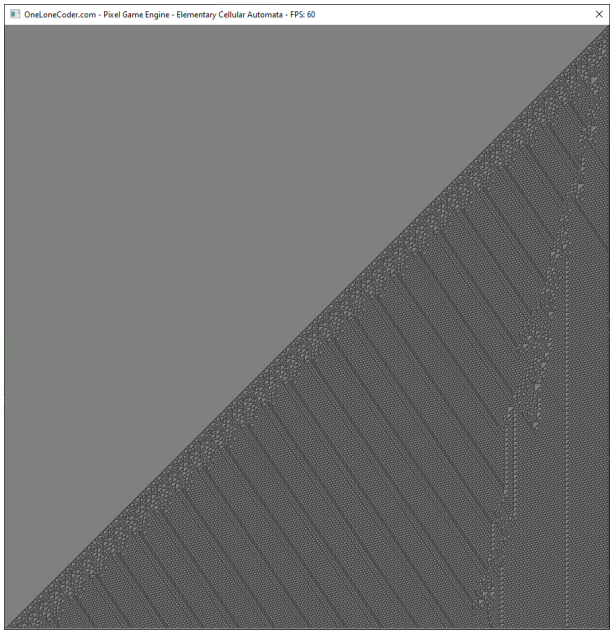
3.5.4. Pravilo 110 - Klasa IV

Kao što je prethodno opisano u poglavlju 3.1.4., pravilo 110 stvara strukture koje imaju sposobnost interakcije. Iz jednostavnih početnih stanja evoluira isključivo ulijevo. Zanimljivost je što bez obzira na početno stanje i omatanje nakon određenog broja generacija završava u stabilnom stanju nalik na automat klase II. Uzrok tome je međusobno uništavanje lokalnih struktura.

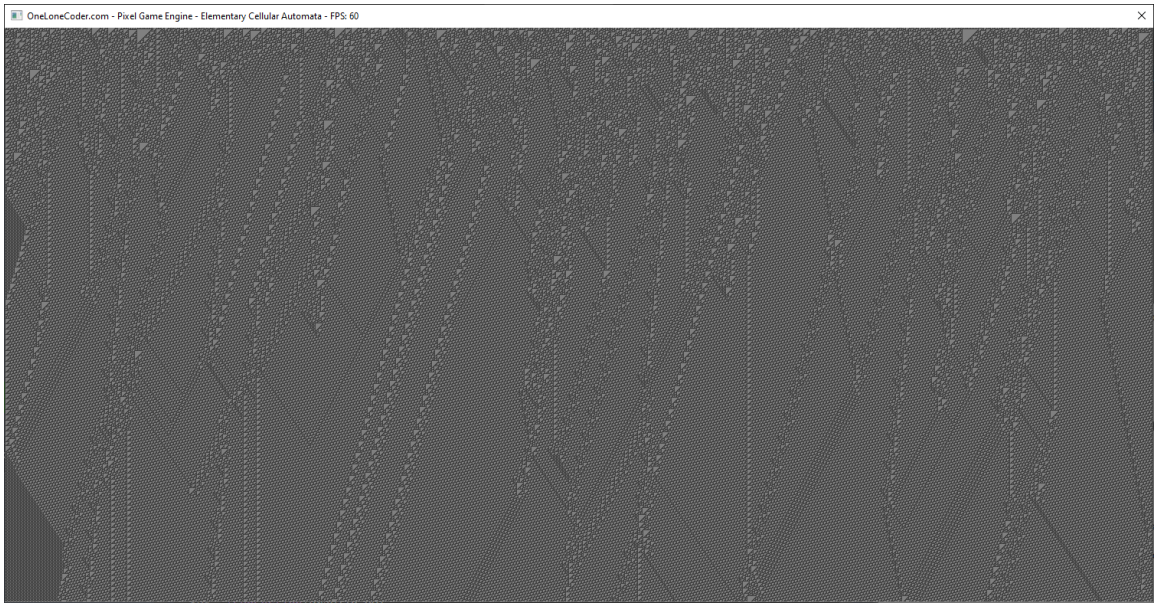


Slika 19: Pravilo 110 iz središnje početne ćelije, Izvor: vlastita slika

Za bolji prikaz ponašanja pravila 110, korisno je postaviti veću rezoluciju s manjom veličinom piksela pri pokretanju programa. S većom rezolucijom jasno su vidljivi sudari karakterističnih jedrilica (engl. *glider*) koje putuju preko pozadinske mreže (engl. *ether*). [8]



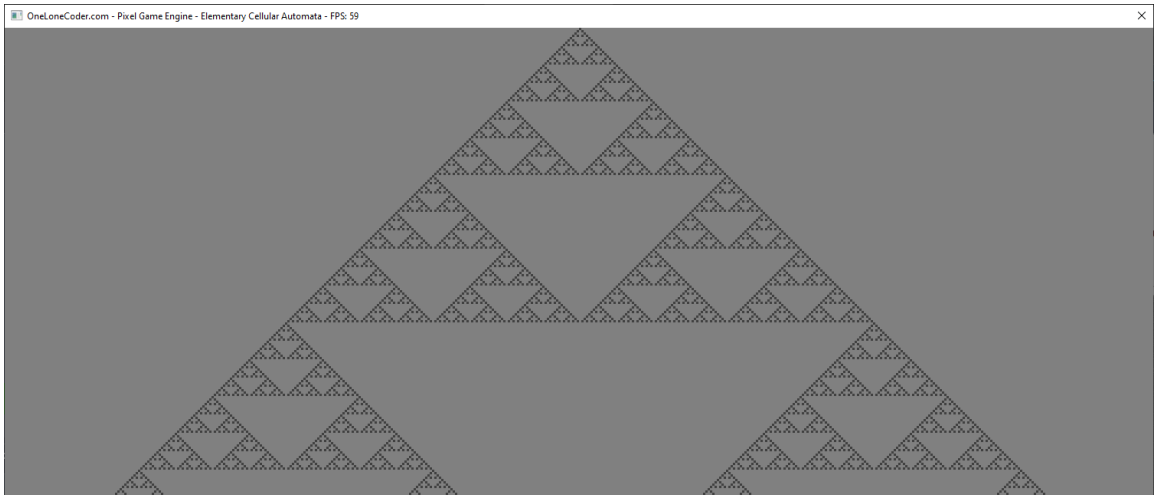
Slika 20: Pravilo 110 iz desne početne ćelije u rezoluciji 900×900×1, Izvor: vlastita slika



Slika 21: Pravilo 110 iz nasumičnog početnog stanja u rezoluciji 1500×750×1, Izvor: vlastita slika

3.5.5. Pravilo 90 - Klasa II/III

Jedno od pravila za koje možemo reći da spada u različite klase ovisno o početnom stanju je pravilo 90. Iz živih ćelija proizvodi stabilni fraktal trokut Sierpińskog, [9] a posebna karakteristika pravila 90 je **aditivnost** [10] - proizvedeni fraktali se preklapaju umjesto da se međusobno unište kao uzorci većine drugih pravila. Aditivnost se jasno vidi kada je u početnom stanju niska gustoća živih stanica, takav slučaj je detaljnije opisan u sljedećem poglavlju.



Slika 22: Pravilo 90 iz središnje početne ćelije, Izvor: vlastita slika

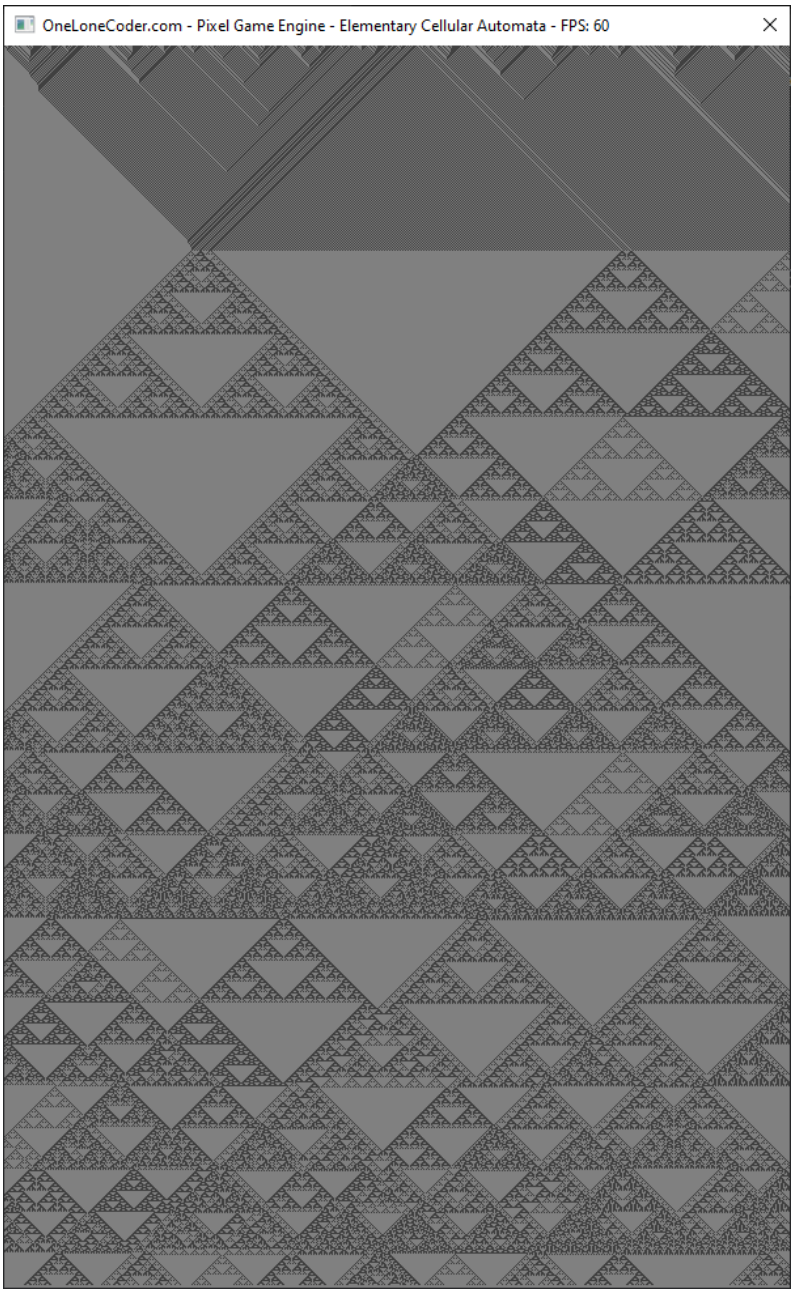
Za razliku od pravila 110 i 30, pravilo 90 i bez omatanja iz nasumičnog početnog stanja zbog aditivnosti nikada ne dostiže stabilno stanje.

Slika 23: Pravilo 90 iz nasumičnog početnog stanja, ovaj uzorak čini mnoštvo preklopljenih trokuta Sierpińskog, Izvor: vlastita slika

3.5.6. Prelijevanje pravila

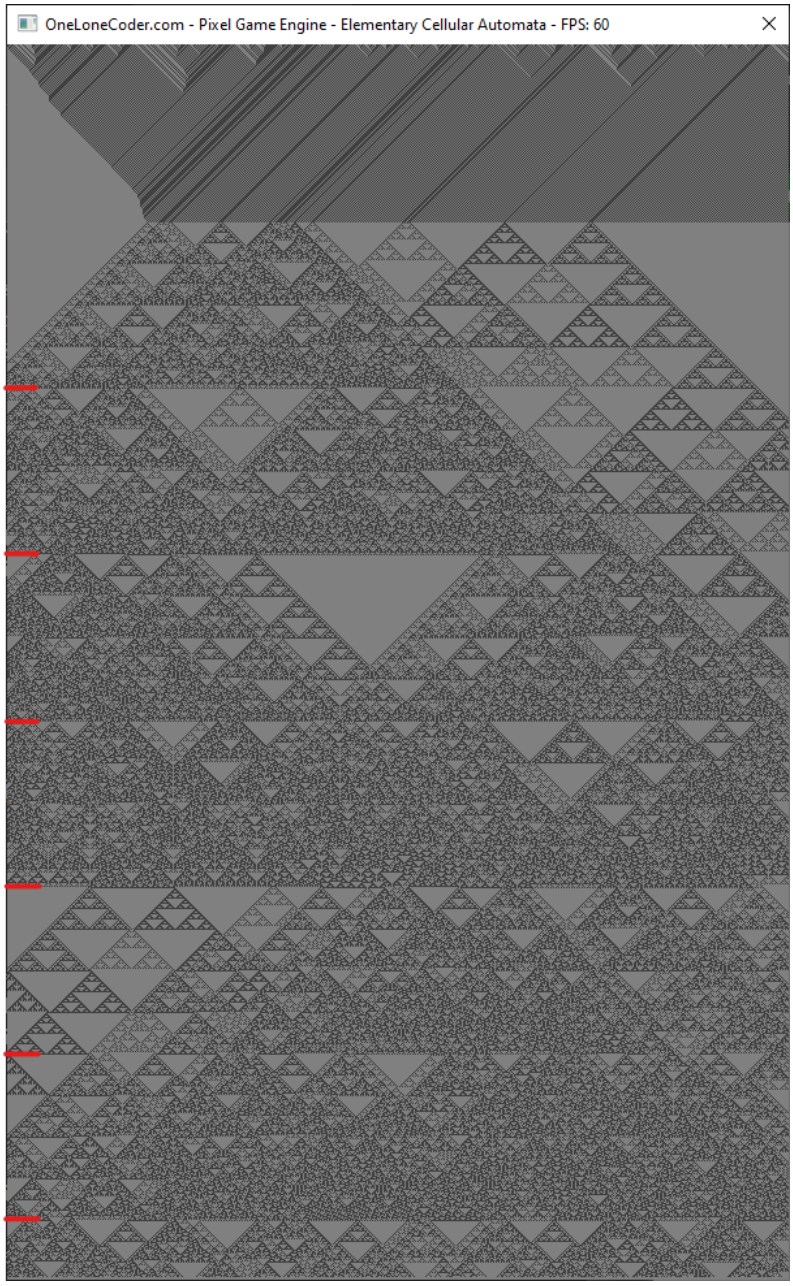
Tijekom razvoja programa pojavila se zanimljiva mogućnost promjene pravila "uživo" tijekom simulacije, tako da zadnja stvorena generacija prethodnog pravila postaje početna generacija sljedećeg, što može stvoriti veoma zanimljive uzorke.

Odličan primjer je prijelaz iz pravila 184 u pravilo 90 kada jasno vidimo njegovo svojstvo aditivnosti, odnosno preklapanje uzoraka kojima su izvorišta linije pravila 184.



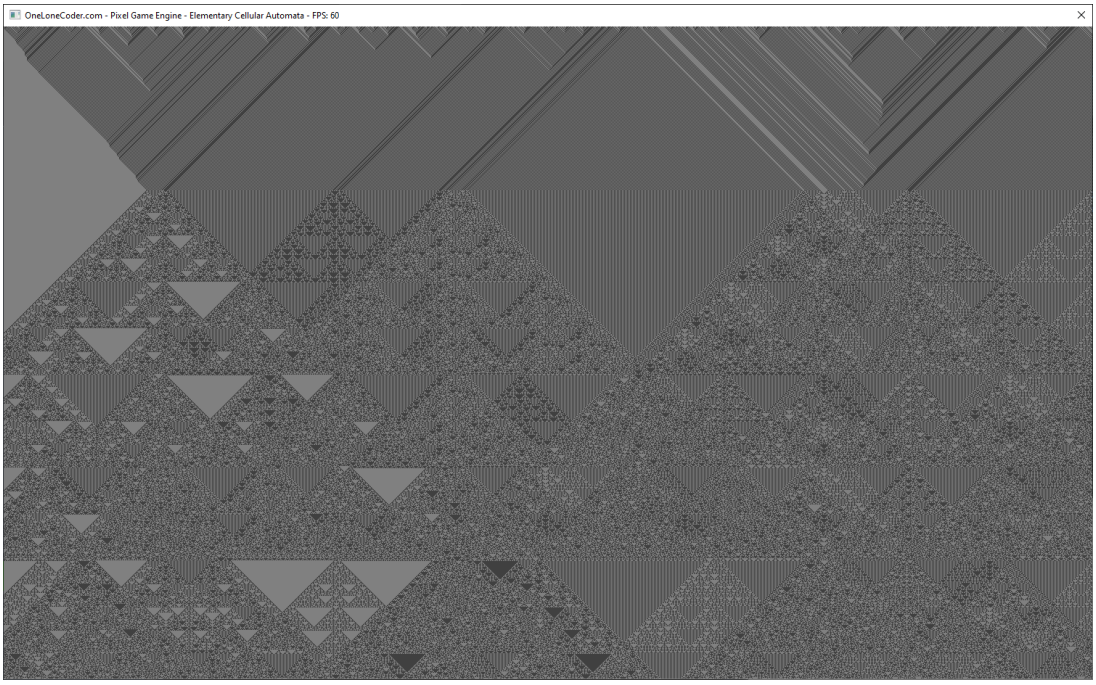
Slika 24: Preljev pravila 184 iz nasumičnog početnog stanja u pravilo 90, Izvor: vlastita slika

Broj linija u trenutku prijelaza određuje ukupnu razinu kaotičnosti pravila 90, a vidljiv je i pravilan interval razine kaotičnosti označen crvenim linijama. Gustoća živih stanica periodično se povećava do najviše razine, zatim pada nazad na nižu.

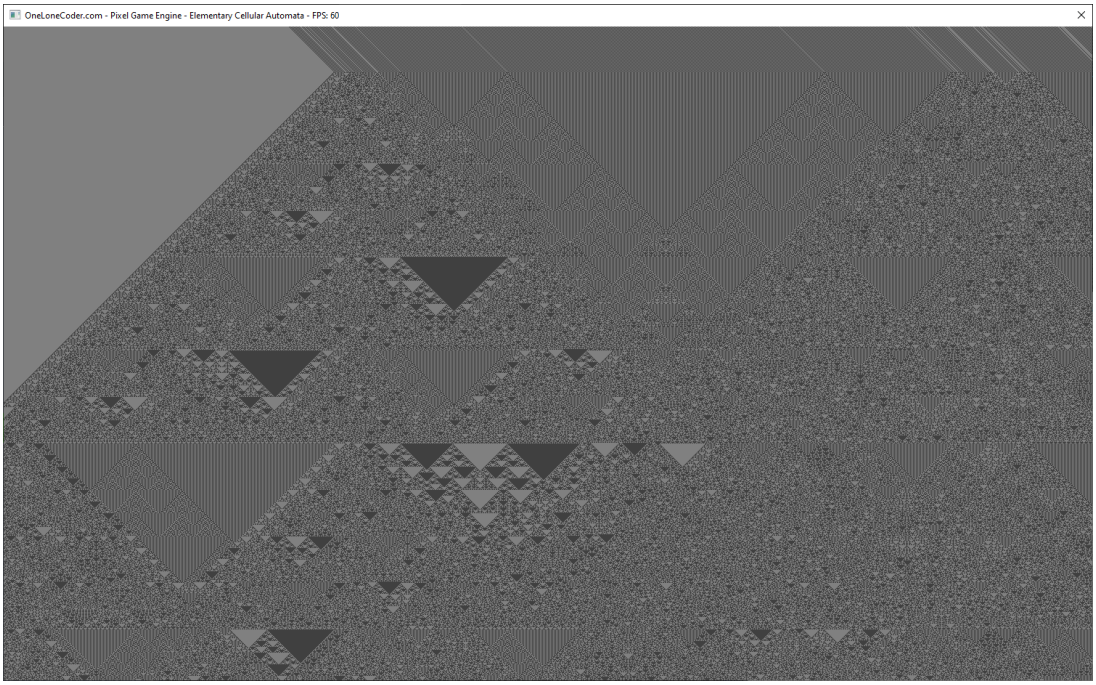


Slika 25: Preljev pravila 184 iz nasumičnog početnog stanja u pravilo 90 s označenim periodima, Izvor: vlastita slika

Još jedno aditivno pravilo s vrlo zanimljivim detaljnim uzorcima je pravilo 150.

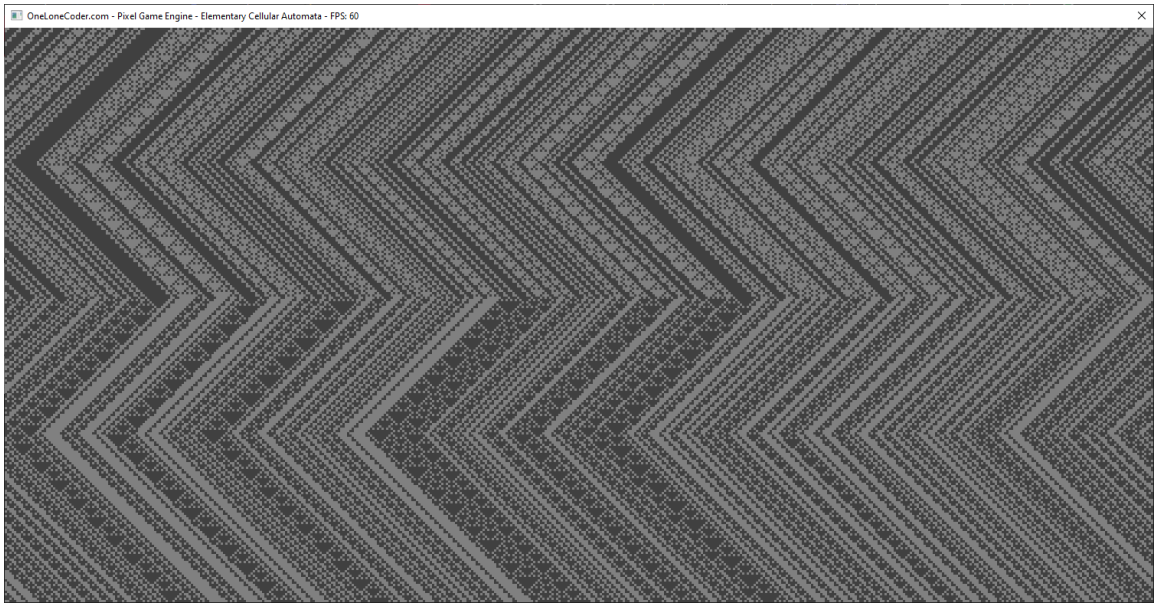


Slika 26: Preljev pravila 184 iz nasumičnog početnog stanja u pravilo 150, Izvor: vlastita slika



Slika 27: Preljev pravila 184 iz nasumičnog početnog stanja u pravilo 150, Izvor: vlastita slika

Kao što je spomenuto u poglavlju 3.2., mnoga pravila imaju komplemente, zrcalna pravila i zrcalne komplemente. Možemo ih istovremeno prikazati pomoću prelijevanja postavljanjem redom primjerice pravila 154, 210, 166 i 180 koja predstavljaju izvorno pravilo, zrcalno pravilo, komplement i zrcalni komplement.



Slika 28: Preljev pravila redom: 154, 210, 166, 180, Izvor: vlastita slika

3.6. Izračun složenosti

Da bi izračunali ukupnu složenost javne metode *Run* koja evoluira automat, moramo najprije izračunati i zbrojiti složenosti svih privatnih metoda koje ona poziva.

```
bool Automaton::GetNextState(bool left, bool center, bool right)
{
    unsigned int ruleIndex = left * 4 + center * 2 + right;  // c1
    return (rule >> ruleIndex) & 1;                          // c2
}
```

$$T_{Max}^{GetNextState()} \leq c_1 + c_2$$
$$T_{Max}^{GetNextState()} = O(1)$$

```
void Automaton::GenerateNextGeneration()
{
    for (unsigned int i = 0; i < width; ++i)                                  // <= xn, c1
    {
        bool left, right, center;                                             // c2

        if (wrap)                                                             // c3
        {
            left = currentGeneration[(i + (width - 1)) % width];              // b1->O(1)
            right = currentGeneration[(i + 1) % width];                       // b2->O(1)
            center = currentGeneration[i];                                    // b3->O(1)
        }
        else
        {
            left = !i ? false : currentGeneration[i - 1];                     // b4->O(1)
            right = i == width - 1 ? false : currentGeneration[i + 1];        // b5->O(1)
            center = currentGeneration[i];                                    // b6->O(1)
        }
        nextGeneration[i] = GetNextState(left, center, right);                // c4
    }
}
```

$$T_{Max}^{GenerateNextGeneration()} \leq n(c_1 + c_2 + c_3 + c_4)$$
$$T_{Max}^{GenerateNextGeneration()} = O(n)$$

```
void Automaton::DrawCurrentGeneration()
{
    for (unsigned int i = 0; i < width; ++i)                    // <= xn, c1
        currentGeneration[i] ? pge->Draw(i, row, LIVE_COLOR) :  // O(1) + O(1) -> c2
                               pge->Draw(i, row, DEAD_COLOR);
}
```

$$T_{Max}^{DrawCurrentGeneration()} \leq n(c_1 + c_2)$$
$$T_{Max}^{DrawCurrentGeneration()} = O(n)$$

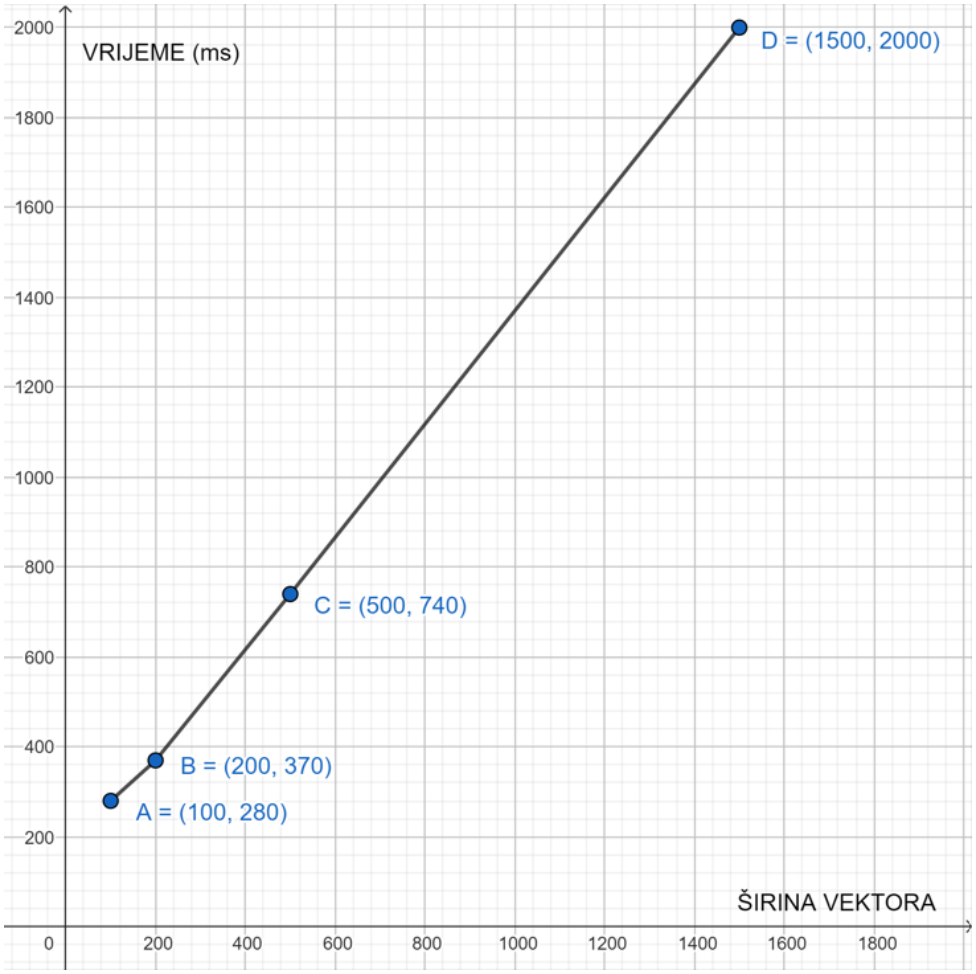
```
void Automaton::Run()
{
    GenerateNextGeneration();                                       // O(n)
    currentGeneration = nextGeneration;                             // c1

    if (int(row) < pge->ScreenHeight() - 1)                         // c2
        row++;                                                      // b1 -> O(1)
    else
        pge->DrawSprite(olc::vi2d(0, -1), pge->GetDrawTarget());    // b2 -> O(1)

    DrawCurrentGeneration();                                        // O(n)
}
```

$$T_{Max}^{Run()} \leq O(n) + c_1 + c_2 + O(n)$$
$$T_{Max}^{Run()} \leq O(n) + O(n)$$
$$T_{Max}^{Run()} = O(n)$$

Kao što je očekivano, brzina algoritma linearno ovisi o širini vektora ćelija odnosno postavljenoj širini simulacije. Vremenska složenost O(n) dokazana je i eksperimentalno, mjerenjem vremena potrebnog za generiranje 700 generacija automata s raznim širinama. Za potrebe testa isključeno je crtanje da bi eliminirali utjecaj grafičkog podsustava na brzinu izvođenja, no čini se kako svejedno postoji određena donja granica na koju ne možemo utjecati.



Slika 29: Graf vremenske složenosti algoritma, Izvor: vlastita slika

4. Zaključak

Elementarni stanični automati savršen su primjer kako i jednostavni sustavi mogu biti veoma zanimljivi i predmet dugogodišnjeg istraživanja. Tijekom razvoja programa naučio sam mnogo o njihovom funkcioniranju i ponašanju, a posebno je impresivno kako se evolucija generacija za svih 256 automata može napisati u jednoj petlji i svega nekoliko linija koda. Nevezano za temu rada, naučio sam dosta i o parsiranju naredbi s čime se nisam prije susreo na ovoj razini.

Ne čudi me što je iz ove teme proizašlo mnogo desetljeća istraživanja jer osim elementarnih staničnih automata postoje i automati s više od dva stanja, automati koji u obzir uzimaju razne brojeve susjednih ćelija, dvodimenzionalni, trodimenzionalni pa čak i četverodimenzionalni automati i tisuće kombinacija spomenutih svojstava. S obzirom koliko su zanimljivi i ovi najjednostavniji sustavi koje sam proučio, očekujem da će oni složeniji biti tema mnogih budućih istraživanja i otkrića.

Najvjerojatnija primjena staničnih automata mogla bi biti u kriptografiji zbog generiranja nasumičnih brojeva, a neke veoma složene vrste pokazale su čak sposobnost detaljne simulacije "organizama" sličnih bakterijama [11]. Jednog dana stanični automati pogonjeni umjetnom inteligencijom mogli bi simulirati i složene organizme.

Rezultat ovog rada je program koji je lak za korištenje, u raznim dimenzijama i rezolucijama učinkovito simulira svih 256 elementarnih staničnih automata i omogućuje detaljno proučavanje njihovog ponašanja i interakcije. Posebno zanimljiva neplanirana mogućnost je promjena automata tijekom simulacije kako je opisano u poglavlju 3.5.6., a sve u svemu bilo je uzbudljivo i poučno ne samo razvijati program već ga i koristiti, pokušavajući složiti najzanimljivije uzorke i pronaći najfascinantnije pojave.

Popis literature

[1] E. W. Weisstein, „Elementary cellular automaton,” 2002. adresa: https://mathworld.wolfram.com/ElementaryCellularAutomaton.html.

[2] E. W. Weisstein. „Wolfram MathWorld.” (2002.), adresa: https://mathworld.wolfram.com.

[3] S. Wolfram i dr., *A new kind of science*. Wolfram media Champaign, IL, 2002., sv. 5. adresa: https://www.wolframscience.com/nks/.

[4] H. Lukšić, *Elementary Cellular Automata*. adresa: https://github.com/HLuksic/ElementaryCellularAutomata.

[5] Javidx9, *olcPixelGameEngine*. adresa: https://github.com/OneLoneCoder/olcPixelGameEngine.

[6] *Rule 60 - Rule properties*. adresa: http://atlas.wolfram.com/01/01/60/01_01_1_60.html#01_01_9_60.

[7] E. W. Weisstein, „Rule 30,” 2002. adresa: https://mathworld.wolfram.com/Rule30.html.

[8] E. W. Weisstein, „Rule 110,” 2002. adresa: https://mathworld.wolfram.com/Rule110.html.

[9] E. W. Weisstein, „Sierpiński Sieve,” 2002. adresa: https://mathworld.wolfram.com/SierpinskiSieve.html.

[10] E. W. Weisstein, „Additive Cellular Automaton,” 2002. adresa: https://mathworld.wolfram.com/AdditiveCellularAutomaton.html.

[11] *Neat AI does Lenia - Conway's game of life arrives in the 21st century*. adresa: https://youtu.be/7-97RhAZhXI.

Popis slika