МИНОБРНАУКИ РОССИИ

Федеральное государственное бюджетное образовательное учреждение высшего образования

«САРАТОВСКИЙ НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ ИМЕНИ Н. Г. ЧЕРНЫШЕВСКОГО»

УТВЕРЖДАЮ
Зав.кафедрой,
к. ф.-м. н., доцент
_____ А. С. Иванов

ОТЧЕТ О ПРАКТИКЕ

студента 4 курса 451 группы факультета КНиИТ
Григорьева Алексея Александровича

вид практики: преддипломная
кафедра: математической кибернетики и компьютерных наук
курс: 4
семестр: 8
продолжительность: 4 нед., с 30.04.2020 г. по 27.05.2020 г.

Руководитель практики от университета,

доцент _____ М. С. Семенов

Руководитель практики от организации (учреждения, предприятия),

доцент _____ М. С. Семенов

Тема практики:«Оптимизации поиска приближенных решений в эволюционирующих клеточных автоматах»

СОДЕРЖАНИЕ

ВВЕДЕНИЕ

Клеточный автомат — дискретная модель, изучаемая теорией автоматов [1]. В общем случае, клеточный автомат состоит из регулярной решетки ячеек, множества состояний, множества входных сигналов и функции перехода. Клеточные автоматы применяются в компьютерном зрении [2], криптографии [3], компьютерной графике [4] и при моделировании различных процессов: физических [5], химических [6], биологических [7]. Одним из наиболее известных является клеточный автомат — игра «Жизнь» [8].

Существует огромное количество конфигураций клеточных автоматов. Меняя число возможных состояний, правила подсчета входных сигналов, начальный сигнал, мы получаем новые множества автоматов, перебор которых невозможен с текущими мощностями. Поэтому, для поиска клеточных автоматов, удовлетворяющих каким-либо условиям, используют алгоритмы нахождения приближенного результата. В недавних работах на схожую тему применялись генетические алгоритмы [9] и нейронные сети [10]. В настоящей работе это осуществляется с использованием генетических алгоритмов.

В основе работы лежит выполнение экспериментов по нахождению приближенного решения клеточными автоматами. Цель каждого эксперимента — стабильное выполнение заданных условий изображениями, полученными после работы клеточных автоматов над изображением из множества начальных состояний. В данной работе начальное изображение задано случайно в каждом пикселе. Для наглядности в качестве цели, которая будет поставлена перед началом экспериментов, будет воспроизведение заданного наперед целевого изображения (паттерна). Благодаря работе с изображениями можно будет не только полагаться на полученные числовые значения, но и сопоставить результат с визуализацией в приложении. После каждого эксперимента должна сохраняться исчерпывающая информация о поведении клеточного автомата. По ней должно быть возможно определить, был ли удачным эксперимент, сколько было затрачено времени, и какая была конфигурация эксперимента.

Эксперименты по оптимизации проводились для двумерных клеточных автоматов первого порядка, с возможными состояниями 0 и 1. Следующее состояние ячейки клеточного автомата определяется 9 соседними ячейками на решетке включая данную или, другими словами, окрестностью Мура порядка 2 [11]. Тип автоматов выбран не случайно: количество возможных правил,

описывающих их, равно 2^{512} [12], что гарантирует невозможность перебора. Более того, не существует и общего аналитического решения для воспроизведения автоматами произвольного изображения, при условии что изначальная двумерная сетка задана случайными значениями. У выбранных в данной работе типов клеточных автоматов также есть преимущества, связанные с возможностью оптимизации алгоритмов с использованием быстрых побитовых операций [13]. Многие из описанных далее результатов возможно перенести на случай клеточных автоматов `N`-го порядка [14], а также для клеточных автоматов с увеличенным количеством состояний или иными правилами переходов.

Целью настоящей работы является создание полноценного приложения для проведения экспериментов по оптимизации поиска приближенного решения на основе паттернов двумерными клеточными автоматами первого порядка. Должно быть создано гибкое приложение с возможностью конфигурации алгоритма. Оптимизация должна быть проведена как за счет скорости исполнения, так и за счет настроек генетического алгоритма. Полученные результаты будут проанализированы в дипломной работе.

1 Окружение для проведения экспериментов

В данном разделе приведено описание проекта и вспомогательных функций для моделирования и визуализации двумерных клеточных автоматов первого порядка.

1.1 Среда разработки

Поставленная задачу выполнена с использованием среды разработки `Unity`. Такой выбор основан удобствами, предоставляемыми средой разработки. В частности, на выбор повлияли следующие её преимущества:

— быстрая и многофункциональная работа с графикой посредством готовых функций и визуального проектирования;
— профессиональные инструменты для отладки и профилирования производительности программы;
— наглядность результата программы.

Программный код написан на языке `C#`. Основная логика экспериментов реализована в сцене MainScene. Весь проект подключен к системе контроля версий Git [15] на базе популярного ресурса для хранения версий Github. Ссылка на «репозиторий»: `https://github.com/Alexflames/cellular-automation`.

1.2 Реализация клеточного автомата

Каждый клеточный автомат представляется ячейками на двумерной сетке. Ради прозрачности процесса и наглядности результатов, эксперимент проводится полностью в реальном времени: в каждую единицу времени все ячейки автоматов на всех сетках переходят в новое состояние, определенное входными сигналами и правилами перехода.

Опишем правила перехода для двумерного клеточного автомата первого порядка. При подсчете входного сигнала в ячейке автомата будут учитываться как и состояние данной ячейки, так и состояния восьми соседних ячеек в окрестности Мура порядка 2, для подсчета берутся значения с последней итерации. Возьмем самый простой случай: автомат может принимать значения 0 и 1. Таким образом, возможны $2^9 = 512$ вариантов входных сигналов. Отсюда и берется упомянутое ранее количество всевозможных правил для данных клеточных автоматов, 2^{512}.

В качестве визуализации клеточных автоматов в `Unity` можно использовать шейдеры. Плюс этого подхода заключается в том, что компьютер не

будет тратить ресурсы оперативной памяти для визуализации автоматов, и вся нагрузка на это отдается видеокарте, находящейся большую часть времени в простое. Шейдеры позволяют создавать крупные визуализации огромной размерности благодаря эффективным вычислениям на видеокарте [16].

1.2.1 Правило перехода

Создадим простейшую реализацию подсчета сигнала для таблицы перехода для каждого клеточного автомата. Для избежания лишних затрат памяти, в основной части программы заранее созданы массивы для временного хранения состояний автомата на следующем шаге. Также это необходимо, чтобы полученный результат не влиял на соседние автоматы в данный момент.

Создадим цикл, который будет выполняться на двумерной сетке клеточных автоматов размерностью `screenSizeInPixels` по X и Y.

```
var nextCAField = nextVirtualScreens[ind];
var screen2DSize = screenSizeInPixels * screenSizeInPixels;
for (short i = 0; i < screenSizeInPixels; i++)
{
   for (short j = 0; j < screenSizeInPixels; j++)
   {
       int signal = 0;
       for (short k = 0; k < 3; k++)
       {
           for (short m = 0; m < 3; m++)
           {
               signal += CAField[
                   (screen2DSize + screenSizeInPixels * i + j
                   + screenSizeInPixels * (k - 1) + (m - 1))
                   % (screen2DSize)] << (k * 3 + m);
           }
       }
       nextCAField[i * screenSizeInPixels + j] = (byte)allRules[ind][signal];
   }
}

for (short i = 0; i < screen2DSize; i++)
{
   CAField[i] = nextCAField[i];
}
```

1.2.2 Оптимизированное вычисление входных сигналов

Заметим, что в вычислении следующего состояния каждого автомата используется 9 обращений к сетке. Более того, при каждом последующем обра-

щении к сетке после первого, повторно считываются 6 состояний. Представим квадрат 3×3, окрестность Мура порядка 2, в виде окна, которое можно сдвигать вправо по сетке. Разобьем окно на 3 горизонтальные линии-буферы и, при переходе в новое состояние (аналогично сдвигу окна вправо), будем считывать лишь 3 новых бита справа. Добавление каждого бита в строку-буфер сопровождается побитовым сдвигом строки на 1 влево, с отсечением крайнего левого бита. Сигналом для текущего автомата будет сумма верхней строки, средней строки с побитовым сдвигом влево на 3 и нижней строки с побитовым сдвигом на 6. Для наглядности процесс изображен на рисунке 1.

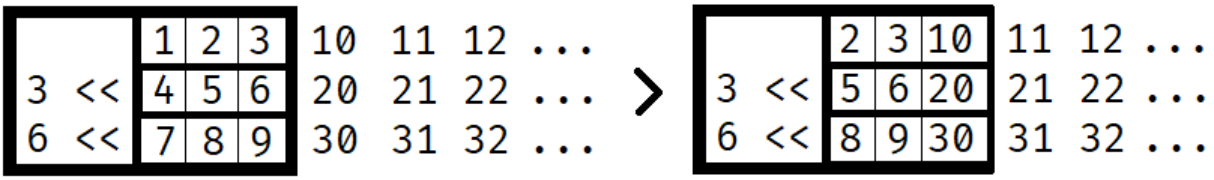

Рисунок 1 – Принцип оптимизированного вычисления входных сигналов с использованием буферов и побитовых сдвигов.

Результат аналогичен тому, что было получено в предыдущем разделе 1.2.1, но будет требоваться примерно в 3 раза меньше операций доступа. Получившийся результат особенно важен для нас, ведь обновление всех автоматов происходит каждый кадр. Полный код новой функции можно посмотреть в приложении А.

1.2.3 Правило перехода на языке шейдеров (HLSL)

Обновление клеточных автоматов возможно полностью перенести в шейдеры. Каждая сетка с клеточным автоматом будет использовать свою текстуру и шейдер.

Для написания правила перехода на языке шейдеров необходимо решить три проблемы.

1. Получение информации о состояниях в предыдущий момент времени.
2. Соотнесение представления в битах (пикселях) с текстурой в трехмерном пространстве.
3. Обновление шейдера только при необходимости.

Возьмем `Custom Render Texture` из среды разработки `Unity`. Данная текстура создана для того, чтобы её можно было обновлять с помощью шейдеров. Первая проблема решается расширением языка HLSL, предоставляе-

мым `Custom Render Texture`. В фрагментном шейдере можно обратиться к переменной `localTexcoord`, в которой хранится состояние текстуры на предыдущий момент отрисовки.

Вторая проблема также решается с использованием глобальных переменных: `_CustomRenderTextureWidth`, предоставляющая ширину объекта в пространстве, и `_CustomRenderTextureHeight`, возвращающая высоту текстуры.

Определим настройки `Custom Render Texture` (рис. 2). Установка параметра `Wrap Mode` в значение `Repeat` задаст периодические граничные условия для сетки. Другими словами, при обращении к пикселям по координатам, превышающим размерность текстуры, будут взяты пиксели с противоположной стороны сетки. Создадим пустой шейдер, код для которого опишем дальше. Настройка `Update Mode` в значение `On Demand` позволяет обновлять состояние текстуры по вызову функции из основной программы, что позволит получить полный контроль над обновлением автоматов.

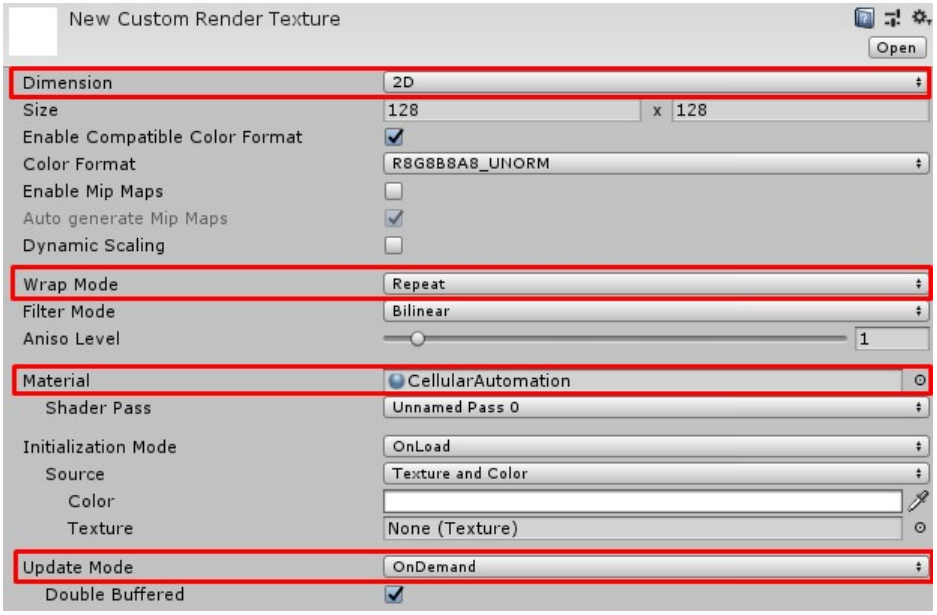

Рисунок 2 – Настройка Custom Render Texture.

Определим uniform-переменную: таблицу переходов размерности 512 (все возможные состояния рассматриваемого клеточного автомата).

```
float _rule[512];
```

Опишем функцию, возвращающую значение пикселя с заданным отступом в сетке относительно текущего клеточного автомата.

```
float4 get(v2f_customrendertexture IN, int x, int y) : COLOR
{
   return tex2D(_SelfTexture2D, IN.localTexcoord.xy +
            fixed2(x / _CustomRenderTextureWidth, y / _CustomRenderTextureHeight));
}
```

Аналогично построению двоичного числа, последовательно, с левого верхнего угла, соберем биты входных сигналов для клеточного автомата в окрестности Мура порядка 2 и получим следующее состояние из таблицы перехода. Данный код необходимо выполнять в фрагментном шейдере [17]

```
float getRule9(v2f_customrendertexture IN) : float
{
   int accumulator = 0;
   for (int i = 2; i >= 0; i--)
   {
       for (int j = 0; j <= 2; j++)
       {
           int roundedAlpha = round(get(IN, i-1, j-1).a);
           accumulator = (accumulator << 1) + roundedAlpha;
       }
   }
   return _rule[accumulator];
}

float4 frag(v2f_customrendertexture IN) : COLOR
{
   return getRule9(IN);
}
```

Осталось объединить шейдер с основной программой. Пусть в массиве `allRules` находятся правила для всех двумерных сеток. Тогда, опуская подробности, инициализацию текстуры можно произвести с помощью следующего кода

```
customRenderTexture.material.SetFloatArray("_rule", allRules[screenInd]);
customRenderTexture.Initialize();
```

Пусть в массиве `custormRenderTextures` содержатся все текстуры (сетки). Тогда, обновить каждый автомат можно, используя следующий фрагмент кода:

```
foreach(var texture in customRenderTextures)
{
   texture.Update();
}
```

Запустим программу и увидим клеточные автоматы в динамике, как изображено на рис. 3.

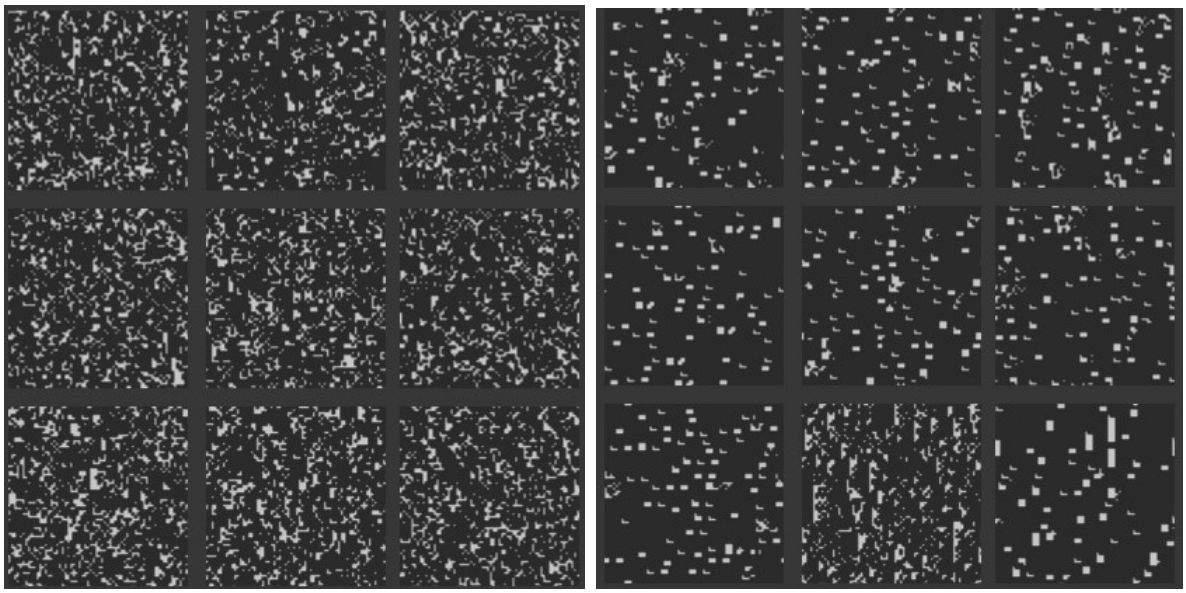

Рисунок 3 – Визуализация клеточных автоматов.

2 Генетический алгоритм для поиска клеточных автоматов

Для получения приближенных результатов, воспользуемся генетическим алгоритмом [18]. **Особи** — сетки с клеточными автоматами, **гены** — правила перехода для клеточных автоматов, **приспособленность** особи — суммарная степень совпадения для каждой окрестности ячейки клеточного автомата некоторому целевому изображению (паттерну), воспроизведение которого и будет являться целью каждого эксперимента.

2.1 Подсчет приспособленности

Для правильной работы генетического алгоритма, нам нужно различать «полезные» особи от «ненужных». Для этого определим приспособленность особи. Приспособленностью особи будем считать процент совпадения фрагментов полученного изображения на сетке с заданным перед началом эксперимента паттерном. Таким образом, мы проверяем окрестность каждого пикселя сетки на совпадение с заданным паттерном.

При подсчете приспособленности учитывается количество допустимых ошибок при сравнении с паттерном. Если число пикселей, не совпадающих с целевым изображением, превышает порог, мгновенно переходим на следующую итерацию. В противном случае, если X — максимальное количество допустимых ошибок, а Y — число несовпадений в паттерне, тогда к ответу прибавим $(1+X-Y)/(X+1)$. Добавляя такой параметр, мы намеренно рискуем получить искаженный результат, не совпадающий полностью с паттерном, но зачастую такая мера необходима для значительного уменьшения времени сходимости к приемлемому результату. Однако выбранная нами формула стимулирует автомат находить результат без ошибок, что частично компенсирует этот недостаток.

Также необходимо учесть, что мы можем искать совпадение не с единственным изображением, а с целым набором, как показано на изображении 4. В таком случае, на каждой итерации, при проверке совпадения окрестности с паттерном, к аккумулятору в функции приспособленности добавится максимальное значение по всем паттернам.

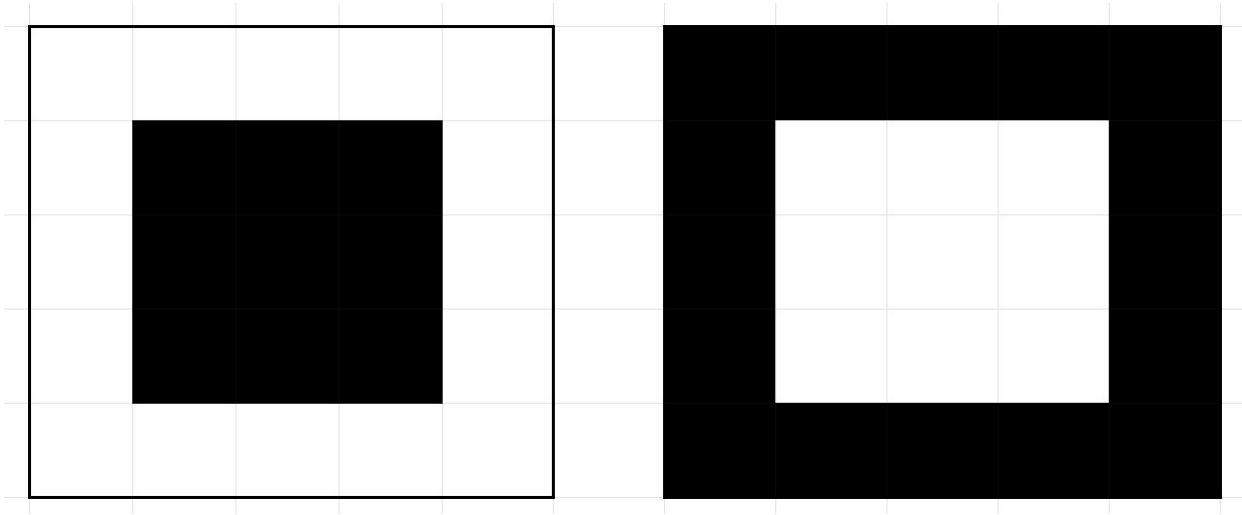

Рисунок 4 – Пример случая, когда проверяется совпадение с несколькими паттернами

Полный код функции приведен в приложении Б.

В идеальном случае, нам хотелось бы высчитать некоторую нормализованную функцию приспособленности, которая бы для сетки любой размерности и любого набора паттернов давала бы значение в диапазоне от 0 до 1 (или 100, для наглядности). Подобная функция учитывала бы максимально возможное количество изображений на паттерне. Таким образом мы могли бы поделить количество вхождений паттерна в изображении на максимально возможное и измерить процент совпадения. В реальности же это достижимо только с помощью полного перебора всей сетки, что в самом простом случае, без оптимизаций, при размере изображения 64×64, требовало бы произвести $2^{64 \times 64}$ подсчетов приспособленности. Поэтому для упрощения, будем считать значение приспособленности равным 100 если изображение полностью состоит из паттернов без пересечения. Например, если исходное изображение имеет размерность 64×64, а паттерн — 4×4, то на изображении возможно разместить $(64 \times 64)/(4 \times 4) = 256$ паттернов.

Подведем подсчет приспособленности на последних итерациях и сделаем возможным устанавливать количество итераций перед процессом эволюции. Между двумя подсчетами приспособленности происходит ровно одно обновление клеточных автоматов на сетке.

Если запустить программу с профилировщиком, можно заметить, что самым затратным по времени местом в программе является именно подсчет приспособленности (см. рис. 5). Это не удивительно, ведь для каждого пикселя сетки приходится проверять соответствие окрестности с заданным паттерном,

а с увеличением его размера в X раз, в столько же раз увеличивается и количество операций доступа при подсчете приспособленности. Далее попробуем оптимизировать этот процесс.

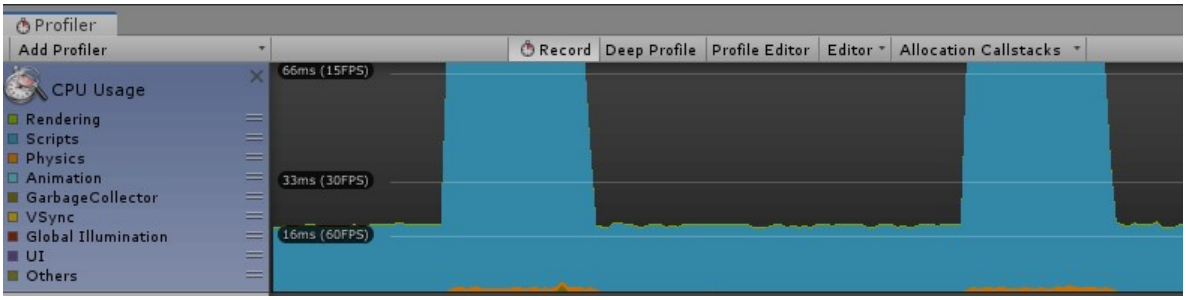

Рисунок 5 – Фрагмент окна профилировщика. Видимые подъемы связаны с процессом подсчета приспособленности

2.2 Подсчет приспособленности на основе данных с GPU

Попробуем снизить нагрузку во время подсчета приспособленности клеточных автоматов на сетке за счет переноса обновления сетки с процессора на видеокарту. Так как визуализация клеточных автоматов с помощью шейдеров уже готова, достаточно считывать изображение перед каждым подсчетом приспособленности.

После считывания, переведем пиксели цвета изображения в числа 0 и 1 и воспользуемся готовой функцией подсчета приспособленности.

```
private float CalculateFitness(Texture2D texture2D, Pattern[] patterns)
{
  texturePixels = texture2D.GetPixels32();
  return CalculateFitness(texturePixels, texture2D.width, texture2D.height, patterns);
}

byte[] texturePix = null;
private float CalculateFitness(Color32[] texturePixels, int texW,
        int texH, Pattern[] patterns)
{
  var texSize = texW * texH;
  texturePix = new byte[texSize];
  for (int i = 0; i < texSize; i++)
  {
      texturePix[i] = texturePixels[i].a;
  }
  return CalculateFitness(texturePix, texW, texH, patterns);
}
```

Вопреки ожиданиям, это не ускорило программу. Это можно объяснить

тем, что скорость передачи данных с видеопамяти в оперативную память крайне медленная [19].

2.3 Оптимизация подсчета приспособленности

Воспользуемся оптимизацией, описанной в разделе 1.2.2, с отличием в том, что паттерн может быть произвольного размера. Для начала переведем двоичные числа, образованные последовательным считыванием слева направа каждой строкой паттерна, в десятичное число и сохраним эту информацию во вспомогательном массиве. Данные числа останутся неизменными. Далее, проведем сравнение битов сетки клеточного автомата и паттерна.

Повторим действия, описанные в разделе 1.2.2 для сетки клеточного автомата. Аналогично, переведем двоичные числа, образованные битами, в десятичные и запишем в массив. Таким образом, размерность массива по обоим измерениям будет совпадать с полученным для паттерна.

Имея два двоичных числа в десятичном представлении, мы можем получить новое двоичное число, которое будет состоять исключительно из различающихся битов в двух исходных числах. Делается это с помощью операции XOR (сложение по модулю 2). В языке программирования C# она обозначается как ^.

В результате, мы получаем новое десятичное число, количество единиц в двоичном представлении которого — количество битов в окрестности ячейки, **не** совпадающих с паттерном. Для быстрого подсчета единиц, воспользуемся параллельным SWAR-алгоритмом, подробную информацию про который можно посмотреть в источнике [20].

После подсчета разницы, для каждого числа-буфера, полученного для сетки, используем побитовый сдвиг влево с выбрасыванием крайнего левого бита (используя бинарное «И») и добавим новое число справа. Повторим алгоритм, пока не дойдем до конца сетки.

Далее вставлен основной фрагмент описанного кода. Полный код доступен в приложении В.

```
for (short j = 0; j < textureWidth; j++)
 {
   cornerPixel = j + i * textureWidth;
   for (byte p = 0; p < patterns.Length; p++)
   {
       curErr[p] = 0;
```

```
    }
    int minErrors = patternErrors + 1;

    for (byte k = 0; k < patternHeight; k++)
    {
      int ind = (qOffset + k) % patternHeight;
      screenLine = screenLines[ind];
      newErr = patternLines[k] ^ screenLine;

      for (byte p = 0; p < patterns.Length; p++)
      {
        newErr = newErr - ((newErr >> 1) & 0x55555555);
        newErr = (newErr & 0x33333333) + ((newErr >> 2) & 0x33333333);
        curErr[p] += (((newErr + (newErr >> 4)) & 0x0F0F0F0F) * 0x01010101) >> 24;
      }

      screenLine <<= 1;
      if (screenLine >= patternCycle) screenLine -= patternCycle;
      nextPixInd = (((i + k) % textureHeight) * textureWidth) + (j + patternWidth) % textureWidth;
      screenLine += texturePixels[nextPixInd];

      screenLines[(qOffset + k) % patternHeight] = (screenLine);
    }

    for (byte p = 0; p < patterns.Length; p++)
    {
        if (curErr[p] < minErrors) minErrors = curErr[p];
    }
    fitness += minErrors <= patternErrors ?
     (1 + patternErrors - minErrors) * 1f / (patternErrors + 1) : 0;
}
```

Даже с учетом подобных и ранее описанных оптимизаций в разделе 1.2.2, данный этап остается самым высоко нагруженным. Поэтому для пользователя программы сделана возможность определить количество подсчетов приспособленности на каждом цикле эволюции.

2.4 Эволюция

Этап эволюции начинается после завершения подсчетов приспособленности клеточных автоматов на последних кадрах. На основании полученных данных будет отбираться половина особей с наивысшей приспособленностью, из них случайным образом будут созданы пары, каждая из которых создаст два потомка. Правила переходов для потомков формируются из генов родителей, которые также сохраняются до следующего этапа эволюции. В работе будут

рассмотрены два вида скрещивания.

Для первого вида скрещивания выберем случайный номер бита в генах (правила перехода длиной 512), будем называть его «разделителем». Все биты первого родителя до разделителя станут основой для правила перехода первого дочернего клеточного автомата, все биты с номером $\geq$ разделитель возьмутся из второго родителя. Для второго дочернего клеточного автомата, до разделителя берутся биты из второго родителя, а после — из первого.

Второй вид скрещивания реализован с помощью случайного выбора родителя для **каждого** бита.

После скрещивания случайным образом мутируем гены автоматов, полученных после этапа эволюции. В случае клеточных автоматов с возможными состояниями 0 и 1, 0 меняется на 1 и наоборот. Разрешим до X мутаций, а вероятность того что выбранный клеточный автомат мутирует — Y. Оба параметра настраиваются пользователем перед экспериментом.

2.5 Сбор статистики и визуализация результатов

Перед началом эксперимента по поиску клеточного автомата, воспроизводящего паттерн, случайным образом создадим идентификационный номер (ID) эксперимента. Название сопутствующих файлов со статистикой будет содержать в суффиксе ID эксперимента. Во время исполнения эксперимента, можно наблюдать за графиком приспособленности (рис. 6) и генофондом популяции (рис. 7-8).

После каждого эксперимента статистика записывается в файлы. Полная информация о настройках и результатах (средняя приспособленность, время, количество итераций эволюции) эксперимента, максимальные и средние значения приспособленности на каждой итерации, опорные биты (см. раздел 3.2). Данные файлы расположены в папке:

`{Расположение_проекта}/Assets/SimulationData/`

После завершения эксперимента, в файл также записываются и правила переходов в виде генов. Файл расположенный в пути:

`{Расположение_проекта}/Assets/SimulationData/Genes/`

Каждый файл имеет название `G-{Имя_Паттерна}-{IDэксперимента}.txt`.

В нем ген каждого клеточного автомата записывается в новой строке, и, соответственно, количество строк в файле определяется числом клеточных автоматов в проведенном эксперименте.

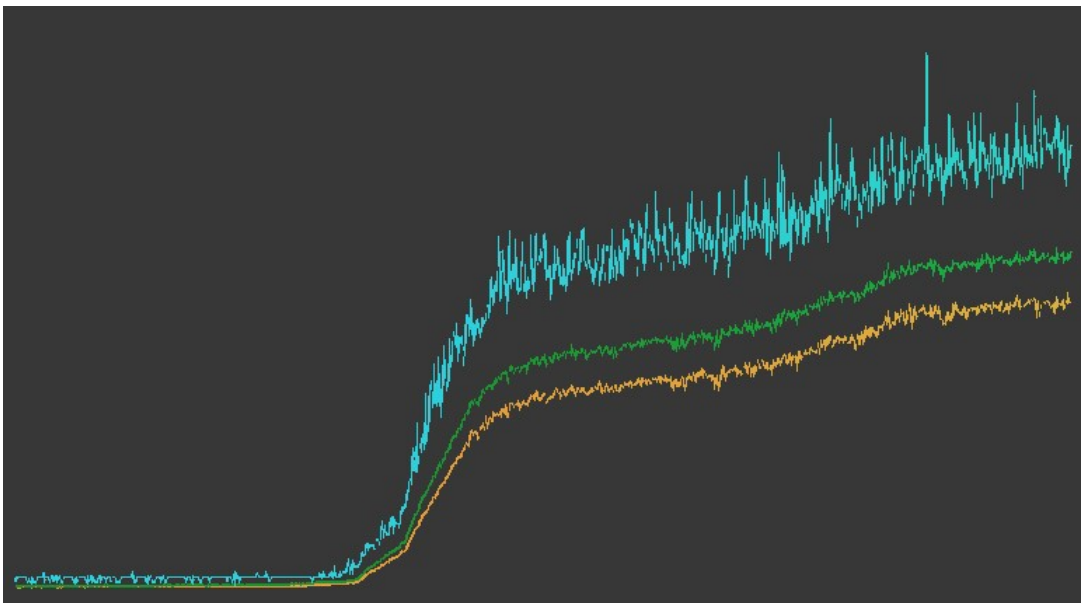

Рисунок 6 – Динамически построенный график приспособленности на эксперименте. Подробные числовые значения доступны в файлах с префиксом «FH-»

Для просмотра клеточных автоматов после эксперимента создана дополнительная сцена в Unity с названием GeneVisualisation. В основном, она отличается от главной сцены тем, что в ней имеется одна большая сетка размерности 1024×1024 и отключена возможность эволюции автоматов. Для считывания файла с генами создан дополнительный скрипт. В редакторе необходимо выбрать файл с данными генов, созданный автоматически после эксперимента. Запустив сцену, можно увидеть, как сетка меняет свое изначальное изображение 9, используя правила перехода из файла. Для переключения между всеми правилами в наборе, необходимо нажать стрелку влево или стрелку вправо на клавиатуре.

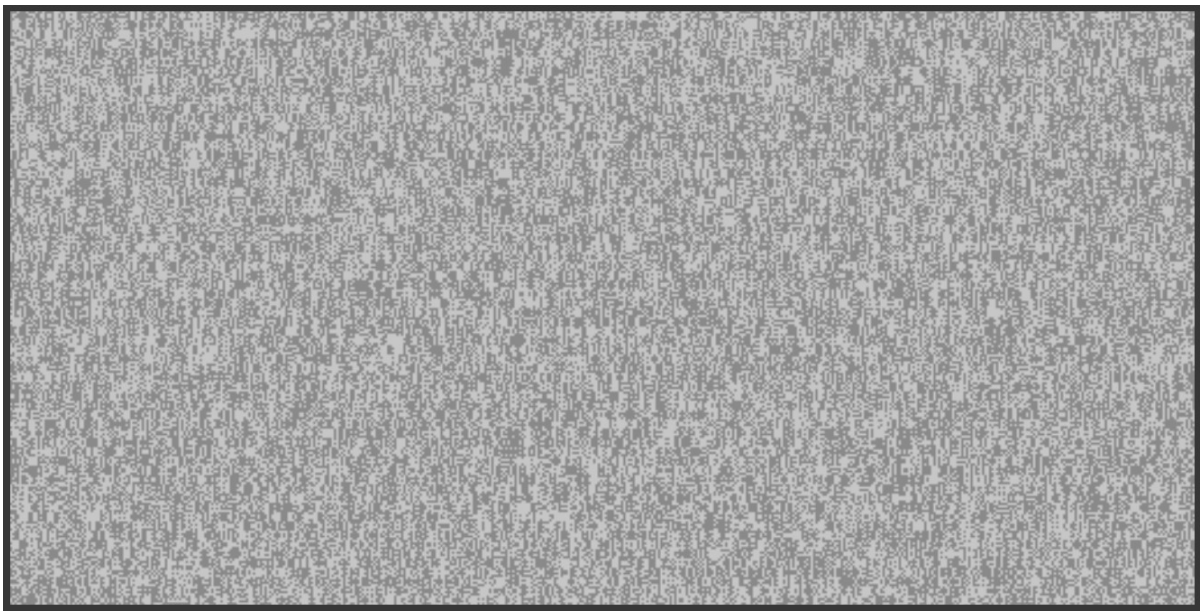

Рисунок 7 – Генофонд популяции на старте. По оси X — биты генов от 0 до 512, по оси Y — клеточные автоматы в популяции (256)

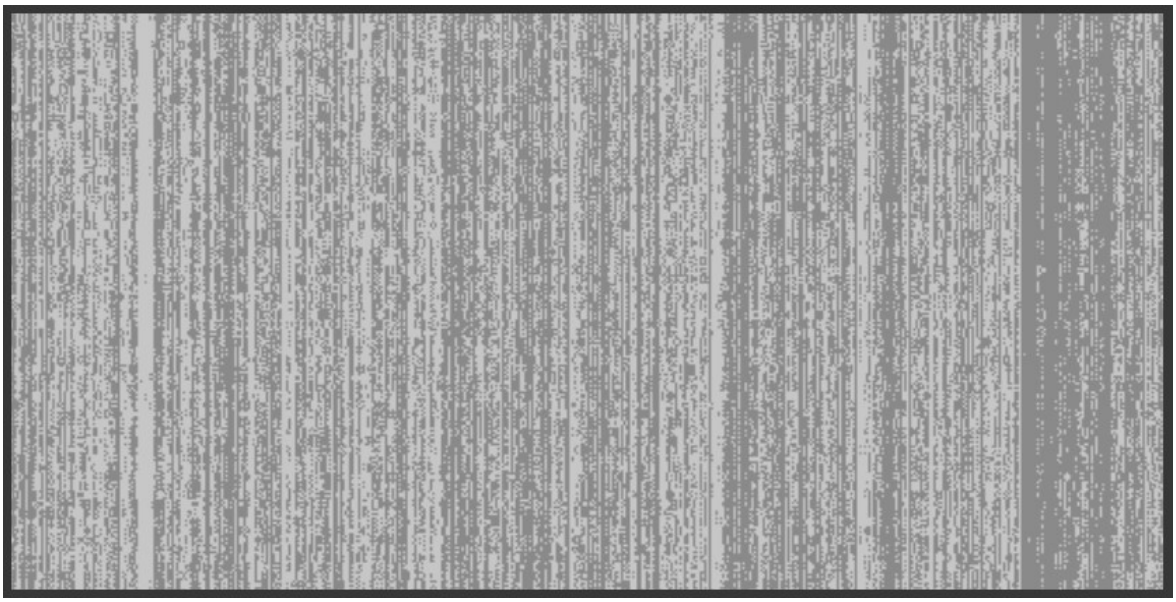

Рисунок 8 – Генофонд популяции в автомате с высокой приспособленностью. Можно заметить, что некоторые биты гена остаются неизменными у всех особей

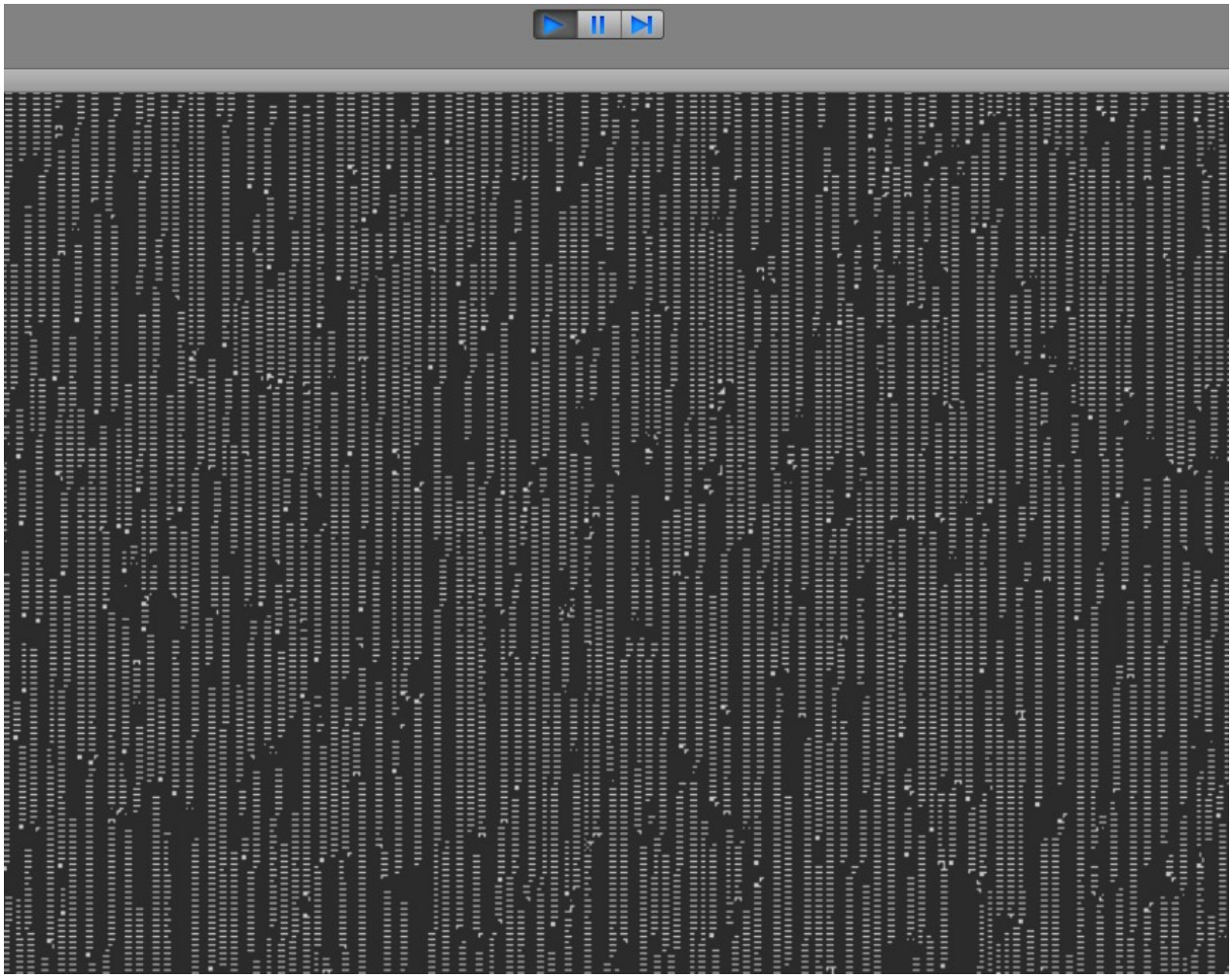

Рисунок 9 – Сцена для визуализации клеточных автоматов после эксперимента. Данный результат получен при попытке воссоздать паттерн «квадрат 3×3 с рамкой в 1 пиксель противоположного цвета».

3 Эксперименты

Данный раздел будет посвящен проведению экспериментов для сбора статистики, с помощью которой в дальнейшем можно выявить зависимость между параметрами эксперимента и скоростью нахождения клеточных автоматов, удовлетворяющих заданным условиям.

Перед выполнением экспериментов были поставлены следующие вопросы. Возможно ли найти такие параметры модели, которые универсально для любого паттерна будут приводить к быстрому нахождению клеточного автомата, моделирующего его? Если нет, то возможно ли подобрать заранее параметры, наиболее подходящие для заданного паттерна?

3.1 Конфигурация параметров эксперимента

Перед началом эксперимента в окне редактора (см. рис. Unity 10) можно определить следующие значения:

- `Update Period` — минимальное время между кадрами, необходимое перед следующим обновлением автомата;
- `Screen Size In Pixels` — размер квадратной сетки (в количестве ячеек) по одному измерению;
- `Virtual Screens In Simulation` — (кратное 4) количество клеточных автоматов в эксперименте;
- `Screens In Simulation` — количество автоматов, для которых будет действовать визуализация;
- `Time To Evolution` — время до первого подсчета приспособленности автоматов;
- `Mutation Percent` — вероятность, определенная для каждого автомата, того, что он мутирует;
- `Mutate Bits Up To` — X, количество битов, которые могут мутировать: от 1 до X;
- `Pivot Bit Fitness Threshold` — пороговое значение приспособленности, начиная с которого будут отслеживаться опорные биты (подробнее в разделе 3.2);
- `Write To Global Pivot Bits` — записывать ли опорные биты в данном эксперименте в файл (подробнее в разделе 3.2);
- `Fitness Calculations Needed` — количество подсчетов приспособлен-

ности перед этапом эволюции;

— `MS Fitness Threshold` — пороговое значение приспособленности для завершения эксперимента и перехода к следующему;
— `MS Calculations After Threshold` — дополнительное количество этапов эволюции после достижения порога (проверка, может ли клеточный автомат развиваться дальше, даже после достижения указанного порога);
— `Evolution Step Limit` — пороговое значение количества этапов эволюции для завершения эксперимента и перехода к следующему;
— `Cross Separation` — проводить ли скрещивание на основе разделителя (подробнее в разделе 2.4).

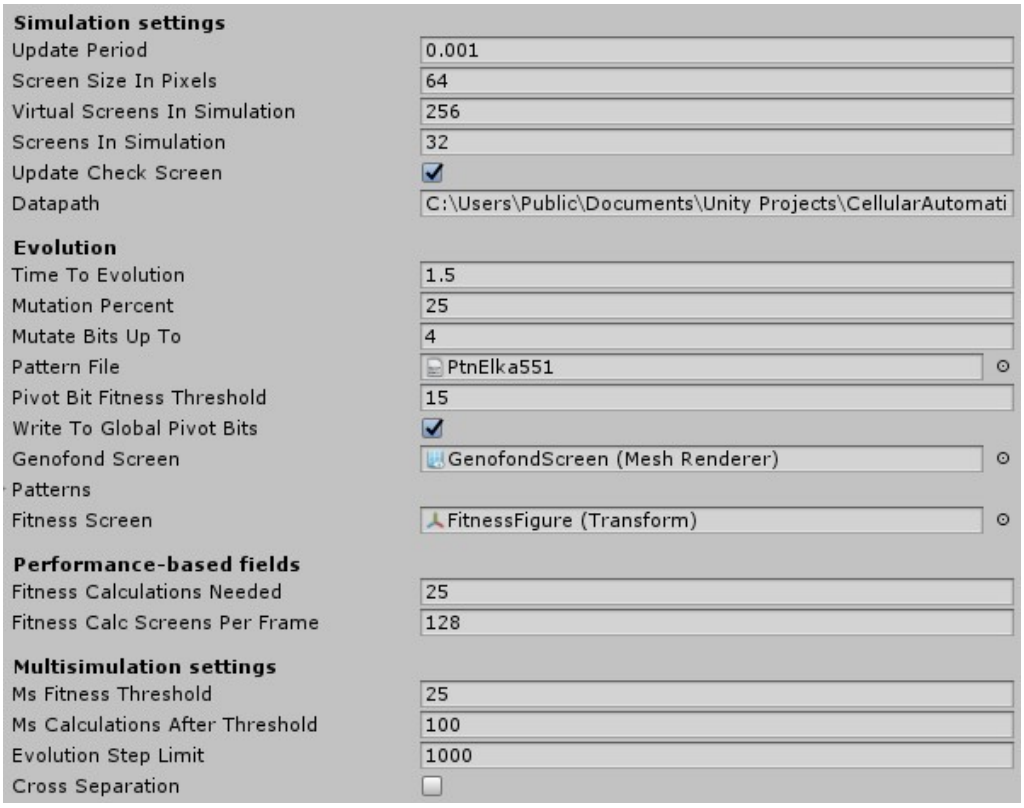

Рисунок 10 – Конфигурация эксперимента.

3.2 Поиск закономерностей в таблице переходов

Предположим, что для любого правила клеточного автомата существуют биты, являющиеся наиболее важными при формировании определенного паттерна. Далее, для удобства, будем называть эти биты «опорными». Тогда, для их нахождения создадим дополнительный массив, в котором будем хранить «ценность» бита. Определим её следующим образом:

1. К значению ценности бита прибавим 1, если бит гена равен единице, и -1 в противном случае.

2. Если приспособленность автомата меньше средней, то взять результат из п.1 и домножить на -1.
3. Итоговая ценность бита берется как значение по модулю от ценности бита.

Будем считать **опорными** те биты, значение по модулю которых превосходит среднее значение по модулю ценности среди всех битов правила.

Введенные нами алгоритм объясняется предположением, что в правиле найдутся биты, которые будут присутствовать только в автоматах с высокой приспособленностью. Добавим дополнительное ограничение на то, что подсчет опорных битов будет вестись лишь начиная с заданного процента приспособленности. Для наглядности будем отображать опорные биты на генофонде клеточных автоматов отдельным (оранжевым) цветом, как на изображении 11.

Рисунок 11 – Генофонд автоматов. Оранжевым цветом отображаются опорные биты.

Однако, само по себе нахождение опорных битах в одном эксперименте не позволит сделать выводы о закономерностях в правилах. Для того чтобы узнать, сохраняются ли опорные биты для повторных попыток найти правила, воспроизводящих заданный паттерн, необходимо найти «глобальные опорные биты». После каждого эксперимента запишем все опорные биты в новую строку в файл. Перед началом следующего эксперимента считаем строки с опорными битами и прибавим $+1$ к каждому глобальному значению ценности бита. Все биты, глобальное значение ценности которых превышает количество множеств опорных битов, будут считаться глобальными опорными битами. Отобразим их на генофонде отдельным (бирюзовым) цветом, как на

изображении 12.

Анализ результатов и выводы будут отражены в дипломной работе.

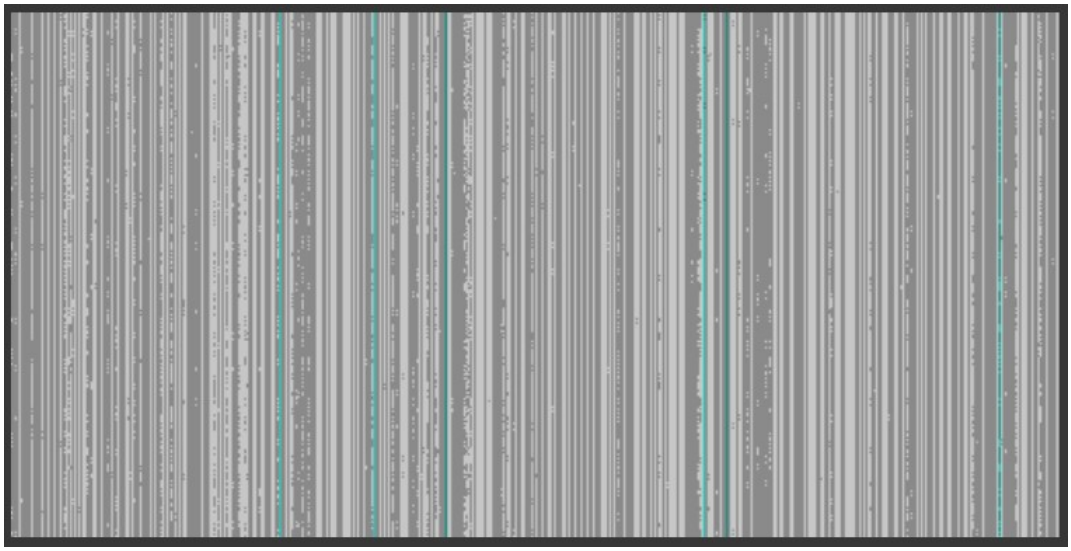

Рисунок 12 – Генофонд автоматов. Бирюзовым цветом отображаются глобальные опорные биты.

3.3 Описание экспериментов

В настоящей работе было проведено 436 экспериментов над паттернами 13:

— 93 над «крестом 5×5 с одной ошибкой»;
— 40 над «елкой 5×5 с одной ошибкой» (в трех конфигурациях);
— 44 над «елкой 5×5 с двумя ошибками» (в двух конфигурациях);
— 236 над «квадратом 3×3 и рамкой 1 пиксель противоположного цвета с двумя ошибками» (в пяти конфигурациях);
— 12 над «крестом 5×5 и рамкой 1 пиксель противоположного цвета с четырьмя ошибками» (в трех конфигурациях);
— 11 над «повернутой в 4 стороны елкой 5×5 с одной ошибкой».

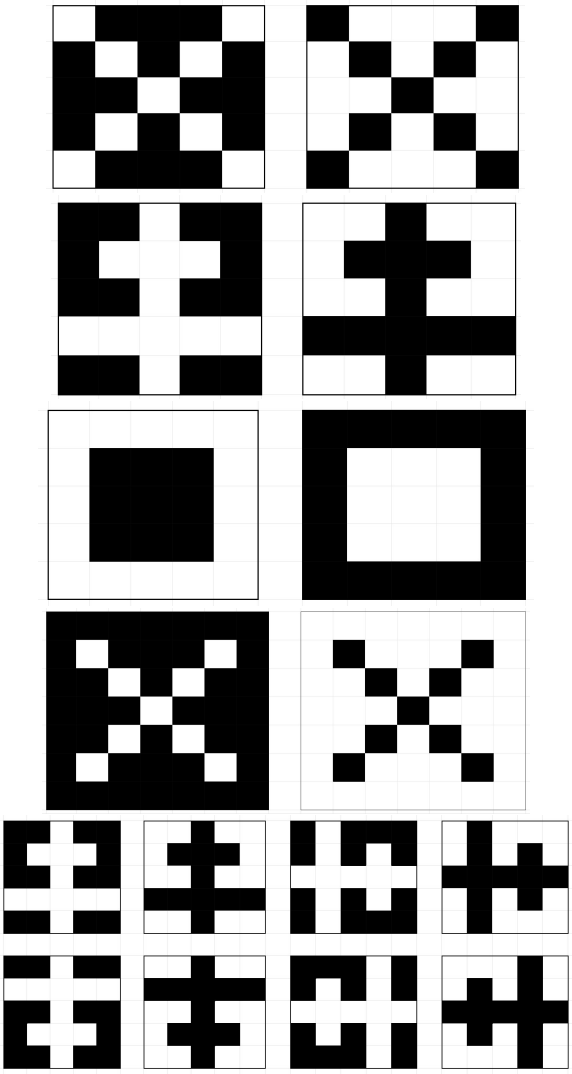

Рисунок 13 – Паттерны: «крест 5×5», «елка 5×5», «квадрат 3×3 и рамка 1 пиксель противоположного цвета», «крест 5×5 и рамка 1 пиксель противоположного цвета», «повернутая в 4 стороны елка 5×5» .

ЗАКЛЮЧЕНИЕ

В результате преддипломной практики было создано полноценное приложение для нахождения двумерных клеточных автоматов первого порядка, воспроизводящих заданный паттерн. Полученное приложение применимо для проведения экспериментов по подбору наилучших параметров для быстрого и эффективного (по значению приспособленности) поиска заданных клеточных автоматов. Полученные результаты можно использовать и для других клеточных автоматов: высшего порядка, с большим числом состояний, с другим множеством начальных состояний.

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1 [Клеточные автоматы - реализация и эксперименты]. — URL: https://www.osp.ru/pcworld/2003/08/166226/ (Дата обращения 27.05.2020). Загл. с экр. Яз. рус.

2 *Rosin, P.* Edge Detection Using Cellular Automata / P. Rosin, X. Sun. — 2014. — Pp. 85–103.

3 *Zhu, B.-P.* Public-key cryptosystem based on cellular automata / B.-P. Zhu, L. Zhou, F.-Y. Liu. — 10 2007. — Vol. 31. — Pp. 612–616.

4 [Conway's Game of Life]. — URL: https://www.conwaylife.com/ (Дата обращения 27.05.2020). Загл. с экр. Яз. англ.

5 *Ladd, A.* An application of lattice-gas cellular automata to the study of brownian motion / A. Ladd, D. Frenkel, M. Colvin. — 01 1988. — Vol. 60. — Pp. 975–978.

6 *Oono, Y.* Discrete model of chemical turbulence / Y. Oono, M. Kohmoto // *Phys. Rev. Lett.* — Dec 1985. — Vol. 55. — Pp. 2927–2931. https://link.aps.org/doi/10.1103/PhysRevLett.55.2927.

7 *Sander, L.* Fractal growth processes / L. Sander // *Nature*. — 08 1986. — Vol. 322. — Pp. 789–793.

8 [Game of Life]. — URL: https://mathworld.wolfram.com/GameofLife.html (Дата обращения 27.05.2020). Загл. с экр. Яз. англ.

9 *Chavoya, A.* Using a genetic algorithm to evolve cellular automata for 2d/3d computational development. — Vol. 1. — 01 2006. — Pp. 231–232.

10 *Mordvintsev, A.* Growing neural cellular automata / A. Mordvintsev, E. Randazzo, E. Niklasson, M. Levin // *Distill*. — 2020. — https://distill.pub/2020/growing-ca.

11 [Moore Neighborhood]. — URL: https://mathworld.wolfram.com/MooreNeighborhood.html (Дата обращения 27.05.2020). Загл. с экр. Яз. англ.

12 *Packard N.H., W. S.* Two-dimensional cellular automata / W. S. Packard, N.H. // *Journal of Statistical Physics*. — 1985. — Vol. 38. — Pp. 901–946.

13 [Fundamentals of Computer Programming]. — Pp. 150–151. — URL: https://books.google.ru/books?id=xYgCAQAAQBAJ&pg=PA150&

lpg=PA150&dq=bit+operations+performance+c%23&source=bl&ots=FPdhyQlS5h&sig=ACfU3U3dz1jBOUOWD_0553muTAbXAM2jFw&hl=ru&sa=X&ved=2ahUKEwiH8M_WntnpAhXMyKYKHbohC3QQ6AEwDXoECAoQAQ (Дата обращения 27.05.2020). Загл. с экр. Яз. англ.

14 *Dennunzio, A.* On the dynamical behaviour of linear higher-order cellular automata and its decidability / A. Dennunzio, E. Formenti, L. Manzoni, L. Margara, A. Porreca // *Information Sciences*. — 02 2019. — Vol. 486.

15 [Git - What is Git?]. — URL: https://git-scm.com/book/en/v2/Getting-Started-What-is-Git%3F (Дата обращения 27.05.2020). Загл. с экр. Яз. англ.

16 [FLOPs per Cycle for CPUs, GPUs and Xeon Phis]. — URL: https://www.karlrupp.net/2016/08/flops-per-cycle-for-cpus-gpus-and-xeon-phis/ (Дата обращения 27.05.2020). Загл. с экр. Яз. англ.

17 [Rendering Pipeline Overview]. — URL: https://www.khronos.org/opengl/wiki/Rendering_Pipeline_Overview (Дата обращения 27.05.2020). Загл. с экр. Яз. англ.

18 [Global Optimization Algorithms]. — URL: http://www.it-weise.de/projects/book.pdf (Дата обращения 27.05.2020). Загл. с экр. Яз. англ.

19 *v. Werkhoven, B.* Performance models for cpu-gpu data transfers // 2014 14th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing. — 2014. — Pp. 11–20.

20 [Bit Twiddling Hacks]. — URL: https://graphics.stanford.edu/~seander/bithacks.html#CountBitsSetParallel (Дата обращения 27.05.2020). Загл. с экр. Яз. англ.

ПРИЛОЖЕНИЕ А

Код оптимизированной функции переходов

```
private void UpdateCA(byte[] CAField, int ind)
{
    long size2D = screenSizeInPixels * screenSizeInPixels;
    int signal = 0;

    for (int i = screenSizeInPixels - 2; i < screenSizeInPixels; i++)
    {
        var iPix = i * screenSizeInPixels;
        for (int j = 0; j < screenSizeInPixels; j++)
        {
            screenSignals[iPix + j] = 0;
        }
    }
    for (short i = 0; i < screenSizeInPixels - 2; i++)
    {
        var iPix = i * screenSizeInPixels;
        var iPixm1 = (iPix + size2D - screenSizeInPixels) % size2D;
        var iPixm2 = (iPix + size2D - screenSizeInPixels - screenSizeInPixels) % size2D;
        signal = CAField[iPix + (screenSizeInPixels) - 1] * 2 + CAField[iPix];
        for (short j = 1; j < screenSizeInPixels; j++)
        {
            signal = (signal << 1) % 8 + CAField[iPix + j];
            var jm1 = j - 1;
            screenSignals[iPixm2 + jm1] += signal << 6;
            screenSignals[iPixm1 + jm1] += signal << 3;
            screenSignals[iPix + jm1] = signal;
        }
        signal = (signal << 1) % 8 + CAField[iPix];
        screenSignals[iPixm2 + screenSizeInPixels - 1] += signal << 6;
        screenSignals[iPixm1 + screenSizeInPixels - 1] += signal << 3;
        screenSignals[iPix + screenSizeInPixels - 1] = signal;
    }

    for (int i = screenSizeInPixels - 2; i < screenSizeInPixels; i++)
    {
        var iPix = i * screenSizeInPixels;
        var iPixm1 = (iPix + size2D - screenSizeInPixels) % size2D;
        var iPixm2 = (iPix + size2D - screenSizeInPixels - screenSizeInPixels) % size2D;
        signal = CAField[iPix + (screenSizeInPixels) - 1] * 2 + CAField[iPix];
```

```
            for (short j = 1; j < screenSizeInPixels; j++)
            {
                signal = (signal << 1) % 8 + CAField[iPix + j];
                var jm1 = j − 1;
                screenSignals[iPixm2 + jm1] += signal << 6;
                screenSignals[iPixm1 + jm1] += signal << 3;
                screenSignals[iPix + jm1] += signal;
            }
            signal = (signal << 1) % 8 + CAField[iPix];
            screenSignals[iPixm2 + screenSizeInPixels − 1] += signal << 6;
            screenSignals[iPixm1 + screenSizeInPixels − 1] += signal << 3;
            screenSignals[iPix + screenSizeInPixels − 1] = signal;
        }

        for (int i = 0; i < size2D; i++)
        {
            CAField[i] = (byte)allRules[ind][screenSignals[(i − screenSizeInPixels + size2D) % size2
                D]];
        }
}
```

ПРИЛОЖЕНИЕ Б

Код функции подсчета приспособленности

```
private float CalculateFitness(byte[] texturePixels, int texW, int texH, Pattern[] patterns)
{
  float fitness = 0;

  int textureWidth = texW; int textureHeight = texH;

  var patternHeight = patterns[0].patternSizeY;
  var patternWidth = patterns[0].patternSizeX;
  var patternErrors = patterns[0].patternErrors;
  var patternsCount = patterns.Length;
  int[] currentErrors = new int[patternsCount];

  for (short i = 0; i < textureHeight; i++)
  {
    for (short j = 0; j < textureWidth; j++)
    {
      int cornerPixel = j + i ∗ textureWidth;
      for (int p = 0; p < patternsCount; p++)
```

```
        {
          currentErrors[p] = 0;
        }

        for (byte ir = 0; ir < patternHeight; ir++)
        {
          for (byte jr = 0; jr < patternWidth; jr++)
          {
            int pixelIndex = (cornerPixel + ir * textureWidth + jr)
              % (textureWidth * textureHeight);
            for (int p = 0; p < patternsCount; p++)
            {
              currentErrors[p] += texturePixels[pixelIndex] ==
                patterns[p].pattern[ir * patternWidth + jr] ? 0 : 1;
            }
          }
        }
        int minErrors = patternErrors + 1;
        for (int p = 0; p < patternsCount; p++)
        {
          minErrors = currentErrors[p] < minErrors ? currentErrors[p] : minErrors;
        }

        fitness += minErrors <= patternErrors ?
          (1 + patternErrors - minErrors) * 1f / (patternErrors + 1) : 0;
      }
    }
    return fitness * 100 / (textureWidth * textureHeight);
}
```

ПРИЛОЖЕНИЕ В

Оптимизированный подсчет паттерна

```
private float CalculateFitnessOptimised(byte[] texturePixels, int texW, int texH, Pattern[] patterns)
{
  float fitness = 0;

  int textureWidth = texW; int textureHeight = texH;
  var tex2D = texW * texH;
  var patternHeight = patterns[0].patternSizeY;
  var patternWidth = patterns[0].patternSizeX;
  var patternErrors = patterns[0].patternErrors;
```

```
var patternRule = patterns[0].pattern;
int[] curErr = new int[patterns.Length];
int patternCycle = (1 << patternHeight);

int[] patternLines = new int[patterns[0].patternSizeX];
int newPatternLine = 0;
for (int i = 0; i < patternHeight; i++)
{
  newPatternLine = 0;
  for (int j = 0; j < patternWidth; j++)
  {
      newPatternLine = (newPatternLine << 1) + patterns[0].pattern[i * patternWidth + j];
  }
  patternLines[i] = newPatternLine;
}

int qOffset = 0;

int newScreenLine = 0;
for (qOffset = 0; qOffset < patternHeight - 1; qOffset++)
{
  newScreenLine = 0;
  for (int j = 0; j < patternWidth; j++)
  {
    newScreenLine = (newScreenLine << 1) + texturePixels[qOffset * patternWidth + j];
  }
  screenLines[qOffset] = newScreenLine;
}

int newLine = 0, iPix = 0, newPixInd = 0, cornerPixel = 0;
int screenLine = 0, newErr = 0, nextPixInd = 0;
for (short i = 0; i < textureHeight; i++)
{
  newLine = 0;
  iPix = ((i + patternHeight - 1) % textureHeight) * textureWidth;
  for (int j = 0; j < patternWidth; j++)
  {
    newPixInd = (iPix + j);
    newLine = (newLine << 1) + texturePixels[newPixInd];
  }
  qOffset = (qOffset + 1) % patternHeight;
```

```
screenLines[qOffset] = (newLine);

for (short j = 0; j < textureWidth; j++)
{
  cornerPixel = j + i * textureWidth;
  for (byte p = 0; p < patterns.Length; p++)
  {
      curErr[p] = 0;
  }
  int minErrors = patternErrors + 1;

  for (byte k = 0; k < patternHeight; k++)
  {
    int ind = (qOffset + k) % patternHeight;
    screenLine = screenLines[ind];
    newErr = patternLines[k] ^ screenLine;

    for (byte p = 0; p < patterns.Length; p++)
    {
      newErr = newErr - ((newErr >> 1) & 0x55555555);
      newErr = (newErr & 0x33333333) + ((newErr >> 2) & 0x33333333);
      curErr[p] += (((newErr + (newErr >> 4)) & 0x0F0F0F0F) * 0x01010101) >> 24;
    }

    screenLine <<= 1;
    if (screenLine >= patternCycle) screenLine -= patternCycle;
    nextPixInd = (((i + k) % textureHeight) * textureWidth) + (j + patternWidth) %
        textureWidth;
    screenLine += texturePixels[nextPixInd];

    screenLines[(qOffset + k) % patternHeight] = (screenLine);
  }

  for (byte p = 0; p < patterns.Length; p++)
  {
      if (curErr[p] < minErrors) minErrors = curErr[p];
  }
  fitness += minErrors <= patternErrors ?
   (1 + patternErrors - minErrors) * 1f / (patternErrors + 1) : 0;
}
```

```
  }

  return fitness * patternWidth * patternHeight * 100 / (textureWidth * textureHeight);
}
```