Аннотация

В работе обосновывается возможность эффективной реализации некоторых игр с использованием клеточных автоматов и разрабатывается методика, обеспечивающая такую реализацию. Реализация получается простой и хорошо формализуется. Вводится понятие расширенного клеточного автомата и доказывается, что он также является клеточным автоматом. Предлагается архитектура системы, с помощью которой реализуются «игровые» взаимодействия локального и глобального масштаба. Предлагается способ формирования проектной документации, которая изоморфно переводится в программный код. Приводится пример игры, спроектированной с использованием предложенного подхода, который иллюстрирует удобство метода. Оформлена проектная документация, в которой, в частности, описаны основные приемы формализации логики игр.

Оглавление

Введение

При реализации некоторых игр, как правило, целесообразно хранить данные о состоянии игры или часть этих данных в виде массива, изоморфного игровому полю. Например, при реализации игр типа «*Змейка*»[1] рационально хранить в матричной форме игровой мир — конфигурацию стен, расположение «яблок», наличие различных призовых элементов. В то же время, такие объекты, как сама змейка, контролируемая игроком, враги, если они предусмотрены и т.д., реализуются отдельно, не являясь интегрированными в матричную структуру. Это приводит к необходимости построения запутанной и сложно формализуемой логики взаимодействия этих объектов с окружающим миром.

В настоящей работе предлагается метод описания логики поведения игровых объектов и игрового мира на основе матричной модели — клеточного автомата. Также, поскольку взаимодействия в игре не являются только локальными, возникает необходимость в дополнительных компонентах, которые контролируют процессы, происходящие в клеточном автомате. Подобный подход позволяет проектировать рассматриваемый класс игр, а затем, руководствуясь лишь рядом правил, переносить проект в код. Поддержание проектной документации, полностью описывающей поведение программы, позволяет легко обнаруживать ошибки и устранять их, а также расширять и дополнять проект в дальнейшем [1].

[1] Правила рассматриваемого варианта игры «*Змейка*» описаны в разд. 4.1.

1. Клеточные автоматы и их адаптация для реализации игр рассматриваемого класса

1.1. Клеточные автоматы

Клеточные автоматы — дискретные детерминированные системы, поведение которых полностью определяется в терминах локальных взаимодействий [2].

Клеточные автоматы были использованы Джоном фон Нейманом для исследования самовоспроизведения [3]. Однако они в гораздо большей мере применяются в совершенно других областях. Если универсальной моделью для последовательных вычислений считается машина Тьюринга, то клеточные автоматы являются такой моделью для параллельных вычислений [4].

Отличительной особенностью клеточных автоматов является то, что они имеют структуру, объединяющую вычислительную компоненту и данные, с которыми автомат оперирует. Эта особенность хорошо сочетается с принципами объектно-ориентированного программирования, поскольку класс обладает такими же свойствами.

Клеточный автомат — это некоторая однородная «решетка», в каждой клетке которой находится конечный автомат. В двумерном случае простейшим вариантом является тетрагональная решетка. Возможны также клеточные автоматы с треугольной или гексагональной решеткой. Отличительная особенность конечных автоматов, являющихся элементами клеточного автомата, — то, что в них не используются входные и выходные воздействия, а, напротив, на каждом шаге новое состояние каждого конечного автомата определяется его собственным состоянием и состоянием его соседей (элементов окрестности).

Понятие окрестности клетки задается как свойство клеточного автомата, причем для каждой из клеток это понятие одинаково. Например, для двумерного клеточного автомата с тетрагональной решеткой чаще всего в качестве соседей рассматривают либо четыре клетки, соприкасающиеся с данной сторонами, либо восемь клеток, соприкасающиеся с данной хотя бы углами.

Клеточные автоматы обычно обладают следующими свойствами [5]:

- решетка однородна;
- взаимодействия локальны;
- множество состояний клетки конечно;
- изменения значений всех клеток происходят одновременно.

Иногда какие-то из этих свойств не выполнены. Например, в реализации, рассмотренной в работе [6], решетка не является однородной, однако остальные свойства клеточных автоматов выполняются.

Формально клеточный автомат — четверка объектов:

$$A = < R, S, O, f >,$$

где R — решетка автомата (рабочее пространство);

S — конечное множество состояний клетки;

O — определение окрестности клетки (множество соседей);

$f : S \times S^{|O|} \to S$ — функция переходов.

В данной работе рассматриваются двумерные клеточные автоматы на **тетрагональной решетке**, а окрестностью считаются клетки, которые соприкасаются сторонами. При практической реализации для удобства

хранения такая решетка может быть представлена в виде матрицы — двумерного массива (рис. 1).

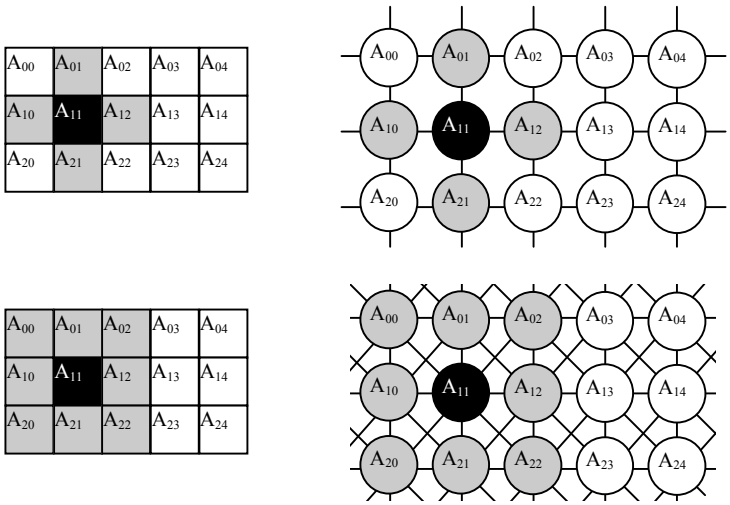


Рис. 1. Тетрагональная решетка с окрестностью из четырех клеток (сверху) и восьми клеток (снизу)

Для унификации терминологии, поскольку у игр рабочее пространство называется «полем», будем называть рабочее пространство клеточных автоматов рассматриваемого типа «тетрагональным полем». Соответственно, каждый конечный автомат — «клетка поля».

1.2. Расширенный клеточный автомат

Покажем, что для описания сложных систем, которыми являются некоторые компьютерные игры, оказывается недостаточно рассматривать только один автомат в каждой клетке.

Рассмотрим пример. Пусть имеется поле, каждая клетка которого представляет «ящик», в который может быть помещен предмет. «Ящик» может быть закрыт или открыт. Это его состояние. В «ящике» может храниться геометрическая фигура: куб, шар, тетраэдр. При этом поведение «ящика» зависит как от того, открыт он или закрыт, так и от того, что в нем содержится. Удобно использовать для хранения

информации о «ящике» две ячейки памяти: для состояния ящика и формы фигуры.

Докажем, что такую структуру в общем случае также можно представить в виде клеточного автомата, — поместить все свойства каждой клетки лишь в одну многозначную ячейку памяти, которая будет являться состоянием клетки. В дальнейшем все ячейки памяти будут рассматриваться как многозначные.

Доказательство. Действительно, пусть каждая клетка содержит k ячеек памяти, которые могут хранить значения из множеств S_i, $i = \overline{0,k-1}$. Если считать, что значение каждой ячейки — состояние некоторого автомата, то формально клеточный автомат, описывающий поведение поля, запишется в виде

$$A' = < R,\ \{S_i\}_{i=0}^{k-1},\ O,\ \{f_i\}_{i=0}^{k-1} >,$$

где $f_i : (S_0 \times \cdots \times S_{k-1}) \times (S_0 \times \cdots \times S_{k-1})^{|O|} \to S_i$ — функция, которая ставит состояние номер «i» текущей клетки в зависимость от всех состояний соседей и ее самой.

Пусть $S := S_0 \times \cdots \times S_{k-1}$. Это множество можно назвать множеством «общих состояний» клетки, оно содержит все возможные комбинации состояний из наборов S_i. Поскольку $\forall i \mid S_i \mid < \infty$, то и $\mid S \mid < \infty$.

Введем также «общую функцию переходов» $f : S \times S^{|N|} \to S$:

$$f(s^0, s^1, \ldots, s^{|O|}) = \left(f_0(s^0, s^1, \ldots, s^{|O|}), \ldots, f_{k-1}(s^0, s^1, \ldots, s^{|O|})\right),$$

где $s^i \in S$.

В рассмотренном примере с «ящиками» $k = 2$. Пусть $S_0 = \{open, closed\}$, $S_1 = \{cube, sphere, tetrahedron\}$. Тогда

$$S = \{(open, cube), (open, sphere), (open, tetrahedron), \ldots, (closed, tetrahedron)\}.$$

При этом $|S| = |S_0||S_1| = 2 \cdot 3 = 6$.

С использованием введенных обозначений, можно построить клеточный автомат

$$A =< R, S, O, f >,$$

поведение которого будет повторять поведение автомата A', причем автомат A будет являться классическим клеточным автоматом.

Назовем клеточный автомат, имеющий в каждой клетке несколько ячеек памяти, **расширенным клеточным автоматом**.

В данной работе будем использовать такие клеточные автоматы. Приведенное доказательство показывает, что такой автомат теоретически представим в виде классического, однако такое представление использоваться не будет.

Обозначим символом A_{ij} клетку автомата, находящуюся в ряду «*i*» и столбце «*j*»[2]. Чтобы обозначить значение определенной ячейки памяти расширенного клеточного автомата, используется верхний индекс: A_{ij}^n — значение ячейки памяти с номером «*n*» клетки A_{ij}. Эта ячейка памяти может содержать **состояния** клетки (в рассмотренном примере это одно состояние: «ящик открыт» или «ящик закрыт») или **свойства** клетки (в примере — форма хранящейся фигуры). Для того, чтобы отличать состояния клетки от свойств, будем обозначать свойства символом D_{ij}^n. Обратим внимание на то, что между состояниями и свойствами клетки сохраняется сквозная нумерация.

---

[2] Избрана принятая в линейной алгебре индексация элементов матрицы, и координаты перечисляются в порядке «строка, столбец»: *i* — *y*-координата элемента, а *j* — *x*-координата, а не наоборот.

Таким образом, в рассмотренном примере A_{ij}^0 — состояние ящика (открыт или закрыт), D_{ij}^1 — форма фигуры.

1.3. Представление и взаимодействие объектов

Клеточные автоматы характеризуются локальным взаимодействием между клетками. При этом они не способны описывать глобальные взаимодействия, такие как, например, появление некоторого объекта в одной части поля по событию, произошедшему в другой части поля. Это объясняется тем, что максимальная скорость распространения информации в такой структуре составляет расстояние до наиболее удаленной клетки в рамках окрестности за один шаг.

В то же время, участники многих процессов, происходящих в природе, тоже ограничены в скорости получения информации, либо же просто не имеют возможности анализировать все доступные данные.

Рассмотрим в качестве примера водителя автомобиля. Скорость распространения любой информации ограничена скоростью света, однако ее в масштабах данного примера можно считать бесконечной. Тем не менее, если представить, что автомобиль перемещается в тумане, и дальность видимости составляет всего три метра, скорость распространения информации становится равной скорости автомобиля: для того, чтобы увидеть, что в десяти метрах перед автомобилем тупик, требуется проехать еще семь метров. Если же тумана нет, в некотором роде похожая ситуация все равно имеет место: если дорога не оборудована специальными техническими средствами, водитель не знает, что происходит за углом, за холмом, в километре впереди. Поэтому клеточные автоматы бывают целесообразно использовать для реализации ряда задач, связанных с моделированием реальных процессов.

Действительно, клеточные автоматы находят применение в моделировании и анализе подобных процессов. Например, их применяют для статистического анализа городского транспортного потока [6]. В этой работе городские дороги представлены в виде клеток поля, а автомобили — в виде состояний некоторых клеток. С помощью такого клеточного автомата моделируется движение транспорта и анализируется возникновение заторов, общая скорость потока и другие параметры.

При реализации игр требуется описывать перемещения и взаимодействия некоторых объектов. При использовании для этого клеточных автоматов о присутствии игрового объекта в некоторой клетке информацию об этом дает состояние этой клетки, а не совпадение координат клетки и объекта, как это происходит при использовании объектно-ориентированного программирования.

Такой подход, будучи единственным возможным при использовании клеточных автоматов (ведь, по определению, данные хранятся только в виде состояний), имеет и ряд положительных свойств для удобства моделирования. Если в объектно-ориентированном программировании для того, чтобы поместить в игровое пространство еще один персонаж, требуется создавать его новый экземпляр, а при удалении — соответственно разрушать его, то, применяя клеточные автоматы, для этого достаточно назначить соответствующее состояние еще одной клетке или группе клеток. Персонаж, созданный в новом месте, будет вести себя так же благодаря однородности структуры клеточного автомата.

2. Построение логики игр

2.1. Декомпозиция управления

Как отмечалось в разд. 1.3, клеточный автомат может описать жизнь и взаимодействие объектов, поведение которых похоже на естественное поведение реальных. В то же время, в компьютерных играх, как правило, присутствует и «сверхъестественное» воздействие, например, судьи или ведущего, роль которого выполняет компьютер.

В качестве такого воздействия рассмотрим установку нового «яблока» на игровое поле в игре «*Змейка*» в тот момент, когда змейка, управляемая игроком, «съедает» очередное яблоко. Поглощение яблока — локальное действие: чтобы съесть яблоко, голова змейки должна находиться в соседней клетке. Выставление яблока — глобальное действие: новое яблоко может появиться как в соседней со змейкой клетке, так и в совершенно другом конце игрового поля.

Примером глобального действия может служить также «воскрешение» врага в игре *Pac-Man*[3]. Если персонаж игры «съедает» специальный призовой объект, он может уничтожать врагов. Уничтожение врага — действие локальное, а новое появление врага, которое выполняется мгновенно и в заданном месте, действие глобальное.

Если бы можно было пренебречь необходимостью мгновенного появления, обе рассмотренные глобальные задачи — выставление яблока и воскрешение врага — решались бы на клеточных автоматах. Можно, например, запускать из точки, где произошло событие «съедено яблоко» или «уничтожен враг», круговую «волну» некоторого состояния. Когда

[3] *Pac-Man* — игра, разработанная в 1980 году японской компанией *Namco*.

«волна» достигнет специально помеченной (одним из своих состояний) клетки, эта клетка изменит свое состояние на необходимое. Такой подход имеет право на существование, однако, во-первых, как было отмечено выше, не обеспечивает мгновенность, а во-вторых, он требует значительного усложнения функции переходов.

Поэтому появляется необходимость «расщепить» (декомпозировать) управляющие функции на:

- функции, выполняемые клеточным автоматом;
- функции, выполняемые внешним управляющим «устройством».

В качестве такого «устройства» будем использовать один или несколько конечных автоматов, построенных на основе *Switch-технологии* [7]. Использование конечных автоматов для внешнего управления позволяет соблюдать столь же строгую формализацию задачи описания управляющей логики, как и при описании клеточного автомата [8]. Такие внешние автоматы будем называть «автоматами-супервизорами» или просто супервизорами.

2.2. Архитектура системы

Поскольку управляющие функции можно декомпозировать на две части, так же делится и сама система:

- клеточный автомат;
- внешние супервизоры.

Структурная схема системы и структурная схема клетки поля изображены на рис. 2 и рис. 3.

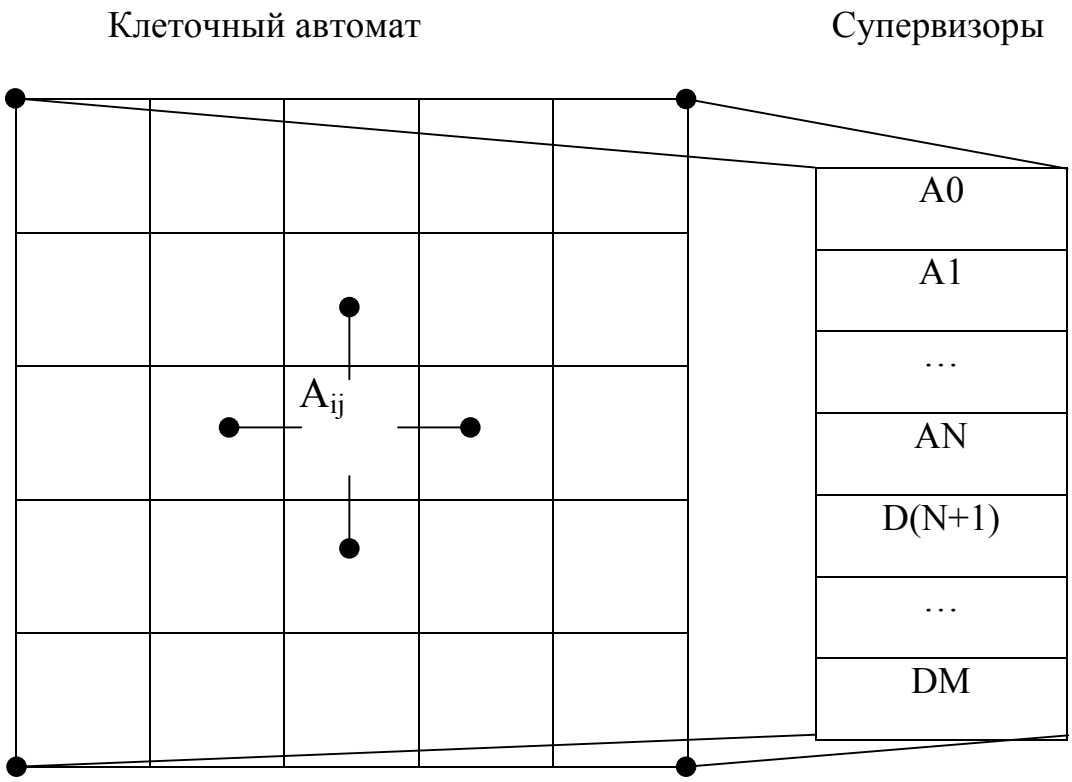


Рис. 2. Структурная схема системы

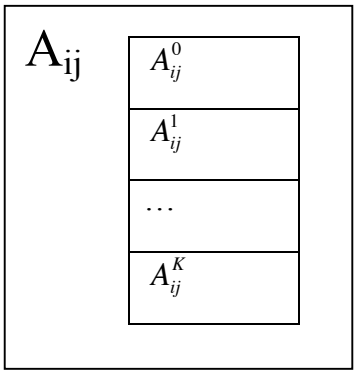


Рис. 3. Структурная схема клетки поля

Клеточный автомат:

- отвечает за локальные взаимодействия, включая перемещение всех объектов;
- каждая клетка получает информацию только о состоянии своих соседей и состоянии супервизоров;
- каждая клетка может модифицировать состояние некоторых супервизоров.

Помимо супервизоров понадобятся и вспомогательные «глобальные» ячейки данных, например, ячейка, в которой хранится текущий счет. Они схожи по роли с супервизорами. Поэтому удобно хранить их данные вместе с состояниями этих автоматов. Будем обозначать супервизоры символами A0, A1, …, AN, а глобальные ячейки данных — D(N+1), D(N+2), …, DM. Таким образом, для супервизоров и глобальных ячеек данных, как и для состояний и свойств клеток поля, используется сквозная нумерация.

При создании игр важно следить за тем, чтобы одни клетки не могли получать значения состояний супервизоров и глобальных ячеек, установленные другими клетками, поскольку это нарушило бы свойство одновременности изменения состояний всех клеток и свойство локальности взаимодействий в клеточном автомате. Проще всего учесть это ограничение, разделив супервизоры и глобальные ячейки на два класса:

- супервизоры и ячейки, состояния которых клетки поля могут изменять, но не могут считывать для последующего использования;
- супервизоры и ячейки, состояния которых могут только считываться клетками.

Если клетка должна произвести с числом, хранящимся в глобальной ячейке, некоторую математическую операцию, например, увеличить счет на единицу, это запись.

В любом случае при изменении состояния супервизоров и значений глобальных ячеек необходимо следить за тем, чтобы результат суперпозиции всех операций не зависел от того, в каком порядке они производятся.

Супервизоры:

- отвечают за глобальные действия;
- имеют доступ ко всем клеткам клеточного автомата как при чтении, так и при записи.

Пример набора супервизоров и глобальных ячеек памяти:

- супервизор A0 имеет два состояния: 0 — игра идет, 1 — игра окончена;
- ячейка D1 содержит счет (целое число);
- ячейка D2 содержит значение ноль, если призовой элемент не действует, или число больше нуля, которое соответствует количеству шагов клеточного автомата до окончания действия призового элемента.

В данном случае состояния автомата A0 и значение ячейки D1 клетками изменяются, а значение ячейки D2 только считывается.

2.3. Описание поведения клеточного автомата

Выше были описаны различные характеристики клеточных автоматов, кроме множества состояний и функции переходов. Предположим, что рассматривается не расширенный клеточный автомат, а клеточный автомат, каждая клетка которого содержит лишь одну

ячейку памяти. Поскольку клеточные автоматы на тетрагональном поле легко представимы в виде матрицы, будем считать, что конечные автоматы клеток организованы именно так. Состояние каждого элемента структуры будем обозначать, как было оговорено в разделе 1.2, символом A_{ij}^0, где i и j — номера строки и столбца, в которых содержится данный автомат, а 0 — номер единственной ячейки памяти.

Соответственно, в случае тетрагонального поля с четыремя соседями функция переходов f получает на вход пять чисел. При этом новое состояния клетки в позиции (i, j) вычисляется следующим образом:

$$A_{ij}^{0new} = f(A_{ij}^0, A_{i,j+1}^0, A_{i-1,j}^0, A_{i,j-1}^0, A_{i+1,j}^0).$$

Такой порядок следования аргументов в этом соотношении выбран по номерам квадрантов плоскости. Новое значение обозначено иначе, нежели то, которое является аргументом функции, для обеспечения одновременности смены состояний при последовательном вычислении значений функции [7], о чем речь пойдет ниже.

Функцию переходов клетки поля, как и в обычных конечных автоматах, удобно задавать в виде **графа переходов**. Его вершинами служат возможные значения состояния клеточного автомата (элементы множества S), а дуги показывают изменение состояний в зависимости от конфигурации состояний соседей. Построение графа выполняется так же, как для конечных автоматов, с той лишь разницей, что вместо входных переменных в качестве условий переходов на дугах указываются условия, составленные из состояний соседей, супервизоров и значений глобальных ячеек памяти, а вместо выходных воздействий — действия над данными, которые следует произвести. В качестве языка для формализации условий удобно использовать синтаксис не математической логики, а языка программирования C.

Пример 1. Рассмотрим клеточный автомат, реализующий «цепную реакцию» и ведущий подсчет «среагировавших» клеток в глобальной ячейке D0 — счетчике. Пусть в начале процесса все клетки имеют состояние «0». Как только (например, как результат действий пользователя) какая-то клетка перейдет в состояние «1», то при следующем шаге все ее соседи также перейдут в это состояние («среагируют»). На следующем шаге все повторится, но клеток в состоянии «1» будет уже больше. На рис. 4 проиллюстрирован этот процесс.

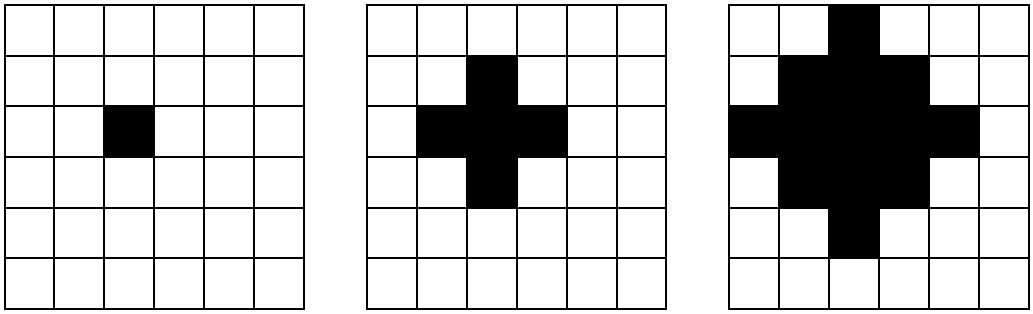


Рис. 4. Цепная реакция на первом, втором и третьем шаге

Граф переходов для клетки такого клеточного автомата изображен на рис. 5.

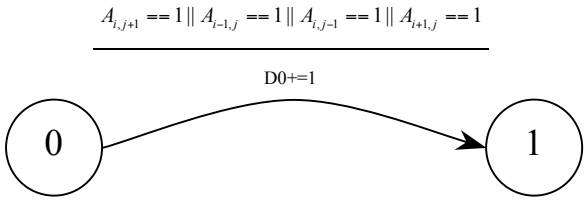


Рис. 5. Граф переходов клеточного автомата, реализующего цепную реакцию

В случае наличия сложных условий, которые с незначительными модификациями используются на нескольких дугах, можно использовать **макросы** — функции тех же аргументов, что и функция f, возвращающие элемент множества $S \cup \{\perp\}$ — состояние одного из соседей или значение «ложь».

Пример 2. Рассмотрим другой, более сложный вариант «цепной реакции» с двумя типами реакции и двумя типами «среагировавших» клеток. Если у «несреагировавшей» клетки есть сосед, «среагировавший» первым образом, она перейдет в первое состояние, если есть сосед, среагировавший вторым образом, она перейдет во второе состояние. Если клетка имеет соседей, среагировавших различным образом, установим следующий приоритет: «справа», «сверху», «слева», «снизу». Подсчет клеток, «среагировавших» тем и другим образом, будет производиться в глобальных ячейках D0 и D1 — счетчиках. Процесс жизни такого клеточного автомата проиллюстрирован на рис. 6.

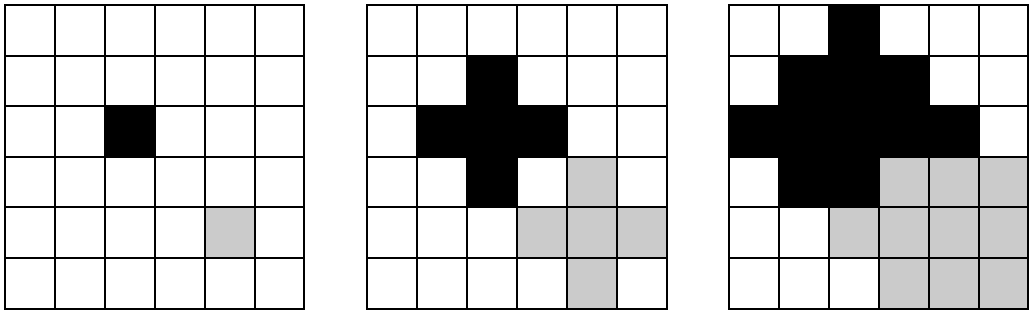


Рис. 6. «Двуцветная» цепная реакция на первом, втором и третьем шаге

На рис. 7 изображен граф переходов клетки клеточного автомата с тремя состояниями, реализующего такой процесс.

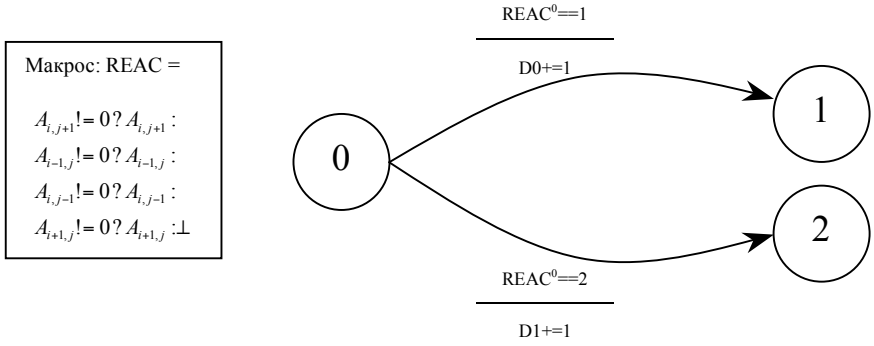


Рис. 7. Граф переходов клеточного автомата, реализующего «двуцветную» реакцию, использующий макрос

Рассмотрим теперь ситуацию с расширенным клеточным автоматом. Для определенности будем считать, что у этого клеточного автомата каждая клетка имеет k ячеек памяти. Из них l — состояния, а оставшиеся $k - l$ — свойства. В таком случае строится l графов переходов — для каждого состояния свой. Среди производимых действий при переходе в клетке, помимо изменения с ее помощью состояний супервизоров, может также производиться и смена свойств или других состояний этой же клетки.

2.4. Границы поля

Если требуется построить клеточный автомат с конечным полем, то необходимо отдельно рассматривать поведение граничных клеток. Функция переходов всегда имеет одинаковое количество аргументов — состояний соседей, но у граничных клеток некоторых соседей может не быть. Существует несколько подходов к устранению этого несоответствия. Предположим, что поле имеет размеры $m \times n$ (m строк и n столбцов), клетки нумеруются индексами $i = \overline{1,m}$ и $j = \overline{1,n}$.

1. Можно ввести фиктивные строки с номерами «0» и «m + 1» и фиктивные столбцы с номерами «0» и «n + 1», новые состояния которых не вычисляются, хотя могут быть изменены супервизорами. Таким образом, все клетки поля, для которых необходимо вычислять значения функции переходов, имеют всех соседей.
2. Можно применить «сворачивание в тор»: верхняя граница поля «замыкается» с нижней, а левая — с правой. Таким образом, например, правым соседом граничной клетки $A_{1,n}$ является левая граничная клетка $A_{1,1}$.

3. Можно совместить первые два подхода и сделать поле «цилиндрическим», то есть «замкнуть» только левую и правую границы и добавить фиктивные строки или, наоборот, «замкнуть» верхнюю и нижнюю границу и добавить фиктивные столбцы.

В примере реализации игры «*Змейка*», рассматриваемом в данной работе, используется первое решение. В то же время, в этом же примере можно было бы применить и второй подход, что придало бы игре дополнительные возможности: змейка смогла бы, уходя в «дырку» в стене, возвращаться с противоположенной стороны поля.

В целом при проектировании игр на клеточных автоматах может применяться любой из подходов.

2.5. Шаг системы

При каждом шаге системы выполняются следующие действия.

1. Создается «фотография» состояния клеточного автомата — ячеек данных всех его клеток.
2. По функциям переходов вычисляются новые состояния для каждой клетки автомата. В качестве параметров функциям переходов передаются состояния клеток окрестности из выполненной «фотографии». Если применяется расширенный клеточный автомат, то новые значения вычисляются для каждого из состояний в порядке возрастания номеров.
3. Вычисляются новые состояния и выполняются действия для всех супервизоров в порядке возрастания номеров.

Эти действия можно формализовать следующим образом (считается, что переменные, связанные квантором $\forall$, принимают все допустимые значения по возрастанию)[4]:

1. $\forall i, j \; A'_{ij} = A_{ij}$ — создание «фотографии».
2. $\forall i, j$:

   $\forall l, p \; Ap = f_l^p(A'_{ij}, A'_{i,j+1}, A'_{i-1,j}, A'_{i,j-1}, A'_{i+1,j}, A0, \ldots, AN)$ — изменение состояний некоторых супервизоров;

   $\forall l \; A_{ij}^l = f_l(A'_{ij}, A'_{i,j+1}, A'_{i-1,j}, A'_{i,j-1}, A'_{i+1,j}, A0, \ldots, AN)$ — изменение состояний клеток (локальные взаимодействия).
3. $\left\|A_{ij}\right\| = F_A\left(\left\|A_{ij}\right\|, A0, \ldots, AN\right)$ — глобальные действия;

   $\forall i \; Ai = F_i\left(\left\|A_{ij}\right\|, A0, \ldots, AN\right)$ — изменение состояний супервизоров.

Здесь функции f_l^p описывают изменение супервизоров клетками, функции F_i — функции переходов супервизоров, а функция F_A описывает глобальные действия.

Пример 3. Запишем на языке C++ программную реализацию шага клеточного автомата, реализующего «двуцветную» реакцию, который был рассмотрен в примере 2. Граф переходов клетки этого автомата изображен на рис. 7.

В приведенном ниже фрагменте кода используются следующие обозначения. `CellAut` — класс, реализующий поведение клеточного автомата. Значения глобальных ячеек памяти хранятся в массиве `state`. `Field` — двумерный массив экземпляров класса `Cell`, реализующего

[4] Пользуясь тем, что для свойств и состояний клеток, а также супервизоров и глобальных ячеек памяти введена сквозная нумерация, в данной записи, для простоты, не будем их различать.

ячейку. Таблица соответствий между математическими обозначениями и обозначениями в коде на языке C++ прведены в табл. 1.

Таблица 1. Соответствия между математическими обозначениями и обозначениями в коде на языке C++ в пространстве имен класса клеточного автомата

Математическое обозначение	Обозначение в C++
A_{ij}	`field[ i][ j] .state`
A_{ij}^{n}	`field[ i][ j] .state[ n]`
AN, DM	`state[ N] , state[ M]`

```
void CellAut::Step()
{
  int i, j;

  // Создание «фотографии»
  memcpy(&snapshot, &field, sizeof(field));

  // Каждая клетка делает шаг
  for (i = 1; i <= FIELD_H; i++)
    for (j = 1; j <= FIELD_W; j++)
      field[ i][ j] .Step(
                    snapshot[ i][ j+1] .state,
                    snapshot[ i-1][ j] .state,
                    snapshot[ i][ j-1] .state,
                    snapshot[ i+1][ j] .state,
                    state);

  // Шаги супервизоров
  // Супервизоров нет
}
```

При этом метод `Step` класса `Cell` реализуется следующим образом. В качестве аргументов метод получает указатели на массивы, хранящие состояния и свойства соседей (`right`, `up`, `left`, `down`), а также указатель на массив, хранящий состояния супервизоров и глобальных ячеек памяти (`super`). Собственное состояние хранится в массиве `state`. Таблица соответствий между математическими обозначениями и обозначениями в коде языка C++ прведены в табл. 2.

Таблица 2. Соответствия между математическими обозначениями и обозначениями в коде на языке C++ в пространстве имен функции переходов как метода класса клетки поля

Математическое обозначение	Обозначение в C++
A_{ij} (i,j — координаты текущей клетки)	`state`
$A_{i,j+1}, A_{i-1,j}, A_{i,j-1}, A_{i+1,j}$	`right, up, left, down`
$A^n_{i,j+1}, A^n_{i-1,j}, A^n_{i,j-1}, A^n_{i+1,j}$	`right[ n] , up[ n] , left[ n] , down[ n]`
AN, DM	`super[ N] , super[ M]`

```
#define REAC \
   ( \
   right[ 0] !=0? right: \
   up[ 0] !=0? up: \
   left[ 0] !=0? left: \
   down[ 0] !=0? down: \
   0 \
   )

void Cell::Step(const int* right, const int* up, const int* left,
const int* down, const int* super)
{
   // Шаги всех автоматов клетки

   // Шаги автомата A^0_ij
   switch (state[ 0] )
   {
   case 0:
      if (REAC[ 0]  == 1) {
         super[ 0]  += 1;
         state[ 0]  = 1;
      }

      if (REAC[ 0]  == 2) {
         super[ 1]  += 1;
         state[ 0]  = 2;
      }
      break;
   }
}
```

Из изложенного следует, что программный код изоморфен графам переходов клеточного автомата и супервизоров и получается из них однозначно. Таким образом, для задания логики игры достаточно построить графы переходов соответствующих автоматов, а код из них можно построить автоматически.

3. Анализ метода

3.1. Реализуемость логики игр с использованием клеточных автоматов

Если подойти к вопросу о реализуемости логики игр с использованием клеточного автомата формально, то ответ однозначен: с использованием описанной в разд. 2 системы, состоящей и клеточного автомата и набора супервизоров, можно реализовать любую игру, базирующуюся на тетрагональном поле. Действительно, если задать функцию переходов клеточного автомата как $f(A_{ij},\ldots) = A_{ij}$, а управление игровой логикой возложить на автоматы-супервизоры, то, поскольку теоретически с использованием конечных автоматов возможно реализовать управление любым вычислительным процессом [7,8], игра также теоретически реализуется.

На практике же такой ответ бесполезен. Имеет значение, возможно ли реализовать логику той или иной игры с **существенным** и **эффективным** использованием клеточного автомата. Ответ на такой вопрос требует анализа самой игры на предмет того, преобладают в игре локальные или глобальные взаимодействия.

Первый шаг применения рассматриваемой схемы — расщепление управления: локальные взаимодействия реализуются клеточным автоматом, глобальные — супервизорами. Чем больше в игре используется локальных взаимодействий, тем существеннее и эффективнее будет использоваться клеточный автомат, и наоборот.

3.2. Реализуемость игры «Змейка»

Рассмотрим в качестве примера игру «*Змейка*». Это игра, где преобладают локальные взаимодействия, такие как:

- голова переходит в пустую клетку;
- хвост уходит с клетки, оставляя ее пустой;
- голова съедает яблоко (или другой игровой объект);
- голова «натыкается» на стену;
- «приз», установленный на поле, исчезает после некоторого количества шагов.

В игре происходят также и глобальные действия:

- как только очередное яблоко съедено, нужно установить где-то на поле новое;
- в некоторые моменты времени где-то на поле должен помещаться приз.

Такой анализ показывает, что игра «*Змейка*» может быть эффективно реализована с использованием конечного автомата. В то же время, не для всех игр это так.

3.3. Реализуемость игры Tetris

Рассмотрим игру *Tetris*[5]. Эта игра вовсе не имеет локальных взаимодействий, если рассматривать в качестве окрестности поля 4 или 8 соседей. Глобальными же действиями являются:

- опускание падающей фигуры на один шаг;
- перемещение падающей фигуры влево или вправо;
- поворот падающей фигуры;

---

[5] Игра, созданная А. Пажитновым совместно с Д. Павловским и В. Герасимовым в СССР в 1985 году.

- удаление заполненного ряда и смещение всего содержимого «стакана» выше него, на один шаг вниз.

В комментариях нуждаются только первые два пункта. На рис. 8 показано как поведение соседа «b» клетки «a» влияет на поведение клетки «d». Если бы в клетке «b» не было блока или если бы он относился к падающей фигуре, клетка «d» должна была бы принять значение клетки «c», но в данном случае ее состояние не изменится. При этом клетка «d» при вычислении своего нового состояния не имеет информации о состоянии клетки «b», так как она не входит в ее окрестность.

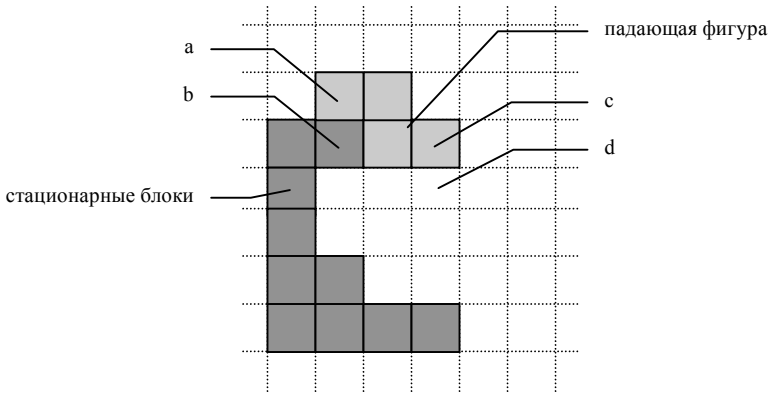


Рис. 8. Глобальность взаимодействий в игре *Tetris*

Подобную проблему можно решить, приняв размер окрестности 9×9 клеток (учитывая, что самая длинная фигура имеет линейный размер четыре клетки), однако описание функции переходов, которая будет являться функцией 81 аргумента, окажется слишком сложной задачей.

Другим подходом к реализации этой игры на клеточным автомате является использование многошагового автомата. Используя многошаговость, можно значительно сократить размер окрестности. Например, можно при каждом этапе опускания фигуры проводить два шага:

- передача данных;
- опускание фигуры.

Таким образом можно сократить необходимый размер окрестности до 5×5. На рис. 9 проиллюстрировано, как информация о нахождении блоков в клетках «c» и «d» на первом шаге передается в клетку «b», а на втором шаге информация, накопленная в клетке «b», влияет на вычисление нового состояния клетки «a». Перечеркнутыми квадратами с двойными рамками обозначены окрестности клеток «a» и «b».

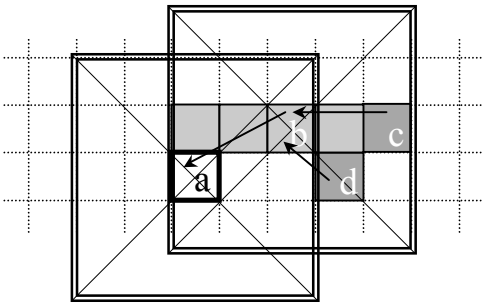


Рис. 9. Уменьшение требуемого размера окрестности за счет применения двухшагового клеточного автомата

Добавляя большее количество промежуточных шагов, можно сократить размер окрестности до окрестности из 4 клеток. Уменьшение размера окрестности сокращает количество аргументов функции переходов, однако не упрощает ее, а, напротив, значительно усложняет. Более того, для обеспечения многошаговости понадобится дополнительный супервизор, глобальная ячейка памяти или свойство клетки, которое будет содержать информацию о том, какой этап вычисления состояний производится в данный момент.

4. Пример реализации игры «Змейка»

4.1. Правила игры

Рассмотрим реализацию уже упоминавшейся ранее игры «*Змейка*» с использованием представленного метода. Существует множество вариаций этой игры, в этой работе мы будем придерживаться следующих правил:

- применяется прямоугольное поле, состоящее из квадратов, размером $m \times n$ клеток. Периметр поля занимает стена;
- персонаж, управляемый игроком — змейка — имеет начальную длину 5 клеток;
- на поле в любой момент времени находится одно яблоко;
- змейка должна «съесть» яблоко — игрок должен совместить голову змейки с яблоком, находящимся на поле;
- при съедании яблока зачисляется одно очко, длина змейки увеличивается на три, а на поле выставляется новое яблоко и яд;
- время от времени на поле появляется призовой элемент (*bonus*), который исчезает через некоторое время. Съедание этого объекта приносит дополнительные очки, но не влияет на длину змейки;
- игра заканчивается, если змейка упирается в стену, саму себя или съедает яд;
- игрок контролирует направление движения головы змейки.

Смысл игры — набрать максимальное количество очков.

4.2. Описание автоматов

Как указано в правилах, действие игры происходит на тетрагональном поле размера $m \times n$. Будем рассматривать тетрагональное поле с окрестностью из четырех клеток.

4.2.1. Описание клеточного автомата

Каждая клетка имеет состояние «тип» и свойства «направление головы», «время жизни» и «желудок».

Состояние A^0 — «тип». Это состояние определяет, что находится в данной клетке в данный момент времени.

Empty — клетка пуста (сокращенно E, в коде — `CA0_E`);

Snake — в клетке находится часть змейки (сокращенно S, в коде — `CA0_S`);

Apple — в клетке находится яблоко (сокращенно F, в коде — `CA0_F`);

Poison — в клетке находится яд (сокращенно P, в коде — `CA0_P`);

Wall — в клетке находится часть стены (сокращенно W, в коде — `CA0_W`);

Bonus — в клетке находится призовой элемент (сокращенно B, в коде — `CA0_B`).

Свойство D^1 — «направление головы». Если в клетке находится голова змейки, это свойство определяет направление ее движения.

0 — в клетке не голова;

1 — →;

2 — ↑;

3 — ←;

4 — ↓.

Свойство D^2 — «время жизни». Определяет, сколько тактов осталось до того момента, когда змейка уйдет с данной клетки. Данное свойство также определяет оставшееся время жизни призового элемента, если A^0==Bonus.

Свойство D^3 — «желудок». Показывает, что было в данной клетке до того, как на нее пришла змейка. В рассматриваемом варианте игры это свойство можно было бы заменить дополнительной глобальной ячейкой памяти, однако оно оставлено для общности и возможности расширения.

4.2.2. Описание супервизоров

Также для реализации игры понадобится супервизор «состояние игры» и глобальная ячейка памяти «счет».

Автомат A0 — «состояние игры».

0 — игра идет;

1 — игра окончена.

Ячейка D1 — «счет». Содержит текущий счет.

4.3. Построение графа переходов для клетки поля

После того, как состояния автоматов выбраны, для построения графа переходов необходимо определить переходы, условия переходов и действия, выполняемые при переходах.

При построении графа переходов для клетки клеточного автомата удобно ввести макрос *APPR_HEAD*, который будет показывать, должна ли при следующем шаге на данную клетку войти голова змейки, и если должна, то с какой стороны.

$$APPR\_HEAD =$$
$$A^0_{i,j-1} == S \,\&\,\& D^1_{i,j-1} == 1 ? A_{i,j-1} :$$
$$A^0_{i+1,j} == S \,\&\,\& D^1_{i+1,j} == 2 ? A_{i+1,j} :$$
$$A^0_{i,j+1} == S \,\&\,\& D^1_{i,j+1} == 3 ? A_{i,j+1} :$$
$$A^0_{i-1,j} == S \,\&\,\& D^1_{i-1,j} == 4 ? A_{i-1,j} : \perp$$

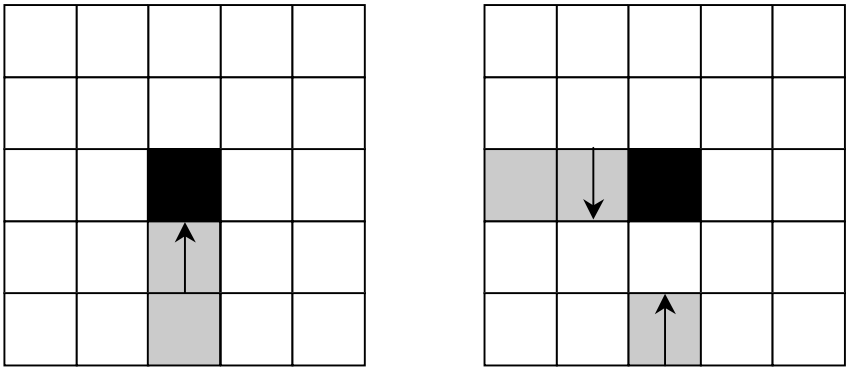
$A_{i-1,j-1}$	$A_{i-1,j}$	$A_{i-1,j+1}$
$A_{i,j-1}$	$A_{i,j}$	$A_{i,j+1}$
$A_{i+1,j-1}$	$A_{i+1,j}$	$A_{i+1,j+1}$

Поясним, как понимается эта запись. Если клетка справа имеет состояние «тип» Snake, и при этом ее свойство «направление головы» отлично от нуля (что значит, что в этой клетке голова змейки), то функция возвращает клетку справа. В противном случае аналогичные условия проверяются поочередно для клетки сверху, слева и снизу. Если же ни для одной из клеток такие условия не выполняются, возвращается логическая ложь.

Таким образом этот макрос возвращает клетку головы, если она при следующем шаге должна зайти на данную клетку (слева находится голова и движется вправо, снизу находится голова и движется вверх и т.д.) или логическую ложь $\perp$, если голова не приближается.

Поскольку макрос возвращает клетку, правомерна запись вида $(APPR\_HEAD)^n$. Такой символ обозначает значение ячейки памяти номер «n» клетки, которую возвратил макрос.

На рис. 10 изображен случай, когда голова приближается снизу и два случая, когда голова не приближается: она находится в соседней клетке, но направлена не на текущую, либо она имеет верное направление, но находится вне окрестности.



Голова приближается снизу

Голова не приближается ни в одном из двух случаев

Рис. 10. Работа макроса *APPR_HEAD*. Текущая клетка отмечена черным, клетки змейки — серые

Рассмотрим теперь основные события, происходящие в игре и сформулируем соответствующие им изменения состояний клеток.

Движение змейки. При движении змейки в ее клетках ($A_{i,j} : A^0_{i,j} == Snake$) должно уменьшаться значение, соответствующее времени жизни змейки. Если этот счетчик уже в состоянии «0», клетка меняет свое состояние на «Empty». Все описанное происходит в том и только том случае, когда в данную клетку не должна зайти голова ($APPR\_HEAD == \perp$). Эти правила порождают две дуги: Snake→Snake и Snake→Empty. Назовем их L1 и L2 соответственно. При этом

L1 — уменьшаем счетчик времени жизни при обычном движении;

L2 — счетчик времени жизни достиг нуля: змейки в этой клетке больше нет.

Сформулируем условия, помечающие эти дуги.

Условие для L1: $D^2_{i,j} > 1 \,\&\&\, !APPR\_HEAD$.

Условие для L2: $D^2_{i,j} == 1 \,\&\&\, !APPR\_HEAD$.

Жизненный цикл призового элемента. На каждом шаге время жизни каждого призового элемента уменьшается на единицу. Если время жизни истекло, клетка меняет свое состояние на «Empty». Добавим еще две дуги: Bonus→Bonus (L3) и Bonus→Empty (L4). При этом

L3 — уменьшаем счетчик времени жизни призового элемента;

L4 — счетчик времени жизни достиг нуля: призовой элемент исчез.

Сформулируем условия, помечающие эти дуги.

Условие для L3: $D^2_{i,j} > 1 \&\& !APPR\_HEAD$.

Условие для L4: $D^2_{i,j} \le 1$.

«Съедание» объектов. Для любого типа клетки, если при следующем шаге в ней должна оказаться голова ($APPR\_HEAD != \perp$), то:

- направление клетки принимает соответствующее состояние головы, которая должна в этой клетке оказаться;
- время жизни принимает соответствующее состояние головы, которая должна в этой клетке оказаться;
- «желудок» принимает состояние «типа» клетки, который она имеет в данный момент;
- состояние клетки изменяется на «Snake».

Такой набор правил порождает дуги в состояние Snake из: Snake (L5), Empty (L6), Bonus (L7), Apple (L8), Poison (L9), Wall (L10). Условия и действия у всех этих дуг формализуются одинаково. При этом

L5, L6, L7, L8, L9, L10 — в эту клетку зашла змейка: текущее состояние помещается в свойство D^3, в клетку копируется состояние и свойства заходящей головы.

Формально такие действия записываются следующим образом:

$$D^1 = (APPR\_HEAD)^1; D^2 = (APPR\_HEAD)^2; D^3 = A^0.$$

Для удобства записи будем обозначать это действие макросом $SNAKE\_EAT$.

Все эти дуги помечены одинаковыми условиями: $APPR\_HEAD$.

Теперь можно построить граф переходов (рис. 11).

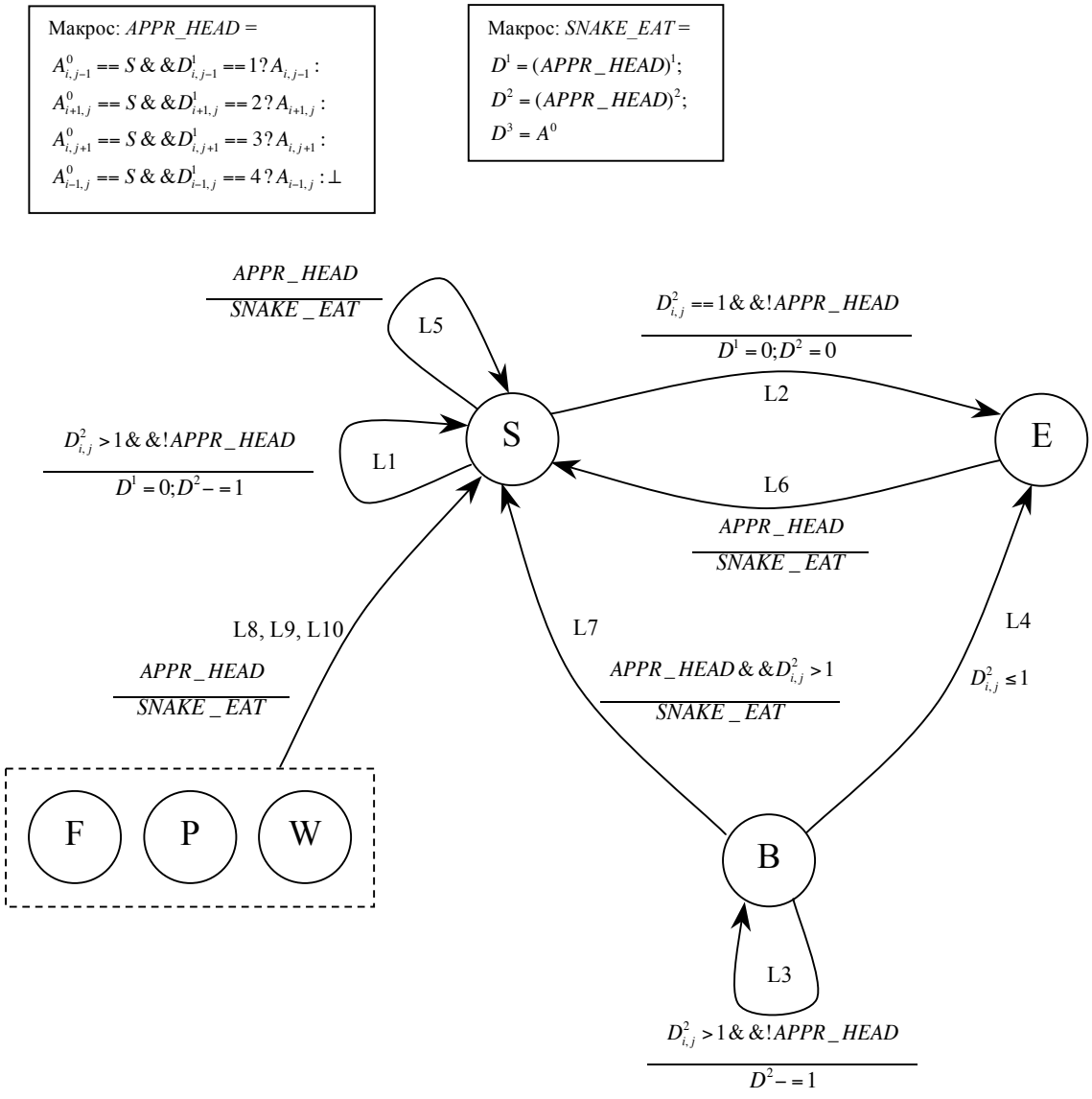


Рис. 11. Граф переходов клеточного автомата игры «*Змейка*»
E — Empty, S — Snake, F — Apple, P — Poison, W — Wall, B — Bonus

В виде программного кода на языке программирования C++ поведение клетки записывается полностью изоморфно. Таблица соответствия математических обозначений обозначениям кода C++ была приведена в разд. 2.5 (табл. 2).

```
#define APPR_HEAD \
  ( \
  (left[ 0] == CA0_S && left[ 1] == 1)?left: \
  (down[ 0] == CA0_S && down[ 1] == 2)?down: \
  (right[ 0] == CA0_S && right[ 1] == 3)?right: \
  (up[ 0] == CA0_S && up[ 1] == 4)?up: \
  0 \
  )

#define SNAKE_EAT \
  state[ 1] = APPR_HEAD[ 1] ; \
  state[ 2] = APPR_HEAD[ 2] ; \
  state[ 3] = state[ 0] ; \
  state[ 0] = CA0_S;


void SnakeCell::Step(const int* right, const int* up, const int*
left, const int* down, const int* super)
{
  switch (state[ 0] )
  {
  case CA0_S:
    // Дуга L1
    if (state[ 2] > 1 && !APPR_HEAD)
    {
      state[ 1] = 0;
      state[ 2] -= 1;
    }

    // Дуга L2
    if (state[ 2] == 1 && !APPR_HEAD)
    {
      state[ 1] = 0;
      state[ 2] = 0;
      state[ 0] = CA0_E;
    }

    // Дуга L5
    if (APPR_HEAD)
    {
      SNAKE_EAT
    }
    break;

  case CA0_E:
    // Дуга L6
    if (APPR_HEAD)
    {
      SNAKE_EAT
    }
    break;
```

```
  case CA0_B:
    // Дуга L3
    if (state[ 2] > 1 && !APPR_HEAD)
      state[ 2] -= 1;

    // Дуга L4
    if (state[ 2] <= 1)
      state[ 0] = CA0_E;

    // Дуга L7
    if (APPR_HEAD && state[ 2] > 1)
    {
      SNAKE_EAT
    }
    break;

  case CA0_F:
  case CA0_P:
  case CA0_W:
    // Дуги L8, L9, L10
    if (APPR_HEAD)
    {
      SNAKE_EAT
    }
    break;
  }
}
```

4.4. Построение графа переходов для супервизоров

На рис. 12 приведен граф переходов для супервизора A0 («состояние игры»).

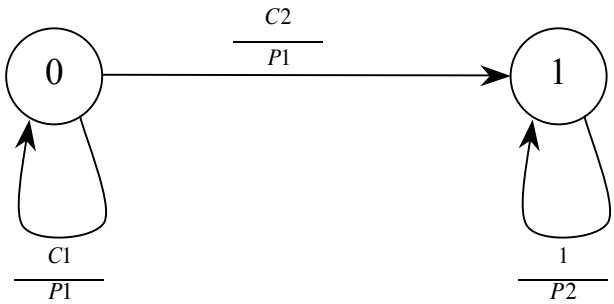


Рис. 12. Граф переходов автомата A0

Условие C1: Змейка не съела саму себя, яд или стену (игра продолжается).

Условие C2: Змейка съела саму себя, яд или стену (игра должна закончиться).

Для удобства реализации, условия C1 и C2 будем проверять уже в процессе выполнения действия P1 и таким обоазом определять, в какое именно состояние делается переход.

Действие P1: Возможно (с некоторой вероятностью), поместить на поле призовой элемент, а для $A_{i,j} : D^1_{i,j} == 0$ произвести шаг по значению свойства D^3 согласно следующему графу переходов (рис. 13).

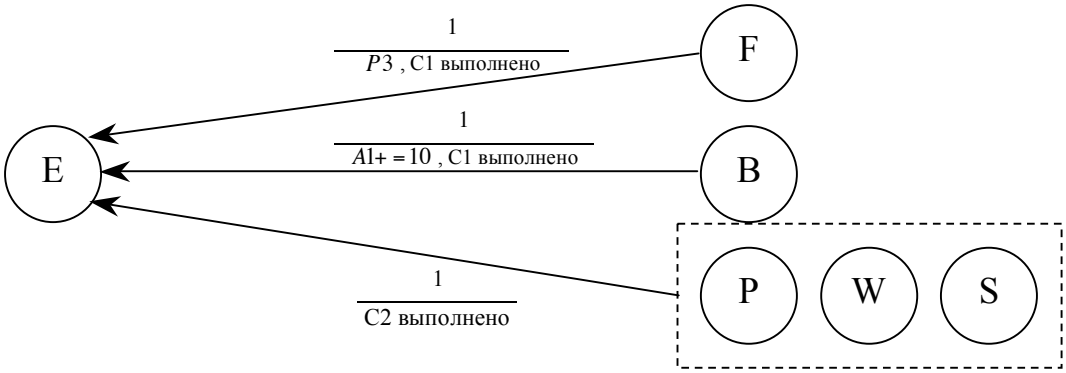


Рис. 13. Граф переходов по значению свойства D^3

Действие P3: $\forall i, j : A^0_{i,j} == Snake \Rightarrow D^2_{i,j} += 3$; поместить на поле новое яблоко и новый яд.

Действие P2: $\forall i, j : A^0_{i,j} = Wall$.

В коде языка C++ полный шаг системы (описанные выше переходы, а также обеспечение шага клеточного автомата), записываются следующим образом. Таблица соответствия математических обозначений обозначениям кода C++ была приведена в разд. 2.5 (табл. 1).

```
// Шаг клеточного автомата
void SnakeCellAut::Step()
{
  int i, j;
  // Создание «фотографии»
  memcpy(&snapshot, &field, sizeof(field));

  // Каждая клетка делает шаг
```

```
  for (i = 1; i <= FIELD_H; i++)
    for (j = 1; j <= FIELD_W; j++)
      field[ i][ j] .Step(
                          snapshot[ i][ j+1] .state,
                          snapshot[ i-1][ j] .state,
                          snapshot[ i][ j-1] .state,
                          snapshot[ i+1][ j] .state,
                          state);

  // Шаги супервизоров

  // Шаги автомата A0
  switch (state[ 0] )
  {
  case 0:
    P1();
    break;

  case 1:
    // Действие P2
    for (i = 1; i <= FIELD_H; i++)
      for (j = 1; j <= FIELD_W; j++)
        field[ i][ j] .state[ 0]  = CA0_W;
    break;
  }

}

// Действие P1
void SnakeCellAut::P1()
{
  // Вероятностная установка призового элемента
  if (rand()%100 == 0) // вероятностный переход
  {                    // поместить призовой элемент
    int k, l;
    k = rand()%FIELD_W + 1;
    l = rand()%FIELD_H + 1;

    if (field[ k][ l] .state[ 0]  == CA0_E)
    {
      field[ k][ l] .state[ 0]  = CA0_B;
      field[ k][ l] .state[ 2]  = 20;
    }
  }


  // Проверка содержимого желудка
  int i, j;
  int *head = 0;

  // Сначала найдем желудок
  // «желудок» -- свойство D^3 головы змейки,
  // то есть такой клетки, у которой (D^1 != 0),
  for (i = 1; i <= FIELD_H; i++)
    for (j = 1; j <= FIELD_W; j++)
      if (field[ i][ j] .state[ 1]  != 0) // это голова
        head = field[ i][ j] .state;
  // теперь head обозначает field[ i][ j] ,
  // где i,j -- координаты головы
```

```
  if (!head)
    return;

  // Переход по графу переходов
  switch (head[3])
  {
  case CA0_F:
    // Действие P3
    for (i = 1; i <= FIELD_H; i++)
      for (j = 1; j <= FIELD_W; j++)
        if (field[i][j].state[0] == CA0_S)
          field[i][j].state[2] += 3;

    state[1] += 1;

    DropItem(CA0_F);
    DropItem(CA0_P);

    head[0] = CA0_E;
    break;

  case CA0_B:
    state[1] += 10;
    head[0] = CA0_E;
    break;

  case CA0_P:
  case CA0_W:
  case CA0_S:
    state[0] = 1;
    head[0] = CA0_E;
    break;
  }

}

// Вспомогательная функция: выбросить на поле яблоко или яд
void SnakeCellAut::DropItem(int item)
{
  int i = 0, j = 0;

  while (!i || !j || field[i][j].state[0] != CA0_E)
  {
    i = rand()%FIELD_W + 1;
    j = rand()%FIELD_H + 1;
  }

  field[i][j].state[0] = item;
}
```

4.5. Интерфейс игры

Реализацией интерфейса игры является отрисовка игровой ситуации, обработка ввода пользователя, возможно, вывод звуковых эффектов и

музыки. При реализации этой игры актуальны отрисовка и обработка ввода. На рис. 14 изображен графический интерфейс игры.

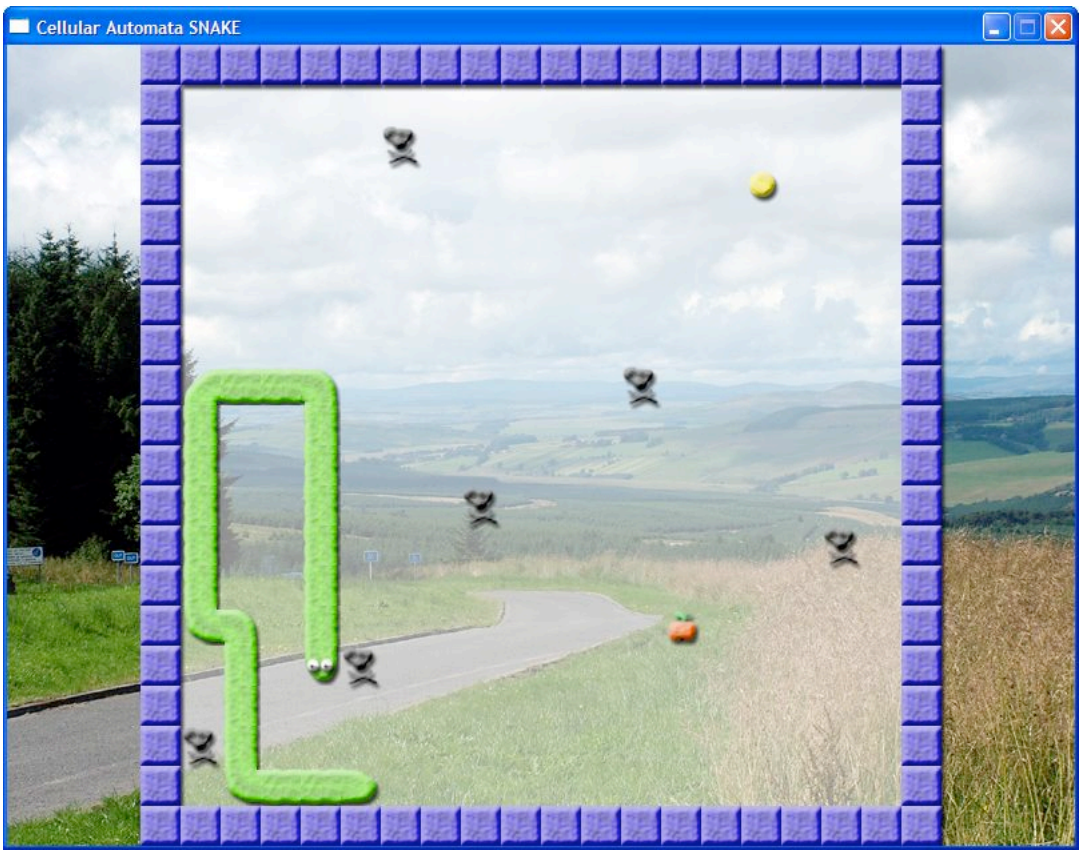


Рис. 14. Графический интерфейс игры «Змейка»

Отрисовка состояния игры состоит из отрисовки поля и отрисовки «общего интерфейса». Последнее никак не относится к логике игры, поэтому рассмотрим только методы отрисовки поля. Ее можно производить разными способами.

Самый простой способ — разделить область экрана, отведенную для отображения поля, на области, занимаемые каждой клеткой, и отображать в каждой из них некоторый спрайт (двумерное изображение). Какой именно спрайт отображается в данной клетке, определяется ее состоянием и свойствами.

Другой, более сложный — определять спрайт не только состоянием отображаемой клетки, но и состоянием и свойствами клеток в некоторой

окрестности (эта окрестность не обязательно должна совпадать с окрестностью, используемой в клеточном автомате). Этот способ и будем применять в игре «*Змейка*».

Если первые два способа имеют тот недостаток, что изменение картины поля происходит дискретно при шаге автомата, то следующий способ частично лишен этого недостатка. Если в игре имеются простые персонажи (которые, конечно, на самом деле задаются лишь состоянием некоторой клетки), которые движутся, их спрайты можно отображать не непосредственно на своем месте, а с некоторым смещением, величина которого непрерывно зависит от времени, прошедшего с предыдущего шага. Этот метод проиллюстрирован на рис. 15.

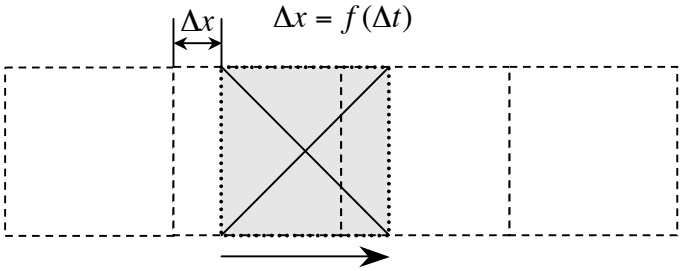


Рис. 15. Смещение спрайта относительно клетки

Для правильного отображения змейки со всеми ее изгибами и закругленными концами понадобятся 10 спрайтов (четыре возможных изгиба, четыре возможных конца, два прямых отрезка) и два спрайта с «глазами» для отметки головы.

На языке C++ отображение поля по второму способу можно реализовать следующим образом. Таблица соответствия математических обозначений обозначениям кода C++ была приведена в разд. 2.5 (табл. 1).

```
// Вспомогательный макрос: показывает,
// соседние ли два целых числа
#define NEIGH(A,B) (ABS((A)-(B)) <= 1)

// Вспомогательный макрос: показывает,
// находится ли в клетке с заданным смещением
// относительно данной соседняя часть змейки
```

```
#define SNAKE(I,J) ( \
  logic.field[ i+(I)][ j+(J)].state[ 0] == CA0_S && \
  NEIGH(logic.field[ i][ j].state[ 2],
logic.field[ i+(I)][ j+(J)].state[ 2]) \
  )

void SnakeGame::Draw()
{
  // Получаем время, прошедшее с предыдущего кадра
  double dt=double(hge->Timer_GetDelta());

  // Производим шаг логической системы
  Step(dt);

  // Нарисовать фон
  sprBG->Render(0, 0);

  // Для каждой клетки поля отображаем ее состояние
  for (int i = 1; i <= FIELD_H; i++)
    for (int j = 1; j <= FIELD_W; j++)
      if (logic.field[ i][ j].state[ 0] != CA0_S)
      {
        // Если клетка не пуста и не содержит змейку,
        // отображаем соответствующий объект
        if (logic.field[ i][ j].state[ 0] != CA0_E)
          sprBlock[ logic.field[ i][ j].state[ 0]]->Render(100+(j-
1)*30.0f, (i-1)*30.0f);
      }
      else
      {
        // Отображаем змейку
        // нужно выбрать, какой именно спрайт следует отобразить

        hgeSprite* spr = sprBlock[ 1]; // По умолчанию
        float delta_x = 0, delta_y = 0;

        // Проверяем все возможные конфигурации соседей,
        // чтобы выбрать нужный спрайт
        if (SNAKE(-1,0) && SNAKE(0,-1))
          spr = sprSnake[ 0];
        if (SNAKE(-1,0) && SNAKE(0,1))
          spr = sprSnake[ 1];
        if (SNAKE(1,0) && SNAKE(0,-1))
          spr = sprSnake[ 2];
        if (SNAKE(1,0) && SNAKE(0,1))
          spr = sprSnake[ 3];

        if (SNAKE(-1,0) && !SNAKE(1,0) && !SNAKE(0,-1) &&
!SNAKE(0,1))
          spr = sprSnake[ 4];
        if (SNAKE(1,0) && !SNAKE(-1,0) && !SNAKE(0,-1) &&
!SNAKE(0,1))
          spr = sprSnake[ 5];
        if (SNAKE(0,-1) && !SNAKE(1,0) && !SNAKE(-1,0) &&
!SNAKE(0,1))
          spr = sprSnake[ 6];
        if (SNAKE(0,1) && !SNAKE(1,0) && !SNAKE(-1,0) &&
!SNAKE(0,-1))
          spr = sprSnake[ 7];
```

```
         if (SNAKE(-1,0) && SNAKE(1,0) && !SNAKE(0,-1) &&
!SNAKE(0,1))
           spr = sprSnake[8];
         if (!SNAKE(-1,0) && !SNAKE(1,0) && SNAKE(0,-1) &&
SNAKE(0,1))
           spr = sprSnake[9];

         // Отображаем выбранный спрайт
         spr->Render(100+(j-1 + delta_x)*30.0f, (i-1 +
delta_y)*30.0f);

         // Если в текущей клетке голова,
         // следует нарисовать глаза змейки:
         // вертикально или горизонтально в зависимости
         // от ориентаци головы
         if (logic.field[i][j].state[1] != 0) // это голова
         {
           if (SNAKE(-1,0) || SNAKE(1,0))
             sprSnake[10]->Render(100+(j-1 + delta_x)*30.0f, (i-1 +
delta_y)*30.0f);
           else
             sprSnake[11]->Render(100+(j-1 + delta_x)*30.0f, (i-1 +
delta_y)*30.0f);
         }
       }
}
```

Данный фрагмент кода нуждается в некоторых пояснениях:

`sprSnake` — массив спрайтов, используемых для отображения змейки;

`logic` — объект, инкапсулирующий систему, описанную в данной работе, в частности `logic.field` — двумерный массив состояний клеток;

`hge` — указатель на объект, инкапсулирующий систему вывода графики[6].

Обработка ввода пользователя в данном случае сводится к реакции на клавиши-стрелки. Клавиши изменяют направление движения головы. Если нажата клавиша, соответствующая текущему направлению, действия не производятся. Если направление обратно, также ничего не

[6] В примере используется графический движок *HGE* компании *Relish Games*.

происходит. Если клавиша соответствует направлению, перпендикулярному текущему, меняется направление головы. На языке C++ фрагмент кода, отвечающий за обработку клавиш, будет выглядеть следующим образом.

```
void SnakeCellAut::KeyPressed(int key)
{
  int i, j;
  int *head;

  // Найдем голову змейки
  for (i = 1; i <= FIELD_H; i++)
    for (j = 1; j <= FIELD_W; j++)
      if (field[ i][ j] .state[ 1]  != 0)
        head = field[ i][ j] .state;

  switch (key)
  {
  case HGEK_RIGHT:
    if (head[ 1]  != 3)
      head[ 1]  = 1;
    break;
  case HGEK_UP:
    if (head[ 1]  != 4)
      head[ 1]  = 2;
    break;
  case HGEK_LEFT:
    if (head[ 1]  != 1)
      head[ 1]  = 3;
    break;
  case HGEK_DOWN:
    if (head[ 1]  != 2)
      head[ 1]  = 4;
    break;
  }

}
```

Полный исходный текст программы «*Змейка*» приведен в Приложении 1.

4.6. Сравнение с другими подходами

В работе [9] рассмотрена реализация игры «*Змейка*» на основе *Switch-технологии* без использования клеточных автоматов. Реализуется игра с аналогичными, но более простыми правилами: нет «призовых

элементов», нет «яда», при съедании яблока длина змейки увеличивается на один, а не на три элемента.

При этом состояние клеток поля не хранится в виде матрицы, а змейка представляет собой очередь (FIFO) из пар координат ее элементов.

Для управления змейкой вводится шесть конечных автоматов, а исходный текст на языке *Java*[7] размещен на 108 страницах, содержит около 3000 строк и имеет объем около 70Кб.

Исходный текст реализации игры с использованием клеточного автомата, на основе подхода, описанного в данной работе, на языке C++ вместе с заголовочными файлами, содержит около 550 строк и имеет объем около 10Кб.

При этом реализация с использованием клеточного автомата реализует игру с более сложными правилами, которые могут без существенного усложнения графов переходов быть расширенными следующим образом:

- после ухода змейки с клетки, где было яблоко, на этом месте остается «мусор» в виде элемента «яд» (потребовало бы добавления одной дуги в граф переходов A^0);
- на поле может быть выстроена произвольная конфигурация стен, причем конфигурация может меняться динамически в процессе игры (не требует изменения графов переходов);
- на поле может единовременно выставляться произвольное количество яблок (не требует изменения графов переходов);

[7] Выразительная сила языков *Java* и C++ примерно одинаковы, поэтому сравнение количества строк и символов в исходных текстах правомерно.

- на поле могут фигурировать несколько «змеек» (потребовало бы добавления еще одного свойства клетки поля и изменения функции отрисовки).

Это сравнение не говорит о несовершенстве *Switch-технологии* или превосходстве подхода, основанного на применении клеточных автоматов в общем случае. В игре «*Змейка*» преобладают локальные взаимодействия, и поэтому применение *Switch-технологии* вместе с клеточным автоматом дает хорошие результаты. В то же время, при реализации, например, игры *Tetris* применение исключительно *Switch-технологии* наверняка дало бы гораздо лучший результат, по причинам, изложенным ранее.

Выводы

1. Любые игры, базирующиеся на тетрагональном поле и имеющие локальные взаимодействия, могут быть реализованы с использованием клеточных автоматов.
2. Некоторые игры на тетрагональном поле, в которых локальные взаимодействия преобладают, при использовании клеточных автоматов могут быть реализованы более эффективно, чем на основе объектно-ориентированного подхода.
3. Реализация компьютерной игры «*Змейка*» с использованием клеточных автоматов по сравнению с ее реализацией [9] на основе *Switch-технологии* проще (два конечных автомата вместо шести), компактнее (объем кода в 7 раз меньше) и удобнее для расширения (существенные усложнения игры могут быть достигнуты при минимальном изменении вычислительного алгоритма).

Источники

1. *Шалыто А.А.* Новая инициатива в программировании — «Движение за открытую проектную документацию» //Мир ПК 2003. № 9, с. 52—56. http://is.ifmo.ru
2. *Тоффоли Т., Марголус Н.* Машины клеточных автоматов. М.: Мир, 1991.
3. *Фон Нейман Дж.* Теория самовоспроизводящихся автоматов. М.: Мир, 1971.
4. *Шалыто А.*, *Туккель Н.* От тьюрингова программирования к автоматному //Мир ПК. 2002, № 2, с. 144—149. http://is.ifmo.ru
5. *Наумов Л.А.*, *Шалыто А.А.* Клеточные автоматы. Реализация и эксперименты //Мир ПК. 2003. № 8, с. 64—71. http://is.ifmo.ru
6. *Esser J.*, *Schrechenberg M.* Microscopic simulation of urban traffic based on cellular automata //International Journal of Modern Physics, Vol. 8, No. 5 (1997) p. 1025—1036.
7. *Шалыто А.А.* Switch-технология. Алгоритмизация и программирование задач логического управления. СПб.: Наука, 1998.
8. *Шалыто А.А.* Технология автоматного программирования //Мир ПК 2003 № 10, с. 74—78. http://is.ifmo.ru
9. *Сапунков В.С., Шалыто А.А.* Система управления игрой «Змейка» //http://is.ifmo.ru/projects/snake

Приложение 1. Исходный текст игры «Змейка»

Prefix.h

```
#pragma once

#include <stdlib.h>
#include <time.h>
#include <string.h>

#include <hge.h>
#include <hgesprite.h>
#include <hgefont.h>
#include <hgeparticle.h>

#include "Snake.h"
#include "Cellaut.h"
#include "Game.h"
```

Game.h

```
#include "Prefix.h"

#define STEP_LENGTH 0.3

class SnakeGame {
private:
  HGE* hge;

  HTEXTURE bg;
  hgeSprite* sprBG;
  hgeSprite* sprBlock[ 6] ;
  hgeSprite* sprSnake[ 12] ;

  double t_since_last_step;

  SnakeCellAut logic;

  void LoadResources();

public:
  int quitting;

  SnakeGame(HGE* n_hge);
  ~SnakeGame();

  void Step(double dt);
  void Draw();
};
```

Cellaut.h

```
#include "Cell.h"

#define A_N    2
```

```
#define FIELD_W20
#define FIELD_H 20

typedef SnakeCell SnakeField[ FIELD_H+2][ FIELD_H+2];

class SnakeCellAut {
private:
  void CheckStomach();
  void DropItem(int item);

public:
  SnakeField field;
  SnakeField snapshot;

  int state[ A_N];
  static int default_state[ A_N];

  SnakeCellAut();
  void Step();
  void P1();
  void P2();

  void KeyPressed(int key);
};
```

Cell.h

```
#define CA0_E  0
#define CA0_S  1
#define CA0_F  2
#define CA0_P  3
#define CA0_W  4
#define CA0_B  5

#define CA_N 4

class SnakeCell {
public:
  int state[ CA_N]; // [ 0]  - A0 (type),
                    // [ 1]  - A1 (direction),
                    // [ 2]  - A2 (TTL),
                    // [ 3]  - A3 (stomach)
  static int border_state[ CA_N];
  static int default_state[ CA_N];

  SnakeCell();
  void Step(const int* right, const int* up, const int* left, const
int* down, const int* super);
};
```

Snake.cpp

```
#include "Prefix.h"

HGE* hge = 0;
SnakeGame* game = 0;
```

```
bool FrameFunc()
{
  if (hge->Input_GetKeyState(HGEK_ESCAPE)) return true;

  hge->Gfx_BeginScene();
  hge->Gfx_Clear(0);

  if (game)
    game->Draw();

  hge->Gfx_EndScene();

  // Continue execution
  return game->quitting == 1;
}

int WINAPI WinMain(HINSTANCE, HINSTANCE, LPSTR, int)
{
  time_t timev;
  time(&timev);
  srand(timev);

  hge = hgeCreate(HGE_VERSION);

  hge->System_SetState(HGE_FRAMEFUNC, FrameFunc);

  hge->System_SetState(HGE_TITLE, "Cellular Automata SNAKE");

  hge->System_SetState(HGE_WINDOWED, true);

  hge->System_SetState(HGE_USESOUND, false);

  if(hge->System_Initiate())
  {
    game = new SnakeGame(hge);
    hge->System_Start();
  }
  else
  {
    MessageBox(NULL, hge->System_GetErrorMessage(), "Error", MB_OK
| MB_ICONERROR | MB_SYSTEMMODAL);
  }

  delete game;

  hge->System_Shutdown();

  hge->Release();

  return 0;
}
```

Game.cpp

```
#include "Prefix.h"

void SnakeGame::LoadResources()
{
  bg = hge->Texture_Load("tex.png");
```

```
  if (!bg)
  {
    quitting = 1;
    return;
  }

  // Инициализация спрайтов
  sprBG = new hgeSprite(bg, 0, 0, 800, 600);
  int i;
  for (i = 0; i < 6; i++)
    sprBlock[i] = new hgeSprite(bg, i*40.0f, 600, 40, 40);

  for (i = 0; i < 12; i++)
    sprSnake[i] = new hgeSprite(bg, i*40.0f, 640, 40, 40);
}

SnakeGame::SnakeGame(HGE *n_hge)
{
  hge = n_hge;
  LoadResources();
  quitting = 0;

  t_since_last_step = 0;
}

SnakeGame::~SnakeGame()
{
  hge->Texture_Free(bg);

  int i;
  for (i = 0; i < 6; i++)
    delete sprBlock[i];

  for (i = 0; i < 12; i++)
    delete sprSnake[i];

}

#define ABS(A) ((A)>0?(A):-(A))

// Вспомогательный макрос: показывает,
// соседние ли два целых числа
#define NEIGH(A,B) (ABS((A)-(B)) <= 1)

// Вспомогательный макрос: показывает,
// находится ли в клетке с заданным смещением
// относительно данной соседняя часть змейки
#define SNAKE(I,J) ( \
  logic.field[i+(I)][j+(J)].state[0] == CA0_S && \
  NEIGH(logic.field[i][j].state[2],
logic.field[i+(I)][j+(J)].state[2]) \
  )

void SnakeGame::Draw()
{
  // Получаем время, прошедшее с предыдущего кадра
  double dt=double(hge->Timer_GetDelta());

  // Производим шаг логической системы
```

```
  Step(dt);

  // Нарисовать фон
  sprBG->Render(0, 0);

  // Для каждой клетки поля отображаем ее состояние
  for (int i = 1; i <= FIELD_H; i++)
    for (int j = 1; j <= FIELD_W; j++)
      if (logic.field[ i][ j] .state[ 0]  != CA0_S)
      {
        // Если клетка не пуста и не содержит змейку,
        // отображаем соответствующий объект
        if (logic.field[ i][ j] .state[ 0]  != CA0_E)
          sprBlock[ logic.field[ i][ j] .state[ 0]] ->
            Render(100+(j-1)*30.0f, (i-1)*30.0f);
      }
      else
      {
        // Отображаем змейку
        // нужно выбрать, какой именно спрайт следует отобразить

        hgeSprite* spr = sprBlock[ 1]; // По умолчанию
        float delta_x = 0, delta_y = 0;

        // Проверяем все возможные конфигурации соседей,
        // чтобы выбрать нужный спрайт
        if (SNAKE(-1,0) && SNAKE(0,-1))
          spr = sprSnake[ 0];
        if (SNAKE(-1,0) && SNAKE(0,1))
          spr = sprSnake[ 1];
        if (SNAKE(1,0) && SNAKE(0,-1))
          spr = sprSnake[ 2];
        if (SNAKE(1,0) && SNAKE(0,1))
          spr = sprSnake[ 3];

        if (SNAKE(-1,0) && !SNAKE(1,0) && !SNAKE(0,-1) &&
!SNAKE(0,1))
          spr = sprSnake[ 4];
        if (SNAKE(1,0) && !SNAKE(-1,0) && !SNAKE(0,-1) &&
!SNAKE(0,1))
          spr = sprSnake[ 5];
        if (SNAKE(0,-1) && !SNAKE(1,0) && !SNAKE(-1,0) &&
!SNAKE(0,1))
          spr = sprSnake[ 6];
        if (SNAKE(0,1) && !SNAKE(1,0) && !SNAKE(-1,0) &&
!SNAKE(0,-1))
          spr = sprSnake[ 7];

        if (SNAKE(-1,0) && SNAKE(1,0) && !SNAKE(0,-1) &&
!SNAKE(0,1))
          spr = sprSnake[ 8];
        if (!SNAKE(-1,0) && !SNAKE(1,0) && SNAKE(0,-1) &&
SNAKE(0,1))
          spr = sprSnake[ 9];

        // Отображаем выбранный спрайт
        spr->Render(100+(j-1 + delta_x)*30.0f, (i-1 +
delta_y)*30.0f);

        // Если в текущей клетке голова,
```

```
          // следует нарисовать глаза змейки:
          // вертикально или горизонтально в зависимости
          // от ориентаци головы
          if (logic.field[ i][ j] .state[ 1]  != 0) // это голова
          {
            if (SNAKE(-1,0) || SNAKE(1,0))
              sprSnake[ 10] ->Render(100+(j-1 + delta_x)*30.0f, (i-1 +
delta_y)*30.0f);
            else
              sprSnake[ 11] ->Render(100+(j-1 + delta_x)*30.0f, (i-1 +
delta_y)*30.0f);
          }
        }
}


void SnakeGame::Step(double dt)
{
  t_since_last_step += dt;

  if (t_since_last_step > STEP_LENGTH)
  {
    t_since_last_step -= STEP_LENGTH;

    logic.Step();
  }

  int key = hge->Input_GetKey();
  switch (key)
  {
  case HGEK_LEFT:
  case HGEK_RIGHT:
  case HGEK_UP:
  case HGEK_DOWN:
    logic.KeyPressed(key);
    break;
  }
}
```

Cellaut.cpp

```
#include "Prefix.h"

int SnakeCellAut::default_state[ A_N]  = { 0, 0} ;

SnakeCellAut::SnakeCellAut()
{
  int i, j;

  for (i = 0; i <= FIELD_H; i++)
    for (j = 1; j <= FIELD_W; j++)
    {
      for (int k = 0; k < CA_N; k++)
        field[ i][ j] .state[ k]  = field[ i][ j] .default_state[ k] ;
    }

  for (i = 0; i <= FIELD_H; i++)
  {
    field[ i][ 1] .state[ 0]  = CA0_W;
```

```
    field[ i][ FIELD_W] .state[ 0]  = CA0_W;
  }
  for (j = 0; j <= FIELD_W; j++)
  {
    field[ 1][ j] .state[ 0]  = CA0_W;
    field[ FIELD_H][ j] .state[ 0]  = CA0_W;
  }

  for (int k = 0; k < A_N; k++)
    state[ k]  = default_state[ k] ;

  // Create the snake
  field[ 5][ 5] .state[ 0]  = CA0_S;
  field[ 5][ 5] .state[ 1]  = 1;
  field[ 5][ 5] .state[ 2]  = 5;

  DropItem(CA0_F);
}

// Шаг клеточного автомата
void SnakeCellAut::Step()
{
  int i, j;
  // Создание "фотографии"
  memcpy(&snapshot, &field, sizeof(field));

  // Каждая клетка делает шаг
  for (i = 1; i <= FIELD_H; i++)
    for (j = 1; j <= FIELD_W; j++)
      field[ i][ j] .Step(
                          snapshot[ i][ j+1] .state,
                          snapshot[ i-1][ j] .state,
                          snapshot[ i][ j-1] .state,
                          snapshot[ i+1][ j] .state,
                          state);

  // Шаги супервизоров

  // Шаги автомата A0
  switch (state[ 0] )
  {
  case 0:
    P1();
    break;

  case 1:
    // Действие P2
    for (i = 1; i <= FIELD_H; i++)
      for (j = 1; j <= FIELD_W; j++)
        field[ i][ j] .state[ 0]  = CA0_W;
    break;
  }

}

// Действие P1
void SnakeCellAut::P1()
{
  // Вероятностная установка призового элемента
  if (rand()%100 == 0) // вероятностный переход
```

```
{                       // поместить призовой элемент
  int k, l;
  k = rand()%FIELD_W + 1;
  l = rand()%FIELD_H + 1;

  if (field[ k][ l].state[ 0] == CA0_E)
  {
    field[ k][ l].state[ 0] = CA0_B;
    field[ k][ l].state[ 2] = 20;
  }
}


// Проверка содержимого желудка
int i, j;
int *head = 0;

// Сначала найдем желудок
// "желудок" -- свойство D^3 головы змейки,
// то есть такой клетки, у которой (D^1 != 0),
for (i = 1; i <= FIELD_H; i++)
  for (j = 1; j <= FIELD_W; j++)
    if (field[ i][ j].state[ 1] != 0) // это голова
      head = field[ i][ j].state;
// теперь head обозначает field[ i][ j],
// где i,j -- координаты головы

if (!head)
  return;

// Переход по графу переходов
switch (head[ 3])
{
case CA0_F:
  // Действие P3
  for (i = 1; i <= FIELD_H; i++)
    for (j = 1; j <= FIELD_W; j++)
      if (field[ i][ j].state[ 0] == CA0_S)
        field[ i][ j].state[ 2] += 3;

  state[ 1] += 1;

  DropItem(CA0_F);
  DropItem(CA0_P);

  head[ 3] = CA0_E;
  break;

case CA0_B:
  state[ 1] += 10;
  head[ 3] = CA0_E;
  break;

case CA0_P:
case CA0_W:
case CA0_S:
  state[ 0] = 1;
  head[ 3] = CA0_E;
  break;
}
```

```
}


// Вспомогательная функция: выбросить на поле яблоко или яд
void SnakeCellAut::DropItem(int item)
{
  int i = 0, j = 0;

  while (!i || !j || field[i][j].state[0] != CA0_E)
  {
    i = rand()%FIELD_W + 1;
    j = rand()%FIELD_H + 1;
  }

  field[i][j].state[0] = item;
}

void SnakeCellAut::KeyPressed(int key)
{
  int i, j;
  int *head;

  // Найдем голову змейки
  for (i = 1; i <= FIELD_H; i++)
    for (j = 1; j <= FIELD_W; j++)
      if (field[i][j].state[1] != 0)
        head = field[i][j].state;

  switch (key)
  {
  case HGEK_RIGHT:
    if (head[1] != 3)
      head[1] = 1;
    break;
  case HGEK_UP:
    if (head[1] != 4)
      head[1] = 2;
    break;
  case HGEK_LEFT:
    if (head[1] != 1)
      head[1] = 3;
    break;
  case HGEK_DOWN:
    if (head[1] != 2)
      head[1] = 4;
    break;
  }

}
```

Cell.cpp

```
#include "Prefix.h"

int SnakeCell::border_state[CA_N] = {CA0_W, 0, 0, CA0_E};
int SnakeCell::default_state[CA_N] = {CA0_E, 0, 0, CA0_E};

SnakeCell::SnakeCell()
{
```

```
  for (int i = 0; i < CA_N; i++)
    state[ i] = border_state[ i] ;
}
#define APPR_HEAD \
  ( \
  (left[ 0] == CA0_S && left[ 1] == 1)?left: \
  (down[ 0] == CA0_S && down[ 1] == 2)?down: \
  (right[ 0] == CA0_S && right[ 1] == 3)?right: \
  (up[ 0] == CA0_S && up[ 1] == 4)?up: \
  0 \
  )

#define SNAKE_EAT \
  state[ 1] = APPR_HEAD[ 1] ; \
  state[ 2] = APPR_HEAD[ 2] ; \
  state[ 3] = state[ 0] ; \
  state[ 0] = CA0_S;


void SnakeCell::Step(const int* right, const int* up, const int*
left, const int* down, const int* super)
{
  switch (state[ 0] )
  {
  case CA0_S:
    // Дуга L1
    if (state[ 2] > 1 && !APPR_HEAD)
    {
      state[ 1] = 0;
      state[ 2] -= 1;
    }

    // Дуга L2
    if (state[ 2] == 1 && !APPR_HEAD)
    {
      state[ 1] = 0;
      state[ 2] = 0;
      state[ 0] = CA0_E;
    }

    // Дуга L5
    if (APPR_HEAD)
    {
      SNAKE_EAT
    }
    break;

  case CA0_E:
    // Дуга L6
    if (APPR_HEAD)
    {
      SNAKE_EAT
    }
    break;

  case CA0_B:
    // Дуга L3
    if (state[ 2] > 1 && !APPR_HEAD)
      state[ 2] -= 1;
```

```
      // Дуга L4
      if (state[ 2] <= 1)
        state[ 0] = CA0_E;

      // Дуга L7
      if (APPR_HEAD && state[ 2] > 1)
      {
        SNAKE_EAT
      }
      break;

    case CA0_F:
    case CA0_P:
    case CA0_W:
      // Дуги L8, L9, L10
      if (APPR_HEAD)
      {
        SNAKE_EAT
      }
      break;
    }
}
```