ЗАДАНИЕ 12
Игра «Жизнь»

Построение модели • Выполним реализацию игры «Жизнь» с поддержкой «рисования» объектов прямо в пространстве модели с использованием мыши.

A Создаем новую модель. Устанавливаем размеры max-pxcor и max-pycor равными 25, размер патча — 10 пикселям.

B При визуализации автомата наша модель будет поддерживать режим трассировки, показывая предыдущее состояние клетки. Для этого нам потребуется три цвета: один цвет для живых клеток (текущее состояние 1), второй — для мертвых клеток, которые были живы на предыдущем шаге, и третий цвет — просто для мертвых клеток. Добавляем к интерфейсу три элемента ввода с типом Color и обозначаем их color-1, color-old-1 и color-0. Настраиваем цвета.

C Включим в модель два режима инициализации: случайное заполнение живыми клетками с заданной плотностью и полная очистка (живых клеток нет). Для этого будем использовать одну и ту же процедуру `setup` с параметром, указывающим режим инициализации. Добавим к интерфейсу две кнопки: в настройках первой указываем команду `setup "random"`, имя кнопки random; в настройках второй используем команду `setup "clear"` и имя кнопки clear. Также добавляем слайдер density с настройками по умолчанию, соответствующий начальной плотности живых клеток.

D Переходим во вкладку с кодом модели. Представлять клетки автомата будем патчами. Но для

визуализации клеток мы будем использовать черепах[1]. Приписываем патчам атрибуты: `state` — текущее состояние клетки, `old-state` — состояние клетки на предыдущем шаге, `l-n` — число живых соседей клетки, `cell` — черепаха, связанная с данным патчем (с данной клеткой автомата).

[1] У черепах больше визуальных атрибутов, чем у патчей.

E Создаем процедуру `setup` с аргументом `mode`, в которой очищаем модель, просим все патчи выполнить свою настройку с помощью команды

```
setup-patch mode
```

сбрасываем таймер.

F Пишем код процедуры `setup-patch` с аргументом `mode`, в котором выполняем следующие действия. Устанавливаем цвет патча равным `color-0`. Устанавливаем нулевое значение старого состояния `old-state`. Установливаем текущее состояние клетки `state` согласно выбранному режиму инициализации: состояние равно 1, если `mode = "random"` и случайное целое число от 0 до 100 меньше значения `density`. Для этого оказывается удобным создать отдельную функцию `get-state [x]`, которая преобразует свой логический аргумент `x` в число: `false` в 0, `true` в 1. Создаем в центре патча черепаху (команда `sprout 1`) круглой формы размера 0.8. Запоминаем созданную черепаху в атрибуте патча `cell`:

```
set cell one-of turtles-here
```

Последней командой вызываем процедуру изменения цвета клетки `recolor`.

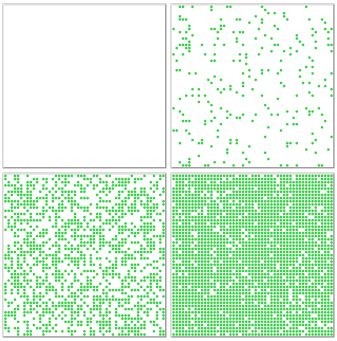

РИС. 12.1 Инициализация модели, слева вверху режим `"clear"`, в остальных случаях режим `"random"` для разных значений параметра `density`

G Пишем код процедуры `recolor`, в которой устанавливаем цвет черепахи `cell`, привязанной к данному патчу, согласно следующей схеме. Если текущее состояние патча равно 1, то используем цвет `color-1`. Если текущее состояние нулевое, а предыдущее (`old-state`) было единичным, то используем цвет `color-old-1`. В противном случае используем цвет `color-0`.

H Проверяем работу кнопок random и clear. Примеры начальных конфигураций модели приведены на рис. 12.1.

I Добавляем две кнопки, привязанные к процедуре `go`, одна стандартная, во второй указываем имя

step и не ставим галочку автонажатия (Forever). Т.е. нажатие кнопки step будет приводить к однократному выполнению процедуры go.

J Пишем код процедуры go. Т.к. игра «Жизнь» функционирует в синхронном режиме, то изменение состояний патчей в нашей модели должно проводиться в два шага. На первом шаге просим все патчи вычислить сумму состояний их соседей, которая как раз равна числу живых соседей:

```
set l-n sum [state] of neighbors.
```

На втором шаге применяем сами правила. Для этого сначала просим все патчи сохранить текущее состояние в атрибуте old-state. Затем вычисляем новое состояние согласно правилам игры «Жизнь», которые можно сформулировать одним логическим условием: новое состояние равно 1, если число живых соседей равно 3 или число живых соседей равно 2 и текущее состояние равно 0:

```
1 let c l-n = 3 or (l-n = 2 and state = 1)
2 set state get-state c
```

После обновления состояния меняем цвет клетки командой recolor.

K Проверяем работу модели, которая должна демонстрировать типичное для игры «Жизнь» поведение — активная хаотическая деятельность вначале, которая затем локализуется и, возможно, совсем исчезает. При этом иногда можно увидеть и простейшие движущиеся объекты — глайдеры[2], которые, впрочем, быстро сталкиваются с другими объектами и обычно уничтожаются (рис. 12.2).

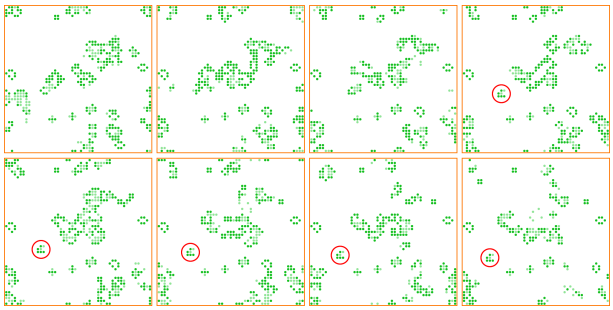

РИС. 12.2 Пример работы модели, красным цветом выделен движущийся глайдер

[2] Более сложные движущиеся объекты получить из начального случайного «бульона» практически не реально.

L Стандартная библиотека команд системы NetLogo частично поддерживает работу с клавиатурой[3] и мышью. Воспользуемся этой поддержкой, для того чтобы рисовать нужные нам начальные конфигурации (например, те же глайдеры), вместо того чтобы ждать, когда они возникнут сами собой из начального случайного распределения. Для этого сначала добавим к интерфейсу кнопку draw с включенным режимом автонажатия Forever. Пока эта кнопка будет нажата, мы сможем рисовать прямо в пространстве модели.

[3] Идея работы с клавиатурой следующая: создаем кнопку с привязанной к ней командой, этой кнопке назначаем ключ (key) — клавиша, нажатие которой вызывает нажатие соотвествующей кнопки. Т.е. получаем искомую связку: клавиша — команда.

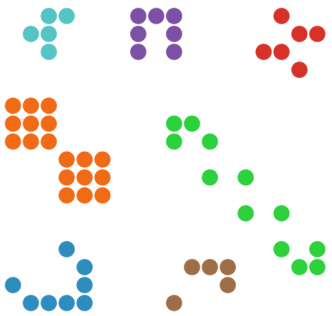

РИС. 12.3 Начальные конфигурации для тестирования модели

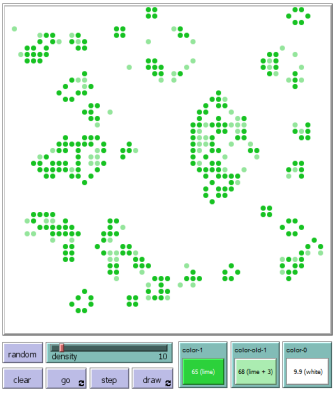

РИС. 12.4 Окончательный интерфейс модели

M Создаем процедуру draw, модель действия которой следующая: пользователь нажимает кнопку мыши и с зажатой кнопкой выделяет какие-то патчи. Как только он отпускает кнопку, процедура завершается (чтобы автоматически вызваться снова). Что делать с выделенными пользователем патчам, мы будем решать по первому патчу. Если он был в состоянии 1, то пользователь хочет его перевести в состояние 0, соответственно, и все остальные патчи в данном действии также будут переводиться нами в состояние 0. Если же первый выделенный патч был нулевой, то будем переводить все патчи в текущем действии в единичное состояние. Реализуется этот подход следующим образом:

```
1  to draw
2    let p patch mouse-xcor mouse-ycor
3    let erase? [state = 1] of p
4    while [mouse-down?] [
5      ask patch mouse-xcor mouse-ycor [
6        set state get-state not erase?
7        recolor
8      ]
9      display
10   ]
11 end
```

В первой строке определяем патч p, на который указывает мышь. Второй строкой задаем режим рисования (стирание — перевод 1 в 0), исходя из состояния патча p. Далее организуем цикл, который работает до тех пор, пока пользователь держит кнопку мыши нажатой. Каждый патч, на который указывает при этом мышь, переводим в новое состояние согласно режиму рисования, меняем цвет соответствующей клетки автомата. На каждой итерации цикла вызываем команду display для при-

нудительной перерисовки модели (иначе пользователь не увидит результата рисования, пока не нажмет кнопку go или step).

N Проверяем работу кнопки draw. Для ее тестирования можно использовать, например, глайдеры, которые можно рисовать в разной ориентации, чтобы они двигались в разные стороны. Пробуем также и другие простые объекты, показанные, например, на рис.12.3.

O Окончательный интерфейс модели приведен на рис. 12.4.

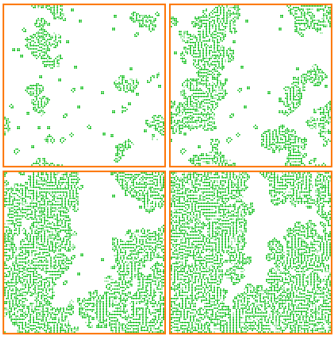

РИС. 12.5 Асинхронная версия игры «Жизнь»

УПРАЖНЕНИЯ

1 Модифицируйте описанную в главе модель, чтобы она работала в *асинхронном* режиме. Для этого надо обновлять состояние клетки *сразу* после вычисления числа живых ее соседей в одной команде `ask`, а не в двух отдельных командах. Пример работы асинхронной версии игры «Жизнь» показан на рис. 12.5.

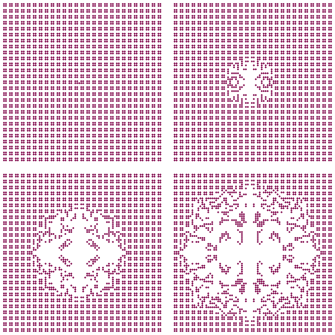

РИС. 12.6 Разрушение агара

2 Включите в модель инициализацию автомата различными регулярными структурами (так называемым *агаром*), например блоками 2×2. Если внести в такую устойчивую конструкцию одну дополнительную живую клетку, то это может привести к разрушению всей исходной конструкции (рис. 12.6).

3 Рисовать при каждом запуске модели начальную конфигурацию по одной клетке не очень удобно. Включите в модель поддержку вставки сразу целых объектов (например, глайдеров) в указанное пользователем место. Тип объекта и его ориентацию пользователь может выбрать в двух выпадающих списках.

4 Добавьте в модель выбор типа локальной окрестности — Мура или фон Неймана. Окрестность фон Неймана реализуется в NetLogo командой `neighbors4`. На рис. 12.7 показано стационарное состояние модели с правилами игры «Жизнь» для окрестности фон Неймана. Исследуйте поведение такого автомата.

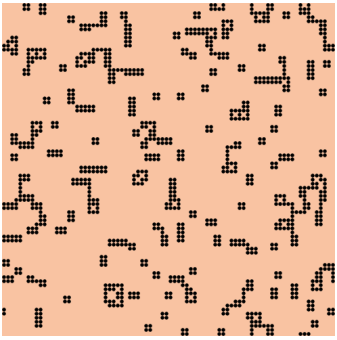

РИС. 12.7 Игра «Жизнь» с окрестностью фон Неймана

5 Обобщите модель на правила произвольного типа (тоталистические). Для этого добавьте к интерфейсу поле ввода rule с типом String, в котором пользователь должен указать логическое условие (предикат) того, что новое состояние клетки будет равно 1. Например, для игры «Жизнь» это условие записывается как

```
l-n = 3 or (l-n = 2 and state = 1).
```

После этого замените строку (J:1) процедуры `go` на команду `let c runresult rule`. Протестируйте модель на разных правилах: у клетки есть живые соседи, есть ровно 1 живой сосед, число соседей нечетно[4]. Результат работы последнего правила с окрестностью фон Неймана показан на рис. 12.8.

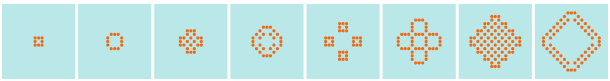

РИС. 12.8 Работа автомата с правилом `l-n mod 2 = 1`

6 Увеличение ранга окрестности существенно увеличивает выразительные возможности клеточных автоматов. Например, на рис. 12.9 приведены примеры работы двоичного клеточного автомата, в котором используются две локальные окрестности Мура рангов $r_a < r_i$, а единственное правило имеет вид: состояние клетки становится равным 1, если $\rho_a > k\rho_i$, где $\rho_{a,i}$ — концентрации живых клеток в соответствующих окрестностях, k — параметр. Неформально это правило работает следующим образом: если внутри меньшей окрестности концентрация единичных клеток больше, чем внутри большей окрестности, то клетка переходит в состояние 1 (активируется ближайшим окружением), иначе клетка переходит в состояние 0 (тормозится дальним окружением). Возникающие при работе автоматов данного типа структуры называют паттернами Тьюринга[5].

7 Другим способом расширения возможностей клеточных автоматов является увеличение числа состояний клеток. Реализуйте в NetLogo автомат с числом состояний m, моделирующий возбудимую среду. Правила этого автомата звучат следующим образом: если клетка находится в состоянии 0 (состояние покоя), то она переходит в состояние 1 (состояние возбуждения), если число возбужденных ее соседей превышает заданный порог. Клетка в состоянии 1 переходит в состояние 2, из состояния 2 в состояние 3 и т.д. Из последнего состояния клетка возвращается в нулевое состояние. Все состояния, большие 1, называются рефрактерными. Пример работы такого автомата приведен на рис. 12.10 (состояние возбуждения показано красным цветом, рефрактерные состояния — оттенками оранжевого цвета).

8 Реализуйте клеточный автомат с окрестностью фон Неймана, с m состояниями и с *циклическим* правилом:

[4] Это аналог правила 90 элементарного клеточного автомата.

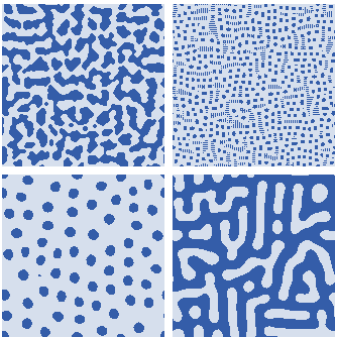

РИС. 12.9 Паттерны Тьюринга

[5] H. Sayama, *Introduction to the Modeling and Analysis of Complex Systems*, Open SUNY Textbooks, 2015, глава 11.

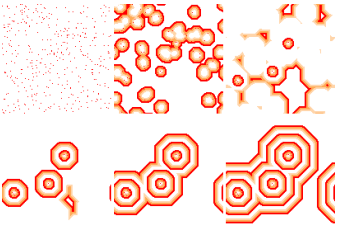

РИС. 12.10 Модель возбудимой среды

если у клетки в состоянии k есть хотя бы один сосед в *следующем* состоянии $(k+1) \mod m$, то данная клетка тоже переходит в это состояние. Пример работы такого автомата, который можно рассматривать в качестве простейшей модели автоколебательной химической реакции Белоусова-Жаботинского[6] для случая $m = 15$, показан на рис. 12.11.

9 Муравей Лэнгтона[7] — это двумерный клеточный автомат с 8 состояниями и простыми правилами, относительно которого доказана его алгоритмическая универсальность. Удобно (особенно при реализации в среде NetLogo) рассматривать данный автомат как некоторое устройство (муравья), совершающее движения по прямоугольной решетке, клетки которой находятся в двух состояниях, например в черном и зеленом. Сам муравей обладает ориентацией (вверх, вниз, вправо и влево). Правила его движения следующие: если текущая клетка, в которой находится муравей, черная, то он перекрашивает клетку в зеленый цвет, делает поворот налево на 90° и передвигается на одну клетку вперед; если клетка под ним зеленая, то она перекрашивается в черный цвет, делается поворот направо и переход на клетку вперед. Если начать с пустого поля, то муравей Лэнгтона примерно 10 000 итераций совершает хаотические движения около стартовой точки, а потом в его поведении происходит самоорганизация — муравей начинает строить «дорогу» в одном из диагональных направлений (рис. 12.12).

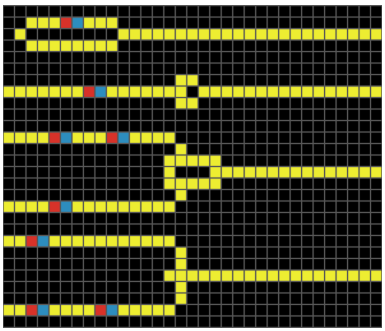

РИС. 12.13 Реализация элементов логических схем в автомате WireWorld: тактовый генератор, диод, логические элементы XOR и OR

10 Еще одним примером[8] алгоритмически универсального двумерного клеточного автомата является автомат WireWorld с окрестностью Мура. Клетки в этом

[6] Н. Горькавый, *Сказка о химике Белоусове, который изготовил жидкие часы*, Наука и жизнь, 2011, № 2.

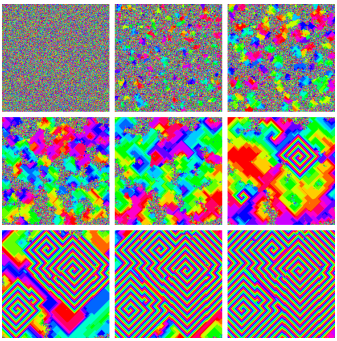

РИС. 12.11 Циклический клеточный автомат

[7] C. Langton, *Studying artificial life with cellular automata*, Physica D: 1986, 22, p. 120–149.

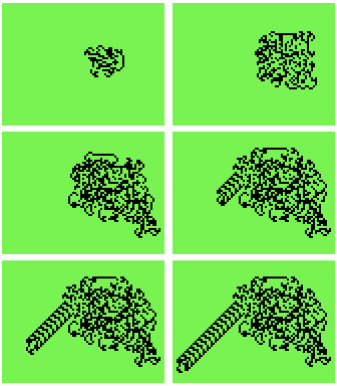

РИС. 12.12 Муравей Лэнгтона

[8] A. K. Dewdney, *Computer Recreations*, Scientific American, 262 (1), 1990, p. 136–139.

автомате могут находиться в четырех состояниях, которые трактуются как: 0 — пустое место, 1 — «голова» электрона, 2 — «хвост» электрона, 3 — проводник. Правила автомата очень простые: состояние 1 переходит в состояние 3, состояние 2 в 1, состояния 3 переходит в 1 только в том случае, если среди соседей клетки ровно одна или две клетки в состоянии 2. Данный автомат позволяет относительно легко моделировать работу разных логических вентилей и других элементов электрических схем (рис. 12.13, состоянию 1 соответствуют красные клетки, состоянию 2 — синие). Выполните реализацию этого автомата с поддержкой рисования схем.