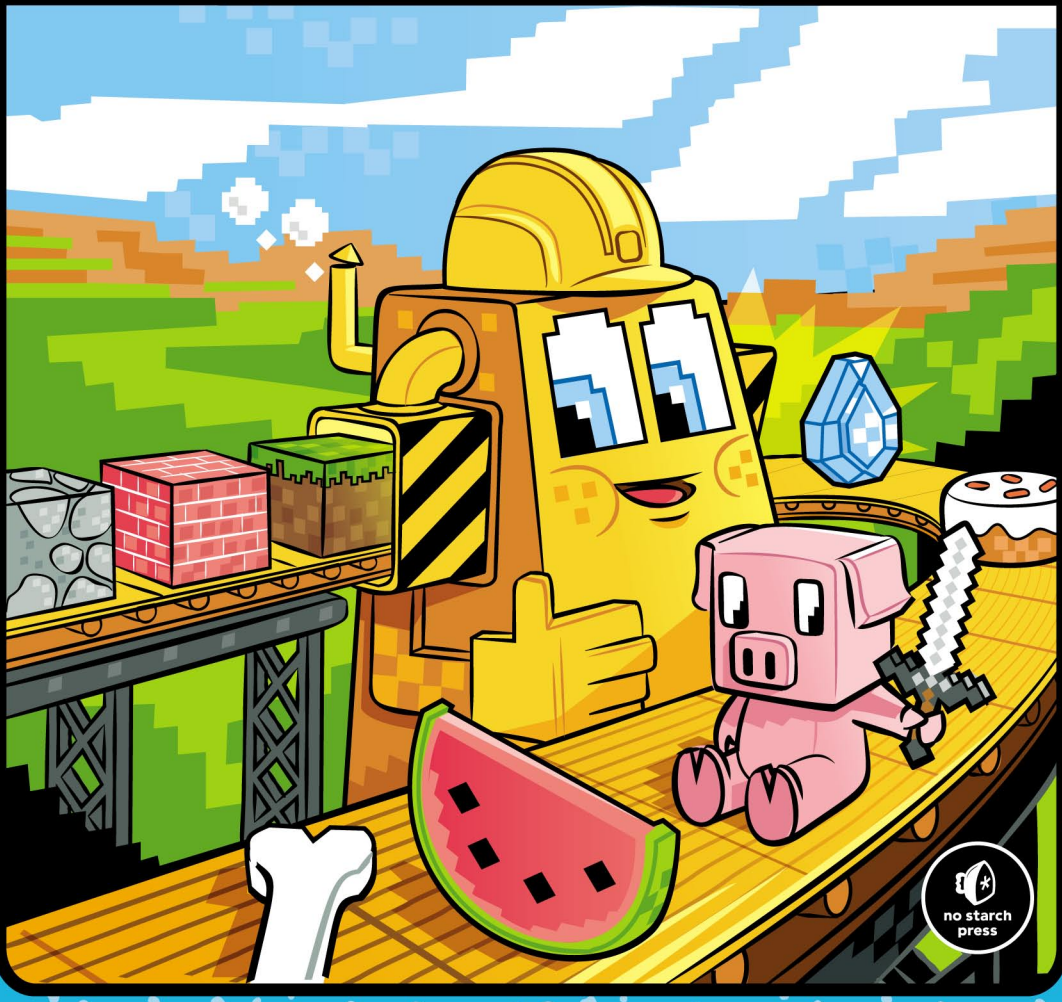
ПРОГРАММИРУЙ В MINECRAFT

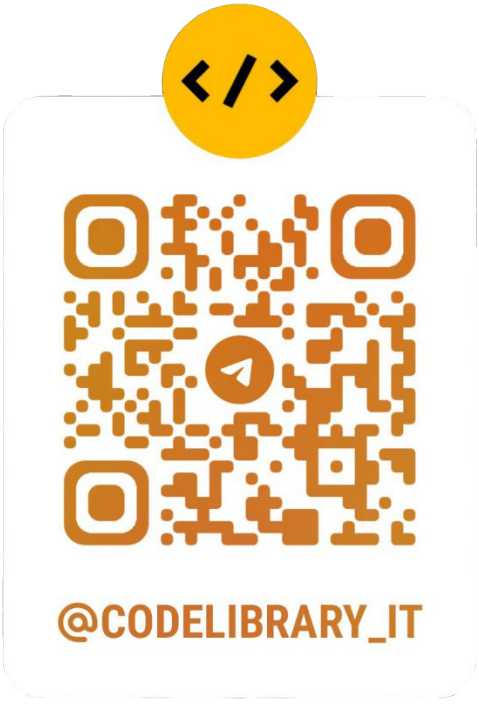
СТРОЙ ВЫШЕ, ВЫРАЩИВАЙ БЫСТРЕЕ, КОПАЙ ГЛУБЖЕ И АВТОМАТИЗИРУЙ ВСЮ СКУЧНУЮ РАБОТУ!

ЭЛ СВЕЙГАРТ

ДЛЯ ДЕТЕЙ СТАРШЕ 10 ЛЕТ



ПРОГРАММИРОВАНИЕ
ДЛЯ ДЕТЕЙ



AL SWEIGART

CODING WITH MINECRAFT

BUILD TALLER, FARM FASTER, MINE DEEPER,
AND AUTOMATE THE BORING STUFF

ЭЛ СВЕЙГАРТ

ПРОГРАММИРУЙ В MINECRAFT

СТРОЙ ВЫШЕ, ВЫРАЩИВАЙ БЫСТРЕЕ, КОПАЙ ГЛУБЖЕ
И АВТОМАТИЗИРУЙ ВСЮ СКУЧНУЮ РАБОТУ!

БОМБОРА™
Москва 2019

УДК 004.4-93
ББК 32.973.26-018
С24

Al Sweigart
CODING WITH MINECRAFT: BUILD TALLER, FARM FASTER, MINE DEEPER, AND AUTOMATE THE BORING STUFF

Свейгарт, Эл.

С24 Программируй в Minecraft. Строй выше, выращивай быстрее, копай глубже и автоматизируй всю скучную работу! / Эл Свейгарт ; [пер. с англ. Райтман М.А.]. — Москва : Эксмо, 2019. — 344 с. — (Программирование для детей).

Новая книга для всех любителей игры Minecraft от автора мировых бестселлеров-самоучителей по программированию Эла Свейгарта. Благодаря своему многолетнему опыту, Свейгарт умеет в увлекательной игровой форме подать даже самую непростую информацию и заинтересовать даже тех маленьких читателей, которых чтение не интересует. Самая популярная у детей во всем мире игра станет еще интереснее, когда ваш ребенок научится использовать и своими руками программировать автоматических помощников-черепашек, делающих в Minecraft за игрока всю самую скучную работу. С этой книгой читатели смогут не только стать продвинутыми геймерами, но и познакомятся с основами программирования и получат ценные навыки, которые в дальнейшем смогут развить.

УДК 004.4-93
ББК 32.973.26-018

Издание для досуга
Для среднего школьного возраста
ПРОГРАММИРОВАНИЕ ДЛЯ ДЕТЕЙ

Эл Свейгарт

ПРОГРАММИРУЙ В MINECRAFT
Строй выше, выращивай быстрее, копай глубже и автоматизируй всю скучную работу!

Главный редактор *Р. Фасхутдинов.* Ответственный редактор *Е. Истомина*
Художественный редактор *А. Гусев.* Младший редактор *Е. Минина*

ООО «Издательство «Эксмо»
123308, Москва, ул. Зорге, д. 1. Тел.: 8 (495) 411-68-86. Home page: www.eksmo.ru E-mail: info@eksmo.ru
Өндіруші: «ЭКСМО» АҚБ Баспасы, 123308, Мәскеу, Ресей, Зорге көшесі, 1 үй. Тел.: 8 (495) 411-68-86.
Home page: www.eksmo.ru E-mail: info@eksmo.ru.
Тауар белгісі: «Эксмо»
Интернет-магазин : www.book24.ru
Интернет-дукен : www.book24.kz
Импортёр в Республику Казахстан ТОО «РДЦ-Алматы».
Қазақстан Республикасындағы импорттаушы «РДЦ-Алматы» ЖШС.
Дистрибьютор и представитель по приему претензий на продукцию, в Республике Казахстан: ТОО «РДЦ-Алматы»
Қазақстан Республикасында дистрибьютор және өнім бойынша арыз-талаптарды
қабылдаушының өкілі «РДЦ-Алматы» ЖШС, Алматы қ., Домбровский көш., 3«а», литер Б, офис 1.
Тел.: 8 (727) 251-59-90/91/92; E-mail: RDC-Almaty@eksmo.kz
Өнімнің жарамдылық мерзімі шектелмеген.
Сертификация туралы ақпарат сайтта: www.eksmo.ru/certification

Сведения о подтверждении соответствия издания согласно законодательству РФ о техническом регулировании можно получить на сайте Издательства «Эксмо» www.eksmo.ru/certification
Өндірген мемлекет: Ресей. Сертификация қарастырылмаған

Подписано в печать 31.08.2018. Формат 70x100$^{1}/_{16}$.
Печать офсетная. Усл. печ. л. 27,87.
Тираж экз. Заказ

ISBN 978-5-04-096471-0

ISBN 978-5-04-096471-0

СОДЕРЖАНИЕ

ОБ АВТОРЕ

Эл Свейгарт – профессиональный разработчик программного обеспечения, совмещающий работу с преподаванием основ программирования детям и взрослым. Эл написал несколько топовых книг для начинающих, включая «Учим Python, делая крутые игры» и «Программирование для детей. Делай игры и учи язык Scratch!».

ВВЕДЕНИЕ

«Еще пара алмазов – и перерыв!», – вспоминаю я, как играл в Minecraft. Мне кровь из носу нужны были алмазы для крафта новой кирки. А кирка была нужна для добычи обсидиана. Обсидиан же требовался, чтобы построить портал в Незер. Я хотел отправиться в Незер за лавой. А лава была нужна… ах, для чего же мне была нужна лава? А, точно! Я обнаружил гигантскую статую феникса в скалах и хотел сделать так, чтобы лава выливалась из глаз и клюва этой потрясающей птицы. Спустя два часа я все еще играл, не в силах оторваться.

Minecraft – захватывающая игра. К началу 2018 года продано свыше 144 миллионов[1] копий этой игры, благодаря чему Minecraft стал второй по популярности компьютерной игрой всех времен и народов, уступив только легендарному «Тетрису». Minecraft – это открытый креативный мир, созданный для сбора ресурсов и строительства самых невероятных конструкций. Ты можешь возводить замки

[1] expandedramblings.com/index.php/minecraft-statistics/

для защиты от наседающих орд зомби, засеивать поля полезными культурами и разводить скот на фермах, или же, объединившись с друзьями, воссоздавать в виртуальном мире шедевры искусства. Minecraft нацелен на самую разную аудиторию: в восторге от этой игры дети, подростки и даже взрослые.

В этой книге ты займешься изучением мода ComputerCraft (также известного под кодовым названием CC), который превратит твое желание крафтить в желание программировать. Но что такое моды и сам ComputerCraft?

ЧТО ТАКОЕ МОДЫ ДЛЯ MINECRAFT?

Сама по себе игра Minecraft, называемая ванильной версией Minecraft, – это только основа. Minecraft может быть модифицирован и расширен программным обеспечением сторонних разработчиков, называемым модами. Эти моды предоставляют игроку дополнительные функции, в том числе новые блоки, устройства, предметы, мобов и даже миры, которые не включены в ванильную версию игры. Благодаря своей популярности игра Minecraft собрала вокруг себя одно из крупнейших сообществ фанатов-модеров компьютерных игр.

Эти фанатские моды можно скачать совершенно бесплатно. Одни моды добавляют возможность исследования космоса и запуска ракет. Другие – магию и заклинания. Ты даже можешь встретить моды Minecraft для строительства собственных парков Юрского периода, высокоскоростных железнодорожных сетей и пчеловодных хозяйств. Для этой книги я выбрал мод ComputerCraft – он поможет тебе научиться программировать.

ЧТО ТАКОЕ COMPUTERCRAFT?

ComputerCraft – это мод для игры Minecraft, который добавляет в нее программируемых роботов, называемых черепашками. Эти роботы невероятно круты и способны делать почти все, что может игрок: копать шахты, рубить

деревья, строить здания, крафтить предметы, сажать семена, доить коров, выпекать торты и многое другое (см. рис. 1). Армия таких квадратных черепашек может автоматически выполнять все рутинные действия, которые игрок обычно делает самостоятельно.

Рис. **1**. *Черепашки, добывающие древесину (слева), и добыча (справа)*

Фишка в том, что сначала ты должен научиться программировать. Черепашки из мода ComputerCraft понимают код, написанный на языке программирования Lua («луна» в переводе с португальского), который используется профессиональными разработчиками ПО в таких областях, как встраиваемые вычислительные системы и видеоигры. Программное обеспечение интерпретатора, выполняющего сценарии на языке Lua, «весит» всего 100 Кб, поэтому его можно легко встроить в другие программные элементы, например моды для Minecraft. Язык Lua часто используется при разработке видеоигр, например World of Warcraft, Dark Souls, Portal 2, Factorio и многих других.

Хотя Lua и проще других языков программирования, он быстрый и мощный. Это пойдет и нам на пользу. Благодаря своей простоте Lua – отличный язык для развития программистских навыков, если у тебя нет опыта программирования.

КАК ПОЛЬЗОВАТЬСЯ ЭТОЙ КНИГОЙ?

Тебе нужно приобрести копию игры Minecraft для операционной системы Windows или macOS, а также скачать бесплатный мод ComputerCraft и бесплатную программу-

лаунчер ATLauncher. См. главу 1, в которой я привел все инструкции по загрузке и установке.

При наборе примеров исходного кода из этой книги не вводи номер в начале каждой строки. Например, если ты встретил указанную ниже строку кода, тебе не нужно будет вводить число 9, указанное слева, точку и пробел следом за ней:

```
9. print ("Как тебя зовут?")
```

Ты должен ввести только это:

```
print ("Как тебя зовут?")
```

Номера строк нужны для того, чтобы по тексту книги ссылаться на те или иные строки в коде программы. Они не относятся к исходному коду программы.

В некоторых случаях в листингах кода ты можешь встретить ненумерованные строки с текстом `...пропуск...`. Такие строки указывают на то, что часть кода была пропущена для экономии бумаги. Сами строки `...пропуск...` не относятся к исходному коду программы.

СТРУКТУРА КНИГИ

Прочитав первые несколько глав, описывающие основные концепции программирования, ты приступишь непосредственно к программированию. Каждая последующая глава этой книги посвящена написанию программы, которую может выполнять твоя черепашка, чтобы помочь тебе выживать и процветать в игре Minecraft. Ты также встретишь дополнительные задания, которые позволят проверить твои навыки программирования.

Вот что ты узнаешь в каждой главе.

► Из главы 1 ты узнаешь, как установить и настроить игру Minecraft и мод ComputerCraft, чтобы начать изучать программирование.

► В главе 2 представлены основные концепции программирования и описана интерактивная оболочка.

► Глава 3 знакомит с редактором файлов, который ты будешь использовать для написания кода первой программы.

► Глава 4 рассказывает, как написать программу для перемещения черепашек в мире Minecraft.

► Глава 5 научит тебя добавлять к танцевальной программе из предыдущей главы новые инструкции программирования.

► В главе 6 ты узнаешь, как научить черепашку рубить деревья и собирать древесину.

► Глава 7 научит экономить время: писать код один раз и использовать его в разных программах.

► Глава 8 развивает тему главы 6: ты создашь полностью автоматическое деревообрабатывающее предприятие, на котором черепашки смогут собирать древесину с разных деревьев.

► Глава 9 направлена на разработку программы для добычи неограниченных запасов булыжника, который черепашки будут использовать в качестве строительного материала в главах 10–13.

► В главе 10 ты построишь кирпичный заводик, превращающий булыжник, производимый программой из главы 9, в каменные кирпичи.

► Глава 11 рассказывает о программе, которая помогает строить каменные стены из кирпичей.

► Из главы 12 ты узнаешь, как написать программу для формирования комнат из стен.

► Глава 13 посвящена созданию полов и потолков в комнатах.

► Глава 14 направлена на добычу еды: черепашки будут сажать и собирать различные виды сельскохозяйственных культур, чтобы твой герой не умер с голоду.

► В главе 15 ты разработаешь программу для копания шахт и добычи руды и других полезных ископаемых.

► В справочнике функций перечислены и описаны все функции из этой книги.

Справочник идентификаторов содержит список из наиболее популярных блоков и предметов Minecraft с указанием их идентификаторов, которые используются в программах.

ПОЛУЧЕНИЕ ПОМОЩИ

В игре Minecraft очень не хватает инструкции для новичков. В ней нет руководства пользователя или даже меню «Справка». Играя в Minecraft, ты сам несешь ответственность за свое обучение. Тебе нужно будет искать в Интернете способы решения актуальных задач, задавать вопросы и искать на них ответы, а иногда попросту экспериментировать на свой страх и риск. Minecraft культивирует у игроков гибкий тип мышления. Даже после того, как они упали в лаву или натолкнулись на нее, пробив стену, они возвращаются к этому месту, чтобы научиться преодолевать проблему.

Тем не менее эта книга посвящена моду ComputerCraft, а не основам игры в Minecraft. Чтобы полноценно воспользоваться этой книгой, в Minecraft ты должен уметь следующее:

► добывать руду, камень, уголь, древесину и другие блоки;

► крафтить верстак и инструменты, такие как топоры, лопаты, факелы и кирки;

► крафтить печь, наполнять ее топливом и переплавлять в ней блоки руды или жарить мясо;

► крафтить ступеньки, лестницы, сундуки, двери, заборы и другие архитектурные элементы, которые будешь использовать при строительстве сооружений;

► сеять семена и собирать урожай.

Если ты не умеешь этого, не волнуйся. Ты можешь всему научиться, воспользовавшись поиском в Интернете. Открой сайт своей любимой поисковой системы и введи слово minecraft вместе с запросом действия, которое хочешь изучить. Например, ты можешь использовать поисковые запросы minecraft добывать руду, minecraft сделать торт или даже просто minecraft руководство для новичков, чтобы найти нужную информацию. Ты также можешь искать видеоролики, показывающие как что-то делать в Minecraft, на веб-сайтах типа www.youtube.com, указывая те же запросы, что и в поисковой системе.

Как я упоминал ранее, поскольку мод ComputerCraft был разработан и обновляется совершенно другими людьми, а не компанией Mojang, на многих сайтах, посвященных этой игре, ты не найдешь информацию о моде. Официальная документация к моду ComputerCraft на английском языке доступна на вики-сайте **www.computercraft.info/wiki/.** Краткая справка на русском языке опубликована на сайте **minecraft-ru.gamepedia.com/ComputerCraft**. Если у тебя возникли вопросы, связанные с этим модом, ты можешь зарегистрировать бесплатную учетную запись на форуме **www.computercraft.info/forums2/** и задавать свои вопросы там. Если же возникли дополнительные вопросы о программах из этой книги, на них ты сможешь получить ответы по адресу **www.reddit.com/r/turtleappstore/**.

ВЕБ-РЕСУРСЫ

Загрузка всех программ, описанных в этой книге, возможна непосредственно из игры Minecraft (см. раздел **«Публикация и загрузка программ в Интернете»** главы 4 для получения подробных инструкций). И хотя Minecraft не поддерживает копирование и вставку кода вне игры, весь код и дополнительные файлы для этой книги доступны на сайт **eksmo.ru/files/minecraft_files.zip.** Там же можно скачать файлы с кодом для выполнения дополнительных заданий, если ты испытываешь затруднения и хочешь проверить решения. Ты также найдешь ссылки

на дистрибутивы необходимого программного обеспечения (для дополнительной информации см. главу 1). Если ты хочешь посмотреть программы других пользователей или поделиться собственными, то можешь сделать это на сайте turtleappstore.com, позволяющем бесплатно публиковать сценарии для мода ComputerCraft (для дополнительной информации см. раздел **«turtleappstore.com»** в главе 4).

ЧТО МЫ УЗНАЛИ

Minecraft – феноменальная игра, которую можно проходить разными способами, что привлекает самую разношерстную публику. В этой книге ты узнаешь, как возводить постройки намного быстрее с помощью мода ComputerCraft, который позволяет программировать на языке Lua специальных роботов-черепашек. Обучаясь программированию на языке Lua с помощью мода ComputerCraft, ты научишься автоматизировать многие рутинные задачи, которые ранее приходилось делать самостоятельно, например добывать полезные ископаемые, вести сельское хозяйство, строить и крафтить.

С помощью игры Minecraft и мода ComputerCraft ты научишься решать проблемы самостоятельно и приобретешь базовые навыки программирования на компьютере.

Поехали!

1

НАЧАЛО РАБОТЫ С COMPUTERCRAFT

Прежде чем ты начнешь программировать роботизированных черепашек, нужно установить и настроить игру Minecraft и мод ComputerCraft. Не волнуйся, бесплатное программное обеспечение ATLauncher сильно упрощает этот процесс.

В этой главе я покажу тебе, как скачать и установить игру Minecraft и мод ComputerCraft, а затем проведу через все этапы настройки, которые нужно выполнить, прежде чем приступить к программированию.

УСТАНОВКА MINECRAFT, JAVA, ATLAUNCHER И COMPUTERCRAFT

Раньше добавлять и настраивать моды в игру Minecraft было трудно, поскольку нужно было выполнить ряд сложных шагов. А теперь ты можешь воспользоваться специальным лаунчером[2] ATLauncher для простой и быстрой загрузки мо-

[2] Программой, запускающей другие программы с определенными настройками.

дов в игре Minecraft. Так как Minecraft, ATLauncher и ComputerCraft разработаны разными компаниями, тебе нужно скачать и установить каждую программу отдельно. Все они доступны для операционных систем Windows, macOS и Ubuntu. Кроме того, мод ComputerCraft нельзя запустить на мобильных платформах, а также игровых приставках Microsoft Xbox и Sony PlayStation.

Моды работают только для Java Edition Minecraft, который также называется версией Minecraft для Windows. Версия Windows 10 для Minecraft не поддерживает моды, хотя она поддерживает новую форму модов, называемых надстройками. Но не волнуйся! Мы будем использовать программное обеспечение ATLauncher для загрузки и установки правильной версии Minecraft.

ПОКУПКА MINECRAFT В ИНТЕРНЕТЕ

Программы Java, ATLauncher и ComputerCraft бесплатны, а вот игра Minecraft распространяется компанией Mojang (теперь принадлежащей корпорации Microsoft) на платной основе. Купить игру можно в Интернете по адресу **www.minecraft.net** после регистрации бесплатной учетной записи Mojang. После оплаты покупки не загружай игру Minecraft с указанного сайта. Вместо этого для загрузки и установки Minecraft на свой компьютер ты запустишь лаунчер ATLauncher. (Если у тебя уже установлена игра Minecraft, ты все равно должен следовать инструкциям по установке игры Minecraft с помощью программы ATLauncher.)

Примечание *Храни свой пароль от учетной записи Mojang в безопасном месте и не говори его друзьям и людям, выдающим себя за сотрудников Mojang. Используй его только для авторизации в лаунчере ATLauncher или на сайте www.minecraft.net и других ресурсах Mojang. Если ты считаешь, что кто-то еще может войти в твою учетную запись, немедленно измени пароль к ней.*

ЗАГРУЗКА И УСТАНОВКА СРЕДЫ JAVA

Среда Java необходима для запуска программ, написанных на языке Java. К таким программам относятся и Minecraft, и ATLauncher. Хотя современные версии игры Minecraft не требуют отдельной установки среды Java, в случае ее отсутствия на компьютере лаунчер ATLauncher попросту не запустится.

Чтобы скачать среду Java, перейди на сайт **java.com/ru/download/**. Нажми красную кнопку Загрузить **Java** бесплатно (Free Java Download), а затем кнопку Согласиться и начать бесплатную загрузку (Agree and Start Free Download). Сохранив скачанный файл в любой папке, запусти его и, следуя указаниям установщика, установи среду Java.

В случае затруднений посмотри видеоинструкцию по установке по адресу **youtu.be/f9SGAdDLAU4**.

ЗАГРУЗКА И УСТАНОВКА ATLAUNCHER

Программа ATLauncher упрощает добавление модов в игру Minecraft. На рис. 1.1 показан сайт **www.atlauncher.com**, на котором ты можешь бесплатно скачать эту программу. Перейди на вкладку **Downloads** в верхней части страницы, чтобы получить доступ к ссылкам на дистрибутив программы и видеоинструкции по установке. Для загрузки и установки программы ATLauncher нажми кнопку с ссылкой на дистрибутив для твоей операционной системы. В случае затруднений посмотри видеоинструкцию по установке. Видеоинструкция также доступна на сайте YouTube по адресу **youtu.be/mXb94DpiF78**.

Внимание ! *Обрати внимание, что перед запуском установки программы ATLauncher нужно поместить скачанный файл дистрибутива в отдельную папку. Не запускай файл установки из корневого каталога или рабочего стола! Программа установки извлекает все файлы в ту же папку, в которой находится и сам файл дистрибутива.*

После установки программы ATLauncher, запусти ее и перейди на вкладку **Accounts** (Учетные записи) справа. Укажи свои имя пользователя и пароль учетной записи Mojang. Ты можешь установить флажок **Remember password** (Запомнить пароль), чтобы не вводить данные своей учетной записи каждый раз, когда запускаешь Minecraft.

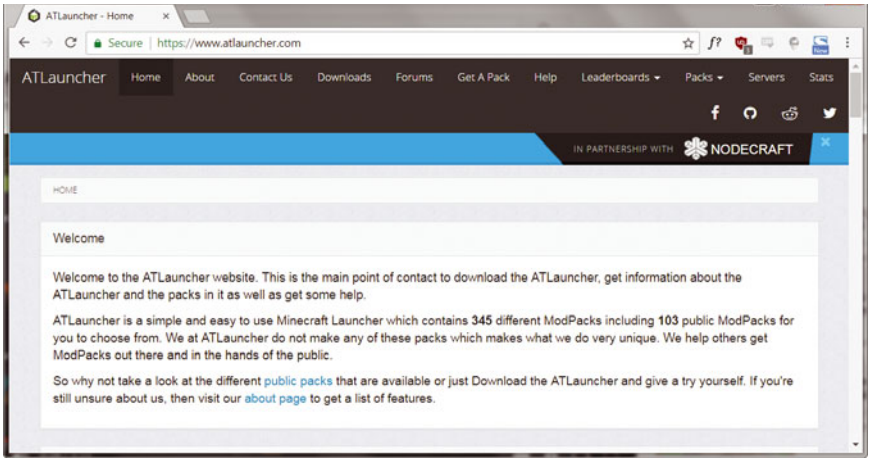


Рис. **1.1.** *Веб-сайт, на котором ты можешь скачать программу ATLauncher*

Чтобы подключиться к серверу Minecraft, тебе нужно установить те же моды (и версии этих модов), что и на сервере. Поскольку существует множество комбинаций модов, фанаты придумали пакеты модов, которые содержат «сбалансированный» набор модов. Сейчас мы загрузим «ванильный» пакет модов (т.е. пакет с очень небольшим количеством модов) и добавим в игру мод ComputerCraft. Перейди на вкладку **Packs** (Пакеты) и найди пакет **Vanilla Minecraft**. Ты добавишь мод ComputerCraft в эту версию игры Minecraft.

Нажми кнопку **New Instance** (Новый экземпляр) в описании мода Vanilla Minecraft. Экземпляр – это копия пакета модов, установленная на твоем компьютере. В появившемся диалоговом окне укажи название экземпляра, например **CC Minecraft**. Обязательно выбери версию игры **1.8.9** (Minecraft 1.8.9) в раскрывающемся списке ниже, потому что это последняя версия игры, которая под-

держивает мод ComputerCraft. Установи флажок **Enable User Lock** (Включить блокировку пользователя). Активация данного параметра означает, что только пользователь текущей учетной записи операционной системы сможет получить доступ к этому экземпляру.

Примечание *Будущие версии ComputerCraft могут быть совместимы с более новыми версиями игры Minecraft. В этом случае обновленные инструкции по установке я опубликую на сайте* ***www.nostarch.com/codingwithminecraft/.***

В появившемся окне **Select Mods to Install** (Выбор модов для установки) установи только один флажок – напротив пункта рядом с **Minecraft Forge (Recommended)** (Minecraft Forge (рекомендуется)), – а затем нажми кнопку **Install** (Установить). Мод Minecraft Forge необходим для работы ComputerCraft.

МНОГОПОЛЬЗОВАТЕЛЬСКАЯ ИГРА С МОДАМИ

Если ты только знакомишься с игрой Minecraft, я рекомендую начинать с однопользовательского режима. Но если ты хочешь подключиться к публичному многопользовательскому серверу, в твоем экземпляре должны быть установлены те же моды, что и на сервере. На большинстве публичных серверов вряд ли будет установлен мод ComputerCraft, поэтому тебе понадобится установить на вкладке Packs (Пакеты) другой пакет модов, а не Vanilla Minecraft. Популярные пакеты модов, которые содержат ComputerCraft, – это Resonant Rise, Sky Factory 2, Space Astronomy, Golden Cobblestone и Yogscast Complete. (Мне больше всего нравится Resonant Rise.)

Ты также можешь пролистать список пакетов модов на вкладке **Packs** (Пакеты) и, нажимая кнопку **View Mods** (Просмотр модов), просмотреть, входит ли в состав мод ComputerCraft. Если ты решил установить один из таких пакетов модов, пропусти раздел «Загрузка и установка ComputerCraft», потому что корректные версии ComputerCraft и Minecraft будут ав-

томатически установле ны вместе с пакетом. Пакетам модовприсваиваются номера версий, поэтому, если ты решил играть на публичном сервере, обязательно устанавливай ту же версию пакета модов, которая установлена на этом сервере.
Чтобы найти публичные серверы для выбранного пакета модов, выполни в Интернете поиск по запросу название_пакета_модов сервер Minecraft или перейди на сайт **www.atlauncher.com** и щелкни мышью по ссылке **Servers** в верхней части страницы.

Когда установка будет завершена, ты найдешь свой экземпляр игры Minecraft на вкладке **Instances** (Экземпляры) в правой части окна ATLauncher. Это экземпляр ванильной версии игры Minecraft, поэтому тебе нужно вручную загрузить и установить мод ComputerCraft.

ЗАГРУЗКА И УСТАНОВКА COMPUTERCRAFT

В этом разделе я расскажу, как загрузить и установить мод ComputerCraft. Также ты можешь просмотреть видеоролик, демонстрирующий, как установить ComputerCraft, на сайте **youtu.be/g4Zs2JY1vi8/**. Скачать мод ComputerCraft ты можешь по адресу **www.computercraft.info/download/**. Щелкни мышью по ссылке **Download ComputerCraft 1.79 (for Minecraft 1.8.9)**, чтобы скачать файл *ComputerCraft1.79.jar*. Этот файл также доступен по адресу **www.nostarch.com/codingwithminecraft/**.

Я держу этот файл в папке лаунчера ATLauncher, чтобы его можно было легко найти. На вкладке **Instances** (Экземпляры) в правой части окна программы ATLauncher нажми кнопку **Edit Mods** (Изменить моды) своего экземпляра Vanilla Minecraft. Откроется диалоговое окно Editing Mods (Изменение модов), как показано на рис. 1.2, в котором в настоящее время добавлен и активирован только мод Minecraft Forge.

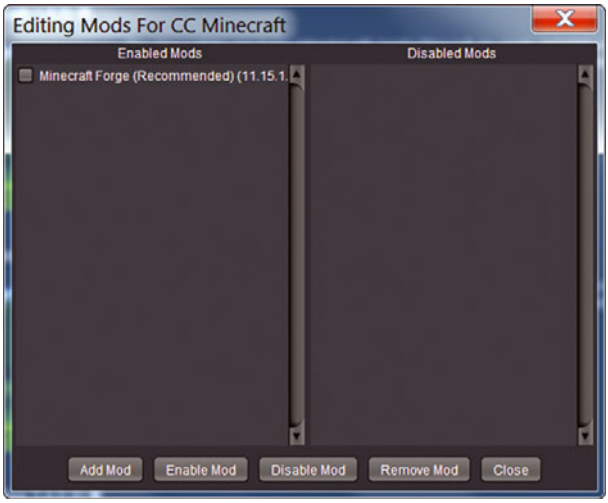


Рис. **1.2.** *Изменение модов для твоего экземпляра игры*

Нажми кнопку **Add Mod** (Добавить мод), в появившемся окне щелкни мышью по кнопке **Select** (Выбрать) и выбери ранее скачанный файл *ComputerCraft1.79.jar*. В раскрывающемся списке **Type of Mod** (Способ модификации) выбери пункт **Inside Minecraft.jar** (Внутри Minecraft.jar).

Затем нажми кнопку **Add** (Добавить).

Пункт **ComputerCraft1.79.jar** появится в правой части окна, в столбце **Disabled Mods** (Отключенные моды). Этот мод добавлен в твой экземпляр Minecraft, но ты должен активировать его, чтобы начать использовать. Установи флажок напротив пункта **ComputerCraft1.79.jar**, а затем нажми кнопку **Enable Mod** (Включить мод) в нижней части окна. Пункт **ComputerCraft1.79.jar** переместится в левую часть окна в раздел **Enabled Mods** (Включенные моды). Нажми кнопку **Close** (Закрыть), чтобы закрыть окно.

Вернувшись на вкладку **Instances** (Экземпляры), нажми кнопку **Play** (Играть) своего экземпляра Minecraft, чтобы запустить игру. При первом нажатии кнопки **Play** (Играть) появится окно с сообщением о том, что доступно обновление, как показано на рис. 1.3. Нажми кнопку **Don't Remind Me Again** (Не напомнить мне снова), чтобы отказаться от обновления и скрыть это окно.

Если ты случайно нажмешь кнопку **Yes** (Да) и обновишь экземпляр, вполне возможно, мод ComputerCraft будет работать неправильно. В этом случае удали экземпляр игры и повтори все шаги по установке и настройке сначала.

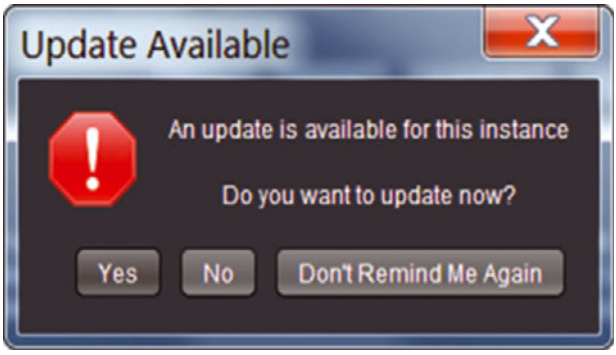


Рис. **1.3.** *Уведомление о доступном обновлении*

НАСТРОЙКА МОДА COMPUTERCRAFT

У мода ComputerCraft есть некоторые настраиваемые параметры, записываемые в текстовый файл *ComputerCraft.cfg*. В большинстве случаев, тебе не нужно редактировать этот файл, и я предполагаю, что ты используешь конфигурацию по умолчанию. Большинство публичных многопользовательских серверов Minecraft также используют настройки по умолчанию. Однако если по какой-то причине ты хочешь изменить настройки мода ComputerCraft, тебе нужно знать, где находится файл конфигурации.

Ты найдешь этот файл на своем компьютере там же, куда установил программу ATLauncher. На моем компьютере под управлением операционной системы Windows я установил программу ATLauncher в папку *C:\ATLauncher*, поэтому файл конфигурации я обнаружил по адресу *C:\ATLauncher\Instances\CCMinecraft\config\ComputerCraft.cfg*. Открой этот файл в текстовом редакторе, например Блокнот (Notepad) или TextMate. Настройки в файле сопровождаются комментариями. Ты можешь изменять параметры, указывая новые значения после знака равенства (=).

ЗАПУСК MINECRAFT

После запуска лаунчера ATLauncher и нажатия кнопки **Play** (Играть) появится главное меню игры Minecraft, как показано на рис. 1.4.

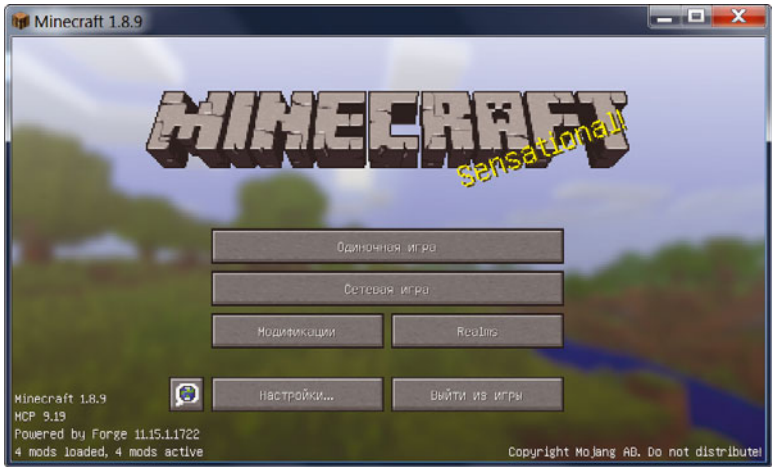


Рис. **1.4.** *Главное меню игры Minecraft*

Нажми кнопку **Одиночная игра** (Singleplayer), откроется список миров, которые ты создал. Если Minecraft ты запустил впервые, список будет пуст – тебе нужно создать свой первый мир.

СОЗДАНИЕ НОВОГО МИРА

В игре Minecraft нет какого-то стандартного набора уровней. Игровой мир генерируется случайным образом и всегда открывает новую, неизвестную область исследования. Нажми кнопку **Создать новый мир** (Create New World), чтобы открыть одноименный экран, как показано на рис. 1.5. Введи название мира, например **Мой мир Computer-Craft**, в текстовое поле **Название мира** (World Name). Затем нажимай кнопку **Режим игры** (Game Mode), пока на ней не отобразится строка **Режим игры: Выживание** (Game Mode: Survival). (Ты также можешь переключать режимы в процессе игры.)

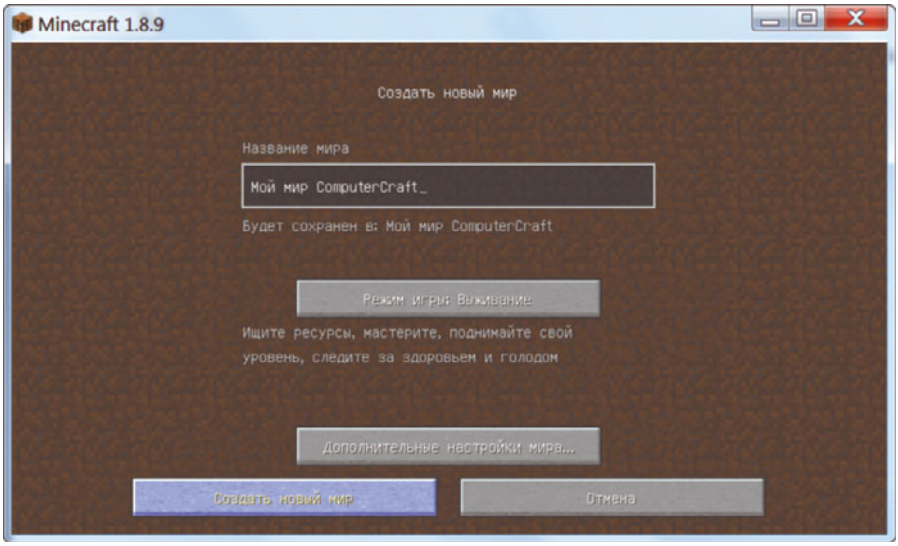


Рис. **1.5.** *Экран Создать новый мир*

Нажми кнопку **Дополнительные настройки мира** (More World Options), а затем кнопку **Использование читов** (Allow Cheats), чтобы на ней отобразилась строка **Использование читов: ВКЛ** (Allow Cheats: ON). Читы позволят тебе переключаться между режимами игры, выживания и творческим, о которых я расскажу далее. Чтобы создать новый мир, нажми кнопку **Создать новый мир** (Create New World).

РАЗЛИЧИЯ ИГРОВЫХ РЕЖИМОВ MINECRAFT

В Minecraft доступно три режима, в которых ты можешь играть: **Творческий** (Creative), **Выживание** (Survival) и **Хардкор** (Hardcore).

В творческом режиме ты можешь сколь угодно летать, не нужно есть, ты не можешь умереть и у тебя есть неограниченное количество любых блоков. Этот режим идеален, если ты хочешь заняться строительством из блоков Minecraft сооружений, таких как замки или гигантские произведения искусства.

В режиме выживания игра становится приближенной к реальности. Тебе нужно будет искать пищу, чтобы твой персонаж не умер от голода. Единственные блоки, с помощью которых ты можешь строить, – те, которые ты добыл самостоятельно. Ты не можешь летать, а если упадешь с обрыва, велик шанс получить травмы или вовсе свернуть шею. Ну а ночью появляются монстры.

Поскольку при создании мира ты разрешил использование читов, ты можешь использовать команду `/gamemode` для переключения между режимами выживания и творческим. Чтобы изменить игровой режим, нажми клавишу **T**, открывающую окно консоли, и введи строку `/gamemode creative` или `/gamemode survival`, как показано на рис. 1.6.

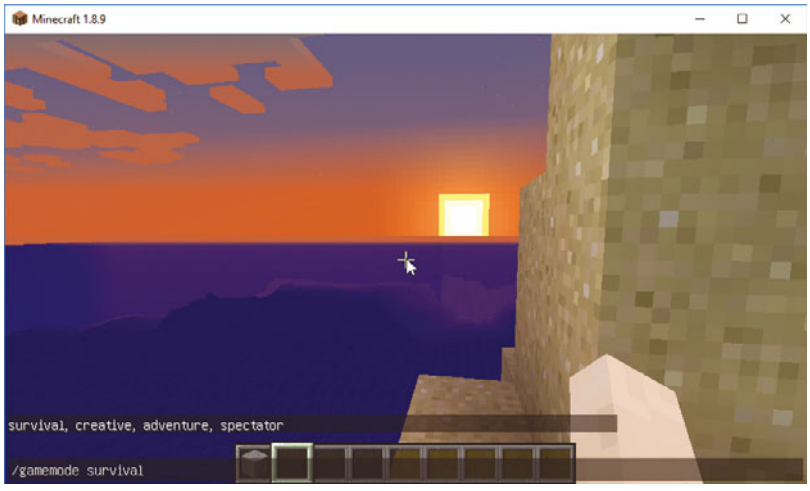


Рис. **1.6.** *Ввод команды* `/gamemode` *в окне консоли*

На мой взгляд, в режиме выживания играть сложнее и интереснее. Ресурсы ограничены, а блоки нужно добывать вручную. Режим выживания также подойдет, если используется мод ComputerCraft. Программируя черепашек, автоматизирующих рутинные задачи, ты всегда будешь в безопасности, сыт и с полным набором блоков и предметов.

Не переживай о том, что в режиме выживания твой персонаж может умереть. Если он погибнет, то уронит

предметы и блоки, которые носил в инвентаре, и вернется к исходной точке спауна, откуда ты сможешь продолжить игру.

Хардкор-режим похож на режим выживания, но у тебя будет только одна жизнь. После смерти игровой мир придется покинуть. Мод ComputerCraft поддерживает Хардкор-режим, но сейчас я рекомендую играть в режиме выживания.

ЧТО МЫ УЗНАЛИ

Вот ты и выполнил все настройки, необходимые для обучения программированию с помощью этой книги. Ты скачал и установил игру Minecraft, среду Java, лаунчер ATLauncher и мод ComputerCraft. Как упоминалось ранее, Java, ATLauncher и ComputerCraft бесплатны, а вот игра Minecraft – нет.

Игра Minecraft так велика и имеет столько модов, что перед началом игры требуется внести кое-какие настройки. Но теперь, когда ты готов играть, ты также можешь приступать к обучению, поэтому давай писать код!

2

ОСНОВЫ ПРОГРАММИРОВАНИЯ

В этой главе ты познакомишься с основами создания роботов Minecraft, называемых черепашками. Давай создадим нашу первую черепашку и дадим ей кличку. Затем мы «покормим» ее и попробуем выполнить пару программных инструкций. Ты также освоишь некоторые базовые концепции программирования и узнаешь, как практиковаться программировать в игре Minecraft.

ЗНАКОМСТВО С ЧЕРЕПАШКАМИ

Язык программирования Logo, созданный в 1960-е годы, ввел понятие программируемых черепашек. В этом языке черепашки были точками на экране, которые можно было запрограммировать для перемещения и рисования линий, создавая удивительные узоры. Изучая язык Logo, можно научиться программировать, творя шедевры с помощью компьютера! На рис. 2.1 показан узор, сделанный с помощью программы, описанной в книге Брайсона Пэйна «Python для детей и родителей. Играй и программируй» (Эксмо, 2017). Это отличный пример Logo-искусства, написанный на языке программирования Python.

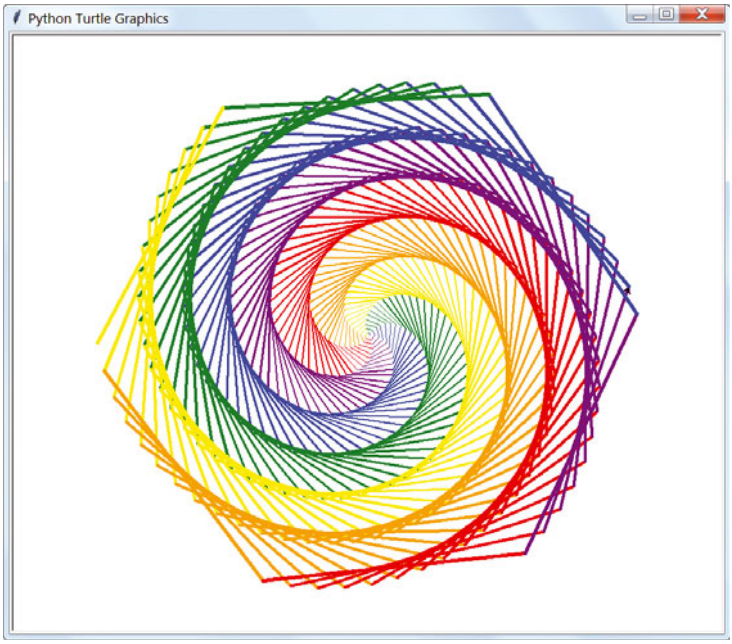


Рис. **2.1.** *Пример красочного черепашьего искусства*

Мод ComputerCraft привносит ту же идею в мир Minecraft, где черепашки – это роботы, которые могут добывать руду и крафтить предметы, слушаясь программ, которые ты напишешь. На рис. 2.2 показано, как выглядят эти блочные роботы-черепашки (также называемые ботами).

Рис. **2.2.** *Черепашка с киркой (слева) и вид черепашки с разных сторон*

Эти роботизированные ящички, на внешний вид, не внушают доверия, но у них большой потенциал. Черепашки могут двигаться в любых направлениях. Они могут парить в воздухе, выживать под водой и даже в лаве, а также

носить предметы. Когда ты освоишь азы программирования, твои черепашки будут выполнять сложные задачи по первой же команде.

СОЗДАНИЕ ЧЕРЕПАШКИ-ШАХТЕРА

Играя в творческом режиме, ты можешь создать черепашку точно так же, как и любой другой предмет или блок. Нажми клавишу **E**, чтобы открыть инвентарь, перейди на вкладку с изображением компаса, а затем набери в строке поиска слово *turtle* (к сожалению, мод не русифицирован), чтобы найти все виды черепашек, как показано на рис. 2.3. В этой книге описана работа с обычными серыми черепашками, а не продвинутыми золотыми. Последние работают аналогично.

Рис. **2.3.** *Поиск черепашек в творческом режиме*

Примечание *Чтобы переключиться в творческий режим, нажми клавишу* **T** *или* **/***, чтобы открыть строку консоли/чата; затем введи команду* `/gamemode creative` *и нажми клавишу* **Enter**. *Чтобы переключиться в режим выживания, нажми клавишу* **T** *или* **/** *и в строке консоли/чата введи команду* `/gamemode survival`.

Но так как мы хотим использовать черепашек в режиме выживания, нужно научиться создавать черепашек (и все остальное) с нуля. Вот как это сделать.

1. Создай компьютер. В рецепте крафта черепашки нужен компьютер, поэтому, прежде чем ты сможешь создать черепашку, нужно скрафтить компьютер из камня, красной пыли[3] и стеклянной панели. (Если ты не знаешь, как сделать камни или стеклянную панель или как получить красную пыль, открой в веб-браузере сайт поисковой системы и введи запрос *minecraft получить красную пыль* или *minecraft получить редстоун* или *minecraft получить стеклянную панель*.) Чтобы создать компьютер, следуй рецепту, показанному на рис. 2.4.

Рис. **2.4.** *Создание компьютера из семи каменных блоков, кучки красной пыли и одной стеклянной панели*

2. Используй компьютер для крафта черепашки. Мы не будем размещать компьютер в игровом мире, потому что он не будет использоваться самостоятельно в этой книге. Нам он нужен как ингредиент крафта в рецепте черепашки. Для создания базовой черепашки нужен компьютер, железные слитки и сундук, как показано в рецепте на рис. 2.5.

Рис. **2.5.** *Создание черепашки из семи железных слитков, одного сундука и одного компьютера*

3. Вооружи черепашку инструментами! На данный момент у нас есть базовая черепашка. Она может перемещаться, но не умеет добывать руду, копать и выполнять другие

[3] В новых версиях игры называется редстоуном *(прим. ред.)*

рабочие действия, пока мы не дадим ей инструмент. Можно вооружить черепашку алмазной киркой, лопатой, топором, мотыгой или мечом. Инструмент или оружие должен быть алмазным и совершенно новым. Изношенные или каменные/деревянные/железные/золотые инструменты не подойдут. Также можно снабдить черепашку верстаком.

Нашу первую черепашку снабдим алмазной киркой, чтобы превратить ее в черепашку-шахтера, как показано в рецепте на рис. 2.6.

Рис. **2.6.** *Крафт черепашки-шахтера из одной простой черепашки и одной алмазной кирки*

ЗАПУСК ПРОГРАММ В ИНТЕРФЕЙСЕ ЧЕРЕПАШКИ

Для запуска черепашки выбери ее в инвентаре и щелкни правой кнопкой мыши, чтобы поместить черепашку в мир так же, как и любой другой блок. Затем щелкни правой кнопкой мыши по установленной черепашке, чтобы открыть ее графический интерфейс (*GUI*), в котором можно запрограммировать черепашку и управлять ее инвентарем.

Учитывай, что игра Minecraft не приостанавливается, когда ты открываешь графический интерфейс, поэтому тебе нужно быть осторожным и следить, чтобы враждебные мобы не подкрались к тебе, пока ты вводишь команды. Рис. 2.7 демонстрирует графический интерфейс черепашки.

Рис. **2.7.** *Графический интерфейс черепашки*

Графический интерфейс состоит из ячеек твоего инвентаря, инвентаря черепашки и оболочки командной строки, в которой ты будешь писать свои программы и вводить команды. Ты можешь хранить несколько экземпляров одного и того же блока или предмета в одной ячейке инвентаря, создавая стек этих блоков или предметов. Большинство блоков или предметов можно складывать до 64 экземпляров в одной ячейке. В инвентаре черепашки 16 ячеек. Ты можешь перетаскивать блоки и предметы из своего инвентаря в черепаший и обратно.

Черепашки хранят в памяти некоторые предустановленные программы, включая `label`, `dance`, `refuel`, `go` и `lua`. Сначала мы должны присвоить нашей черепашке имя с помощью программы `label`. В этой книге я назвал черепашек именами четырех итальянских художниц эпохи Ренессанса: Софонисба, Лавиния, Артемизия и Элизабетта. Ты можешь использовать эти имена или указать любые другие.

Щелкни правой кнопкой мыши по черепашке, чтобы открыть графический интерфейс, и запусти программу `label`, указав в оболочке командной строки название программы `label`, далее команду `set`, чтобы назначить имя

черепашки, а затем имя Sofonisba. Оболочка командной строки не поддерживает русский язык, поэтому команды и имена придется писать на английском языке. Затем нажми клавишу **Enter**:

```
> label set Sofonisba
Computer label set to "Sofonisba"
```

Слова `set Sofonisba` называются аргументами командной строки. Они сообщают программе `label`, что нужно делать. Вместе программа `label` и аргументы командной строки `set Sofonisba` образуют команду, которую должна выполнить черепашка.

Символ > в начале строки кода называется приглашением. Ты вводишь команды в оболочке командной строки после символа >, а после того, как черепашка их выполнила, появляется новый символ >, ожидающий ввода следующей команды.

Внимание ! *Ты можешь поместить черепашку в свой инвентарь, собрав ее с помощью кирки, но делай это только после того, как присвоил черепашке имя. В противном случае черепашка потеряет все свое топливо и удалит все свои программы. (К топливу мы вернемся в следующем разделе.) Собранная черепашка выбрасывает весь свой инвентарь, как и при сборе сундука.*

Теперь, когда с помощью программы `label` мы задали имя черепашке, `Sofonisba`, давай выполним команду `dance`, чтобы запустить еще одну предустановленную программу:

```
> dance
Preparing to get down...
Press any key to stop the groove
```

Нажми клавишу **Esc**, чтобы закрыть интерфейс черепашки. Ты увидишь, что когда программа `dance` выполняется, черепашка в случайном направлении вращается вокруг своей оси. Щелкни правой кнопкой мыши по че-

репашке, чтобы снова открыть ее интерфейс. Программа `dance` все еще выполняется и ожидает остановки нажатием любой клавиши. Ты также можешь завершить любую программу, удерживая сочетание клавиш **Ctrl+T** в течение одной секунды при открытом графическом интерфейсе.

ПЛЯШИ, ПЛЯШИ, ЧЕРЕПАШКА: ПОВТОРНЫЙ ЗАПУСК КОМАНД

Если ты хочешь выполнить программу, которую запускал ранее, то можешь воспользоваться следующим советом. Если нажимать клавиши ↑ и ↓, в оболочке командной строки будут отображаться предыдущие выполненные программы. Например, ты ранее запустил программу `dance`:

```
> dance
```

Если ты хочешь еще раз запустить программу `dance`, тебе не нужно снова вводить ее название с клавиатуры; просто нажми клавишу ↑, и оболочка командной строки отобразит последнюю выполненную команду. Если ты снова нажмешь клавишу ↑, оболочка командной строки продолжит отматывать историю ранее введенных команд, начиная с последней введенной команды. Этот прием также работает и в оболочке Lua, которая описана в разделе «**Освоение языка программирования Lua**» далее в этой главе.

«КОРМЛЕНИЕ» ЧЕРЕПАШКИ

Черепашки могут двигаться вперед, назад, вверх или вниз (и даже летать!). Но чтобы двигаться, их нужно кормить, причем не едой, а… топливом! Любые предметы, которые горят в печи, можно скормить и черепашкам в виде топлива, при этом одна единица топлива позволяет перемещать черепашку на расстояние в один блок. В табл. 2.1 показано, сколько единиц топлива получается из того или иного предмета.

Табл. 2.1. Источники топлива черепашки

Пункт	Название товара	Единицы топлива
	Палка	5
	Деревянные инструменты (кирка, лопата и т.п.)	10
	Древесина	15
	Доски	15
	Каменный или древесный уголь	80
	Стержень ифрита[4]	120
	Угольный блок	800
	Ведро лавы	1000

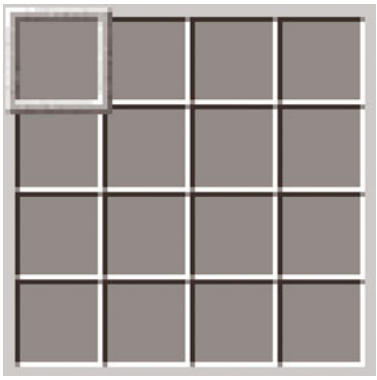
Наибольшее количество единиц топлива предоставляет ведро лавы. Обрати внимание, что древесина и доски обеспечивают 15 единиц топлива. Ты можешь из одного блока древесины сделать четыре блока досок и получить в итоге в четыре раза больше топлива. Проще всего исполь-

[4] В новых версиях игры называется огненным жезлом *(прим. ред.)*

зовать каменный или древесный уголь, который легко добыть и которым можно надолго снабдить черепашек в качестве топлива. Уголь предоставляет 80 единиц топлива, а угольные блоки – целых 800. Поскольку для изготовления угольного блока требуется всего девять кусков угля, эффективнее превращать уголь в угольные блоки перед «скармливанием» черепашке. Стержни ифрита встречаются очень редко (их выбрасывают только побежденные ифриты[5] в Незере) и относительно малоэффективны как источник топлива, поэтому лучше их приберечь для крафта.

Базовая черепашка может хранить вплоть до 20 000 единиц топлива, необходимого для движения. Оно не тратится при повороте черепашки, в процессе добычи, крафта, копания – словом, всего, что делает черепашка. Парение в воздухе также не требует топлива.

Чтобы «накормить» черепашку, топливо надо поместить в ее инвентарь, в текущую ячейку, выделенную толстой рамкой, как показано на рис. 2.8.



После того, как ты поместишь топливо в эту ячейку, запусти программу `refuel`. Если ты поместил более одного экземпляра предмета, используемого в качестве топлива, ты можешь в команде указать количество предметов, которые нужно сжечь, после имени программы `refuel`.

Рис. **2.8.** *Ячейка для топлива в инвентаре черепашки имеет более толстую границу, чем другие ячейки*

Например, введи в оболочке командной строки следующую команду:

```
> refuel 64
Fuel level is 5120
```

[5] В новых версиях игры называются всполохами *(прим. ред.)*

Эта команда запускает программу `refuel` и добавляет в качестве топлива 64 предмета из числа тех, что помещены в ячейку топлива. Количество единиц топлива, которое будет добавлено, зависит от типа топлива в ячейке. Чтобы узнать, сколько топлива заправлено в черепашку, укажи в качестве аргумента `0`:

```
> refuel 0
Fuel level is 5120
```

Эта команда покажет, сколько топлива заправлено в черепашку, без необходимости добавления в качестве топлива предметов из инвентаря.

Внимание! *Если ты задумал собрать черепашку в свой инвентарь с помощью кирки, предварительно присвой черепашке имя командой* `label`*, иначе черепашка потеряет все топливо, которое ты в нее добавил. Обязательно выполни программу* `label`*, чтобы сначала присвоить черепашке имя!*

ПЕРЕМЕЩЕНИЕ ЧЕРЕПАШКИ

Теперь, когда черепашка «сыта», давай переместим ее. Из оболочки командной строки мы можем запустить программу `go`, чтобы приказать черепашке двигаться вперед/назад. В оболочке командной строки введи следующую команду:

```
> go forward
```

После того как ты нажмешь клавишу **Esc**, чтобы закрыть графический интерфейс, черепашка переместится вперед на один блок. Обрати внимание: если перед черепашкой будет стоять блок, преграждающий путь, черепашка будет ждать, пока этот блок не будет удален. Если ты хочешь завершить программу, нажми и удерживай сочетание клавиш **Ctrl+T** в течение одной секунды.

Чтобы вернуть черепашку назад, введи следующую команду:

```
> go back
```

Черепашка вернется обратно на один блок. Введи число после аргумента `forward` или `back` для перемещения черепашки на указанное число блоков, например так:

```
> go forward 2
```

Черепашка сделает два шажка вперед (переместится на два блока).

Поворачивать черепашку можно таким же образом. Укажи слово `left` или `right`, а затем число – сколько раз черепашка должна повернуть налево или направо на 90°. Например, команда `go left` повернет черепашку на 90° налево. Команда `go right 2` повернет черепашку направо дважды, т.е. на 180°.

В одной команде можно объединить несколько движений, например так:

```
> go forward 2 up right forward down back 3 left
```

Черепашка пройдет вперед на два блока, поднимется вверх на один блок, повернет один раз направо (на 90°), пройдет вперед еще на один блок, затем вниз на один блок, пройдет назад на три блока и, наконец, повернет один раз налево.

Существуют сокращения, которые ты можешь использовать в командах `go`: `fd` для слова `forward`, `bk` – `back`, `lt` – `left`, `rt` – `right` и `dn` – `down`. Вышеприведенная команда в кратком виде будет выглядеть так:

```
> go fd 2 up rt fd dn bk 3 lt
```

Эта команда перемещает черепашку точно так же, как и в предыдущем примере, за исключением того, что набирать букв нужно гораздо меньше.

ОСВОЕНИЕ ЯЗЫКА ПРОГРАММИРОВАНИЯ LUA

Запуск команд, которые выполняют черепашки, – удобная штука, но настоящая мощь проявляется, когда ты пишешь свои собственные программы на языке программирования Lua. Язык программирования используется для написания инструкций, понятных компьютеру. Сейчас ты не понимаешь, для чего используются эти инструкции и программистские слова. А это важные элементы языка – строительные блоки для полезных программ, которые ты будешь разрабатывать чуть позже. Мы будем по одной изучать инструкции языка Lua.

ЗАПУСК ОБОЛОЧКИ LUA

У языка Lua есть специальная оболочка *Lua*, которая позволяет вводить и выполнять команды на языке Lua в графическом интерфейсе черепашки. Тебе нужно запустить программу `lua` для доступа к оболочке:

```
> lua
Interactive Lua prompt.
Call exit() to exit.
lua>
```

Эта команда, как показано на рис. 2.9, запускает оболочку Lua, в которой ты будешь вводить все свои инструкции Lua. Оболочка Lua удобна для просмотра результата каждой инструкции, потому что ты можешь вводить команды по очереди, экономя свое время.

Примечание *В этой книге оболочка Lua – это то, где ты вводишь код на языке Lua. В оболочке в начале каждой строки отображается приглашение* `lua>`*, после которого ты можешь вводить свои инструкции на языке Lua. Оболочка командной строки – это то, где ты вводишь команды для запуска программ. В ней более простое приглашение из*

одного символа >. Эти оболочки легко спутать. Если у тебя в игре появляются сообщения об ошибках при выполнении кода, дважды проверь, какую оболочку ты используешь, посмотрев на приглашение.

Рис. **2.9.** *Оболочка Lua в графическом интерфейсе черепашки*

Начнем с простой инструкции, которая поворачивает черепашку налево. Введи следующую инструкцию в оболочке Lua:

```
lua> turtle.turnLeft()
true
```

По мере ввода Lua попытается автоматически завершить инструкцию, которую, по ее мнению, ты собрался напечатать. Ты можешь нажать клавишу **Tab**, чтобы принять предложение оболочки и сэкономить время. На рис. 2.10 я набрал текст `turtle.tu`, и функция автозавершения предлагает использовать команду `turtle.turnLeft(` (нажатие клавиши **Tab** приведет к автоматической подстановке выделенных символов `rnLeft(`, а затем я могу ввести оставшуюся скобку `)`.

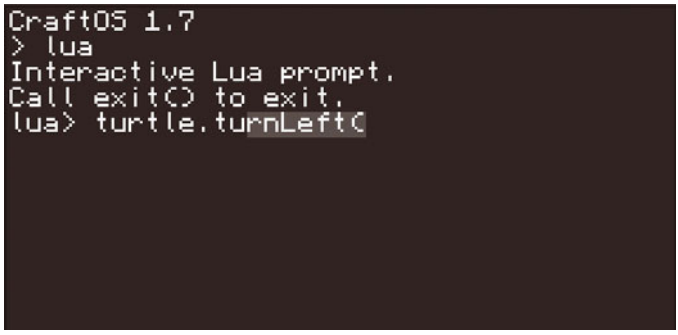


Рис. **2.10.** *Функция автозавершения предлагает использовать команду* `turtle.turnLeft(`*, когда я набираю* `turtle.tu`*.*

Слово `true` (истина), которое появляется после ввода команды, называется возвращаемым значением и позволяет узнать, что черепашка успешно повернула налево. Многие инструкции черепашек вернут значение `true`, если введенная инструкция выполнима и `false` (ложь), если нет. (Значения `true` и `false` называются логическими и объясняются в главе 5.)

После возврата значения `true` снова появится приглашение `lua>`, ожидающее ввода следующей инструкции. Нажми клавишу **Esc**, чтобы увидеть, что черепашка повернулась налево, как показано на рис. 2.11.

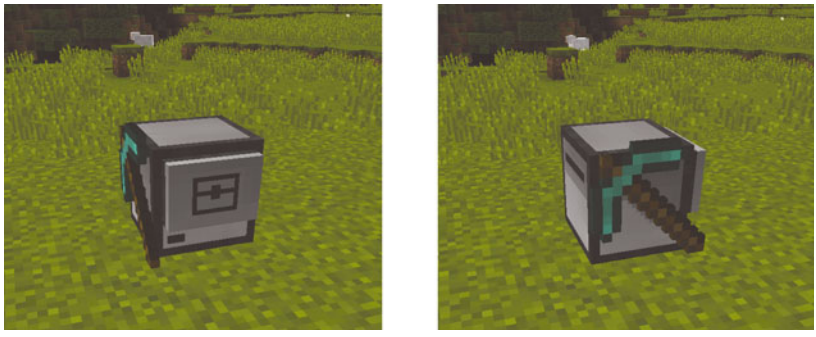


Рис. **2.11.** *Черепашка до и после выполнения инструкции* `turtle.turnLeft()`

Щелкни правой кнопкой мыши по черепашке, чтобы снова отобразить ее интерфейс, и введи следующую инструкцию в оболочку Lua:

```
lua> turtle.turnRight()
true
```

Теперь, когда ты нажмешь клавишу **Esc**, то увидишь, что черепашка повернулась направо и находится в изначальном положении. Инструкции `turtle.turnLeft()` и `turtle.turnRight()` – это функции. В программировании функция является своего рода минипрограммой внутри программы. Функции содержат набор инструкций для выполнения какого-либо действия. Когда функция запускается с помощью инструкции на языке Lua, говорят, что функция вызывается.

Функции `turtle.turnLeft()` и `turtle.turnRight()` находятся в модуле `turtle`. Модуль представляет собой набор функций. Чтобы вызывать функции внутри модуля `turtle`, тебе нужно напечатать слово `turtle` с точкой перед именем функции. Модуль `turtle` содержит много функций, позволяющих черепашке двигаться, добывать руду, копать, размещать блоки и делать все остальное. Ты узнаешь о многих из этих функций и модулей из этой книги!

ЧАО, LUA: ВЫХОД ИЗ ОБОЛОЧКИ LUA

Обрати внимание, что нажатие клавиши **Esc** закрывает графический интерфейс, но в следующий раз, когда ты щелкнешь правой кнопкой мыши по черепашке, ты вернешься к оболочке Lua. Ты можешь выйти из оболочки Lua и вернуться к приглашению оболочки командной строки, вызвав функцию `exit()`. Введи в оболочку Lua следующую команду:

```
lua> exit()
>
```

После выполнения этой функции ты увидишь, что приглашение `lua>` сменилось на приглашение оболочки командной строки `>`.

МАТЕМАТИЧЕСКИЕ ЗАДАЧКИ С LUA

Не волнуйся, если ты считаешь себя далеким от математики, Lua выполнит за тебя все вычисления! Lua умеет решать математические задачи лучше калькулятора, если ты задашь ей правильные инструкции.

Вернись к приглашению Lua и введи следующую команду:

```
lua> 2 + 2
4
```

Мы называем такие математические задачки выражениями. Две цифры `2` в выражении называются значениями. В этом конкретном случае это числовые значения. Знак `+` в выражении называется оператором. Когда ты передаешь эти значения и оператор оболочке Lua в виде выражения, Lua вычисляет результат выражения в виде одного значения, которое является ответом на математическую задачу. Выражение `2 + 2` вычисляется как `4`.

Пробелы в выражениях не важны. Ты можешь не указывать пробелы или ввести столько, сколько хочешь. Введи в оболочку Lua следующие команды:

```
lua> 3+4
7
lua> 3             +   4
7
```

Несмотря на то, что ты можешь набирать столько пробелов, сколько хочется, большинство программистов считают правильным набирать один пробел между значениями и операторами, чтобы код был более комфортным для чтения.

Оболочка Lua поддерживает математические операторы сложения (+), вычитания (–), умножения (*), деления (/) и возведения в степень (^). Введи указанные ниже команды

в оболочке Lua, чтобы увидеть примеры использования перечисленных операторов:

```
lua> 100 + 1
101
lua> 10 - 4
6
lua> 7 * 5
35
lua> 21 / 3
7
lua> 3^2
9
```

Хотя в некоторых из этих операций используются другие символы, а не те, что ты изучал в школе, они работают точно так же.

Значение само по себе также является выражением: оно вычисляется самим собой. Вернись к приглашению Lua и введи следующую команду:

```
lua> 42
42
```

Как видишь, значение `42` само по себе вычисляется в `42`. Выражения всегда вычисляются в одно значение, даже в случаях небольших выражений, таких как `42`.

Выражения могут быть любой длины. Введи в оболочку Lua следующую команду:

```
lua> 1 + 2 + 3 + 4 + 5
15
```

Это выражение содержит пять значений и четыре оператора, и оболочка Lua вычислила его в единственное значение `15`. Lua вычисляет выражение по шагам, слева направо, хотя ты и не видишь все эти шаги на своем компьютере.

ОЙ, ОШИБКА!

Обрати внимание, что в выражениях на умножение в языке Lua используется звездочка `*`, а не значок крестика × и не буква `x`. Если ты попробуешь использовать значок × или букву `x` в качестве оператора умножения, оболочка Lua выдаст ошибку. Смотри:

```
lua> 5 x 5
bios.lua:14: [string "lua"]:1:
unexpected symbol
```

Ошибка означает, что оболочка Lua не смогла понять инструкцию, которую ты ввел.
Убедись, что твоя команда набрана корректно, и повтори ее ввод.
Ошибки в программировании могут сбивать с толку. Если ты не понимаешь, что означает сообщение, ты можешь набрать текст сообщения об ошибке (например, «`unexpected symbol`» в этом примере) в поле ввода на сайте поисковой системы и найти объяснение.

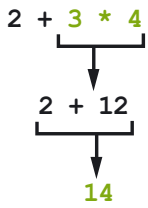
ПОРЯДОК ДЕЙСТВИЙ

Правила, определяющие, какие части выражения вычисляются первыми, называются порядком действий. Порядок действий в Lua такой же, что и в математике. Вычисление производится слева направо, сначала выполняются операции в круглых скобках, затем операции возведения в степень `^`, затем операции умножения и деления `*` и `/`, за которыми следуют операции сложения и вычитания `+` и `-`. В оболочке командной строки введи следующую команду:

```
lua> 2 + 3 * 4
14
```

Выражение `2 + 3 * 4` вычисляется как `14`, а не `20`, потому что сначала вычисляется операция `3 * 4 = 12`, а затем `2 + 12`, что в итоге равно `14`.

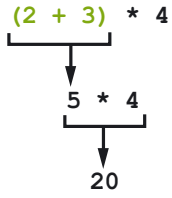
Оболочка Lua вычисляет это выражение по шагам, как показано ниже:



Ты можешь изменить порядок выполнения операций с помощью круглых скобок. Введи в оболочку Lua следующую команду:

```
lua> (2 + 3) * 4
20
```

Теперь сначала выполняется операция `(2 + 3)`, дающая `5`, а затем операция `5 * 4`, приводя в итоге к результату `20`. Lua вычисляет это выражение, как показано ниже:



ГЕНЕРАЦИЯ СЛУЧАЙНЫХ ЧИСЕЛ

Оболочка Lua также умеет генерировать случайные числа. Многие игры и программы используют случайные значения (задумайся, в скольких настольных играх используются случайные числа, генерируемые с помощью игральных кубиков), поэтому тебе нужно уметь создавать и использовать случайные числа. Для этого ты воспользуешься функцией `math.random()`, которая при каждом вызове возвращает случайное число. Обрати внимание, что функция

random() находится внутри модуля math. Введи в оболочку Lua следующие команды:

```
lua> math.random(1, 6)
6
lua> math.random(1, 6)
1
lua> math.random(1, 6)
1
lua> math.random(1, 6)
3
lua> math.random(100, 200)
142
```

Поскольку возвращаемые значения случайны, числа в твоем выводе наверняка будут отличаться от приведенных в этом примере. Обрати внимание, что, когда мы вызываем функцию random(), в круглых скобках указываем два числа через запятую. Эти значения являются аргументами вызова функции. Здесь аргументы – значения, передаваемые функции при ее вызове и сообщающие ей, как себя вести. Когда ты помещаешь аргументы в круглые скобки после имени функции, ты передаешь аргументы вызову функции. Функция random() принимает два аргумента: минимально и максимально допустимое случайное число, которое должен возвращать вызов функции. Если вызову функции ты передаешь значения 1 и 6, возвращаемое число всегда будет находиться в диапазоне от 1 до 6. Если ты передашь 100 и 200, возвращаемое значение будет находиться в диапазоне от 100 до 200.

Во всех случаях, когда тебе нужно добавить случайное значение в свою программу, ты будешь использовать функцию random(). Мы сделаем это в программе mydance, которую напишем в главе 4.

СОХРАНЕНИЕ ЗНАЧЕНИЙ С ПОМОЩЬЮ ПЕРЕМЕННЫХ

Чтобы в программе использовать значения несколько раз, их можно хранить в переменных. Переменная – это словно коробочка в памяти компьютера. В этой коробочке ты можешь сохранить одно значение (см. рис. 2.12). Чтобы создать переменную, ты пишешь имя переменной, затем знак равенства (=), за которым следует значение, которое ты хочешь сохранить в переменной. Например, если ты хочешь записать, сколько угля у тебя есть, ты можешь использовать инструкцию `coal = 10` для хранения значения `10` в переменной с именем `coal`. (К примеру, эта переменная может хранить, сколько угля добыла черепашка.) Такая команда называется инструкцией присваивания, а знак равенства (=) называется оператором присваивания.

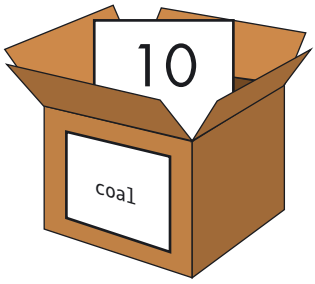


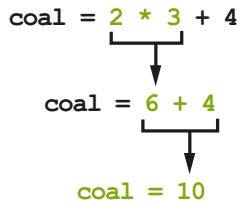
Рис. **2.12.** *Инструкция* `coal = 10` *– это как будто ты сказал программе: «Переменной* `coal` *назначено числовое значение* `10`*»*

Ты можешь использовать переменную в выражениях вместо непосредственного значения. Например, введи в оболочку Lua следующие команды:

```
lua> coal = 10
lua> coal + 4
14
lua> coal + 10
20
lua> 12 - coal + 3
5
```

В первой строке ты объявляешь (т.е. сообщаешь оболочке, что теперь есть переменная с этим именем) переменную `coal` и присваиваешь ей значение `10`. После этого значение `10` подставляется везде, где ты используешь переменную `coal`.

Справа от оператора = ты можешь поместить не только числовое значение, но и выражение, чтобы присвоить переменной результат его вычисления. Результат вычисления будет сохранен в виде значения переменной. Если ввести инструкцию `coal = 2 * 3 + 4`, Lua вычислит значение переменной `coal` как `10`, см. ниже:



Кроме того, по принципу вычисления значения из самого себя, ты можешь посмотреть, какое значение хранится в переменной, введя ее имя в оболочке Lua:

```
lua> coal
10
```

Ранее мы присвоили переменной `coal` число 10, поэтому, когда мы вводим имя переменной `coal`, вычисляется значение `10`.

Давай изменим значение, хранящееся в переменной `coal`, с помощью еще одного оператора присваивания. Эта операция называется перезаписью переменной. Введи в оболочку Lua следующие команды:

```
lua> coal = 100
lua> 2 + coal + 3
105
lua> coal = 200
```

```
lua> 2 + coal + 3
205
```

Переменные могут хранить только одно значение. Если значение переменной перезаписывается новым значением, старое значение отбрасывается и стирается, как показано на рис. 2.13.

Рис. **2.13.** *Переменной* `coal` *присваивается новое значение* `200`*, а старое значение* `100` *стирается*

После присвоения переменной значения ты можешь использовать ее в других инструкциях присваивания. Используя переменную с именем `counter`, введи следующие команды в оболочке Lua:

```
❶ lua> counter = 0
  lua> counter
  0
❷ lua> counter = counter + 1
  lua> counter
❸ 1
❹ lua> counter = counter + 1
  lua> counter
  2
❺ lua> counter = counter + 100
  lua> counter
  102
```

Сначала присвой переменной `counter` значение `0` ❶, а затем назначь ей значение, равное `counter + 1` ❷ В результате значение переменной `counter` теперь равно `1` ❸. Ты можешь продолжать добавлять `1` к значению переменной `counter` ❹, нажимая клавишу ↑, чтобы выводить предыдущие инструкции, как и в оболочке командной строки. Добавление `1` к значению переменной `counter` позволяет использовать эту переменную для подсчета. Этот метод будет полезен в будущих программах, если, к примеру, тебе понадобится посчитать количество выполнений команды. Ты можешь менять значение переменной `counter` столько раз, сколько нужно, и даже можешь изменить его с помощью величины, отличной от `1`, например `100` ❺.

Переменные полезны для хранения результатов вычислений или других данных, которые ты хочешь использовать в программе. Большинство программ, которые ты будешь писать (и почти все программы в этой книге), используют переменные.

ИМЕНА ПЕРЕМЕННЫХ

Имена переменных должны начинаться с буквы или символа подчеркивания и состоять только из латинских букв, цифр и символов подчеркивания. Имена переменных обычно описывают то, что хранят, например `FuelPercentage`, `spaceLeft` или `coal`. Имена переменных зависимы от регистра, т.е. одно и то же имя переменной в разном написании считается различным. Например, `coal`, `Coal` и `COAL` – три разные переменные.

Кроме того, в таблице ниже приведены слова, которые зарезервированы в языке Lua и не могут быть использованы в качестве имен переменных:

`and`	`break`	`do`	`else`	`elseif`
`end`	`false`	`for`	`function`	`if`
`in`	`local`	`nil`	`not`	`or`
`repeat`	`return`	`then`	`true`	`until`

ПРОВЕРКА «СЫТОСТИ» ЧЕРЕПАШКИ

Давай попробуем использовать переменные и выражения в оболочке Lua для вычисления количества угля, необходимого для полного наполнения черепашки топливом. Функция `turtle.getFuelLevel()` возвращает текущий уровень топлива, а функция `turtle.getFuelLimit()` – максимальное количество топлива, которое может хранить черепашка. Введи следующие команды в оболочке Lua:

```
   lua> turtle.getFuelLimit()
   20000
   lua> turtle.getFuelLevel()
   968
❶ lua> spaceLeft = turtle.getFuelLimit() - turtle.get
FuelLevel()
   lua> spaceLeft
   19032
❷ lua> coalNeeded = spaceLeft / 80
   lua> coalNeeded
   237.9
```

Выражение `turtle.getFuelLimit() - turtle.getFuelLevel()` вычисляет объем пространства, оставшегося в топливном баке черепашки Œ. Значение, полученное в итоге, сохраняется в переменной `spaceLeft`. Поскольку один кусок угля равен 80 единицам топлива, количество угля, необходимого для наполнения топливного бака черепашки, можно вычислить по формуле `spaceLeft / 80`. Полученное значение сохраняется в переменной `coalNeeded`.

Как видишь, вызовы функций, выражения и переменные могут быть скомбинированы в удобный код для робота-черепашки. Когда ты узнаешь больше принципов программирования, создаваемый тобой код станет существенно сложнее.

ЧТО МЫ УЗНАЛИ

В этой главе ты узнал, как создавать и управлять черепашками. Роботизированные черепашки создаются так же, как и любой другой предмет в Minecraft, и могут быть оснащены только новыми алмазными инструментами. Чтобы двигаться, черепашки потребляют топливо, такое как древесина и уголь. Щелчком правой кнопкой мыши по черепашке можно открыть ее графический интерфейс, в котором отображается инвентарь черепашки и оболочка командной строки. В оболочке командной строки ты вводишь команды и запускаешь программы. Черепашки уже знают некоторые программы, например `label`, `dance` и `refuel`.

Ты также научился основам программирования на языке Lua в оболочке Lua. Простейший тип инструкций в языке Lua – выражения, такие как `2 + 2`. Выражения состоят из значений (например, `2`) и операторов (например, `+`) и вычисляются до единого значения.

Язык Lua также включает в себя функции, которые можно представить как мини-программы внутри программы на языке Lua. Вызывая эти функции, можно выполнять простые действия, например поворачивать черепашку или генерировать случайные числа. Вызовы функций приводят к вычислению, или возврату значений.

Значения могут храниться в переменных, которые в дальнейшем можно использовать в программе; код практически всех программ содержит переменные. Для объявления переменных и сохранения в них значений используется оператор присваивания `=`, например так: `coal = 10`.

Хотя концепции программирования, о которых ты узнал в этой главе, могут показаться скучными в сравнении с крафтом замечательных роботов, это необходимые строительные блоки программного обеспечения. В главе 3 ты воспользуешься полученными знаниями и напишешь настоящие программы!

3

БЕСЕДЫ С ЧЕРЕПАШКОЙ

Они живые! В главе 2 ты создал своего первого робота-черепашку, накормил ее и оживил. Твое робототехническое создание может похвастаться телом, но, к сожалению, обделено мозгами. Мозг черепашки из игры Minecraft – это программа, которая указывает черепашке, что ей делать. Без запущенной программы твой робот стоит как истукан. Давай слегка очеловечим робота, написав программу, которая позволит нам общаться с ним.

УЧИМ ЧЕРЕПАШКУ ЗДОРОВАТЬСЯ

Мы воспользуемся встроенной в черепашку программой `edit`, с помощью которой создадим новый файл и напишем нашу первую программу. Такие программы, как `edit`, которую мы запустим для написания других программ, называются текстовыми редакторами. Щелкни правой кнопкой мыши по черепашке, чтобы открыть ее графический интерфейс, а затем введи в оболочке командной строки следующую команду:

```
> edit hello
```

Каждый раз, когда ты хочешь создать или открыть файл программы, тебе нужно написать ключевое слово `edit`, за которым следует имя файла. Код `edit hello` запускает программу `edit` и создает, а затем открывает для редактирования файл с именем `hello`. В этом файле мы напишем инструкции нашей программы. Все инструкции будут выполняться одна за другой, по очереди, так же, как и в том случае, когда мы вводили их в оболочке Lua в главе 2. Программа `hello` отобразит на экране слова `Hello, world!` — традиционное приветствие программистов всех стран.

Несмотря на то, что экран текстового редактора кажется крохотным, ты можешь вводить длинные инструкции в одной строке. Чтобы перемещать мигающий курсор на другие строки, используй клавиши клавиатуры, перечисленные в табл. 3.1.

Табл. **3.1.** *Клавиши клавиатуры, используемые в текстовом редакторе*

Клавиша клавиатуры	Действие
←, ↑, ↓ и →	Перемещает курсор в направлении стрелки на клавише
PgUp, **PgDn**	Перемещает курсор вверх или вниз сразу на несколько строк
Backspace	Удаляет текст после курсора
Del	Стирает текст перед курсором
Home	Перемещает курсор к началу строки
End	Перемещает курсор в конец строки
Tab	Автоматически завершает набираемую инструкцию

Инструкции программы называются исходным кодом. Введи следующий исходный код программы `hello` в редактор; только не вводи номера строк, точки после них и пробелы в начале каждой строки. Номера строк используются для удобства указания на отдельные строки кода примеров. Я по строкам объясню код в этой главе, поэтому не беспокойся, если не понимаешь действие каждой строки.

hello

```
1. print('Hello, world!')
2. print('I am ' .. os.getComputerLabel())
3. print('What is your name?')
4. name = io.read()
5. textutils.slowPrint('Nice to meet you, ' .. name)
```

Когда ты закончишь ввод кода, нажми клавишу **Ctrl**, чтобы открыть строку меню сохранения/выхода в нижней части оболочки. Программа `edit` с открытым меню будет выглядеть так, как показано на рис. 3.1.

Рис. **3.1.** *Код программы* `hello`*, набранный в редакторе*

Команда **[Save]** окружена скобками, демонстрируя, что выделена. Нажми клавишу **Enter**, чтобы сохранить файл `hello` в черепашке. Позднее ты можешь открыть, изменить и запустить программу `hello`. Теперь снова нажми клавишу **Ctrl**, чтобы вновь открыть меню сохранения/выхода. Нажми клавишу →, чтобы выделить пункт **[Exit]**, а затем нажми клавишу **Enter**, чтобы закрыть файл и текстовый редактор и вернуться к оболочке командной строки.

ЗАПУСК ПРОГРАММЫ HELLO

В оболочке командной строки ты можешь запустить программу `hello` точно так же, как ты запускал программы `label`, `dance` и `refuel`. Просто введи слово `hello` в оболочке командной строки и нажми клавишу **Enter**:

```
> hello
Hello, world!
I am Sofonisba
What is your name?
```

МОЯ ПРОГРАММА НЕ РАБОТАЕТ!

Если при запуске программы у тебя возникла ошибка `hello:1: attempt to concatenate string and nil`, ты не назначил своей черепашке имя с помощью команды `label`. В строке оболочки после приглашения > выполни команду `label set Sofonisba`, чтобы твоя черепашка откликалась на имя `Sofonisba`. (Ты также можешь использовать любое другое имя на латинице.)

Черепашка поприветствует весь мир, представится и спросит, как тебя зовут. После ввода имени (на английском языке), например `Vasya`, программа будет выглядеть так:

```
> hello
Hello, world!
I am Sofonisba
What is your name?
Vasya
Nice to meet you, Vasya
```

Когда программа завершится, оболочка вновь отобразит приглашение > в ожидании запуска новой программы.

Программное обеспечение, которое запускает программы на языке программирования Lua, называется интерпретатором *Lua*. Я пишу сокращенно, без слова «интерпретатор», просто Lua, ссылаясь как на язык програм-

мирования, так и на программное обеспечение, которое запускает код, написанный на этом языке. Позиция в коде, где в настоящее время интерпретатор Lua выполняет инструкции, называется исполнением. Оно всегда начинается с первой строки программы. Говорят, что интерпретатор исполняет (или выполняет) инструкцию в строке 1. Затем он перемещается ниже и исполняет следующую команду или строку исходного кода.

Посмотри на выведенный на экран программой текст, а затем взгляни на исходный код программы `hello`. Прежде чем я объясню, как она работает, попробуй самостоятельно выяснить, что делает каждая строка кода. Попробуй изменить строку `print('Hello, world!')` на `print('Privet!')`. Затем сохрани программу, выйди из редактора и снова запусти программу `hello`, чтобы узнать, что изменилось. Это верные шаги на пути к тому, чтобы стать крутым программистом: угадай, что делает код, определи правильность своего предположения, выясни, почему твоя догадка неверна (если это так), и поэкспериментируй с кодом, чтобы узнать, что меняется.

ПРОСМОТР СПИСКА ФАЙЛОВ С ПОМОЩЬЮ КОМАНДЫ LS

Если ты забыл, какие файлы хранятся в черепашке, ты можешь выполнить команду `ls` (или `list`, обрати внимание, что это строчная буква `l`, а не единичка `1`), чтобы вывести их имена в виде списка. В оболочке командной строки введи следующую команду:

```
> ls
rom
hello
```

Команда `ls` выводит список файлов, в том числе программ и папок, которые содержат другие файлы. В папке `rom` сохранены разные файлы, которые предустановлены в памяти черепашки, но мы не будем использовать их в этой книге.

ВЫВОД ТЕКСТА С ПОМОЩЬЮ ФУНКЦИИ PRINT()

Давай рассмотрим программу `hello`, изучив каждую строку. Первая строка выглядит следующим образом:

hello

```
1. print('Hello, world!')
```

Эта строка представляет собой вызов функции `print()`, которая выводит на экран текст `Hello, world!`. Ты можешь поэкспериментировать с функцией `print()` в оболочке Lua. Запусти программу `lua` и введи следующую команду:

```
lua> print('Hello, world!')
Hello, world!
1
```

Когда тебе нужно определить, что делает тот или иной фрагмент кода, ты можешь ввести его для проверки в оболочку Lua. Как видишь, функция `print()` занимается выводом текста на экран. `1` в конце – это возвращаемое значение функции `print()`, сообщающее о количестве строк текста, которые были отображены. `1` не будет отображаться при вызове функции `print()` в программе, подобной `hello`. Это значение отображается только здесь, потому что оболочка Lua выводит возвращаемые значения всех функций.

Вспомни, в главе 2 мы говорили о том, что функция `math.random()` при вызове принимает два значения внутри круглых скобок. Эти аргументы сообщают функции диапазон случайных чисел, одно из которых функция должна вернуть. Функция `print()` также принимает аргумент, который в этом случае является текстом `'Hello, world!'`. Такие текстовые значения называются строковыми или просто строками, о чем мы поговорим далее.

СТРОКИ

Ты уже использовал в своих программах числовые значения, например `2` и `10`, но также тебе понадобится использовать и буквы. Строка – это тип значения, который содержит текст вместо цифр. Подобно числовым значениям, строки могут храниться в переменных и использоваться в выражениях. Строки и числа – это типы данных этих значений. Ты узнаешь о других типах данных, например логических и `nil`, в главе 5.

Поскольку строки состоят из букв, которые могут быть похожи на команды языка Lua, интерпретатор Lua должен знать, когда начинается и заканчивается строковое значение, чтобы не путать его с исполняемым кодом на языке Lua. Строки начинаются и заканчиваются одинарной кавычкой ', как мы это писали в значении `'Hello, world!'`. Одиночная кавычка не является частью текста; она просто отмечает, где строка начинается и заканчивается в исходном коде. Ты также можешь использовать и двойные кавычки ", например, так: `"Hello, world!"`.

Можно использовать пустое строковое значение без какого-либо текста между двумя кавычками, так `''` или так `""`. Такое значение называется пустой строкой.

СОЕДИНЕНИЕ СТРОК С ПОМОЩЬЮ КОНКАТЕНАЦИИ

Давай взглянем на строку 2 в исходном коде программы `hello`:

hello

```
2. print('I am ' .. os.getComputerLabel())
```

Строка 2 содержит вызов функции `print()`, но значение, которое мы передаем функции `print()`, включает некоторый незнакомый код на языке Lua. Символы `..` называются оператором конкатенации строк. Он ведет себя как оператор сложения `+` для чисел, только предназначен для

строк. Как оператор + может сложить два числа вместе в новое число, оператор .. может конкатенировать, или объединить, два строковых значения в одну строку. Введи в оболочку Lua следующую команду:

```
lua> 'Hello,' .. 'world!'
Hello,world!
```

Эта строка кода является выражением, так же, как 2 + 2. 'Hello,' и 'world!' – значения, а .. – оператор. Как и все выражения, 'Hello,' .. 'world!' вычисляется в одно значение, которое в этом случае становится строкой Hello,world!.

Интерпретатор Lua не добавляет пробелы, когда ты конкатенируешь строки. Поскольку ни одна из двух строк в нашем примере не содержит пробел, между словами в результирующей строке тоже нет пробела. Если тебе нужно указать пробел между словами в твоем тексте, ты должен добавить его к одному из строковых значений. Например, так 'Hello, ' .. 'world!' или так 'Hello,' .. ' world!', чтобы в результате получить Hello, world!.

В строке 2 программы конкатенируются две строки, одна из которых содержит значение 'I am ', а вторая – значение, возвращаемое новой функцией с именем os.getComputerLabel(). Я расскажу, как работает эта функция, в следующем разделе.

ИЗВЛЕЧЕНИЕ ИМЕН ЧЕРЕПАШЕК

В главе 2 ты научился присваивать черепашкам имена, используя программу label. Твои программы могут извлекать эти имена в виде строковых значений с помощью функции os.getComputerLabel(). Вызовы функции могут быть частью выражения, точно так же, как и значения, потому что вызов функции приводит к вычислению возвращаемого ею значения. Введи следующие команды в оболочку Lua, чтобы разобраться, как работает функция os.getComputerLabel():

```
lua> os.getComputerLabel()
Sofonisba
lua> turtleName = os.getComputerLabel()
lua> print(turtleName)
Sofonisba
1
```

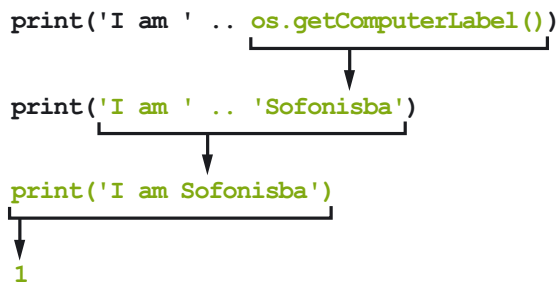
Вызов функции приводит к вычислению строкового значения в виде имени черепашки, поэтому команда `turtleName = os.getComputerLabel()` приводит к тому же результату, что и `turtleName = ,Sofonisba'`.

Еще раз взгляни на строку 2 в программе `hello`:

hello

```
2. print('I am ' .. os.getComputerLabel())
```

Выражение `'I am ' .. os.getComputerLabel()` конкатенирует имя черепашки со строкой `'I am '`, чтобы получить одну строку. Эта единая строка передается функции `print()`, поэтому при запуске программы на экране появляется текст `I am Sofonisba`. Строка 2 – это просто выражение, которое вычисляется в одно значение, как показано ниже:



Помни, что функция `print()` возвращает значение количества отображенных строк, поэтому оно равно `1`. Строковое значение, которое выводит функция `print()`, не является его возвращаемым значением.

ОБРАБОТКА ВВОДА С КЛАВИАТУРЫ ПРИ ПОМОЩИ ФУНКЦИИ IO.READ()

Взгляни на строки 3 и 4 в программе `hello`:

hello

```
3. print('What is your name?')
4. name = io.read()
```

Строка 3 содержит еще один вызов функции `print()`. А в строке 4 используется новая функция – `io.read()`.

При вызове функция `io.read()` приостанавливает исполнение программы, ожидая, пока пользователь не наберет что-либо на клавиатуре и не нажмет клавишу **Enter**. Введенный пользователем текст возвращается из функции `io.read()` в виде строки, которая присваивается переменной `name`.

Если пользователь ввел имя `Vasya`, переменной будет присвоено строковое значение `'Vasya'`.

ДОПОЛНИТЕЛЬНОЕ ЗАДАНИЕ: КРАТКО О СЕБЕ

Ты написал программу приветствия, которая позволяет сказать черепашке твое имя, но ты также можешь рассказать черепашке о себе немного больше. Напиши программу, в которой черепашка спрашивает пользователя не только как его зовут, но и сколько ему лет и какая его любимая компьютерная игра. Сохрани каждый из полученных ответов в отдельную переменную, а затем используй функцию `print()` и оператор `..`, чтобы показать их пользователю.

ТЕКСТ С ЭФФЕКТОМ ПИШУЩЕЙ МАШИНКИ

Возможно, ты заметил, что, когда ты запустил программу `hello`, текст в последней строке выводился по буквам, как будто он медленно печатается на пишущей машинке. Такой эффект достигается с помощью функции `textutils.slowPrint()`:

```
hello
5. textutils.slowPrint('Nice to meet you, ' .. name)
```

Аргумент функции `slowPrint()` – это строка `'Nice to meet you, '`, конкатенируемая со строковым значением переменной `name`, так как программа приветствует пользователя и запрашивает у него имя. Строка 5 – последняя строка исходного кода, поэтому программа прекращает исполнение (завершает работу) после ее выполнения.

ПЕРЕИМЕНОВАНИЕ ЧЕРЕПАШЕК

Ты можешь изменить имя черепашки, вызвав функцию `os.setComputerLabel()` и передав ей строковое значение. Введи следующие команды в оболочке Lua.

```
lua> os.setComputerLabel('Elisabetta')
lua> os.getComputerLabel()
Elisabetta
```

Теперь твою черепашку зовут `Elisabetta`. Ты можешь переименовывать свою черепашку хоть каждый день, используя функцию `os.setComputerLabel()`.

ДОПОЛНИТЕЛЬНОЕ ЗАДАНИЕ: ПЕРЕИМЕНОВАНИЕ ЧЕРЕПАШЕК

Попробуй написать собственную версию программы `label`. Ты можешь выполнить команду `edit mylabel` для создания нового файла. Вызови в программе функции `print()` и `io.read()` и спроси игрока, каким именем следует назвать черепашку, а затем передай это имя функции `os.setComputerLabel()`.

ЧТО МЫ УЗНАЛИ

В этой главе мы создали программу, выводящую на экран текст `Hello, world!` – классическое приветствие начинающих программистов со всего мира. Эта программа умеет

выводить текст (с помощью функций `print()` и `textutils.slowPrint()`) и вводить его (с помощью функции `io.read()`). В языке Lua текст используется в виде строковых значений (или просто строк), которые могут быть сохранены в виде значений переменных и использованы в выражениях, точно так же, как и числовые значения.

Строковые значения строк также поддерживают использование операторов, как и числа. Оператор конкатенации `..` объединяет две строки для формирования новой единой строки.

Программа `hello` в этой главе была всего лишь первым шагом в написании программ для твоих роботов. В главе 4 ты узнаешь еще несколько приемов программирования и научишь черепашку не только двигаться, но и плясать! Погнали!

4

И ПУСТЯТСЯ ЧЕРЕПАШКИ В ПЛЯС!

Мы научили черепашек говорить. Давай научим их ходить. Или, еще лучше, давай научим их танцевать! Хотя и существует готовая программа с названием `dance`, она слишком примитивна и просто передвигает черепашку случайным образом. В этой главе мы создадим гораздо более интересную программу – танцующей черепашки. Танцевальные па черепашки будут выглядеть так, как показано на рис. 4.1.

КОД ПРОГРАММЫ ТАНЦУЛЕК

Используя текстовый редактор, создай программу с именем `mydance`, выполнив команду `edit mydance` в оболочке командной строки. В текстовом редакторе введи показанные ниже строки кода. Не забудь, что номера строкуказывать не нужно. В книге они для удобства, чтобы можно было обращаться к строкам по их номерам.

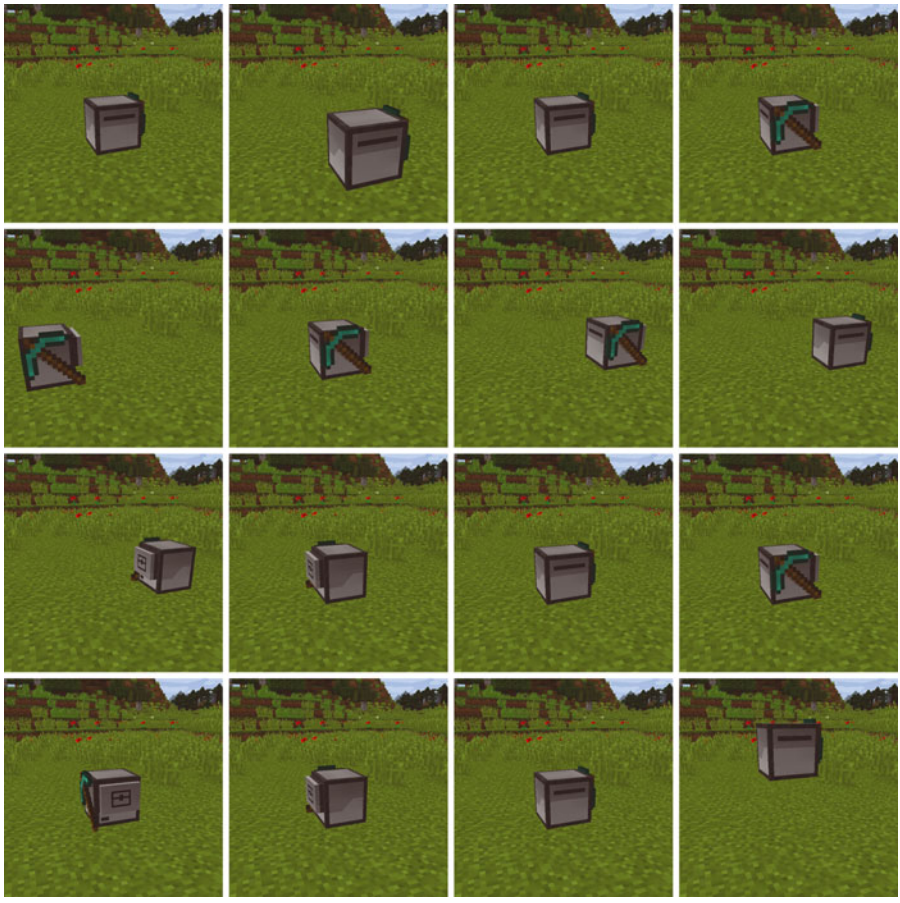


Рис. **4.1.** *Танцевальные па черепашки*

mydance

```
 1. --[[Программа танцулек Эла Свейгарта
 2. Make the turtle dance!]]
 3.
 4. print('Time to dance!')
 5.
 6. -- Черепашка начинает танцевать
 7. turtle.forward()
 8. turtle.back()
 9. turtle.turnRight()
10. turtle.forward()
11. turtle.back()
12. turtle.back()
```

```
13. turtle.turnLeft()
14. turtle.turnLeft()
15. turtle.back()
16. turtle.turnRight()
17.
18. -- Черепашка крутится
19. for i = 1, 4 do
20.   turtle.turnRight()
21. end
22.
23. turtle.up()
24. turtle.down()
25. print('Done.')
```

Сохрани программу после ввода всех инструкций. Нажав клавишу **Ctrl**, открой меню. Выбери пункт **[Save]** и нажми клавишу **Enter**. Затем выйди из редактора, нажав клавишу **Ctrl**, выбрав пункт **[Exit]** и нажав клавишу **Enter**.

ЗАПУСК ПРОГРАММЫ MYDANCE

После выхода из редактора запусти программу `mydance` в оболочке командной строки:

```
> mydance
Time to dance!
Done.
```

На экране появится текст `Time to dance!`. Когда робот начнет танцевать, на экране появится текст `Done`. Чтобы рассмотреть маленький танец черепашки, нажми клавишу **Esc** сразу после запуска программы, до того как появится текст `Done`. Все па, которые совершает черепашка, – это именно те движения, которые ты запрограммировал! Рис. 4.2 демонстрирует, как черепашка движется, слушаясь функций `turtle.forward()` и `turtle.back()`, которые вызываются в строках 7 и 8.

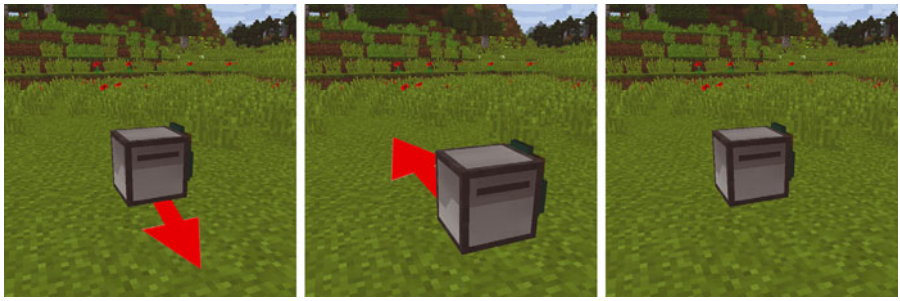


Рис. **4.2.** *Первые два танцевальных па черепашки*

Давай рассмотрим каждую инструкцию в коде программы `mydance`.

ИСПОЛЬЗОВАНИЕ КОММЕНТАРИЕВ В КОДЕ

Первые две строки в программе содержат комментарии. Интерпретатор Lua игнорирует комментарии, потому что это, по сути, заметки программиста. В данном комментарии описывается, что делает программа и кто ее автор.

Примечание *Обрати внимание, что в листингах, приведенных в книге, комментарии набраны на русском языке, чтобы тебе было понятно, за что отвечает та или иная часть кода. При работе с Lua использовать русский язык у тебя не получится.*

mydance

```
1. --[[Программа танцулек Эла Свейгарта
2. Make the turtle dance!]]
3.
4. print('Time to dance!')
5.
6. -- Черепашка начинает танцевать
```

Комментарий, который начинается в строке 1, продолжается в строке 2. Это многострочный комментарий, поскольку располагается более чем на одной строке. Мно-

гострочные комментарии начинаются с символов `--[[` и завершаются символами `]]`.

Строка 4 содержит вызов функции `print()`, которая выводит на экран текст `Time to dance!`. Именно этот текст появляется при запуске программы. С функцией `print()` ты уже встречался в главе 3.

Строка 6 – это тоже комментарий, только однострочный. Однострочный комментарий начинается с символов `--` и заканчивается в конце строки, то есть он занимает только одну строку. Интерпретатор Lua игнорирует эту строку, потому что это просто текст, который поясняет, что будет происходить далее. Это примечание для людей, которые будут читать код программы.

ТАНЦЕВАЛЬНЫЕ ПА ЧЕРЕПАШКИ

После строки 6 следует несколько строк кода с функциями, вызывая которые, программа `mydance` заставляет черепашку двигаться.

mydance

```
 6. -- Черепашка начинает танцевать
 7. turtle.forward()
 8. turtle.back()
 9. turtle.turnRight()
10. turtle.forward()
11. turtle.back()
12. turtle.back()
13. turtle.turnLeft()
14. turtle.turnLeft()
15. turtle.back()
16. turtle.turnRight()
```

В главе 2 ты уже познакомился с функциями `turtle.turnLeft()` и `turtle.turnRight()`. Но есть еще четыре функции для перемещения черепашки, которые ты пока не знаешь: `turtle.up()`, `turtle.down()`, `turtle.forward()` и `turtle.back()`.

ЭКСПЕРИМЕНТИРУЕМ С ПЕРЕМЕЩЕНИЯМИ ЧЕРЕПАШКИ

Давай поэкспериментируем с этими функциями, предназначенными для движения. Убедись, что ты находишься в оболочке Lua (вместо приглашения > должно быть указано `lua>`), и выполни следующие инструкции:

```
lua> turtle.up()
true
lua> turtle.down()
true
lua> turtle.forward()
true
lua> turtle.back()
true
```

Не забывай перед указанием направления добавлять слово `turtle.` и убедись, что у черепашки есть топливо. Функции `turtle.up()` и `turtle.down()` перемещают черепашку вверх, в воздух, и возвращают вниз, на землю. Функции `turtle.forward()` и `turtle.back()` перемещают черепашку вперед, то есть в направлении, в котором она «смотрит», и назад, то есть в противоположном направлении. Эти движения показаны на рис. 4.3.

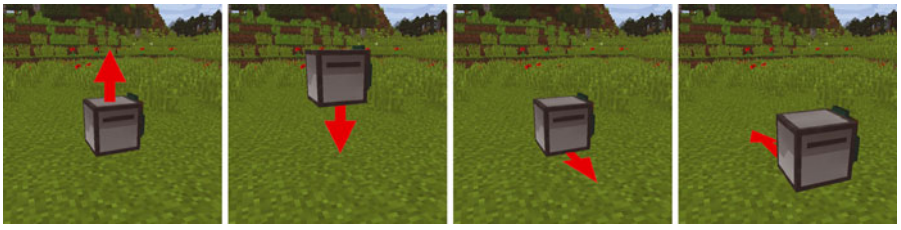


Рис. **4.3.** *Эксперименты с функциями движения черепашки*

Если черепашка перемещается в направлении, которое требует функция движения, то эта функция возвращает значение `true`. (`true` (Истина) и `false` (Ложь) – это логические (или булевы) значения, которые мы подробно рассмотрим в главе 5.)

Поставь черепашку на землю, а в оболочке Lua вызови функцию `turtle.down()`, например, так:

```
lua> turtle.down()
false
Movement obstructed
```

Поскольку на пути черепашки находится блок, она не сможет переместиться в требуемом направлении, тогда вызов функции `turtle.down()` вернет значение `false` и выведет на экран строку с сообщением об ошибке. Если у черепашки нет топлива (она «проголодалась»), передвижения также не произойдет, но отобразится другое сообщение:

```
lua> turtle.forward()
false
Out of fuel
```

Таким образом функции движения информируют о причине, по которой движение невозможно. Создавая программы, описанные в этой книге, ты будешь часто вызывать функции движения для перемещения черепашек.

ЦИКЛЫ: ЭФФЕКТ ЗАЕВШЕЙ ПЛАСТИНКИ

До сих пор исполнение программы `mydance` осуществлялось способом, который называется потоком исполнения: исполнение программы начиналось с первой строки исходного кода и последовательно перемещалось вниз, выполняя каждую строку. Однако несколько инструкций можно объединить в группу, чтобы повторить исполнение этой группы инструкций несколько раз – циклически. Для этого существуют специальные инструкции. Такие инструкции, изменяющие порядок исполнения программы – последовательного исполнения строки за строкой, – называются инструкциями управления потоком.

Чтобы понаблюдать за работой одной из таких инструкций, называемой циклом *for*, в оболочке Lua веди следующую строку кода:

```
lua> for i = 1, 4 do print(i) end
1
2
3
4
```

Введенная тобой инструкция и есть инструкция цикла `for`. Такой цикл может несколько раз повторить исполнение указанной группы команд. Последовательность команд, подлежащая исполнению, располагается между ключевыми словами `do` и `end` и называется блоком, а каждое исполнение блока называется итерацией. В нашем примере, блок состоит всего из одной функции `print(i)`. (Программисты в циклах часто используют именно переменную `i`. Ее имя происходит, скорее всего, от слова «iteration», что означает «итерация».) Блок может состоять не из одной, а из нескольких строк кода, но в нашем примере этот блок прост и содержит вызов только одной функции. Цикл совершает над функцией `print(i)` итерации от `1` до `4`, то есть просто вызывает эту функцию четыре раза подряд.

В итоге цикл `i = 1, 4 do print(i) end` последовательно выводит на экран числа от `1` до `4`. В начале исполнения цикла интерпретатор Lua присваивает переменной `i` значение `1`, превращая вызов в код `print(1)`, и отображает на экране `1`. Это первая итерация цикла `for`. Далее следует инструкция конца блока – `end` – и исполнение снова возвращается к началу блока. На второй итерации значение переменной `i` увеличивается на `1`, и на экране отображается число `2`. Исполнение кода продолжается до тех пор, пока переменная `i` не станет равна `4`, то есть числу после запятой. Это будет четвертая и последняя итерация цикла `for`.

В таком цикле можно использовать любые числа. Например, чтобы организовать цикл от `10` до `13`, в оболочке Lua введи следующий код:

```
lua> for i = 10, 13 do print(i) end
10
11
12
13
```

Если указать третье число, которое называется шагом цикла, то переменная цикла `i` на каждой итерации будет увеличиваться не на `1`, а на указанный шаг. Например, попробуй в оболочке Lua выполнить следующую инструкцию:

```
lua> for i = 10, 20, 2 do print(i) end
10
12
14
16
18
20
```

В этой строке кода шаг равен `2`, поэтому интерпретатор Lua будет выводить все четные числа в диапазоне от `10` до `20`.

В качестве шага можно использовать и отрицательные числа, тогда переменная цикла будет не увеличиваться, а уменьшаться, то есть отсчет будет производиться в обратную сторону от первоначального значения. Следующий код демонстрирует действие шага `-1`:

```
lua> for i = 4, 1, -1 do print(i) end
4
3
2
1
```

Этот код производит обратный отсчет в диапазоне от `4` до `1`. Попробуй изменить шаг на `-2` и посмотри, что произойдет. Циклы – это простой способ выполнить несколько раз группу команд, без необходимости многократно набирать одни и те же команды. В главе 5 ты узнаешь и о других инструкциях управления потоком.

ВРАЩЕНИЕ ЧЕРЕПАШКИ

Вернемся к программе `mydance`. Строки с 19 по 21 содержат цикл `for`, указывающий черепашке совершить поворот вокруг своей оси.

mydance

```
18. -- Черепашка крутится
19. for i = 1, 4 do
20.   turtle.turnRight()
21. end
```

При написании циклов `for`, в каждой строке кода, из которого состоит блок цикла, обычно используется отступ (то есть дополнительные пробелы в начале строки). Такие отступы облегчают понимание, какие именно строки находятся внутри блоков, особенно если внутри одного блока создаются другие (в главе 5 мы сделаем это). Чтобы видеть, где цикл начинается и где заканчивается, ключевое слово `end` размещают на том же уровне отступа, что и ключевое слово `for`.

В приведенном примере строка 20, код которой поворачивает черепашку направо, выполняется четыре раза. Таким образом, черепашка разворачивается на 360 градусов. Но если в строке 19 указать значения `i = 1, 8`, черепашка совершит два полных оборота, как показано на рис. 4.4.

Рис. **4.4.** *Черепашка выполняет два оборота, поворачиваясь направо восемь раз*

ПРЫЖОК НА МЕСТЕ

Наконец, в строках 23 и 24 черепашка совершит прыжок на месте, то есть поднимется вверх, а затем опустится:

mydance

```
23. turtle.up()
24. turtle.down()
25. print('Done.')
```

Такой прыжок изображен на рис. 4.5. Последняя строка выводит на экран слово `Done.`. Строка 25 последняя, ниже нет исполняемого кода, и программа завершается.

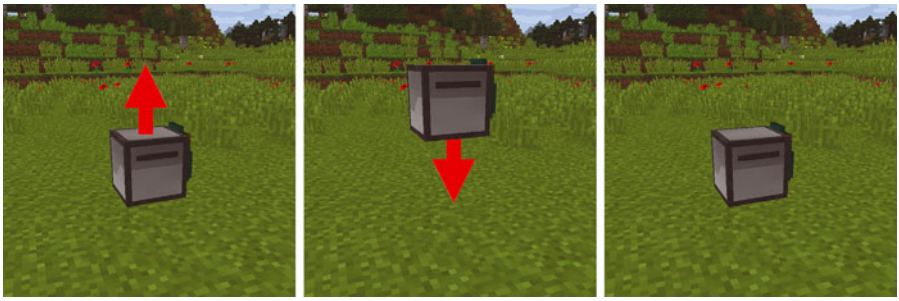


Рис. **4.5.** *Черепашка совершает прыжок, когда вызываются функции* `turtle.up()` *и* `turtle.down()`.

ДОПОЛНИТЕЛЬНОЕ ЗАДАНИЕ: НОВЫЙ ТАНЕЦ

Попробуй создать для черепашки свои собственные танцевальные движения. Для этого ты можешь использовать функции перемещения и поворота, научить черепашку прыгать, крутиться и танцевать в соответствии с твоей собственной хореографической программой!

ПУБЛИКАЦИЯ И ЗАГРУЗКА ПРОГРАММ В ИНТЕРНЕТЕ

После того, как ты написал программу для своей черепашки, ты можешь поделиться ею со своими друзьями. Чтобы загружать свои программы в Интернет, ты можешь воспользоваться сайтом *pastebin.com*, где пользователи общаются онлайн и обмениваются исходными кодами программ, размещая их на сайте. Ты тоже можешь загружать коды своих программ на сайт **pastebin.com**, используя программу `pastebin`, хранящуюся в памяти черепашки. В оболочке командной строки выполни показанную ниже команду, чтобы загрузить программу `mydance` на сайт **pastebin.com**:

```
> pastebin put mydance
Connecting to pastebin.com... Success.
Uploaded as
https://pastebin.com/BLCJbpQJ
```

```
Run «pastebin get BLCJbpQJ» to download
anywhere
```

После загрузки программы для нее будет сгенерирован уникальный веб-адрес. Когда я загружал свою программу с помощью команды `pastebin`, с сайта пришло сообщение, что программа `mydance` была успешно загружена по адресу `https://pastebin.com/BLCJbpQJ/`. После этого любой желающий может просмотреть мою программу, перейдя по этому адресу в своем браузере. Ты тоже можешь загрузить мою программу в свою черепашку, введя в оболочке командной строки следующую команду:

```
> pastebin get BLCJbpQJ mydance
Connecting to pastebin.com... Success.
Downloaded as mydance
```

Строка `get BLCJbpQJ mydance` сообщает сайту **pastebin.com**, что ты хочешь получить программу, сохраненную по адресу **https://pastebin.com/BLCJbpQJ/**, и сохранить ее в памяти черепашки под именем `mydance`. Теперь у тебя есть способ загружать файлы, чтобы делиться ими с другими программистами.

Если команда `pastebin` при выполнении отображает сообщение `Connecting to pastebin.com... Failed.`, проверь подключение к Интернету и корректность указанного аргумента команды (значения `BLCJbpQJ` или любого другого адреса, назначенного сайтом). Если команда `pastebin` выводит сообщение `File already exists`, т.е. файл уже существует, то существующую программу `mydance` сначала следует удалить. Как? Читай дальше.

Удаление файлов из памяти черепашки

Когда ты запускаешь программу `pastebin`, ты сообщаешь ей, что хочешь загрузить файл и сохранить его под именем `mydance` в памяти черепашки. Но если программа с таким именем уже существует (например, та, которую ты создал с помощью команды `edit mydance`), нужно выбрать другое имя или удалить существующую программу `mydance`.

Для удаления программы из памяти черепашки, введи команду `delete`, а затем имя удаляемой программы. Например, так:

```
> delete mydance
```

Эта команда удалит программу `mydance`. Теперь, чтобы загрузить программу из Интернета, ты снова можешь выполнить команду `pastebin get BLCJbpQJ mydance`.

ОГРАНИЧЕНИЯ НА САЙТЕ PASTEBIN.COM

Имей в виду, что для каждого сервера Minecraft сайт **pastebin.com** позволяет выполнять не более 25 загрузок в сутки. Это ограничение может затруднить обновление программ, особенно если на твоем сервере множество игроков.

TURTLEAPPSTORE.COM

Делиться своими программами в Интернете и без ограничений можно также на сайте **turtleappstore.com**. Вместо команды `pastebin` можно использовать программу `appstore`. Для ее загрузки в оболочке командной строки выполни команду `pastebin get iXRkjNsG appstore`:

```
> pastebin get iXRkjNsG appstore
Connecting to pastebin.com... Success.
Downloaded as appstore
```

Команда `appstore` умеет загружать программы в память черепашки так же, как и `pastebin`. Например, ты можешь выполнить команду `appstore get AlSweigart mydance`, чтобы загрузить мою программу `mydance` из моего аккаунта на веб-сайте **turtleappstore.com**.

Чтобы загружать свои программы и скачивать программы других пользователей, перейди на сайт **turtleappstore.com** и бесплатно зарегистрируйся. На этом сайте ты найдешь программы, написанные другими пользователями, и сможешь просмотреть их код! На этом сайте также размещены все программы из этой книги.

ДОПОЛНИТЕЛЬНОЕ ЗАДАНИЕ: БЕГУЩИЙ В ЛАБИРИНТЕ

Построй небольшой лабиринт из блоков, как показано на рис. 4.6, а затем напиши программу для черепашки, которая сможет пройти через него. Тебе придется точно запрограммировать ходы черепашки, чтобы она смогла выбраться из лабиринта. Если черепашка застрянет, ты всегда сможешь начать все сначала, вернув черепашку на исходную позицию.

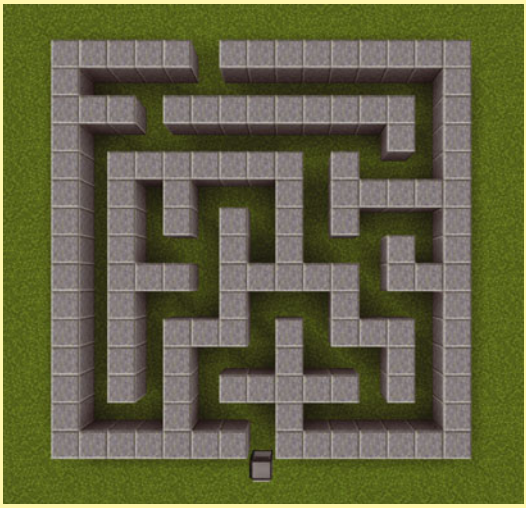


Рис. **4.6.** *Черепашка на входе в большой лабиринт*

ЧТО МЫ УЗНАЛИ

В этой главе ты изучил несколько новых концепций программирования. Ты узнал, как использовать комментарии, чтобы делать заметки, которые интерпретатор Lua игнорирует. Эти комментарии могут служить напоминанием или описанием того, для чего служат различные части твоей программы. Однострочные комментарии начинаются с символов `--` и заканчиваются в конце строки, а многострочные помещаются между символами `--[[` и `]]`.

Ты научился перемещать черепашку с помощью функций движения: `turtle.forward()`, `turtle.back()`, `turtle.turnLeft()`, `turtle.turnRight()`, `turtle.up()` и `turtle.down()`.

В этой главе также рассмотрен новый способ управления потоком исполнения, называемый циклом `for`. Эти циклы позволяют исполнять один и тот же блок кода указанное количество раз.

Теперь ты можешь использовать программу `pastebin`, которая предустановлена в памяти черепашек ComputerCraft, чтобы делиться своими программами с друзьями в Интернете на сайте **pastebin.com**. Аналогичные возможности предоставляет программа `appstore`, если посетить сайт **turtleappstore.com**.

В следующей главе мы усовершенствуем программу `mydance`, научимся использовать новые типы данных, операторы и инструкции управления потоком. Эти новые знания позволят тебе автоматизировать труд черепашек Minecraft!

5

ЧЕРЕПАШКИ – КОРОЛЕВЫ ТАНЦПОЛА

Танцевальная программа в главе 4 была довольно простой: черепашка просто повторяла одни и те же движения. В этой главе мы создадим новую программу, под управлением которой черепашка будет совершать различные движения случайным образом. Во время разработки этой программы ты узнаешь о цикле `while`, логических типах данных и специальном значении `nil`, а также о том, как запускать или пропускать блоки кода, в зависимости от соблюдения или, наоборот, несоблюдения определенных условий.

РАЗРАБОТКА ПРОГРАММЫ

В оболочке командной строки запусти текстовый редактор, выполнив команду `edit mydance2`. В текстовом редакторе введи показанные ниже строки кода. Не забудь, что номера строк указывать не нужно. В книге они для удобства, чтобы было можно обращаться к строкам по их номерам

mydance2

```
1. --[[Улучшенная Программа танцулек Эла Свейгарта
2. Make the turtle dance better!]]
```

```
 3.
 4. local isUp = false
 5. local isBack = false
 6. local danceMove
 7. print('Hold Ctrl-T to stop dancing.')
 8. while true do
 9.   danceMove = math.random(1, 5)
10.
11.   if danceMove == 1 then
12.     -- поворот налево
13.     print('Turn to the left!')
14.     turtle.turnLeft()
15.
16.   elseif danceMove == 2 then
17.     -- поворот направо
18.     print('Turn to the right!')
19.     turtle.turnRight()
20.
21.   elseif danceMove == 3 then
22.     -- движения вперед/назад
23.     if isBack then
24.       print('Move forward!')
25.       turtle.forward()
26.       isBack = false
27.     else
28.       print('Move back!')
29.       turtle.back()
30.       isBack = true
31.     end
32.
33.   elseif danceMove == 4 then
34.     -- движения вверх/вниз
35.     if isUp then
36.       print('Get down!')
37.       turtle.down()
38.       isUp = false
39.     else
40.       print('Get up!')
41.       turtle.up()
42.       isUp = true
43.     end
```

```
44.
45.   else
46.     -- кружится
47.     print('Spin!')
48.     for i = 1, 4 do
49.       turtle.turnLeft()
50.     end
51.   end
52. end
```

Сохрани программу после ввода всех инструкций. Нажав клавишу **Ctrl**, открой меню. Выбери пункт **[Save]** и нажми клавишу **Enter**. Затем выйди из редактора, нажав клавишу **Ctrl**, выбрав пункт **[Exit]** и нажав клавишу **Enter**.

ЗАПУСК ПРОГРАММЫ MYDANCE2

После выхода из редактора, в оболочке командной строки запусти программу `mydance2`. Сразу после запуска программы нажми клавишу **Esc** и понаблюдай, как черепашка совершает случайные танцевальные движения. Исполнение программы будет продолжаться до тех пор, пока ты не остановишь ее, удерживая сочетание клавиш **Ctrl+T** в течение одной секунды. После остановки программы, вернись в командную оболочку, где ты увидишь примерно следующее:

```
> mydance2
Hold Ctrl-T to stop dancing.
Spin!
Get down!
Move back!
Move forward!
Spin!
Get up!
Turn to the left!
Terminated
```

На рис. 5.1 показана вертящаяся черепашка.

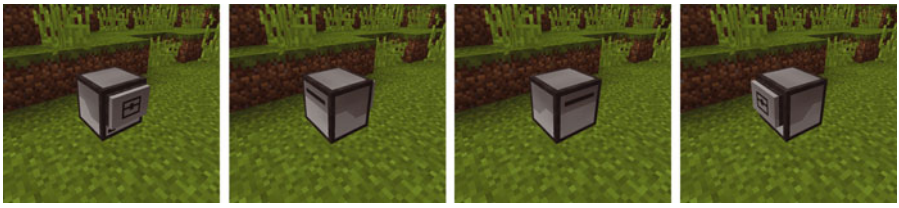


Рис. **5.1.** *Вертящаяся черепашка*

Если при запуске этой программы возникают ошибки, тщательно сравни свой код с листингом в этой книге, возможно, ты найдешь у себя опечатки. Если программу исправить не удается, удали файл, выполнив команду `delete mydance2`, и загрузи программу заново, выполнив команду `pastebin get QAH0uYqS mydance2`.

Код программы `mydance2` содержит логические типы данных, специальное значение `nil`, циклы `while`, инструкции `if` и многое другое. Давай рассмотрим все это, шаг за шагом.

ЛОГИЧЕСКИЕ ТИПЫ ДАННЫХ

Первая пара строк кода программы содержит комментарии, которые поясняют, что это за программа и кто ее автор. Строки 4 и 5 – это инструкции присваивания:

mydance2

```
1. --[[Улучшенная программа танцулек Эла Свейгарта
2. Make the turtle dance better!]]
3.
4. local isUp = false
5. local isBack = false
```

Обеим переменным, `isUp` и `isBack`, присваиваются значения `false`. Так ты сообщаешь программе, что это не строковые значения, так как они не окружены одинарными или двойными кавычками. Это логические *(или булевы)* значения — новый тип данных. Логические переменные могут принимать лишь одно из двух логических значений: `true` (истина) или `false` (ложь). (Слово «булево»

происходит от фамилии известного математика XIX века Джорджа Буля.)

Танец черепашек включает движения вверх и назад. Чтобы черепашка не слишком удалялась от своего исходного положения, мы вводим переменные `isUp` и `isBack`, с помощью которых будем отслеживать, перемещалась ли черепашка вверх или назад. Хотя эти переменные и объявляются в начале программы, используем мы их в самом конце кода.

Тип данных nil

Обрати внимание: в строке 6 объявляется переменная `danceMove` без присвоения ей значения:

mydance2

```
6. local danceMove
```

Несмотря на то, что это не прописано в коде явно, переменной `danceMove` автоматически присваивается значение `nil`. `nil` – единственное значение типа данных, которое так и называется – `nil`. Тип данных `nil` означает, что значения нет, оно отсутствует. Любая локальная инструкция, которая не присваивает значения объявленной переменной, присваивает ей значение `nil`. Код в строке 6 равнозначен строке `danceMove = nil`.

Цикл while

Помимо цикла `for` существует и другой тип цикла – *`while`*. Этот цикл исполняет блок кода до тех пор, пока выполняется какое-либо условие (т.е. его значение равно `true`). Условие – это выражение, которое может быть истинным (`true`) или ложным (`false`). Условие – это контрольная часть инструкции `while`, оно проверяется во время каждой итерации, а блок кода – исполняемая часть, то есть действие, которое должно выполняться, если условие истинно.

Строка 7 содержит напоминание игроку о том, что он может завершить программу, удерживая сочетание клавиш **Ctrl+T**, а код в строке 8 – это инструкция `while`:

mydance2

```
7. print('Hold Ctrl-T to stop dancing.')
8. while true do
9.   danceMove = math.random(1, 5)
```

Синтаксис инструкции состоит из ключевого слова `while`, за которым следует условие (в данном случае, `true`) и ключевое слово `do`, которое отмечает начало блока. Исполняемые инструкции находятся внутри блока кода, который начинается в строке 9. Конец блока отмечен ключевым словом `end`. (Поскольку в этой программе блок цикла длинный, во фрагменте кода выше показана лишь первая строка блока и не показано ключевое слово `end`, которое находится в строке 31.) Когда исполнение программы достигает ключевого слова `end`, управление передается в начало блока, где заново проверяется условие. Таким образом, выполнение цикла (итерации) продолжается до тех пор, пока условие истинно, т.е. равно `true`. Когда условие становится ложным (`false`), цикл `while` завершается.

На рис. 5.2 показан пример цикла `while`.

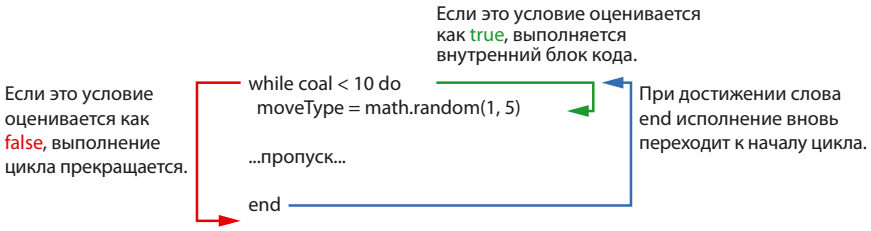


Рис. **5.2.** *Исполнение цикла* `while`.

Цикл `while` исполняется до тех пор, пока условие не примет значение `false` или `nil`. Во многих условиях используются операторы сравнения, как и в этом примере. Здесь в условии используется выражение `coal < 10`, которое проверяет значение переменной `coal`, меньше ли оно `10`. (Позже в этой главе ты узнаешь больше об операторах сравнения.) Условия, в которых используются операторы сравнения, могут принимать значение как `true`, так и `false`. Если указанное в условии числовое или строковое соотношение выполняется (строковые функции тоже могут

возвращать числовые значения), считается, что условие принимает значение `true`.

Помни, что выражение может состоять из одного значения, в этом случае выражение будет оценивать само себя. Вспомни, как в главе 2 ты вводил в оболочке Lua значение `42`, которое и оценивалось как значение `42`. В программе `mydance2` условие в строке 8 выполняется, и, поскольку оно выполняется всегда, условие цикла `while` всегда будет истинно (`true`). Таким образом, цикл будет исполняться до тех пор, пока пользователь не остановит его, завершив программу. Такой цикл называется бесконечным. В рассматриваемой программе использован бесконечный цикл, потому что мы хотим, чтобы черепашка продолжала танцевать, пока мы не завершим программу, удерживая сочетание клавиш **Ctrl+T**.

В нашей программе черепашка совершает пять различных танцевальных движений, которые выбираются случайным образом. Выбор основан на значении переменной `danceMove`. Код в строке 9 вызывает функцию `math.random()` для генерации случайного числа в диапазоне от `1` до `5` и сохраняет это случайное число в виде значения переменной `danceMove`.

ПРИНЯТИЕ РЕШЕНИЙ С ПОМОЩЬЮ ИНСТРУКЦИЙ IF

Напомню, что поток исполнения – это порядок выполнения строк кода. Обычно поток исполняется сверху вниз. Циклы, такие как `for` или `while`, – это операции управления потоком, поскольку изменяют порядок его исполнения, возвращая программу к началу цикла.

`if` – это другой тип инструкции управления потоком. Строка 11 содержит инструкцию `if`, которая допускает исполнение блока кода только в том случае, если условие цикла истинно (`true`):

mydance2

```
11.   if danceMove == 1 then
```

```
12.      -- поворот налево
13.      print('Turn to the left!')
14.      turtle.turnLeft()
15.
16.    elseif danceMove == 2 then
```

Синтаксис инструкции состоит из ключевого слова `if`, за которым следует условие, и ключевого слова `then`, отмечающего начало блока кода. В строке 12 начинается блок исполняемого кода, который выполняется, если условие истинно. Ключевые слова `elseif`, `else` или `end` отмечают конец блока инструкции `if`. В нашем случае строка 16 завершает блок ключевым словом `elseif`, которое мы подробно рассмотрим позже.

Условие в строке 11 содержит код, который тебе еще незнаком, потому что содержит другой оператор сравнения. Давай рассмотрим операторы сравнения!

СРАВНЕНИЕ ДВУХ ЗНАЧЕНИЙ С ПОМОЩЬЮ ОПЕРАТОРОВ СРАВНЕНИЯ

Операторы сравнения выполняют сравнение двух значений и присваивают условию соответствующее значение – `true` или `false`. Условие в строке 11 – `danceMove == 1` – содержит оператор сравнения «равно» (==). Оператор == проверяет, равны ли значения с двух сторон от него, и возвращает значение `true`, если значения равны, или `false`, если нет. Если значение переменной `danceMove` равно `1`, исполняется код в строках 12 – 14 (между ключевыми словами `then` и `elseif`). Этот код выводит на экран сообщение `Turn to the left!` и поворачивает черепашку налево.

Будь осторожен, не путай оператор сравнения «равно» (==) и оператор присваивания (=). Оператор сравнения == проверяет, равны ли значения по обе стороны от оператора, и возвращает значение `true`, если это так, тогда как оператор присваивания = присваивает переменной, имя которой указывается перед оператором, значение, которое указано после оператора. В табл. 5.1 перечислены операторы сравнения, поддерживаемые интерпретатором Lua.

Таблица 5.1. Операторы сравнения в языке Lua

Оператор сравнения	Описание
==	Равно
~=	Не равно
>	Больше
<	Меньше
>=	Больше или равно
<=	Меньше или равно

Ради эксперимента в оболочке Lua введи следующие выражения:

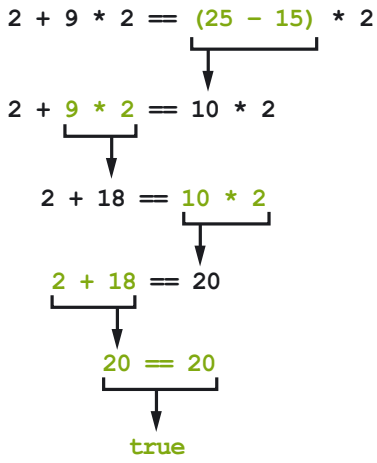
```
❶ lua> 2 == 2
  true
  lua> 4 == 2 + 2
  true
❷ lua> 2 ~= 3
  true
  lua> 2 < 3
  true
  lua> 3 < 3
❸ false
  lua> 3 > 2
  true
  lua> 2 >= 2
  true
❹ lua> 2 + 9 * 2 == (25 - 15) * 2
true
```

Теперь тебе должно быть понятно, что выражения с операторами сравнения всегда возвращают логическое значение `true` или `false`. Оператор «равно» (==) сравнивает два значения и возвращает значение `true`, если они равны ❶. Оператор «не равно» (~=) возвращает значение `true`, если, наоборот, они не равны ❷. Операторы «меньше» (<) и «больше» (>) работают так же, как ты изучал на уроках математики. Они определяют, больше или меньше значение слева от оператора по сравнению со значением справа от него, и возвращают `true`, если условие выполняется. Но если значения по обе стороны от оператора рав-

ны друг другу, или неравенство не выполняется, оператор < или > вернет `false` ❸.

Операторы «больше или равно» (>=) и «меньше или равно» (<=) выполняют такое же сравнение, что и операторы «меньше» (<) и «больше» (>), но вернут `true` также и в том случае, если значения по обе стороны оператора равны.

Операторы сравнения можно использовать и для сравнения выражений, например, `2 + 9 * 2 == (25 - 15) * 2` ❹. Выражения такого типа можно привести к выражению сравнения двух значений, как показано ниже:



В итоге такие выражения будут определены как истинные или ложные.

АЛЬТЕРНАТИВНЫЕ ВЫЧИСЛЕНИЯ С ПОМОЩЬЮ ИНСТРУКЦИЙ ELSEIF

В конец блока инструкции `if` можно добавить инструкцию `elseif`. Фактически, эта инструкция работает почти так же, как и сама инструкция `if`. Они обе имеют собственное ключевое слово (`if` или `elseif`), собственное условие и у обеих ключевое слово `then` отмечает начало блока. Исполняемые инструкции размещаются внутри блока, а ключевое слово `elseif`, `else` или `end` отмечает его конец.

Инструкция `elseif` должна быть сопряжена с инструкцией `if`, то есть исполнять свой блок, если условие инструкции `if` ложно, а условие инструкции `elseif` при этом истинно. Проще можно сказать так: «Если `if` истинно, сделай это, а если `if` ложно, а `elseif` истинно, сделай то». Если оба условия ложны, ни один блок кода не выполняется, и исполнение переходит к следующей части программы.

Строки 17–19 исполняются, если условие в строке 11, `danceMove == 1`, ложно (`false`), а условие в строке 16, `danceMove == 2`, истинно (`true`):

mydance2

```
16.   elseif danceMove == 2 then
17.     -- поворот направо
18.     print('Turn to the right!')
19.     turtle.turnRight()
20.
21.   elseif danceMove == 3 then
```

Если переменной `danceMove` присвоено значение `2`, код в строке 9 выводит сообщение `Turn to the right!`, а код в строке 10 поворачивает черепашку направо.

ВЛОЖЕННЫЕ БЛОКИ КОДА

Для случая, если переменной `danceMove` присвоено значение `3`, в строке 21 размещена еще одна инструкция `elseif`. Внутри блока инструкции `elseif`, в строке 23, помещена другая инструкция `if` и ее блок кода, который начинается в строке 24 и заканчивается в строке 26. Добавление блока кода внутрь другого блока называется вложением. Обрати внимание, что, поскольку строки 24–26 находятся внутри нового блока, отступ увеличивается на два пробела. Этот отступ не обязателен, но код становится более удобным для чтения.

mydance2

```
21.   elseif danceMove == 3 then
```

```
22.     -- движения вперед/назад
23.     if isBack then
24.       print('Move forward!')
25.       turtle.forward()
26.       isBack = false
```

Инструкция `if` вложена в блок `elseif`, поэтому исполняется, только если условие инструкции `elseif` истинно. Вложенная инструкция `if` необходима, потому что танцевальное движение в строке 21 отличается от остальных. Этот танцевальный пируэт перемещает черепашку вперед, но мы не хотим, чтобы черепашка уходила слишком далеко; а то она может и вовсе скрыться с глаз! Значит надо проверять, переместилась ли черепашка вперед, и если да, то нужно вернуть ее назад.

Именно это мы и делаем с помощью вложенной инструкции `if`, которая содержит переменную `isBack`. Мы объявили ее в начале программы, а теперь используем для хранения логического значения, указывающего, вернулась ли черепашка в точку начала своего движения.

Запомни, значение переменной может использоваться внутри блока только в том случае, если это значение вычисляется или присваивается внутри того же самого блока. Поэтому, если переменной `isBack` присвоено значение `true`, условие в строке 23 истинно и исполняются строки 24–26, которые перемещает черепашку вперед и присваивают переменной `isBack` значение `false`.

ОТСТУПЫ

Отступы в исходном коде Lua необязательны, но добавленные пробелы в начале строк кода упрощают его чтение. Глядя на отступы, ты можешь сразу определить, где начинаются и заканчиваются блоки кода. Сравни следующие листинги – в примере слева есть отступы, а в примере справа нет:

-- Этот код легче читать!	-- Этот код труднее читать.

if isBack then
  print('Move forward!')
  turtle.forward()
  isBack = false
else
  print('Move back!')
  turtle.back()
  isBack = true
end
	if isBack then
	print('Move forward!')
	turtle.forward()
	isBack = false
	else
	print('Move back!')
	turtle.back()
	isBack = true
	end

В листинге слева четко видно, где начинаются и заканчиваются блоки кода. А вот в листинге справа инструкции `else` и `end` еще придется поискать. Ставить отступы – это хорошая привычка.

ПРИНЯТИЕ РЕШЕНИЯ... ИЛИ ИНСТРУКЦИЯ ELSE

Код в строке 27 – это инструкция `else`, который завершает блок инструкции `if`. Блок кода, следующий за инструкцией `else`, исполняется, если все предыдущие условия `if` и `elseif` являются ложными. В нашем примере инструкция `else` сопряжена с инструкцией `if` из строки 23, но не с инструкциями `if` или `elseif` из строк 11, 16 и 21. Причина в том, что инструкция `else` вложена в инструкцию `elseif` строки 21, а это означает, что она исполняется только в том случае, если условие в строке 21 истинно.

Код в строках 28–30 исполняется, если условие в строке 23 ложно. Блок кода после инструкции `else` заканчивается, когда программа достигает ключевого слова `end`. Блок `else` работает только в том случае, если условия всех предыдущих инструкций `if` и `elseif` ложны, поэтому она сама не имеет никакого условия.

mydance2

```
27.     else
28.       print('Move back!')
```

```
29.       turtle.back()
30.       isBack = true
31.     end
```

Если переменной `isBack` присвоено значение `false`, исполняется код в строках 28–30, который выводит на экран сообщение `Move back!`, перемещает черепашку назад и присваивает переменной `isBack` значение `true`.

Обрати внимание, что, если переменной `isBack` присвоено значение `true`, программа присваивает ей значение `false`. Если переменной `isBack` присвоено значение `false`, программа присваивает ей значение `true`. В результате, каждый раз, когда переменной `danceMove` присваивается значение `3`, черепашка чередует движения вперед и назад. Таким образом, черепашка никогда не уйдет слишком далеко назад или вперед. Она будет перемещаться между двумя позициями.

ПЕРЕМЕЩЕНИЕ ВВЕРХ И ВНИЗ

Код в строке 33 – это инструкция `elseif`, которая проверяет, присвоено ли переменной `danceMove` значение `4`. Обрати внимание, что, поскольку инструкция `if` в строке 23 (`if isBack then`) вложена в предыдущий блок `elseif`, программа пропустит ее, если значение переменной `danceMove` не равно `3`, и исполнение перейдет к инструкции `elseif` в строке 33. Это означает, что инструкция `elseif` в строке 33 сопряжена с инструкцией `if` в строке 11.

mydance2

```
33.   elseif danceMove == 4 then
34.     -- движения вверх/вниз
35.     if isUp then
36.       print('Get down!')
37.       turtle.down()
38.       isUp = false
39.     else
40.       print('Get up!')
```

```
41.        turtle.up()
42.        isUp = true
43.      end
```

Код в строках 35–43 аналогичен коду в строках 23–31. Если значение переменной `danceMove` равно `4`, вложенная инструкция `if` в строке 35 исполнит свой блок, если переменной `isUp` при этом присвоено значение `true`. В противном случае инструкция `else` в строке 39 выполнит свой блок кода.

В коде программы используются переменные `isBack` и `isUp` для определения, какой шаг сделать, если переменной `danceMove` присвоено значение `3` или `4`. Исполнение инструкций `if`, `elseif` и `else` напрямую зависит от значений этих трех переменных.

Рис. 5.3 демонстрирует, как черепашка движется вверх и вниз.

Рис. **5.3.** *Черепашка движется вверх и вниз*

ПОВОРОТ КРУГОМ

Еще одна инструкция `else` находится в строке 45 и сопряжена с инструкцией `if` в строке 11. Если все условия инструкций `if` и `elseif` в строках 11, 16, 21, 33 и 45 ложны, будет исполнен блок именно этой инструкции `else`:

mydance2

```
 8. while true do
 9.   danceMove = math.random(1, 5)
10.
11.   if danceMove == 1 then
```

```
    ...пропуск...

45.   else
46.     -- кружится
47.     print('Spin!')
48.     for i = 1, 4 do
49.       turtle.turnLeft()
50.     end
51.   end
52. end
```

В этом блоке (который размещен в строках 46–51), выводится сообщение `Spin!` и запускается цикл, который поворачивает черепашку налево четыре раза. Такой разворот оставляет черепашку в первоначальном направлении.

Строки 50–52 содержат инструкции `end`. Взглянув на отступы, их можно легко сопоставить с инструкциями, к которым они относятся, следующим образом.

- Строка 50 – это конец блока инструкции `for`, начинающейся в строке 48;
- Строка 51 – это конец блока инструкции `else`, начинающейся в строке 45;
- Строка 52 – это конец блока инструкции `while`, начинающейся в строке 8.

Запомни, исполняется один и только один из блоков, следующих за инструкциями `if`, `elseif` и `else`. Все остальные блоки игнорируются. Таким образом, после исполнения одного из этих блоков исполнение переходит к инструкции `end` в строке 51. Следующая строка кода – инструкция `end` цикла `while`, поэтому исполнение возвращается к строке 8 и перепроверяет условие цикла `while`. Поскольку это условие выполняется всегда, исполнение программы продолжится, и черепашка совершит еще один танцевальный пируэт.

Из-за бесконечного цикла нет причин для завершения этой программы. Черепашка будет продолжать танцевать бесконечно долго, пока ты не завершишь программу, удерживая сочетание клавиш **Ctrl+T**.

ДОПОЛНИТЕЛЬНОЕ ЗАДАНИЕ: ЧЕРЕПАШКА-ЧАСОВОЙ

Отправь свою черепашку нести караульную службу. Используя циклы, заставь черепашку пройти маршем вперед 10 шагов, затем развернуться и промаршировать назад. Помести весь этот код в бесконечный цикл, чтобы черепашка продолжала маршировать до тех пор, пока у нее не закончится топливо или игрок не нажмет сочетание клавиш **Ctrl+T**.

ЧТО МЫ УЗНАЛИ

В этой главе ты узнал о циклах и условиях, которые позволяют повторять один и тот же код. Ты также узнал, как использовать инструкции `if`, `elseif` и `else` и операторы сравнения для исполнения или пропуска блоков кода. Эти инструкции управления потоком позволяют запрограммировать черепашку на принятия решений, которые определяют ее дальнейшее поведение. Операторы сравнения, производя сравнение значений, возвращают логический результат: `true` или `false`. Тип данных `nil` имеет только одно значение, `nil`, что означает отсутствие значения.

Конечно, если захочется дополнительно дрессировать черепашек, то при написании кода нужно будет использовать множество инструкций `if` и циклов. Но танцующая черепашка – это уже неплохо для начала!

6

ПРОГРАММИРОВАНИЕ ЧЕРЕПАШКИ-ЛЕСОРУБА

Пришло время новых знаний и проверки их в мире Minecraft. Мы запрограммируем черепашку так, чтобы она рубила деревья. С помощью черепашек-лесорубов будут решены проблемы с добычей древесины!

Вырубка деревьев в Minecraft вручную – довольно нудная задача. Это медленный процесс, он изнашивает инструменты, и, чтобы полностью срубить дерево, нужно добраться до самого верхнего деревянного блока. А вот черепашки смогут срубить блок древесины в один прием, их инструменты не изнашиваются, да и подниматься наши роботы могут на любую высоту, как это показано на рис. 6.1.

Прежде чем мы сможем написать программу обработки деревьев, тебе нужно изучить некоторые дополнительные возможности черепашки и обдумать, как именно должна работать программа.

Рис. **6.1.** *Четыре черепашки рубят высокое тропическое дерево*

ОСНАЩЕНИЕ ЧЕРЕПАШЕК ИНСТРУМЕНТАМИ

Чтобы черепашка могла рубить деревья, ее нужно оснастить совершенно новым алмазным инструментом. Черепашек можно оборудовать алмазными кирками, лопатами, топорами, мотыгами или мечами, но ни железный, ни изношенный алмазный инструмент работать не будет. Зато черепашки не изнашивают инструмент при работе!

Чтобы оснастить черепашку инструментом, нужно поместить его в текущую ячейку инвентаря (ту, которая выделена толстой рамкой) черепашки. Возьми алмазную кирку и помести ее в текущую ячейку. Затем запусти оболочку Lua, введя следующую команду:

```
> lua
Interactive Lua prompt.
Call exit() to exit.
```

Затем, чтобы экипировать черепашку добавленным инструментом, выполни следующую команду:

```
lua> turtle.equipLeft()
```

Черепашку можно оснастить сразу двумя инструментами: одним слева, а другим справа. Если же нужно «разоружить» черепашку, вызови функцию `turtle.equipLeft()` или `turtle.equipRight()`, ничего не добавляя в текущую ячейку. Инструмент будет убран и помещен в инвентарь.

Черепашку можно оснастить любыми алмазными инструментами, но алмазная кирка универсальна. Алмазная лопата позволяет выкапывать земляные блоки, а алмазный топор – вырубать древесину, но не они не могут добывать камень или руду. Алмазная кирка позволяет добывать все типы блоков, поэтому мы и будем ее использовать для всех черепашек в этой книге.

Вооруженная киркой черепашка может вызвать функцию `turtle.dig()`, с помощью которой она может добывать блоки и древесину. Это мы и изучим в следующем разделе.

Алгоритм рубки дерева

Прежде чем писать код, давай тщательно продумаем, что нужно делать черепашке-лесорубу. Предварительное планирование поможет выявить ошибки до того как программа будет написана, а не после. Как говорят опытные плотники, «семь раз отмерь, один отрежь». Проработаем алгоритм работы черепашки-лесоруба. Алгоритм – это последовательность действий, которые надо выполнить компьютеру для решения поставленной задачи.

Чтобы срубить дерево, мы помещаем черепашку у его основания, рубим, двигаемся вперед, рубим над черепашкой, поднимаемся вверх, а затем повторяем последние два шага для всего дерева. Когда черепашка полностью вырубит дерево, она вернется на землю, чтобы ее можно было забрать. На рис. 6.2–6.6 показан весь процесс.

Для перемещения черепашки мы будем использовать функции `turtle.forward()` и `turtle.up()`. А чтобы заставить черепашку рубить (то есть добывать древесину), используем функцию `turtle.dig()`, которая будет добывать

Рис. **6.2.** *Черепашка повернута к дереву и начинает рубку в нижней его части*

Рис. **6.3.** *Черепашка вырубает нижний блок дерева, а затем движется вперед, так, чтобы очутиться под деревом*

Рис. **6.4.** *Черепашка вырубает блок, который находится над ней, а затем движется вверх на одну позицию*

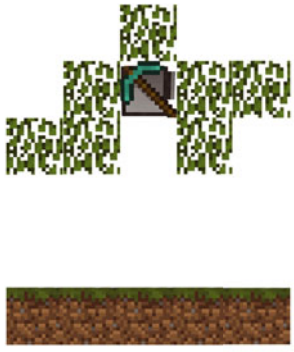


Рис. **6.5.** *Черепашка продолжает вырубать деревянные блоки, пока над ней не останется древесины*

Рис. **6.6.** *Черепашка возвращается на землю, чтобы игрок мог ее забрать. Листья пропадут через некоторое время*

блок, расположенный перед ней, а также функцию `turtle.digUp()`, которая добывает блок над черепашкой.

Деревья в Minecraft достигают разной высоты, поэтому хотелось бы написать программу, в которой не будет жестко кодироваться высота дерева. Жесткое кодирование означает программирование ограниченного, фиксированного решения. Программы с жестким кодированием не могут обрабатывать различные ситуации, так как для

этого требуется переписать код. Например, можно написать такой код:

```
turtle.digUp()
turtle.up()
turtle.digUp()
turtle.up()
turtle.digUp()
turtle.up()
turtle.digUp()
```

Эта программа проста и понятна, но с ней черепашка может рубить только те деревья, высота которых составляет ровно четыре блока. Если же ты хочешь, чтобы черепашка вырубала деревья разных размеров, этот далеко не идеальный код придется переписать.

Но сначала давай разработаем алгоритм для вырубки деревьев любого размера. К примеру, алгоритм может состоять из следующих шагов:

1. Черепашка, повернутая в направлении дерева, помещается у его основания.
2. Черепашка вырубает нижний блок дерева, расположенный прямо перед ней.
3. Черепашка движется под дерево.
4. Вырубает блок, расположенный над ней, и поднимается выше, до тех пор, пока над ней не останется блоков древесины.
5. Движется вниз, пока не вернется на землю.
6. Стоп.

Этот алгоритм позволит черепашке рубить деревья любой высоты. Его ты реализуешь в программе `choptree`.

КОД ПРОГРАММЫ CHOPTREE

В оболочке командной строки выполни команду `edit choptree`, чтобы запустить текстовый редактор. В текстовом редакторе введи показанные ниже строки

кода. Помни, что номера строк указывать не нужно. В книге они для удобства, чтобы можно было обращаться к строкам кода по их номерам.

choptree

```
 1. --[[Программа вырубки деревьев Эла Свейгарта
 2. Chops down the tree in front of turtle.]]
 3.
 4. if not turtle.detect() then
 5.   error('Could not find tree!')
 6. end
 7.
 8. print('Chopping tree...')
 9.
10. if not turtle.dig() then -- вырубка основания дерева
11.   error('Turtle needs a digging tool!')
12. end
13.
14. turtle.forward() -- перемещение под дерево
15. while turtle.compareUp() do
16.   -- вырубка, пока обнаруживается древесина
17.   turtle.digUp()
18.   turtle.up()
19. end
20.
21. -- возвращение на землю
22. while not turtle.detectDown() do
23.   turtle.down()
24. end
25.
26. print('Done chopping tree.')
```

Сохрани программу после ввода всех инструкций. Нажав клавишу **Ctrl**, открой меню. Выбери пункт **[Save]** и нажми клавишу **Enter**. Затем выйди из редактора, нажав клавишу **Ctrl**, выбрав пункт **[Exit]** и нажав клавишу **Enter**.

ЗАПУСК ПРОГРАММЫ CHOPTREE

Выбери черепашку и помести новую алмазную кирку в ее инвентарь. Найди дерево в мире Minecraft и установи черепашку так, чтобы передней частью она была обращена к нижнему блоку выбранного дерева, как показано на рис. 6.7.

Рис. **6.7.** *Установи черепашку у основания дерева*

Щелкни правой кнопкой мыши по черепашке, чтобы открыть ее графический интерфейс. Убедись в наличии топлива у черепашки и что ее текущая ячейка пуста — в ней она будет сохранять вырубленные блоки. Затем запусти программу `choptree` и понаблюдай, как черепашка вырубит все дерево. После того, как черепашка добудет всю древесину, она вернется на землю, где ты сможешь забрать из ее инвентаря все добытые блоки.

Если при запуске этой программы возникают ошибки, тщательно сравни свой код с листингом в этой книге и исправь все опечатки. Если исправить программу не удается, удали файл, выполнив команду `delete choptree`, а затем загрузи программу заново с помощью команды `pastebin get 8NgPXXxN choptree`.

ОБНАРУЖЕНИЕ БЛОКОВ С ФУНКЦИЯМИ ОБНАРУЖЕНИЯ ЧЕРЕПАШКИ

Давай рассмотрим каждую строку исходного кода программы `choptree`. Сначала черепашке нужно проверить, есть ли перед ней дерево.

choptree

```
1. --[[Программа вырубки деревьев Эла Свейгарта
2. Chops down the tree in front of turtle.]]
3.
4. if not turtle.detect() then
```

Функция `turtle.detect()` возвращает значение `true`, если какой-либо блок находится перед черепашкой, или `false`, если перед ней ничего нет. Функция также возвращает значение `false`, если перед черепашкой вода или лава, потому что черепашка может перемещаться по этим типам блоков.

Подобно вызовам `turtle.digUp()` и `turtle.digDown()`, функции `turtle.detectUp()` и `turtle.detectDown()` обнаруживают блоки выше и ниже черепашки соответственно. Хотя функция `turtle.detect()` может определить, находится ли твердый блок перед черепашкой, она не может узнать, какой именно это блок. Программе приходится полагаться на то, что пользователь поместит черепашку перед деревом.

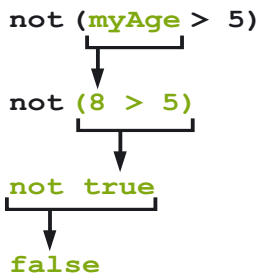
Перед функцией `turtle.detect()` в строке 4 находится логический оператор `not`, который мы рассмотрим в следующем разделе.

ЛОГИЧЕСКИЙ ОПЕРАТОР NOT

Оператор `not` поддерживает только логические выражения, возвращая значение, противоположное аргументу. Таким образом, not true – это false, а not false – это true. В оболочке Lua введи следующие команды:

```
lua> not true
false
lua> not false
true
lua> myAge = 8
lua> not (myAge > 5)
false
```

Логический оператор `not` можно использовать не только в конструкциях типа `not false`, но и в любых других логических выражениях, например `not (myAge > 5)`. Это работает следующим образом:



В строке 4 программы `choptree` оператор `not` изменяет логическое значение, возвращаемое функцией `turtle.detect()`, на противоположное. Если перед черепашкой нет блока, функция `turtle.detect()` вернет `false`, а оператор `not` обратит это значение в `true`. Строку `if not turtle.detect() then` можно прочитать как «если черепашка не обнаруживает блок, она выполняет следующий блок кода». Если какой-либо блок обнаружен перед черепашкой, выполняется блок кода, который следует за инструкцией `if`.

Существует еще два логических оператора, `and` и `or`. Мы не будем использовать их в программе `choptree`, но рассмотрим, как они работают, поскольку они пригодятся в других программах.

Логический оператор and

Логический оператор `and` оценивает два логических значения и возвращает результат `true`, если они оба истинны (`true`). Если хотя бы одно из них ложно (`false`), то и оператор вернет значение `false`. В оболочке Lua введи следующие инструкции:

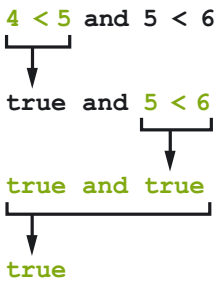
```
lua> true and true
true
lua> true and false
false
lua> false and true
false
lua> false and false
false
```

Порядок расположения оцениваемых значений не влияет на оценку выражения в целом.

Этот оператор можно использовать и в сложных выражениях, например так:

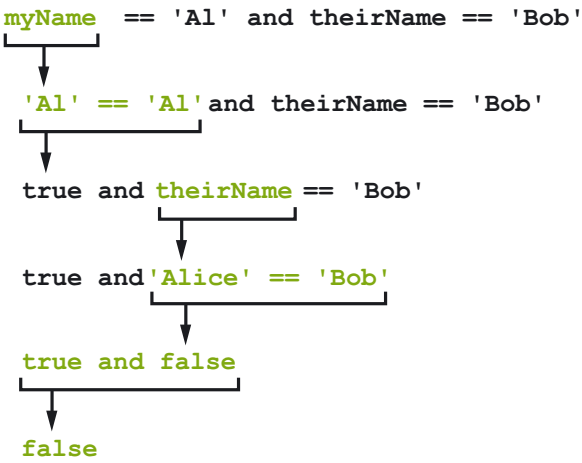
```
lua> 4 < 5 and 5 < 6
true
lua> myName = 'Al'
lua> theirName = 'Alice'
lua> myName == 'Al' and theirName == 'Bob'
false
```

Выражение `4 < 5 and 5 < 6` оценивается следующим образом:



Поскольку с обеих сторон оператора `and` выражения истинны, то и оператор вернет значение `true`.

Но в строке `myName == 'Al' and theirName == 'Bob'` выражения с обеих сторон оператора не истинны:



ЛОГИЧЕСКИЙ ОПЕРАТОР OR

Логический оператор `or` оценивает два логических значения и возвращает результат `true`, если одно из них истинно. Если оба значения ложны (`false`), все выражение принимает значение `false`. Введи в оболочке Lua следующие инструкции:

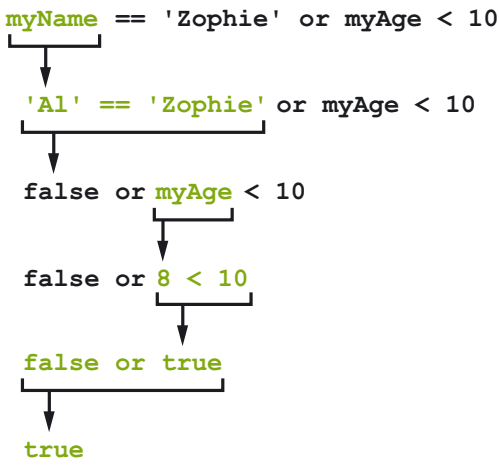
```
lua> true or true
true
lua> true or false
true
lua> false or true
true
lua> false or false
false
```

В отличие от оператора `and`, оператор `or` вернул значение `false` только в последнем случае.

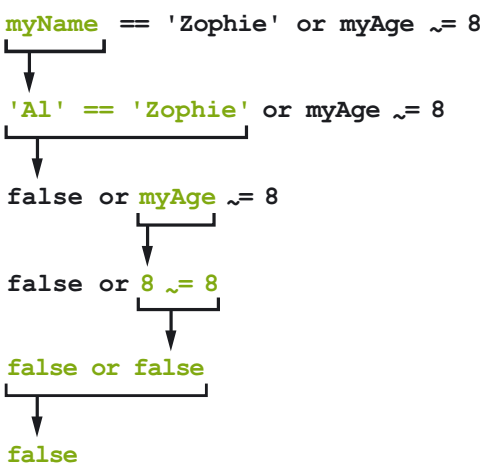
Этот оператор также можно использовать в сложных выражениях, например так:

```
lua> 10 > 5 or 'Hello' == 'Hello'
true
lua> myName = 'Al'
lua> myAge = 8
lua> myName == 'Zophie' or myAge < 10
true
lua> myName == 'Zophie' or myAge ~= 8
false
```

Выражение `myName == 'Zophie' or myAge < 10` оценивается следующим образом:



В рассматриваемой строке одна часть ложна, а другая истинна, и выражение в целом оценивается как истинное. Но в строке `myName == 'Zophie' or myAge ~= 8` ложны обе части выражения:



Поэтому вся строка возвращает результат `false`.

Как оператор `and`, так и оператор `or`, применимы в выражениях, использующих разные типы данных.

Логические операторы `and`, `or` и `not` позволяют создавать более сложные условия для инструкций `if`, `elseif` и `while`.

ЗАВЕРШЕНИЕ ПРОГРАММ С ПОМОЩЬЮ ФУНКЦИИ ERROR()

Вернемся к программе `choptree`. Если при запуске программы перед черепашкой нет какого-либо блока, то есть условие `not turtle.detect()` в строке 4 возвращает значение `false`, программа должна завершиться с сообщением об ошибке. Обычно программы завершают работу по достижении последней строки кода. Но ты можешь вызвать функцию `error()` со строковым аргументом, чтобы завершить программу раньше и отобразить сообщение об ошибке. Если вызвать функцию `error()` без передачи ей строкового аргумента, программа просто прекращает исполнение, не отображая никакого сообщения.

Нам нужно вывести сообщение об ошибке, если черепашка не повернута к дереву, поэтому в строке 5 мы передаем функции `error()`строковый аргумент:

choptree

```
4. if not turtle.detect() then
5.   error('Could not find tree!')
6. end
```

Если перед черепашкой нет блока, программа завершает работу и выводит в интерфейсе черепашки сообщение `choptree:5:Could not find tree!`. Значение `choptree:5` добавлено интерпретатором Lua. Оно указывает, что ошибку вызвала строка 5 программы `choptree`. В результате, если перед черепашкой нет дерева для вырубки, программа прекращает работу.

ВЫРУБКА ДРЕВЕСИНЫ С ПОМОЩЬЮ ЧЕРЕПАШКИ

Если условие `not turtle.detect()` в строке 4 кода возвращает значение `false`, т.е. черепашка обнаруживает перед собой блок, функция `error()` вызвана не будет. Вместо этого программа `choptree` вырубает нижний блок, вызывая функцию `turtle.dig()`, а затем перемещает черепашку под дерево. Добытый блок помещается в текущую ячейку (или другую ячейку, если текущая занята). Код в строках 8–14 выводит сообщение о том, что программа запущена, вырубает блок и перемещает черепашку под дерево.

choptree

```
 8. print('Chopping tree...')
 9.
10. if not turtle.dig() then -- вырубка основания дерева
11.   error('Turtle needs a digging tool!')
12. end
13.
14. turtle.forward() -- перемещение под дерево
```

Обрати внимание, что у черепашки должна быть кирка, иначе функция `turtle.dig()` не сработает и вернет значение `false`. Если это произойдет, черепашка не сможет вырубить дерево. Если функция `turtle.dig()` возвращает значение `false`, условие `not turtle.dig()` принимает значение `true`, поэтому код в строке 11 остановит программу и отобразит сообщение об ошибке `choptree:11:Turtle needs a digging tool!`.

Функции `turtle.digUp()` и `turtle.digDown()`, подобно вызову `turtle.dig()`, также добывают блоки, но не перед черепашкой, а выше или ниже ее соответственно.

СРАВНЕНИЕ БЛОКОВ С ПОМОЩЬЮ ФУНКЦИЙ СРАВНЕНИЯ

Функция `turtle.compare()` сравнивает блок, к которому повернута черепашка, с блоком в ее текущей ячейке. Если они идентичны, функция `turtle.compare()` вернет значение `true`. Если они различны, функция `turtle.compare()` вернет значение `false`. Функции `turtle.compareUp()` и `turtle.compareDown()` выполняют аналогичные действия, за исключением того, что они сравнивают блок выше или ниже черепашки соответственно.

Поскольку первая часть программы вырубает самый нижний блок дерева, перед началом перемещения черепашки вверх по дереву в ее текущей ячейке уже должен быть блок. Нам нужно проверить, идентичен ли блок над черепашкой блоку в ее текущей ячейке, поэтому до запуска программы важно убедиться, что текущая ячейка пуста. Для этого мы используем функцию `turtle.compareUp()`. Эта функция проверяет блок над черепашкой, а не перед ней, и возвращает значение `true`, если над черепашкой обнаружен блок древесины.

choptree

```
15. while turtle.compareUp() do
16.   -- вырубка, пока обнаружена древесина
17.   turtle.digUp()
18.   turtle.up()
19. end
```

Цикл `while`, блок кода которого начинается в строке 15, использует в своем условии значение, которое возвращает функция `turtle.compareUp()`. То есть пока над черепашкой есть дерево, этот цикл будет выполняться. Код в строке 17 вырубает блок над черепашкой, а код в строке 18 перемещает ее вверх. Цикл останавливается, когда функция `turtle.compareUp()` возвращает значение `false` – это происходит, когда ствол дерева заканчивается.

Обрати внимание, что в ситуациях, когда это возможно, можно также использовать функцию `turtle.compareDown()` для проверки блоков под черепашкой.

ВОЗВРАЩЕНИЕ НА ЗЕМЛЮ

Цикл `while` завершается, когда над черепашкой больше нет древесины, после чего она должна вернуться на землю.

choptree

```
21. -- возвращение на землю
22. while not turtle.detectDown() do
23.   turtle.down()
24. end
25.
26. print('Done chopping tree.')
```

Код в строке 22 начинает новый цикл `while`, в условии которого используется выражение `not turtle.detectDown()`. Цикл продолжается до тех пор, пока черепашка не обнаружит блок под собой, то есть, пока условие возвращает значение `true`, вызывается функция `turtle.down()`. Когда цикл завершится и черепашка вернется на землю, программа выведет сообщение `Done chopping tree.`, чтобы игрок знал, что пришло время забрать черепашку (вместе с собранной древесиной).

ЗАПУСК ПРОГРАММ И ФУНКЦИЯ SHELL.RUN()

Программу `choptree` будет проще использовать, если сделать так, чтобы она запускалась сразу же, как только черепашка окажется перед деревом. Мы можем это сделать, используя специальную программу `startup`, которая будет запускаться при открытии графического интерфейса черепашки, когда та стоит перед деревом. Давай напишем такую программу.

Запусти текстовый редактор, введя команду `edit startup` в оболочке командной строки. В текстовом редакторе введи следующую строку кода:

startup

```
shell.run('choptree')
```

Функция `shell.run()` запустит строку кода, которую ты ей передашь, как если бы ты ввел ее в оболочке командной строки. В нашем случае при первом открытии графического интерфейса черепашки будет запущена программа `startup`, которая, в свою очередь, запустит программу `choptree`, и не нужно вводить название программы `choptree` в оболочке командной строки. Функция `shell.run()` возвращает значение `true`, если она может успешно запустить программу, и значение `false`, если не может найти указанную программу или запуск программы вызывает ошибку – функцию `error()`.

ДОПОЛНИТЕЛЬНОЕ ЗАДАНИЕ: К ЦЕНТРУ ЗЕМЛИ

Создай программу, с помощью которой черепашка выроет глубокую дыру, а не срубит дерево. Это можно сделать, вызывая в цикле функции `turtle.digDown()` и `turtle.down()`. Затем создай цикл, вызывающий функцию `turtle.up()`, столько же раз, сколько функцию `turtle.down()` до этого. Этот цикл вернет черепашку на поверхность. Ты же не хочешь, чтобы она остановилась на дне шахты?

ЧТО МЫ УЗНАЛИ

Мы запрограммировали черепашку-лесоруба, чтобы ускорить добычу древесины при лесозаготовках. Программу `choptree` можно загрузить в несколько черепашек. Установи одну черепашку перед деревом и, пока она добывает древесину, поставь другую черепашку перед другим деревом и т.д. Таким образом ты сможешь получить огромное количество древесины в кратчайшие сроки!

Древесину, которую добывают черепашки, можно использовать в качестве топлива, чтобы «кормить» самих черепашек. Обязательно убедись, что из каждого блока древесины будет изготовлено четыре блока досок, потому что доски, как и древесина, обеспечивают черепашку 15 единицами топлива.

Черепашка-лесоруб – это хорошее подспорье, но все еще нужно много потрудиться, чтобы ставить каждую черепашку у основания дерева. В следующих нескольких главах ты узнаешь, как многократно использовать единожды написанный код и как создать деревообрабатывающую ферму, программируя черепашек на самостоятельное выращивание и вырубку деревьев.

7

СОЗДАНИЕ МОДУЛЕЙ ДЛЯ МНОГОКРАТНОГО ИСПОЛЬЗОВАНИЯ КОДА

По мере того, как программы становятся более сложными, возникает желание сэкономить время и повторно использовать уже написанный код. Это можно организовать, если использовать модули – программы, содержащие функции, которые могут быть вызваны твоими приложениями.

В этой главе ты узнаешь, как создавать пользовательские функции и собственные модули. Ты напишешь функции, которые черепашки-фермеры (и все остальные, что ты создашь впоследствии) в главе 8 будут загружать из модуля.

СОЗДАНИЕ ФУНКЦИЙ С ПОМОЩЬЮ ИНСТРУКЦИИ FUNCTION

Чтобы создать свою собственную функцию, надо использовать инструкцию `function`. Синтаксис этой инструкции состоит из ключевого слова `function`, за которым следует

имя функции, заключенное в круглые скобки. Код, помещенный между инструкциями `function` и `end`, – это тело функции.

Давай посмотрим, как это работает. Создай новую программу, выполнив команду `edit hellofunction` и введя следующий код:

hellofunction

```
1. print('Start of the program.')
2.
3. function hello()
4.   print('Hello, world!')
5. end
6.
7. hello()
8. hello()
9. print('End of the program.')
```

Затем запусти программу `hellofunction` из оболочки командной строки. Результат будет выглядеть так:

```
> hellofunction
Start of the program.
Hello, world!
Hello, world!
End of the program.
```

Строка 1 выводит сообщение `Start of the program.`, которое не относится к функции. Строка 3 создает функцию `hello()`, но обрати внимание, что функция `hello()` здесь не запускается, вместо этого исполнение переходит к строке 7.

Код функции при ее создании не исполняется, он будет исполнен только при ее вызове. Для вызова функции в программе указывается ее имя с добавлением круглых скобок. Такая конструкция сообщает программе, что следует исполнить код, содержащийся в функции с указанным именем. Строка 7 вызывает функцию `hello()`, и исполнение переходит к строке 4, где была создана эта функция. Эта строка выводит на экран сообщение `Hello, world!`.

Когда вызов функции завершается, программа возвращается к строке, которая вызывала функцию, и продолжает работу. В нашем случае, когда исполнение достигает инструкции `end` функции `hello()`, оно возвращается к строке 7, где была вызвана функция, а затем переходит к строке 8. Строка 8 также вызывает функцию `hello()`, то есть исполнение снова возвращается строке 4, и на экран еще раз выводится сообщение `Hello, world!`. Затем, когда исполнение достигает конца функции `hello()`, происходит возврат к строке 8. Наконец, строка 9 выводит текст `End of the program.`, и программа завершает работу.

Функция `hello()` очень проста, но все равно проще вызвать функцию, чем два раза набирать инструкцию вызова функции `print()`. По мере усложнения твоих программ, оформление кода в виде функций станет более актуальным, особенно после того, как ты научишься использовать аргументы.

АРГУМЕНТЫ И ПАРАМЕТРЫ

При вызове функций им можно передавать различные значения. Эти значения называют аргументами, а переменные, которым присваиваются эти аргументы, называются параметрами. Они используются внутри функции так же, как и переменные.

Чтобы узнать, как работают аргументы и параметры, создай новую программу, выполнив команду `edit sayhello` и введя код, приведенный ниже. Под управлением этой программы черепашка выводит на экран приветственное сообщение другой черепашке:

sayhello

```
1. function sayHello(name) -- name - это параметр
2.   print('Hello, ' .. name)
3. end
4.
5. sayHello('Artemisia') -- 'Artemisia' - это аргумент
6. sayHello('Elisabetta') -- 'Elisabetta' - это аргумент
```

В этой программе переменная `name` в функции `sayHello()` в строке 1 – это параметр. А строковые значения `'Artemisia'` и `'Elisabetta'` в строках 5 и 6 – это аргументы.

В строке 1 создается функция `sayHello()`, но код функции выполняется только тогда, когда происходит вызов самой функции, то есть когда исполнение программы переходит к строке 5. Эта строка содержит вызов функции `sayHello()`. Когда мы вызываем функцию подобным образом, как в строке 5, мы говорим, что передаем аргумент в вызов функции. Строка 5 передает значение `'Artemisia'` в качестве аргумента функции `sayHello()`, вызов которой возвращает исполнение программы к строке 2. Затем переменной `name`, которая является параметром функции, присваивается переданный аргумент `'Artemisia'`. В строке 2 переменная `name` используется при вызове функции `print()`, поэтому программа выводит текст `Hello, Artemisia`.

Когда исполнение программы достигает конца функции `sayHello()`, оно возвращается к строке 5, а затем переходит к строке 6. В ней снова вызывается функция `sayHello()`, но на этот раз в качестве аргумента передается значение `'Elisabetta'` и переменной `name` присваивается это значение. Поэтому, когда исполнение программы вновь возвращается к строке 2, на экран выводится сообщение `Hello, Elisabetta`.

После исполнения кода функции `sayHello()` программа возвращается к строке 6. Поскольку строк кода больше нет, программа завершается. Запусти программу `sayhello`, и она отобразит следующее:

```
> sayhello
Hello, Artemisia
Hello, Elisabetta
```

Функция `sayHello()` будет печатать строки в соответствии с переданными ей аргументами.

ВОЗВРАЩАЕМЫЕ ЗНАЧЕНИЯ

Вызовы функций почти всегда имеют возвращаемое значение, которое можно использовать подобно любому другому. Например, вызовы функции можно использовать в выражениях как переменные. Для примера в оболочке Lua введи следующую инструкцию:

```
lua> math.random(1, 6) + 1
5
```

Вызов функции `math.random(1, 6)` возвращает случайное значение от `1` до `6`. В этом примере возвращаемое значение равно `4`, поэтому все выражение вычисляется как `4 + 1`, выдавая значение `5`. (Вероятно, ты получишь другое число, так как функция `math.random()` возвращает случайное число.)

При создании функции возвращаемое значение указывается с помощью инструкции `return`. Ее синтаксис состоит из ключевого слова `return` и следующего за ним значения или выражения. Чтобы посмотреть на практике, как работает инструкция `return`, создай новую программу, выполнив команду `edit givecandy`, и введи следующий код:

givecandy

```
 1. function candiesToGive(name)
 2.   if name == 'Al' then
 3.     return 10
 4.   end
 5.
 6.   return 2
 7. end
 8.
 9. lavCandy = candiesToGive('Lavinia')
10. alCandy = candiesToGive('Al')
11. print('Lavinia gets ' .. lavCandy .. ' pieces')
12. print('Al gets ' .. alCandy .. ' pieces')
```

Функция `candiesToGive()` возвращает значение `10` или `2` в зависимости от того, какое имя передается в качестве па-

раметра name. Если передается значение 'Al', инструкция if в строке 2 принимает значение true, тогда исполнение переходит к строке 3, и вызов функции возвращает значение 10. Если функции передается какое-либо другое значение, инструкция if принимает значение false, исполнение переходит к строке 6, и функция возвращает значение 2. Аргументы можно рассматривать как входные данные функции, а возвращаемые значения как выходные.

В строке 9, где функции candiesToGive() передается значение 'Lavinia', функция возвращает значение 2, т.е. переменной lavCandy присваивается значение 2. В строке 10 переменной alCandy присваивается значение 10, потому что функция candiesToGive() возвращает значение 10, если в качестве аргумента ей передается значение 'Al'. Переменные lavCandy и alCandy можно использовать при вызове функции print(), как это делается в строках 11 и 12, потому что обе переменные сохраняют возвращаемые значения.

Если ты запустишь эту программу, то увидишь следующее:

```
> givecandy
Lavinia gets 2 pieces
Al gets 10 pieces
```

СОЗДАНИЕ МОДУЛЯ ФУНКЦИЙ

Вместо того чтобы в каждой программе заново создавать функции, можно один раз написать их и поместить в модуль. Тогда программа, которая загрузит такой модуль, сможет воспользоваться его функциями. Это называется многократным использованием кода.

Модули создаются в редакторе так же, как и программы. После создания ты можешь импортировать модули в другие программы. В этой главе ты узнаешь, как создавать модули, а в следующей главе – как их использовать в своих программах. Давай создадим модуль hare и дадим воз-

можность всем нашим черепашкам пользоваться его встроенными функциями.

Чтобы создать модуль `hare`, в оболочке командной строки выполни команду `edit hare` и введи следующий код:

hare

```
 1. --[[Модуль с функциями Эла Свейгарта
 2. Provides useful utility functions.]]
 3.
 4. -- функция selectItem() ищет ячейку
 5. -- с указанным блоком и возвращает
 6. -- true, если найдена, и false - если нет
 7. function selectItem(name)
 8.   -- проверка всех ячеек инвентаря
 9.   local item
10.   for slot = 1, 16 do
11.     item = turtle.getItemDetail(slot)
12.     if item ~= nil and item['name'] == name then
13.       turtle.select(slot)
14.       return true
15.     end
16.   end
17.
18. return false -- блок не найден
19. end
20.
21.
22. -- функция selectEmptySlot() ищет
23. -- пустую ячейку и возвращает true,
24. -- если находит, и false - если нет
25. function selectEmptySlot()
26
27.   -- перебирание всех ячеек
28.   for slot = 1, 16 do
29.     if turtle.getItemCount(slot) == 0 then
30.       turtle.select(slot)
31.       return true
32.     end
33.   end
34.   return false -- свободной ячейки нет
35. end
```

Код этого модуля состоит из 35 строк, но их нужно набрать только один раз. Без этого модуля такой код придется набирать в каждой программе с аналогичными возможностями, которую ты захочешь написать. Модули экономят много времени! В следующем разделе мы поэкспериментируем с функциями модуля `hare` в оболочке Lua.

Если при запуске этой программы возникают ошибки, тщательно сравни свой код с листингом в этой книге и исправь все опечатки. Если исправить программу не удается, удали файл, выполнив команду `delete hare`, а затем загрузи модуль с помощью команды `pastebin get wwzvaKuW hare`. (Обрати внимание, что модуль `hare`, который ты загружаешь с сайта pastebin, содержит все функции, описанные в книге, а не только те, что добавлены в этой главе.)

ЗАГРУЗКА МОДУЛЯ С ПОМОЩЬЮ ФУНКЦИИ OS.LOADAPI()

Модуль `hare` содержит две функции: `selectItem()` и `selectEmptySlot()`. Функция `selectItem()` выбирает ячейку инвентаря, содержащую указанный блок или предмет, а функция `selectEmptySlot()` выбирает первую пустую ячейку инвентаря. Обе функции весьма полезны при работе с инвентарем черепашки.

Встроенные модули Lua (такие как `math` и `os`) загружаются автоматически для всех программ черепашек, но сейчас тебе нужно загрузить модуль, созданный тобой. Для этого используется функция `os.loadAPI()`. Обрати внимание, что в ComputerCraft модули также называются программными интерфейсами (Application Programming Interfaces, API). Чтобы использовать функцию `os.loadAPI()`, передай ей в виде строкового значения имя модуля, который ты хочешь загрузить. Например, чтобы использовать в своей программе модуль `hare`, нужно ввести код `os.loadAPI('hare')`.

Функция `os.loadAPI()` сделает функции этого модуля доступными для вызывающей программы. Функция вер-

нет значение `true`, если модуль был найден и загружен, и `false`, если модуль не существует (например, если ты сделал опечатку, вот так – `os.loadAPI('har')`). В следующем разделе ты загрузишь модуль `hare` и вызовешь некоторые из его функций.

ЭКСПЕРИМЕНТЫ С МОДУЛЕМ HARE

Чтобы увидеть, как работает функция `selectItem()`, открой интерфейс черепашки и помести в инвентарь блок дубовой древесины и несколько других блоков, как показано на рис. 7.1.

Рис. **7.1.** *Инвентарь черепашки содержит блок дубовой древесины и несколько других блоков, текущая – ячейка 1*

Затем открой оболочку Lua и для выбора ячейки с дубовой древесиной введи следующий код:

```
lua> os.loadAPI('hare')
true
lua> hare.selectItem('minecraft:log')
true
```

Первая строка кода загружает модуль `hare`, а вторая – выбирает ячейку инвентаря с дубовой древесиной, передавая функции `selectItem()` строку `'minecraft:log'`.

Строка `'minecraft:log'` – это идентификатор Minecraft-блока дубовой древесины. Идентификатор – это уникальная строка, начинающаяся со значения `'minecraft:'`, которая в игре Minecraft используется для кодирования любых блоков и предметов. Список идентификаторов можно найти в разделе **«Список идентификаторов блоков»** приложения Б. В разделе **«Получение информации о содержимом ячейки»** далее в этой главе ты узнаешь, как извлечь идентификатор блока/предмета, содержащегося в ячейке, с помощью функции `turtle.getItemDetail()`, вместо того, чтобы искать его.

Если взглянуть на инвентарь черепашки после вызова функции `hare.selectItem()`, ты увидишь, что выбрана ячейка с блоком древесины, как показано на рис. 7.2. Эта функция ищет и выбирает ячейку с древесиной независимо от того, в какой ячейке инвентаря находится блок древесины. Попробуй переместить дубовую древесину в другую ячейку и снова вызови функцию `hare.selectItem('minecraft:log')`. Будет выбрана другая ячейка с древесиной, и эта ячейка будет выделена рамкой.

Рис. **7.2.** *Ячейка с древесиной выделена после выполнения функции* `hare.selectItem('minecraft:log')`

Функция `selectEmptySlot()` выбирает первую пустую ячейку инвентаря, которую обнаруживает. Эта функция понадобится, когда ты соберешься добывать какие-либо

блоки, так как для их хранения понадобится пустая ячейка. В оболочке Lua выполни следующий код:

```
lua> os.loadAPI('hare')
true
lua> hare.selectEmptySlot()
true
```

После вызова функции `hare.selectEmptySlot()` будет выбрана первая пустая ячейка, как показано на рис. 7.3.

Рис. **7.3.** *Выделена первая пустая ячейка после вызова функции* `hare.selectEmptySlot()`

Обе эти функции будут использованы в программе лесофермы из главы 8, но они могут пригодиться и во многих других программах для черепашек. Давай рассмотрим код, из которого состоит наш модуль.

РАБОТА С ИНВЕНТАРЕМ ЧЕРЕПАШКИ

Первые две строки кода модуля `hare` содержат общий комментарий к программе, а последующие комментарии поясняют, какие действия производит функция `selectItem()`.

hare

```
1. --[[Модуль с функциями Эла Свейгарта
```

```
2. для решения полезных задач.]]
3.
4. -- Функция selectItem() ищет ячейку
5. -- с указанным блоком и возвращает
6. -- true, если найден, и false - если нет
7. function selectItem(name)
```

Инструкция `function selectItem()` расположена в строке 7. Запомни, что блок кода после инструкции `function` исполняется каждый раз, когда функция вызывается. Блок кода не исполняется, если инструкция `function` встречается впервые. Чтобы понять код функции `selectItem()`, тебе сначала нужно изучить некоторые функции, которые используются для работы с инвентарем черепашки.

Каждая черепашка имеет инвентарь, состоящий из 16 ячеек, как показано на рис. 7.4.

Рис. **7.4**. *Нумерация ячеек инвентаря черепашки*

У черепашек есть три встроенные функции, которые позволяют взаимодействовать с инвентарем с помощью номеров ячеек: `turtle.select()`, `turtle.getItemCount()` и `turtle.getItemDetail()`. Давай рассмотрим, как они работают.

ВЫБОР ЯЧЕЙКИ ИНВЕНТАРЯ

Ты можешь выбрать текущую ячейку, используя функцию `turtle.select()` и передав ей номер желаемой ячейки. Номера ячеек показаны на рис. 7.4. Чтобы посмотреть, как ра-

ботает эта функция, в оболочке Lua введи следующие строки кода:

```
lua> turtle.select(2)
true
lua> turtle.select(16)
true
```

После вызова функции обрати внимание на толстую рамку, которая обозначает текущую ячейку. Многие функции черепашки используют именно текущую ячейку, поэтому весьма полезно знать, как ее выбрать. Часто бывает нужно выбрать текущей определенную ячейку, например, содержащую какой-либо нужный блок или предмет.

ПОДСЧЕТ КОЛИЧЕСТВА ПРЕДМЕТОВ В ЯЧЕЙКЕ

Если ты захочешь узнать, сколько предметов/блоков находится в ячейке инвентаря, вызови функцию `turtle.getItemCount()` и передай ей номер ячейки. В оболочке Lua введи следующие строки кода:

```
lua> turtle.getItemCount(1)
1
lua> turtle.getItemCount(16)
0
```

Если ты не передашь функции `turtle.getItemCount()` аргумент с номером ячейки, функция проверит текущую ячейку. Эта функция очень проста, она возвращает количество элементов в ячейке инвентаря, но ничего не сообщает о том, что именно в ячейке находится. Чтобы получить подробную информацию о содержимом ячейки, тебе понадобится функция `turtle.getItemDetail()`.

ПОЛУЧЕНИЕ ИНФОРМАЦИИ О СОДЕРЖИМОМ ЯЧЕЙКИ

Ты можешь получить информацию о том, что содержится в инвентаре черепашки, вызвав функцию `turtle.getItem-`

Detail() и передав ей номер ячейки, которую ты хочешь проверить. Помести блок древесины в ячейку 1, а затем в оболочке Lua введи следующий код:

```
lua> turtle.getItemDetail(1)
{
  count = 1,
  name = «minecraft:log»,
  damage = 0,
}
```

Значение, возвращаемое функцией turtle.getItemDetail(), указывает на то, что в ячейке 1 находится один блок древесины без повреждений (актуально для инструментов, оружия и брони). Если указанная ячейка пуста, функция turtle.getItemDetail() возвращает значение nil. Если функции turtle.getItemDetail() не передавать аргумент, она вернет информацию о текущей ячейке.

Возвращаемое этой функцией значение принципиально отличается от значений, которые ты видел до этого, потому что это значение табличного типа.

ТАБЛИЧНЫЕ ЗНАЧЕНИЯ

Табличные данные могут быть представлены значениями самых разных типов. Подобно тому, как строковые значения начинаются и заканчиваются кавычками, табличные значения начинаются и заканчиваются фигурными скобками {}. Хранящиеся в таблице данные образуют пары ключ-значение, которые записываются в одну строку, и эти строки с парами отделяются запятыми. Чтобы создать новую таблицу и посмотреть, как выглядят пары ключ-значение, в оболочке Lua введи следующую строку кода:

```
lua> myStuff = {logs=5, stone=4, arrows=10}
```

Эта строка присваивает переменной myStuff табличное значение {logs = 5, stone = 4, arrows = 10}. Три числовых значения – 5, 4 и 10, это значения из пары ключ-значение, а идентифицировать их можно по соответствую-

щим ключам: 'logs', 'stone' и 'arrows'. Вместе ключ и значение образуют пару ключ-значение, например logs = 5, где сначала указывается ключ, затем знак равенства (=), а затем значение. Ключи и значения могут быть представлены практически любыми типами данных, например ключ может быть целым числом, а значение – строкой. Единственное исключение состоит в том, что ключ таблицы не может быть значением nil.

Чтобы получить доступ к отдельным значениям внутри таблицы, укажи переменную myStuff, а затем в квадратных скобках [] ключ. Такое выражение вернет значение таблицы, связанное с этим ключом. Например, для доступа к значению таблицы, которую мы создали, в оболочке Lua введи следующее:

```
lua> myStuff['logs']
5
lua> 'I have ' .. myStuff['stone'] .. ' stone.'
I have 4 stone.
lua> 'I need ' .. (12 - myStuff['arrows']) .. ' more
arrows to have a dozen.'
I need 2 more arrows to have a dozen.
```

Первая строка кода возвращает значение, связанное с ключом 'log', это число 5. Поскольку выражение вычисляет значение, его можно использовать так же, как и любое другое значение, например при конкатенации строк или при выполнении математических действий (если это значение является числом).

ОБЗОР ТАБЛИЦЫ, ВОЗВРАЩАЕМОЙ ФУНКЦИЕЙ TURTLE.GETITEMDETAIL()

Давай снова рассмотрим функцию turtle.getItemDetail(). Помести один деревянный блок в ячейку 1, а затем, в оболочке Lua введи следующее:

```
lua> item = turtle.getItemDetail(1)
lua> item['name']
```

```
minecraft:log
lua> item['count']
1
```

Вводя код `['name']`, ты получаешь значение ключа `name` таблицы, т.е. `minecraft:log`. Точно так же можно получить и другие значения, например связанные с ключом `count`. Теперь у тебя есть способ получить информацию об инвентаре черепашки в виде строковых и числовых значений! Обрати внимание: если функция `turtle.getItemDetail()` возвращает значение `nil`, потому что ячейка пуста, попытка использовать квадратные скобки приведет к ошибке:

```
lua> item = turtle.getItemDetail(16)
lua> item['name']
lua:1: attempt to index ? (a nil value)
```

Чтобы эта ошибка не возникала, следует использовать инструкцию `if` и убедиться, что переменной `item` не присвоено значение `nil`. Хотя этот код, мы и не будем использовать в этой главе, в будущих программах он нам пригодится:

```
local item = turtle.getItemDetail(16)
if item ~= nil then
    print(item['name'])
end
```

С помощью этого кода, вызов `item['name']` исполняется только в том случае, если значение переменной `item` не равно (~=) `nil`.

ГЛОБАЛЬНАЯ И ЛОКАЛЬНАЯ ОБЛАСТИ ВИДИМОСТИ

Функция `selectItem()` должна найти в инвентаре черепашки блок или предмет, имя которого соответствует строке, передаваемой параметром `name` функции. Функция в строке 9 объявляет локальную переменную с именем `item`.

Эта локальная переменная существует только внутри функции `selectItem()` и не может быть использована в каком-либо другом месте кода программы.

hare

```
 7. function selectItem(name)
 8.   -- проверка всех ячеек инвентаря
 9.   local item
```

Ты уже встречался с ключевым словом `local` в разделе **«Логические типы данных»** главы 5, но ты еще не знаешь об области видимости. Область видимости – это часть программы, в которой может использоваться переменная. В языке Lua предусмотрено два типа области видимости переменных: глобальная и локальная.

Переменные, объявленные в глобальной области видимости, существуют вне какой-либо функции и доступны во всех частях программы. Это может вызвать проблемы, если ты будешь неосмотрительно использовать имя такой переменной, потому что, с одной стороны, программа будет помнить значение этой переменной, присвоенное ей в другом месте, с другой стороны, будет позволять любым функциям перезаписывать в эту переменную новые значения. Чтобы таких проблем не возникало, можно создавать переменные в локальной области видимости. Такие переменные создаются каждый раз при обращении к функции. Когда переменная объявляется локально, она существует только внутри функции и перестает существовать после ее завершения.

Используя локальные переменные, можно использовать несколько переменных с одинаковым именем, если они будут существовать в разных областях видимости. Поскольку в будущем ты добавишь в модуль `hare` больше функций, мы объявим переменную `item` локально, только для функции `selectItem()`. Даже если тебе понадобится переменная, к которой можно получить доступ в пространстве всей программы, все равно лучше объявлять переменные локально в каждой функции, это позволит избежать проблем.

Переменные, используемые в цикле `for`, такие как переменная `i` в выражении `for i = 1, 4 do`, являются локальными переменными в цикле `for`. Эти переменные не будут существовать вне цикла, так же как локальные переменные функций не будут существовать вне этих функций.

ПОИСК ПРЕДМЕТА С ПОМОЩЬЮ ЦИКЛА FOR

После объявления всех переменных в строке 10 создается еще одна переменная с именем `slot`, которая является локальной для цикла `for`.

hare

```
10.   for slot = 1, 16 do
11.     item = turtle.getItemDetail(slot)
12.     if item ~= nil and item['name'] == name then
13.       turtle.select(slot)
14.       return true
15.     end
16.   end
17.
18. return false -- блок не найден
19. end
```

Переменная `slot` в выражении `for slot = 1, 16 do` не будет существовать после завершения цикла, т.е. после выполнения инструкции `end`. Цикл `for` перебирает все номера ячеек от 1 до 16. Внутри цикла, в строке 11, переменной `item` присваивается табличное значение, возвращаемое функцией `turtle.getItemDetail(slot)`. Код в строке 12 проверяет, что значение переменной `item` не равно `nil`, а также проверяет, совпадает ли имя предмета/блока с параметром `name`. Совпадение означает нахождение искомого элемента. Строка 14 возвращает значение `true`, таким образом сообщая программе, что функция `selectItem()` выделила ячейку с искомым предметом/блоком.

Инструкция `end` в строке 15 завершает конструкцию `if`, а инструкция `end` в строке 16 завершает весь цикл. Если

исполнение совершило все итерации цикла `for` и не обнаружило предмет/блок с соответствующим именем, функция возвращает значение `false` (код в строке 18). Наконец, инструкция `end` в строке 19 завершает функцию.

ВЫБОР ПУСТОЙ ЯЧЕЙКИ ИНВЕНТАРЯ

Функция `selectEmptySlot()` пытается найти пустую ячейку в инвентаре. Встретив первую пустую ячейку, функция выбирает ее при помощи вызова `turtle.select()` и возвращает значение `true`. Если инвентарь черепашки полностью заполнен, функция вернет значение `false`. Как и `selectItem()`, функция `selectEmptySlot()` содержит цикл `for`, который проверяет ячейки с 1 по 16. Цикл `for` в этой функции также объявляет переменную `slot` в строке 28. Это возможно, поскольку эта переменная `slot` и переменная `slot`, объявленная в строке 10, существуют в разных локальных областях видимости.

hare

```
25. function selectEmptySlot()
26
27.   -- перебирание всех ячеек
28.   for slot = 1, 16 do
29.     if turtle.getItemCount(slot) == 0 then
30.       turtle.select(slot)
31.       return true
32.     end
33.   end
34.   return false -- свободного места нет
35. end
```

Внутри цикла код в строке 29 вызывает функцию `turtle.getItemCount()`, чтобы проверить, равно ли нулю количество предметов/блоков в проверяемой ячейке. Когда код в строке 29 находит первую пустую ячейку, код в строке 30 выбирает эту ячейку, а код в строке 31 возвращает значение `true`.

Если выполнены все итерации цикла `for`, а свободная ячейка не обнаружена, код в строке 34 возвращает значение `false`.

ДОПОЛНИТЕЛЬНОЕ ЗАДАНИЕ: МОДУЛЬ ТАНЦЕВАЛЬНЫХ ДВИЖЕНИЙ

Попробуй создать новый модуль танцевальных движений, которые сможет выполнять черепашка. Ты можешь написать функции для выполнения прыжков, переворотов и лунной походки (движение задом).

ЧТО МЫ УЗНАЛИ

В этой главе ты узнал, как создавать модули и свои собственные функции при помощи инструкции `function`. Функции могут иметь входные данные или параметры, которые передаются в качестве аргументов. Функции также могут возвращать значение программе, которая ее вызвала.

При каждом вызове функции создается новая локальная область видимости. Когда функция завершается, эта область видимости перестает существовать вместе со всеми переменными в ней. Это значит, что переменные могут иметь одно и то же имя, если они объявляются в разных областях видимости. Переменная цикла `for` также существуют только в своей локальной области видимости, внутри цикла. Функции твоего модуля `hare` вызывают другие функции ComputerCraft: `turtle.select()`, `turtle.getItemDetail()` и `turtle.getItemCount()`.

Ты можешь размещать функции, которые пишешь, внутри модулей, чтобы другие программы могли их использовать. Создание собственных функций и модулей – это метод программирования, который позволяет разрабатывать более сложные программы. В главе 8 ты будешь использовать модуль `hare` при создании лесофермы, на которой выращенные деревья будут вырубаться черепашками в автоматическом режиме.

8

ЗАПУСК АВТОМАТИЗИРОВАННОЙ ЛЕСОФЕРМЫ

В главе 6 мы запрограммировали черепашку на рубку деревьев, но перед запуском программы игроку приходилось вручную ставить черепашку перед каждым деревом. Установка черепашки перед каждым деревом, которое ты хочешь срубить, не намного эффективнее, чем рубить деревья вручную. Но ручной труд можно исключить и полностью автоматизировать весь процесс производства древесины, а заодно запрограммировать черепашку на посадку саженцев и выращивание деревьев. Такая черепашка будет высаживать саженец, срубать дерево, когда оно вырастет, и сажать новый саженец. Весь этот процесс она должна повторять без какого-либо вмешательства со стороны человека! Чтобы сделать это, мы напишем программу под названием `farmtrees`. Тогда ты сможешь настроить несколько черепашек на выращивание леса и заготовку древесины, как показано на рис. 8.1.

Мы возьмем модуль `hare` (из главы 7), чтобы повторно использовать уже написанный код, и программу `choptree`

(из главы 6). Так у нас не будет необходимости переписывать весь этот код!

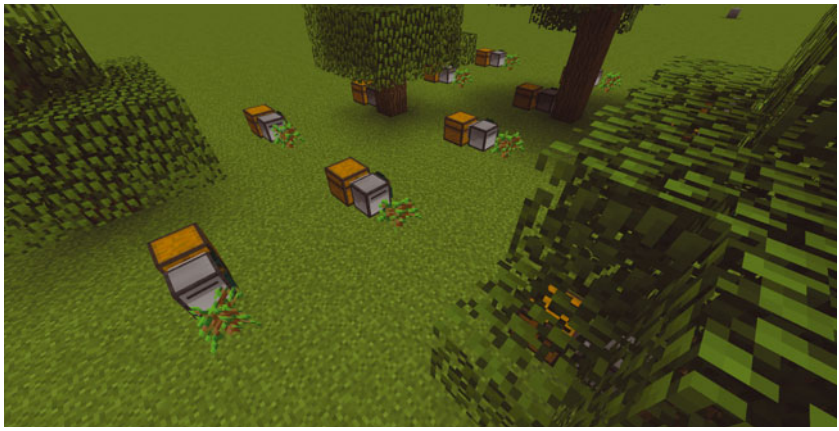


Рис. **8.1.** *Автоматизированная лесоферма, обслуживаемая черепашками под управлением программы* `farmtrees`

ПРОЕКТИРОВАНИЕ ПРОГРАММЫ ЛЕСОФЕРМЫ

Избавимся от необходимости ставить черепашку перед каждым деревом. Оставим ее на одном месте и запрограммируем самостоятельно сажать и выращивать саженцы, а также использовать костную муку, чтобы ускорить их рост. Затем, как только саженец превратится в дерево, черепашка срубит его и поместит заготовленную древесину в сундук.

Вот несколько шагов, которые должна выполнять программа лесозаготовки:

1. Убедиться, что модуль `hare` и программа `choptree` существуют.
2. Выбрать саженцы деревьев в инвентаре черепашки. Закрыть программу, если саженцев нет.
3. Посадить саженец.
4. Посаженный саженец периодически удобрять костной мукой, пока он не вырастет.
5. Запустить программу `choptree`.

6. Поместить собранную древесину в сундук.
7. Повторить весь процесс.

Черепашка будет повторять весь процесс до тех пор, пока не закончатся саженцы деревьев или костная мука. Теперь, когда мы знаем, что должен делать код, давай напишем его.

КОД ПРОГРАММЫ FARMTREES

В оболочке командной строки запусти текстовый редактор, выполнив команду `edit farmtrees`. В текстовом редакторе введи показанные ниже строки кода. Помни, что номера строк указывать не нужно. В книге они для удобства, чтобы можно было обращаться к строкам по их номерам.

farmtrees

```
 1. --[[Программа лесофермы Эла Свейгарта
 2. Выращивание и рубка деревьев.]]
 3.
 4. os.loadAPI('hare') -- загрузка модуля hare
 5.
 6. local blockExists, item
 7. local logCount = 0
 8.
 9. -- проверка существования программы choptree
10. if not fs.exists('choptree') then
11.   error('You must install choptree program first.')
12. end
13.
14. while true do
15.   -- проверка наличия саженцев в инвентаре
16.   if not hare.selectItem('minecraft:sapling') then
17.     error('Out of saplings.')
18.   end
19.
20.   print('Planting...')
21.   turtle.place() -- высадка саженца
22.
```

```
23.   -- цикличное выполнение, пока дерево растет
24.   while true do
25.     blockExists, item = turtle.inspect()
26.     if blockExists and item['name'] ==
        'minecraft:sapling' then
27.       -- «dye» - идентификатор костной муки
28.       if not hare.selectItem('minecraft:dye') then
29.         error('Out of bone meal.')
30.       end
31.
32.       print('Using bone meal...')
33.       turtle.place() -- использование костной муки
34.     else
35.       break -- дерево выросло
36.     end
37.   end
38.
39.   hare.selectEmptySlot()
40.   shell.run('choptree') -- запуск программы choptree
41.
42.   -- перемещение и поворот к сундуку
43.   turtle.back()
44.   turtle.turnLeft()
45.   turtle.turnLeft()
46.
47.   -- помещение древесины в сундук
48.   while hare.selectItem('minecraft:log') do
49.     logCount = logCount + turtle.getItem Count()
50.     print('Total logs: ' .. logCount)
51.     turtle.drop()
52.   end
53.
54.   -- возвращение к месту высадки саженца
55.   turtle.turnLeft()
56.   turtle.turnLeft()
57. end
```

Сохрани программу после ввода всех инструкций. Нажав клавишу **Ctrl**, открой меню. Выбери пункт **[Save]** и нажми клавишу **Enter**. Затем выйди из редактора, нажав клавишу **Ctrl**, выбрав пункт **[Exit]** и нажав клавишу **Enter**.

ЗАПУСК ПРОГРАММЫ FARMTREES

Перед запуском программы `farmtrees` нужно выполнить небольшую подготовку. Во-первых, убедись, что сундук находится прямо за черепашкой, а в инвентаре черепашки есть саженцы и костная мука, как показано на рис. 8.2.

Рис. **8.2.** *Установи сундук позади черепашки, а саженцы и костную муку положи в ее инвентарь*

Костную муку можно изготовить из костей, которые выпадают из убитых скелетов. Для изготовления костной муки используй рецепт, показанный на рис. 8.3.

Рис. **8.3.** *Создание трех кучек костной муки из одной кости*

Во-вторых, убедись, что ты оставил одну ячейку инвентаря пустой, так как в ней будет сохраняться древесина, которую вырубит черепашка.

Когда закончишь все приготовления, запусти программу `farmtrees`. Ты должен увидеть, как черепашка сажает саженец, а затем, когда он вырастает в дерево, срубает его. Потом черепашка вернется на землю и подойдет к сундуку,

чтобы выгрузить в него добытую древесину. После разгрузки черепашка развернется в исходное положение и повторит весь процесс заново.

Если при запуске программы возникают ошибки, тщательно сравни свой код с листингом в этой книге, чтобы найти возможные опечатки. Если программу исправить не удается, удали файл, выполнив команду `delete farmtrees`, а затем загрузи корректный код из Интернета, выполнив команду `pastebin get v5h8AgGs farmtrees`.

ТИПЫ ДЕРЕВЬЕВ В MINECRAFT

В Minecraft растут деревья самых разнообразных типов, каждое из которых имеет свой собственный саженец. Одни деревья при использовании программы `farmtrees` подходят лучше, другие – хуже. Черепашка будет вырубать дерево, двигаясь по стволу прямо вверх. Следовательно, если дерево ветвистое, черепашка будет обрабатывать его довольно медленно, ведь ей придется передвигаться по всему стволу. Поэтому для заготовки наиболее удобны деревья с высокими узкими стволами, такие как дуб, ель, береза и тропические деревья, показанные на рис. 8.4. Акация слишком ветвиста, а каждый саженец темного дуба следует сажать на пространстве 2 × 2 блока, то есть потребуется слишком много места для выращивания.

Рис. **8.4.** *Слева направо: дуб, ель, береза и тропическое дерево*

При вызове функции `hare.selectItem()` в строке 16, у всех саженцев будет одно и то же имя – `'minecraft:sapling'`[6], поэтому убедись, что это дуб, ель, береза или тропическое дерево, когда будешь помещать саженцы в инвентарь черепашки.

ЗАГРУЗКА ЧАНКОВ В MINECRAFT

В программе `farmtrees` тебе не нужно контролировать работу черепашки, но все же отходить от нее слишком далеко тоже не стоит. Это связано с особенностями генерации мира Minecraft. Фактически мир бесконечен, потому что новые части мира генерируются там, где находишься ты. Однако, поскольку для загрузки всего мира требуется слишком много памяти, игра Minecraft загружает только те области, в которых находится игрок. Мир Minecraft разделен на зоны в 16 блоков в длину и 16 блоков в ширину[7], которые называются чанками и которые подгружаются в игровой мир по мере перемещения игрока. При путешествии игрока по миру новые чанки загружаются в оперативную память компьютера, а чанки, от которых он удаляется, выгружаются.

Этот момент необходимо учитывать при работе с черепашками, потому что они могут быть выгружены вместе с чанками, в которых они находятся. Например, на моем компьютере черепашки выгружаются, если я удаляюсь от них на расстояние в 400–450 блоков. А если на своем компьютере ты уменьшил значение параметра **Дальность прорисовки** (Render Distance) (его можно найти в разделе **Настройки графики** (Video Settings)), черепашки будут выгружаться еще раньше. Таким образом, если ты создашь и разместишь несколько черепашек-фермеров и уйдешь от них слишком далеко, они перестанут работать. Поэтому лучше держать под присмотром работающих черепашек и размещать их как можно ближе друг к другу.

Теперь, когда ты знаешь особенности генерации мира Minecraft, давай подробнее рассмотрим программу `farmtrees`.

[6] С версии 1.13 разные *(прим. перев.)*

[7] И 256 блоков в глубину *(прим. перев.)*

ЗАГРУЗКА МОДУЛЕЙ С ПОМОЩЬЮ ФУНКЦИИ OS.LOADAPI()

После комментариев, описывающих программу в строках 1 и 2, код в строке 4 загружает модуль `hare`, который ты создал в главе 7.

farmtrees

```
1. --[[Программа лесофермы Эла Свейгарта
2. Выращивание и рубка деревьев.]]
3.
4. os.loadAPI('hare') -- загрузка модуля hare
5.
6. local blockExists, item
7. local logCount = 0
```

Чтобы программа могла вызывать функции `hare.selectEmptySlot()` и `hare.selectItem()`, которые были описаны в главе 7, сначала нужно вызвать функцию `os.loadAPI()` и передать ей строку `'hare'`, чтобы загрузить этот модуль.

В строках 6 и 7 объявляются переменные, которые упрощают поиск. Параметры `blockExists` и `item` сохраняют значения, которые сообщат черепашке, вырос ли посаженный саженец в дерево. Переменная `logCount` отслеживает объемы урожая, при первом запуске программы ее значение равно `0`.

ПРОВЕРКА ФАЙЛОВ С ПОМОЩЬЮ ФУНКЦИИ FS.EXISTS()

Прежде чем черепашка начнет выращивать деревья, она должна проверить, что программа `choptree` существует. Иначе, если черепашка попытается запустить программу `farmtrees`, ты увидишь сообщение об ошибке.

farmtrees

```
 9. -- проверка существования программы choptree
10. if not fs.exists('choptree') then
```

```
11.   error('You must install choptree program first.')
12. end
```

Наличие программы `choptree` можно проверить с помощью функции `fs.exists()`, которая принимает имя файла и возвращает значение `true`, если файл с этим именем существует, и `false` – если нет. Если программа `choptree` не существует, код в строке 11 завершает работу программы и отображает сообщение об ошибке `'You must install choptree program first.'`.

ВЫБОР САЖЕНЦЕВ В ИНВЕНТАРЕ ЧЕРЕПАШКИ

Код в строке 14 начинает цикл `while`, который сажает деревья и собирает древесину. Каждый раз, когда программа заканчивает вырубку выросшего дерева, она возвращается к строке 14, чтобы повторить процесс. Поскольку условие цикла `while` всегда истинно, это бесконечный цикл, исполнение которого не прекратится до тех пор, пока программа не завершится вызовом функции `error()` или инструкции `break()`, действие которой я объясню в разделе **«Прерывание цикла с помощью инструкции `break`»** далее в этой главе.

Внутри цикла `while` программа сначала проверяет наличие саженцев в инвентаре черепашки.

farmtrees

```
14. while true do
15.   -- проверка наличия саженцев в инвентаре
16.   if not hare.selectItem('minecraft:sapling') then
17.     error('Out of saplings.')
18.   end
```

В строке 16 используется функция `selectItem()` модуля `hare` для выбора ячейки, содержащей саженцы. Если в инвентаре черепашки саженцев нет, функция возвращает значение `false`. В этом случае код в строке 17 завершает программу, вызывая функцию `error()` и отображая сообщение `'Out of saplings.'`.

ПОСАДКА ДЕРЕВА

Если проверка на наличие саженцев пройдена успешно, код в строке 20 отображает сообщение о том, черепашка собирается посадить саженец (`'Planting...'`). Код в строке 21 вызывает функцию `turtle.place()`, которая производит высадку.

farmtrees

```
20.   print('Planting...')
21.   turtle.place() -- высадка саженца
```

Функция `turtle.place()` помещает блок из текущей ячейки инвентаря перед черепашкой. Эта функция применима для саженцев, досок, каменных кирпичей или любых других блоков, которые игрок захочет разместить в мире Minecraft. Напомню, что в строке 16 выбрана ячейка инвентаря с саженцами, поэтому функция `turtle.place()` высаживает саженец. На рис. 8.5 показан саженец после того, как он был посажен черепашкой.

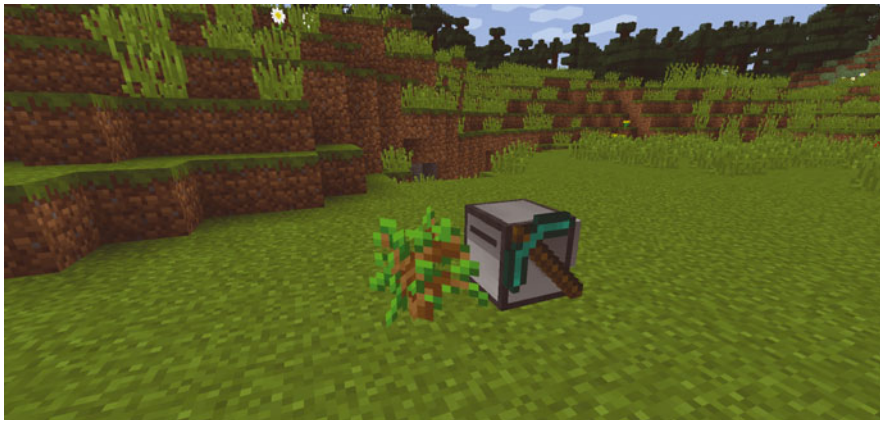


Рис. **8.5.** *Саженец, посаженный черепашкой*

Примечание *Если понадобится, ты также можешь использовать функции `turtle.placeUp()` и `turtle.placeDown()` для размещения блоков выше и ниже черепашки. Эти функции возвращают значение `true`, если установка блока произошла успешно. Если что-то мешает размеще-*

нию блока, например другой блок, эти функции возвращают значение `false`.

ПРОВЕРКА БЛОКОВ И ОЖИДАНИЕ РОСТА ДЕРЕВЬЕВ

После посадки саженца исполнение переходит в другой цикл `while`, который вложен внутрь предыдущего, начавшегося в строке 14. Этот внутренний цикл `while` начинается в строке 24. С помощью него программа проверяет блок, стоящий перед черепашкой, используя функцию `turtle.inspect()`, которая похожа на `turtle.detect()`, за исключением того, что функция `turtle.detect()` возвращает только значение `true` или `false`, если присутствует блок, тогда как функция `turtle.inspect()` возвращает табличное значение со сведениями о том, какой блок находится перед черепашкой.

farmtrees

```
23.   -- цикличное выполнение, пока дерево растет
24.   while true do
25.     blockExists, item = turtle.inspect()
```

Функция `turtle.inspect()` возвращает два значения, которые хранятся в переменных `blockExists` и `item`, объявленных в начале программы. Первое возвращаемое значение (хранящееся в переменной `blockExists`) является логическим значением `true`, если блок находится перед черепашкой, или `false` – если блока нет. Второе возвращаемое значение (хранящееся в переменной `item`) – это табличное значение с информацией о блоке. Если перед черепашкой нет блока, второе возвращаемое значение будет равно `nil`.

Запустив функцию `turtle.inspect()` в оболочке Lua, можно получить имя любого блока, если он расположен прямо перед черепашкой. Например, если перед черепашкой находится саженец, то при запуске этой команды ты увидишь следующее:

```
lua> turtle.inspect()
true
```

```
{
  state = {
    type = «oak»,
    stage = 0,
  },
  name = «minecraft:sapling»,
  metadata = 0,
}
```

Инструкция `if` в строке 26 проверяет, находится ли перед черепашкой блок и является ли он саженцем. Если это так, то код в строках 28–33 будет неоднократно удобрять его костной мукой, чтобы ускорить рост саженца.

farmtrees

```
26.     if blockExists and item['name'] ==
        'minecraft:sapling' then
27.       -- «dye» - идентификатор костной муки
28.       if not hare.selectItem('minecraft:dye') then
29.         error('Out of bone meal.')
30.       end
```

Код в строке 28 проверяет наличие костной муки в инвентаре черепашки. Идентификатор костной муки в игре Minecraft – `'minecraft:dye'`[8]. Подобно тому, как саженцы разных деревьев имеют один и тот же идентификатор `'minecraft:sapling'`, разные красители (костная мука относится к белым красителям) также имеют один и тот же идентификатор.

Если в инвентаре черепашки нет костной муки, функция `hare.selectItem('minecraft:dye')` возвращает значение `false`, а код в строке 29 завершает программу сообщением об ошибке `'Out of bone meal.'`.

Если костная мука в инвентаре есть, исполнение продолжается до строки 32, где программа выводит на экран сообщение о том, что собирается сделать черепашка (`'Using bone meal...'`), а затем посыпает костной мукой саженец, который растет перед черепашкой.

[8] В версии 1.13 – 'minecraft:bone_meal' *(прим. перев.)*

farmtrees

```
32.       print('Using bone meal...')
33.       turtle.place() -- использование костной муки
```

Исполнение пропускает инструкцию `else` в строке 34, потому что ее условие ложно, и, так как в цикле `while` больше нет строк кода, исполнение снова переходит к строке 24.

ПРЕРЫВАНИЕ ЦИКЛА С ПОМОЩЬЮ ИНСТРУКЦИИ BREAK

Если условие в строке 26 ложно, и перед черепашкой не обнаружено саженца, исполнени переходит к строке 34, и выполняется блок кода после инструкции `else`. Саженец исчез, потому что он превратился в дерево, и теперь перед черепашкой стоит деревянный блок. Обработка инструкции `break` в строке 35 приводит к тому, что программа завершает цикл `while`, начавшийся в строке 24.

farmtrees

```
34.     else
35.       break -- дерево выросло
36.     end
37.   end
```

Инструкция `break` состоит только из ключевого слова `break`. Выполнение этой инструкции приводит к тому, что программа немедленно выходит из цикла `while`, не перепроверяя его условие. То есть инструкция `break` в строке 35 перемещает исполнение в первую строку за пределами цикла, в строку 39.

Код в строке 36 завершает блок кода инструкции `else`, а код в строке 37 завершает цикл `while`.

ЗАПУСК ДРУГИХ ПРОГРАММ С ПОМОЩЬЮ ФУНКЦИИ SHELL.RUN()

Функция `shell.run()` позволяет программе выполнять команды так же, как и в оболочке командной строки. Про-

грамма farmtrees использует эту функцию для запуска программы choptree.

farmtrees

```
39.   hare.selectEmptySlot()
40.   shell.run('choptree') -- запуск программы choptree
```

Код в строке 39 вызывает функцию selectEmptySlot() модуля hare, которая проверяет, что добытая черепашкой древесина будет сохранена в пустую ячейку. (В противном случае программа choptree не запустится.) Затем код в строке 40 запускает программу choptree, вызывая функцию shell.run() и передавая ей строковое значение 'choptree'.

Строка, переданная в функцию shell.run(), аналогична команде, которую ты вводишь в оболочке командной строки. Вызов функции shell.run('choptree') в программе приводит к тому же результату, что и запуск программы choptree в оболочке командной строки.

Так твоя программа может запускать любые другие программы. Когда программа choptree завершается, исполнение переходит к строке 40.

ВЫГРУЗКА ЧЕРЕПАШКОЙ ДОБЫТОЙ ДРЕВЕСИНЫ

В конце программы choptree черепашка спускается на землю, где изначально был посажен саженец. Затем она возвращается в исходное положение, отступая на один блок, и дважды поворачивается налево, чтобы оказаться перед сундуком и выгрузить в него древесину.

farmtrees

```
42.   -- перемещение и поворот к сундуку
43.   turtle.back()
44.   turtle.turnLeft()
45.   turtle.turnLeft()
```

Функция `turtle.drop()` выгружает содержимое текущей ячейки инвентаря в сундук, перед которым она находится (или на землю, если сундук отсутствует). Если передать функции `turtle.drop()` число, то из текущей ячейки можно выгрузить указанное количество блоков. Например, показанная ниже команда в оболочке Lua выгружает только один блок:

```
lua> turtle.drop(1)
```

Примечание. *Если понадобится, ты также можешь использовать функции `turtle.dropUp()` и `turtle.dropDown()` для выгрузки блоков выше и ниже черепашки, а не перед ней.*

Цикл `while` выбирает существующие в инвентаре черепашки блоки древесины. Одновременно внутри цикла подсчитывается количество блоков древесины, которое выгружается в сундук.

farmtrees

```
47.   -- помещение древесины в сундук
48.   while hare.selectItem('minecraft:log') do
49.     logCount = logCount + turtle.getItemCount()
50.     print('Total logs: ' .. logCount)
51.     turtle.drop()
52.   end
```

Вызов функции `hare.selectItem()` в строке 48 возвращает значение `true`, если обнаружен элемент с идентификатором `'minecraft:log'`, а это значит, что цикл `while` будет продолжаться до тех пор, пока в инвентаре черепашки есть древесина.

Вызов функции `turtle.getItemCount()` возвращает количество блоков в текущей ячейке. Это значение в строке 49 добавляется к числу, хранящемуся в переменной `logCount`. Таким образом, отслеживается общее количество блоков древесины, скопившихся в сундуке, а код в строке 50 отображает это общее количество. В строке 51 древесина

выгружается из инвентаря черепашки в сундук перед ней с помощью функции `turtle.drop()`.

Код в строках 55 и 56 дважды поворачивают черепашку налево, возвращая ее в исходное положение, лицом к ферме. Код в строке 57 завершает цикл `while`, начавшийся в строке 14, поэтому исполнение возвращается к строке 14. Теперь черепашка готова посадить следующий саженец.

farmtrees

```
54.    -- возвращение к месту высадки саженца
55.    turtle.turnLeft()
56.    turtle.turnLeft()
57. end
```

Напомню, что цикл `while`, начавшийся в строке 14, бесконечен. Программа `farmtrees` завершится только в том случае, если будет вызвана функция `error()` в строке 17 или 29. Это может произойти, если у черепашки закончатся либо саженцы, либо костная мука. Игрок может пополнить запасы черепашки даже во время выполнения программы. Пока игрок снабжает черепашку саженцами и костной мукой, программа будет выполняться бесконечно.

ИЗМЕНЕНИЕ КОДА, ЕСЛИ НЕТ КОСТНОЙ МУКИ

По мере того, как улучшаются твои навыки программирования, у тебя могут возникать идеи по улучшению своих программ уже после того, как они написаны. Переписывать программы можно сколько угодно. Например, ты можешь обнаружить, что запас костной муки черепашки расходуют быстрее, чем ты убиваешь скелетов. Роботы не умеют скучать, поэтому черепашку можно запрограммировать на ожидание, пока дерево вырастет естественным образом, вместо того, чтобы прекращать работу из-за отсутствия костной муки.

В настоящее время черепашки запрограммированы на полное прерывание своей работы, если в инвентаре нет костной муки. Чтобы перепрограммировать их на ожидание

естественного роста дерева без костной муки, измени код в строке 29 (`error('Out of bone meal.')`) на следующее:

farmtrees

```
29.         os.sleep(10)
```

Функция `os.sleep()` приостанавливает программу на заданное время. Измененный код в строке 29 «усыпит» черепашку на 10 секунд, прежде чем она продолжит выполнение остальной части программы. После паузы в 10 секунд черепашка проверяет, не вырос ли саженец в дерево.

Другое преимущество функции `os.sleep()` вместо простого цикла заключается в том, что при вызове функции `os.sleep()` Minecraft может прекратить исполнение кода черепашки, а затем возобновить его после паузы. Приостановка исполнения одной программы может предотвратить задержку всего производства, особенно если в нем участвуют несколько черепашек.

ДОПОЛНИТЕЛЬНОЕ ЗАДАНИЕ: ЧЕРЕПАШКА-ТРУДОГОЛИК

Нет нужды доверять черепашке заниматься только одним деревом. Ты можешь посадить несколько деревьев в ряд и поручить их обработку одной черепашке, чтобы она проверяла все деревья каждые несколько минут. Напиши программу, которая будет запускать программу `farmtrees` для нескольких деревьев, как показано на рис. 8.6.

Придется изменить код программы `farmtrees` так, чтобы черепашка проводила проверку, выросло ли из саженца дерево, а потом переходила к следующему дереву, вместо того, чтобы постоянно проверять одно и то же, в цикле. Таким образом, ты сможешь повторно использовать код программы `farmtrees` и строить гораздо более крупные фермы.

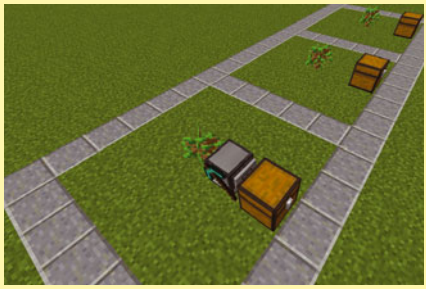


Рис. **8.6.** *Одна черепашка ухаживает за саженцами на трех участках*

ЧТО МЫ УЗНАЛИ

Главная ценность программы `farmtrees` в том, что она может масштабироваться при добавлении черепашек. Если ты добавишь десять черепашек, производство древесины возрастет. Эта древесина будет полезна не только в качестве строительного материала, ее также можно сжечь в печи, чтобы превратить в древесный уголь. Его можно использовать как топливо для черепашек, а это значит, что этим черепашкам уже не понадобится человек, который будет их заправлять.

В этой главе ты узнал о разных типах деревьев, об областях видимости переменных и о том, что черепашки продолжают работать только до тех пор, пока загружена область мира Minecraft с чанками, в которых они существуют.

Ты также изучил некоторые дополнительные функции черепашки. Функция `fs.exists()` проверяет, существует ли указанная программа. В программе `farmtrees` функция `turtle.place()` сажала в землю саженцы, но ты также можешь использовать ее для размещения перед черепашкой любых других блоков. Функция `turtle.inspect()` позволяет черепашке определять, какой блок находится перед ней. Функция `turtle.drop()` выгружает содержимое ячейки перед черепашкой на землю или в сундук. Функция `shell.run()` позволяет программе `farmtrees` запускать другие программы, такие как `choptree`.

Инструкция `break` приводит к немедленному выходу программы из цикла `while` или `for`. Функция `os.sleep()` позволяет приостановить исполнение программы на указанное количество секунд.

В главе 9 мы расширим наше автоматизированное производство и начнем создавать каменные кирпичи. Наш заводик создаст необходимые для строительства блоки, и тебе больше не придется лезть за ними в шахту!

9

СТРОИТЕЛЬСТВО ГЕНЕРАТОРА БУЛЫЖНИКА

Самые распространенные блоки, которые только можно найти, – это каменные. Из них добывается бесформенный булыжник, но его можно переплавить в камень, из которого крафтятся каменный кирпичи – их ты можешь использовать для строительства красивых зданий. Уф! Как много предстоит работы в опасных темных шахтах, прежде чем можно будет получить каменные кирпичи.

В этой главе ты узнаешь, как создать бесконечный генератор булыжника, а затем ты напишешь программу для черепашек, позволяющую автоматизировать процессы добычи булыжника и его переплавки. Разместив производственную линию в безопасном месте около дома, ты обеспечишь себя неограниченным количеством каменных кирпичей, пригодных для строительства.

ПРОЕКТ ГЕНЕРАТОРА БУЛЫЖНИКА

Ты уже знаешь, что в Minecraft булыжник в основном добывается в шахтах, но его также можно получить, смешивая

текущую воду с текущей лавой, в результате чего образуется булыжник. Мы применим эти знания для создания генератора, который будет создать неограниченное количество блоков булыжника. Черепашка сможет добывать булыжник бесконечно долго, так как ее инструменты не изнашиваются.

Чтобы создать генератор булыжника, следуй схеме, показанной на рис. 9.1. У меня генератор построен из стекла, но ты можешь использовать любые другие негорючие блоки. Также тебе понадобится ведро воды и ведро лавы. Ведро можно создать из трех железных слитков, а затем тебе следует зачерпнуть лаву и воду. Хотя источники лавы можно найти на поверхности, чаще всего они встречаются глубоко под землей, вблизи бедрока. Воду можно взять из реки, озера или океана.

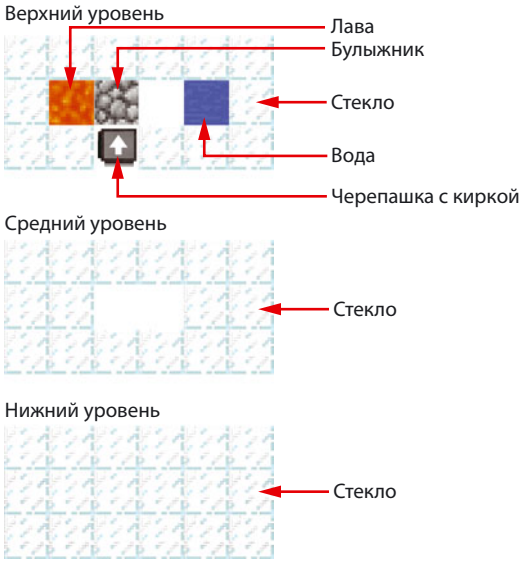


Рис. **9.1.** *Взгляд с высоты птичьего полета на генератор булыжника*

При размещении лавы и воды обязательно вылей лаву из ведра первой. В противном случае поток воды будет смешиваться с неподвижной лавой (а не с текущей), превращая ее в обсидиан. Блок булыжника, показанный на рисунке, в генераторе размещать не нужно. Он будет сформирован автоматически.

На рис. 9.2 показан готовый генератор булыжника.

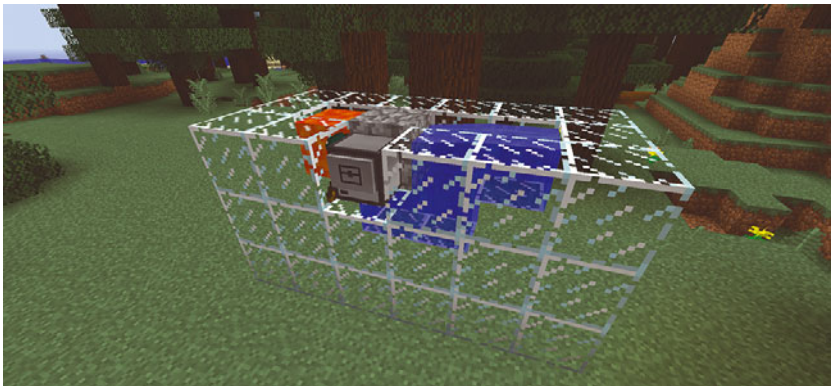


Рис. **9.2.** *Готовый генератор булыжника с черепашкой в пустом пространстве верхнего слоя*

При каждом извлечении готового булыжника, в освободившееся пространство затекает лава, и создается новый блок. Ты можешь сколь угодно долго добывать булыжник в шахте, но при этом изнашиваются кирки. А вот если в нишу верхнего слоя генератора поместить черепашку, повернув ее лицевой частью к центру генератора, черепашка будет добывать булыжник без дополнительных затрат. При этом она даже не использует топливо, так как ей не нужно двигаться!

УСТАНОВКА ПЕЧЕЙ ДЛЯ ПЛАВКИ БУЛЫЖНИКА

Теперь у нас есть неограниченный запас булыжника, но это еще не все. Нам нужно его переплавить для переработки в камень. Для этого мы создадим программу `cobminer`, под управлением которой черепашка будет извлекать булыжник из генератора, а затем помещать добытый булыжник в печи для переплавки. Перед созданием программы `cobminer` нужно выполнить небольшие приготовления.

Во-первых, нужно усовершенствовать генератор, добавив пять печей в средний слой, сразу за черепашкой, как показано на рис. 9.3.

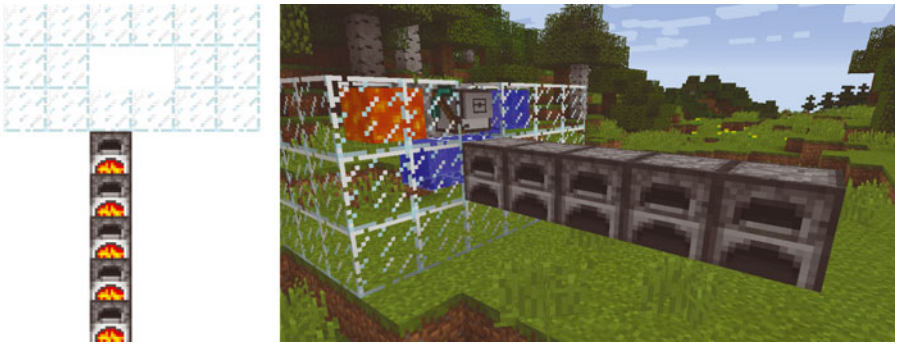


Рис. **9.3.** *Пять печей, добавленных на среднем слое генератора булыжника (слева), и вид генератора с печами в игре*

После запуска созданной программы черепашка будет добывать булыжник до тех пор, пока не наберет полный стек из 64 штук. Затем она станет двигаться вдоль печей, опуская в них булыжник. Печи будут его плавить, превращая в камень. При заполнении всех печей черепашка подождет пять минут, прежде чем начинать новую загрузку булыжника. Весь этот процесс будет повторяться в бесконечном цикле.

В главе 10 мы создадим программу `brickcrafter` для подключения второй черепашки. Под управлением этой программы черепашка будет извлекать выплавленный камень из печей и использовать его для крафта каменных кирпичей. Хранить готовые блоки черепашка будет в ближайшем сундуке.

КОД ПРОГРАММЫ COBMINER

Чтобы написать программу `cobminer`, выполни команду `edit cobminer` в оболочке командной строки и введи следующий код:

cobminer

```
1. --[[Программа генератора булыжника Эла Свейгарта
2. Добыча булыжника с помощью генератора, черепашка 1
   из 2]]
3.
4. os.loadAPI('hare') -- загрузка модуля hare
```

```
 5. local numToDrop
 6. local NUM_FURNACES = 5
 7.
 8. print('Starting mining program...')
 9. while true do
10.   -- добыча булыжника
11.   if turtle.detect() then
12.     print('Cobblestone detected. Mining...')
13.     turtle.dig() -- добыча булыжника
14.   else
15.     print('No cobblestone. Sleeping...')
16.     os.sleep(0.5) -- пауза в полсекунды
17.   end
18.
19.   -- проверка, полон ли стек булыжника
20.   hare.selectItem('minecraft:cobblestone')
21.   if turtle.getItemCount() == 64 then
22.     -- проверка, заправлена ли топливом черепашка
23.     if turtle.getFuelLevel() < (2 * NUM_FURNACES)
        then
24.       error('Turtle needs more fuel!')
25.     end
26.
27.     -- загрузка булыжника в печи
28.     print('Dropping off cobblestone...')
29.     for furnacesToFill = NUM_FURNACES, 1, -1 do
30.       turtle.back() -- перемещение к печи
31.       numToDrop = math.floor(turtle.getItemCount() /
          furnacesToFill)
32.       turtle.dropDown(numToDrop) -- загрузка
          булыжника в печи
33.     end
34.
35.     -- возврат к генератору булыжника
36.     for moves = 1, NUM_FURNACES do
37.       turtle.forward()
38.     end
39.
40.     if turtle.getItemCount() > 0 then
41.       print('All furnaces full. Sleeping...')
42.       os.sleep(300) -- пауза в 5 минут
```

```
43.     end
44.   end
45. end
```

Сохрани программу после ввода всех инструкций. Нажав клавишу **Ctrl**, открой меню. Выбери пункт **[Save]** и нажми клавишу **Enter**. Затем выйди из редактора, нажав клавишу **Ctrl**, выбрав пункт **[Exit]** и нажав клавишу **Enter**.

Кроме того, тебе понадобится модуль `hare`, который можно скачать, выполнив команду `pastebin get wwzvaKuW hare`.

ЗАПУСК ПРОГРАММЫ COBMINER

Создав генератор булыжника с пятью печами, установи черепашку, повернув ее лицевой стороной к блоку булыжника, и запусти программу `cobminer`. Черепашка начнет добывать булыжник, пока не наберет стек в 64 блока, который затем загрузит в печи. Пока ты не доберешься до главы 10 и не напишешь программу `brickcrafter`, загружать топливо в печи и извлекать из них переплавленные каменные блоки придется вручную. Чтобы получить топливо для печей, придется брать блоки древесины у черепашек-лесорубов и пережигать эти блоки в уголь в отдельных печах. Давай подробно рассмотрим каждую часть программы `cobminer`.

Если при ее запуске возникают ошибки, тщательно сравни свой код с листингом в этой книге, может быть, у тебя возникли опечатки. Если опечаток нет, но программа все равно не работает, удали файл, выполнив команду `delete cobminer`, а затем загрузи код заново, выполнив команду `pastebin get YhvSiw7e cobminer`.

НАСТРОЙКА ПРОГРАММЫ И ДОБАВЛЕНИЕ КОНСТАНТ

Первые две строки программы содержат обычный комментарий, который описывает, что это за программа.

cobminer

```
1. --[[Программа генератора булыжника Эла Свейгарта
2. Добыча булыжника с помощью генератора, черепашка 1 из
2]]
3.
4. os.loadAPI('hare') -- загрузка модуля hare
5. local numToDrop
```

Код в строке 4 загружает модуль `hare`, который программа сможет вызвать с помощью функции `hare.selectItem()`. В строке 5 объявляется переменная с именем `numToDrop`, которая будет использоваться позже.

В строке 6 объявляется переменная `NUM _ FURNACES`, которая содержит целое число, соответствующее количеству печей, расположенных за черепашкой.

cobminer

```
6. local NUM_FURNACES = 5
```

В нашем примере печей пять, но ты можешь установить их сколько захочешь. В любом случае, переменной `NUM _ FURNACES` следует присвоить значение, равное количеству установленных печей.

Имя переменной `NUM _ FURNACES` указано прописными буквами, потому что это фиксированная переменная, называемая константой. Это значит, что ее значение не изменяется в процессе исполнения программы. Запись имен констант в верхнем регистре – это просто правило оформления. Константы – сами по себе обычные переменные, а имя прописными буквами просто напоминает, что не следует писать код, который изменяет значение этой переменной. Использование переменной, значение которой никогда не меняется, может показаться странным, но константы делают код более понятным и упрощают внесение изменений в будущем.

Смотри. Допустим, в коде нужно указать количество печей. Но если ты используешь число вместо константы во всем коде, а затем изменишь количество печей, тебе придется обновлять все вхождения, где используется это число. Если же использовать константу типа `NUM _ FURNACES`,

то код не нужно обновлять — достаточно изменить значение в инструкции присваивания (т.е. одну строку кода, где ты присваиваешь константе значение), которая в нашем случае находится в строке 6. Константы делают код понятным и упрощают его изменение.

ДОБЫЧА БУЛЫЖНИКА ИЗ ГЕНЕРАТОРА

Со строки 9 начинается цикл `while`. Это основной цикл программы, в котором черепашка добывает булыжник, перемещается вдоль печей и загружает булыжник в печи.

cobminer

```
 8. print('Starting mining program...')
 9. while true do
10.   -- добыча булыжника
11.   if turtle.detect() then
12.     print('Cobblestone detected. Mining...')
13.     turtle.dig() -- добыча булыжника
```

Первая часть цикла, отвечающая за добычу булыжника, начинается в строке 11. Функция `turtle.detect()` возвращает значение `true`, если блок булыжника находится перед черепашкой. В этом случае программа выводит сообщение `'Cobblestone detected. Mining...'` и, вызывая функцию `turtle.dig()` в строке 13, добывает булыжник.

Но, если булыжника нет, потому что он еще не успел образоваться после добычи блока, функция `turtle.detect()` возвращает значение `false`. Тогда условие инструкции `if` принимает значение `false`, и выполняется блок кода после инструкции `else`, расположенной в строке 14.

cobminer

```
14.   else
15.     print('No cobblestone. Sleeping...')
16.     os.sleep(0.5) -- пауза в полсекунды
17.   end
```

Этот код отображает сообщение `'No cobblestone. Sleeping...'` и вызывает функцию `os.sleep(0.5)` для при-

остановки исполнения программы на полсекунды, чего достаточно для формирования нового блока. Он будет добыт, когда программа возобновит работу. Когда черепашка наберет полный стек булыжника (64 блока), ей будет нужно переплавить его в печах.

ВЗАИМОДЕЙСТВИЕ С ПЕЧАМИ

Интерфейс печи содержит три ячейки: топливную для загрузки топлива, например угля, вводную для загрузки сырья на переплавку и выводную, где результат переплавки сохраняется до тех пор, пока игрок не заберет блоки. Положение черепашки относительно печи определяет возможные действия черепашки, например, загрузку топлива в печь, загрузку сырья для переплавки или сбор готового продукта. Если черепашка находится сбоку от печи, она может загружать или очищать топливную ячейку. Если черепашка находится над печью, она может загружать или очищать вводную ячейку печи. Если черепашка находится ниже печи, она может извлекать результат переплавки. На рис. 9.4 показаны эти положения черепашки.

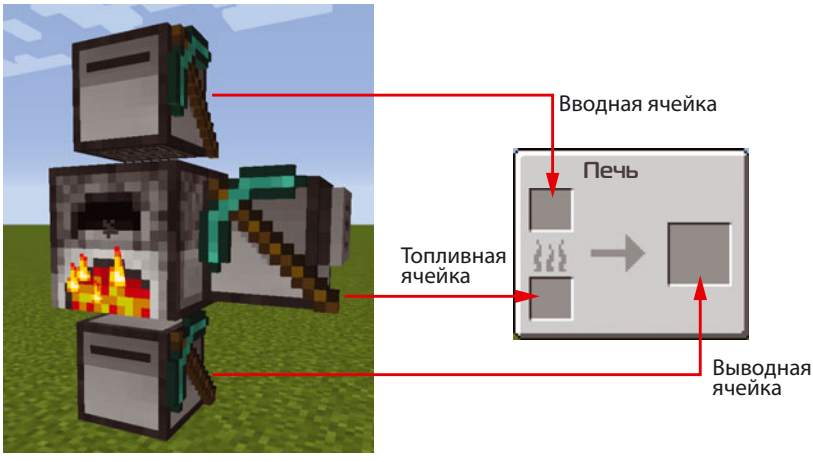


Рис. **9.4.** *Положение черепашки определяет, с какой ячейкой печи она взаимодействует*

Теперь черепашка загрузит блоки булыжника в печи.

УЛУЧШЕНИЕ КОДА С ПОМОЩЬЮ КОНСТАНТ

Убедившись, что существует блок булыжника для добычи, программа проверяет количество уже собранных черепашкой блоков, достигло ли оно максимального значения – 64. Если максимум достигнут, черепашка готовится загрузить булыжник в печи, для чего проверяет, достаточно ли у нее топлива для перемещения к печам и обратно, к генератору булыжника.

cobminer

```
19.   -- проверка, полон ли стек булыжника
20.   hare.selectItem('minecraft:cobblestone')
21.   if turtle.getItemCount() == 64 then
22.     -- проверка, заправлена ли топливом черепашка
23.     if turtle.getFuelLevel() < (2 * NUM_FURNACES)
        then
24.       error('Turtle needs more fuel!')
25.     end
```

Программа с помощью функции `hare.selectItem()` в строке 20 обращается к черепашке. Код в строке 21 вызывает функцию `turtle.getItemCount()`, чтобы проверить количество блоков булыжника, собранных черепашкой. Если блоков булыжника в общей сложности 64, программа вызывает функцию `turtle.getFuelLevel()`, чтобы проверить уровень топлива черепашки.

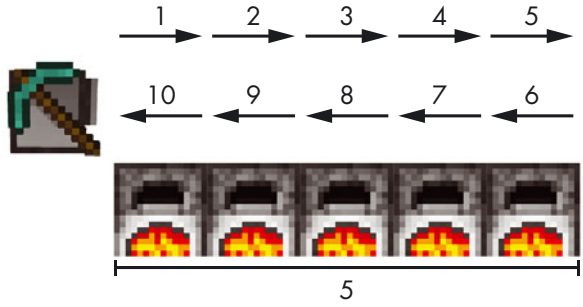


Рис. **9.5.** *Необходимое количество топлива в два раза больше количества печей, так как черепашке необходимо перемещаться не только ко всем печам, но и затем вернуться на исходную позицию*

Код в строке 23 проверяет, чтобы уровень топлива черепашки был не менее чем результат операции `2 * NUM_FURNACES`. Это значение мы берем, потому что черепашке требуется топливо для перемещения к каждой печи и возвращения в исходное положение, как показано на рис. 9.5.

Если у черепашки недостаточно топлива, код в строке 24 вызывает функцию `error()`, выводящую текст `'Turtle needs more fuel!'`. Затем код в строке 25 завершает работу программы.

ЗАГРУЗКА БУЛЫЖНИКА В ПЕЧИ

Если у черепашки накопилось 64 блока булыжника и имеется достаточное количество топлива для перемещения по печам и возвращения на исходную позицию, она может начать движение и загрузку булыжника, как показано на рис. 9.6.

Рис. **9.6.** *Черепашка загружает блоки булыжника в печи*

Цикл `for` в строке 29 немного отличается от циклов `for`, которые мы использовали ранее. Шаг цикла `for` может равен 1, в случае вызова `for i = 1, 10 do`. Ты также можешь использовать шаг другой величины, указав его непосредственно при вызове. Вместо добавления `1` при каждой итерации цикл `for` добавит число, указанное в аргументе шага. Если использовать отрицательное число, например, `for i = 10, 1, -1 do`, счетчик цикла будет работать на уменьшение.

cobminer

```
27.     -- загрузка булыжника в печи
28.     print('Dropping off cobblestone...')
29.     for furnacesToFill = NUM_FURNACES, 1, -1 do
30.       turtle.back() -- перемещение к печи
31.       numToDrop = math.floor(turtle.getItemCount() /
          furnacesToFill)
32.       turtle.dropDown(numToDrop) -- загрузка
          булыжника в печи
33.     end
```

Цикл `for` в строке 29 согласно значению константы `NUM _ FURNACES` (т.е., `5`) определяет количество перемещений черепашки назад. И так как отсчет производится от `5` до `1`, а не от `1` до `5`, мы можем использовать переменную `furnacesToFill` цикла `for` в строке 31, чтобы рассчитать, сколько блоков булыжника нужно загрузить в каждую печь. В этом вычислении используется функция `math.floor()`. Давай посмотрим, как работает эта функция.

ОКРУГЛЕНИЕ ЧИСЕЛ С ПОМОЩЬЮ ФУНКЦИЙ MATH.FLOOR() И MATH.CEIL()

Функция `math.floor()` округляет в меньшую сторону переданное ей число и возвращает округленное, а функция `math.ceil()` округляет переданное число в бóльшую сторону и возвращает его. Чтобы посмотреть, как эти функции работают, в оболочке Lua введи следующие инструкции:

```
lua> math.floor(4.2)
4
lua> math.floor(4.9)
4
lua> math.floor(10.5)
10
lua> math.floor(12.0)
12
lua> math.ceil(4.2)
5
```

```
lua> math.ceil(4.9)
5
lua> math.ceil(10.5)
11
❶ lua> math.ceil(12.0)
12
```

Передача дробного значения функции `math.floor()` приводит к отображению целой части числа, без десятичного разделителя, тогда как передача такого же значения функции `math.ceil()` округляет число до следующего целого числа. Если передать функции `math.ceil()` число со значением `0`, оно, как ты можешь видеть, округляется до ближайшего целого числа ❶. Функции округления помогут нам распределять булыжник в печи равномерно.

Вычисление количества булыжника для загрузки в каждую печь

В игре Minecraft вводная ячейка в интерфейсе печи может содержать не более 64 блоков для переплавки. Для достижения максимальной эффективности нашего заводика мы хотим, чтобы все печи переплавляли булыжник одновременно. Чтобы рассчитать, сколько блоков булыжника нужно загрузить в каждую печь, мы разделим количество блоков булыжника в текущей ячейке на значение константы `NUM _ FURNACES`. Поскольку в нашем случае (печей пять) эта операция деления не может привести к целому числу, т.е. , мы передадим это число функции `math.floor()`, которая округлит значение `12,8` до `12`. Затем, в каждую печь мы можем загрузить именно 12 блоков булыжника, чтобы все печи смогли переплавлять блоки одновременно.

Но тут возникает несколько проблем. Например, если у нас есть 64 блока булыжника и 5 печей, черепашка загрузит по 12 блоков в каждую печь, и 4 блока останется. Черепашка может добывать булыжник быстрее, чем печи могут его плавить, а каждая печь может содержать не более 64 блоков в вводной ячейке. Для каждой заполненной печи, которая не может больше принимать булыжник, черепаш-

ка будет сохранять часть своих блоков, – в нашем случае 12. Чтобы решить эту проблему, мы выполним другой расчет. Давай рассмотрим строки 29–33:

cobminer

```
29.     for furnacesToFill = NUM_FURNACES, 1, -1 do
30.       turtle.back() -- перемещение к печи
31.       numToDrop = math.floor(turtle.getItemCount() /
          furnacesToFill)
32.       turtle.dropDown(numToDrop) -- загрузка
          булыжника в печи
33.     end
```

С помощью переменной `numToDrop` в строке 31 вычисляется количество блоков булыжника для каждой печи – `numToDrop = math.floor(turtle.getItemCount() / furnacesToFill)`. Но вместо того, чтобы вычислить один раз количество булыжника для загрузки в одну печь и сохранить это число в переменной `numToDrop`, ее значение пересчитывается каждый раз, когда черепашка переходит к следующей печи.

В табл. 9.1 показано, как значение переменной `numToDrop` вычисляется при каждой итерации цикла `for`, когда все печи пусты.

Табл. 9.1. Значения переменной `numToDrop`, если все печи пусты

Итерация	**`math.floor(turtle.getItemCount()/furnacesToFill)`**	**Значение переменной `numToDrop`**	**Количество блоков для загрузки в одну печь**
Первая	`math.floor(64/5)`	12	12
Вторая	`math.floor(52/4)`	13	13
Третья	`math.floor(39/3)`	13	13
Четвертая	`math.floor(26/2)`	13	13
Пятая	`math.floor(13/1)`	13	13
			Всего: 64

Теперь притворимся, что вторая печь заполнена, потому что игрок сбросил в нее часть добытого булыжника. Таким образом, во вторую печь загружать булыжник нельзя. Но поскольку значение переменной `numToDrop` пересчитывается при каждой итерации цикла `for`, программа автоматически увеличит количество блоков булыжника, предназначенных для остальных печей. В табл. 9.2 показано, как в этом случае значение переменной `numToDrop` рассчитывается при каждой итерации. Обрати внимание, что на второй итерации количество попадающих в печь блоков равно 0, потому что вторая печь заполнена.

Табл. 9.2. Значения переменной `numToDrop`, если вторая печь заполнена

Итерация	`math.floor(turtle.getItemCount()/furnacesToFill)`	Значение переменной `numToDrop`	Количество блоков для загрузки в одну печь
Первая	`math.floor(64/5)`	12	12
Вторая	`math.floor(52/4)`	13	0
Третья	`math.floor(52/3)`	17	17
Четвертая	`math.floor(35/2)`	17	17
Пятая	`math.floor(18/1)`	18	18
			Всего: 64

Строки кода 29–33 показывают, что если немного продумать код, то печи можно заставить работать с максимальной эффективностью. Когда цикл `for` завершится, черепашка будет находиться рядом с последней печью и ей нужно будет вернуться на исходную позицию.

ВОЗВРАЩЕНИЕ ЧЕРЕПАШКИ НА ИСХОДНУЮ ПОЗИЦИЮ

Строки кода 36–38 перемещают черепашку назад, пока она не окажется перед блоком булыжника.

cobminer

```
35.      -- возврат к генератору булыжника
36.      for moves = 1, NUM_FURNACES do
37.        turtle.forward()
38.      end
```

На этом этапе проверяется наличие блоков булыжника у черепашки. Напомню, что черепашка может добывать булыжник быстрее, чем печи могут его переплавлять. Очень скоро все печи будут полностью заполнены, а у черепашки будет 64 блока булыжника. Если у черепашки в инвентаре остается булыжник, а все печи полны, то черепашка не может загрузить блоки в печи.

cobminer

```
40.      if turtle.getItemCount() > 0 then
41.        print('All furnaces full. Sleeping...')
42.        os.sleep(300) -- пауза в 5 минут
43.      end
44.    end
45. end
```

Код в строке 43 завершает блок инструкции `if`, начинающийся в строке 40, код в строке 44 заканчивает блок инструкции `if`, который начинается в строке 21, а код в строке 45 завершает цикл `while`, начинающийся в строке 9. Наконец, исполнение программы возвращается к строке 9, и черепашка продолжает добывать булыжник и заполнять печи, пока не закончится ее топливо.

ДОПОЛНИТЕЛЬНОЕ ЗАДАНИЕ: ДОБЫЧА КАМНЯ С БОЛЬШИМ КОЛИЧЕСТВОМ ПЕЧЕЙ

Попробуй создать генератор булыжника с более чем пятью печами. Тебе нужно будет изменить значение константы `NUM_FURNACES` в программе, чтобы черепашка использовала дополнительные печи.

Как и в программе лесофермы из главы 8, ты можешь расширить свое производство, построив несколько генераторов булыжника. Ты также можешь добавить больше

печей, не забыв изменить значение константы NUM_FURNACES. (Пять или шесть печей вполне достаточно, чтобы переплавлять булыжник из одного генератора. Иначе черепашка не сможет добывать булыжник достаточно быстро, и печи будут простаивать!)

ЧТО МЫ УЗНАЛИ

В этой главе ты узнал, как построить генератор булыжника, в котором смешиваются потоки лавы и воды для бесконечного производства блоков булыжника. Ты использовал программу cobminer, чтобы черепашка собирала блоки булыжника и загружала их в печи для переплавки. Ты узнал о константах – переменных, которые не изменяют своих значений, делая код более удобным для чтения. Кроме того, мы повторили, что шаг аргумента цикла for можно указать явно и цикл for может отсчитываться вниз, а не вверх. Наконец, ты узнал, как функции math.floor() и math.ceil() могут округлять число в меньшую или большую сторону соответственно.

В главе 10 мы применим программу cobminer для строительства заводика каменного кирпича, которая будет оперировать уже двумя черепашками. Мы напишем программу, которая поручит другой черепашке вытаскивать переплавленные каменные блоки из печей и перерабатывать их в каменный кирпич с помощью программы brickcrafter.

10

ПРОИЗВОДСТВО КАМЕННЫХ КИРПИЧЕЙ

Теперь у тебя есть черепашка, которая добывает булыжник из генератора и загружает булыжник в печи для переплавки. Процесс переплавки автоматизирован, но доставать каменные блоки из печей для их дальнейшей переработки приходится вручную. Все это хорошо, но не дотягивает до полноценного заводика по производству каменного кирпича. Нужно запрограммировать вторую черепашку, чтобы она вытаскивала переплавленный камень и перерабатывала его в каменные кирпичи. Как только будет подготовлена команда черепашек, как это показано на рис. 10.1, можно будет в автоматическом режиме получать кирпичи, пригодные для строительства.

Рис. **10.1.** *Полностью автоматизированный завод по производству каменного кирпича*

РАЗРАБОТКА ПРОГРАММЫ ПРОИЗВОДСТВА КАМЕННОГО КИРПИЧА

Первая черепашка, работающая по программе `cobminer`, расположена над печами. Вторая черепашка будет двигаться ниже печей, чтобы она могла вытаскивать блоки из выводных ячеек печей. Напомню, что из камня или булыжника добываются блоки булыжника, которые затем загружаются в печи для переплавки. Полученные в результате переплавки блоки камня перерабатываются в каменный кирпич.

Как и `cobminer`, программа второй черепашки сначала должна проверить, что у черепашки достаточно топлива для перемещения вдоль печей и возвращения на исходную позицию. Двигаясь вдоль печей, черепашка собирает блоки камня, которые были выплавлены из булыжника. Собрав стек из 64 блоков, она создаст из них 64 каменных кирпича и поместит их в сундук. Если в выводной ячейке печей скопилось недостаточно камня, черепашка подождет две минуты, чтобы количество переплавленного булыжника увеличилось. Весь процесс черепашка повторяет циклически.

СОЗДАНИЕ КРАФТ-ЧЕРЕПАШКИ

Для начала черепашку надо научить изготавливать каменные кирпичи из камня. Черепашки могут создавать различные блоки и предметы с помощью функции `turtle.craft()`. Однако, чтобы задействовать эту функцию, ты должен оснастить черепашку верстаком, по аналогии оснащения черепашки алмазной киркой. Выбери черепашку, только не ту, которая уже работает с программой `cobminer`, а еще одну, и помести верстак в текущую ячейку черепашки. Затем в оболочке Lua выполни следующий код:

```
lua> turtle.equipLeft()
```

Этот код уберет имеющийся на левой стороне черепашки инструмент и оснастит ее верстаком. Рис. 10.2 демонстрирует черепашку и ее инвентарь до и после запуска функции `turtle.equipLeft()`.

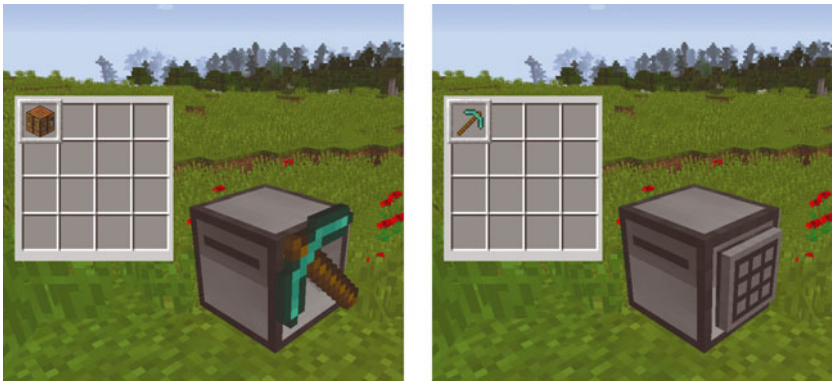


Рис. **10.2.** *Функция* `turtle.equipLeft()` *заменяет кирку на верстак. Черепашка перед выполнением кода (слева) и после (справа)*

Теперь твоя черепашка может вызвать функцию `turtle.craft()`. Она больше не может обращаться к функции `turtle.dig()`, но нам это и не нужно.

Примечание *Если ты хочешь, чтобы черепашка могла и крафтить и добывать одновременно, ты можешь оснастить черепашку двумя инструментами. Для этого*

понадобятся две функции: `turtle.equipLeft()` *для размещения одного инструмента с левой стороны черепашки и* `turtle.equipRight()`*, чтобы экипировать ее с другой стороны. На рис. 10.3 показана черепашка с алмазной киркой слева и верстаком справа. Такая черепашка может вызывать как функцию* `turtle.dig()`*, так и* `turtle.craft()`*.*

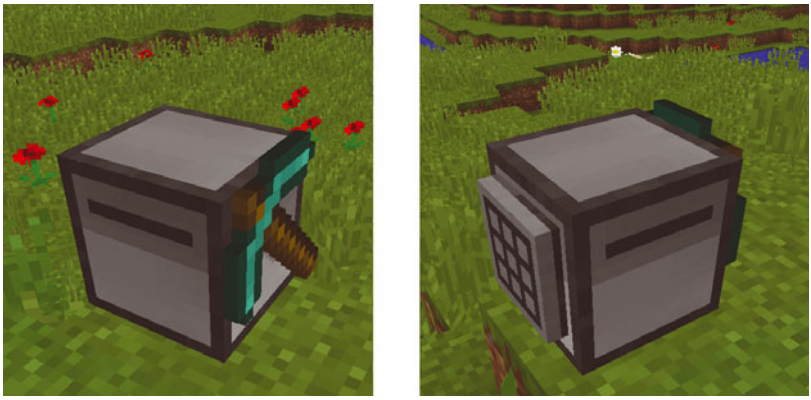


Рис. **10.3.** *Черепашка с алмазной киркой на левой стороне (слева) и верстаком на правой (справа)*

Черепашка, оборудованная верстаком, может крафтить только в соответствии с рецептом, компоненты которого собраны в ее инвентаре. При вызове функции `turtle.craft()` созданный предмет или блок помещается в текущую ячейку инвентаря.

Например, чтобы скрафтить каменный кирпич, поместить четыре каменных блока в инвентарь черепашки, как показано на рис. 10.4. Убедись, что все остальные ячейки инвентаря пусты, и в оболочке Lua выполни команды `turtle.select(16)` и `turtle.craft()`:

```
lua> turtle.select(16)
true
lua> turtle.craft()
true
```

На рис. 10.4 показан графический интерфейс черепашки до и после крафта каменных кирпичей. Обрати внимание,

что на этом рисунке в качестве текущей настроена ячейка под номером 16, и созданный блок помещается туда.

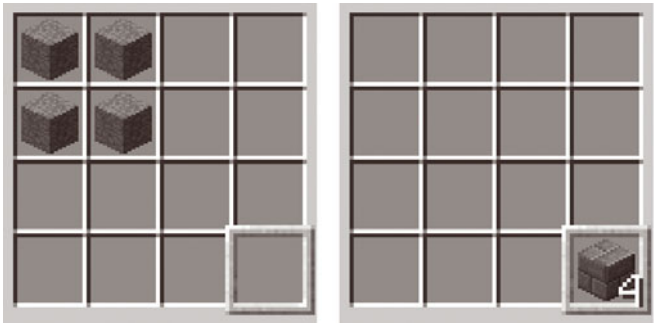


Рис. **10.4.** *Инвентарь черепашки до (слева) и после (справа) вызова функции* `turtle.craft()`

Если блоки в инвентаре черепашки не будут соответствовать рецепту Minecraft, функция `turtle.craft()` вернет значение `false`, и ты увидишь сообщение об ошибке:

```
lua> turtle.craft()
false
No matching recipes
```

Теперь, когда настройка и проектирование программы второй черепашки завершены, приступим к написанию кода. Новую программу назовем `brickcrafter`.

КОД ПРОГРАММЫ BRICKCRAFTER

В оболочке командной строки выполни команду `edit brickcrafter` и введи следующий код:

brickcrafter

```
1. --[[Программа генератора булыжника Эла Свейгарта
2. Крафт каменных кирпичей из камня в печах, черепашка
   2 из 2]]
3.
4. print('Starting stone brick crafting program...')
5.
6. local NUM_FURNACES = 5
```

```
local brickCount = 0
while true do
  -- проверка, заправлена ли топливом черепашка
  if turtle.getFuelLevel() < (2 * NUM_FURNACES) then
    error('Turtle needs more fuel!')
  end

  turtle.select(1) -- помещение камня в ячейку 1

  -- сбор камня из печей
  for i = 1, NUM_FURNACES do
    turtle.suckUp(64 - turtle.getItemCount(1)) --
get stone from furnace
    if turtle.getItemCount(1) == 64 then
      break -- остановка, если собрано 64 блока камня
    end
    if i ~= NUM_FURNACES then
      turtle.back() -- перемещение к следующей печи
    end
  end

  -- обработка блоков камня
  if turtle.getItemCount(1) == 64 then
    turtle.transferTo(2, 16) -- помещение камня в
    ячейку 2
    turtle.transferTo(5, 16) -- помещение камня в
    ячейку 5
    turtle.transferTo(6, 16) -- помещение камня в
    ячейку 6
    turtle.select(16) -- помещение каменных кирпичей
    в ячейку 16
    turtle.craft() -- крафт каменных кирпичей
    brickCount = brickCount + 64
    print('Total stone bricks: ' .. brickCount)
  else
    print('Not enough stone yet. Sleeping...')
    os.sleep(120) -- ожидание 2 минуты
  end

  -- перемещение к сундуку (около первой печи)
  for i = 1, NUM_FURNACES - 1 do
    turtle.forward()
```

```
44.   end
45.   turtle.turnLeft() -- поворот лицевой стороной к
      сундуку
46.   turtle.select(16) -- выбор каменных кирпичей
47.   turtle.drop() -- помещение каменных кирпичей в
      сундук
48.   turtle.turnRight() -- поворот к генератору
49. end
```

Сохрани программу после ввода всех инструкций. Нажав клавишу **Ctrl**, открой меню. Выбери пункт **[Save]** и нажми клавишу **Enter**. Затем выйди из редактора, нажав клавишу **Ctrl**, выбрав пункт **[Exit]** и нажав клавишу **Enter**.

ЗАПУСК ПРОГРАММЫ BRICKCRAFTER

Чтобы начать работу с программой `brickcrafter`, понадобится черепашка, управляемая программой `cobminer`, и генератор булыжника с печами из главы 9. Понадобится также сундук (обычный или большой) и вторая черепашка, которой будет управлять программа `brickcrafter`. Установи сундук рядом с генератором, а вторую черепашку помести под первую печь, как показано на рис. 10.5. В сундук будут складываться все созданные каменные кирпичи. Удостоверься, что инвентарь черепашки пуст, иначе программа `brickcrafter` не будет работать корректно.

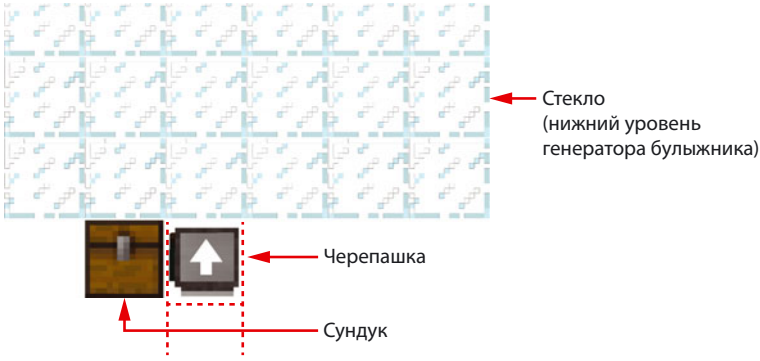


Рис. **10.5.** *Установка сундука и второй черепашки. Белая стрелка показывает, в каком направлении должна быть ориентирована черепашка. Пунктирные линии показывают, где расположены печи среднего слоя генератора булыжника*

На рис. 10.6 изображен генератор булыжника, полностью настроенный на производство каменного кирпича.

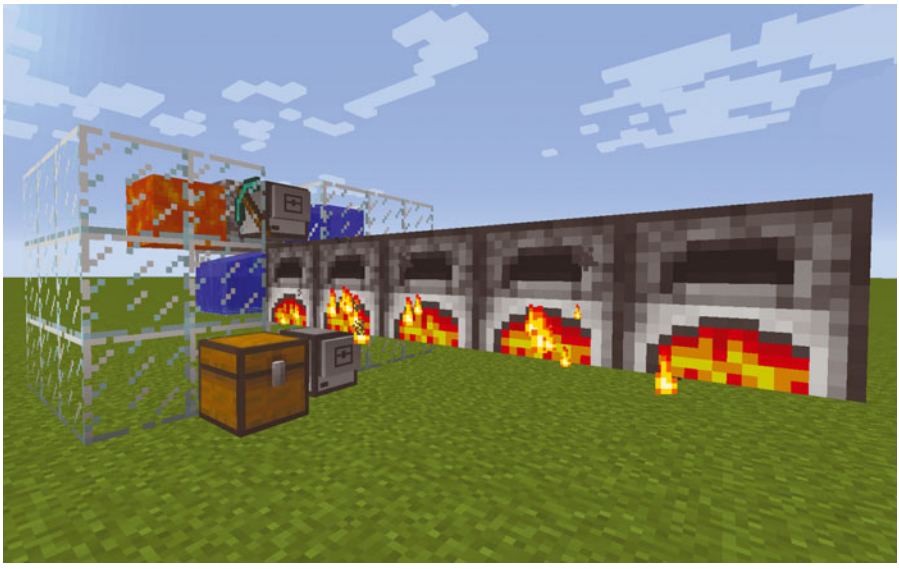


Рис 10.6. Рядом с генератором булыжника поставь сундук для хранения каменных кирпичей

Под управлением программы `brickcrafter` черепашка перемещается под печами и вынимает переплавленный камень из всех печей. Затем она обрабатывает каменные блоки в соответствии с рецептом крафта каменного кирпича, вызывая функцию `turtle.craft()`, а готовые кирпичи складывает в сундук, расположенный рядом с генератором. Поскольку черепашка должна двигаться вперед и назад, программа проверяет уровень ее топлива и останавливает работу, если топлива недостаточно.

Если при запуске программы возникают ошибки, тщательно сравни свой код с листингом в этой книге, возможно, были возможные опечатки. Если опечаток нет, но ошибки возникают, удали файл, выполнив команду `delete brickcrafter`, а затем загрузи код с помощью команды `pastebin get 1zS07K3U brickcrafter`.

НАСТРОЙКА ПРОГРАММЫ BRICKCRAFTER

Первые несколько строк программы содержат обычные, описывающие код, комментарии и вызов `print()`, который выводит игроку сообщение о том, что программа запущена. Затем объявляются две переменные, которые мы в дальнейшем будем использовать в программе.

brickcrafter

```
 1. --[[Программа генератора булыжника Эла Свейгарта
 2. Крафт каменных кирпичей из камня в печах, черепашка
    2 из 2]]
 3.
 4. print('Starting stone brick crafting program...')
 5.
 6. local NUM_FURNACES = 5
 7. local brickCount = 0
```

Константа `NUM _ FURNACES` в этой программе выполняет ту же задачу, что и в программе `cobminer` из главы 9. Переменная `brickCount` отслеживает количество созданных черепашкой каменных кирпичей.

ПРОВЕРКА «СЫТОСТИ» ЧЕРЕПАШКИ

Со строки 8 начинается бесконечный цикл основной части программы. Он продолжается до тех пор, пока у черепашки есть топливо или пока игрок не нажмет сочетание клавиш **Ctrl+T**, чтобы завершить программу.

brickcrafter

```
 8. while true do
 9.   -- проверка, заправлена ли топливом черепашка
10.   if turtle.getFuelLevel() < (2 * NUM_FURNACES) then
11.     error('Turtle needs more fuel!')
12.   end
```

В теле этого цикла программа сначала проверяет, что у черепашки достаточно топлива, чтобы добраться до каждой печи и затем вернуться к сундуку. Строки кода 10–12

аналогичны строкам 23–25 в программе `cobminer` из главы 9. Код строки 10 проверяет, что количество топлива у черепашки как минимум вдвое больше числа печей. Если это не так, вызов функции `error()` в строке 11 завершит работу программы.

СБОР КАМНЯ ИЗ ПЕЧЕЙ

Если у черепашки достаточно топлива, чтобы пройти подо всеми печами и вернуться, она выбирает ячейку 1 инвентаря для хранения собранных блоков, так что, когда она будет вынимать камень из печи, он будет размещен именно в этой ячейке.

brickcrafter

```
14.   turtle.select(1) -- помещение камня в ячейку 1
```

Затем исполнение переходит к циклу `for` в строке 17. На каждой итерации цикла черепашка вытаскивает переплавленный камень из печи, под которой находится, и помещает его в ячейку 1. Когда черепашка полностью заполнит ячейку (64 блока), цикл завершится, чтобы можно было крафтить каменный кирпич из собранных блоков. А пока ячейка инвентаря не заполнена, исполнение цикла продолжается, и черепашка переходит к следующей печи, и так далее.

brickcrafter

```
16.   -- сбор камня из печей
17.   for i = 1, NUM_FURNACES do
18.     turtle.suckUp(64 - turtle.getItemCount(1)) --
        get stone from furnace
19.     if turtle.getItemCount(1) == 64 then
20.       break -- остановка, если собрано 64 блока
          камня
21.     end
22.     if i ~= NUM_FURNACES then
23.       turtle.back() -- перемещение к следующей печи
24.     end
25.   end
```

Функция `turtle.suckUp()` вынимает каменные блоки из печи над черепашкой и сохраняет их в выбранной ячейке. Можно передать этой функции число и ограничить количество блоков, помещаемых в выбранную ячейку черепашки. Также можно воспользоваться функциями `turtle.suck()` и `turtle.suckDown()`, чтобы принимать блоки из печей, которые находятся перед или под черепашкой соответственно.

В нашем случае, поскольку черепашка находится под печами, она вытаскивает предметы из выводной ячейки печи. Черепашка может хранить не более 64 каменных блоков в ячейке 1 инвентаря, поэтому, чтобы исключить переполнение, мы выполняем некоторые математические вычисления. Функция `turtle.getItemCount(1)` возвращает количество блоков, которые уже есть в ячейке инвентаря. Полученное число мы вычитаем из 64, чтобы получить количество блоков, которые черепашка еще должна достать из печей. Например, если в ячейке инвентаря 1 уже есть 30 каменных блоков, а в печи находится 64 каменных блока, черепашка не должна вынимать все 64 блока. Вместо этого она должна вынуть лишь 64 - 30 = 34 блока.

Если в ячейке 1 инвентаря находится 64 блока, добавление блоков прекращается. Инструкция `if` в строке 19 возвращает значение `true`, и выполнение цикла `for` прекращается.

Если ячейка инвентаря еще не заполнена, черепашка продолжает движение и переходит к следующей печи. Так продолжается до тех пор, пока черепашка не достигнет последней печи. Это произойдет, когда `i` – переменная цикла `for` – станет равна значению константы `NUM _ FURNACES`. Таким образом, исполнение цикла продолжается, пока выполняется условие инструкции `if` в строке 22, если значение переменной `i` не равно значению константы `NUM _ FURNACES`. Если условие не выполняется, исполнение цикла прекращается.

ИЗГОТОВЛЕНИЕ КИРПИЧЕЙ

Когда черепашка соберет 64 каменных блока, их нужно переработать в кирпичи. Напомню, что черепашка производит каменные кирпичи, применяя рецепт, как ты это делаешь на верстаке, но крафтит их в собственном инвентаре. Для переработки каменных блоков в каменные кирпичи нужно поместить одинаковое количество каменных блоков в четыре ячейки инвентаря, чтобы сформировать квадрат, как показано на рис. 10.7.

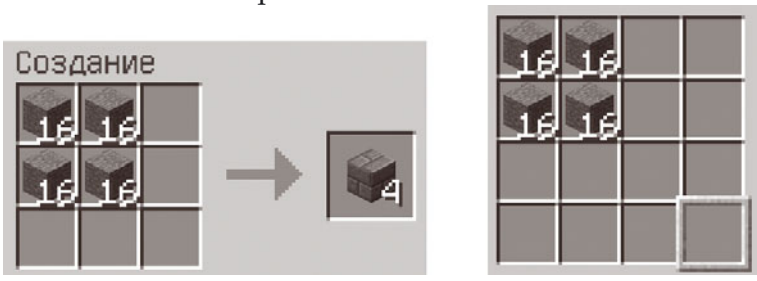


Рис. **10.7.** *Рецепт крафта каменных кирпичей в игре (слева) и инвентаре черепашки (справа)*

Когда блоки собраны и размещены в соответствии с рецептом крафта, вызов функции `turtle.craft()` приводит к созданию каменных кирпичей и сохранению их в текущей ячейке. Если ты хочешь, чтобы черепашка создавала только определенное количество кирпичей, даже если у нее достаточно блоков для большего количества, передай в функцию `turtle.craft()` целочисленный аргумент. Например, функция `turtle.craft(1)` будет добавлять только один каменный блок в каждую из четырех ячеек и создавать один каменный кирпич.

Код в строке 28 проверяет наличие 64 каменных блоков в ячейке 1 инвентаря. Если это так, исполнение переходит к строке 29.

brickcrafter

```
27.   -- обработка блоков камня
28.   if turtle.getItemCount(1) == 64 then
29.     turtle.transferTo(2, 16) -- помещение камня в
```

```
ячейку 2
30.     turtle.transferTo(5, 16) -- помещение камня в
        ячейку 5
31.     turtle.transferTo(6, 16) -- помещение камня в
        ячейку 6
32.     turtle.select(16) -- помещение каменных кирпичей
        в ячейку 16
33.     turtle.craft() -- крафт каменных кирпичей
34.     brickCount = brickCount + 64
35.     print('Total stone bricks: ' .. brickCount)
```

Функция `turtle.transferTo()` перемещает блоки из текущей ячейки в другую. Ты можешь указать, в какую именно, передав целое число в качестве первого параметра функции. Второй параметр определяет количество перемещаемых блоков. Вызов функции `turtle.transferTo(2, 16)` в строке 29 перемещает 16 каменных блоков из текущей ячейки (под номером 1, как указано в строке 14) в ячейку 2. Код в строках 30 и 31 перемещает 16 каменных блоков в ячейки 5 и 6. Теперь расположение блоков в инвентаре черепашки соответствует рецепту, показанному на рис. 10.7. Затем код в строке 32 вызывает функцию `turtle.select(16)` для выбора ячейки 16 в качестве текущей, поэтому, когда в строке 33 вызывается функция `turtle.craft()`, обработанные каменные кирпичи будут помещены в ячейку 16.

В строках 34 и 35 производится подсчет и вывод на экран количества изготовленных черепашкой кирпичей.

Если черепашка, после прохождения всех печей не собрала 64 каменных блока в ячейку 1, код в строке 38 приостанавливает исполнение программы на 120 секунд (две минуты), чтобы печи переплавили дополнительные каменные блоки.

brickcrafter

```
36.   else
37.     print('Not enough stone yet. Sleeping...')
38.     os.sleep(120) -- ожидание 2 минуты
39.   end
```

После возобновления работы программы черепашка вернется к первой печи. Независимо от того, были ли созданы каменные кирпичи, черепашке нужно вернуться к первой, ближайшей к генератору булыжника, печи, чтобы наш робот мог собрать как можно больше переплавленных камней.

ВОЗВРАЩЕНИЕ ЧЕРЕПАШКИ НА ИСХОДНУЮ ПОЗИЦИЮ

Чтобы переместить черепашку в исходное положение, нам нужно двигать ее в сторону первой печи, пока наш робот не окажется прямо под ней. Там черепашка сложит в сундук свои кирпичи. За эти движения отвечает цикл `for`, начинающийся в строке 42.

brickcrafter

```
41.   -- перемещение к сундуку (около первой печи)
42.   for i = 1, NUM_FURNACES - 1 do
43.     turtle.forward()
44.   end
45.   turtle.turnLeft() -- поворот лицевой стороной к
      сундуку
46.   turtle.select(16) -- выбор каменных кирпичей
47.   turtle.drop() -- помещение каменных кирпичей в
      сундук
48.   turtle.turnRight() -- поворот к генератору
49. end
```

Код в строках 42–44 управляет перемещением черепашки вперед таким образом, чтобы она вернулась в исходное положение. То есть черепашка совершает `NUM_FURNACES - 1` последовательных перемещений, пока не окажется ниже первой печи. В строке 6 константе `NUM_FURNACES` было присвоено значение `5`, поэтому черепашка совершит четыре последовательных перемещения. Код в строке 45 поворачивает черепашку лицевой стороной к сундуку, а код в строке 46 выбирает все имеющиеся каменные кирпичи, которые были созданы и сохранены в ячейке 16

инвентаря. Затем код в строке 47 помещает выбранные каменные кирпичи в сундук (или ничего не делает, если кирпичи не были изготовлены), а код в строке 48 поворачивает черепашку лицевой стороной к генератору булыжника.

Финальная инструкция `end` в строке 49 завершает бесконечный цикл `while`, начавшийся в строке 8, поэтому, когда программа достигает этой точки, исполнение снова переходит к строке 8. Таким образом, весь процесс сбора камня, переработка его в каменные кирпичи и сохранение готовых кирпичей в сундуке повторяются снова и снова.

ДОПОЛНИТЕЛЬНОЕ ЗАДАНИЕ: КОНДИТЕРСКАЯ ФАБРИКА

Для приготовления торта в Minecraft необходимо следовать одному из самых сложных рецептов в игре, который требует молоко, яйцо, сахар и пшеницу, как показано на рис. 10.8.

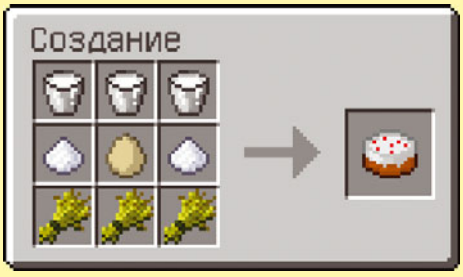


Рис. **10.8.** *Рецепт крафта торта в игре Minecraft*

Создай программу, которая научит черепашку печь торты. Размести ингредиенты в отдельных сундуках, а затем запрограммируй черепашку так, чтобы она двигалась вдоль ряда сундуков, выбирала ингредиенты для торта из каждого сундука, делала торт и помещала готовый торт в отдельный сундук. Если какой-нибудь сундук с ингредиентом, окажется пуст, пусть черепашка ожидает пару минут, прежде чем снова проверить, не появился ли в сундуке нужный ингредиент.

СТРОИТЕЛЬСТВО ЗДАНИЯ ЗАВОДИКА

Наш генератор булыжника и черепашки, то есть основная производственная линия, до сих пор остаются на улице, где они подвергаются воздействию дождя и ветра. Это сложно назвать заводом! Теперь, когда у тебя есть неограниченный источник булыжника и каменных кирпичей, можно построить заводское здание, как показано на рис. 10.9 и 10.10.

Рис. **10.9.** *Здание для размещения генератора булыжника и черепашек, построенное из собственной продукции*

Рис. **10.10.** *Интерьер с двумя генераторами булыжника и декоративным водопадом лавы*

Довольно скоро твои черепашки произведут множество каменных кирпичей, как показано на рис. 10.11.

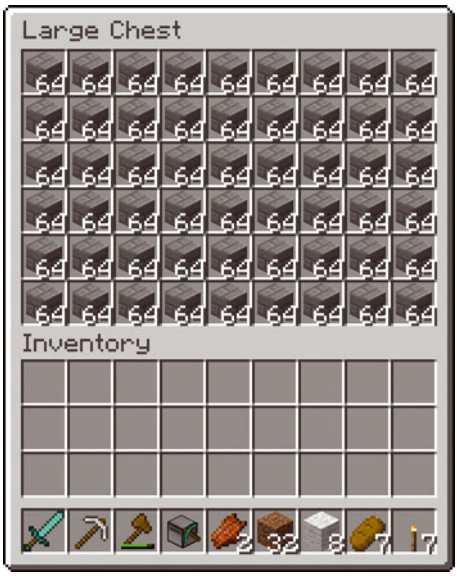


Рис. **10.11.** *Проблем с кирпичом больше нет благодаря возможностям программирования!*

Строительство здания для генератора булыжника необязательно, но, по крайней мере, тебе не придется добывать каменные блоки вручную, если ты решишься на его постройку. В главах 11 и 12 ты узнаешь, как запрограммировать черепашек, чтобы они возводили стены и строили комнаты!

ЧТО МЫ УЗНАЛИ

В этой главе ты узнал, как сменить инструмент черепашки на верстак, используя функции `turtle.equipLeft()` и `turtle.equipRight()`. Оснащение черепашки верстаком позволяет ей крафтить при помощи функции `turtle.craft()`.

Чтобы вынимать какие-либо предметы или блоки из контейнеров, таких как сундуки или печи, следует использовать функцию `turtle.suck()`. Этой функции мож-

но передать числовой аргумент, чтобы ограничить количество изымаемых предметов/блоков. Функции `turtle.suckUp()` и `turtle.suckDown()` позволяют черепашке взаимодействовать с контейнерами выше или ниже нее соответственно.

Перед вызовом функции `turtle.craft()` нужно расположить компоненты в инвентаре черепашки в соответствии с рецептом, так же, как ты это делал бы на верстаке. Для размещения ингредиентов рецепта в разных ячейках можно использовать функцию `turtle.transferTo()`. После вызова функции `turtle.craft()` создаваемые предметы появятся в текущей ячейке черепашки.

В следующей главе, заимев каменные кирпичи, давай начнем программировать черепашек так, чтобы они начали строить для нас гигантские здания!

11

ВОЗВЕДЕНИЕ СТЕН

Строительство из блоков может оказаться опасным и долгим процессом. Опасным, потому что можно упасть с большой высоты, если строить высокие здания в режиме выживания.

Вместо того, чтобы строить стены самостоятельно, можно запрограммировать черепашку и поручить ей эти опасные и нудные строительные работы. В этой главе мы разработаем алгоритм для строительства стен любого размера, добавив несколько новых функций в модуль `hare`. Хотя мы и сосредоточимся на строительстве стен из каменных кирпичей, эти функции пригодны для работы с любыми блоками, например землей, стеклом или даже арбузами, как показано на рис. 11.1.

Начнем с добавления двух функций в модуль `hare`: одна будет подсчитывать количество блоков в инвентаре черепашки, а другая – возводить стены.

Рис. **11.1.** *Черепашки строят стены из блоков земли, каменных кирпичей, стекла и арбузов*

ДОРАБОТКА МОДУЛЯ HARE

Мы добавим три новые функции, которые будут считать блоки в инвентаре, выбирать и размещать блоки и возводить стены. Мы будем использовать эти функции и в других программах из этой книги, поэтому поместим их в модуль `hare`, который мы создали в главе 7. Первые 35 строк модуля `hare` останутся без изменений. Поэтому я покажу код только для строк 36 – 107.

В оболочке командной строки выполни команду `edit hare`. Перемести курсор в нижнюю часть файла и продолжи код, вводя следующее:

hare

```
    ...пропуск...
36.
37.
38. -- countInventory() возвращает количество
39. -- всех блоков в инвентаре черепашки
40. function countInventory()
```

```
41.   local total = 0
42.
43.    for slot = 1, 16 do
44.      total = total + turtle.getItemCount(slot)
45.    end
46.   return total
47. end
48.
49.
50. -- selectAndPlaceDown() выбирает непустую ячейку
51. -- и помещает блок из нее под черепашку
52. function selectAndPlaceDown()
53.
54.   for slot = 1, 16 do
55.     if turtle.getItemCount(slot) > 0 then
56.       turtle.select(slot)
57.       turtle.placeDown()
58.       return
59.     end
60.   end
61. end
62.
63.
64. -- buildWall() создает стену,
65. -- расположенную перед черепашкой
66. function buildWall(length, height)
67.   if hare.countInventory() < length * height then
68.     return false -- недостаточно блоков
69.   end
70.
71.   turtle.up()
72.
73.   local movingForward = true
74.
75.   for currentHeight = 1, height do
76.     for currentLength = 1, length do
77.       selectAndPlaceDown() -- размещение блока
78.       if movingForward and currentLength ~= length
          then
79.         turtle.forward()
80.       elseif not movingForward and currentLength ~=
          length then
```

```
81.           turtle.back()
82.         end
83.       end
84.       if currentHeight ~= height then
85.         turtle.up()
86.       end
87.       movingForward = not movingForward
88.     end
89.
90.     -- стена готова; переход к финальной позиции
91.     if movingForward then
92.       -- черепашка около начальной позиции
93.       for currentLength = 1, length do
94.         turtle.forward()
95.       end
96.     else
97.       -- черепашка около финальной позиции
98.       turtle.forward()
99.     end
100.
101.    -- спуск на землю
102.    for currentHeight = 1, height do
103.      turtle.down()
104.    end
105.
106.    return true
107. end
```

Сохрани программу и выйди из редактора после ввода всех инструкций. Или загрузи этот модуль, выполнив команду `pastebin get wwzvaKuW hare`.

ПОДСЧЕТ БЛОКОВ В ИНВЕНТАРЕ С ПОМОЩЬЮ ФУНКЦИИ COUNTINVENTORY()

Прежде чем черепашка начнет строить, нужно проверить, что в ее инвентаре достаточно блоков для строительства. Предположим, что каждый элемент в инвентаре черепашки является блоком, который будет использован при возведении стены. Функция `countInventory()` возвращает общее количество блоков во всех ячейках инвентаря черепашки.

hare

```
38. -- countInventory() возвращает количество
39. -- всех блоков в инвентаре черепашки
40. function countInventory()
41.   local total = 0
```

Мы сохраним общее количество блоков в переменной, которой изначально присвоили значение `0`. Затем мы объявим переменную `slot`, которая будет отслеживать, какая из ячеек инвентаря проверяется.

Затем цикл `for`, начинающийся в строке 43, выполнит проверку всех 16 ячеек инвентаря, используя переменную `slot`.

hare

```
43.    for slot = 1, 16 do
44.      total = total + turtle.getItemCount(slot)
45.    end
46.   return total
47. end
```

Для каждой ячейки программа вызывает функцию `turtle.getItemCount()` и передает ей значение переменной `slot`. Количество блоков, находящееся в каждой ячейке инвентаря, добавляется к общей сумме. После окончания цикла, в строке 46, функция возвращает итоговое значение, которое теперь содержит количество всех блоков в инвентаре черепашки.

Обрати внимание, что функция `turtle.getItemCount()` подсчитывает все содержимое инвентаря, а не только каменные блоки.

ВЫБОР И РАЗМЕЩЕНИЕ БЛОКА

Чтобы начать строить, нам нужна черепашка, которая будет выкладывать блоки из своего инвентаря. Для этого добавим функцию с именем `selectAndPlaceDown()` в модуль `hare`. Эта функция выбирает первую непустую ячейку, а затем устанавливает блок из этой ячейки под черепашкой.

hare

```
50. -- selectAndPlaceDown() выбирает непустую ячейку
51. -- и помещает блок из нее под черепашку
52. function selectAndPlaceDown()
53.
54.   for slot = 1, 16 do
55.     if turtle.getItemCount(slot) > 0 then
56.       turtle.select(slot)
57.       turtle.placeDown()
58.       return
59.     end
60.   end
61. end
```

В строке 54 начинается цикл `for`, который присваивает переменной `slot` значение `1` в первой итерации, `2` – во второй и т. д., до `16`. Таким образом, функция `selectAndPlaceDown()` сканирует все ячейки инвентаря черепашки.

Внутри этого цикла, код в строке 55 проверяет, сколько элементов находится в ячейке, вызывая функцию `turtle.getItemCount(slot)`. Если в ячейке находится хотя бы один блок, функция `turtle.select(slot)` в строке 56 выбирает этот ячейку, а функция `turtle.placeDown()` из строки 57 размещает блок под черепашкой. Далее код в строке 58 вызывает функцию `selectAndPlaceDown()`.

Эта функция `selectAndPlaceDown()` работает независимо от того, какие строительные блоки находятся в инвентаре черепашки и в каких ячейках они расположены. Если в инвентаре черепашки ничего нет, цикл завершается и никаких действий не производится. Функции `buildWall()` и `buildRoom()` будут вызывать `selectAndPlaceDown()` как подфункцию.

РАЗРАБОТКА АЛГОРИТМА ВОЗВЕДЕНИЯ СТЕН

Теперь создадим функцию `buildWall()`, которой передадим два параметра для управления размерами возводимой стены: один – это длина стены (`length`), другой – ее высота (`height`). Прежде чем писать код функции, давай разрабо-

таем ее алгоритм, которому и будет следовать черепашка, чтобы построить стену.

Когда черепашка начинает строить, она стоит на земле, затем ей надо подняться вверх и положить блок под себя. Далее ей надо двигаться вперед и продолжать подкладывать под себя блоки, до тех пор, пока не получится цепочка блоков заданной длины, `length`. Когда цепочка готова, черепашка повторит этот процесс, но уже в противоположном направлении.

Она будет продолжать движение вперед и назад, выстраивая цепочки блоков, пока не достигнет указанной высоты, `height`. Предположим, что значение переменной `length` равно `4`, а переменной `height` – `2`. На рис. 11.2 показана будущая стена размером 4 × 2 блока и то, как черепашка начинает ее строить.

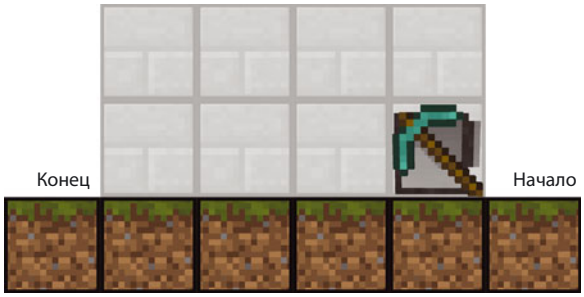


Рис. **11.2.** *Вид сбоку черепашки и будущей стены размером 4 × 2 блока*

Для начала черепашка должна подняться и поместить блок под себя, как показано на рис. 11.3.

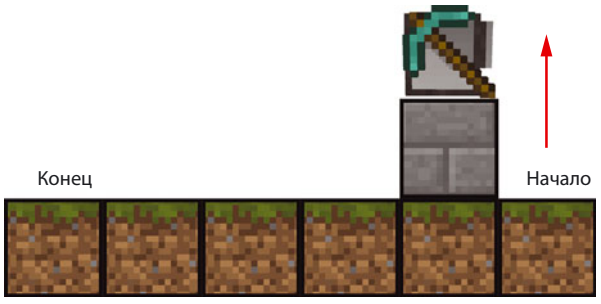


Рис. **11.3.** *Черепашка поднимается и помещает блок под себя*

Поскольку черепашка строит стену длиной в четыре блока, черепашка перемещается вперед на три пролета, каждый раз помещая под себя блок. Если бы черепашка строила стену длиной в шесть блоков, она продвигалась бы вперед на пять пролетов. Обрати внимание на шаблон: количество перемещений черепашки равно `length - 1`. Чтобы уложить `length` блоков, черепашка должна переместиться на один шаг для укладки каждого блока, кроме последнего, в результате чего ей надо совершить `length - 1` перемещений. Таким образом, функция `buildWall()` позволяет строить стены любой длины.

Когда черепашка построит первую цепочку блоков для нашей стены размером 4 × 2 блока, результат будет выглядеть так, как показано на рис. 11.4.

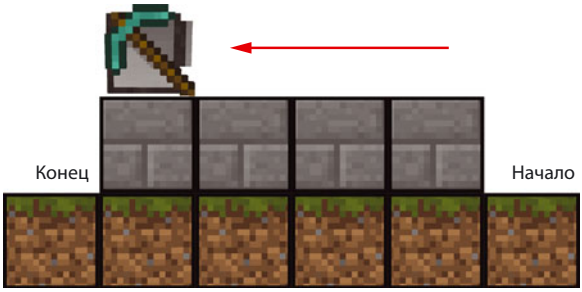


Рис. **11.4.** *Черепашка движется вперед, размещая блоки под собой*

Черепашка повторяет этот процесс, за исключением того, что на этот раз она движется назад, как показано на рис. 11.5.

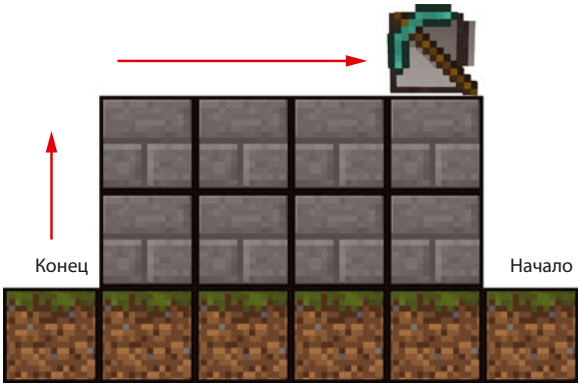


Рис. **11.5.** *Черепашка движется вверх, а затем назад, размещая блоки под собой*

Если бы черепашке было задано большее количество рядов, она повторила бы весь процесс соответствующее количество раз, двигаясь вперед и назад. Но, поскольку черепашка построила заданное количество рядов, указанное в нашем примере, она останавливается. В главе 12 мы возведем четыре стены, чтобы построить комнату, поэтому алгоритм должен всегда возвращать черепашку на землю на противоположном конце от начала строительства. Таким образом, черепашка сможет начать постройку следующей стены.

Существует два варианта перемещения черепашки в конец стены, в зависимости от того, в каком направлении она двигалась в последний раз. Если черепашка в последний раз двигалась вперед, ее надо переместить вниз `height` раз. Если черепашка в последний раз двигалась в обратном направлении, ее нужно переместить вперед `length` раз, а затем опустить `height` раз. Оба варианта поместят черепашку на земле в конце стены, как показано на рис. 11.6.

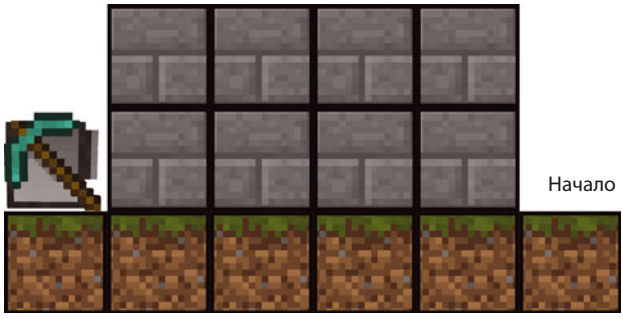


Рис. **11.6.** *Черепашка заканчивает строительство на земле в дальнем конце стены*

Функция `buildWall()` позволяет строить стены любой длины и высоты. Например, если у тебя достаточно блоков, ты можешь сделать высокую стену, как показано на рис. 11.7. Для черепашки это не имеет значения.

Рис. **11.7.** *Высокая стена размером 4 × 12 блоков построена по тому же алгоритму, что и стена размером 4 × 2 блока*

Давай рассмотрим функцию, которая реализует этот алгоритм.

ФУНКЦИЯ BUILDWALL()

Функция `buildWall()` начинается с подсчета количества предметов в инвентаре черепашки. Предполагается, что все эти элементы – это блоки, которые будут использоваться для строительства стены. Количество блоков, необходимое для построения стены длиной `length` и высотой `height`, составляет `length * height` штук, что и проверяется в строке 67.

hare

```
64. -- buildWall() создает стену,
65. -- расположенную перед черепашкой
```

```
66. function buildWall(length, height)
67.   if hare.countInventory() < length * height then
68.     return false -- недостаточно блоков
69.   end
```

Если у черепашки недостаточно блоков для построения стены, функция в строке 68 возвращает значение `false`, а затем завершает выполнение функции.

Если блоков достаточно, исполнение переходит к строке 71, и черепашка перемещается вверх на один блок, чтобы начать строительство цепочки блоков. Затем переменной `movingForward` присваивается значение `true`.

hare

```
71.   turtle.up()
72.
73.   local movingForward = true
```

Эта переменная отслеживает направление движения черепашки. При строительстве первого ряда черепашка движется вперед, поэтому значение переменной `movingForward` устанавливается равным `true`. Но для постройки следующего ряда черепашке нужно двигаться назад, поэтому значение переменной изменяется на `false`. Значение переменной будет чередоваться в каждой цепочке блоков, поскольку черепашка будет менять направление движения.

Чтобы следить за высотой стены, построенной черепашкой, программе нужна еще одна переменная, которую мы назовем `currentHeight`. В строке 75 запускается цикл `for`, в котором переменная `currentHeight` выполняет итерации от `1` до `height`. На первой итерации переменной `currentHeight` присваивается значение `1`, на следующей – значение `2` и так далее, до последней итерации, где переменной `currentHeight` присваивается значение переменной `height`.

hare

```
75.   for currentHeight = 1, height do
76.     for currentLength = 1, length do
```

```
77.         selectAndPlaceDown() -- размещение блока
78.         if movingForward and currentLength ~= length
            then
79.           turtle.forward()
80.         elseif not movingForward and currentLength ~=
            length then
81.           turtle.back()
82.         end
83.       end
```

Но по мере увеличения высоты стены также необходимо отслеживать длину каждого ряда на каждом шаге высоты. Для этого мы будем использовать другую переменную, называемую `currentLength`, которая объявляется во вложенном цикле `for` в строке 76. Этот цикл вложен в цикл `for`, начавшийся в строке 75. На каждой итерации основного цикла `for` программа запускает и вложенный цикл `for`. Итерации этого вложенного цикла производятся согласно значению переменной `currentLength` от `1` до `length`. На рис. 11.8 показано, какие значения переменные `currentHeight` и `currentLength` принимают в каждой точке построения стены размером 4 × 4 блока. Красная стрелка указывает направление движения черепашки.

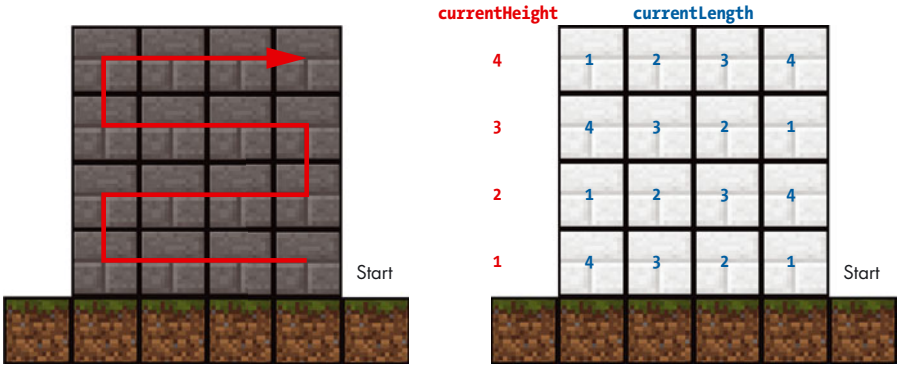


Рис. **11.8.** *Значения переменных* `currentHeight` *и* `currentLength` *в каждой позиции построения стены размером 4 × 4 блока*

Внутри цикла, в строке 77 вызывается функция `selectAndPlaceDown()`, чтобы установить блок под черепашкой. Код в строках 78–82 перемещает черепашку вперед или на-

зад, в зависимости от значения переменной `movingForward` и текущего положения черепашки. Переменная `currentLength` подсчитывает, сколько блоков по всей длине ряда уже положено. Когда ее значение достигает `length`, черепашка останавливается, чтобы не выйти за край стены.

Напомню, что черепашке не нужно двигаться на последней итерации. Поскольку цикл `for`, начавшийся в строке 76, производит итерации от `1` до `length`, на последнем шаге переменная `currentLength` принимает значение `length`. Поэтому условия в строках 78 и 80 выполняются только в том случае, если значение переменной `currentLength` не равно (`~=`) `length`.

Инструкция `end` в строке 82 завершает цикл `for`, начавшийся в строке 76. Этот цикл завершается, когда полностью заканчивается строительство ряда.

Далее, если черепашка не находится на последней итерации цикла `for`, начавшегося в строке 75, ей надо переместиться на одну позицию вверх. Вызов функции `turtle.up()` в строке 85 выполнит это перемещение только в том случае, если значение переменной `currentHeight` не равно `height` (то есть если цикл `for` не совершает последнюю итерацию).

hare

```
    if currentHeight ~= height then
      turtle.up()
    end
    movingForward = not movingForward
  end
```

Код в строке 87 переключает – присваивает противоположное значение – логической переменной `movingForward`. Если переменной присвоено значение `true`, код присваивает ей значение `false`, и наоборот. Если черепашка двигалась вперед, чтобы построить самый высокий ряд блоков, то переключатель присвоит переменной `movingForward` значение `false`. Если черепашка двигалась назад, чтобы построить самый высокий ряд блоков, то переключатель в строке 87 присвоит переменной `moveForward` значение

true. Код в строке 87 изменяет логическое значение переменной movingForward на противоположное с помощью простой команды not movingForward.

Инструкция end в строке 88 завершает цикл for, который начался в строке 75. Этот цикл завершается, если постройка стены завершена. Теперь черепашке нужно вернуться на землю. Для этого ей нужно попасть на сторону, противоположную началу строительства, а затем спуститься на землю.

Напомню, что код в строке 87 переключает значение переменной movingForward, чтобы изменить направление движения черепашки. Так, если переменной movingForward присвоено значение true, черепашка ближе всего к исходной точке. Поэтому, чтобы сойти со стены, черепашке необходимо переместиться вперед length раз, что она и делает в строках 93 и 94.

hare

```
90.   -- стена готова; переход к финальной позиции
91.   if movingForward then
92.     -- черепашка около начальной позиции
93.     for currentLength = 1, length do
94.       turtle.forward()
95.     end
```

Но если переменной movingForward присвоено значение false, черепашка уже находится на противоположном конце стены, и поэтому ей достаточно переместиться только на один шаг вперед, что и происходит в строке 98.

hare

```
96.   else
97.     -- черепашка около финальной позиции
98.     turtle.forward()
99.   end
```

Независимо от того, двигалась черепашка вперед или назад, чтобы опуститься на землю, ей нужно переместиться на height шагов вниз, поэтому цикл for в строке 102

вызывает функцию `turtle.down()`, совершая `height` итераций.

hare

```
101.   -- спуск на землю
102.   for currentHeight = 1, height do
103.     turtle.down()
104.   end
105.
106.   return true
107. end
```

Когда стена построена, а черепашка находится в финальной позиции, функция `buildWall()` возвращает значение `true` в строке 106, указывая на то, что стена успешно построена. Инструкция `end` в строке 107 завершает код функции `buildWall()`.

Функция `buildWall()`, которую мы добавили в модуль `hare`, очень полезна, потому что возведение стен – это процесс, который ты, возможно, захочешь повторить во многих программах. Но если ты просто хочешь построить стену и не намерен запускать оболочку Lua, потом набирать команды `os.loadAPI('hare')` и `hare.buildWall(4, 2)`, можно написать программу, которая все это сделает самостоятельно. Давай создадим такую программу и назовем ее `buildwall`.

РАЗРАБОТКА И ЗАПУСК ПРОГРАММЫ BUILDWALL

В оболочке командной строки выполни команду `edit buildwall` и введи следующий код:

buildwall

```
1. --[[Программа для строительства стен Эла Свейгарта
2. Строит стены.]]
3.
4. os.loadAPI('hare')
5.
```

```
 6. -- обработка аргументов командной строки
 7. local cliArgs = {...}
 8. local length = tonumber(cliArgs[1])
 9. local height = tonumber(cliArgs[2])
10.
11. if length == nil or height == nil or cliArgs[1] ==
    '?' then
12.   print('Usage: buildwall <length> <height>')
13.   return
14. end
15.
16. print('Building...')
17. if hare.buildWall(length, height) == false then
18.   error('Недостаточно блоков.›)
19. end
20. print('Done.')
```

Сохрани программу и выйди из редактора после ввода всех инструкций.

Установив черепашку и поместив блоки каменного кирпича (или любого другого строительного блока) в ее инвентарь, щелкни по ней правой кнопкой мыши для открытия ее графического интерфейса. Чтобы посмотреть, как черепашка строит стену в четыре блока длиной и два блока высотой, в оболочке командной строки введи `edit buildwall 4 2` и нажмите клавишу **Enter**.

Если при запуске этой программы возникают ошибки, тщательно сравни свой код с листингом в этой книге, чтобы исключить возможные опечатки. Если исправить программу не удается, удали файл, выполнив команду `delete buildwall`, а затем загрузи код, выполнив команду `pastebin get 1aZ8BhNX buildwall`.

ЗАГРУЗКА МОДУЛЯ HARE

Первая часть программы `buildwall` загружает модуль `hare` и вызывает функцию `hare.buildWall()`.

buildwall

```
 1. --[[Программа для строительства стен Эла Свейгарта
 2. Строит стены.]]
```

```
3.
4. os.loadAPI('hare')
```

Хотя значительная часть работы выполняется с помощью кода модуля `hare`, программа `buildwall` содержит некоторые дополнительные функции. При вызове функции `hare.buildWall()` программа должна знать, какие значения следует передавать в виде параметров `length` и `height`. Программа `buildwall` может получать эти значения из аргументов командной строки, которые введет игрок при запуске программы `buildwall` из оболочки командной строки. Ранее ты передал программе `buildwall` в качестве аргументов командной строки значения `4` и `2`. Аргументы командной строки ты уже использовал при запуске других программ. Например, в главе 2, когда ты запускал программу `label`, ты назначал имя черепашке командой `set Sofonisba` в строке `label set Sofonisba`. Аргументы командной строки сохраняются как специальные табличные значения, называемые массивом. Это объект Lua с именем `{...}`, который состоит из двух фигурных скобок с тремя пробелами между ними. Прежде чем рассмотреть код, который обрабатывает аргументы командной строки, давай посмотрим, как работают массивы.

ИСПОЛЬЗОВАНИЕ МАССИВОВ

Таблицы, о которых ты узнал из главы 7, могут хранить несколько значений. Массивы – это другой тип данных, который тоже может сохранять несколько значений одновременно. Значения в таблице хранятся в виде пар ключ-значение, тогда как значения в массивах хранятся упорядоченно. Значения в массиве также называются элементами массива. Чтобы получить доступ к какому-либо значению, хранящемуся в массиве, нужно указать числовое значение, которое определяет расположение этого элемента массива и называется индексом.

Примечание *Технически массивы в Lua – это еще один тип табличных данных. Таблицы с парами ключ-значение,*

которые мы использовали в предыдущих главах, называются картоподобными таблицами, а массивы – массивоподобными таблицами.

Код для массивов аналогичен коду для таблиц. При создании массива используй фигурные скобки `{}`, как и при создании таблиц, но опусти ключи. Ключи вводить не нужно, потому что в массиве позиция значения – это его индекс.

Чтобы создать собственный массив, в интерактивной оболочке введи следующую команду:

```
lua> pets = {'mouse', 'cat', 'dog'}
```

Чтобы получить доступ к какому-либо значению, сохраненному в массиве, введи имя массива и укажи в квадратных скобках `[]` индекс нужного элемента:

```
lua> pets[1]
mouse
lua> pets[2]
cat
lua> 'I have a pet ' .. pets[3]
I have a pet dog
```

В языке Lua, в отличие от других языков программирования, нумерация элементов массива начинается с 1, поэтому для доступа к первому значению в массиве `pets` нужно ввести команду `pets[1]`.

ИСПОЛЬЗОВАНИЕ АРГУМЕНТОВ КОМАНДНОЙ СТРОКИ

Возвращаясь к программе `buildwall`, скажу, что код в строке 7 сохраняет значения массива аргументов командной строки в переменной с именем `cliArgs`. Эта переменная позволяет получать доступ к отдельным значениям внутри массива с помощью квадратных скобок: `cliArgs[1]` – это первый аргумент командной строки, `cliArgs[2]` – второй и т. д.

buildwall

```
 6. -- обработка аргументов командной строки
 7. local cliArgs = {...}
 8. local length = tonumber(cliArgs[1])
 9. local height = tonumber(cliArgs[2])
```

Все значения аргументов командной строки сохраняются в виде строк, даже если они выглядят как числа, например, `'4'` или `'2'`. Чтобы преобразовать значения из строкового типа в числовой, код в строках 8 и 9 передает аргументы командной строки в функцию `tonumber()`, которая возвращает числовые значения переданных строк. Мы сохраним эти числовые значения в переменных `length` и `height`.

ВЫВОД СООБЩЕНИЙ ПОЛЬЗОВАТЕЛЮ

Если игрок не вводит аргументы командной строки (или вводит только один вместо двух), информации для запуска программы будет недостаточно. Возможно, игрок не знает, что должен ввести значения для переменных `length` и `height`, или, может быть, он забыл набрать их. В таких случаях полезно, чтобы программа отображала сообщение, которое напомнит игроку, как ее правильно запустить. Если игрок укажет символ `?` в качестве первого аргумента командной строки, также появится сообщение.

buildwall

```
11. if length == nil or height == nil or cliArgs[1] ==
    '?' then
12.   print('Usage: buildwall <length> <height>')
13.   return
14. end
```

Если аргументы командной строки отсутствуют или не являются числами, переменным `cliArgs[1]` и `cliArgs[2]` будет присвоено значение `nil`. Если в функцию `tonumber()` передать значение `nil` или строку, которая не содержит числа, она вернет `nil`. Например, если игрок не укажет аргументы командной строки, в строке 8 переменной `length` будет присвоено значение `nil`. Если игрок передает только

один аргумент, в строке 9 значение nil будет присвоено переменной height. Условие в строке 11 проверяет, присвоено ли переменной length или height значение nil, и не указан ли символ '?' в качестве первого аргумента командной строки.

В ситуациях, когда игрок вводит неправильные аргументы или не вводит их вовсе, функция print() выводит сообщение, которое предлагает игроку ввести имя buildwall и аргументы командной строки – length и height. Угловые скобки <>, в которые заключены слова length и height, указывают на то, что игрок не должен вводить сами слова length и height, а указать числовые значения.

ВЫЗОВ ФУНКЦИИ HARE.BUILDWALL() ДЛЯ ПОСТРОЙКИ СТЕНЫ

Всю работу в программе buildwall выполняют функции buildWall(), selectAndPlaceDown() и countInventory() модуля hare. Таким образом, код программы buildwall очень прост. После обработки аргументов командной строки он просто вызывает функцию hare.buildWall() в строке 16 и присваивает значения переменным length и height.

buildwall

```
16. print('Building...')
17. if hare.buildWall(length, height) == false then
18.   error('Недостаточно блоков.›)
19. end
20. print('Done.')
```

Код в строке 16 вызывает функцию print(), выводящую сообщение, информирующее пользователя о запуске программы. Если блоков недостаточно, функция hare.buildWall() возвращает значение false и программа не запускается. Это проверяется в строке 17, в этом случае пользователь видит сообщение об ошибке. Код в строке 20 выводит сообщение о том, что программа завершила постройку стены, или сообщает об ошибке.

Вот и весь код программы `buildwall`! За все остальное отвечает код модуля `hare`.

ДОПОЛНИТЕЛЬНОЕ ЗАДАНИЕ: ПРОГРАММА MYLABEL С АРГУМЕНТАМИ КОМАНДНОЙ СТРОКИ

Вернись к программе `mylabel` из главы 3 и измени код так, чтобы игрок мог указать имя черепашки не с помощью функции `io.read()`, а с помощью аргумента командной строки. Для этого добавь строку кода `local cliArgs = {...}`. Теперь записать новое имя черепашки можно, используя код `cliArgs[1]` вместо функции `io.read()`. Обязательно добавь вывод сообщения пользователю, если он не укажет новое имя в качестве аргумента командной строки.

Обрати внимание: встроенная программа `label` также поддерживает работу аргументов командной строки. Использование аргументов командной строки ускоряет выполнение программ, так как игрок может вводить значения параметров непосредственно при запуске программы.

ЧТО МЫ УЗНАЛИ

В этой главе мы дополнили модуль `hare` новыми функциями. Функция `countInventory()` возвращает общее количество блоков (предметов) в инвентаре черепашки. Это полезно, так как можно проверить, достаточно ли у черепашки блоков для строительства. Функция `selectAndPlaceDown()` позволяет черепашке выбрать определенный блок в инвентаре и использовать его в мире Minecraft. С помощью этих функций ты можешь написать код для постройки стен любого размера!

Ты узнал, как писать программы, которые принимают аргументы командной строки. Ты уже использовал аргументы командной строки в главе 2 (например, когда вводил команду `set Sofonisba` в строке `label set Sofonisba`). Но в этой главе ты узнал, что можешь получить доступ к табличным значениям объекта `{...}` и прочитать аргументы командной строки, которые игрок ввел при запуске программы.

Если твои программы будут запускать другие пользователи, они, вероятно, не будут читать код Lua, чтобы выяснить, какие аргументы командной строки необходимо передать твоей программе. Поэтому программа может выводить сообщение, которое предоставит краткое описание необходимых аргументов командной строки.

В главе 12 мы создадим стеностроительную программу (вот такие у меня каламбуры), чтобы строить комнаты с четырьмя стенами.

12

СТРОИТЕЛЬСТВО КОМНАТ

Если соединить четыре стены, получится комната! В этой главе мы создадим функцию `buildRoom()`, которая будет вызывать функцию `buildWall()` четыре раза, чтобы построить комнату, подобную той, что показана на рис. 12.1. Эта глава короткая и простая, потому что ты уже выполнил значительную часть работы, когда писал функцию `buildWall()` в главе 11. А когда ты закончишь работу над функцией `buildRoom()`, то с ее помощью сможешь построить сложное сооружение, например замок!

Рис. **12.1.** *Комната из каменного кирпича, размером 5 × 6 × 4 блоков*

ПРОЕКТИРОВАНИЕ АЛГОРИТМА ПОСТРОЕНИЯ КОМНАТ

Давай разработаем алгоритм, который позволит строить комнаты любого размера. Код этого алгоритма поместим в функцию с именем `buildRoom()` в модуль `hare`. Например, если черепашке нужно построить комнату длиной в три блока и шириной в четыре блока, она может построить четыре стены, как показано на рис. 12.2. Черепашка начинает с левого нижнего угла комнаты, а затем строит стены, двигаясь по часовой стрелке.

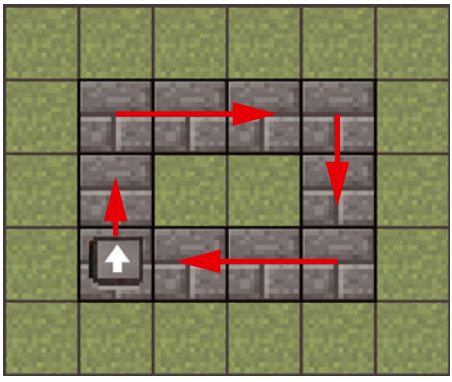


Рис. **12.2.** *Вид сверху на черепашку, строящую комнату размером 3 × 4 блока. Начав строительство, черепашка повернута в направлении белой стрелки*

Поскольку стены перекрываются в углах комнаты, длина и ширина комнаты отличаются от размеров ее стен. Например, чтобы построить комнату размером 3 × 4 блока, как показано на рис. 12.2, ты не сможешь просто построить две стены длиной в три блока и две стены длиной в четыре блока, так как углы должны перекрываться. Чтобы правильно задать длину стен комнаты, надо учитывать перекрытие углов. Взгляни на размер каждой стены на рис. 12.3, где изображены комнаты с размерами 3 × 4 и 4 × 6 блоков.

Как видишь, комната размером 3 × 4 блока состоит из двух стен длинною в 2 блока и двух стен длиной в 3 блока. Комната размером 4 × 6 блоков состоит из двух стен длиной 3 блока и двух стен длиной 5 блоков.

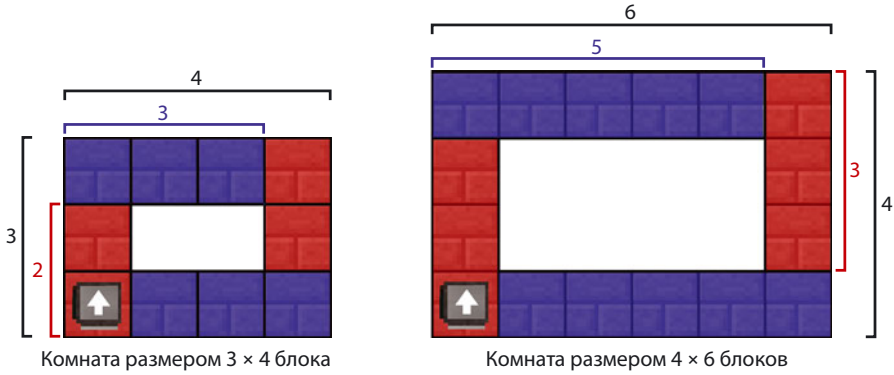


Рис. **12.3.** *Комнаты с размерами 3 × 4 (слева) и 4 × 6 (справа) блоков*

Обрати внимание на шаблон нового алгоритма. Четыре стены имеют два разных размера: один размер – ширина комнаты минус один, и второй – длина комнаты минус один. Алгоритм вызывает функцию `buildWall()`, которую мы создали в главе 11. Черепашка начинает движение в левом нижнем углу, как показано на рис. 12.3, и начинает строительство с левой стороны комнаты (она показана красным цветом). Затем, в левом верхнем углу рисунка, черепашка поворачивает направо, в соответствии с алгоритмом, и строит другую стену (показана синим цветом). Затем наш робот снова поворачивает направо, строит красную стену с правой стороны комнаты, и наконец поворачивая направо, строит синюю стену внизу комнаты. Когда черепашка заканчивает строительство четвертой стены, то оказывается там, откуда она начинала. Используя этот шаблон в качестве алгоритма, функция `buildRoom()` позволяет создавать комнаты любого размера.

ДОРАБОТКА МОДУЛЯ HARE

Поместим функцию `buildRoom()` в модуль `hare`, как это было сделано с функцией `buildWall()`, чтобы ее могли использовать и другие программы. В оболочке командной строки выполни команду `edit hare`. Перемести курсор в конец файла и добавь следующий код:

hare

```
     ...пропуск...
110. -- buildRoom() строит четыре стены
111. -- и потолок
112.   function buildRoom(length, width, height)
113.   if hare.countInventory() < (((length - 1) *
       height * 2) +
           ((width - 1) * height * 2)) then
114.     return false -- недостаточно блоков
115.   end
116.
117.   -- строительство четырех стен
118.   buildWall(length - 1, height)
119.   turtle.turnRight()
120.
121.   buildWall(width - 1, height)
122.   turtle.turnRight()
123.
124.   buildWall(length - 1, height)
125.   turtle.turnRight()
126.
127.   buildWall(width - 1, height)
128.   turtle.turnRight()
129.
130.   return true
131. end
```

Сохрани программу и выйди из редактора после ввода всех инструкций. Этот модуль можно также загрузить, выполнив команду `pastebin get wwzvaKuW hare`.

ВЫЧИСЛЕНИЕ ОБЩЕГО КОЛИЧЕСТВА БЛОКОВ, НЕОБХОДИМОГО ДЛЯ СТРОИТЕЛЬСТВА КОМНАТЫ

Прежде чем функция `buildRoom()` начнет строительство комнаты, необходимо проверить, что в инвентаре черепашки достаточно строительных блоков. Для этого нужно вычислить количество блоков, необходимое для возведения всех четырех стен комнаты. Количество блоков, необходи-

мое для одной стены, рассчитывается в строке 67 модуля hare, вот так:

```
length * height
```

Но на этот раз длина стены на один блок короче длины комнаты. Первая из четырех стен должна быть длиной length - 1, поэтому наши вычисления принимают вид:

```
(length - 1) * height
```

И поскольку черепашка построит две таких стены, мы умножим это число на 2:

```
(length - 1) * height * 2
```

Остальные две стены будут на один блок короче ширины комнаты, то есть длиной в width - 1 блок, поэтому мы добавим блоки для этих стен в расчет, следующим образом:

```
((length - 1) * height * 2) + ((width - 1) * height * 2)
```

Это окончательная формула для вычисления количества блоков, которое потребуется для строительства комнаты. Ее мы будем использовать в функции buildRoom().

КОД ФУНКЦИИ BUILDROOM()

Чтобы узнать, сколько блоков находится в инвентаре черепашки, код в строке 113 вызывает функцию hare.countInventory(). Если это число меньше необходимого количества блоков, которое рассчитывается по формуле из предыдущего раздела, исполнение переходит к строке 114. В этой строке функция buildRoom() возвращает значение false. Код функции использует это возвращаемое значение для оповещения о том, что комната не была построена.

hare

```
110. -- buildRoom() строит четыре стены
```

```
111. -- и потолок
112.   function buildRoom(length, width, height)
113.   if hare.countInventory() < (((length - 1) *
       height * 2) +
           ((width - 1) * height * 2)) then
114.     return false -- недостаточно блоков
115.   end
```

В противном случае исполнение переходит к коду, следующему за этой функцией. Таким образом, если блоков достаточно, происходит вызов функции `buildWall()` и код в строке 118 начинает строительство первой стены.

hare

```
118.   buildWall(length - 1, height)
```

Когда черепашка построит одну стену, по окончании строительства функция `buildWall()` перемещает черепашку обратно на землю. Если черепашка строит комнату размером 3 × 4 блока, то после исполнения кода в строке 118, строение будет выглядеть так, как показано на рис. 12.4.

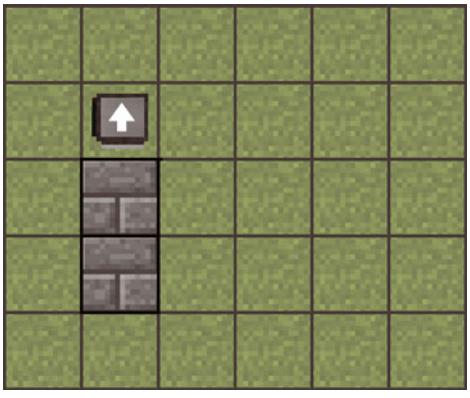


Рис. **12.4.** *Вид комнаты размером 3 × 4 блока после завершения строительства первой стены*

Прежде чем черепашка сможет построить вторую стену, она должна повернуть направо. Такой поворот производится в строке 119 с помощью функции `turtle.turnRight()`, а код в строке 121 начинает строительство второй стены.

hare

```
119.    turtle.turnRight()
120.
121.    buildWall(width - 1, height)
```

Обрати внимание, что длина этой стены равна `width - 1`, где `width` – ширина комнаты. Если черепашка строит комнату размером 3 × 4 блока, после исполнения кода в строке 121 комната будет выглядеть так, как показано на рис. 12.5.

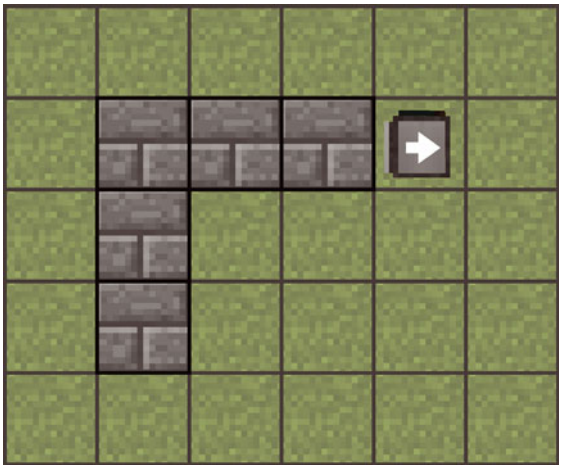


Рис. **12.5.** *Вид комнаты размером 3 × 4 блока после постройки второй стены*

В строках 124–128 происходит строительство двух других стен:

hare

```
122.    turtle.turnRight()
123.
124.    buildWall(length - 1, height)
125.    turtle.turnRight()
126.
127.    buildWall(width - 1, height)
128.    turtle.turnRight()
129.
130.    return true
131. end
```

После исполнения этого кода строительство комнаты завершается, и функция `buildRoom()` возвращает значение `true` в строке 130, сообщая об успешной постройке комнаты. Черепашка возвращается в исходную позицию начала строительства, поворачивается в первоначальном направлении, но останется наверху стен, которые она построила, как показано на рис. 12.6.

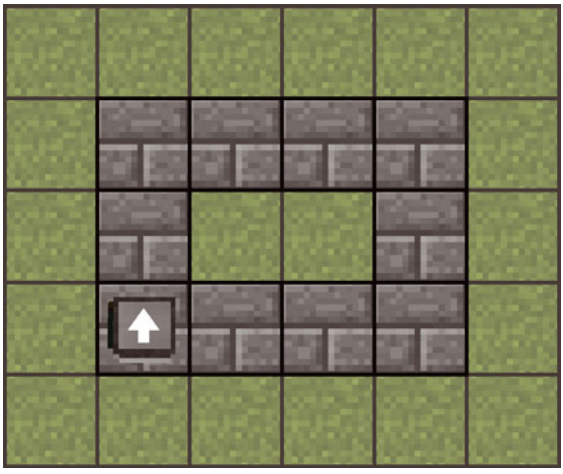


Рис. **12.6.** *Комната размером 3 × 4 блока построена, черепашка возвращается к исходной позиции, но остается наверху*

Инструкция `end` в строке 131 заканчивает блок кода функции `buildRoom()`.

КОД ПРОГРАММЫ BUILDWALL

В главе 11 мы создали программу `buildwall`, которая вызывает функцию `hare.buildWall()`, чтобы игрок мог легко построить стену в оболочке командной строки. Теперь создадим программу, аналогичную `buildroom`, чтобы вызывать функцию `hare.buildRoom()`. В оболочке командной строки выполни команду `edit buildroom` и введи следующий код:

buildroom

```
1. --[[Программа для строительства комнат Эла Свейгарта
```

```
 2. Строит комнаты из четырех стен.]]
 3.
 4. os.loadAPI('hare')
 5.
 6. -- обработка аргументов командной строки
 7. local cliArgs = {...}
 8. local length = tonumber(cliArgs[1])
 9. local width = tonumber(cliArgs[2])
10. local height = tonumber(cliArgs[3])
11.
12. if length == nil or width == nil or height == nil or
    cliArgs[1] == '?' then
13.   print('Usage: buildroom <length> <width> <height>')
14.   return
15. end
16.
17. print('Building...')
18. if hare.buildRoom(length, width, height) == false
    then
19.   error('Недостаточно блоков.›)
20. end
21. print('Done.')
```

Сохрани программу и выйди из редактора после ввода всех инструкций. Тебе также понадобится модуль `hare`, который можно скачать, выполнив команду `pastebin get wwzvaKuW hare`.

Как и `buildwall`, программа `buildroom` основывается главным образом на функциях модуля `hare`. Она получает аргументы командной строки из массива `{...}`, выводит сообщение пользователю, если это необходимо, а затем вызывает функцию `hare.buildRoom()`. Наша программа почти идентична программе `buildwall`, за исключением того, что мы используем функцию `hare.buildRoom()` вместо `hare.buildWall()`.

ЗАПУСК ПРОГРАММЫ BUILDWALL

Установив черепашку и положив в ее инвентарь 72 блока каменных кирпичей (или любых других строительных блока), щелкни по ней правой кнопкой мыши, чтобы открыть

графический интерфейс. Чтобы посмотреть, как черепашка строит комнату из пяти блоков в длину, шести блоков в ширину и четырех блоков в высоту, в оболочке командной строки выполни команду `buildroom 5 6 4`.

С помощью программ `buildwall` и `buildroom` ты можешь быстро возводить высокие замки или другие структуры, не укладывая вручную ни одного блока! Ты можешь создать несколько черепашек, запускающих либо программу `buildRoom`, либо `buildWall` для совместной работы, как показано на рис. 12.7.

Рис. **12.7.** *Черепашки, строящие замок, под управлением программ* `buildwall` *и* `buildroom`

Если при запуске этой программы возникают ошибки, тщательно сравни свой код с листингом в этой книге, чтобы найти возможные опечатки. Если исправить программу не удается, удали файл, выполнив команду `delete buildroom`, а затем загрузи ее снова, выполнив команду `pastebin get U0WVM4wg buildroom`.

ДОПОЛНИТЕЛЬНОЕ ЗАДАНИЕ: КОМНАТА В ФОРМЕ КРЕСТА

Попробуй создать программу, которая построит комнату в форме креста, а не прямоугольную, как показано на рис. 12.8. Такая программа будет похожа на `buildroom`, но будет включать двенадцать вызовов функции `hare.buildWall()` вместо четырех, а также нужно будет просчитать все повороты, которые должна совершить черепашка.

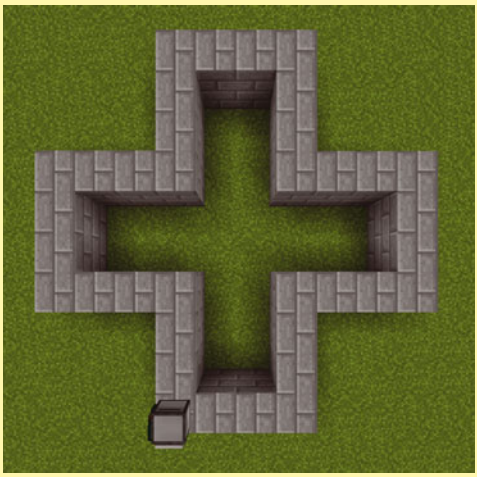


Рис. **12.8.** *Черепашка построила комнату в форме креста*

ЧТО МЫ УЗНАЛИ

Хотя в этой главе я и не объяснил новых концепций программирования, это была хорошая практика по разработке алгоритмов. Чем больше идей о том, что ты хочешь получить от черепашки, превращается в конкретные описания действий для нее, тем лучшим программистом ты становишься.

В главе 11 ты написал функцию для постройки стен, в этой главе – функцию для строительства комнат. А в следующей главе разработаешь алгоритм построения полов и потолков.

13

СТРОИТЕЛЬСТВО ПОЛА И ПОТОЛКА

Комнаты, которые мы построили с помощью функции `buildRoom()` в главе 12, крепки, но не защищают от дождя. В этой главе мы напишем функцию `buildFloor()` для строительства полов и потолков, а также разработаем программу `buildfloor` для вызова этой функции.

Для строительства пола и потолка будет использоваться одна и та же функция, потому что потолок – это всего лишь пол, построенный сверху, как показано на рис. 13.1.

Согласно алгоритму создания полов, черепашка будет перемещаться вдоль и поперек прямоугольной области, выполняя определенные действия в каждой ее точке. В этой главе черепашка будет размещать блоки по всей площади прямоугольной области, но этот алгоритм достаточно гибок, чтобы выполнять и другие задачи. К примеру, в главе 14 мы используем этот алгоритм для разработки программы черепашки-фермера, которая будет засеивать поля прямоугольной формы. Использование гибких алгоритмов вместо жестко запрограммированных решений позволяет использовать один и тот же код для решения множества задач!

Рис. **13.1.** *Код для строительства пола может строить и потолки*

ПРОЕКТИРОВАНИЕ АЛГОРИТМА ВЫКЛАДЫВАНИЯ ПЕРЕКРЫТИЯ

Построение горизонтальной поверхности подобно строительству стены, но вместо того, чтобы двигаться вверх после укладки ряда (строки) блоков для строительства следующего ряда, черепашка будет перемещаться влево или вправо, переходя к следующему ряду (столбцу). Процесс размещения блоков пола/потолка я назову выкладыванием, а выстраиваемую горизонтальную поверхность – перекрытием.

Сконструируем функцию `sweepField()`, которая будет принимать три параметра: `length`, `width` и `sweepFunc`. Параметры `length` и `width` будут определять размер перекрытия, как показано на рис. 13.2.

Язык программирования Lua допускает использование функций в виде параметров. Параметру `sweepFunc` будет передаваться функция, которая будет вызываться в каждой позиции перекрытия, где потребуется выполнение каких-либо действий.

Когда функция `buildFloor()` будет вызывать функцию `sweepField()` для строительства перекрытия, то в виде параметра `sweepFunc` будет передана функция, которая укладывает блоки под черепашкой при каждом ее перемещении. В качестве значений параметров можно передавать любые функции, таким образом, функцию `sweepField()` можно настроить на выполнение любых действий. Но, поскольку основное внимание в этой главе уделяется созданию перекрытий, мы рассмотрим алгоритм программы `buildfloor`, которая будет строить именно перекрытия – полы и потолки.

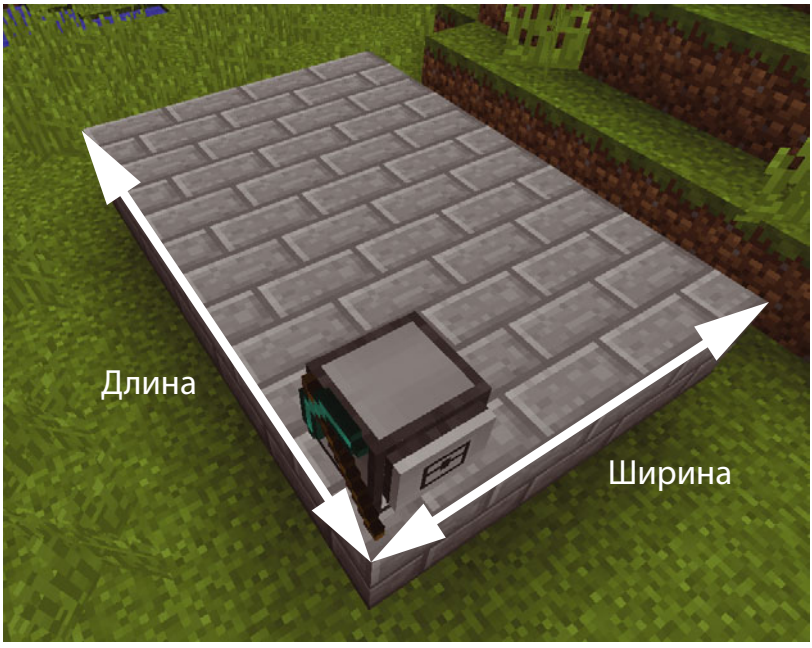


Рис. **13.2.** *Строительство перекрытия: параметр* `length` *– это расстояние перед черепашкой, а параметр* `width` *– это расстояние справа от черепашки*

СТРОИТЕЛЬСТВО ПЕРЕКРЫТИЙ

Программа `buildfloor` будет принимать два аргумента командной строки: `length` и `width`. Как упоминалось ранее, эти аргументы определяют размер нашего перекрытия.

На рис. 13.3 стрелкой сверху вниз обозначен путь, который проделает черепашка, чтобы выложить перекрытие размером 3 × 4 блока.

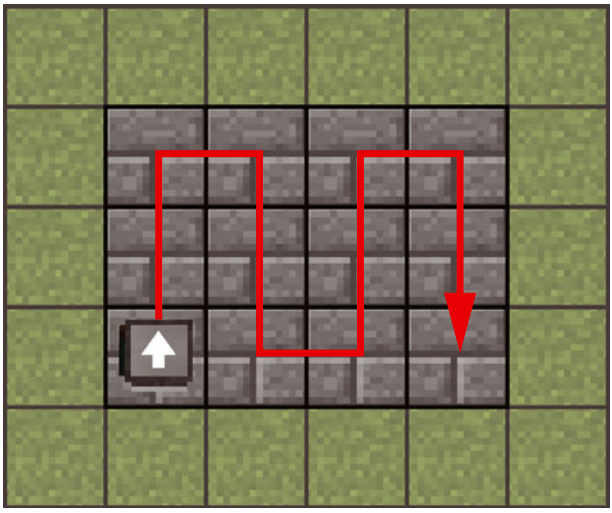


Рис. **13.3.** *Путь, который проделает черепашка, чтобы выложить перекрытие размером 3 × 4 блока*

Чтобы построить перекрытие, черепашка сначала движется вперед, вдоль первого ряда (столбца) (если смотреть на рисунок), помещая под себя камень при каждом перемещении. Для выбора функции, которая будет производить требуемые действия, используется параметр `sweepFunc`, который мы рассмотрим более подробно в разделе **«Вызов функции** `sweepFunc()`» далее в этой главе. Пока просто запомни, что в этот параметр передается функция, которая будет вызываться при каждом перемещении черепашки в пределах строящегося перекрытия. В программе `buildFloor` функция, передаваемая параметру `sweepFunc`, – это `selectAndPlaceDown()`. Это функция модуля `hare`, которая выбирает непустую ячейку инвентаря черепашки и устанавливает блок из выбранной ячейки под черепашкой. Так как черепашка начинает движение в левом нижнем углу (если смотреть сверху), ей нужно совершить `length - 1` перемещений вдоль первого ряда (столбца), чтобы достигнуть последнего блока, как показано на рис. 13.4.

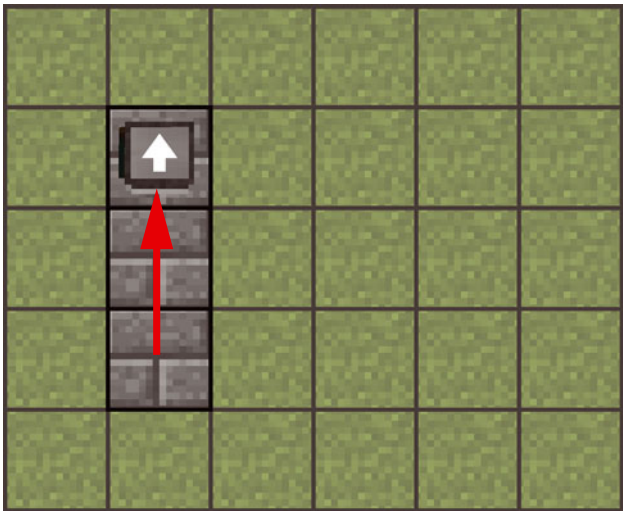


Рис. **13.4.** *После* `length - 1` *перемещений черепашка находится над последним блоком первого ряда (столбца)*

Чтобы перейти к следующему ряду (столбцу), черепашка должна повернуть направо и передвинуться на одну позицию, а затем снова повернуть направо, чтобы оказаться в положении, показанном на рис. 13.5, справа.

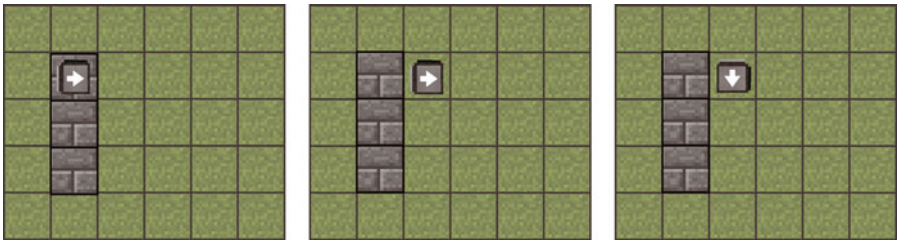


Рис. **13.5.** *Черепашка поворачивается направо, движется вперед и снова поворачивается, чтобы подготовиться к построению следующего ряда (столбца)*

Далее черепашка выполняет код, аналогичный использованному при строительстве первого ряда (столбца). Она перемещается вперед `length - 1` раз, вызывая функцию `selectAndPlaceDown()` при каждом передвижении, чтобы установить под собой блок. Таким образом, выстраивается два ряда (столбца) блоков, как показано на рис. 13.6.

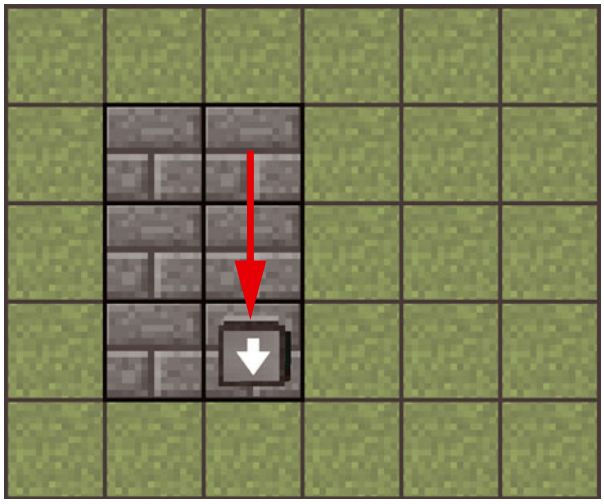


Рис. **13.6.** *Положение черепашки после завершения строительства второго ряда (столбца)*

Теперь, чтобы перейти к следующему ряду (столбцу), черепашка должна повернуть налево, передвинуться вперед и снова повернуть налево, как показано на рис. 13.7.

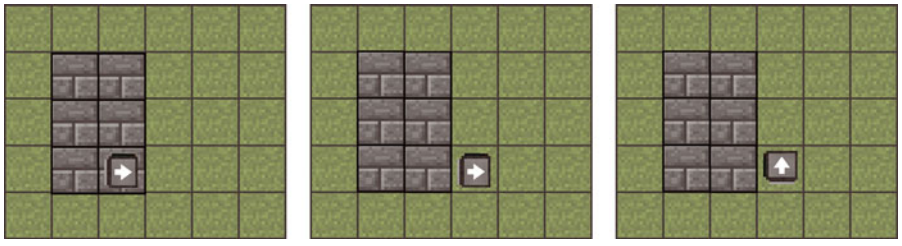


Рис. **13.7.** *После того, как построен второй ряд (столбец), черепашка должна повернуть налево, шагнуть вперед, и снова повернуть налево*

Так черепашка продолжает двигаться вперед и назад по рядам до тех пор, пока количество построенных рядов (столбцов) не станет равным заданной ширине перекрытия. Когда черепашка завершит выполнение этой части алгоритма, перекрытие будет построено, но работа еще не будет закончена! После того, как перекрытие готово, можно было бы запустить программу `buildroom`, чтобы построить комнату. Но в таком случае нам нужно, чтобы черепашка находилась в той же позиции и была ориентирована в том

же направлении, что и при начале строительства перекрытия. Возвращение черепашки в исходное положение гарантирует, что стены, которые она построит при помощи программы `buildroom`, будут точно соответствовать построенному перекрытию.

ВОЗВРАТ К ИСХОДНОЙ ПОЗИЦИИ

Возврат черепашки в исходное положение и направление, соответствующие началу строительства, возможно двумя путями, в зависимости от того, из четного или нечетного количества блоков составлена ширина перекрытия.

Если перекрытие по ширине состоит из четного количества блоков, черепашка будет находиться в правом нижнем углу, как показано на рис. 13.8.

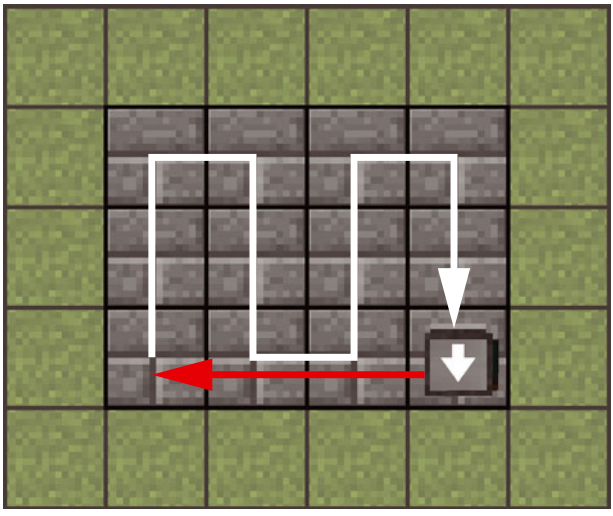


Рис. **13.8.** *Черепашка находится в самом правом блоке нижнего ряда (строки), если перекрытие по ширине состоит из четного количества блоков*

Красная линия на рис. 13.8 показывает путь, который должна совершить черепашка, чтобы вернуться в исходное положение и повернуться в исходном направлении. В этом случае черепашка должна:

1. повернуть направо;

2. переместиться вперед `width - 1` раз;
3. повернуть направо.

Однако если перекрытие по ширине состоит из нечетного количества блоков, черепашка заканчивает в верхней части перекрытия, как показано на рис. 13.9.

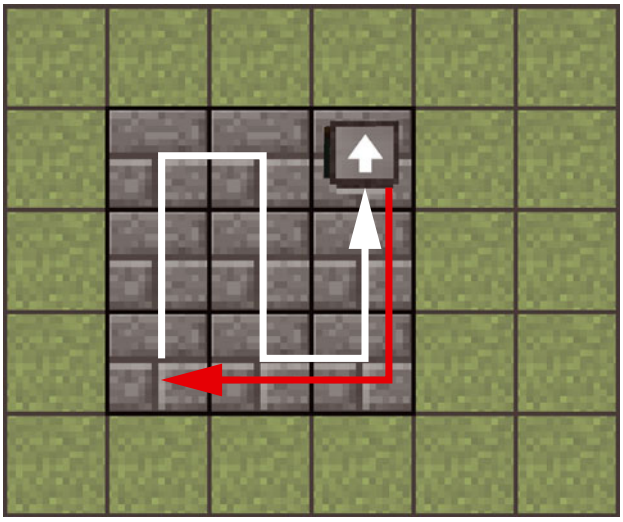


Рис. **13.9.** *Черепашка находится в самом правом блоке верхнего ряда (строки), если перекрытие по ширине состоит из нечетного количества блоков*

Красная линия на рис. 13.9 показывает путь, который должна совершить черепашка, чтобы вернуться к исходному положению и направлению. В этом случае черепашка должна:

1. переместиться назад `length - 1` раз;
2. повернуть налево;
3. переместиться вперед `width - 1` раз;
4. повернуть направо.

Алгоритм возвращения ты заложишь в функцию `sweepField()`, но прежде, чем писать саму функцию, нужно понять, как передавать одну функцию другой.

ПЕРЕДАЧА ОДНОЙ ФУНКЦИИ В ДРУГУЮ

Функции `buildWall()` и `buildRoom()`, которые мы использовали в главах 11 и 12, настраиваемые и позволяют строить стены и комнаты любого размера, потому что принимают параметры длины, ширины и высоты – `length`, `width` и `height`. Эти параметры позволяют создавать стены и комнаты разных размеров без изменения исходного кода.

В этой главе мы добавим в модуль `hare` функцию `sweepField()` и напишем программу `buildFloor` для вызова данной функции. В настоящее время функция `sweepField()` перемещает черепашку по прямоугольному перекрытию, помещая под черепашку каменный кирпич при каждом ее перемещении.

Но функцию `sweepField()` можно изменить так, чтобы черепашка, перемещаясь по прямоугольной ферме, сажала семена, собирала пшеницу, либо рыла под собой прямоугольные шахты. Код, который перемещает черепашку по шаблону выкладывания, остается неизменным, но действия черепашки могут быть различными.

Такую гибкость в настройке функции `sweepField()` обеспечивает параметр `sweepFunc`, в который передается функция, определяющая действия, производимые черепашкой при ее передвижении по прямоугольному полю. В Lua ты можешь передавать в виде аргументов функции, так же, как ты передавал функциям целые числа или строки. Давай для примера создадим простую программу, которая передает функцию в виде параметра другой функции. В оболочке командной строки введи команду `edit announce` для создания новой программы и введи следующий код:

announce

```
1. function hello()
2. print('Hello there!')
3. end
4.
5. function goodbye()
6. print('Goodbye!')
```

```
7. end
8.
9. function announce(func)
10. print('About to call the function.')
11. func()
12. print('Function called.')
13. end
14.
15. announce(hello) -- после имени hello нет круглых
    скобок
16. announce(goodbye) -- после имени goodbye нет круглых
    скобок
```

Сохрани программу и выйди из редактора после ввода всех инструкций. Ты также можешь загрузить эту программу, выполнив команду `pastebin get sML2CbZ3 announce`.

После запуска этой программы в оболочке командной строки ты увидишь сообщение:

```
> announce
About to call the function.
Hello there!
Function called.
About to call the function.
Goodbye!
Function called.
```

У функции `announce()` есть один параметр – `func`. При первом вызове этой функции, в строке 15, в качестве аргумента ей передается функция `hello()`, как значение параметра `func`. Обрати внимание, что скобки в имени передаваемой функции не указываются. Скобки указывают Lua, что следует «вызвать эту функцию», но если круглых скобок нет, то вместо вызова Lua передает функцию `hello()` параметру `func` функции `announce()`. Функция `hello()` сохраняется в виде параметра `func` точно так же, как сохранялись бы целое число или строка. Затем в строке 11 вызывается функция `func()`, поскольку она имеет круглые скобки. Ну и поскольку в качестве значения параметра `func` было передано значение `hello`, вызывается

функция hello() и программа выводит сообщение Hello there!.

Затем, в строке 16, происходит повторный вызов функции announce(), но на этот раз параметру func передается значение goodbye, а значит в строке 11 вызывается функция goodbye(), и выводится сообщение Goodbye!.

Код функции announce() не изменяется. Просто он может вызывать любую функцию, переданную параметру func. Аналогично функция sweepField(), которую мы создадим, будет вызывать функцию, переданную параметру sweepFunc. Функция buildFloor() будет передавать функцию hare.selectAndPlaceDown() в качестве параметра функции sweepField(), поэтому черепашка будет размещать под собой блоки, перемещаясь по прямоугольному перекрытию.

ДОРАБОТКА МОДУЛЯ HARE

Поскольку функция sweepField() полезна для многих других программ, мы поместим ее в модуль hare, который начали разрабатывать в главе 7. Функция buildFloor(), которую мы также добавим в этот модуль, будет вызывать функцию sweepField(). В оболочке командной строки выполни команду edit hare. Перемести курсор в нижнюю часть файла и продолжи код, введя следующее:

hare

```
     ...пропуск...
134. -- sweepField() перемещается по
135. -- строкам и столбцам области спереди
136. -- и справа от черепашки, обращаясь к
137. -- параметру sweepFunc в каждой позиции
138. function sweepField(length, width, sweepFunc)
139.   local turnRightNext = true
140.
141.   for x = 1, width do
142.     for y = 1, length do
143.       sweepFunc()
144.
```

```
145.       -- не перемещаться вперед в последней строке
146.       if y ~= length then
147.         turtle.forward()
148.       end
149.     end
150.
151.     -- не поворачиваться в последнем столбце
152.     if x ~= width then
153.       -- переход к следующему столбцу
154.       if turnRightNext then
155.         turtle.turnRight()
156.         turtle.forward()
157.         turtle.turnRight()
158.       else
159.         turtle.turnLeft()
160.         turtle.forward()
161.         turtle.turnLeft()
162.       end
163.
164.       turnRightNext = not turnRightNext
165.     end
166.   end
167.
168.   -- возвращение к исходной позиции
169.   if width % 2 == 0 then
170.     turtle.turnRight()
171.   else
172.     for y = 1, length - 1 do
173.       turtle.back()
174.     end
175.     turtle.turnLeft()
176.   end
177.
178.   for x = 1, width - 1 do
179.     turtle.forward()
180.   end
181.   turtle.turnRight()
182.
183.   return true
184. end
185.
```

```
186.
187. -- buildFloor() выкладывает
188. -- прямоугольное перекрытие
189. -- из блоков в инвентаре
190. function buildFloor(length, width)
191.   if countInventory() < length * width then
192.     return false -- недостаточно блоков
193.   end
194.
195.   turtle.up()
196.   sweepField(length, width, selectAndPlaceDown)
197. end
```

Сохрани программу и выйди из редактора после ввода всех инструкций. Ты также можешь скачать этот модуль, выполнив команду `pastebin get wwzvaKuW hare`.

ВЫЗОВ ФУНКЦИИ SWEEPFUNC()

Давай рассмотрим новые функции в модуле `hare` и начнем с `sweepField()`. Эта функция перемещает черепашку по всему выкладываемому перекрытию, вызывая функцию, переданную в виде параметра. Параметры `length` и `width` указывают функции `sweepField()`, сколько строк и столбцов составляет перекрытие.

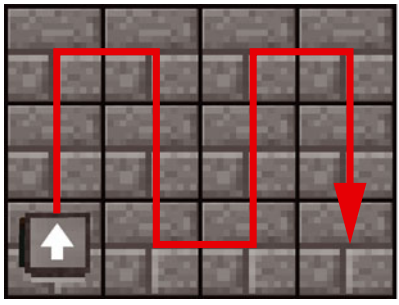
Чтобы функция `sweepField()` действовала на всем заданном пространстве, мы используем один цикл для прохода по столбцам, а другой – для прохода по строкам.

hare

```
134. -- sweepField() перемещается по горизонтальным
135. -- и вертикальным рядам спереди и справа от
     черепашки,
136. -- to the right of the turtle, calling
137. -- обращаясь к параметру sweepFunc в каждой позиции
138. function sweepField(length, width, sweepFunc)
139.   local turnRightNext = true
140.
141.   for x = 1, width do
142.     for y = 1, length do
143.       sweepFunc()
```

В конце каждого столбца черепашка чередует поворот направо и налево, поэтому код в строке 139 изначально присваивает переменной `turnRightNext` значение `true`.

Далее следуют два вложенных цикла, позволяющие отследить, в каком столбце и в какой строке находится черепашка. Переменная `x` отслеживает столбцы, а переменная `y` – строки. Когда значение переменной `y` равно значению `length`, черепашка находится в последней строке текущего столбца. Когда значение переменной `x` равно значению `width`, черепашка находится в последнем столбце текущей строки. На рис. 13.10 показаны значения переменных `x` и `y` для всех блоков, из которых черепашка строит перекрытие. Обрати внимание, что значения переменной `y` начинаются с `1` с обеих сторон, потому что черепашка чередует движение вверх и вниз при перемещении по столбцам.



x = 1 y = 3	x = 2 y = 1	x = 3 y = 3	x = 4 y = 1
x = 1 y = 2	x = 2 y = 2	x = 3 y = 2	x = 4 y = 2
x = 1 y = 1	x = 2 y = 3	x = 3 y = 1	x = 4 y = 3

Рис. **13.10.** *Путь, по которому движется черепашка, когда она выкладывает перекрытие (слева), и значения переменных `x` и `y` каждого блока (справа)*

Внутри вложенного цикла, код в строке 143 вызывает функцию `sweepFunc()`. Напомню, что `sweepFunc` – это параметр, а не имя функции. В нашей программе нет функции `sweepFunc()`, поэтому, чтобы определить функцию с таким именем, надо передать какую-либо функцию параметру `sweepFunc`. Таким образом, в `sweepField()` может быть передана любая функция, которая и будет вызываться в строке 143 как `sweepFunc()`.

ПЕРЕМЕЩЕНИЕ ПО СТРОКАМ И СТОЛБЦАМ

После вызова функции `sweepFunc()` черепашке нужно пройти к позиции следующего выкладываемого блока. Внутренний цикл `for`, начинающийся в строке 142, будет совершать итерации по переменной `y` от `1` до `length`. Черепашке нужно перемещаться вверх или вниз по `length - 1` строке в каждом столбце. Таким образом, в строках 146–148 будет происходить вызов функции `turtle.forward()` при каждой итерации цикла `for`, за исключением последней (когда значение переменной `y` равно `length`).

hare

```
145.       -- не перемещаться вперед в последней строке
146.       if y ~= length then
147.         turtle.forward()
148.       end
149.     end
```

Когда исполнение достигает строки 149, цикл `for`, который начался в строке 142, завершается, а черепашка находится в последней строке текущего столбца. Если черепашка еще не достигла последнего столбца (это так, если значение переменной `x` не равно значению `width`), ей нужно переместиться в следующий столбец.

Направление, в котором черепашка должна повернуть, чтобы перейти к следующему столбцу, зависит от того, какое значение присвоено переменной `turnRightNext`, `true` или `false`. (См. рис. 13.5 и 13.7, чтобы понять, куда должна поворачивать черепашка.) Если переменной `turnRightNext` присвоено значение `true`, в строках 155–157 происходит поворот черепашки направо, перемещение вперед и снова поворот направо. Если переменной `turnRightNext` присвоено значение `false`, в строках 159–161 черепашка поворачивает налево, продвигается вперед и опять поворачивает налево. В обоих случаях черепашка оказывается в начале следующего столбца.

hare

```
151.      -- не поворачиваться в последнем столбце
152.      if x ~= width then
153.        -- переход к следующему столбцу
154.        if turnRightNext then
155.          turtle.turnRight()
156.          turtle.forward()
157.          turtle.turnRight()
158.        else
159.          turtle.turnLeft()
160.          turtle.forward()
161.          turtle.turnLeft()
162.        end
163.
164.        turnRightNext = not turnRightNext
165.      end
166.    end
```

Всякий раз, когда черепашка переходит к следующему столбцу, она должна изменить направление поворота, поэтому код в строке 164 переключает логическое значение переменной `turnRightNext`. Инструкция `end` в строке 165 завершает конструкцию `if`, начавшуюся в строке 152, а инструкция `end` в строке `166` завершает цикл `for`, начавшийся в строке 141. Если исполнение программы достигло строки 166, значит, черепашка прошла все перекрытие и готова вернуться на исходную позицию.

ОПРЕДЕЛЕНИЕ ЧЕТНОСТИ ЧИСЛА С ПОМОЩЬЮ ОПЕРАТОРА ДЕЛЕНИЯ ПО МОДУЛЮ

Вернуть черепашку на исходную позицию, после того как она закончит выкладывать перекрытие, можно двумя путями. Напомню, что эти пути были показаны на рис. 13.8 и 13.9 и что путь, который должна совершить черепашка, зависит от количества блоков по ширине перекрытия – четное оно или нечетное. Чтобы определить, четное или нечетное данное значение, нужно проверить, делится ли оно на 2 без остатка. Если да, то число четное. Если же деление на 2 дает в остатке 1 – число нечетное.

Поэтому, чтобы определить, четное или нечетное данное значение, нужно просто найти остаток после деления на два. Мы можем это сделать, используя оператор деления по модулю (%). (Это название не связано с модулями Lua.)

Чтобы посмотреть, как работает оператор деления по модулю, в оболочке Lua введи следующее:

hare

```
lua> 6 % 2
0
lua> 7 % 2
1
lua> 8 % 2
0
```

Оператор деления по модулю используется подобно обычному оператору деления (/): укажи делимое, оператор % и делитель (в нашем случае это число 2). Четные числа при делении по модулю на 2 в остатке всегда будут давать значение 0, потому что делятся на 2 без остатка. Нечетные числа в остатке всегда будут иметь значение 1.

Используя этот трюк, мы сможем определить, четное или нечетное количество блоков использовано по ширине перекрытия.

ОБРАТНЫЙ ПУТЬ В СЛУЧАЕ ЧЕТНОГО И НЕЧЕТНОГО КОЛИЧЕСТВА БЛОКОВ ПО ШИРИНЕ

Путь, который должна проделать черепашка, чтобы вернуться в исходное положение, зависит от того, четное или нечетное количество блоков использовано по ширине перекрытия, поэтому мы будем использовать инструкцию if.

hare

```
168.    -- возвращение к исходной позиции
169.    if width % 2 == 0 then
170.      turtle.turnRight()
171.    else
172.      for y = 1, length - 1 do
```

```
173.       turtle.back()
174.     end
175.     turtle.turnLeft()
176.   end
```

Если по ширине перекрытия использовано четное количество блоков, то выражение `width % 2` будет равно `0`, а значит, условие инструкции `if` в строке 169 выполняется (`true`). В этом случае код в строке 170 поворачивает черепашку направо. В противном случае, если по ширине перекрытия использовано нечетное количество блоков, код в строках 172–174 перемещают черепашку назад на другой конец ряда, а код в строке 175 поворачивает черепашку налево. В обоих случаях черепашка оказывается в начале последнего столбца, обращенной налево.

Затем код в строках 178–180 перемещают черепашку вперед на `width - 1` блоков, чтобы вернуть черепашку в исходное положение. А код в строке 181 поворачивает ее направо, чтобы вернуть к первоначальному направлению.

hare

```
178.   for x = 1, width - 1 do
179.     turtle.forward()
180.   end
181.   turtle.turnRight()
182.
183.   return true
184. end
```

В этот момент черепашка находится в исходном положении, поэтому код в строке 183 возвращает значение `true`. Инструкция `end` в строке 184 завершает код функции `sweepField()`.

Напомню, что функция `sweepField()`написана не только для построения пола или потолка. Мы будем использовать эту функцию для достижения различных целей, потому что с ее помощью черепашка, выкладывающая перекрытие, может вызывать различные функции и применять их в каждой позиции перекрытия. А вот чтобы строить именно перекрытия, мы создадим функцию `buildFloor()`.

СОЗДАНИЕ ФУНКЦИИ BUILDFLOOR()

С помощью функции `sweepField()` функция `buildFloor()` строит прямоугольное перекрытие, беря блоки из инвентаря черепашки. Функция `buildFloor()` принимает значения переменных `length` и `width` в качестве аргументов для определения размеров выкладываемого перекрытия.

hare

```
187. -- buildFloor() выкладывает
188. -- прямоугольное перекрытие
189. -- из блоков в инвентаре
190. function buildFloor(length, width)
191.   if countInventory() < length * width then
192.     return false -- недостаточно блоков
193.   end
```

Код в строке 191 вызывает функцию `countInventory()` и сравнивает количество блоков в инвентаре черепашки с количеством блоков, необходимых для строительства перекрытия. Для перекрытия длиной `length` и шириной `width` блоков потребуется `length * width` блоков. Если у черепашки недостаточно блоков для строительства перекрытия заданных размеров, выполнение функции прекращается.

Если же блоков достаточно, код в строке 195 перемещает черепашку вверх на одну позицию, так, чтобы черепашка могла размещать под собой блоки, когда начнет выкладывать перекрытие.

hare

```
195.   turtle.up()
196.   sweepField(length, width, selectAndPlaceDown)
197. end
```

Код в строке 196 передает в функцию `sweepField()` значения длины (`length`), ширины (`width`) и функцию `selectAndPlaceDown()`. Именно вызов функции `selectAndPlaceDown()` в каждой позиции перекрытия будет выбирать блок из инвентаря черепашки и размещать его под черепашкой.

Это все, что касается функции buildFloor(). Функция короткая, потому что значительная часть работы выполняется функцией sweepField().

СОЗДАНИЕ ПРОГРАММЫ BUILDFLOOR

Мы поместили функции buildFloor() и sweepField() в модуль hare, чтобы другие программы могли вызывать эти функции. Но было бы удобно иметь программу только для построения перекрытий. Так же, как программы buildwall и buildroom из глав 11 и 12, программа buildfloor будет обрабатывать аргументы командной строки, выводить сообщение пользователю и вызывать функцию hare.buildFloor() для построения перекрытия.

В оболочке командной строки выполни команду edit buildfloor и введи следующий код:

buildfloor

```
 1. --[[Программа строительства перекрытий Эла Свейгарта
 2. выкладывает перекрытие.]]
 3.
 4. os.loadAPI('hare')
 5.
 6. -- обработка аргументов командной строки
 7. local cliArgs = {...}
 8. local length = tonumber(cliArgs[1])
 9. local width = tonumber(cliArgs[2])
10.
11. if length == nil or width == nil or cliArgs[1] ==
    '?' then
12.   print('Usage: buildwall <length> <width>')
13.   return
14. end
15.
16. hare.buildFloor(length, width)
```

Сохрани программу и выйди из редактора после ввода всех инструкций.

Поскольку программа buildfloor аналогична программам buildwall и buildroom, я не буду еще раз объяснять

код. Программа просто принимает аргументы командной строки для определения длины и ширины перекрытия, которые функция `hare.buildFloor()` передает функции `sweepField()`.

ЗАПУСК ПРОГРАММЫ BUILDFLOOR

Размести черепашку и щелкни по ней правой кнопкой мыши, чтобы открыть ее графический интерфейс. Загрузи по меньшей мере 30 каменных кирпичей (или любых других строительных блоков) в ее инвентарь. В оболочке командной строки выполни команду `buildfloor 5 6`, чтобы увидеть, как черепашка построит перекрытие пять блоков длиной и шесть блоков шириной.

После этого можешь выполнить команду `buildroom 5 6 4`, чтобы построить стены над полом. Когда эта комната будет закончена, поверни черепашку направо и снова выполни команду `buildfloor 5 6`, чтобы построить плоскую крышу для этой комнаты. Теперь у тебя есть все необходимое для создания прямоугольных комнат.

Если при запуске этой программы возникают ошибки, тщательно сравни свой код с листингом в этой книге, – возможно, ты найдешь опечатки. Если исправить программу не удается и ошибки по-прежнему возникают, удали файл, выполнив команду `delete buildfloor`, а затем загрузи программу заново, выполнив команду `pastebin get Epr9CndN buildfloor`.

СОЗДАНИЕ УЗОРЧАТОГО ПЕРЕКРЫТИЯ

Как упоминалось ранее, функция `sweepField()` очень гибка в настройке и позволяет передать параметру `sweepFunc` любую функцию. Давай напишем программу `buildcheckerboard`, которая построит клетчатое перекрытие из угольных и кварцевых блоков.

Ты можешь создать угольные блоки из девяти кусков угля, а кварцевые – из четырех кусков кварца. На рис. 13.11 показаны их рецепты. Обрати внимание, что можно

использовать только Незер-кварц из измерения Незер (см. **minecraft-ru.gamepedia.com/Кварц_Нижнего_мира**).

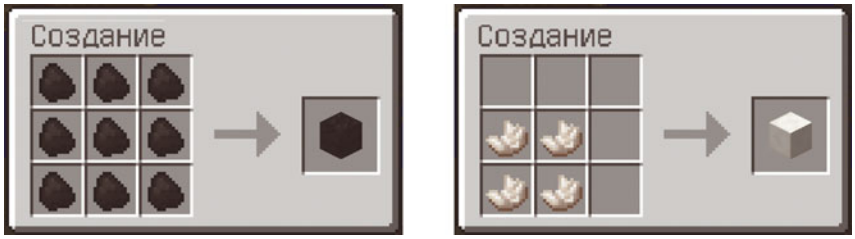


Рис. **13.11.** *Создание угольных (слева) и кварцевых блоков (справа)*

Поскольку мы делаем клетчатое перекрытие, давай сделаем его в виде шахматной доски, чтобы можно было сыграть в Minecraft в шашки с черепашками.

КОД ПРОГРАММЫ BUILDCHECKERBOARD

Чтобы сделать шахматную доску размером 8 × 8 блоков, нам нужно скрафтить 32 угольных и 32 кварцевых блока. После того, как мы создадим программу `buildcheckerboard`, мы изменим и цвета черепашек, чтобы можно было сыграть партию в шашки на доске размером 8 × 8 блоков. В оболочке командной строки выполни команду `edit buildcheckerboard` и введи следующий код:

buildcheckerboard

```
 1. --[[Программа для узорчатого строительства Эла
    Свейгарта
 2. Строит клетчатое перекрытие.]]
 3.
 4. os.loadAPI('hare')
 5.
 6. -- обработка аргументов командной строки
 7. local cliArgs = {...}
 8. local length = tonumber(cliArgs[1])
 9. local width = tonumber(cliArgs[2])
10.
11. if length == nil or width == nil or cliArgs[1] ==
```

```
    '?' then
12.   print('Usage: buildcheckerboard <length> <width>')
13.   return
14. end
15.
16. local placeBlack = true
17.
18. function placeCheckerboard()
19.   -- выбор угля или кварца в зависимости от значения
      переменной placeBlack
20.   if placeBlack then
21.     hare.selectItem('minecraft:coal_block')
22.   else
23.     hare.selectItem('minecraft:quartz_block')
24.   end
25.
26.   turtle.placeDown()
27.   placeBlack = not placeBlack
28. end
29.
30. turtle.up()
31. hare.sweepField(length, width, placeCheckerboard)
```

Сохрани программу и выйди из редактора после ввода всех инструкций.

ЗАПУСК ПРОГРАММЫ BUILDCHECKERBOARD

Помести 32 угольных и 32 кварцевых блока в инвентарь черепашки, а затем в оболочке командной строки выполни команду `buildcheckerboard 8 8`. Черепашка построит шахматное покрытие размером 8 × 8 блоков, которое изображено на рис. 13.12.

Если при запуске новой программы возникают ошибки, тщательно сравни свой код с листингом в этой книге, возможно, найдешь опечатки. Если исправить программу не удается и ошибки по-прежнему возникают, удали файл, выполнив команду `delete buildcheckerboard`, а затем загрузи программу заново, выполнив команду `pastebin get QQQK3mqk buildcheckerboard`.

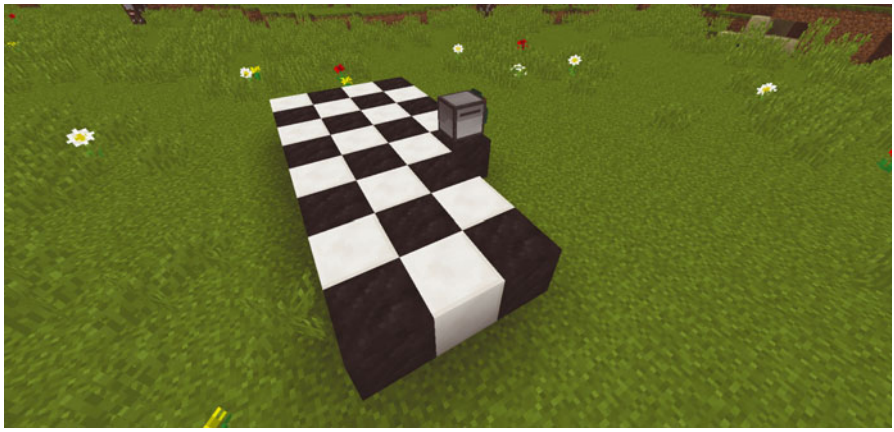


Рис. **13.12.** *Черепашка в процессе изготовления шахматной доски*

Мы написали программу `buildcheckerboard`, чтобы сделать клетчатое перекрытие в виде стандартной шахматной доски, но ты можешь создавать клетчатые потолки и полы любого размера. Они могут быть частью причудливого замка, как показано на рис. 13.13.

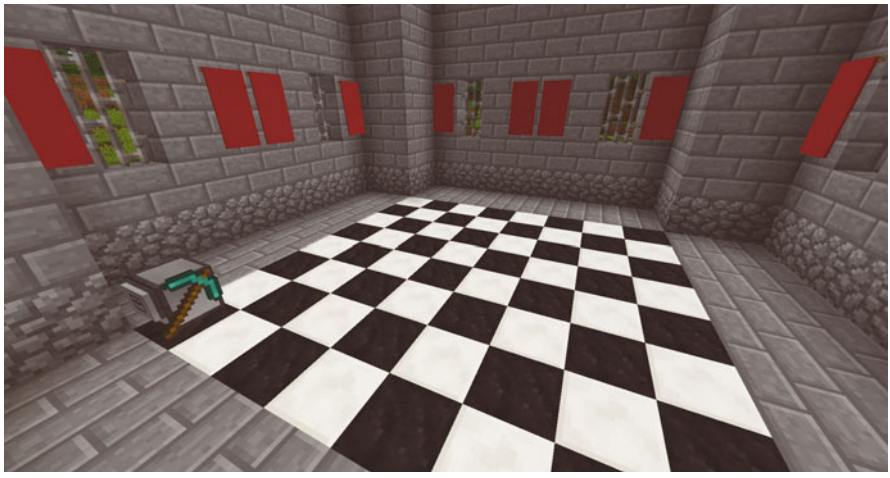


Рис. **13.13.** *Шахматный пол в замке, построенном черепашкой*

Если ты создашь 24 черепашки, а также подготовишь красный краситель и костную муку, то, щелкая правой кнопкой мыши по черепашкам, их можно покрасить в красный и белый цвета соответственно. Таким образом, ты создашь комплект для игры в шашки, как показано на рис. 13.14.

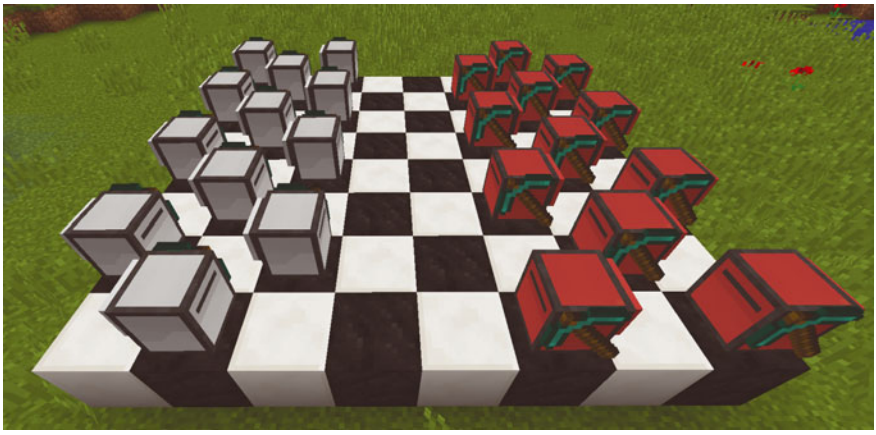


Рис. **13.14.** *Игра в черепашьи шашки*

Теперь, даже создание гигантского поля для шашек размером 32 × 32 блока, как показано на рис. 13.15, стало простой задачей, ведь черепашка сделает все за тебя.

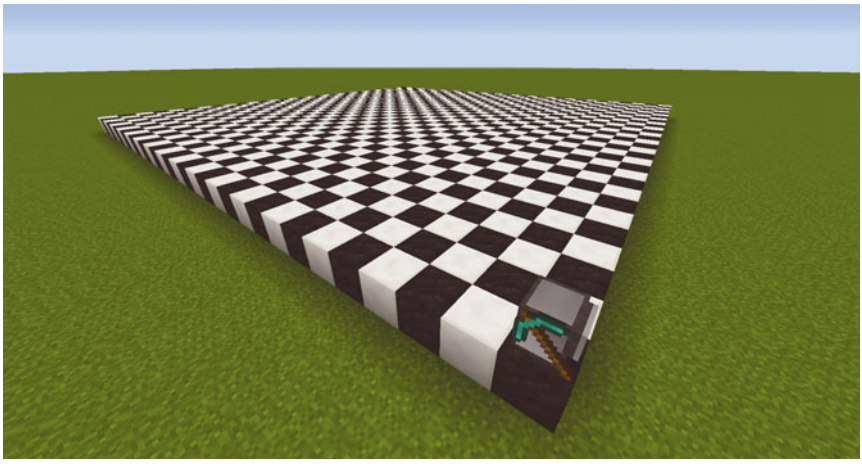


Рис. **13.15.** *Поле для шашек размером 32 × 32 блока, построенное черепашкой*

КОД ФУНКЦИИ PLACECHECKERBOARD()

В отличие от функции `hare.selectAndPlaceDown()`, которая просто выбирает любую ячейку, содержащую какие-либо блоки, и помещает один из блоков под черепашкой, функция `placeCheckerboard()` сначала проверяет

значение переменной placeBlack, чтобы определить, какой блок следует выбрать: угольный или кварцевый.

Переменная placeBlack объявляется в строке 16 с помощью инструкции local, вне каких-либо функций.

buildcheckerboard

```
16. local placeBlack = true
```

Далее функция placeCheckerboard() проверяет, истинно (true) или ложно (false) значение этой переменной, чтобы определить, какой блок выбрать.

buildcheckerboard

```
18. function placeCheckerboard()
19.   -- выбор угля или кварца в зависимости от значения
      переменной placeBlack
20.   if placeBlack then
21.     hare.selectItem('minecraft:coal_block')
22.   else
23.     hare.selectItem('minecraft:quartz_block')
24.   end
25.
26.   turtle.placeDown()
```

Если переменной placeBlack присвоено значение true, программа вызывает функцию hare.selectItem(), чтобы выбрать ячейку инвентаря, содержащую угольные блоки. Если переменной placeBlack присвоено значение false, программа выбирает ячейку инвентаря, содержащую кварцевые блоки. После того как с помощью инструкций if и else определен тип блока, который нужно выбрать с помощью функции hare.selectItem(), код в строке 26 размещает выбранный блок под черепашкой.

Затем код в строке 27 переключает логическое значение переменной placeBlack.

buildcheckerboard

```
27.   placeBlack = not placeBlack
28. end
```

Если переменной было присвоено значение `true`, то присваивается `false`. И наоборот. Это значит, что при следующем вызове функции `placeCheckerboard()` функцией `sweepField()`, будет выбран блок с противоположным цветом. Таким образом, этот код создает шахматное покрытие.

Вызов функции sweepField()

После завершения функции `placeCheckerboard()` в строке 28 исполнение переходит к строке 30. Эта строка перемещает нашего робота на одну позицию вверх, чтобы черепашка могла размещать под собой блоки.

buildcheckerboard

```
30. turtle.up()
31. hare.sweepField(length, width, placeCheckerboard)
```

Затем происходит вызов функции `hare.sweepField()`, которой передаются значения аргументов командной строки `length`, `width` и функция `placeCheckerboard`. Имя передаваемой функции записывается без круглых скобок, так как программа не вызывает функцию `placeCheckerboard()`, а передает ее в качестве значения аргумента.

Дополнительное задание: разноцветная шахматная доска

Нет необходимости использовать исключительно черный уголь и белый кварц, чтобы построить шахматную доску. Попробуй загрузить другие блоки для создания разных стилей шахматной доски: золотые и железные блоки, синюю и красную шерсть либо арбузы с тыквами. Тебе придется изменить вызовы функции `hare.selectItem()`, чтобы выбирать различные типы блоков.

Что мы узнали

В этой главе мы создали общий алгоритм для выкладывания перекрытия. По этому алгоритму черепашка про-

ходит по всем позициям прямоугольной области, выполняя в каждой позиции определенное действие. Программа `buildfloor` использует этот алгоритм для построения полов и потолков, но ты сможешь применять этот алгоритм для решения множества других задач. Это возможно, потому что Lua позволяет передавать функции другим функциям так же легко, как ты передаешь строковые или целочисленные аргументы. Например, когда функция `selectAndPlaceDown()` была передана функции `sweepField()` в программе `buildfloor`, черепашка построила перекрытие. Но когда более специализированная функция `placeCheckerboard()` была передана функции `sweepField()` в программе `buildcheckerboard`, черепашка построила шахматную доску.

Так как функция `sweepField()` настолько гибка в настройке, для программирования новых действий не требуется много дополнительного кода. В главе 14 мы будем использовать функцию `sweepField()` для создания автоматизированной фермы, на которой все операции по посадке и уборке урожая будут автоматизированы.

14

ПРОГРАММИРОВАНИЕ ЧЕРЕПАШКИ-ФЕРМЕРА

В режиме выживания необходимо заботиться о пропитании, но охота и сбор пищи отнимают много времени. Сельское хозяйство и сбор урожая – лучшее решение, но содержать крупную ферму довольно хлопотно. Давай выращивать и собирать огромный урожай с помощью одной запрограммированной черепашки-фермера, как показано на рис. 14.1.

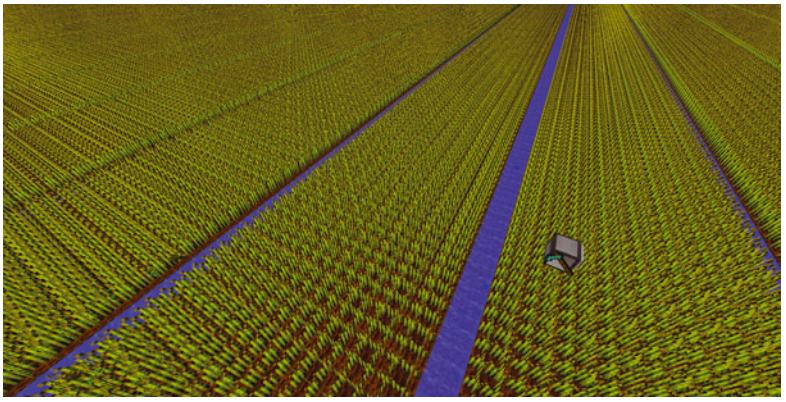


Рис. **14.1.** *На крупной пшеничной ферме может работать всего одна черепашка*

В этой главе мы создадим алгоритм, который позволит твоим черепашкам, выращивать, собирать и сдавать урожай на хранение.

ПОДГОТОВКА ПШЕНИЧНОГО ПОЛЯ

Чтобы выращивать пшеницу в Minecraft, сначала нужно подготовить поле. Для этого понадобится мотыга, ровный участок земляных или травяных блоков, ведро, доступ к источнику воды, например к океану или реке (источник воды не обязательно должен находиться рядом с участком), блоки забора и сундук.

Сначала перейди к источнику воды с пустым ведром и, с ведром в руке, щелкни правой кнопкой мыши по воде, чтобы наполнить ведро. Затем подойди к своему участку земли или травы. Обработай участок мотыгой и подготовь пашню размером 9 × 9 блоков. Поскольку пашня без близости воды быстро превращается обратно в необработанную землю, тебе надо выкопать яму в центре поля, а затем щелкнуть по ней правой кнопкой мыши с полным воды ведром в руках и вылить воду в яму. Таким образом ты заполнишь яму водой.

Одного такого блока воды достаточно для увлажнения поля на расстоянии до четырех блоков в любом направлении, то есть участка размером 9 × 9 блоков. Чтобы создать более крупную ферму, можно разместить несколько таких орошаемых полей рядом друг с другом.

По периметру фермы надо установить забор, чтобы не пускать коров, овец и враждебных мобов, которые будут мешать черепашкам. В левом нижнем углу установи сундук, на высоте одного блока над землей, в этот сундук черепашки будут складывать урожай. На рис. 14.2 показано, как должно выглядеть поле.

Примечание *Одна черепашка может обрабатывать прямоугольное поле любого размера. Самое главное, размести один блок воды для полива на каждом участке размером 9 × 9 блоков.*

Также понадобится забор по периметру всего прямоугольного поля (независимо от его размера), чтобы не пускать враждебных мобов и животных.

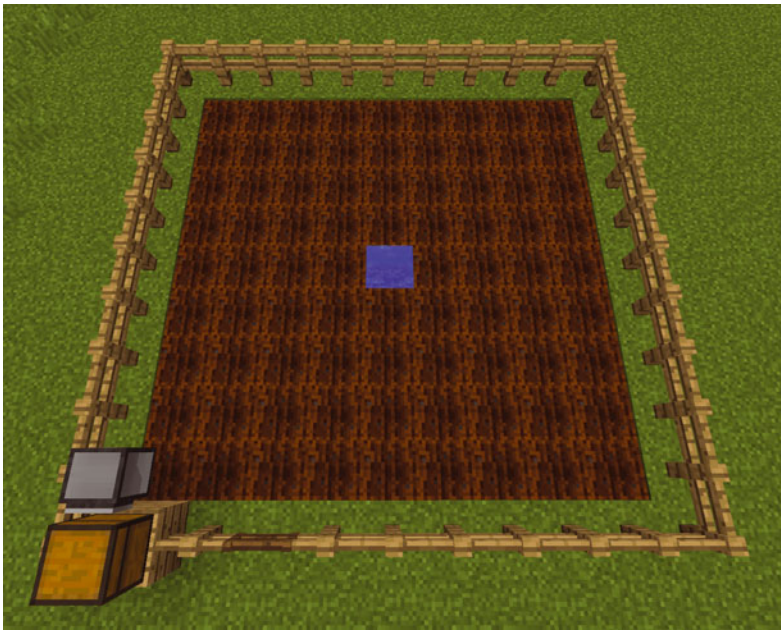


Рис. **14.2.** *Пшеничное поле размером 9 × 9 блоков с источником воды посередине. В левом нижнем углу находятся черепашка и сундук*

Теперь, когда поле подготовлено, давай рассмотрим алгоритм, на основе которого программа `farmwheat` будет управлять фермой.

РАЗРАБОТКА АЛГОРИТМА УПРАВЛЕНИЯ ПШЕНИЧНОЙ ФЕРМОЙ

В этой главе мы напишем программу `farmwheat`. Но сначала нужно спроектировать алгоритм этой программы. Прежде чем черепашка начнет заниматься полем, нужно проверить два момента. Во-первых, черепашка проверяет, что рядом с ней есть сундук. Если она не сможет его найти, ей негде будет хранить собранную пшеницу. Поскольку программа `farmwheat` будет содержать функцию `hare.sweepField()`, которую мы написали в главе 13, черепашке

также необходимо проверить, что у нее достаточно топлива, чтобы полностью обработать поле.

После того как черепашка убедится, что у нее есть доступ к сундуку и достаточное количество топлива, она вызовет функцию `hare.sweepField()` и начнет движение по всему пространству поля. Поскольку разные блоки пшеницы созревают с разной скоростью (в большинстве случаев, в течение 30–60 минут – т.е. 2–3 игровых дня Minecraft), очевидно, что не на всем поле пшеница созреет одновременно. Поэтому нам нужно создать функцию, которая будет не только вызывать функцию `hare.sweepField()`, но и определять, что черепашка должна делать на каждом конкретном блоке поля.

Черепашка должна выполнить одно из трех действий:

- посеять семена, если участок не засажен;
- не делать ничего, если пшеница посажена, но еще не созрела;
- убрать пшеницу, если она созрела, и вновь посеять семена.

После того как черепашка выполнит одно из указанных трех действий на всем пространстве поля, она вернется в исходное положение в соответствии с алгоритмом, который мы заложили в функцию `hare.sweepField()` в главе 13. В данном случае исходное положение черепашки находится рядом с сундуком. Черепашка поворачивается к сундуку, складывает в него собранную пшеницу, а затем снова поворачивается к полю. Потом черепашка отдыхает в течение 10 минут, чтобы дать посеянной пшенице время созреть, и снова обрабатывает поле.

Теперь, когда ты понимаешь алгоритм, мы можем использовать его для разработки фермерской программы. Начнем с добавления вспомогательной функции `findBlock()` в модуль `hare`. К этой функции черепашка будет обращаться, чтобы убедиться, что она находится рядом с сундуком.

ДОРАБОТКА МОДУЛЯ HARE

Функция findBlock() поворачивает черепашку вокруг своей оси, чтобы та могла рассмотреть, находится ли рядом определенный блок. Эта функция поможет нашей черепашке-фермеру найти ближайший сундук для хранения собранной пшеницы. Функция findBlock()также может пригодиться и в других программах, поэтому добавим ее в модуль hare.

В оболочке командной строки выполни команду edit hare. Переместите курсор в нижнюю часть файла и продолжи код, добавив следующие строки:

hare

```
     ...пропуск...
200. -- findBlock() поворачивает черепашку для
201. -- поиска блока с указанным идентификатором
202. function findBlock(name)
203.   local result, block
204.
205.   for i = 1, 4 do
206.     result, block = turtle.inspect()
207.     if block ~= nil and block['name'] == name then
208.       return true
209.     end
210.     turtle.turnRight()
211.   end
212.   return false
213. end
```

Сохрани программу и выйди из редактора после ввода всех инструкций. Ее также можно скачать, выполнив команду pastebin get wwzvaKuW hare.

Функция findBlock() очень проста в использовании. Ты передаешь функции findBlock() строку с идентификатором блока, который следует найти. Например, для поиска сундука, мы передадим функции идентификатор 'minecraft:chest'. Функция findBlock() может повернуть черепашку направо до четырех раз, используя цикл for в строке 205, а черепашка на каждой итерации про-

веряет идентификатор блока, расположенного перед ней, с помощью функции `turtle.inspect()`.

Функция `turtle.inspect()` возвращает одно из двух значений: логическое значение `true`, если блок находится перед черепашкой, или `false`, если блока нет. Когда есть блок для проверки, функция также возвращает данные о блоке. В строке 206 мы сохраняем логическое значение, возвращенное функцией `turtle.inspect()`, в переменной `result`, а сведения о блоке в переменной `block`. На каждой итерации цикла в строке 207 проверяется, существует ли блок и если да, то совпадает ли его идентификатор с переданным функции `findBlock()`.

Если черепашка находит искомый блок, она перестает поворачиваться и возвращает значение `true`. В противном случае черепашка продолжает вращаться, пока не совершит четыре поворота. Если после четырех поворотов черепашка не находит заданный блок, функция `findBlock()` возвращает значение `false`. Теперь, благодаря этой вспомогательной функции, мы можем написать код основной программы управления фермой – `farmwheat`.

КОД ПРОГРАММЫ FARMWHEAT

Давай напишем программу с нашим алгоритмом. В оболочке командной строки выполни команду `edit farmwheat` и введи следующий код:

farmwheat

```
 1. --[[Программа для выращивания пшеницы Эла Свейгарта
 2. Выращивает и собирает пшеницу.
 3. Предполагается, что поле находится
 4. спереди и справа от черепашки, а
 5. сзадии нее установлен сундук.]]
 6.
 7. os.loadAPI('hare')
 8.
 9. -- обработка аргументов командной строки
10. local cliArgs = {...}
11. local length = tonumber(cliArgs[1])
```

```
12. local width = tonumber(cliArgs[2])
13.
14. -- отображение информации об использовании
15. if length == nil or width == nil or cliArgs[1] == '?'
    then
16.   print('Usage: farmwheat <length> <width>')
17.   return
18. end
19.
20. print('Hold Ctrl-T to stop.')
21.
22. -- проверка доступа к сундуку
23. if not hare.findBlock('minecraft:chest') then
24.   error('Must start next to a chest.')
25. end
26.
27. -- поворот к полю
28. turtle.turnLeft()
29. turtle.turnLeft()
30.
31.
32. -- checkWheatCrop() собирает урожай
33. -- и сажает семена
34. function checkWheatCrop()
35.   local result, block = turtle.inspectDown()
36.
37.   if not result then
38.     turtle.digDown() -- подготовка пашни
39.     plantWheatSeed()
40.   elseif block ~= nil and block['name'] ==
      'minecraft:wheat' and block['metadata'] == 7 then
41.     -- сбор урожая и посев семян
42.     turtle.digDown()
43.     print('Collected wheat.')
44.     plantWheatSeed()
45.   end
46. end
47.
48.
49. -- plantWheatSeed() пробует посадить
50. -- семена пшеницы под черепашкой
51. function plantWheatSeed()
```

```
52.   if not hare.selectItem('minecraft:wheat_seeds')
      then
53.     print('Warning: Low on seeds.')
54.   else
55.     turtle.placeDown() -- посев семян
56.     print('Planted seed.')
57.   end
58. end
59.
60.
61. -- storeWheat() помещение урожая
62. -- в сундук
63. function storeWheat()
64.   -- поворот к сундуку
65.   if not hare.findBlock('minecraft:chest') then
66.     error('Could not find chest.')
67.   end
68.
69.   -- помещение пшеницы в сундук
70.   while hare.selectItem('minecraft:wheat') do
71.     print('Dropping off ' .. turtle.getItemCount() ..
        ' wheat...')
72.     if not turtle.drop() then
73.       error('Wheat chest is full!')
74.     end
75.   end
76.
77.   -- поворот к полю снова
78.   turtle.turnLeft()
79.   turtle.turnLeft()
80. end
81.
82.
83. -- начало работы на ферме
84. while true do
85.     -- проверка уровня топлива
86.   if turtle.getFuelLevel() < (length * width + length
      + width) then
87.     error('Turtle needs more fuel!')
88.   end
89.
```

```
90.    -- посев пшеницы
91.    print('Sweeping field...')
92.    hare.sweepField(length, width, checkWheatCrop)
93.    storeWheat()
94.
95.    print('Sleeping for 10 minutes...')
96.    os.sleep(600)
97. end
```

После ввода всех этих инструкций сохрани программу и выйди из редактора.

ЗАПУСК ПРОГРАММЫ FARMWHEAT

Перед запуском программы `farmwheat` следует убедиться, что расстояние до источника воды составляет не более четырех блоков и что сундук находится за черепашкой в ее исходной позиции, в левом нижнем углу поля, как показано на рис. 14.3. Черепашку надо оборудовать алмазной мотыгой, чтобы она могла превращать в пашню землю или траву. После того как черепашка настроена, в оболочке командной строки введи команду `farmwheat 9 9`, чтобы черепашка начала обрабатывать поле размером 9 × 9 блоков.

Рис. 14.3. Черепашка размещена на высоте одного блока над землей, а сундук установлен позади нее

Имей в виду, что если ты уйдешь от черепашки слишком далеко, она перестанет работать. Если черепашка отключится на середине поля ее нужно будет переместить обратно в левый нижний угол поля, и снова запустить программу, чтобы черепашка продолжила заниматься фермой. Наш робот продолжит, как и раньше, перемещаться по полю, проверять зрелость пшеницы, собирать ее и производить посев.

Если при запуске этой программы возникают ошибки, тщательно сравни свой код с листингом в этой книге, чтобы найти возможные опечатки. Если исправить программу не удается, удали файл, выполнив команду `delete farmwheat`, а затем загрузи программу снова, выполнив команду `pastebin get SfcB8b55 farmwheat`.

КОНФИГУРАЦИЯ ПРОГРАММЫ FARMWHEAT

Первые пять строк программы содержат обычные комментарии, описывающие, что она делает и кто разработал программу. Код в строке 7 загружает модуль `hare`, чтобы программа могла вызывать его функции.

Затем программа считывает аргументы командной строки для получения длины и ширины поля. Если аргументы командной строки не указаны, программа выводит сообщение пользователю.

farmwheat

```
 9. -- обработка аргументов командной строки
10. local cliArgs = {...}
11. local length = tonumber(cliArgs[1])
12. local width = tonumber(cliArgs[2])
13.
14. -- отображение информации об использовании
15. if length == nil or width == nil or cliArgs[1] ==
    '?' then
16.   print('Usage: farmwheat <length> <width>')
17.   return
18. end
```

Поскольку аргументы командной строки – это всегда строковые значения, а переменные `length` и `width` – всегда числа, код в строках 11 и 12 передает первый и второй аргументы командной строки функции `tonumber()`. Возвращаемые значения сохраняются в качестве значений переменных `length` и `width`.

Если отсутствует первый или второй аргумент командной строки, то функция `tonumber()` вернет значение `nil`, которое будет присвоено переменной `length` или `width`, соответственно. Если отсутствует второй аргумент командной строки или если первый аргумент командной строки представляет собой значение `'?'`, условие инструкции `if` в строке 15 принимает значение `true`, и выполняется код в строках 16 и 17. Код в строке 16 отображает сообщение пользователю, а инструкция `end` в строке 17 завершает работу программы.

Затем программа выполняет дополнительные шаги по подготовке, проверяя, что у черепашки есть все необходимое для запуска программы `farmwheat`.

farmwheat

```
20. print('Hold Ctrl-T to stop.')
21.
22. -- проверка доступа к сундуку
23. if not hare.findBlock('minecraft:chest') then
24.   error('Must start next to a chest.')
25. end
26.
27. -- поворот к полю
28. turtle.turnLeft()
29. turtle.turnLeft()
```

Код в строке 20 выводит сообщение пользователю о том, что он может завершить программу, удерживая сочетание клавиш **Ctrl+T**. Затем черепашка пытается найти сундук, вызывая функцию `hare.findBlock('minecraft:chest')`. Если она не находит сундук, код в строке 24 завершает программу сообщением об ошибке, потому что нет необходимости продолжать программу, если нет сундука для хранения пшеницы.

Поскольку сундук находится за черепашкой и пшеничным полем, черепашке нужно дважды повернуть налево (строки 28 и 29), чтобы снова оказаться повернутой лицевой стороной к полю.

КОД ФУНКЦИЙ, ИСПОЛЬЗУЕМЫХ В ОСНОВНОЙ ПРОГРАММЕ

Давай напишем функцию с именем `checkWheatCrop()`, которая сообщает черепашке, что делать на каждом участке поля, которое она обрабатывает. Затем напишем еще одну функцию, `plantWheatSeed()`, которая содержит инструкции по выращиванию семян пшеницы. Мы напишем еще и третью функцию, с именем `storeWheat()`, которая проинструктирует черепашку о том, что она должна делать после сбора урожая. Эти три функции будут вызываться из основной части программы `farmwheat`.

ОТСЛЕЖИВАНИЕ УРОЖАЯ

Функция `checkWheatCrop()` проверяет пространство под черепашкой и определяет, какое действие должна произвести черепашка, находясь в данной позиции. Мы передадим функцию `checkWheatCrop()` функции `sweepField()`, чтобы она вызывалась в каждой позиции поля, которое обрабатывает черепашка.

Показанный ниже код инструктирует черепашку посадить семена, если под черепашкой нет пшеницы.

farmwheat

```
32. -- checkWheatCrop() собирает урожай
33. -- и сажает семена
34. function checkWheatCrop()
35.   local result, block = turtle.inspectDown()
36.
37.   if not result then
38.     turtle.digDown() -- подготовка пашни
39.     plantWheatSeed()
```

Функция `turtle.inspectDown()` осуществляет непосредственную проверку блока под черепашкой. Если там нет блока, переменной `result` в строке 35 присваивается значение `false`, а код в строке 38 обрабатывает блок земли (или травы). Код в строке 39 вызывает функцию `plantWheatSeed()`, которую мы создадим в следующем разделе. Функция `plantWheatSeed()` сеет семена пшеницы на блоке пашни под черепашкой.

Если пространство под черепашкой не пусто (то есть содержит пшеницу или семена), черепашка проверяет, содержит ли пространство зрелую пшеницу.

farmwheat

```
40.   elseif block ~= nil and block['name'] ==
      'minecraft:wheat' and           block['metadata']
       == 7 then
41.     -- сбор урожая и посев семян
42.     turtle.digDown()
43.     print('Collected wheat.')
44.     plantWheatSeed()
45.   end
46. end
```

Код в строке 40 сначала проверяет, присвоено ли переменной `block` значение `nil`, а затем проверяет, совпадает ли значение `block['name']` с идентификатором блока пшеницы. Если под черепашкой ничего нет, функция `turtle.inspectDown()` в строке 35 присваивает переменной `block` значение `nil`. Это могло бы привести к тому, что `block['name']` в строке 40 вызовет ошибку, потому что переменная `block` будет иметь значение `nil` вместо допустимого табличного значения. Для предотвращения этой ошибки код в строке 40 сначала проверяет, что переменной `block` не присвоено значение `nil`.

Табличное значение переменной `block` также содержит ключ `'metadata'`, значение которого указывает, как долго росла пшеница. Если семена пшеницы были посажены только что, ключ `'metadata'` имеет значение `0`. Если же он

имеет значение 7, пшеница созрела, и последнее условие в строке 40 выполняется (true).

В целом код в строке 40 проверяет соблюдение трех разных условий: находится ли под черепашкой какой-нибудь блок, является ли он блоком пшеницы и созрел ли этот блок пшеницы. Если какое-либо из этих условий возвращает значение false, исполнение переходит к строке 46, которая завершает блок кода, и черепашка ничего не делает.

Если в строке 40, соблюдены (true) все условия, код в строке 42 собирает пшеницу, вызывая функцию turtle.digDown(). Независимо от того, добывают ли черепашки руду, рубят дерево или собирают урожай пшеницы, функции turtle.dig() просто собирают блоки под черепашкой, с помощью инструмента из ее инвентаря. А вот код в строке 44 вызывает функцию plantWheatSeed(), которая не только собирает пшеницу, но и засеивает на ее месте новую. Далее мы рассмотрим эту функцию.

ПОСЕВ ПШЕНИЦЫ

Поскольку в строках 39 и 44 требуется один и тот же код, разумно поместить его в отдельную функцию plantWheatSeed(), тогда не придется набирать его дважды. Вообще, при написании программ на языке Lua код будет более читабельными, если использовать функции для избавления от дублирующего кода.

Первая задача, которую выполняет функция plantWheatSeed()в строке 51, – это выбор ячейки инвентаря черепашки, которая содержит семена пшеницы. Эта задача решается передачей в функцию hare.selectItem() значения 'minecraft:wheat _ seeds'.

farmwheat

```
49. -- plantWheatSeed() пробует посадить
50. -- семена пшеницы под черепашкой
51. function plantWheatSeed()
52.   if not hare.selectItem('minecraft:wheat_seeds')
      then
53.     print('Warning: Low on seeds.')
```

Если семян нет, черепашка выводит предупреждающее сообщение, посев не производит и продолжает работу в соответствии с остальной частью программы. Завершение программы в этом случае не требуется, потому что черепашка получит семена, собрав зрелую пшеницу. Но если в строке 52 функция `selectItem()` возвращает значение `true`, семена будут взяты из инвентаря черепашки.

Блок кода, который следует за инструкцией `else` в строке 54, засевает семена и отображает игроку сообщение об этом.

farmwheat

```
54.   else
55.     turtle.placeDown() -- посев семян
56.     print('Planted seed.')
57.   end
58. end
```

В строке 55 функция `turtle.placeDown()` засевает семена, если они обнаружены в текущей ячейке инвентаря, а под черепашкой находится пашня. Затем код в строке 56 выводит сообщение о том, что семена посажены, а код в строке 58 завершает блок кода функции `plantWheatSeed()`.

ХРАНЕНИЕ ПШЕНИЦЫ

После того, как черепашка обработает все поле, ей нужно поместить собранную пшеницу в сундук. Для этого программа вызывает функцию `storeWheat()`. Эта функция инструктирует черепашку найти сундук и повернуться к нему лицевой стороной.

farmwheat

```
61. -- storeWheat() помещение урожая
62. -- в сундук
63. function storeWheat()
64.   -- поворот к сундуку
65.   if not hare.findBlock('minecraft:chest') then
66.     error('Could not find chest.')
67.   end
```

Код в строке 65 вызывает функцию `hare.findBlock()` и передает ей значение `'minecraft:chest'`, из-за чего черепашка начинает вращение вокруг своей оси и останавливается, когда обнаруживает сундук. Если черепашка не может найти сундук, вызов функции `hare.findBlock()` возвращает значение `false`, а код в строке 66 завершает программу с сообщением об ошибке.

Если сундук обнаружен, выполнение программы продолжается. При этом черепашка оказывается обращенной к сундуку. Далее она выбирает ячейку инвентаря, содержащую пшеницу, и перемещает ее содержимое в сундук, вызывая функцию `turtle.drop()`.

farmwheat

```
69.   -- помещение пшеницы в сундук
70.   while hare.selectItem('minecraft:wheat') do
71.     print('Dropping off ' .. turtle.getItemCount() ..
       ' wheat...')
72.     if not turtle.drop() then
73.       error('Wheat chest is full!')
74.     end
75.   end
```

В строке 70 начинается цикл `while`, условие которого проверяет значение, возвращаемое вызовом функции `hare.selectItem('minecraft:wheat')`. Пока пшеница находится в инвентаре черепашки, эта функции возвращает значение `true`, и цикл `while` выполняется. Код в строке 71 внутри этого цикла сообщает игроку количество пшеницы в текущей ячейке, а код в строке 72 пытается поместить эту пшеницу в сундук перед черепашкой.

Если сундук уже заполнен, вызов функции `turtle.drop()` в строке 72 возвращает значение `false`. В этом случае код в строке 73 завершает программу с сообщением об ошибке. В противном случае цикл выполняется, пока пшеница еще есть, а затем исполнение переходит к строке 78.

Когда черепашка взаимодействует с сундуком, она повернута спиной к полю. Чтобы снова оказаться в поле, ей нужно развернуться.

farmwheat

```
77.    -- поворот к полю снова
78.    turtle.turnLeft()
79.    turtle.turnLeft()
80. end
```

Код в строках 78 и 79 дважды поворачивает черепашку налево, чтобы она снова «смотрела» на поле. Код в строке 80 завершает функцию `storeWheat()`.

Теперь у нас есть все три функции, которые будут вызываться из основной программы: `checkWheatCrop()`, `plantWheatSeed()` и `storeWheat()`. Давай займемся сельским хозяйством!

РАБОТА В ЦИКЛЕ

Рассмотрим основной цикл программы (в строках 32–80), который будет вызывать все необходимые функции. В этом цикле черепашка сначала проверяет, что у нее достаточно топлива. Если для обработки поля топлива достаточно, наш робот начинает выращивать и собирать урожай, сохранять собранную пшеницу в сундуке, потом отдыхать 10 минут, чтобы пшеница успела созреть, и повторять весь процесс с начала.

На первом шаге цикла черепашка проверяет, что у нее достаточно топлива для перемещения по всему полю и возвращения в исходное положение.

farmwheat

```
83. -- начало работы на ферме
84. while true do
85.      -- проверка уровня топлива
86.    if turtle.getFuelLevel() < (length * width +
       length + width) then
87.      error('Turtle needs more fuel!')
88.    end
```

Условие цикла `while` в строке 84 всегда истинно. То есть это бесконечный цикл, поэтому программа завершается, только если вызывается функция `error()` или игрок удерживает сочетание клавиш **Ctrl+T**.

Чтобы выполнить первый шаг, нам нужно составить уравнение для расчета количества топлива, которое потребуется черепашке, чтобы обработать все поле. Для упрощения расчета мы несколько переоценим потребности черепашки в топливе.

Во-первых, определим, сколько топлива требуется черепашке для перемещения по каждому столбцу поля. Она использует одну единицу топлива при каждом перемещении на один блок, таким образом, для перемещения по столбцу, длина которого составляет `length` блоков, потребуется совершить `length - 1` шагов, и, соответственно, потратить `length - 1` единиц топлива. Но черепашка использует топливо и при перемещении из одного столбца в другой, тогда общее количество, необходимое для передвижения по всему столбцу, составит `length - 1 + 1` единиц.

Затем полученное число умножаем на количество строк в поле. Таким образом, общее количество перемещений, и, соответственно единиц топлива, необходимых для обработки поля, составит `length * width`.

Кроме того, черепашке необходимо топливо, чтобы вернуться в исходное положение. Если черепашка завершает работу в дальнем от исходного положения конце поля, ей нужно переместиться вниз вдоль одного столбца (количество шагов в длину) и назад вдоль одной строки (количество шагов в ширину). Тогда формула для расчета необходимого количества топлива, приобретает вид `length * width + length + width`. На рис. 14.4 показана диаграмма этой формулы.

Эта формула рассчитана на крайний случай, потому что черепашка может закончить работу не в самой отдаленной точке поля, и тогда, чтобы вернуться в исходное положение, ей будет достаточно пройти назад вдоль одной строки. Кроме того, путь по красной стрелке вдоль крайнего левого столбца, составит `length - 1` блоков с учетом начальной позиции, да и синяя стрелка должна быть длиной `length - 1` блоков для учета углового блока. Но эта формула вполне приемлема, потому что лучше переоценить, чем недооценить количество необходимого топлива.

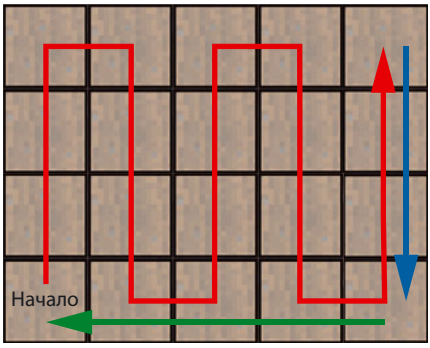


Рис. **14.4.** *Диаграмма, показывающая каждую часть операции* `length * width + length + width`

Если у черепашки недостаточно топлива, условие в строке 86 возвращает значение `false`, а код в строке 87 завершает программу, вызывая функцию `error()`, и выводит сообщение, оповещающее игрока, что черепашке требуется больше топлива. Если же условие в строке 86 возвращает значение `true`, черепашка может начать работу, вызвав функцию `hare.sweepField()` в строке 92.

farmwheat

```
90.    -- посев пшеницы
91.    print('Sweeping field...')
92.    hare.sweepField(length, width, checkWheatCrop)
93.    storeWheat()
```

Как и функции из предыдущих глав, функция `hare.sweepField()` принимает значения длины и ширины поля, вводимые пользователем, и управляет перемещением черепашки по всему пространству поля. Код в строке 92 передает функцию `checkWheatCrop()` функции `sweepField()`.

Не забывай, что при указании имени функции `checkWheatCrop()` в качестве параметра, не надо указывать круглые скобки, иначе интерпретатор Lua просто вызовет эту функцию и передаст функции `hare.sweepField()` значение, возвращаемое `checkWheatCrop()`, а не саму функцию.

Функция `hare.sweepField()` возвращает черепашку в исходную позицию после сбора урожая со всего поля. После того как черепашка закончит обрабатывать поле, ей необходимо переместить собранную пшеницу из своего инвентаря в сундук, который находится рядом с исходной позицией. Для этого мы и написали функцию `storeWheat()`, которую вызываем в строке 93.

Мы хотим, чтобы черепашка сажала и собирала урожай непрерывно, но посеянной пшенице требуется время для созревания. Поэтому, если черепашка немедленно начнет обрабатывать поле снова, пшеница не успеет вырасти, и черепашка лишь напрасно потратит топливо. Чтобы этого избежать, в строке 95 мы делаем 10-минутную паузу и сообщаем об этом игроку, выводя сообщение.

farmwheat

```
95.   print('Sleeping for 10 minutes...')
96.   os.sleep(600)
97. end
```

Код в строке 96 вызывает функцию os.sleep(), передавая в качестве параметра численное значение 600, чтобы черепашка остановилась на 600 секунд (т.е. 10 минут). В строке 97 инструкция end завершает цикл while, который начался в строке 84.

ДОПОЛНИТЕЛЬНОЕ ЗАДАНИЕ: ГИГАНТСКИЕ ПШЕНИЧНЫЕ ПОЛЯ

В этой главе мы создали ферму размером 9×9 блоков, но программа `farmwheat` позволяет обрабатывать пространства любого размера. Создай гигантское пшеничное поле, как показано на рис. 14.1. Затем запусти программу с аргументами командной строки, соответствующими размеру поля, чтобы черепашка обработала его!

СОВЕТЫ ПО АВТОМАТИЗАЦИИ ДРУГИХ ВИДОВ ЗЕМЛЕДЕЛИЯ

Значительная часть кода программы `farmwheat` находится в функции `hare.sweepField()`. Но, поскольку ты можешь любую функцию передать функции `hare.sweepField()`, она может выполнять множество других задач.

Если ты захочешь создать другие типы ферм с помощью мода ComputerCraft, ты можешь писать и передавать функции, которые будут использовать код `hare.sweepField()` на фермах с другими растениями и даже животными. В следующих подразделах я дам несколько советов по разработке программ для ферм различного типа.

ОВОЩНЫЕ ПЛАНТАЦИИ

Пшеницу можно легко заменить овощами, просто изменив несколько значений и имен в имеющемся коде. Например, функции `checkWheatCrop()` и `plantWheatSeed()` можно переименовать в `checkVegCrop()` и `plantVeg()`. Для картофеля и моркови в игре Minecraft не используются семена. Картофель и морковь можно сажать непосредственно в пашню, используя функцию `turtle.placeDown()` при уборке созревшего урожая. На рис. 14.5 показана черепашка, выращивающая морковь.

Тебе также потребуется заменить идентификаторы блоков Minecraft, используемые программой. Например, вместо вызова `hare.selectItem('minecraft:wheat _ seeds')` программа овощеводства должна использовать код `hare.selectItem('minecraft:potato')` для выращивания картофеля и `hare.selectItem('minecraft:carrot')` для выращивания моркови.

Черепашка не должна перекладывать весь урожай овощей в сундук, потому что ей понадобится посадочный материал, когда она вновь начнет обрабатывать поле.

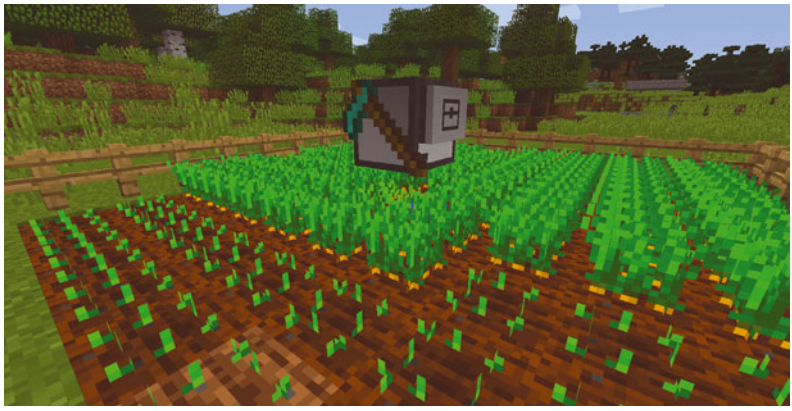


Рис. **14.5.** *Черепашка, запрограммированная на выращивание моркови*

ДОЙКА КОРОВ И СТРИЖКА ОВЕЦ

С пустым ведром черепашка может доить коров. Также наш робот может стричь шерсть с овец, если в его инвентаре есть ножницы. Оснащать черепашку ножницами или ведром, как инструментами, не нужно, достаточно просто поместить их в текущую ячейку инвентаря черепашки. Чтобы черепашка могла работать, ее надо приподнять над коровами и овцами, на высоту двух блоков над уровнем земли (в отличие от других ферм, где черепашка поднимается на один блок). На рис. 14.6 показана черепашка, поднятая над землей на высоту двух блоков, чтобы она могла взаимодействовать с коровами и овцами.

Рис. **14.6.** *Чтобы доить коров или стричь овец, черепашка должна находиться на высоте двух блоков над уровнем земли*

Чтобы подоить корову, в текущей ячейке инвентаря черепашки должно находиться пустое ведро. Когда черепашка находится над коровой, вызови функцию `turtle.placeDown()`, и она заполнит ведро молоком. Для стрижки овец в текущей ячейке должны быть ножницы. Когда черепашка будет находиться над овцой, вызови функцию `turtle.placeDown()`, и она постригет овцу. Состриженную шерсть черепашка соберет автоматически.

Коровы и овцы будут разбредаться, но ты сможешь удержать их в пределах фермы, построив по периметру забор. Возможно, животные будут пытаться убежать от черепашки, но им вряд ли это удастся, так как черепашка проворнее. Кроме того, поскольку коровы и овцы не являются блоками, вызов функции `turtle.inspectDown()` не сработает. Вместо этого черепашке придется вызывать функцию `turtle.placeDown()` вслепую и наудачу использовать пустое ведро или ножницы из текущей ячейки.

ПТИЦЕФАБРИКА

Куры производят яйца каждые пять-десять минут, но, как и все дропы, если их не поднять, яйца исчезают через пять минут. Можно запрограммировать черепашек так, чтобы они обрабатывали поле, где пасутся куры, и собирали яйца с земли.

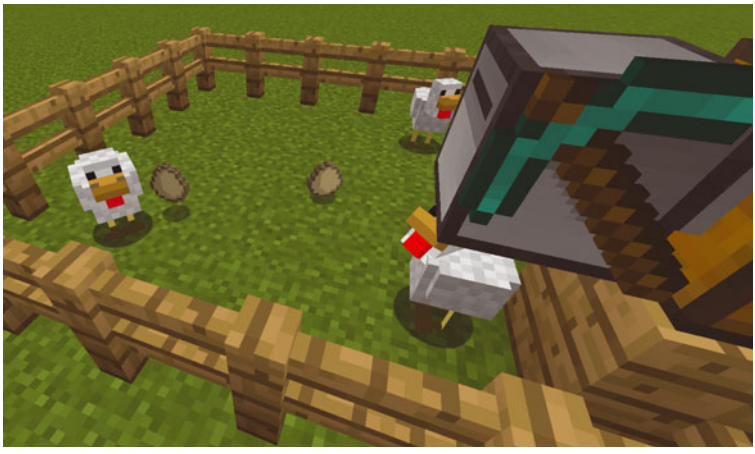


Рис. **14.7.** *Черепашка собирает яйца, снесенные курами*

Для создания птицефермы загони кур на огороженное прямоугольное поле, обслуживаемое черепашкой. Затем передай функцию `turtle.suckDown()` (без круглых скобок) функции `hare.sweepField()`, чтобы черепашка забирала любые предметы, которые валяются на земле. На рис. 14.7 изображена черепашка, собирающая яйца. Подобно животным, яйца не являются блоками, поэтому функция `turtle.inspectDown()` не сможет их идентифицировать.

ВЫРАЩИВАНИЕ КАКТУСОВ И САХАРНОГО ТРОСТНИКА

Кактусы и сахарный тростник требуют особых условий для выращивания. Например, нужен как минимум, один блок свободного пространства, разделяющий растения, чтобы они могли свободно расти. Поэтому потребуется изменить код. На ферме для выращивания кактусов надо будет оставлять пустые горизонтальные и вертикальные полосы земли.

Кроме того, кактусы могут расти только на песчаных блоках, как показано на рис. 14.8.

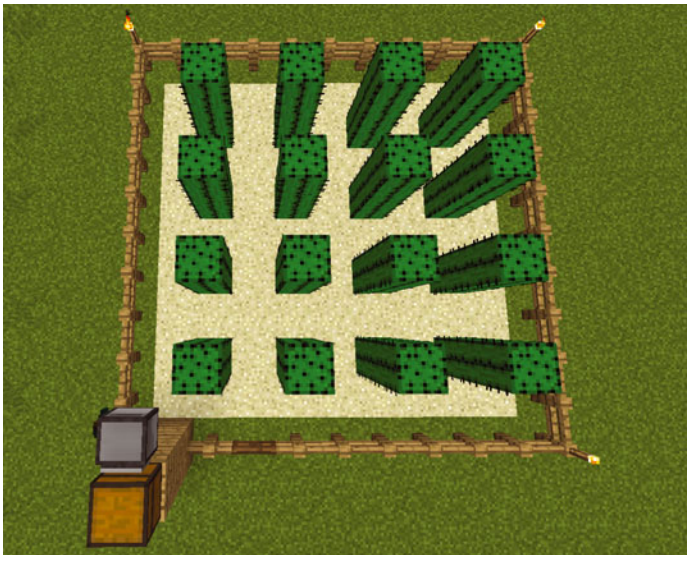


Рис. **14.8.** *Кактусы растут только на песке и должны располагаться на расстоянии одного блока между друг другом*

Сахарный тростник растет только на песчаных или земляных блоках, которые прилегают к воде, поэтому каждый ряд между рядами тростника должен состоять из водяных блоков, как показано на рис. 14.9.

Рис. **14.9.** *Черепашка выращивает сахарный тростник. Обрати внимание, что сахарный тростник следует сажать рядом с водой*

Обе культуры вырастают на высоту до трех блоков. Поэтому черепашку нужно поднять на высоту четырех блоков, чтобы она могла вызывать функцию `turtle.digDown()` для сбора урожая, выросшего до максимальной высоты. Идентификаторы Minecraft для кактуса и сахарного тростника – это `'minecraft:cactus'` и `'minecraft:reeds'`[9], соответственно.

ЧТО МЫ УЗНАЛИ

Многократно используя код `hare.sweepField()`, ты можешь легко программировать черепашек на автоматический сбор различных культур. В этой главе ты узнал, как использовать черепашек для обработки земли, посева семян и сбора зрелой пшеницы. Вызов функции `turtle.inspectDown()` возвращает табличное значение, содержащее ключ `'metadata'` со значением в диапазоне от `0` до `7`. Если значение равно `7`, значит, пшеница полностью созрела. Че-

[9] В версии 1.13 – 'minecraft:sugar_cane' *(прим. перев.)*

репашка с алмазной мотыгой использует это значение, чтобы определить, когда пшеница готова к уборке, собирает ее и производит посев семян.

Алгоритм функции `hare.sweepField()` был использован при решении множества различных задач. В главе 15 мы разработаем новый алгоритм, который позволит копать ямы в форме лестницы, уходящей в землю.

15

ПРОГРАММИРОВАНИЕ ЧЕРЕПАШКИ-ШАХТЕРА

Теперь черепашки выполняют значительную часть необходимых тебе работ: от рубки деревьев и изготовления каменных кирпичей до возделывания сельскохозяйственных угодий. Но еще есть ресурсы, которые необходимо добывать из-под земли, разрабатывая рудники. Это железо, золото, алмазы, уголь, редстоун и лазурит.

Черепашки могут и это! У них в памяти есть предустановленная программа под названием `excavate`, которую ты можешь использовать, чтобы прокопать квадратное отверстие прямо к основанию мира Minecraft. Однако, как показано на рис. 15.1, можно легко упасть в эти глубокие колодцы, что делает их опасными.

В этой главе мы напишем программу добычи полезных ископаемых под названием `stairminer`, которая выкапывает колодец с лестницей, уходящей под землю, как показано на рис. 15.2. Около поверхности черепашка будет добывать в основном землю и булыжник, но по мере разра-

ботки рудника вглубь, сможет находить и добывать разную руду и алмазы.

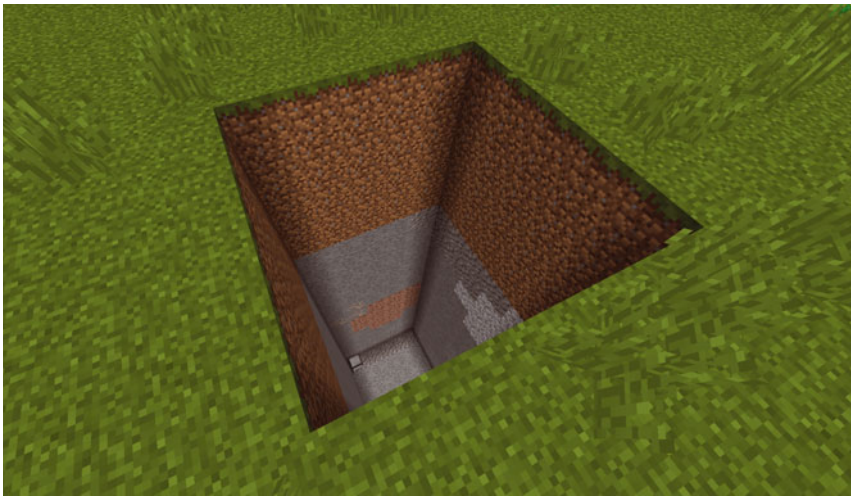


Рис. **15.1.** *Программа* `excavate` *создает глубокие, опасные колодцы*

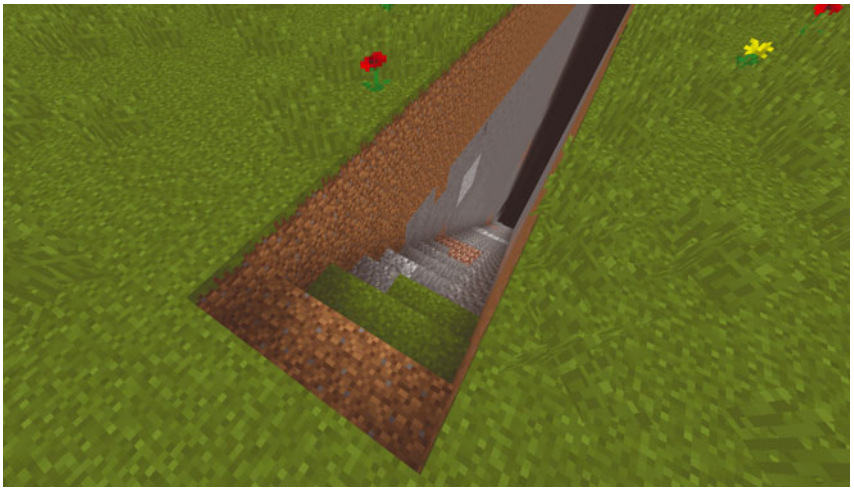


Рис. **15.2.** *Программа* `stairminer` *создает лестницу для безопасной добычи полезных ископаемых*

После того, как мы создадим программу `stairminer`, можно будет добывать руду, не опасаясь упасть в пропасть! Эти лестницы также позволят безопасно спускаться на дно шахт, чтобы, например, прокопать горизонтальные туннели или построить подземную базу.

РАЗРАБОТКА АЛГОРИТМА СОЗДАНИЯ ШАХТЫ С ЛЕСТНИЦЕЙ

Начнем с разработки алгоритма создания шахты с лестницей. Вместо того, чтобы просто копать вниз до бедрока, программа `stairminer` вырубает узор в форме лестницы. Таким образом черепашка не наделает смертельно опасных дыр, в которые можно упасть. Разработка грунта начинается с поверхности земли, как показано на рис. 15.3.

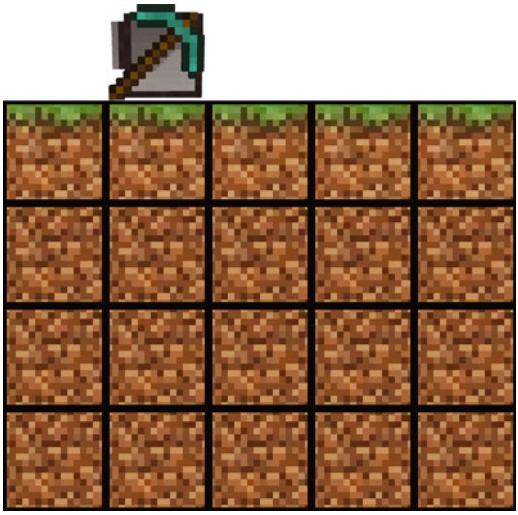


Рис. **15.3.** *Черепашка на поверхности перед началом добычи полезных ископаемых*

Нам нужно, чтобы черепашка прокопала на один блок вниз, затем переместилась на один блок вперед, прокопала на блока вниз, затем опять передвинулась на блок вперед и выкопала уже три блока и т. д. Черепашка должна продолжать копать до тех пор, пока не достигнет бедрока или глубины, указанной игроком.

Для разработки эффективной программы нужно определить конкретные действия, которые должна совершить черепашка, чтобы реализовать описанный алгоритм поведения. Каждый раз, после того как черепашка спускается вниз, она должна возвращаться на поверхность, прежде чем

копать снова. Поскольку каждый ход использует топливо, то количество ходов, совершаемых черепашкой, хотелось бы сделать минимальным. Мы сделаем это, запрограммировав черепашку так, чтобы она копала вверх, когда будет выбираться на поверхность. В результате черепашка будет использовать топливо настолько эффективно, насколько это вообще возможно, поскольку будет не только выбираться на поверхность, но и разрабатывать грунт над собой.

Давай рассмотрим действия, на которых построен алгоритм создания шахты с лестницей, и выясним, как можно минимизировать расход топлива, используемого черепашкой на каждом этапе.

Как показано на рис. 15.4, черепашка начинает разработку с блока, отмеченного символом ×. Таким образом она создает первую ступень лестницы. Поскольку черепашка не двигается на этом этапе, она не использует топливо. Первый шаг лестницы нужно выкопать один раз в начале программы, поэтому алгоритм больше не повторяет этот шаг.

Рис. **15.4.** *Черепашка начинает копать под собой*

Затем черепашка перемещается вперед на один блок к следующему блоку, от которого будет рыть колодец вниз, как показано на рис. 15.5. Это действие она будет повторять каждый раз, когда будет начинать копать новый колодец с поверхности земли.

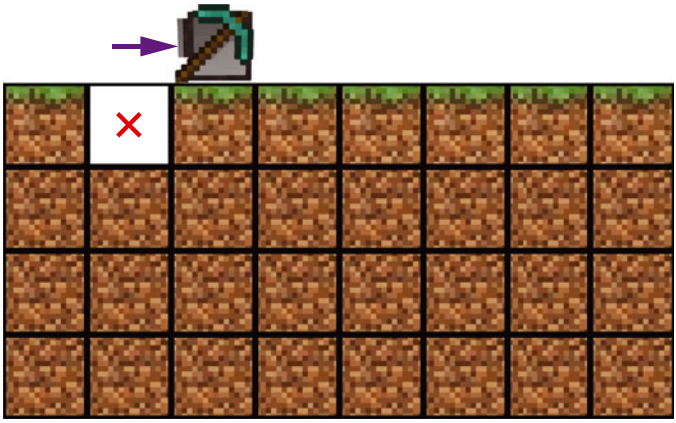


Рис. **15.5.** *После выкапывания первого блока черепашка движется вперед на один блок, чтобы начать копать следующий колодец вниз, уже на два блока*

Когда черепашка начинает разработку следующего колодца, она раскапывает на один блок больше, чем на предыдущем шаге, и опускается на дно колодца, который разрабатывает. На рис. 15.6 показан этот процесс.

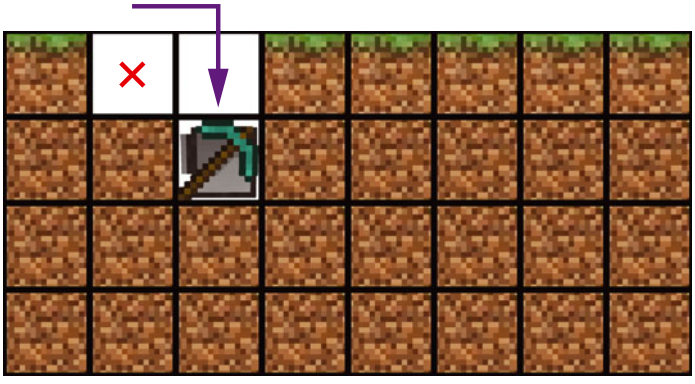


Рис. **15.6.** *Черепашка копает следующий колодец, начиная с поверхности*

После того как черепашка закончит добычу в этом колодце, разработку следующего она начнет с его дна. Черепашка добудет блок, который находится перед ней, переместится вперед и окажется в следующем колодце. Там она добывает блок, который находится под ней, делая этот колодец на од-

ну позицию глубже предыдущего, как показано на рис. 15.7. Добыча блока под черепашкой не требует топлива, поэтому черепашка использует только одну единицу топлива для добычи двух блоков. Эти действия повторяются каждый раз, когда разработка колодца начинается внутри шахты.

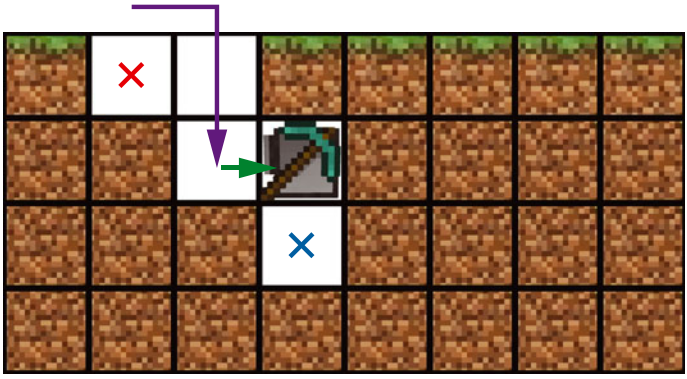


Рис. **15.7.** *Черепашка выкапывает блок перед собой, перемещается на один блок вперед, а затем выкапывает блок под собой*

В завершающей части алгоритма черепашка должна выйти на поверхность, чтобы полностью разработать текущий колодец. На рис. 15.8 финальный путь черепашки показан оранжевой стрелкой.

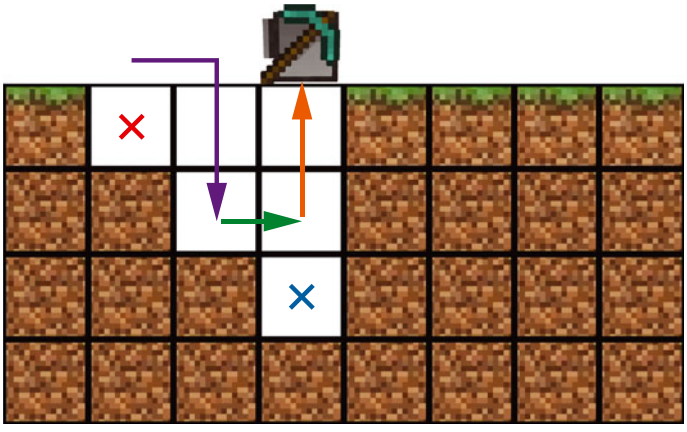


Рис. **15.8.** *Черепашка заканчивает разработку колодца, возвращаясь на поверхность*

Все описанные действия, кроме начального, черепашка повторяет циклически. Другими словами, черепашка ко-

пает вперед, начиная новый колодец, и, не опускаясь на его дно, заканчивает его добычу, двигаясь вверх, повторяя пути, обозначенные на рис. 15.9 фиолетовыми, зелеными и оранжевыми стрелками. По этому шаблону происходит разработка горного массива, по два колодца за один проход вниз, а затем вверх. Выполнение шаблона осуществляется до тех пор, пока черепашка не достигнет бедрока или указанной глубины.

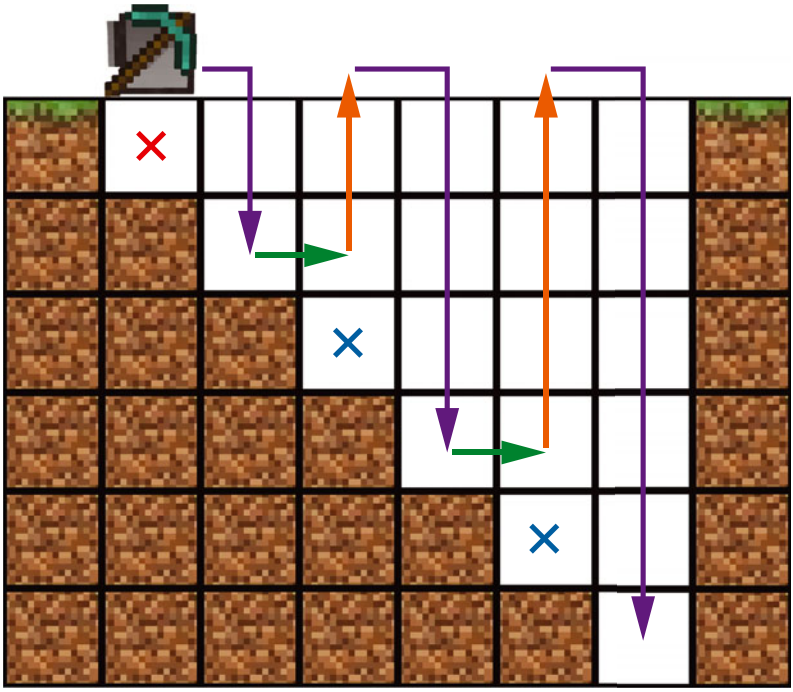


Рис. **15.9.** *Шаблон лестницы, разрабатываемой черепашкой*

Прежде чем приступить к разработке программы `stairminer`, в которой используется только что описанный алгоритм создания шахты с лестницей, необходимо написать некоторые вспомогательные функции. Они нам понадобятся для обработки ситуаций, когда черепашке будут попадаться песчаные или гравийные блоки.

ДОРАБОТКА МОДУЛЯ HARE

Если ты думаешь, что вызов функции `turtle.dig()` всегда очищает пространство перед черепашкой, то это не так. Песчаные и гравийные блоки над блоком, который добывает черепашка, падают вниз и блокируют ее. И, поскольку разработанный нами алгоритм создания шахты с лестницей предполагает, что черепашка периодически будет очищать пространство перед или над собой, нам нужна такая функция, с помощью которой черепашка будет раскапывать такие падающие блоки до тех пор, пока требуемое пространство полностью не освободится.

На рис. 15.10 показаны три песчаных блока, уложенных друг на друга перед черепашкой. Чтобы собрать песок, черепашка должна вызвать функцию `turtle.dig()` три раза.

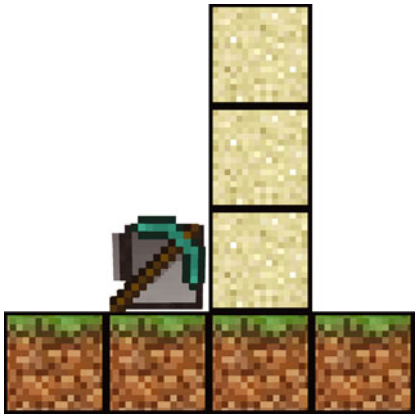


Рис. **15.10.** *Черепашка должна добыть блок три раза, чтобы очистить пространство перед собой, потому что песчаные блоки будут падать вниз*

Для этих целей мы напишем функции `digUntilClear()` и `digUpUntilClear()`, которые почти идентичны, но разрабатывают пространство перед и над черепашкой, соответственно.

Поскольку мы сможем использовать эти функции не только в этой программе, добавим их в модуль `hare`. В оболочке командной строки выполни команду `edit hare`. Перемести курсор в нижнюю часть файла и продолжи код, добавив следующие строки:

hare

```
     ...пропуск...
216. -- digUntilClear() продолжает добычу перед
черепашкой,
217. -- пока обнаруживает блоки (используется, если
218. -- песок или гравий падают на пути разработки)
219. function digUntilClear()
```

```
220.   while turtle.detect() do
221.     if not turtle.dig() then
222.       return false
223.     end
224.   end
225.   return true
226. end
227.
228. -- digUpUntilClear() продолжает добычу над
     черепашкой,
229. -- пока обнаруживает блоки (используется, если
230. -- песок или гравий падают на пути разработки)
231. function digUpUntilClear()
232.   while turtle.detectUp() do
233.     if not turtle.digUp() then
234.       return false
235.     end
236.   end
237.   return true
238. end
```

Сохрани программу и выйди из редактора после ввода инструкций. Эту программу также можно скачать, выполнив команду `pastebin get wwzvaKuW hare`.

КОД ФУНКЦИЙ DIGUNTILCLEAR() И DIGUPUNTILCLEAR()

Функция `digUntilClear()` вызывает функцию `turtle.dig()` до тех пор, пока перед черепашкой обнаруживаются блоки. Функция возвращает значение `true`, если пространство перед черепашкой пусто, и `false`, если пространство не может быть очищено (например, если черепашка сталкивается с бедроком, который невозможно разрушить).

hare

```
216. -- digUntilClear() продолжает добычу перед
     черепашкой,
217. -- пока обнаруживает блоки (используется, если
218. -- песок или гравий падают на пути разработки)
```

```
219. function digUntilClear()
220.   while turtle.detect() do
221.     if not turtle.dig() then
222.       return false
223.     end
224.   end
225.   return true
226. end
```

Наша функция должна продолжать вызывать функцию `turtle.dig()`, пока перед черепашкой есть какой-либо блок, поэтому в строке 220 начинается бесконечный цикл `while`. Этот цикл продолжается, пока функция `turtle.detect()` возвращает значение `true`, указывая на то, что черепашка обнаружила перед собой блок.

Код в строке 221 вызывает функцию `turtle.dig()`, которая возвращает значение `false`, если черепашка не может разрушить блок перед собой. Функция `turtle.dig()` возвращает значение `false` по двум причинам: если обнаружено пустое пространство (и добывать нечего) или если обнаружен блок бедрока (который нельзя добыть). При обнаружении пустого пространства исполнение не переходит в цикл `while`, потому что функция `turtle.detect()` вернула бы значение `false`, и условие `not turtle.detect()` стало бы `true`. В свою очередь, если функция `turtle.dig()` внутри цикла возвращает значение `false`, это значит, что черепашка встретила блок, который она не может обработать.

В противном случае исполнение цикла продолжается, пока пространство перед черепашкой не будет очищено. И лишь после этого цикл завершается в строке 224. По завершении цикла код в строке 225 возвращает значение `true`.

Код функции `digUpUntilClear()` в строках 228–238 почти идентичен функции `digUntilClear()`, за исключением того, что вместо функции `turtle.dig()` используется функция `turtle.digUp()` и вместо функции `turtle.detect()` – функция `turtle.detectUp()`.

Благодаря этим вспомогательным функциям, становится возможна реализация алгоритма создания шахты с лестницей. Теперь давай приступим к написанию программы `stairminer`!

КОД ПРОГРАММЫ STAIRMINER

Программа `stairminer` выкапывает шахту в форме лестницы в направлении перед черепашкой. В оболочке командной строки выполни команду `edit stairminer` и введи следующий код:

stairminer

```
 1. --[[ Программа создания шахты с лестницей Эла
    Свейгарта
 2. Разработка со строительством ступенек.]]
 3.
 4. os.loadAPI('hare')
 5.
 6. local cliArgs, targetDepth, columnDepth, result,
    errorMessage
 7.
 8. cliArgs = {...}
 9. targetDepth = tonumber(cliArgs[1])
10.
11. -- отображение информации об использовании
12. if targetDepth == nil or cliArgs[1] == '?' then
13.   print('Usage: stairminer <depth>')
14.   return
15. end
16.
17. turtle.digDown()
18.
19. columnDepth = 2
20. while true do
21.   -- движение вперед
22.   hare.digUntilClear()
23.   turtle.forward()
24.
25.   -- разработка во время спуска
```

```
26.   for i = 1, columnDepth do
27.     -- проверка на наличие бедрока
28.     result, errorMessage = turtle.digDown()
29.     if errorMessage == 'Unbreakable block detected'
        then
30.       print('Hit bedrock. Done.')
31.       return
32.     else
33.       turtle.down()
34.     end
35.   end
36.
37.   -- проверка соблюдения условия
38.   print('Current depth: ' .. columnDepth)
39.   if союбукв44.   -- движение вперед
45.   hare.digUntilClear()
46.   turtle.forward()
47.   turtle.digDown()
48.
49.   -- проверка, достаточно ли топлива, чтобы
      подняться и снова опуститься
50.   while turtle.getFuelLevel() < (columnDepth * 2) do
51.     -- попытка использовать топливо в инвентаре
52.     for slot = 1, 16 do
53.       turtle.select(slot)
54.       turtle.refuel()
55.     end
56.
57.     if turtle.getFuelLevel() < (columnDepth * 2)
        then
58.       print('Please load more fuel...')
59.       os.sleep(10)
60.     end
61.   end
62.
63.   -- проверка заполненности инвентаря
64.   while hare.selectEmptySlot() == false do
65.     print('Please unload the inventory...')
66.     os.sleep(10)
67.   end
68.
```

```
69.    -- разработка во время подъема
70.    for i = 1, columnDepth do
71.      hare.digUpUntilClear()
72.      turtle.up()
73.    end
74.
75.    columnDepth = columnDepth + 2
76. end
```

Сохрани программу и выйди из редактора после ввода всех инструкций.

ЗАПУСК ПРОГРАММЫ STAIRMINER

Установи черепашку на землю и запусти программу `stairminer`, передав ей целочисленный аргумент для указания максимальной глубины. Например, команда `stairminer 6` создаст ступенчатую шахту глубиной в шесть блоков. Обрати внимание, что значения, передаваемые в программу, округляются до ближайшего четного числа, так как черепашка всегда перемещается вниз по одному колодцу, а затем перемещается вверх по следующему колодцу. Таким образом, команда `stairminer 9` приведет к строительству шахты глубиной в 10 блоков, так же, как если бы была выполнена команда `stairminer 10`.

Если при запуске этой программы возникают ошибки, тщательно сравни свой код с листингом в этой книге, чтобы найти возможные опечатки. Если исправить программу не удается, удали файл, выполнив команду `delete stairminer`, а затем загрузи программу, выполнив команду `pastebin get PGH1WYpH stairminer`.

НАСТРОЙКА ПРОГРАММЫ STAIRMINER

Первые четыре строки кода программы `stairminer` содержат комментарии, описывающие программу, и вызов функции `os.loadAPI()` для загрузки модуля `hare`. После настройки программы объявляются несколько переменных, которые затем будут использованы.

stairminer

```
 6. local cliArgs, targetDepth, columnDepth, result,
errorMessage
```

Строка 6 содержит инструкцию `local` для создания пяти переменных. Переменная `cliArgs` содержит аргументы командной строки, которые программа использует в процессе исполнения. Переменные `targetDepth` и `columnDepth` отслеживают, как глубоко следует черепашке опуститься, и как глубоко она уже опустилась, соответственно. Переменные `result` и `errorMessage` сохраняют значения, возвращаемые при вызове функции `turtle.digDown()`.

Значение аргумента командной строки передается переменной `targetDepth`. Если игрок не ввел число, или это был символ `'?'`, программа отображает сообщение пользователю.

stairminer

```
 8. cliArgs = {...}
 9. targetDepth = tonumber(cliArgs[1])
10.
11. -- отображение информации об использовании
12. if targetDepth == nil or cliArgs[1] == '?' then
13.   print('Usage: stairminer <depth>')
14.   return
15. end
```

Аргументы командной строки хранятся в таблице `{...}`, которая создается в строке 8, под именем `cliArgs`. Если игрок не указывает численный аргумент командной строки (например если игрок выполняет команду `stairminer hello`), выражение `tonumber(cliArgs [1])` в строке 9 возвращает значение `nil`, которое присваивается переменной `targetDepth`. Код в строке 12 проверяет, присвоено ли переменной `targetDepth` значение `nil` или используется аргумент командной строки `'?'`. Если выполняется условие этой строки, код в строках 13 и 14 выводит сообщение пользователю.

СОЗДАНИЕ ПЕРВОЙ ЛЕСТНИЦЫ

Напомню, что черепашке не нужно двигаться вниз, чтобы выкопать первый блок. Она начинает с поверхности, выкапывает первый блок, затем переходит к следующему колодцу. Код в строке 17 выкапывает один блок. Это первый шаг в построении лестницы.

stairminer

```
17. turtle.digDown()
```

В первом колодце нужно выкопать только один блок, поэтому это действие в алгоритме больше не участвует. Остальную часть лестницы черепашка создает, используя цикл `while`, с помощью которого она поочередно копает то вверх, то вниз.

Первые два колодца черепашка копает вниз, а третий вверх. После первых трех колодцев она копает вниз каждый четный колодец и вверх каждый нечетный. Давай рассмотрим код, который черепашка использует, чтобы копать четные колодцы.

РАЗРАБОТКА ВНИЗ

Переменная `columnDepth` хранит количество блоков, которые черепашка должна выкопать в текущем колодце.

stairminer

```
19. columnDepth = 2
20. while true do
21.   -- движение вперед
22.   hare.digUntilClear()
23.   turtle.forward()
```

Поскольку черепашка копает первый колодец вне цикла, значение переменной `columnDepth` начинается с `2` и увеличивается в конце каждой итерации цикла `while`, который начинается в строке 20. Внутри цикла `while` код в строке 22 выкапывает блок перед черепашкой с помощью функции `hare.digUntilClear()`. Если на это место падают

песчаные или гравийные блоки, эта функция выкопает их все, до того, как код в строке 23 переместит черепашку вперед на одну позицию. Когда интерпретатор выполнит код в строке 23, черепашка сможет выкопать блок под собой, уже находясь в следующем колодце.

После перемещения черепашки вперед, внутри цикла `while` начинается цикл `for`.

stairminer

```
25.   -- разработка во время спуска
26.   for i = 1, columnDepth do
27.     -- проверка на наличие бедрока
28.     result, errorMessage = turtle.digDown()
29.     if errorMessage == 'Unbreakable block detected'
        then
30.       print('Hit bedrock. Done.')
31.       return
32.     else
33.       turtle.down()
34.     end
35.   end
```

Цикл `for`, начинающийся в строке 26, инструктирует черепашку копать вниз, вызывая функцию `turtle.digDown()` в строке 28. Однако добыча может завершиться непредвиденно, если черепашка достигнет дна мира Minecraft, который состоит из бедрока, не поддающегося обработке. В этом случае, если под черепашкой находится бедрок, в строке 28 функция `turtle.digDown()` возвращает значение `false` для своего первого возвращаемого значения (которое мы сохраняем в переменной `result`), и строку `'Unbreakable block detected'` для второго возвращаемого значения (которое мы сохраняем в переменной `errorMessage`). Код в строке 29 проверяет наличие строки `'Unbreakable block detected'` в переменной `errorMessage`, и если эта строка там есть, код в строке 30 отображает сообщение игроку `'Hit bedrock. Done.'`. Если черепашка встречает бедрок, она не может продолжать добычу блоков, поэтому программа завершается в строке 31.

В противном случае, если под черепашкой не обнаружен бедрок, она может опуститься вниз, и программа продолжит выполнение кода после инструкции `else` в строке 32. Строка 33 находится внутри блока инструкции `else` и перемещает черепашку вниз, в пространство, очищенное после добычи блока. Код в строке 34 завершает блок инструкции `else`, а строка 35 завершает цикл `for`. Код внутри цикла `for` в строках 26 – 35 производит `columnDepth` итераций (если в процессе его исполнения черепашка не достигнет бедрока).

После завершения цикла `for` программа переходит к исполнению остальной части цикла `while`.

stairminer

```
37.   -- проверка соблюдения условия
38.   print('Current depth: ' .. columnDepth)
39.   if columnDepth >= targetDepth then
40.     print('Done.')
41.     return
42.   end
```

Код в строке 38 выводит сообщение, содержащее значение переменной `columnDepth`, которое указывает, на какое количество блоков опустилась черепашка, то есть сколько итераций совершено в цикле `for` в строках 26–35. Инструкция `if` в строке 39 проверяет, достигло ли значение переменной `columnDepth` (глубина колодца, который копает черепашка) величины большей или равной, чем значение переменной `targetDepth`. Если условие истинно, код в строке 40 сообщает игроку, что программа выполнена, а инструкция `return` в строке 41 завершает программу. Напомню, что инструкция `return` вне какой-либо функции, например в строке 41, завершает работу программы ComputerCraft.

Если черепашка в процессе копания не встретила бедрок, она должна очистить пространство впереди, переместиться и выкопать блок под собой. С выполнения этих действий начинается выработка следующего колодца.

stairminer

```
44.    -- движение вперед
45.    hare.digUntilClear()
46.    turtle.forward()
47.    turtle.digDown()
```

В строке 45 черепашка вызывает функцию `hare.digUntilClear()`, чтобы выкопать блок перед собой. Код в строке 46 перемещает черепашку вперед, вызывая функцию `turtle.forward()`. Код в строке 47 выкапывает блок, который находится под черепашкой, вызывая функцию `turtle.digDown()`. На рис. 15.11 показано состояние лестницы, которую выкопала черепашка после исполнения кода в строке 47. Далее черепашка начнет копать текущий колодец снизу вверх.

Но прежде чем черепашка начнет движение вверх, нужно проверить, что у нее достаточно топлива для подъема. Давай рассмотрим, как происходит проверка уровня топлива.

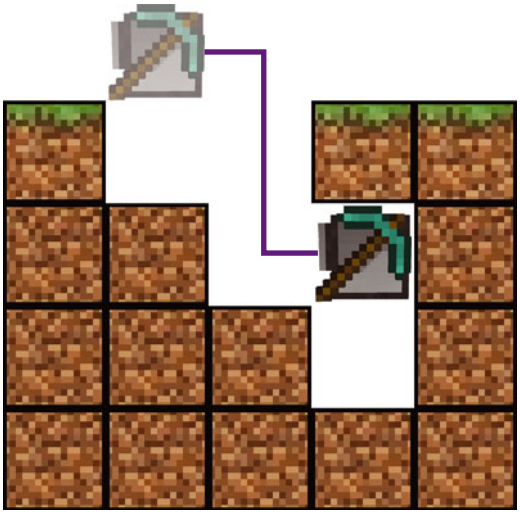


Рис. **15.11.** *Лестничная шахта после исполнения кода в строке 47*

ПРОВЕРКА «СЫТОСТИ» ЧЕРЕПАШКИ

Если у черепашки заканчивается топливо, когда она копает вниз четный колодец, за ней можно легко последовать

и вытащить на поверхность. Но если топливо закончится в тот момент, когда она пробивает вверх нечетный колодец, добраться до нее будет уже не так просто. Например, крайне неприятно повторение ситуации, изображенной на рис. 15.12.

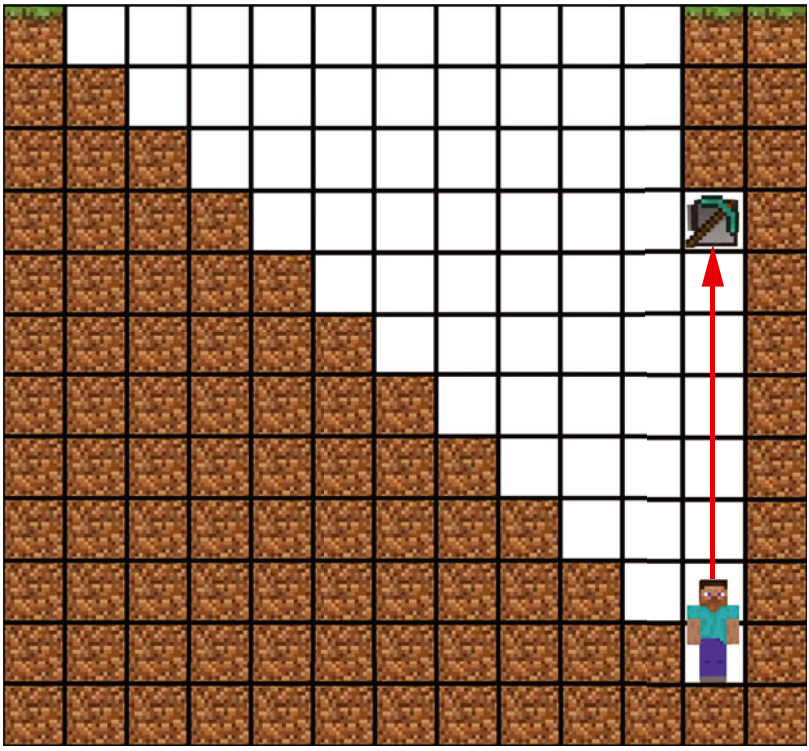


Рис. **15.12.** *Если у черепашки заканчивается топливо на полпути вверх внутри глубокого нечетного колодца, ее может быть трудно достать*

Поэтому нужно проверить, что у черепашки достаточно топлива, чтобы достичь дна колодца, а затем вернуться обратно. Поскольку цикл `for` в строках 26–35 перемещает черепашку вниз `columnDepth` раз, то, чтобы добраться до поверхности, ей надо переместиться такое же количество раз вверх. Следовательно, в общей сложности, черепашка должна совершить `columnDepth * 2` перемещений. (Напомню, что для перемещения на одну позицию черепашка использует одну единицу топлива; добыча блока и поворот топлива не требуют.)

Если у черепашки нет по крайней мере `columnDepth * 2` единиц топлива, она должна попытаться использовать в качестве топлива любые элементы, которые содержатся в ее инвентаре. Когда черепашка вырубает шахту, она часто добывает уголь или другие блоки, которые может использовать в качестве топлива. Если топливо на исходе, мы можем инструктировать черепашку использовать добытый уголь. В строке 50 начинается цикл `while`, который проверяет уровень топлива черепашки, вызывая функцию `getFuelLevel()`.

stairminer

```
49.   -- проверка, достаточно ли топлива, чтобы подняться
      и снова опуститься
50.   while turtle.getFuelLevel() < (columnDepth * 2) do
```

Если у черепашки меньше топлива, чем `columnDepth *2` единиц, исполнение переходит к циклу `while`, в котором происходит поиск топлива в инвентаре черепашки.

Вложенный цикл `for` ищет топливо в каждой ячейке инвентаря черепашки.

stairminer

```
51.     -- попытка использовать топливо в инвентаре
52.     for slot = 1, 16 do
53.       turtle.select(slot)
54.       turtle.refuel()
55.     end
```

В строке 52 этого цикла, переменная `slot` принимает значения от `1` до `16`. На каждой итерации цикла код в строке 53 выбирает ячейку в инвентаре черепашки, а код в строке 54 пытается использовать ее содержимое в качестве топлива. Если содержимое выбранной ячейки непригодно для использования в качестве топлива, функция `turtle.refuel()` ничего не делает.

Израсходовав весь инвентарь, который удалось использовать в качестве топлива, черепашка снова перепроверяет топливные остатки. Если топлива все равно не хватает, черепашка выводит сообщение `'Please load more fuel...'`

и ждет 10 секунд, чтобы игрок вручную поместил топливо в ее инвентарь.

stairminer

```
57.     if turtle.getFuelLevel() < (columnDepth * 2)
        then
58.       print('Please load more fuel...')
59.       os.sleep(10)
60.     end
61.   end
```

Код в строке 57 проверяет уровень топлива черепашки, используя то же условие, что и цикл `while` в строке 50. Если у черепашки топлива больше, чем `columnDepth * 2` единиц, программа пропускает блок кода конструкции `if`. Тогда исполнение, достигнув строки 61, возвращается к строке 50, чтобы перепроверить условие цикла `while`. Это условие то же самое, что и в строке 77, поэтому тоже будет ложным (`false`). Таким образом, при достаточном количестве топлива исполнение цикла `while` не происходит.

Если же условие в строке 57 выполняется (`true`) и топлива недостаточно, то код в строке 58 отображает сообщение `'Please load more fuel...'` и ждет 10 секунд при помощи вызова функции `os.sleep(10)` в строке 59. Затем, после достижения конца цикла `while`, происходит возвращение к строке 50, чтобы перепроверить условие.

Если топливо помещается в инвентарь черепашки, оно потребляется на следующей итерации цикла `while`. Если игрок не загрузил топливо и его уровень остается прежним, программа снова возвращается к циклу `while`. Когда он запустится, опять отобразится сообщение `'Please load more fuel...'`, и опять будет пауза в 10 секунд. Так будет продолжаться до тех пор, пока игрок не загрузит достаточно топлива, чтобы функция `getFuelLevel()` вернула число, большее, чем `columnDepth * 2`.

После того как программа убедится, что у черепашки достаточно топлива, необходимо проверить, чтобы инвентарь черепашки не был переполнен, иначе она не сможет

добыть следующий блок. Давай посмотрим, как это проверяется.

ПРОВЕРКА ИНВЕНТАРЯ ЧЕРЕПАШКИ

После того как черепашка выполнила проверку уровня топлива, программа оставляет цикл `while` и выполняет проверку заполненности инвентаря. Если он полностью заполнен, все добытые блоки черепашка будет попросту выбрасывать, то есть работать совершенно впустую. Заполнение инвентаря черепашка проверяет на дне выкопанного колодца, поэтому, если инвентарь заполнен, игрок может добраться до черепашки, чтобы разгрузить ее.

Функция `selectEmptySlot()` в модуле `hare` выбирает первую пустую ячейку в инвентаре черепашки. Но для хранения блоков, выкопанных при строительстве лестницы, выбирать пустую ячейку не нужно. И поскольку функция `selectEmptySlot()` возвращает значение `false`, если нет свободных ячеек в инвентаре, мы можем использовать ее, чтобы определить, заполнен ли инвентарь черепашки.

stairminer

```
63.   -- проверка заполненности инвентаря
64.   while hare.selectEmptySlot() == false do
65.     print('Please unload the inventory...')
66.     os.sleep(10)
67.   end
```

Цикл `while` в строке 64 содержит функцию `selectEmptySlot()`, чтобы проверить, есть ли в инвентаре хотя бы одна пустая ячейка. Если нет, исполнение переходит к строке 65, которая отображает сообщение `'Please unload the inventory...'`, а код в строке 66 вызывает ожидание продолжительностью 10 секунд. Этот код аналогичен строкам 58 и 59 для проверки топлива. Цикл `while` останавливается только после того, как игрок выгрузит блоки из инвентаря черепашки и функция `selectEmptySlot()` вернет значение `true`. После проверки инвентаря исполнение продолжается со строки 70, а черепашка начинает копать вверх.

РАЗРАБОТКА ВВЕРХ

Поскольку черепашка находится на глубине в `columnDepth` блоков ниже поверхности, ей необходимо прокопать целый колодец, чтобы вернуться наверх. Перемещение вверх и добыча блоков производятся в цикле `for`, который начинается в строке 70.

stairminer

```
69.   -- разработка во время подъема
70.   for i = 1, columnDepth do
71.     hare.digUpUntilClear()
72.     turtle.up()
73.   end
```

Код в строке 71 вызывает функцию `digUpUntilClear()`, а следующая строка перемещает черепашку вверх. Мы используем функцию `hare.digUpUntilClear()` вместо `turtle.digUp()`, потому что выше черепашки может находиться песок или гравий. С окончанием цикла черепашка возвращается на поверхность.

Код в строке 75 увеличивает значение переменной `columnDepth` на `2`, потому что одна итерация цикла выстраивает два колодца, каждый из которых глубже предыдущего на один блок, следовательно, через два колодца глубина увеличивается на два блока.

stairminer

```
75.   columnDepth = columnDepth + 2
76. end
```

Увеличивая значение переменной `columnDepth` на `2`, черепашка перемещается на два блока глубже каждый раз, когда копает вниз. Строка 76 завершает цикл `while`, начавшийся в строке 20. Когда исполнение возвращается к строке 20, строки 22 и 23 переместят черепашку вперед к следующему колодцу, который ей надо прокопать вниз и затем вернуться на поверхность. Черепашка останавливается только тогда, когда значение переменной `columnDepth` равно или больше значения `targetDepth` (которое игрок

задал в качестве аргумента командной строки) или когда черепашка натыкается на бедрок.

После выполнения программы `stairminer` у тебя будет шахта в форме лестницы, которая обеспечит безопасный доступ глубоко под землю, а инвентарь черепашки будет заполнен блоками, добытыми при разработке.

ДОПОЛНИТЕЛЬНОЕ ЗАДАНИЕ: ВЫСОКИЙ ТОННЕЛЬ

Возможно, на дне шахты с лестницей тебе захочется выкопать туннель. В памяти черепашки записана готовая программа с именем `tunnel`, которая добывает пространство размером 3 × 2 блока перед черепашкой и длиной, которую ты укажешь в виде аргумента командной строки. Например, ты можешь выполнить команду `tunnel 10`, и черепашка прокопает туннель размером 3 × 2 × 10 блоков. Эти туннели достаточно высоки, чтобы игрок мог пройти, но могут вызывать клаустрофобию. Попробуй создать собственную программу `hightunnel`, которая тоже будет копать туннель, но уже высотой в четыре блока.

ЧТО МЫ УЗНАЛИ

Программа `stairminer` использует сложный алгоритм, но как только ты его напишешь, всю тяжелую работу по добыче полезных ископаемых возьмет на себя черепашка. Даже если тебе не нужны блоки, которые соберет черепашка при строительстве шахты, ты сможешь использовать эту программу, чтобы строить лестницы для подземного логова со стенами из каменного кирпича (их изготовит генератор булыжника). С тех пор, как ты научился программировать черепашек, твои возможности стали по-настоящему безграничными!

Изучая программирование и учась говорить на языке компьютера, ты можешь выстраивать армии черепашек для автоматизации многих задач в игре Minecraft. Программирование – это полезное и забавное умение, и я надеюсь, что ты продолжишь экспериментировать, но уже по

своему усмотрению. Когда дело доходит до программирования, всегда есть чему поучиться, поэтому попробуй придумать свои собственные программы. Удачи!

ПРИЛОЖЕНИЕ А. СПИСОК ФУНКЦИЙ

В этом разделе ты найдешь краткое описание функций, используемых в этой книге. Эти функции сгруппированы по программным интерфейсам (Application Programming Interface – API), известным также как модули. Это неполный список функций ComputerCraft. Более полное и подробное описание всех доступных функций можно получить по адресу **www.computercraft.info/wiki/Category:APIs**.

Если ты захочешь узнать больше о Lua, воспользуйся справочным руководством, которое доступно в Интернете по адресу **lua.org.ru/contents_ru.html**. Обрати внимание, что в моде ComputerCraft используется язык Lua 5.1, хотя доступны новые версии Lua.

Примечание *Некоторые элементы синтаксиса функций в этом разделе отформатированы курсивом, указывая, где должен находиться параметр и, при необходимости, его тип.*

ИНТЕРФЕЙС FS (ФАЙЛОВАЯ СИСТЕМА)

Черепашки используют файловую систему, совместимую с той, что установлена на твоем компьютере. Таким образом, ты можешь обращаться к этим файлам, используя интерфейс `fs` и имя файла:

- **`fs.delete`** (**имяфайла**) удаляет файл с именем **имяфайла**
- **`fs.exists`** (**имяфайла**) возвращает значение **`true`**, если существует файл с именем **имяфайла**; в противном случае возвращает значение **`false`**

Примечание *Программы, описанные в этой книге, могут взаимодействовать с файлами, загружаемыми в память черепашки, но не с внешними файлами компьютера, на котором установлена игра Minecraft.*

ИНТЕРФЕЙС HARE

Читая эту книгу, ты писал программы для модуля (API) с именем `hare`. В отличие от других модулей, перечисленных в этом разделе, `hare` не предустановлен в моде ComputerCraft, поэтому, чтобы его загрузить, надо в оболочке командной строки выполнить команду `pastebin get wwzvaKuW hare`. Любая программа, предполагающая использование модуля API-интерфейса `hare`, должна содержать вызов `os.loadAPI('hare')`, тогда программа сможет использовать функции этого модуля.

- **`hare.buildFloor (длина, ширина)`** из блоков в инвентаре черепашки строит пол, для которого числами указаны его **длина** и **ширина**.
- **`hare.buildRoom (длина, ширина, высота)`** из блоков в инвентаре черепашки строит комнату, для которой числами указаны ее **длина, ширина и высота**.

- **hare.buildWall (длина, высота)** из блоков в инвентаре черепашки строит стену, для которой числами указаны ее **длина** и **высота**.

- **hare.countInventory()** возвращает общее количество блоков и предметов во всех ячейках инвентаря черепашки.

- **hare.digUntilClear()** продолжает добывать блок перед черепашкой, пока не очистит его полностью. Эта функция используется, когда гравий или песок осыпаются на место выкопанного блока.

- **hare.digUpUntilClear()** аналогична функции **hare.digUntilClear()** за исключением того, что она очищает пространство над, а не перед черепашкой.

- **hare.findBlock (идентификатор)** вращает черепашку вокруг своей оси, в поисках блока с указанным строковым **идентификатором** и останавливается, если находит его. Если черепашка не может найти указанный блок, она останавливается в исходном направлении. Если блок найден, функция возвращает значение **true**, а в противном случае – **false**.

- **hare.selectAndPlaceDown()** выбирает непустую ячейку инвентаря и помещает блок из этой ячейки под черепашку.

- **hare.selectEmptySlot()** выбирает пустую ячейку инвентаря и возвращает значение **true**, если такая ячейка обнаружена; в противном случае возвращает значение **false**.

- **hare.selectItem (идентификатор)** выбирает ячейку инвентаря, содержащую блок или предмет с указанным строковым идентификатором. Возвращает значение **true**, если искомый блок или предмет найден; в противном случае возвращает значение **false**.

- **hare.sweepField (длина, ширина, sweepFunc)** перемещает черепашку по каждому блоку в прямоугольной области, для которой числами указаны ее **длина, ширина,** и вызывает функцию **sweepFunc()** после каждого перемещения.

ИНТЕРФЕЙС IO (ВВОДА/ВЫВОДА)

Использование интерфейса `io` отображать на экране и обрабатывать текст,, введенный с клавиатуры. Существует несколько таких функций, но наиболее важной из них является `io.read()`:

- **`io.read()`**. Когда игрок вводит с клавиатуры какое-либо значение, после нажатия клавиши **Enter** функция возвращает это значение как строковую переменную.

ИНТЕРФЕЙС MATH (МАТЕМАТИЧЕСКИЕ ФУНКЦИИ)

Язык Lua содержит встроенные математические функции, которые можно вызывать даже из программ, не связанных с Lua в моде ComputerCraft. Этот модуль включает следующие функции:

- **`math.ceil(число)`** возвращает **`число`**, округленное в бóльшую сторону.

- **`math.floor(число)`** возвращает **`число`**, округленное в меньшую сторону.

- **`math.random(число1, число2)`** возвращает случайное целое в диапазоне между **`число1`** и **`число2`** включительно. Указание аргументов **`число1`** и **`число2`** необязательно. Вызов функции без аргументов возвращает случайное десятичное число в диапазоне от **`0.0`** до **`1.0`**. Если не указано **`число1`**, функция возвращает целое число в диапазоне от **`1`** до **`число2`**.

ИНТЕРФЕЙС OS (ОПЕРАЦИОННАЯ СИСТЕМА)

Операционная система ComputerCraft предоставляет следующие функции, которые могут использоваться черепашками и внутриигровыми компьютерами.

- **`os.getComputerLabel()`** возвращает метку (имя) черепашки в виде строковой переменной.

● **os.loadAPI(имяфайла)** загружает программу со строковым именем **имяфайла** в качестве модуля, чтобы текущая программа могла вызывать ее функции.

● **os.setComputerLabel(метка/nil)** Присваивает черепашке имя, указанное в качестве строкового значения **метка**. Если вместо **метки** указано значение **nil**, функция стирает имя черепашки.

● **os.sleep(время)** Приостанавливает выполнение программы на **время** секунд. Значение указывается числом.

ИНТЕРФЕЙС SHELL (ОБОЛОЧКА КОМАНДНОЙ СТРОКИ)

Черепашки, как и игрок, могут выполнять команды в оболочке командной строки. Запускать такие команды из своих программ, ты сможешь, используя функцию:

shell.run(команда) выполняет **команду**, как если бы игрок ввел аналогичную строку в оболочке командной строки. Функция возвращает значение **false**, если указанная **команда** не существует, при ее выполнении произошел сбой или произошел вызов функции **error()**; в противном случае возвращает значение **true**.

ИНТЕРФЕЙС STRING (СТРОКОВЫЕ ФУНКЦИИ)

Интерфейс `string` является частью Lua, и ты можешь вызывать эти функции из программ, не связанных с ComputerCraft. Хотя эти функции и не описаны в этой книге, я включил их в данный раздел, потому что они полезны.

● **string.find (строка1, строка2)** ищет **строку1** внутри **строки2** и возвращает два целых числа: начало и конец **строки2** в строке **строке1**. Например, выражение **string.find('hello', 'el')** возвращает 2 и 3, потому что 'el' начинается со второго символа строки 'hello' и заканчивается в третьем. Если с**трока2** не содержится в **строке1**, функция возвращает значение nil. Аргумент **строка2** может со-

держать текстовые шаблоны, описание которых выходит за рамки этой книги. Подробнее о текстовых шаблонах можно узнать на странице lua.org.ru/contents_ru.html (раздел 6.4.1).

● **string.sub** (**строка**, **начало**, **длина**) Возвращает часть **строки**, для которой числами указаны позиция **начала** и **длина** – количество символов. Аргумент **длина** необязателен. Если он не передается, подстрока начинается с позиции **начало** и продолжается до конца строки. Например, выражение **string.sub('hello', 3, 2)** возвращает **'ll'**, а выражение **string.sub('hello', 2)** возвращает **'ello'**.

ИНТЕРФЕЙС TEXTUTILS (ТЕКСТОВЫЕ ЭФФЕКТЫ)

Используя интерфейс `textutils`, ты можешь отображать текст по одному символу, создавая эффект пишущей машинки:

● **textutils.slowPrint(строка, скорость)** аналогична функции **print()**, за исключением того, что выводит **строку**, поочередно печатая по одному символу. Аргумент **скорость** не обязателен, он лишь указывает скорость печати (количество символов в секунду).

ИНТЕРФЕЙС TURTLE (УПРАВЛЕНИЕ ЧЕРЕПАШКОЙ)

Интерфейс `turtle` содержит функции, которые твои программы могут вызывать, чтобы черепашка выполняла определенные действия. Давай рассмотрим эти функции и действия, совершаемые ими.

СТРОИТЕЛЬНЫЕ ФУНКЦИИ

Вызывая эти функции, черепашки могут строить, укладывая блоки. Но одна и та же функция может выполнять различные действия, в зависимости от того, какой именно

предмет или блок находится в текущей ячейке инвентаря черепашки.

● **`turtle.place()`** выполняет действие с предметом/блоком из текущей ячейки инвентаря черепашки. Для строительных блоков эта функция укладывает блок в мире Minecraft. Однако, если это один из специальных предметов, перечисленных в табл. 1, то он будет использован способом, указанным в этой таблице. Функция возвращает значение **`false`**, если черепашка не может уложить блок или выполнить с блоком предписанные действия.

● **`turtle.placeDown()`** аналогична функции **`turtle.place()`**, но выполняет действие в пространстве под черепашкой.

● **`turtle.placeUp()`** аналогична функции **`turtle.place()`**, но выполняет действие в пространстве над черепашкой.

Табл. 1. Специальные предметы, которые могут использовать функции `place`.

Предмет	Действие
Броня	Устанавливает броню на доспехи.
Лодка	Помещает лодку на воду.
Красители	Окрашивает шерсть.
Пустое ведро	Собирает лаву или воду. Может также подоить корову.
Фейерверк	Запускает фейерверк.
Огниво	Поджигает горючий блок или активирует портал в Незер.
Вагонетка	Помещает вагонетку на рельсы.
Саженцы, цветы или семена	Выращивает цветы или семена.
Ножницы	Стрижет овец и собирает шерсть.
Табличка	Устанавливает табличку с текстом. Чтобы написать текст на табличке, передай в функцию строку. Например, `turtle.place('This\nis a\nsign.')`.
Яйца призыва	Спаунит моба.

ТОПЛИВНЫЕ ФУНКЦИИ

Черепашка тратит единицу топлива каждый раз, когда перемещается на расстояние в один блок. Если топливо заканчивается, черепашка не может передвигаться. По этой причине важно учитывать необходимость дозаправки, и другие функции, связанные с топливом:

- **`turtle.getFuelLevel()`** возвращает количество топлива, которое в настоящее время есть у черепашки. Возвращает значение **`'unlimited'`**, если в файле конфигурации ComputerCraft.cfg отключена необходимость использования топлива.

- **`turtle.getFuelLimit()`** возвращает максимальное количество топлива, которое может хранить черепашка. Для большинства черепашек предел составляет 200 000 единиц; для некоторых других лимит составляет 100 000 единиц. Возвращает значение **`'unlimited'`**, если в файле конфигурации *ComputerCraft.cfg* отключена необходимость использования топлива.

- **`turtle.refuel(количество)`** добавляет в качестве топлива содержимое текущей ячейки в указанном **`количестве`**. Численный аргумент **`количество`** необязателен. Если он не указан, функция добавляет все содержимое текущей ячейки.

ФУНКЦИИ, СВЯЗАННЫЕ С ИНВЕНТАРЕМ

Инвентарь черепашки состоит из 16 пронумерованных ячеек. Ты можешь использовать функции, связанные с инвентарем, чтобы черепашка выполняла различные действия. Эти функции часто принимают численные параметры, чтобы определить, в какой из пронумерованных ячеек следует выполнить действие.

- **`turtle.compareTo(номерячейки)`** возвращает значение **`true`**, если блок/предмет в текущей ячейке совпадает с блоком/предметом в ячейке с указанным числом **`номерячейки`**; в противном случае возвращает значение **`false`**.

● **turtle.drop(количество)** Перемещает **количество** блоков/предметов из текущей ячейки в мир Minecraft или в контейнер перед черепашкой. **Количество** указывать необязательно. Если оно не указано, функция вынимает все блоки/предметы из текущей ячейки. Возвращает значение **true**, если какие-либо блоки/предметы были извлечены; в противном случае возвращает значение **false**.

● **turtle.dropDown(количество)** аналогична функции turtle.drop(), за исключением того, что эта функция извлекает предметы в пространство или контейнер под черепашкой.

● **turtle.dropUp(количество)** аналогична **turtle.drop()**, за исключением того, что эта функция извлекает предметы в пространство или контейнер над черепашкой.

● **turtle.equipLeft()** оснащает черепашку инструментом с левой стороны, если он содержится в текущей ячейке. Возвращает значение **true**, если экипировка произведена; в противном случае возвращает значение **false**.

● **turtle.equipRight()** оснащает черепашку инструментом с правой стороны, если он содержится в текущей ячейке. Возвращает значение **true**, если экипировка произведена; в противном случае возвращает значение **false**.

● **turtle.getItemCount(номерячейки)** возвращает количество блоков/предметов в ячейке с номером **номерячейки**. Используется текущая ячейка, если номерячейки не указан.

● **turtle.getItemDetail(номерячейки)** возвращает табличное значение с информацией о содержимом ячейки с номером **номерячейки**. Возвращает значение **nil**, если ячейка пуста. Используется текущая ячейка, если номерячейки не указан.

● **turtle.getItemSpace(номерячейки)** Возвращает количество свободного места в ячейке с номером **номерячейки**. Используется текущая ячейка, если номерячейки не указан.

● **turtle.getSelectedSlot()** возвращает номер текущей ячейки (от **1** до **16**).

- **turtle.select(номерячейки)** делает указанную ячейку текущей.

- **turtle.suck(количество)** собирает **количество** предметов из пространства или контейнера перед черепашкой и помещает их либо в текущую ячейку (если она пуста), либо в ближайшую пустую ячейку. Численный аргумент **количество** необязателен. Если он не указан, загружается полный стек. (Для большинства блоков полный стек равен 64 экземплярам, хотя некоторые предметы, такие как яйца, снежки или пустые ведра, могут складироваться только в количестве до 16 штук, а оружие и инструменты и вовсе по одному экземпляру.) Возвращает значение **true**, если какие-либо блоки/предметы были взяты; в противном случае возвращает значение **false**.

- **turtle.suckDown(количество)** аналогична функции **turtle.suck()**, за исключением того, что эта функция принимает блоки/предметы из пространства или контейнера под черепашкой.

- **turtle.suckUp (количество)** аналогична функции **turtle.suck()**, за исключением того, что эта функция принимает блоки/предметы из пространства или контейнера над черепашкой.

- **turtle.transferTo(номерячейки, количество)** Переносит количество блоков/предметов из текущей ячейки в ячейку с номером **номерячейки**. Численный аргумент **количество** необязателен. Если он не указан, функция пытается передать все блоки/предметы из текущей ячейки. Возвращает значение **true**, если какие-либо блоки/предметы были перенесены; в противном случае возвращает значение **false**.

ФУНКЦИИ ДВИЖЕНИЯ

Черепашки могут двигаться в любом направлении, если пространство на пути свободно. Перечисленные ниже функции ты можешь использовать, чтобы перемещать черепашку в разных направлениях. Все функции движения

возвращают значение `true`, если черепашка совершила перемещение; в противном случае они возвращают `false`.

- **`turtle.back()`** перемещает черепашку на один блок назад.
- **`turtle.down()`** перемещает черепашку на один блок вниз.
- **`turtle.forward()`** перемещает черепашку на один блок вперед.
- **`turtle.turnLeft()`** поворачивает черепашку налево; топливо при этом не используется.
- **`turtle.turnRight()`** поворачивает черепашку направо; топливо при этом не используется.
- **`turtle.up()`** перемещает черепашку на один блок вверх.

Функции восприятия

Черепашки могут определять блоки, расположенные перед ними, а также выше или ниже их самих, используя следующие функции.

- **`turtle.compare()`** возвращает значение **`true`**, если блок перед черепашкой имеет тот же тип, что и блок в текущей ячейке; в противном случае возвращает значение **`false`**.
- **`turtle.compareDown()`** аналогична **`turtle.compare()`**, но сравнивает блок под черепашкой с блоком в текущей ячейке.
- **`turtle.compareUp()`** аналогична **`turtle.compare()`**, но сравнивает блок над черепашкой с блоком в текущей ячейке.
- **`turtle.detect()`** возвращает значение **`true`**, если перед черепашкой находится блок; в противном случае возвращает значение **`false`**.
- **`turtle.detectDown()`** аналогична функции **`turtle.detect()`**, но проверяет наличие блока под черепашкой.

● **turtle.detectUp()** аналогична функции **turtle.detect()**, но проверяет наличие блока над черепашкой.

● **turtle.inspect()** возвращает два значения: **true** и табличное значение с информацией о блоке перед черепашкой. Если ни один блок не находится перед черепашкой, возвращается значение **false**.

● **turtle.inspectDown()** аналогична функции **turtle.inspect()**, но возвращает информацию о блоке под черепашкой.

● **turtle.inspectUp()** аналогична функции **turtle.inspect()**, но возвращает информацию о блоке над черепашкой.

ФУНКЦИИ, СВЯЗАННЫЕ С ИНСТРУМЕНТАМИ

Черепашки могут выполнять действия с использованием инструментов, которыми они оборудованы. Для каждого инструмента определены собственные функции и действия, которые черепашка может выполнять. Ты можешь оборудовать черепашек новыми алмазными кирками, лопатами, топорами, мечами, мотыгами и верстаками. Для этого предусмотрены следующие функции:

● **turtle.attack():** если черепашка экипирована мечом, она атакует находящееся перед ней. Функция возвращает значение **true**, если атака произведена; в противном случае возвращает значение **false**, например, если атаковать было нечего.

● **turtle.attackDown()** аналогична функции **turtle.attack()**, но черепашка атакует пространство под собой.

● **turtle.attackUp()** аналогична функции **turtle.attack()**, но черепашка атакует пространство над собой.

● **turtle.craft (количество)**. Производит **количество** предметов или блоков из блоков, содержащихся в инвентаре черепашки, а готовую продукцию помещает в текущую ячейку. Так как крафт производится по соответ-

ствующему рецепту, для работы этой функции, необходимо размещать блоки в инвентаре в соответстви с рецептом. Численный аргумент **количество** необязателен. Если он не указан, функция изготовит столько предметов или блоков, на сколько хватит ингредиентов в инвентаре. Возвращает значение **true**, если крафт был выполнен успешно; в противном случае возвращает значение **false**, например, если ингредиенты не соответствуют рецепту.

- **turtle.dig()** выкапывает или добывает блок перед черепашкой. Если черепашка оборудована киркой, то она добывает блоки. Если мотыгой, то обрабатывает землю в пашню. Возвращает значение **true**, если операция завершилась успешно; в противном случае возвращает значение **false**.

- **turtle.digDown()** аналогична функции **turtle.dig()**, но черепашка обрабатывает блок под собой.

- **turtle.digUp()** аналогична функции **turtle.dig()**, но черепашка обрабатывает блок над собой. Обрати внимание, что эта функция не позволяет обрабатывать землю.

ФУНКЦИИ LUA

Следующие функции относятся к внутренним командам интерпретатора Lua, поэтому вводить имя модуля перед именем функции для вызова не надо.

- **error(**сообщение**)** завершает работу программы и выводит текстовое **сообщение**. Строковый аргумент **сообщение** указывать необязательно.

- **exit()** выходит из интерактивной оболочки. Ты можешь использовать эту функцию, только находясь в интерактивной оболочке.

- **print(строка/число)** отображает значение аргумента строка/число на экране, а затем начинает новую строку. Аргумент строка/число указывать необязательно. В этом случае, функция просто начинает новую строку.

ПРИЛОЖЕНИЕ Б. СПИСОК ИДЕНТИФИКАТОРОВ

В этой книге при написании программ мы использовали идентификаторы блоков Minecraft.

В этом приложении перечислены идентификаторы блоков, которые ты можешь использовать при внесении изменений в рассмотренный нами код или при написании новых программ. Прежде чем рассмотреть список идентификаторов, давай узнаем, как можно определить идентификатор нужного блока.

ПОИСК ИДЕНТИФИКАТОРА БЛОКА

Чтобы воспользоваться каким-либо блоком, нужно знать его идентификатор, ведь различные типы блоков требуют различных действий. Когда черепашка сталкивается с каким-нибудь блоком, как показано на рис. 1, чтобы определить тип этого блока, используй функцию `turtle.inspect()`.

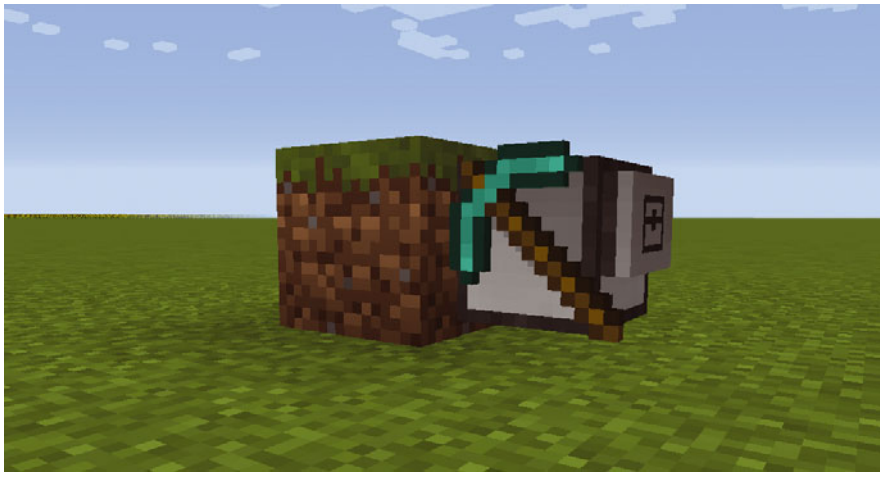


Рис. **1**. *Черепашка, обращенная к блоку травы перед вызовом функции* `turtle.inspect()`

Ты можешь получить всю информацию, связанную с блоком, вызвав функцию `turtle.inspect()` и ознакомившись с табличным значением, которое вернет функция. Ключ `'name'` в возвращенной таблице содержит значение идентификатора блока, с которым столкнулась черепашка. Например если черепашка находится перед блоком травы и ты вызываешь функцию `turtle.inspect()`, в оболочке Lua возвращаемая таблица будет выглядеть так:

```
lua> turtle.inspect()
true
{
  state = {
    snowy = false,
  },
  name = «minecraft:grass»,
  metadata = 0,
}
```

Идентификатор блока травы – `"minecraft:grass"`. В большинстве случаев идентификатора вполне достаточно, чтобы использовать нужные блоки. Но в некоторых случаях могут потребоваться и другие ключи таблицы, которую возвращает функция `turtle.inspect()`.

Различия между блоками с одинаковым идентификатором[10]

Некоторые блоки имеют один и тот же идентификатор, и, таким образом, для их однозначной идентификации необходимы другие ключи из соответствующей таблицы данных. Например, дубовые доски и еловые доски имеют одинаковый идентификатор[11] `"minecraft:planks"`, но их ключи `metadata` имеют разные значения. Если вызвать функцию `turtle.inspect()`, когда черепашка находится перед дубовыми досками, функция возвращает следующую таблицу:

```
lua> turtle.inspect()
true
{
  state = {
    variant = "oak",
  },
  name = "minecraft:planks",
  metadata = 0,
}
```

Но если черепашка находится перед еловыми досками, функция возвращает следующую таблицу:

```
lua> turtle.inspect()
true
{
  state = {
    variant = "spruce",
  },
  name = "minecraft:grass",
  metadata = 1,
}
```

Хотя идентификатор обоих блоков один и тот же – `"minecraft:planks"`, значения ключа `metadata` разные

[10] С версии 1.13 разные *(прим. перев.)*

[11] Идентификаторы в списке приведены для версии Minecraft 1.13 *(прим. перев.)*.

(0 для дуба и 1 для ели). Таким образом, эти два типа блоков можно различать. Таблица данных содержит еще один ключ, state, которому соответствует переменная variant. Значение этой переменной также предоставляет дополнительную информацию о запрашиваемом блоке.

СПИСОК ИДЕНТИФИКАТОРОВ БЛОКОВ

В табл. 1 перечислены идентификаторы большинства блоков, с которыми может взаимодействовать твоя черепашка. Однако в этой таблице не указаны различия между блоками, которые используют одинаковые идентификаторы. Например, в таблице указан идентификатор досок, но не указаны ключи, по которым различаются дубовые и еловые доски. Таким образом, если тебе понадобится найти разницу между блоками с одинаковым идентификатором, придется воспользоваться функцией turtle.inspect().

Табл. 1. Идентификаторы блоков и предметов в игре Minecraft[12]

Значок	Название	Идентификатор
	Акациевая дверь	'minecraft:acacia _ door'
	Акациевый забор	'minecraft:acacia _ fence'
	Акациевая калитка	'minecraft:acacia _ fence _ gate'
	Акациевые ступеньки	'minecraft:acacia _ stairs'
	Активирующие рельсы	'minecraft:activator _ rail'
	Алмазная руда	'minecraft:diamond _ ore'

[12] Идентификаторы в списке приведены для версии Minecraft 1.13 *(прим. перев.)*.

Значок	Название	Идентификатор
	Алмазный блок	`'minecraft:diamond _ block'`
	Арбуз	`'minecraft:melon'`
	Бедрок	`'minecraft:bedrock'`
	Белая глазурованная керамика	`'minecraft:white _ glazed _ terracotta'`
	Белый шалкеровый ящик	`'minecraft:white _ shulker _ box'`
	Березовая дверь	`'minecraft:birch _ door'`
	Березовый забор	`'minecraft:birch _ fence'`
	Березовая калитка	`'minecraft:birch _ fence _ gate'`
	Березовые ступеньки	`'minecraft:birch _ stairs'`
	Бетон (белый)	`'minecraft:white _ concrete'`
	Бирюзовая глазурованная керамика	`'minecraft:cyan _ glazed _ terracotta'`
	Бирюзовый шалкеровый ящик	`'minecraft:cyan _ shulker _ box'`
	Блок коричневого гриба	`'minecraft:brown _ mushroom _ block'`
	Блок красного гриба	`'minecraft:red _ mushroom _ block'`
	Блок нароста из Незера	`'minecraft:nether _ wart _ block'`

Значок	Название	Идентификатор
	Блок портала в Энд	'minecraft:end _ gateway'
	Блок слизи	'minecraft:slime _ block'
	Блок снега	'minecraft:snow _ block'
	Большегрузная весовая пластина	'minecraft:heavy _ weighted _ pressure _ plate'
	Бревно (Акация/Темный дуб)	'minecraft:acacia _ log' 'minecraft:dark _ oak _ log'
	Булыжная ограда	'minecraft:cobblestone _ wall'
	Булыжник	'minecraft:cobblestone'
	Булыжные ступеньки	'minecraft:cobblestone _ stairs'
	Верстак	'minecraft:crafting _ table'
	Весовая пластина	'minecraft:light _ weighted _ pressure _ plate'
–	Воздух	'minecraft:air'
	Воронка	'minecraft:hopper'
	Выбрасыватель	'minecraft:dropper'
	Высокая трава	'minecraft:tall _ grass'
	Высокие цветы (разные)	'minecraft:sunflower' 'minecraft:lilac' 'minecraft:tall _ grass' 'minecraft:large _ fern' 'minecraft:rose _ bush' 'minecraft:peony'

Значок	Название	Идентификатор
	Глина (блок)	`'minecraft:clay'`

	Голова моба (разные)	`'minecraft:skeleton_skull'`
		`'minecraft:skeleton_wall_skull'`
		`'minecraft:wither_skeleton_skull'`
		`'minecraft:wither_skeleton_wall_skull'`
		`'minecraft:zombie_head'`
		`'minecraft:zombie_wall_head'`
		`'minecraft:player_head'`
		`'minecraft:player_wall_head'`
		`'minecraft:creeper_head'`
		`'minecraft:creeper_wall_head'`
		`'minecraft:dragon_head'`
		`'minecraft:dragon_wall_head'`

	Голубая глазурованная керамика	`'minecraft:light_blue_glazed_terracotta'`
	Голубой шалкеровый ящик	`'minecraft:light_blue_shulker_box'`
	Гравий	`'minecraft:gravel'`
	Губка	`'minecraft:sponge'`
	Датчик дневного света	`'minecraft:daylight_detector'`
	Дверь из темного дуба	`'minecraft:dark_oak_door'`
	Дверь из тропического дерева	`'minecraft:jungle_door'`

Значок	Название	Идентификатор
	Деревянная кнопка	`'minecraft:oak _ button'`

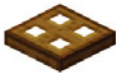
	Деревянная плита	`'minecraft:oak _ slab'`
		`'minecraft:spruce _ slab'`
		`'minecraft:birch _ slab'`
		`'minecraft:jungle _ slab'`
		`'minecraft:acacia _ slab'`
		`'minecraft:dark _ oak _ slab'`

	Динамит	`'minecraft:tnt'`

	Доски (разные)	`'minecraft:oak _ planks'`
		`'minecraft:spruce _ planks'`
		`'minecraft:birch _ planks'`
		`'minecraft:jungle _ planks'`
		`'minecraft:acacia _ planks'`
		`'minecraft:dark _ oak _ planks'`

	Древесина (Бревно)	`'minecraft:oak _ log'`
		`'minecraft:spruce _ log'`
		`'minecraft:birch _ log'`
		`'minecraft:jungle _ log'`
		`'minecraft:acacia _ log'`
		`'minecraft:dark _ oak _ log'`

	Дубовая дверь	`'minecraft:oak _ door'`
	Дубовая нажимная плита	`'minecraft:oak _ pressure _ plate'`
	Дубовый забор	`'minecraft:oak _ fence'`
	Дубовые ступеньки	`'minecraft:oak _ stairs'`

Значок	Название	Идентификатор
	Дубовый люк	`'minecraft:oak _ trap-door'`
	Еловая дверь	`'minecraft:spruce _ door'`
	Еловая калитка	`'minecraft:spruce _ fence _ gate'`
	Еловый забор	`'minecraft:spruce _ fence'`
	Еловые ступеньки	`'minecraft:spruce _ stairs'`
	Железная дверь	`'minecraft:iron _ door'`
	Железная руда	`'minecraft:iron _ ore'`
	Железные прутья	`'minecraft:iron _ bars'`
	Железный блок	`'minecraft:iron _ block'`
	Железный люк	`'minecraft:iron _ trap-door'`
	Желтая глазурованная керамика	`'minecraft:yellow _ glazed _ terracotta'`
	Желтый шалкеровый ящик	`'minecraft:yellow _ shulker _ box'`
	Забор из темного дуба	`'minecraft:dark _ oak _ fence'`
	Забор из тропического дерева	`'minecraft:jungle _ fence'`

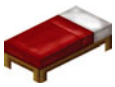
Значок	Название	Идентификатор
	Зараженные блоки (разные)	'minecraft:infested _ stone' 'minecraft:infested _ cobblestone' 'minecraft:infested _ stone _ bricks' 'minecraft:infested _ mossy _ stone _ bricks' 'minecraft:infested _ cracked _ stone _ bricks' 'minecraft:infested _ chiseled _ stone _ bricks'
	Зеленая глазурованная керамика	'minecraft:green _ glazed _ terracotta'
	Зеленый шалкеровый ящик	'minecraft:green _ shulk- er _ box'
	Зельеварка	'minecraft:brewing _ stand'
	Земля	'minecraft:dirt'
	Золотая руда	'minecraft:gold _ ore'
	Золотой блок	'minecraft:gold _ block'
	Изумрудная руда	'minecraft:emerald _ ore'
	Изумрудный блок	'minecraft:emerald _ block'
	Какао	'minecraft:cocoa'
	Кактус	'minecraft:cactus'
	Калитка	'minecraft:fence _ gate'

Значок	Название	Идентификатор
	Калитка из темного дуба	`'minecraft:dark_oak_fence_gate'`
	Калитка из тропического дерева	`'minecraft:jungle_fence_gate'`
	Каменная кнопка	`'minecraft:stone_button'`
	Каменная нажимная плита	`'minecraft:stone_pressure_plate'`
	Каменная плита	`'minecraft:stone_slab'`

	Каменная плита	`'minecraft:stone_brick_slab'`
		`'minecraft:nether_brick_slab'`
		`'minecraft:quartz_slab'`

	Каменные кирпичи	`'minecraft:stone_bricks'`
		`'minecraft:mossy_stone_bricks'`
		`'minecraft:cracked_stone_bricks'`
		`'minecraft:chiseled_stone_bricks'`

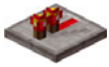
	Каменный уголь	`'minecraft:coal_ore'`
	Камень	`'minecraft:stone'`
	Картофель	`'minecraft:potato'`
	Кварцевые ступеньки	`'minecraft:quartz_stairs'`
	Кварцевый блок	`'minecraft:quartz_block'`
	Керамика (белая)	`'minecraft:white_terracotta'`
	Кирпичи	`'minecraft:bricks'`

Значок	Название	Идентификатор
	Кирпичные ступеньки	'minecraft:brick _ stairs'
	Книжная полка	'minecraft:bookshelf'
	Ковер (белый)	'minecraft:white _ carpet'
	Компаратор	'minecraft:comparator'
	Коричневая глазурованная керамика	'minecraft:brown _ glazed _ terracotta'
	Коричневый гриб	'minecraft:brown _ mushroom'
	Коричневый шалкеровый ящик	'minecraft:brown _ shulker _ box'
	Костный блок	'minecraft:bone _ block'
	Котел	'minecraft:cauldron'
	Красная глазурованная керамика	'minecraft:red _ glazed _ terracotta'
	Красные незеритовые кирпичи	'minecraft:red _ nether _ bricks'
	Красный гриб	'minecraft:red _ mushroom'
	Красный песчаник	'minecraft:red _ sandstone'
	Красный шалкеровый ящик	'minecraft:red _ shulker _ box'
	Кровать (белая)	'minecraft:white _ bed'

Значок	Название	Идентификатор
	Крюк	'minecraft:tripwire _ hook'
	Кувшинка	'minecraft:lily _ pad'
	Лазуритовая руда	'minecraft:lapis _ ore'
	Лазуритовый блок	'minecraft:lapis _ block'
	Лаймовая глазурованная керамика	'minecraft:lime _ glazed _ terracotta'
	Лаймовый шалкеровый ящик	'minecraft:lime _ shulk- er _ box'
	Лед	'minecraft:ice'
	Лестница	'minecraft:ladder'
	Лианы	'minecraft:vine'
	Липкий поршень	'minecraft:sticky _ pis- ton'
	Листья	'minecraft:oak _ leaves' 'minecraft:spruce _ leaves' 'minecraft:birch _ leaves' 'minecraft:jungle _ leaves' 'minecraft:acacia _ leaves' 'minecraft:dark _ oak _ leaves'
	Магма	'minecraft:magma _ block'

Значок	Название	Идентификатор
	Мак	`'minecraft:poppy'`
	Маяк	`'minecraft:beacon'`
	Мертвый куст	`'minecraft:dead _ bush'`
	Мицелий	`'minecraft:mycelium'`
	Морковь	`'minecraft:carrot'`
	Морской фонарь	`'minecraft:sea _ lantern'`
	Наблюдатель	`'minecraft:observer'`
	Наковальня	`'minecraft:anvil'`
	Нарост из Незера	`'minecraft:nether _ wart'`
	Незер-кварцевая руда	`'minecraft:nether _ quartz _ ore'`
	Незер-портал	`'minecraft:nether _ portal'`
	Незерит	`'minecraft:netherrack'`
	Незеритовые кирпичи	`'minecraft:nether _ bricks'`
	Незеритовый забор	`'minecraft:nether _ brick _ fence'`
	Незеритовые ступеньки	`'minecraft:nether _ brick _ stairs'`

Значок	Название	Идентификатор
	Нотный блок	`'minecraft:note_block'`
	Обожженная глина	`'minecraft:hardened_clay'`
	Обсидиан	`'minecraft:obsidian'`
	Огонь	`'minecraft:fire'`
	Одуванчик	`'minecraft:dandelion'`
	Оранжевая глазурованная керамика	`'minecraft:orange_glazed_terracotta'`
	Оранжевый шалкеровый ящик	`'minecraft:orange_shulker_box'`
	Паутина	`'minecraft:cobweb'`
	Пашня	`'minecraft:farmland'`
	Песок	`'minecraft:sand'`
	Песок душ	`'minecraft:soul_sand'`
	Песчаник	`'minecraft:sandstone'`
	Песчаниковые ступеньки	`'minecraft:sandstone_stairs'`
	Печь	`'minecraft:furnace'`
	Плита из красного песчаника	`'minecraft:smooth_red_sandstone'`

Значок	Название	Идентификатор
	Плотный лед	`'minecraft:packed _ ice'`
	Повторитель	`'minecraft:repeater'`
	Портал в Энд	`'minecraft:end _ portal'`
	Поршень	`'minecraft:piston'`
	Призмарин	`'minecraft:prismarine'`
	Проигрыватель	`'minecraft:jukebox'`
	Пурпур	`'minecraft:purpur _ block'`
	Пурпурная глазурованная керамика	`'minecraft:magenta _ glazed _ terracotta'`
	Пурпурный шалкеровый ящик	`'minecraft:magenta _ shulker _ box'`
	Пурпуровая плита	`'minecraft:purpur _ slab'`
	Пурпуровые ступеньки	`'minecraft:purpur _ stairs'`
	Пурпуровый пилон	`'minecraft:purpur _ pillar'`
	Пшеница	`'minecraft:wheat'`
	Раздатчик	`'minecraft:dispenser'`
	Рамка портала в Энд	`'minecraft:end _ portal _ frame'`
	Рассадник	`'minecraft:mob _ spawner'`

Значок	Название	Идентификатор
	Растение хоруса	`'minecraft:chorus _ plant'`
	Редстоуновая руда	`'minecraft:redstone _ ore'`
	Редстоуновый блок	`'minecraft:redstone _ block'`
	Редстоуновый факел (активный)	`'minecraft:redstone _ torch'`
	Редстоуновый факел (неактивный)	`'minecraft:redstone _ wall _ torch'`
	Редстоуновый фонарь	`'minecraft:redstone _ lamp'`
	Рельсы	`'minecraft:rail'`
	Рельсы с датчиком	`'minecraft:detector _ rail'`
	Розовая глазурованная керамика	`'minecraft:pink _ glazed _ terracotta'`
	Розовый шалкеровый ящик	`'minecraft:pink _ shulker _ box'`
	Рычаг	`'minecraft:lever'`

	Саженец (разные)	`'minecraft:oak _ sapling'`
		`'minecraft:spruce _ sapling'`
		`'minecraft:birch _ sapling'`
		`'minecraft:jungle _ sapling'`
		`'minecraft:acacia _ sapling'`
		`'minecraft:dark _ oak _ sapling'`

Значок	Название	Идентификатор
	Сахарный тростник	`'minecraft:sugar _ can'`
	Свекла	`'minecraft:beetroot'`
	Светильник Джека	`'minecraft:jack _ o _ lan-tern'`
	Светокамень	`'minecraft:glowstone'`
	Светящаяся редстоуновая руда	`'minecraft:lit _ red-stone _ ore'`
	Серая глазурованная керамика	`'minecraft:gray _ glazed _ terracotta'`
	Серая глазурованная керамика	`'minecraft:light _ gray _ glazed _ terracotta'`
	Серый шалкеровый ящик	`'minecraft:gray _ shulk-er _ box'`
	Серый шалкеровый ящик	`'minecraft:light _ gray _ shulker _ box'`
	Синий лед	`'minecraft:blue _ ice'`
	Синий шалкеровый ящик	`'minecraft:blue _ shulk-er _ box'`
	Синяя глазурованная керамика	`'minecraft:blue _ glazed _ terracotta'`
	Снег	`'minecraft:snow'`
	Сноп сена	`'minecraft:hay _ block'`
	Стекло	`'minecraft:glass'`

Значок	Название	Идентификатор
	Стекло (белое)	`'minecraft:white _ stained _ glass'`
	Стеклянная панель	`'minecraft:glass _ pane'`
	Стеклянная панель (белая)	`'minecraft:white _ stained _ glass _ pane'`
	Стержень Энда	`'minecraft:end _ rod'`
	Стоячая вода	`'minecraft:water'`
	Стоячая лава	`'minecraft:lava'`
	Ступеньки из каменного кирпича	`'minecraft:stone _ brick _ stairs'`
	Ступеньки из красного песчаника	`'minecraft:red _ sandstone _ stairs'`
	Ступеньки из темного дуба	`'minecraft:dark _ oak _ stairs'`
	Ступеньки из тропического дерева	`'minecraft:jungle _ stairs'`
	Сундук	`'minecraft:chest'`
	Сундук-ловушка	`'minecraft:trapped _ chest'`
	Табличка	`'minecraft:sign'`
	Текущая вода	`'minecraft:flowing _ water'`
	Текущая лава	`'minecraft:flowing _ lava'`

Значок	Название	Идентификатор
	Торт	'minecraft:cake'
	Трава	'minecraft:grass'
	Тропа	'minecraft:grass _ path'
	Тыква	'minecraft:pumpkin'
	Угольный блок	'minecraft:coal _ block'
	Факел	'minecraft:torch'
	Фиолетовая глазурованная керамика	'minecraft:purple _ glazed _ terracotta'
	Фиолетовый шалкеровый ящик	'minecraft:purple _ shulker _ box'
	Флаг (белый)	'minecraft:white _ banner'
	Цветок хоруса	'minecraft:chorus _ flower'
	Цветочный горшок	'minecraft:flower _ pot'
	Цемент (белый)	'minecraft:white _ concrete _ powder'
	Чародейский стол	'minecraft:enchanting _ table'
	Черная глазурованная керамика	'minecraft:black _ glazed _ terracotta'
	Черный шалкеровый ящик	'minecraft:black _ shulker _ box'

Значок	Название	Идентификатор
	Шерсть (белая)	`'minecraft:white _ wool'`
	Эндер-сундук	`'minecraft:ender _ chest'`
	Эндерняк	`'minecraft:end _ stone'`
	Эндерняковые кирпичи	`'minecraft:end _ stone _ bricks'`
	Энергорельсы	`'minecraft:powered _ rail'`
	Яйцо дракона	`'minecraft:dragon _ egg'`

ЗАГРУЗКА ПРИМЕРОВ

Загрузить программы можно прямо в интерфейсе черепашки с сайта **pastebin.com**. Для этого нужно следовать инструкциям, приведенным в разделе «**Публикация и загрузка программ в Интернете**» главы 4. На сайте **turtleappstore.com** можно найти как программы из этой книги, так и приложения, созданные другими пользователями.

Требования

Вот что тебе потребуется, чтобы следовать указаниям этой книги. (Программное обеспечение используется одинаковое, независимо от того, какая операционная система установлена на твоем компьютере – Windows, macOS или Linux.)

- Официальная платная версия игры Minecraft доступна по адресу **minecraft.net**.
- Пакет Java доступен для бесплатной загрузки по адресу **www.java.com/ru/download/**.
- Программное обеспечение ATLauncher доступно для бесплатного скачивания по адресу **www.atlauncher.com**.
- Модуль ComputerCraft доступен для бесплатного скачивания по адресу **www.computercraft.info**.

Подробные инструкции по установке приведены в главе 1.

Также тебе понадобится компьютер под управлением операционной системы Windows 7 или macOS 10.10 либо более поздней версии. Мод ComputerCraft несовместим с версиями игры Minecraft Pocket Edition или Minecraft Raspberry Pi Edition.

АЛФАВИТНЫЙ УКАЗАТЕЛЬ

КОГДА ВЫ ДАРИТЕ КНИГУ, ВЫ ДАРИТЕ ЦЕЛЫЙ МИР

ХОТИТЕ ЗНАТЬ БОЛЬШЕ?

Заходите на сайт:
https://eksmo.ru/b2b/

Звоните по телефону:
+7 495 411-68-59, доб. 2261

ВАШ ЛОГОТИП
НА ОБЛОЖКЕ

ВАШ ЛОГОТИП НА КОРЕШКЕ

ОБРАЩЕНИЕ
К КЛИЕНТАМ
НА ОБЛОЖКЕ

ПРОГРАММИРОВАНИЕ В MINECRAFT

ПУСТЬ АРМИЯ РОБОТОВ СТРОИТ, ВЫРАЩИВАЕТ, КОПАЕТ И ДЕЛАЕТ ВСЮ СКУЧНУЮ РАБОТУ ЗА ТЕБЯ!

Ты добываешь алмазы, крафтишь инструменты, строишь разные сооружения. А что, если бы армия запрограммированных роботов делала все это вместо тебя, причем в любое время дня и ночи? Ага, тебе понравилась эта идея – тогда скорее приступай к чтению этой книги!

НА ПРИМЕРАХ ИЗ ЯЗЫКА LUA

Благодаря этой книге ты узнаешь:

- Что можно сделать в бесплатном моде ComputerCraft
- Как запрограммировать роботов, принимающих умные решения самостоятельно
- Как с помощью кода заставить ботов выращивать что угодно – от пшеницы до кактусов
- Как сделать фабрику, производящую стройматериалы постоянно

И множество других крутых способов прокачать себя в Minecraft!

Об авторе

Эл Свейгарт – разработчик программ и преподаватель программирования для детей и взрослых, а также автор нескольких бестселлеров, посвященных программированию для начинающих.

«Отличная книга, показывающая, что увлечение игрой можно направить в обучающее русло. Интересно, что в книге все объясняется на интересных практических примерах, актуальных для детей».

Илья Высоцкий,
ведущий методист учебного направления Minecraft школы цифрового творчества «Кодабра»

В КОНЦЕ КАЖДОЙ ГЛАВЫ ТЕБЯ ЖДУТ БОНУСНЫЕ ЗАДАНИЯ, КОТОРЫЕ ПОМОГУТ ВЫВЕСТИ ТВОИ КОДЕРСКИЕ НАВЫКИ НА НОВЫЙ УРОВЕНЬ.

no starch press
www.nostarch.com

ISBN 978-5-04-096471-0

БОМБОРА

Бомбора — это новое название Эксмо Non-fiction, лидера на рынке полезных и вдохновляющих книг. Мы любим книги и создаем их, чтобы вы могли творить, открывать мир, пробовать новое, расти. Быть счастливыми. Быть на волне.

bomborabooks
www.bombora.ru