СДЕЛАТЬ ИГРУ – ПРОЩЕ ПРОСТОГО

ПРОГРАММИРОВАНИЕ В ROBLOX

ЗАНДЕР БРАМБО

ДМК ПРЕСС ИЗДАТЕЛЬСТВО

Зандер Брамбо

Программирование в Roblox. Сделать игру – проще простого

Coding Roblox Games Made Easy

The ultimate guide to creating games with Roblox Studio and Lua programming

Zander Brumbaugh

Packt›
BIRMINGHAM – MUMBAI

Программирование в Roblox. Сделать игру – проще простого

Создание игр с помощью Roblox Studio
и языка программирования Lua от «А» до «Я»

Зандер Брамбо

Москва, 2022

УДК 681.3.07
ББК 32.973.26–018.2.75
Б87

Б87 Зандер Брамбо

Программирование в Roblox. Сделать игру – проще простого: Создание игр с помощью Roblox Studio и языка программирования Lua от «А» до «Я» / пер. с англ. М. А. Райтмана. – М.: ДМК Пресс, 2022. – 198 с.: ил.

ISBN 978-5-97060-982-8

В этой книге описывается работа на развлекательной платформе Roblox – от программирования в Roblox Lua до создания игр в жанре обби и «Королевская битва». Подробно рассмотрены возможности Roblox Studio, приёмы изменения настроек игры, сценарии программирования. Читатель сможет выполнить практические упражнения, используя примеры кода, и узнает, как достичь максимальной популярности игры путем внедрения хорошей механики, монетизации и маркетинговых методов.

Книга предназначена для всех, кто интересуется разработкой игр на Roblox, – как новичков, так и тех, кто знаком с платформой и хочет углубить навыки ее использования.

ISBN 978-1-80056-199-1 (англ.)
ISBN 978-5-97060-982-8 (рус.)

Посвящается моим матери и отцу.
Вы всегда рядом с мной, во всем поддерживаете меня
и постоянно вдохновляете становиться лучшей версией себя.

– Зандер Брамбо (Zander Brumbaugh)

Оглавление

Предисловие от издательства

Отзывы и пожелания

Мы всегда рады отзывам наших читателей. Расскажите нам, что вы думаете об этой книге, – что понравилось или, может быть, не понравилось. Отзывы важны для нас, чтобы выпускать книги, которые будут для вас максимально полезны.

Вы можете написать отзыв на нашем сайте **www.dmkpress.com**, зайдя на страницу книги и оставив комментарий в разделе «Отзывы и рецензии». Также можно послать письмо главному редактору по адресу **dmkpress@gmail.com**; при этом укажите название книги в теме письма.

Если вы являетесь экспертом в какой-либо области и заинтересованы в написании новой книги, заполните форму на нашем сайте по адресу **http://dmkpress.com/authors/publish_book/** или напишите в издательство по адресу **dmkpress@gmail.com**.

Список опечаток

Хотя мы приняли все возможные меры для того, чтобы обеспечить высокое качество наших текстов, ошибки все равно случаются. Если вы найдете ошибку в одной из наших книг – возможно, ошибку в основном тексте или программном коде, – мы будем очень благодарны, если вы сообщите нам о ней. Сделав это, вы избавите других читателей от недопонимания и поможете нам улучшить последующие издания этой книги.

Если вы найдете какие-либо ошибки в коде, пожалуйста, сообщите о них главному редактору по адресу **dmkpress@gmail.com**, и мы исправим это в следующих тиражах.

Нарушение авторских прав

Пиратство в интернете по-прежнему остается насущной проблемой. Издательства «ДМК Пресс» и Packt Publishing очень серьезно относятся к вопросам защиты авторских прав и лицензирования. Если вы столкнетесь в интернете с незаконной публикацией какой-либо из наших книг, пожалуйста, пришлите нам ссылку на интернет-ресурс, чтобы мы могли применить санкции.

Ссылку на подозрительные материалы можно прислать по адресу электронной почты **dmkpress@gmail.com**.

Мы высоко ценим любую помощь по защите наших авторов, благодаря которой мы можем предоставлять вам качественные материалы.

Об авторе

Зандер Брамбо – программист, менеджер проектов и разработчик игр. В игры, которые были созданы им самим или при его участии, играло более 200 млн человек. Зандер активно взаимодействует с сообществом Roblox и командой по связям с разработчиками Roblox, содействуя в продвижении и поддерживая платформу Roblox. Он является автором популярных игр, таких как Power **Simulator**, **Munching Masters** и **Magicial Simulator**. Благодаря своим наработкам в создании игр он окончил колледж, а в данный момент учится в Вашингтонском университете и является частью Школы компьютерных наук и инженерии Пола Г. Аллена. На момент написания этой книги Зандеру 18 лет.

О рецензентах

Эндрю Береза – человек с глубоким стратегическим мышлением и специалист по решению проблем, обладающий особым умением выявлять закономерности и связи в абстрактных и запутанных системах. Он занимается разработкой игр на всех уровнях, от творческого руководства и дизайна уровней до пользовательского интерфейса и программирования. Является признанным экспертом по платформе Roblox с более чем 10-летним опытом разработки и создал 3-месячный акселератор (2017 г.) и 5-месячный инкубатор (2018 г.), размещенный в корпоративной штаб-квартире Roblox в Сан-Матео, Калифорния. На платформе Roblox к числу его игр относятся **2PGFT**, **Miner's Haven**, **Azure Mines**, **War Games**, **Vesteria** и **Build Island**. Еще до Roblox он много работал с играми, создавая читы для картриджей Super Mario World и моды для Minecraft.

Ян Ханф, также известный на платформе Roblox как **Hanfian**, занимается созданием игр с 2012 года. Он учился в колледже Тихоокеанского университета и за 5 лет получил степень магистра компьютерных наук. Основным направлением его работы были графика и симуляции. В 2019 году Ян работал над популярной в Roblox игрой Anime Fighting Simulator, которая завоевала международный успех. В настоящее время Ян проживает в Калифорнии и борется с раком мозга четвертой стадии, но надеется продолжить делать игры в будущем.

Благодарности

Я благодарен за возможность поработать вместе с Джулиусом Квиндипаном, известным в Roblox под ником MarmDev, и всей командой, которая создавала Anime Fighting Simulator.

И последнее, но не по значению: я хотел бы поблагодарить свою маму, чья карьера в области компьютерных наук вдохновила меня пойти по ее стопам.

Предисловие

По мере чтения этой книги вы получите практический опыт работы на платформе Roblox. Мы начнем с обзора разработки Roblox, а затем научимся работать с Roblox Studio. По мере продвижения вы узнаете все необходимое: от программирования в Roblox Lua до создания игр в жанре обби и «Королевская битва». Ближе к концу мы рассмотрим вопросы разработки игр, поговорим о том, как достигнуть максимальной популярности игры путем внедрения хорошей механики, монетизации и маркетинговых методов.

После изучения этого руководства Roblox у вас появятся навыки, необходимые для работы в команде или даже для руководства над командой, и вы сможете создавать игровые миры из своих фантазий и позволите игрокам по всему миру окунуться в них.

Для кого предназначена эта книга

Эта книга предназначена для всех, кто интересуется разработкой игр на платформе Roblox, а также для тех, кто уже знаком с Roblox и хочет изучить лучшие рекомендации, приемы и практики для разработки в Roblox.

О чем эта книга расскажет

В *главе 1* мы познакомимся с основными концепциями разработки Roblox, возможностями разработки на этой платформе, с тем, как зарабатывать деньги на играх и чего стоит ожидать с учетом опыта проектов других разработчиков.

В *главе 2* мы будем учиться работать в Roblox Studio. Рассмотрим основные элементы управления, такие как движение и управление камерой, взаимодействие с экземплярами в рабочем пространстве, как использовать бесплатные ресурсы и редактировать информацию об игре.

Для изучения *главы 3* вам не понадобится знание языков программирования. Вы узнаете, как программировать в Roblox Lua, начиная с простых инструкций типа `print("Hello, world")` и заканчивая универсальными конструкциями программирования.

В *главе 4* акцент делается на сценариях программирования, специфичных для Roblox, с которыми люди, даже обладающие общими знаниями в области программирования, незнакомы.

В *главе 5* мы применим все уже приобретенные знания и создадим простую, но рабочую и полноценную игру. В качестве жанра выберем обби, и для создания игры вам нужно будет использовать переменные, события, функции и свойства.

В *главе 6* вам нужно будет собрать воедино все, чему вы научились, чтобы создать игру в жанре «Королевская битва». Вам нужно будет воспользоваться всем, что вы узнали из книги, и одновременно с этим вы получите новые знания в области безопасности и организации. К концу этой главы вы прочувствуете весь процесс создания игры с нуля.

Глава 7 – это наиболее обширная часть книги, посвященная вопросам, не связанным непосредственно с программированием. В этой главе основное внимание будет уделено расширению тех навыков, которые позволят вам стать не только программистом, но и специалистом по продвижению игр, для чего мы воспользуемся тремя «М»: Механикой, Монетизацией и Маркетингом.

Как извлечь из книги максимум

Для выполнения упражнений из этой книги вам потребуется следующее аппаратное и программное обеспечение:

Аппаратное и программное обеспечение, рассматриваемое в книге	Необходимая операционная система
Roblox Studio	Windows, macOS, Chrome OS
Roblox Player	Windows, macOS, Chrome OS

Полный список системных требований для всех компонентов Roblox можно найти по следующей ссылке: **https://en.help.roblox.com/hc/en-us/articles/203312800**.

Если вы используете электронную версию этой книги, мы советуем вам набирать код примеров самостоятельно или взять его из нашего репозитория GitHub (ссылка доступна в следующем разделе). Это поможет вам избежать возможных ошибок, связанных с копированием и вставкой кода.

Загрузка примеров кода

Вы можете загрузить примеры кода для этой книги на сайте GitHub по адресу **https://github.com/PacktPublishing/Coding-Roblox-Games-Made-Easy**. В случае обновления кода вы увидите изменения в файлах в репозитории GitHub.

Чтобы скачать файлы кода, выполните следующие действия:

1) перейдите по указанной ссылке на сайте GitHub;
2) нажмите кнопку **Code** и выберите пункт **Download ZIP**.

Загрузив файл архива, распакуйте его, используя последнюю версию следующего ПО:

- WinRAR/7-Zip для Windows;
- Zipeg/iZip/UnRarX для Mac;
- 7-Zip/PeaZip для Linux.

Загрузите цветные изображения

Мы также предоставляем файл PDF с цветными изображениями скриншотов/диаграмм, использованных в этой книге. Вы можете скачать его по ссылке: **https://static.packt-cdn.com/downloads/9781800561991_ColorImages.pdf.**

Цветные изображения также можно найти здесь: **https://github.com/PacktPublishing/Coding-Roblox-Games-Made-Easy/tree/main/Color%20Images**

Условные обозначения

В этой книге используется несколько специальных обозначений.

`Моноширинный шрифт`: фрагменты кода в тексте, имена таблиц базы данных, имена папок и файлов, расширения файлов, пути, URL-адреса, пользовательский ввод и ссылки Twitter. Пример: «Когда игрок присоединяется к игре, его данные нужно добавить в словарь `sessionData`».

Блок кода оформляется следующим образом:

```
dataMod.removeSessionData = function(player)
    local key = player.UserId
    sessionData[key] = nil
end
```

Полужирный текст: обозначает новый термин, важное слово или слова, которые вы видите на экране. Это могут быть, например, элементы меню или фразы в диалоговых окнах. Например: «Чтобы создать пользовательский интерфейс для игры, перейдите к службе **StarterGui** на панели **Explorer**».

> **Советы или важные примечания**
>
> ...выглядят вот так.

Отзывы

Пожалуйста, оставьте отзыв о книге. Почему бы не оставить отзыв на сайте, где вы приобрели книгу, после ее прочтения? Благодаря этому потенциальные читатели смогут увидеть ваше непредвзятое мнение и использовать его для принятия будущего решения о покупке, мы в издательстве сможем понять, что вы думаете о наших продуктах, а наши авторы увидят ваши отзывы о своем творении. Спасибо!

Часть I

Введение в разработку на Roblox

В этой части мы познакомимся с основными концепциями разработки на Roblox. Затем изучим принципы работы в ПО Roblox Studio, его возможности в разработке и увидим, чего стоит ожидать от первых проектов.

Эта часть состоит из следующих глав:

- *глава 1 «Введение в разработку на Roblox»*;
- *глава 2 «Знакомство с рабочей средой».*

Глава 1

Введение в разработку на Roblox

Roblox – это крупнейшая развлекательная платформа, не имеющая аналогов как для игроков, так и для создателей игр. На Roblox заходит более 100 млн активных пользователей в месяц, что позволяет как новым, так и опытным разработчикам создавать успешные игры, в которые можно играть по всему миру бесплатно. Благодаря множеству ресурсов и огромному дружелюбному сообществу разработчиков со всего мира, с которыми можно связаться, выход в сферу разработки игр Roblox – это возможность, которую нельзя найти больше нигде.

Перед тем как окунуться в технические детали работы с Roblox Studio и программирования на Lua, нам сперва нужно ознакомиться с тем, что вообще может дать вам разработка игр на Roblox и из чего обычно состоит повседневное взаимодействие разработчика с платформой. К концу этой главы у вас будет более полное представление о различных типах ролей, которые разработчики могут выполнять в игровых проектах, о том, как становление разработчиком Roblox может принести вам финансовую выгоду и чего ожидать после разработки и публикации своих первых игр.

В этой главе мы рассмотрим следующие основные темы:

- узнаем о преимуществах разработки на Roblox;
- выделим несколько типов разработчиков;
- рассмотрим перспективы первых проектов.

Технические требования

Для изучения этого раздела вам не потребуется никакого программного обеспечения или дополнительных материалов, так как в основном здесь будет информация и теория. Возможно, вам будет полезно иметь под руками компьютер с доступом в интернет, чтобы вы могли зайти на некоторые сайты или в приложения, обсуждаемые в данной главе.

Преимущества разработки на Roblox

Платформа Roblox непрерывно развивается и растет с момента ее создания в 2006 году, и в последние годы этот рост значительно ускорился. По состоянию на 2020 год в Roblox ежемесячно играют более 100 млн уникальных пользователей. Сейчас – лучший момент, чтобы стать разработчиком Roblox, так как множество новых игроков ищут новые игры от новых авторов.

Финансовые возможности в Roblox

Вероятно, одним из самых больших факторов, способствующих общему росту популярности Roblox среди разработчиков, является возможная финансовая выгода. Может быть, именно это и есть один из ваших основных мотивов к изучению данной платформы, и в этом нет ничего удивительного.

Сейчас самые популярные игры на платформе Roblox ежегодно приносят разработчикам десятки миллионов долларов США на внутриигровых покупках.

Как разработчик вы можете зарабатывать деньги на играх с помощью процесса под названием **Developer Exchange**, или, если кратко, **DevEx**. В Roblox есть виртуальная валюта под названием **робаксы**, которую можно покупать за реальные деньги. Когда игрок покупает робаксы, баланс его учетной записи пополняется, и он может свободно тратить валюту на любую игру, которую захочет. Когда игрок тратит робаксы на одну из внутриигровых покупок, 70 % робаксов от этой покупки получает разработчик игры, а Roblox берет 30%-ную комиссию.

Выделим два типа внутриигровых покупок:

- **игровые пропуски, или гейм-пассы**: разовые покупки;
- **продукты для разработчиков**: покупаются несколько раз за что-то вроде внутриигровой валюты.

Вы должны накопить в общей сложности 100 000 робаксов и быть старше 13 лет, чтобы иметь право на участие в программе DevEx. В дополнение к продажам, которые вы делаете напрямую, поклонники ваших игр, у которых есть премиальная подписка Roblox, приносят вам дополнительные робаксы в зависимости от того, сколько времени они проводят в вашей игре, хотя эта сумма обычно составляет лишь часть дохода. Кроме того, есть некоторые закрытые программы, в которые можно попасть только по приглашению и которые позволяют вам продавать за робаксы аксессуары для персонажей, плагины и многое другое. Однако скоро эти программы планируется сделать общедоступными.

Работая в команде разработчиков в группе Roblox, разработчики могут получать доход в робаксах напрямую либо через процент от дохода от игры. Прямые выплаты организуются просто, и их можно настроить на странице

Configure Group, но платежи не регулярны. Выплата разработчикам процента от дохода от игры выполняется автоматически после проверки. Важно помнить, что такая выплата возможна только в том случае, если игра размещена в группе, а не в личном профиле.

Важная заметка

После того как игрок совершает внутриигровую покупку, Roblox выдерживает 3-дневный период ожидания, прежде чем эти робаксы будут перечислены в групповой или личный аккаунт, чтобы убедиться, что покупка была законной.

Больше информации о программе обмена разработчиками можно найти здесь: **https://www.roblox.com/developer-exchange/help**.

Развитие профессиональных навыков

Помимо финансовой прибыли, разработчик Roblox получает уникальную возможность развивать другие свои навыки, которые могут оказаться полезными в иных профессиональных средах. Независимо от того, выполняете ли вы роль менеджера проекта или обычного программиста, у вас будут развиваться навыки работы в команде и общения. Одно из самых востребованных качеств, которое ищут работодатели, особенно в STEM, где бóльшая часть работы выполняется в команде, – это способность координировать свою работу и общаться внутри команды. Разработка на Roblox, на мой взгляд, является одним из лучших мест для студентов, изучающих информатику, где можно начать осваивать принципы совместной работы. Платформа помогает развить не только способности в программировании, но и лидерские качества и, в конечном итоге, навыки управления финансами.

Преимущества совместной работы

В целом работать с другими людьми не всегда обязательно, и существуют популярные игры, созданные в одиночку. Но в Roblox сама система настоятельно побуждает разработчиков работать над играми вместе, при этом каждый разработчик имеет в проекте одну или несколько ролей. В настоящее время почти все топовые игры в разделе **Popular** были созданы командой из двух или более человек. Благодаря большому успеху и последующему росту команды разработчиков некоторых игр выросли до более чем 20 человек, хотя это пока редкое явление.

Лучший способ присоединиться к сообществу и найти других разработчиков для сотрудничества – это Twitter или Discord. Имея в Twitter аккаунт, посвященный вашей разработке, вы можете публиковать свои лучшие творения и общаться с другими, более популярными разработчиками. Новые

связи с другими разработчиками в **Roblox Twitter Community (RTC)** могут дать вам возможность поработать с более известными людьми и набрать популярность. Discord – это приложение для общения, похожее на **Slack**. В Discord есть множество серверов, ориентированных на разработку в Roblox, и на них вы можете поделиться своими трудами, обсудить работу с другими членами сообщества или найти новых людей, с которыми можно объединиться.

Не менее важную роль в сообществе разработчиков играют ютуберы, создающие контент по Roblox. Когда пользователи YouTube создают контент, демонстрирующий ваши игры, это привлекает аудиторию, и количество ваших игроков, скорее всего, увеличится благодаря такой рекламе. Контакт с такими людьми может обеспечить продвижение ваших проектов в будущем, а также найти единомышленников. С создателями контента не всегда просто связаться, но они часто сидят в Discord или Twitter, а также отвечают на запросы по электронной почте.

В целом описанный здесь тип сетевого взаимодействия одинаков во многих сферах деятельности. Создание имиджа, популярности и репутации играет ключевую роль в вашей карьере. Законное и профессиональное выполнение вашей работы тоже приносит плоды, но не столь быстро. А коммуникативные способности позволят вам легче общаться с новыми людьми, которые могут принести пользу как вашей работе, так и репутации.

Типы разработчиков

Как упоминалось ранее, сообщество разработчиков Roblox весьма разнообразно. Каждый разработчик приносит на платформу свой уникальный стиль и технику. В разработке игр есть множество ролей, которые может выполнять каждый член команды. Чаще всего в Roblox у каждого разработчика имеется один основной навык, в рамках которого он работает в проекте. Наиболее распространенными типами разработчиков в Roblox являются программисты, дизайнеры уровней, разработчики 3D-моделей, дизайнеры UI/UX (интерфейса) и различные художники. Каждый разработчик, выполняющий свою роль, одинаково важен для создания качественного готового продукта.

Программисты

Программисты создают ядро любой игры. Программист реализует все: от хранения данных игрока и создания работающего оружия до любых более сложных и редких игровых функций. В Roblox программисты используют язык Lua, быстрый процедурный язык на основе C, который обычно применяется в индустрии разработки игр. Вы увидите, что синтаксис языка Lua проще других языков и более ориентирован на человека, чем большинство языков высокого уровня. Из-за этого многие программисты

считают кривую обучения довольно плавной, и если вы начинаете изучать программирование с Lua, в будущем переход на другие языки будет прост, поскольку в нем используется общий синтаксис из нескольких языков программирования.

Моделисты

3D-моделисты создают все, что вы видите в игре: от мебели, домашних животных и предметов до любых других визуальных элементов разного размера. Модели можно создавать прямо в Roblox Studio, но большинство 3D-моделей создают в бесплатном приложении под названием Blender. Тому есть много причин, в частности детали, используемые для создания моделей внутри Roblox Studio, блочные, с их помощью не удастся получить гладкие или сложные формы, а в специальном ПО для моделирования это делается легко.

Дизайнеры уровней

Дизайнеры уровней создают миры ваших игр. Будь то холодный коридор в дрейфующем космическом корабле или жаркая засушливая пустыня с далеким оазисом – именно дизайнеры уровней создают первое впечатление об игре, когда игроки присоединяются к ней.

Важно, чтобы дизайнер уровней проекта был достаточно опытным, чтобы он умел отразить желаемое видение в игре. Может показаться, что дизайнеры уровней и моделисты – это одно и то же, но это не так. Дизайнеры уровней больше сосредоточиваются на общей карте и большом мире для проектов, что, впрочем, не мешает им использовать Blender и создавать 3D-модели отдельных ресурсов.

Дизайнеры UI/UX

Дизайнеры UI/UX (или дизайнеры интерфейса) создают экраны, с которыми взаимодействуют игроки внутри вашей игры. Например, дизайнер пользовательского интерфейса создает экран инвентаря игрока, полосу здоровья и другие видимые элементы интерфейса.

Зачастую UI – это первое, что игрок увидит в вашей игре, поэтому важно, чтобы дизайнер мог создать визуально привлекательный интерфейс, соответствующий стилю самой игры.

Если и другие типы разработчиков – аниматоры, музыкальные продюсеры, графические дизайнеры и иные художники. Все перечисленные типы разработчиков важны для создания сильного, законченного продукта, и для успеха проекта важно, чтобы все они были одинаково компетентны в своей сфере. Вы должны определить, какие из предыдущих ролей вас интересуют больше всего, чтобы затем узнать о них больше и развить соответствующие навыки для будущего использования.

Чего ожидать от ранних проектов

Начало карьеры разработчика у всех разное, но уверенно можно сказать одно: ваш первый игровой проект не будет номером один на странице популярных игр, и это нормально! Как бы обидно это ни звучало, золотые горы не свалятся на вас легко и просто. Мой личный опыт подсказывает, что поддерживать популярную игру, в которую одновременно играют десятки тысяч игроков (или больше), довольно сложно. Новым разработчикам сперва нужно набраться опыта и привыкнуть к платформе, и лишь потом начать создавать популярные игры. На рис. 1.1 изображена миниатюра моей первой игры под названием **Endure**. Здесь невооруженным глазом видно, что игра сделана любителем.

Рис. 1.1. Endure была одной из моих первых игр, и она требовала явной доработки

Многие начинающие разработчики часто сталкиваются с препятствиями, когда начинают работу над проектом с огромным азартом, но слабым техническим пониманием, и впоследствии оказываются вынуждены отказаться от проекта, так как не удается реализовать все задуманное. Лучший способ обойти эти проблемы – составить план разработки и задокументировать функции и механики, которые должны быть включены в ваш проект. В процессе вы можете пересматривать и реструктурировать свое видение по мере необходимости, учитывая, что популярно среди игроков Roblox и что вы вообще можете реализовать с имеющимися способностями команды.

Важнее всего, конечно, мотивация и преданность делу, ибо без них проект так и останется в столе. Важно помнить, что игры не всегда доставляют удовольствие игрокам, если их создание не приносит удовольствия разработчикам. Работая с другими людьми, убедитесь, что все в команде согласны с направлением и примерной дорожной картой разработки игры. Оптимальная цель как для вашего психического здоровья, так и для качества ваших проектов – делать каждую новую игру лучше, чем предыдущая. Вы можете потратить месяцы на разработку довольно посредственной игры, но это будет необходимая для будущей работы основа. Каждый новый проект дает вам немного больше опыта и признания. На рис. 1.2 показан прямой результат применения этих методов – мою игру **Power Simulator** запустили более 100 млн раз, а ее миниатюра гораздо более привлекательна для потенциальных игроков благодаря профессиональному дизайну.

Рис. 1.2. Power Simulator – это моя самая успешная на сегодняшний день игра и результат двухлетнего опыта

Какая бы судьба ни ожидала ваши ранние проекы, важно всегда смотреть в будущее и пытаться улучшить игровой опыт для ваших игроков и процесс разработки для себя. Только работой и стараниями можно достичь успеха.

Резюме

Прочитав эту главу, вы должны запомнить несколько ключевых моментов: какие возможности существуют на платформе Roblox, чем занимаются разные типы разработчиков и как найти в сообществе людей, с которыми можно будет создавать совместные проекты.

Теперь, когда вы узнали много нового о работе в сети, о том, чего ожидать от ранней разработки, а также о том, как начать развиваться как разработчик, я верю, что вы сумеете совершить удивительные вещи на платформе

Roblox и развить навыки, которые пригодятся вам в любой профессиональной среде.

В следующей главе мы начнем знакомиться с Roblox Studio, программой, в которой вы будете создавать свои игры. Знание всех особенностей вашей рабочей среды, в том числе тех, которые сперва могут быть незаметны, поможет повысить эффективность вашей разработки и общую производительность в будущих проектах.

ЛАНЬ®

Глава 2

Знакомство с рабочей средой

Полноценное владение инструментами рабочей среды позволяет вам как разработчику более эффективно продвигать свои проекты. Знакомство с интерфейсами, с помощью которых разработчики Roblox ежедневно создают игры, поможет вам создать свои собственные.

Начиная с этой главы вы научитесь создавать в вашем профиле **плейсы** для игры, изменять внешние настройки игры, работать в Roblox Studio и пользоваться дополнительными ресурсами, которые предлагает вам Roblox. Вы также научитесь пользоваться многими небольшими, но удобными функциями этой программы.

В данной главе мы рассмотрим следующие темы:

- страница создания игры;
- начало работы с Roblox Studio;
- знакомство с плагинами и панелью инструментов.

Технические требования

Для изучения этой главы вам потребуется доступ к компьютеру, соответствующий требованиям для запуска Roblox Studio. Чтобы использовать большинство функций Roblox Studio, вам также нужно будет стабильное подключение к интернету. Более подробную информацию о системных требованиях Roblox можно найти по ссылке: **https://en.help.roblox.com/hc/en-us/articles/203312800.**

Страница создания игры

На вкладке **Create** на веб-сайте Roblox отображаются все созданные вами игры и загруженные ресурсы, в порядке от самых новых к старым. Вкладки на этой странице позволяют выбирать различные ресурсы. Кроме того, здесь вы можете выполнять многие действия для изменения различных общих настроек игры. Чтобы попасть на страницу **Create**, перейдите по ссылке **https://www.roblox.com/develop** – и попадете на страницу, изображенную на рис. 2.1.

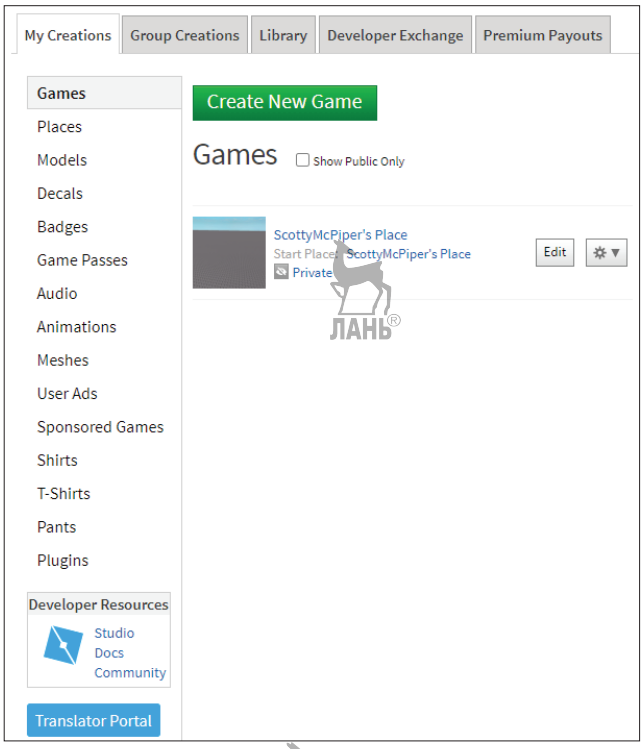


Рис. 2.1. На странице **Create** отображается все, что связано с разработкой

Чтобы продолжить работу с этой главой, вам необходимо перейти на вкладку **Create**. На этой вкладке найдите игру по умолчанию, которая была создана при активации вашей учетной записи. Если по какой-либо причине на вкладке **Create** списка игр нет, нажмите кнопку **Create New Game**, прокрутите появившееся окно вниз и нажмите кнопку **Create Game**. После этого вы можете вернуться на страницу **Create** и продолжить. Теперь все готово к изучению того, как настроить внешние параметры вашей игры и использовать другие ресурсы на странице **Create**.

Настройка параметров игры и плейса

Наверняка вы задумывались о том, что в игре есть определенные настройки, которые нужно менять, например возможное количество игроков на сервере или платформы, с которых можно играть. Для таких настроек не требуется никакого программирования, и бóльшая их часть выполняется на странице **Create**.

Перейдя в раздел **Games** на странице **Create**, который можно выбрать в левой части экрана, щелкните по иконке шестеренки в правой части игры, как показано на рис. 2.2. Нажав на этот значок, вы увидите раскрывающееся меню, в котором перечислены различные варианты.

Рис. 2.2. Меню, выпадающее из значка шестеренки на сайте Roblox

В этом меню нас будут интересовать подменю **Configure Game** (Настройка игры) и **Configure Start Place** (Настройка начального плейса), так как в них есть много важной информации.

Меню Configure Game

В подменю **Configure Game** вы можете изменить множество настроек игры. Например, настройку конфиденциальности можно задать как **Public** (Общедоступный) или **Private** (Приватный), как показано на рис. 2.3.

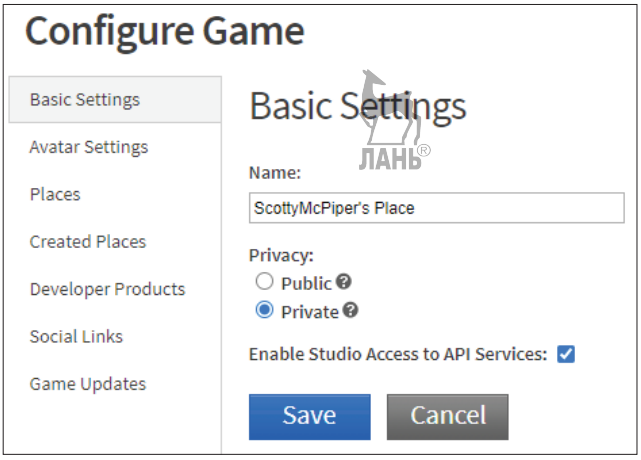


Рис. 2.3. В меню **Configure Game** изменяются внешние настройки игры

Давайте рассмотрим разницу между опциями **Public** и **Private**:

- когда ваша игра приватна (**Private**), в нее смогут играть только игроки, для которых вы установили разрешения вручную;

- когда игра общедоступна (**Public**), все пользователи Roblox смогут играть в вашу игру. Мы обсудим, как изменить эти разрешения для конкретных пользователей, когда перейдем к изучению Roblox Studio.

В меню **Configure Game** и **Configure Start Place** есть вкладка, на которой можно создавать и настраивать продукты разработки. Процесс их создания достаточно прост для понимания: вы можете добавить имя, цену, описание и изображение для своих продуктов для разработчиков. Они появятся в том виде, в каком вы их настроили, на всех ваших игровых серверах, и будут обновляться в режиме реального времени. Когда мы дойдем до создания собственной игры, то обсудим, как связать покупку как игровых пропусков, так и продуктов разработки с вашими скриптами, чтобы все работало правильно.

В нижних вкладках на странице **Configure Game** сайта Roblox можно размещать ссылки на разные аккаунты в социальных сетях. Это единственное место, где нужно добавлять ссылки на социальные сети, так как Roblox в своих условиях обслуживания указывает, что определенные ссылки на социальные сети могут показываться или скрываться в зависимости от возраста, указанного в учетной записи пользователя, чтобы защитить более молодых игроков. Как продемонстрировал печальный опыт других разработчиков, нарушение этого пункта может привести к ограничениям в отношении вашей учетной записи или всей игры.

Меню Configure Start Place

В меню **Configure Start Place** вы найдете много опций, влияющих на внешний вид и игру целом. Некоторые из этих настроек показаны на рис. 2.4. Первым делом вы попадете на вкладку **Basic Settings**, где можно изменить название вашей игры и описание, которое будет отображаться под ним.

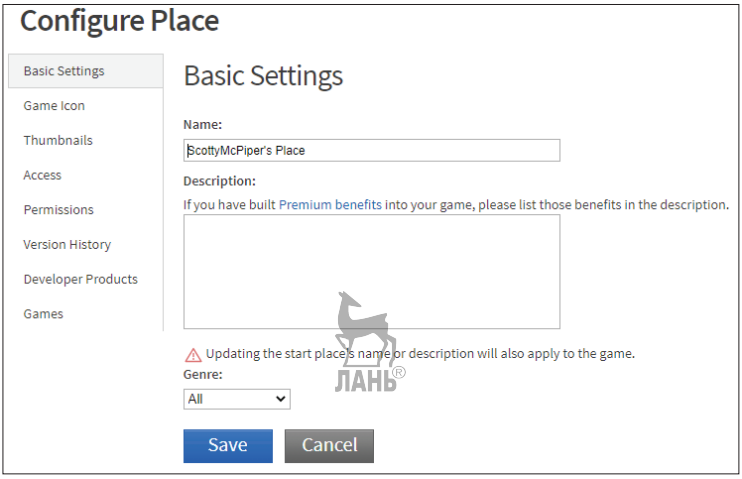


Рис. 2.4. Страница **Configure Start Place** похожа на страницу **Configure Game**, но на ней другие настройки

Вторая вкладка под названием **Game Icon** (Значок игры) позволяет вам изменять значок игры. Значок игры – это небольшое квадратное изображение, которое обычно формирует у игроков первое впечатление о вашей игре, когда они видят его на странице игры или в профиле. Аналогично на вкладке **Thumbnails** (Миниатюры) вы можете загрузить изображения, которые позволяют игроку получить первое впечатление перед запуском игры. За 500 робаксов вы также можете загрузить видео продолжительностью до 30 секунд. Это позволяет показать посетителям игровой процесс, что часто дает хороший эффект.

На вкладке **Access** (Доступ) вы можете изменить настройки, касающиеся доступа к игре. Перед вами появится список флажков, соответствующих платформам, на которых доступна ваша игра. Когда мы приступим к созданию вашей первой игры, вам нужно будет включить или отключить нужные параметры в зависимости от механики вашей игры и ее кросс-платформенной совместимости. Ниже расположены опции, позволяющие продать доступ к вашей игре. В большинство игр на Roblox можно играть бесплатно, вы можете сделать сам доступ к своей игре предметом монетизации (хотя обычно это приводит к снижению числа игроков).

Настройка **Maximum Player Count** задается в зависимости от того, сколько игроков могут играть в вашу игру одновременно на одном сервере. Выбор этого значения остается за вами, но важно учитывать ограничения производительности вашей игры, а также возможность использований тех или иных механик, если в игре слишком много или слишком мало игроков. Этот параметр также влияет на то, как Roblox заполняет серверы игроками. По умолчанию Roblox оптимизирует заполнение сервера в зависимости от этой опции, и менять ее не стоит. В некоторых ситуациях имеет смысл зарезервировать несколько серверных слотов, если окажется, что в эту игру круто играть группой из друзей, но для новичка нет причин менять эту настройку.

Настройка **Access** может принимать значения **Everyone** (Все) или **Friends/Group Members** (Друзья/Члены группы) в зависимости от того, размещается ли игра в профиле игрока или в группе. Вы можете изменить этот параметр по своему усмотрению, но обычно ограничения приводят к снижению числа игроков.

Последняя настройка на вкладке **Access** позволяет вам продавать **частные серверы** (раньше они назывались VIP-серверами). Частные серверы – это ежемесячная подписка, приобретаемая игроками для того, чтобы они всегда могли играть с друзьями или с теми, кого пригласят. Поскольку частные серверы – это регулярная покупка, они могут продолжать приносить пассивный доход даже после того, как популярность игры снизится.

На рассмотренных страницах настроек есть и другие вкладки, которые не были рассмотрены в этом разделе, но большинство из них устарели или при обычных обстоятельствах вам не нужны. Когда вы станете более опытным разработчиком и возникнет необходимость редактировать эти параметры, вы уже будете понимать, что к чему.

Важное примечание

Вы не можете одновременно продавать доступ к игре и частные серверы. При продаже доступа минимальная цена составляет 25 робаксов, а максимальная – 1000 робаксов. При продаже частных серверов минимальная цена составляет 10 робаксов, если вы не хотите, чтобы они были бесплатными. На максимальную цену частных серверов ограничений нет.

Другие настройки плейса

Вернемся на страницу **Create**. В раскрывающемся меню, ранее показанном на рис. 2.2, еще много нерассмотренных опций. Обязательно ознакомьтесь с ними перед созданием первой игры. Не все опции будут рассмотрены подробно, но вы должны знать, что они из себя представляют, так как они вам пригодятся при разработке более сложных проектов.

Сразу после опций **Configure Game** и **Configure Start Place** есть опция **Configure Localization**. На этой странице вы можете выполнять ручной перевод вашей игры на различные поддерживаемые языки. Вы можете добавлять разных пользователей или группы Roblox для перевода игры, добавлять или удалять языки, на которые должна быть переведена ваша игра, и просматривать, какие фразы из вашей игры уже были переведены. Кроме того, в середине 2020 года сотрудники компании Roblox объявили, что они начали внедрять в лучшие игры на платформе автоматический перевод как игровых веб-страниц, так и внутриигровых текстов, а после пробного периода планируют расширить это на большее количество игр. Подробнее почитать о настройке локализации можно в статье от Roblox: **https://developer.roblox.com/en-us/articles/Introduction-to-Localization-on-Roblox**.

Следующий пункт меню – **Create Badge** (Создать значок). Значки – это предметы, которые присуждаются игрокам и отображаются в профиле. После присуждения значок нельзя выдать снова, если игрок не удалит его из своего инвентаря. Значки в основном используются для выставления напоказ или в качестве нового контента, но иногда они применяются как своего рода хакерский способ отслеживания движения данных. Например, если вы хотите, чтобы все игроки, участвовавшие в предварительном тестировании вашей игры, получили в награду что-то особенное, самый удобный способ сделать это – выдать им значок. С помощью этой техники, даже несмотря на то, что при выпуске игры хранилища данных будут сброшены, вы все равно можете определить, владеет ли игрок нужным значком, и впоследствии наградить его за то, что он одним из первых попробовал вашу игру.

Опция **Create Game Pass** открывает новую вкладку на странице **Create**. Как и в случае продуктов для разработчиков, вы можете добавить игровому пропуску название, описание и изображение с помощью интерфейса, показанного на рис. 2.5.

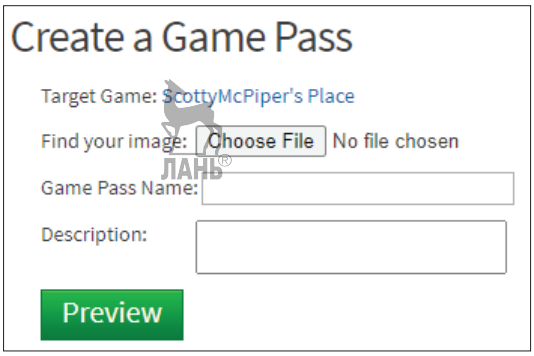


Рис. 2.5. Интерфейс для создания игровых пропусков аналогичен интерфейсу продуктов для разработчиков

Созданный игровой пропуск появится в списке, а щелчок по значку шестеренки рядом с ним позволит вам настроить его цену, выставить его на продажу или убрать, а также задать упомянутые выше настройки.

Опция **Add Gear** (Добавить бонус) – это, вероятно, устаревшая система продажи созданных компанией Roblox бонусов внутри игры. В большинстве игр такие бонусы в наши дни уже не используются, но, возможно, вам будет интересно попробовать создать какой-нибудь такой бонус. В этом случае лучше всего было бы создать игровой пропуск для бонусов, так как при продаже своих продуктов Roblox перечисляет вам лишь 10 % от продажи, а не 70. Бонусы можно найти, выполнив поиск по названию в коллекции бесплатных ресурсов, которые мы рассмотрим в следующем разделе.

Вкладка **Developer Stats** (Статистика разработчика) вам очень пригодится, когда вы начнете отслеживать эффективность своих игр. Страница, на которую ведет эта кнопка, является центром всей игровой аналитики. Здесь вы можете увидеть среднее время вовлечения игрока, общую сумму, полученную от внутриигровых продаж до уплаты налогов, и количество пользователей на различных платформах. Все это можно просмотреть в виде наборов данных за разные промежутки времени. Кроме того, вы можете экспортировать аналитику своей игры в файл CSV или XLSX, если эти данные нужны вам еще для чего-то.

Чтобы ваша игра стала популярна, у Roblox предусмотрено два встроенных вида продвижения под названием **User Ads** (Реклама для пользователей) и **Sponsored Games** (Спонсорство). Продвижение вашей игры через эти системы небесплатно (в робаксах), и чем больше вы вкладываете, тем больше игроков увидит рекламу (см. рис. 2.6). Вы можете выбрать, на каких платформах будет проходить ваша кампания, то есть если ваша игра недоступна на Xbox, вам не нужно тратить деньги на продвижение для игроков с Xbox.

Разница между рекламой и спонсорством игры заключается в том, как ваш рекламный материал будет показываться игрокам. Например, при использовании **Sponsored Games** значок вашей игры будет отображаться

везде, где выводится список игр, в том числе на главной странице или когда пользователь ищет игру с помощью поиска. Под значком будет приписка **Sponsored Ad**. При использовании системы **User Ads** вам нужно будет загрузить изображение определенного размера (список размеров будет представлен). Реклама производит больший CTR (количество кликов, деленное на показы), но игроки видят ее реже. Причина этого в том, что большинство блокировщиков рекламы браузеров автоматически скрывают такую рекламу. Лично я пользуюсь системой **Sponsored**, хотя в сообществе разработчиков до сих пор ведутся споры о том, какая система лучше. Эффективность в конечном итоге будет зависеть от того, насколько дизайн вашего значка или рекламного изображения привлекателен.

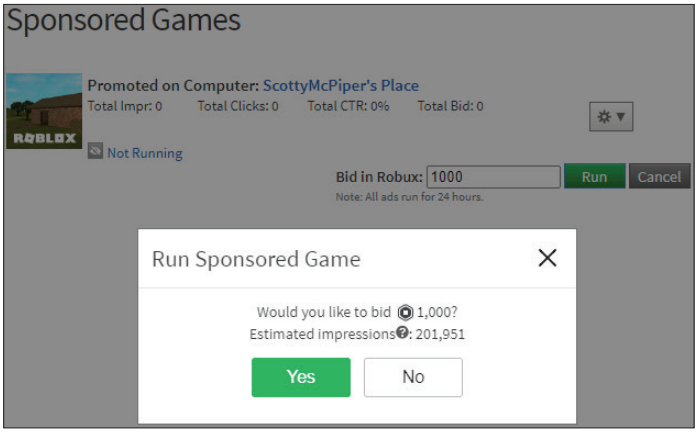


Рис. 2.6. Чем больше робаксов вы вложите, тем больше предполагаемых показов получите

Важное примечание

Вам может быть полезно знать, что количество показов линейно зависит от вложенной суммы, то есть если вы вложите в рекламу в два раза больше, то получите примерно в два раза больше показов. Количество показов, которые вы получаете от продвижения, будет меняться ежедневно в зависимости от того, сколько других разработчиков продвигают свои игры. Для получения дополнительной информации о системе User Ads зайдите на сайт **https://developer.roblox.com/en-us/articles/Advertising**.

Библиотека и магазин аватаров

Library (Библиотека) – это коллекция игровых ресурсов, созданных другими пользователями. Когда вы работаете в Studio, библиотека доступна на панели инструментов **Toolbox**. Здесь разработчики могут найти множество готовых сценариев, аудиофайлов, моделей, изображений, мешей

и плагинов, которые можно использовать в своих играх бесплатно, без указания авторства. Это отличный ресурс, и им можно воспользоваться, но есть некоторые вещи, о которых следует помнить. Во-первых, нельзя злоупотреблять бесплатными моделями. В этом нет ничего постыдного, но многие члены сообщества разработчиков часто смотрят свысока на чрезмерное использование бесплатных ресурсов, поскольку в этом случае в вашем проекте получается мало вашей собственной работы. Кроме того, хотя большинство моделей полностью безопасны для использования, существуют и злоумышленники, которые помещают в модели сценарии, вызывающие задержки или неожиданное поведение игры. При поиске изображений, аудиофайлов или мешей такая проблема неочевидна. Кроме того, в отличие от моделей, эти типы ресурсов не так заметны, и их проще использовать, не вредя своей репутации. Лучший совет – использовать бесплатные модели в умеренных количествах, принимая также во внимание их оценки от других пользователей и проверяя их на наличие нежелательных сценариев.

Важное примечание

Хочу предупредить: если вы решите загрузить на панель Toolbox свои собственные ресурсы, они должны соответствовать правилам модерации. Загрузка неприемлемых вещей или объектов, защищенных авторским правом, может привести к мерам модерации в отношении вашей учетной записи. В зависимости от серьезности вашего правонарушения наказание может варьироваться от удаления ресурса до блокировки вашей учетной записи.

Avatar Shop (Магазин аватаров), который раньше назывался **Catalog** (Каталог), представляет собой коллекцию одежды и аксессуаров для персонажей Roblox. Большинство аксессуаров созданы командой Roblox, но некоторые из них созданы членами сообщества, которые были допущены к участию в программе **UGC** (User-Generated Content, Пользовательский контент). Загружать одежду на сервер может кто угодно и бесплатно. Эти предметы одежды и аксессуары можно бесплатно использовать в играх либо как часть дизайна персонажа, либо как дополнение к карте, что тоже бывает.

Теперь вы можете воспользоваться функциями страницы **Create** для создания собственных игр. Вы научились создавать новые игры и выполнять полную настройку их внешних параметров, настраивать продвижение игры для привлечения новых игроков и использовать бесплатные ресурсы для облегчения процесса разработки. Все это поможет вам успешно создавать свои проекты, и скоро вы познакомитесь с Roblox Studio и начнете писать программный код.

Начало работы с Roblox Studio

Roblox Studio – это программа, в которой вы будете создавать все свои игры. Studio оснащена множеством инструментов, содержит надстройки и работает как полноценный центр разработки от концепции проекта до его выпуска.

Вернитесь на страницу **Create** и нажмите кнопку **Edit** рядом со значком шестеренки. Если Studio еще не загружена, вам будет предложен набор инструкций по ее загрузке, а после установки программа запустится автоматически – см. рис. 2.7.

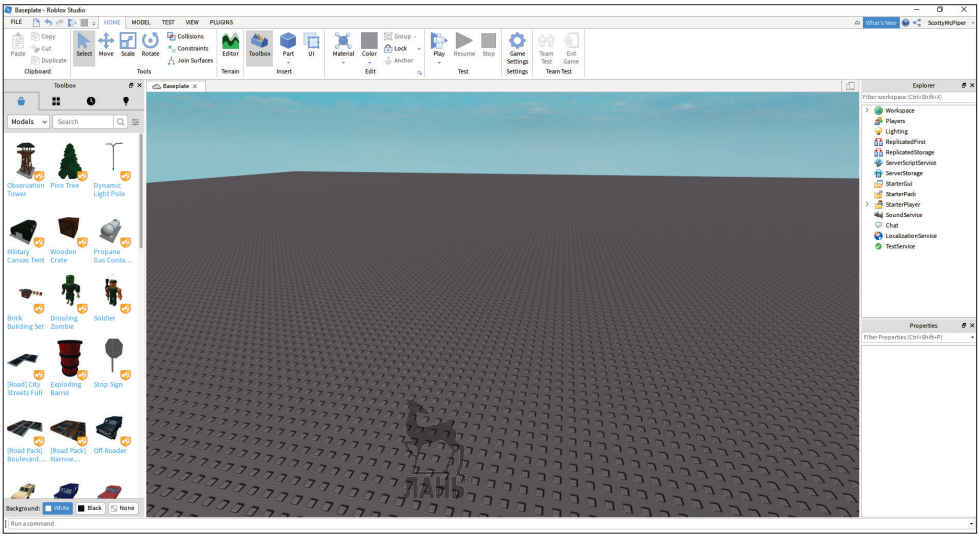


Рис. 2.7. При нажатии кнопки **Edit** вам будет предложено загрузить программу Studio, если она не установлена

К концу этого раздела вы узнаете, что интересного есть на вкладках и меню Roblox Studio, познакомитесь с различными инструментами Studio и будете готовы приступить к настройке своей первой игры.

Меню File и настройки

Как и в большинстве приложений, в меню **File** (Файл) находятся основные операции и подменю приложения. Некоторые из них показаны на рис. 2.8. Начинающему разработ-

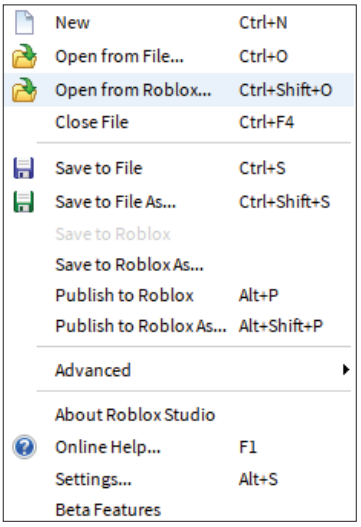


Рис. 2.8. При нажатии кнопки **Edit** вам будет предложено загрузить Studio, если она не установлена

чику нужны не все эти пункты, некоторые из них используются в рамках повседневной работы в Studio.

Начнем сверху вниз: опция **New** (Новый) открывает новое окно Studio, где в рабочем пространстве (**Workspace**) будет лишь пустая плоскость. Команды **Open from File...** (Открыть из файла) и **Open from Roblox...** (Открыть из Roblox) открывают новый сеанс Studio, и вам будет предложено выбрать существующий проект. Опция **Close file** (Закрыть файл) не закроет окно, а откроет страницу, похожую на **Open from Roblox...**, где вы увидите список созданных вами игр.

Команды **Save to File** (Сохранить в файл) и **Save to File As...** (Сохранить в файл как...) позволяют сохранить имеющийся файл плейса или выбрать новый. Файлы плейсов сохраняются с расширением `.rbxl`. Эти файлы обычно весят немного, и их легко перемещать между носителями. Следует помнить (и мы вернемся к этому позже), что некоторые загружаемые в игру ресурсы, например изображения, не будут отображаться, если файл ресурса открыт где-то еще, помимо игры, поскольку файлы находятся по специфичному для игры пути. Команды **Save to Roblox** (Сохранить в Roblox) и **Save to Roblox As...** (Сохранить в Roblox как...) работают аналогично, но игра будет сохраняться на серверах Roblox.

Команды **Publish to Roblox** (Опубликовать в Roblox) и **Publish to Roblox As...** (Опубликовать в Roblox как) используются для обновления игры. Когда вы вносите в игру изменения, программа Studio периодически выполняет автосохранение, но изменения при этом не распространяются на новые серверы. Чтобы применить изменения, вы должны **опубликовать** игру в Roblox. Важно помнить, что публикация новой версии игры не влияет на работающие серверы. Чтобы применить изменения полностью, нужно закрыть все серверы игры или постепенно перевести их на текущую версию. Вы можете отключить серверы игры, перейдя на ее веб-страницу и нажав кнопку **Shut Down All Servers** (Отключить все серверы) в меню, изображенном на рис. 2.9. Это меню можно найти на веб-странице вашей игры, нажав на иконку в виде трех точек в правом верхнем углу.

Рис. 2.9. Три точки для переключения этого меню видны во всех играх, которые вы можете редактировать

Помните, что при использовании этой опции все игроки будут отключены от игры, и им нужно будет перезагрузить ее. Опция **Migrate To Latest Update** (Перейти на последнее обновление) тоже отключает игроков от игры, но сводит к минимуму время повторного присоединения игроков и перебои в работе игры в целом за счет создания новых серверов и отключения старых с интервалом в 6 минут, а не всех сразу. Этот вариант обновления более удобен, если вы не публикуете патч или

что-то такое, что не обязательно должно дойти до игроков сию же минуту.

Опция **Online Help** (Помощь онлайн) приведет вас на веб-сайт Roblox, предназначенный для помощи разработчикам. На этом сайте вы найдете не только учебные пособия, но и множество документации и других вспомогательных ресурсов, которые мы рассмотрим позже в этой главе.

При нажатии на кнопку **Settings** (Настройки) в Studio откроется новое диалоговое окно. В нем вы найдете много настроек, некоторые из которых довольно сложные и используются редко. Новичку большинство приведенных здесь настроек не нужно, за исключением некоторых, которые вы можете просто изменить ради удобства. Эти настройки носят в основном косметический характер, и мы поговорим о них в следующих разделах.

Движение и управление камерой

Управление камерой в Roblox Studio выполняется довольно просто. Ваша камера будет двигаться относительно направления, в котором вы смотрите:

- клавиша *W* перемещает камеру вперед;
- клавиша *A* перемещает камеру вперед;
- клавиша *S* перемещает камеру назад;
- клавиша *D* поворачивает камеру вперед;
- клавиша *E* перемещает камеру вверх;
- клавиша *Q* перемещает камеру вниз;
- если вы зажмете клавишу *Shift* при перемещении камеры, движение будет происходить медленнее, что удобно для изучения мелких объектов в рабочем пространстве.

Чтобы изменить направление камеры, зажмите правую кнопку мыши и переместите ее. С помощью колесика мыши вы можете увеличить масштаб камеры. По умолчанию колесо прокрутки будет перемещать вашу камеру в направлении мыши, но можно также изменять масштаб в направлении взгляда камеры, сняв флажок **Camera Zoom to Mouse Position** (Масштаб камеры по положению мыши) в Studio.

Если вам не нравится текущая скорость камеры, вы можете изменить ее, перейдя на вкладку **Studio** в меню **Settings** и изменив соответствующие значения в разделе **Camera**, как показано на рис. 2.10.

Если производительность вашей системы позволяет, вы также можете увеличить графические настройки Studio. Для этого откройте окно **Settings** и перейдите на вкладку **Rendering** (Рендеринг). В верхней вкладке **Rendering** есть настройки **Quality Level** (Качество) и **Edit Quality Level** (Изменить качество), которые по умолчанию равны **Automatic**. Рекомендуется изменить эти значения на **Level 21**, так как в этом случае будет видна подробная информация о рабочем пространстве.

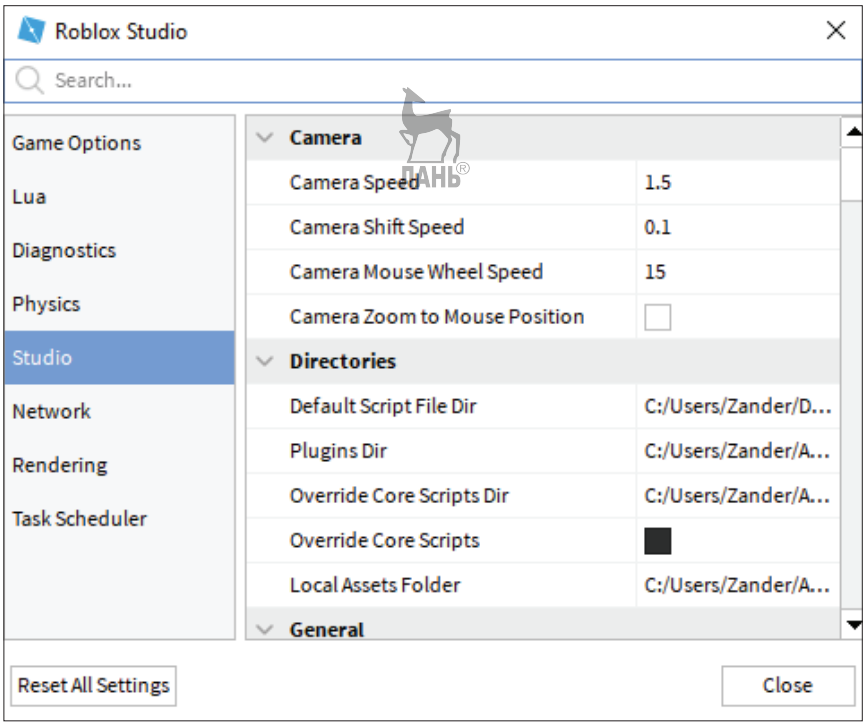


Рис. 2.10. Косметические настройки для Studio можно найти на вкладке **Studio** в меню настроек

Панель Explorer

Перейдите на вкладку **Home** (Главная) Studio и нажмите кнопку **Part** (Деталь), после чего перед вами появится окошко. Если вы выберете деталь и нажмете клавишу *F*, ваша камера сфокусируется на детали. Это применимо к любой выбранной детали. В правом верхнем углу стандартного экрана Studio вы увидите значок панели **Explorer** (Проводник). На панели **Explorer** отображаются все присутствующие в игре объекты, чаще называемые **экземплярами**. Об экземплярах и других объектах, которые отображаются на панели **Explorer**, мы более подробно поговорим в следующих главах. Пока в верхней части панели вы увидите пункт **Workspace** с иконкой в виде глобуса, как показано на рис. 2.11.

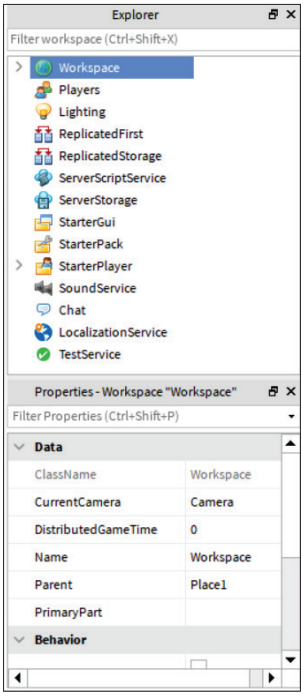


Рис. 2.11. Меню **Explorer** и **Properties** с выбранным пунктом **Workspace**

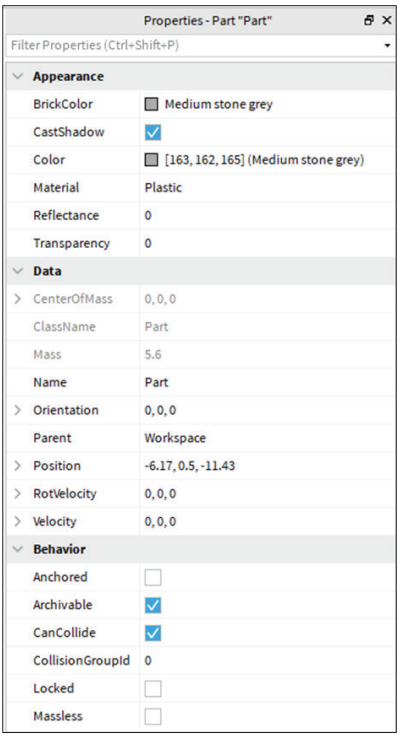


Рис. 2.12. В меню **Properties** отображаются свойства экземпляра детали

Выбрав деталь, вы увидите все ее настройки в меню **Properties**. Свойства экземпляров задают их поведение и внешний вид. Чаще всего у видимых деталей меняются свойства **Anchored** (Закрепление), **CanCollide** (Возможность столкновения), **Transparency** (Прозрачность), **Material** (Материал) и **Color** (Цвет) – см. рис. 2.12.

Рассмотрим эти пять свойств подробнее.

- Когда у детали включено свойство **Anchored**, она не перемещается под действием силы тяжести или любых других сил, связанных с физикой. Деталь, у которой это свойство отключено, называется незакрепленной.
- Свойство **CanCollide** определяет, может ли деталь сталкиваться с любым другим экземпляром в рабочем пространстве. Если это свойство отключено, деталь будет падать сквозь карту, пока не уничтожится. В этом случае разумно включить закрепление модели.
- Свойство **Transparency** позволяет изменять степень видимости детали в диапазоне от 0 до 1.
- **Material** – это список различных текстур деталей, которые также имеют разные визуальные и физические свойства, например отражательную способность, плотность, эластичность и трение. Эти значения можно изменить с помощью свойства **CustomPhysicalProperties** (Пользовательские физические свойства).
- Свойство **Color** позволяет менять цвет детали. Поддерживаются палитры RGB и HSV. В Roblox также есть меню именованных готовых цветов в свойстве **BrickColor** (Готовый цвет).

`Model` (Модель) – это еще один вид экземпляра, используемый для объединения нескольких экземпляров поменьше. Удерживая клавишу *Ctrl*, вы можете выбрать несколько деталей на панели **Explorer** или в рабочем пространстве (**Workspace**). Выбрав несколько деталей, вы можете нажать комбинацию клавиш *Ctrl+G*, чтобы объединить все выбранные экземпляры в новую модель. Это удобно и с точки зрения организации деталей, и для работы с некоторыми инструментами, которые мы обсудим в следующем разделе.

Работа с инструментами в Studio

Инструмент **Select** (Выбрать) используется для перемещения элементов в рабочем пространстве. Если вы щелкнете (зажав клавишу) по детали в рабочем пространстве и переместите мышь, деталь будет следовать за курсором. При работе с закрепленными деталями нужно отключить опцию **Join Surfaces** (Объединение поверхностей), так как эта опция создает лишние экземпляры всего, что перетаскивается. В то же время эта функция может быть полезна при работе с незакрепленными деталями, поскольку она соединяет их друг с другом, но при этом на них могут воздействовать физические силы. Это свойство находится справа от четырех инструментов под вкладкой **Home** или **Model**.

Инструмент **Move** (Перемещение) – это набор из шести маркеров, с помощью которых можно перемещать выбранную деталь. Маркеры по умолчанию располагаются в векторах относительного направления: вперед, назад, вверх, вниз, влево и вправо. Нажимая на маркеры и перемещая их вдоль выбранной оси, вы можете перемещать детали точно вдоль указанной оси, не меняя положения других. У этого инструмента есть и иные, менее очевидные функции, а именно использование мировых координат, свойства **Snap to Grid** (Привязка к сетке) и **Collisions** (Столкновения). Если вы нажмете комбинацию клавиш *Ctrl+L*, маркеры будут располагаться в глобальных координатах, а не в системе координат экземпляра. Эта функция особенно важна, если деталь вращается, а перемещение требуется в глобальных координатах. Свойство **Collisions** включено по умолчанию и не позволяет перемещать детали сквозь друг друга. Иногда это бывает удобно, но вы, скорее всего, захотите отключить это свойство, поскольку оно не позволяет вам перемещать детали даже при малейшем столкновении с другими. Свойство **Snap to Grid** находится на вкладке **Model** и дает вам больше гибкости в контроле перемещения детали. Это свойство можно оставить, если хотите, но многие разработчики задают это свойство равным 0, чтобы иметь больше свободы в перемещении модели в рабочем пространстве.

Инструмент **Scale** (Масштаб) позволяет вам масштабировать размеры моделей и отдельных деталей. Как и инструмент **Move**, его можно использовать совместно с **Snap to Grid**. Маркеры инструмента **Scale** всегда относительные, поэтому переключение на мировые координаты не будет иметь никакого эффекта. Также этот инструмент позволяет выполнять **вертикальное масштабирование** и **полное масштабирование**. Удерживая клавишу *Ctrl* при масштабировании детали, вы можете зеркально отразить эффект масштабирования относительно другой стороны выбранной оси. Модели всегда масштабируются как единое целое, то есть сохраняют свои относительные размеры, но отдельный экземпляр тоже можно масштабировать целиком, удерживая клавишу *Shift*.

Инструмент **Rotate** (Повернуть) имеет шесть маркеров, которые позволяют вращать деталь вокруг осей *x*, *y* и *z*. Этот инструмент поддерживает переключение на мировые координаты и свойство **Snap to Grid**. Свойство

Snap to Grid инструмента **Rotate** находится выше, чем у **Scale** и **Move**, и принимает желаемое изменение в градусах, а не в **Studs**.

Инструмент **Transform** (Трансформация) сочетает в себе все четыре вышеупомянутых инструмента и находится на вкладке **Model**. Вы можете использовать его по своему усмотрению, и он довольно удобен тем, что позволяет свободно использовать функциональные возможности всех четырех инструментов без необходимости переключения между ними. Удобно, что вдоль нижней части выбранной детали есть сетка, помогающая вам ориентироваться.

Настройки в меню Game Settings

Мы уже знаем, как можно изменить многие настройки игры на странице **Create**, но есть и такие параметры игры, которые можно изменить только в Studio. Меню **Game Settings** похоже на страницу **Create**, но используется в первую очередь для отображения дополнительных настроек и доступа к ним. На рис. 2.13 показан вид меню **Game Settings** и его вкладок.

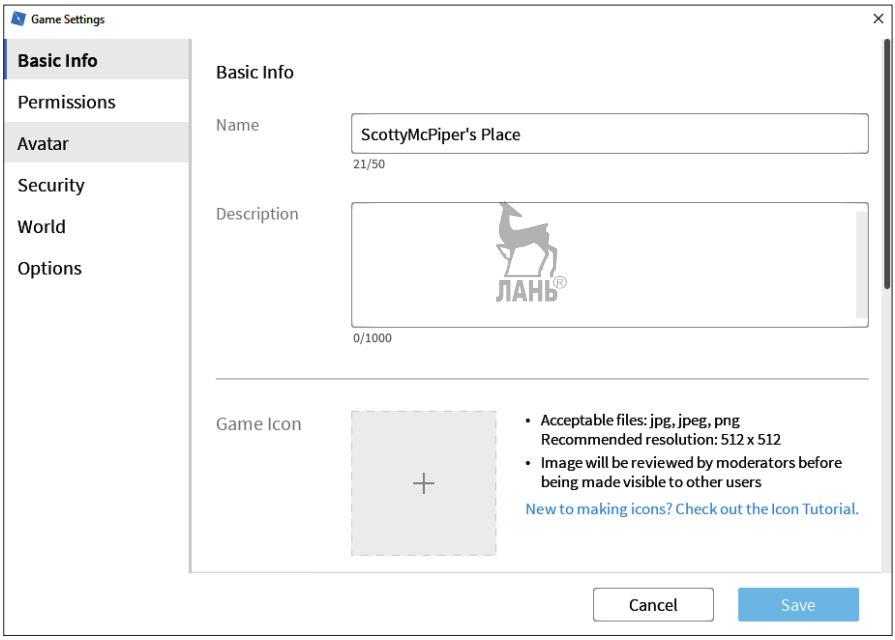


Рис. 2.13. В меню **Game Settings** вы можете изменять настройки со страницы **Create** и многие другие

Недавно в Studio появилась вкладка меню **Permissions**. Раньше, если вам было нужно, чтобы вашу игру подтестировало несколько людей, нужно было выдать им статус разработчика либо выделить всех тестировщиков в отдельную группу, в которой больше никого нет. С помощью этой функции вы можете изменять разрешения для разных пользователей или групп, разрешая им играть или вносить изменения в игру. Если у пользователя

или роли есть разрешение **Edit**, вы сможете использовать функцию **Team Create**, которая будет рассмотрена в следующем разделе.

Параметры и предустановки на вкладке **Avatar** раньше находились в меню **Configure Game** на странице **Create**, но были перемещены для упрощения доступа и потому, что эти настройки связаны со Studio напрямую. Здесь и на других вкладках вы увидите, что большинство настроек можно не менять, поскольку значения по умолчанию почти всегда вам подойдут. Но для общей информации поясним: у Roblox есть два основных типа персонажей, R15 и R6, где число в имени означает, из скольких деталей состоит модель персонажа. Во время программирования бывает необходимо определить часть тела, которая существует только в одном из этих шаблонов, например руку. Чтобы ограничить количество перебираемых условий, можно указать, какой тип шаблона используется в вашей игре, а также задать параметры масштабирования и анимации. В основном поведение, достигаемое этими настройками, можно реализовать программно, но для простоты можно использовать меню.

В меню **Options** (Опции) можно включить функцию **Collaborative Editing** (Совместное редактирование). Эта функция тоже появилась недавно, используется для управления исходным кодом и весьма удобна при организации одновременной работы нескольких программистов. При редактировании сценария с этой опцией в **Team Create** под вкладкой **View** появляется кнопка **Drafts**. Внесенные изменения будут сохраняться только локально, пока сценарий не будет закоммичен. Если правки в программу уже были внесены, вы можете увидеть различия и выполнить слияние перед фиксацией. У этой функции нет всей силы инструментов контроля версий вроде Git, но зато ее можно использовать из коробки без дополнительных настроек.

Вкладка **Security** (Безопасность) невероятно важна во время разработки новой игры с запрограммированными системами. Например, если вы хотите, чтобы ваша игра могла взаимодействовать с удаленными серверами или разрешить хранилищам данных Roblox сохранять данные, когда вы работаете в Studio, в этом меню необходимо включить нужные опции. Также вы можете настроить сторонние продажи и сторонние телепорты. Если вы вносите в игру хотя бы минимальную защиту (которую мы рассмотрим далее в книге), включение этих настроек пригодится вам в будущем.

На вкладке **World** (Мир) вы можете изменять различные настройки того, как игрок взаимодействует с рабочим пространством, включая настройки физики и различные свойства персонажа игрока. Некоторые изменения удобно вносить здесь, но в целом все эти настройки можно задать в сценариях или путем настройки **сервисов**, о которых мы поговорим, как только вы начнете программировать. В качестве примера изменения этих настроек можно поменять настройки экземпляра **Humanoid** персонажа или свойство **Gravity**, чтобы изменить высоту прыжка игроков. Как программист вы должны уметь менять все эти свойства без меню, но это не жесткое правило, а вопрос предпочтений и удобства.

> **Важное примечание**
>
> В 2020 году сотрудники компании Roblox объявили, что намерены перенести все функции со страницы Create на отдельную страницу в Studio. На данный момент информации об этом мало, вполне вероятно, что интерфейс будет похож на меню **Game Settings**.

Вкладка View

На вкладке **View** в Studio вы найдете множество дополнительных инструментов, а также переключатели для различной информации, которая может отображаться при разработке. Некоторые из этих опций показаны на рис. 2.14. Как обычно, рассмотрим данный функционал для ознакомления.

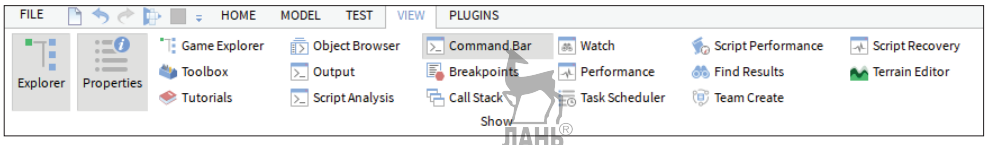


Рис. 2.14. Вкладка **View** позволяет разработчикам использовать дополнительные инструменты и просматривать скрытую информацию

Один из наиболее важных инструментов Studio для управления загруженными ресурсами – окно **Game Explorer**. В этом окне вы можете не только просмотреть все ресурсы, загруженные непосредственно в игру, но и использовать функцию **Bulk Import** (Импортировать все). Эта функция позволяет загружать в игру сразу до 50 ресурсов по одному через сайт. Важно помнить, особенно работая с изображениями, что при загрузке ресурса экземпляр, содержащий его, не может быть напрямую передан в другую игру, поскольку путь к изображению задается игрой. Вы можете найти URL-адрес ресурса с помощью менеджера, который позволяет использовать ресурс и просматривать его где угодно.

Функция **Team Create** открывает в Studio новое окно, используемое для совместной работы над проектами. С ее помощью проект может редактироваться одновременно несколькими разработчиками, что устраняет проблему перезаписи изменений, внесенных другими разработчиками на локальных машинах. Чтобы предоставить другим пользователям разрешение на редактирование вашей игры, вам нужно будет перейти на страницу **Game Settings** и там найти вкладку **Permissions**. В ней вы можете найти пользователей по имени и убедиться, что у них есть разрешение **Edit**.

Command Bar (Панель команд), которую мы рассмотрим подробнее, когда будем учиться программировать, – это место выполнения кода Lua, и ее можно использовать во время тестирования или просто в Studio. **Command Bar** – это полезный инструмент для сложных задач, требующих довольно много времени, например управления ресурсами и данными.

Окно **Output** (Вывод) почти всегда используется вместе с **Command Bar** во время тестирования, так как здесь отображается весь выводимый и исполняемый код. Кроме того, если сценарий в вашей игре выдает ошибку или любую другую информацию, она попадает в это окно. Ошибки, предупреждения и информационные сообщения в окне интерактивные, и, кликнув по ним, вы перейдете к сценарию и строке, которая произвела вывод. Если Studio нужно вывести дополнительную информацию, например результат некоторого действия, или сообщить о том, что игрок **Team Create** сделал что-то важное, эта информация также выводится в окно **Output**.

Нажав комбинацию клавиш *Shif+F5*, вы можете вывести частоту кадров в игре и дополнительную информацию о рендеринге. Переключение опции **Summary** на вкладке **View** отображает частоту кадров в Studio, а также информацию о рендеринге, которая обычно доступна во время игры.

В Roblox Studio есть много интересных функций и инструментов, и из-за этого программа иногда может зависать либо аварийно завершиться при работе со слишком большим числом деталей одновременно, даже в 64-битной версии. Когда это происходит, Roblox практически всегда локально сохраняет файл игры на вашем компьютере. Более того, если поломка произошла в среде **Team Create**, вы можете с помощью вкладки **Script Recovery** (Восстановление сценария) восстановить потерянный прогресс и применить его к текущей версии игры.

Вкладка Test

На вкладке **Test** (Тестирование) Studio есть несколько различных режимов, с помощью которых вы можете проверить функциональность вашей игры. Перед выпуском игры вы должны проверить, что в ней нет ошибок или нежелательного поведения. Даже если все системы вашей игры работают, нужно проверить правильность работы на всех целевых платформах. Чем больше платформ ваша игра будет поддерживать, тем больше потенциальных игроков смогут в нее играть. Вы можете выбрать один из режимов инициализации в выпадающем меню под кнопкой **Play**, как показано на рис. 2.15.

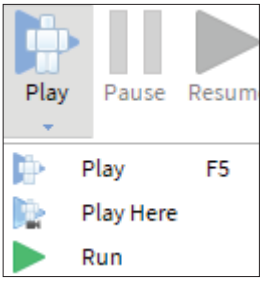


Рис. 2.15. В Studio есть три режима инициализации для тестирования

Кнопка **Play** работает по-разному в зависимости от настроек рабочего пространства.

Если в разделе **Team** на панели **Explorer** есть экземпляры **Team**, а в рабочем пространстве есть **Spawn Location**, игрок случайным образом появится в одной из локаций возрождения команды. Если точек возрождения или команд нет, игрок появится в нейтральной точке возрождения, если такая есть. Если их нет, то игроки будут появляться в точке с координатами *0, 0, 0*.

Опция **Play Here** порождает вас там, где в данный момент находится ваша камера, независимо от настроек точек возрождения. Если у вас есть

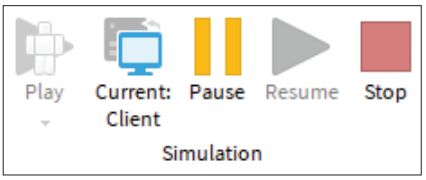


Рис. 2.16. Кнопки **Pause**, **Resume**, **Stop** и **Current** доступны только во время тестирования

сценарий, который вручную устанавливает положение персонажа, он будет иметь приоритет, так как режим тестирования позиционирует игроков до загрузки клиента.

Опции **Play** и **Play Here** загружают вас в качестве клиента, а это означает, что будут загружены как локальные, так и серверные сценарии, но панель команд позволяет вносить только локальные изменения. Однако, нажав кнопку **Current** под вкладкой **Test** (см. рис. 2.16), вы можете переключаться между представлениями клиента и сервера и использовать панель команд сервера, а также свободно перемещать камеру. Отметим также, что нажатие комбинации клавиш *Shift+P* в режиме просмотра клиента или в игре позволяет переключаться между стандартной и свободной камерами.

Команда **Run** запускает игру лишь на сервере. Клиенты создаваться не будут, и у вас будет доступ только к панели команд сервера и рабочему пространству сервера. В этом режиме вы не сможете использовать функцию отображения **Current**.

Local Server – это режим тестирования вашей игры с несколькими пользователями (не путать с настоящими людьми). У каждого создаваемого вами клиента (игрока) есть свое окно, в котором вы можете управлять ими индивидуально, а также одно окно, где отображается информация с точки зрения сервера. Вы можете задать количество клиентов от 0 до 8, используя выпадающее меню под кнопкой **Test type**. Подробнее об отношениях «клиент–сервер» мы поговорим в следующей главе.

Опция **Team Test** позволяет вам и другим пользователем с доступом **Team Create** совместно проводить тестирование в среде, очень похожей на среду локального сервера. Этот метод лучше, чем **Local Server**, так как вам нужно управлять лишь одним персонажем, вы можете проверить правильность загрузки внешности персонажей и можете использовать API, для которого требуются свойства реального игрока вроде **UserId**. **UserId** – это уникальный идентификационный номер, который есть у каждого пользователя Roblox. Как только вы запускаете **Team Test**, иконка **Start** меняется на **Join**, а также появляется красная кнопка выключения, позволяющая завершить сеанс.

Инструменты на вкладке **Emulation** позволяют вам увидеть, как игра будет выглядеть для игроков, заходящих в игру с разных устройств и из разных регионов. Этот инструмент позволяет проверить, правильно ли пользовательский интерфейс в вашей игре масштабируется на дисплеях разных размеров и правильно ли выполнена локализация под регион игрока. Начнем с эмулятора устройства. Во время теста, помимо изменения окна просмотра в Studio, схема управления тоже подстроится под выбранную платформу, позволяя вам оценить взаимодействие с мобильными

устройствами, Xbox и компьютером. Второй инструмент – это эмулятор игрока. Вы можете изменять регион игрока и политики, действующие на этого пользователя, чтобы вносить необходимые изменения.

Настройка Studio для облегчения рабочего процесса

Система вкладок Roblox Studio – это удобная и гибкая система организации доступных вам инструментов. Когда вы наберетесь опыта в разработке, вам может захотеться поудобнее расположить инструменты, которые вы используете чаще других. Щелкнув по стрелке вниз рядом со вкладками Studio, как показано на рис. 2.17, вы сможете добавлять или удалять определенные действия на панель **Quick Access Toolbar** (Панель быстрого доступа).

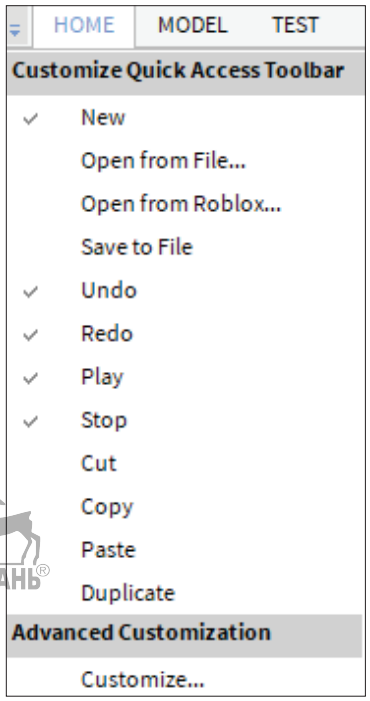


Рис. 2.17. Список настроек для панели быстрого доступа

Здесь вы увидите список основных действий, которые можно добавить или убрать с панели инструментов, а если вы нажмете кнопку **Customize...**, откроется небольшое диалоговое окно, в котором вы увидите практически все связанные с кнопками действия, которые можете выполнить в Studio. Изменив конфигурацию кнопок по умолчанию, вы сможете повысить эффективность своей работы в Studio и разработки, поскольку теперь вам нужно будет меньше переключаться между страницами.

Настройка панели быстрого доступа может повысить эффективность разработки, но это еще не все. Существует также множество надстроек, позволяющих выполнять рутинные или сложные задачи одним щелчком мыши. На странице **Toolbox** или **Library** щелкните по вкладке **Plugins**, где вы увидите множество плагинов, позволяющих отображать некоторую информацию или выполнять некоторые действия. Как мы говорили ранее в разделе «*Библиотека и магазин аватаров*», важно осторожно выбирать плагины для установки (и вообще, это касается всех ресурсов). Плагины могут выполнять программы в обход мер защиты Roblox и изменять в Studio практически все. Это не означает, что плагины вообще нельзя использовать. По мере вашего роста они могут стать необходимы, и в сообществе есть много разработчиков, занимающихся созданием качественных плагинов. Перед установкой плагина просто следует проверить, что он используется большим числом пользователей и имеет хороший рейтинг.

После изучения этого раздела вы можете начать добавлять в рабочем пространстве экземпляры и экспериментировать с ними, можете прове-

рить, правильно ли работают ваши системы на разных платформах и у разных игроков, а также просмотреть все возможности Studio и определить, какие функции для ваших проектов нужнее. Эти навыки будут особенно важны в следующей главе, когда мы начнем использовать Studio для создания своих первых программ. В целях всестороннего развития, прежде чем начать программировать, ознакомьтесь с документацией и другими ресурсами от Roblox, чтобы лучше узнать о процессе разработки в целом и об основных конструкциях программирования.

Использование ресурсов Roblox

Платформа Roblox в числе прочего обладает уникальной особенностью: изобилием документации, учебных пособий и других ресурсов для разработчиков. По мере изучения различных аспектов программирования вы обнаружите, что подробная документация по темам, с которыми вы работаете, невероятно ценна.

Учебники и ресурсы

Для удобства организации информации о языке Lua и других типах разработки Roblox создали отдельный раздел сайта со статьями и ссылками на API. Здесь вы можете узнать, какие функции и события есть у различных типов экземпляров, какие аргументы принимает каждая функция, а также ознакомиться с изменениями обновлений Studio и документацией по программированию и прочему связанному контенту.

Также на сайте вы найдете учебные пособия по нескольким основным игровым системам для начинающих. В этой книге мы тоже научимся программировать на Lua, начиная с самых простых концепций, но дополнительная практика и игровые шаблоны, которые вы можете изучить на сайте разработчика, помогут вам найти хорошие решения и применить их в своих собственных проектах. Для доступа к этим ресурсам не требуется входа в систему или других дополнительных шагов – достаточно перейти по ссылке **https://developer.roblox.com/**.

Форум разработчиков

Developer Forum (Форум разработчиков) – это раздел на сайте Roblox, посвященный исключительно обсуждениям разработчиков, совместной работе и быстрой помощи в разработке. На форуме вы найдете множество категорий для самых разных областей разработки, и решение большинства проблем, с которыми вы можете столкнуться при разработке проекта, можно найти в течение нескольких минут, если с этой проблемой уже кто-то сталкивался в прошлом.

Прежде чем создавать на форуме новые темы, вы должны пройти обучение и провести на сайте определенное количество времени. Это необходимо для защиты от распространения спама и нежелательного контента

по различным каналам. Если вы заработаете себе репутацию на форуме разработчиков, получая отзывы и лайки от других пользователей, то получите повышение до роли **Regular** (Обычный пользователь). Эта роль показывает, что у вас уже есть работы с форумом и Roblox в целом, и вам можно дать доступ к дополнительным каналам. Участники, которые активно и регулярно вносят свой вклад в развитие форума и создают контент, могут получить роль **Top Contributor** (Ведущий эксперт) или **Community Sage** (Мудрец сообщества). Эти роли зарезервированы, в основном выдаются администраторами вручную и являются признаком большого опыта пользователя. Эти роли также дают дополнительные разрешения.

Как мы уже говорили в предыдущей главе, общение в сети и поиск новых контактов с другими разработчиками – это самый лучший способ продвинуться как разработчик и повысить свой уровень репутации на платформе. На форуме нет излишней формальности, но полупрофессиональное поведение может помочь вам продвинуться быстрее. Для начала зайдите на форум разработчиков: **https://devforum.roblox.com/**.

С помощью форума разработчиков и сайта разработчиков вы не только обретете больше уверенности в себе как разработчик, но и начнете продвигать себя в сообществе, помогать другим или получать помощь.

Резюме

В этой главе вы узнали, как на странице **Create** на сайте Roblox создавать новые игры, изменять ее настройки, добавлять элементы монетизации и продвигать игру, чтобы ее могли увидеть миллионы пользователей. Более того, вы познакомились с Roblox Studio и теперь можете создавать новые экземпляры и изменять их свойства, управлять внутренними настройками игры, настраивать Studio для повышения эффективности разработки и управлять средой с помощью встроенных инструментов. Наконец, мы рассмотрели ресурсы, которые Roblox предоставляет своим разработчикам. На этих ресурсах вы можете пополнить свой багаж знаний, получить помощь в разработке и пообщаться с другими членами сообщества.

Следующая глава начнется со знакомства с языком Roblox Lua и общими конструкциями программирования, которые лягут в основу всей вашей будущей работы разработчика игр и программиста в целом. К концу следующей главы вы научитесь создавать свои собственные программы на языке Lua, писать код в правильном стиле и оптимизировать код.

Часть II

Программирование в Roblox

Эта часть – самая большая в книге и посвящена ее основной теме: программированию игр на Roblox. Вы научитесь программировать на языке Roblox Lua с нуля. Предполагается, что у вас нет никакого опыта в программировании. Затем мы применим полученные знания для создания реального игрового приложения.

В этой части представлены следующие главы:

- *глава 3 «Введение в язык Roblox Lua»;*
- *глава 4 «Сценарии программирования Roblox»;*
- *глава 5 «Пишем обби-игру»;*
- *глава 6 «Создание игры в жанре “Королевская битва”».*

Глава 3

Введение в язык Roblox Lua

Roblox Lua – это процедурный язык программирования, адаптированная версия языка **Lua**. Язык Lua был создан в 1993 г. из-за запретов на торговлю программным обеспечением в Бразилии, из-за которых пользователи не могли покупать специализированное программное обеспечение за пределами страны. Поэтому язык изначально наделялся широкими возможностями по настройке, а за основу был взят язык C, чтобы программисты могли создавать свои собственные реализации, используя Lua C API.

Цель этой главы – изучить вещи, необходимые для создания первых программ на Roblox Lua, и сделать шаг к становлению опытным программистом. Поскольку не предполагается, что у вас уже есть опыт в программировании, мы начнем с простейших понятий: переменных и других универсальных конструкций программирования. После еще нескольких глав вы сможете создавать в Roblox полноценные собственные игры.

В этой главе мы рассмотрим следующие темы:

- создание переменных и условные операторы;
- объявление и использование циклов;
- функции и события;
- стиль программирования и эффективность кода.

Поехали!

Технические требования

В этой главе вы будете полноценно работать в Studio, поэтому обратитесь к техническим требованиям, приведенным в разделе «Технические требования» главы 2. Обратите внимание, что самостоятельно проверять наличие обновлений Roblox Studio или Roblox Player не нужно, так как они будут обновляться автоматически при запуске. Поэтому все новые функции и меры безопасности всегда будут появляться у вас сразу.

Код, использованный в этой главе, можно найти в репозитории книги на GitHub по адресу **https://github.com/PacktPublishing/Coding-Roblox-Games-Made-Easy/tree/main/Chapter03**.

Создание переменных и условные операторы

В программировании **переменной** называют способ хранения в коде **данных разных типов**. Когда вы создаете переменную, вы присваиваете ей идентификатор, позволяющий обращаться к ней позже, что удобно. В большинстве языков программирования переменные **типизированы**, то есть тип переменной должен быть объявлен при ее создании. Например, если вы создаете переменную, в которой хранится число, в дальнейшем в ней нельзя хранить ничего, кроме чисел. В языке Lua переменные не типизированы. Это значит, что переменная, в которой изначально хранилось число, позже может содержать что-нибудь другое.

> **Важная заметка**
>
> Отметим, что существует также типизированная версия языка Lua, но в настоящее время она находится в стадии бета-тестирования и недоступна для широкой публики.

Прежде чем приступить к программированию, вам нужно изучить наиболее используемые типы данных.

Типы данных

Целые числа (**Integer**) – это любые целые числа, входящие в 64-битный диапазон. Если точнее, 64-битное целое число со знаком может принимать значения от -2^{63} до $2^{63} - 1$. В среде Roblox это не имеет особого значения, но вообще целые числа, не содержащие дробной части, лучше подходят для процессов, которым эта самая информация о дробной части не нужна. Например, если вы считаете яйца, снесенные курицей, лучше использовать целочисленное значение, так как курица не может снести половину яйца. В противном случае вам потребовались бы ресурсы для хранения несуществующей дробной части. На диаграмме ниже показано, как используются 64 бита в памяти для представления целого числа со знаком в двоичном формате.

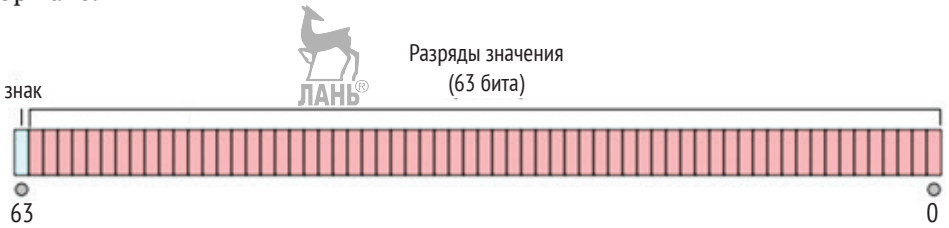


Рис. 3.1. У целого числа со знаком первый бит отвечает за знак, а остальные биты содержат значение

Вещественные числа (**Numbers**) могут иметь и целые, и дробные части, что позволяет хранить в них практически любое число от $(-1{,}8 \times (10^{308}))$ до $(1{,}8 \times 10^{308})$. Такое отличие в диапазоне возможных значений связано с тем, как в компьютерных системах форматируется сам тип данных. На диаграмме ниже показано, чем отличается хранение вещественных чисел от целых. Этот тип данных еще называют **числом с плавающей точкой двойной точности**, а иногда просто **double** (подтип **float**). Для представления чисел с плавающей точкой на двоичном уровне используется так называемый *технический стандарт IEEE-754*. Поскольку значения представляются в двоичном коде и получены из системы счисления с основанием 2, некоторые значения в них идеально представлены в нашем мире с основанием 10. Эти неточности называются **ошибками плавающей точки**. В областях вроде обработки данных эти ошибки имеют значения, но для среды разработки игр они настолько малы, что их обычно можно игнорировать.

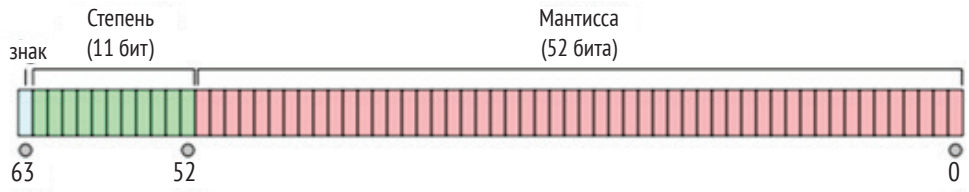


Рис. 3.2. Биты у чисел с плавающей точкой распределяются не так, как у целых, но их количество такое же

Логические значения (**Booleans**) – это простой тип данных с двумя значениями: `true` или `false`. Такие переменные часто называют **булевыми**. Важно отметить, что в Lua числа 1 и 0 имеют только числовые значения и не используются в булевой логике, как в других языках.

Строки (**Strings**) как тип данных в большинстве языков представляют собой массив отдельных символов. Но в Lua отдельные символы не являются типом данных. Строки используются для вывода букв, цифр и других символов клиенту, а также во внутренних задачах.

В языке Lua еще есть таблицы (**tables**), которые больше всего похожи на массивы в других языках. Этот тип данных может содержать бесконечное количество значений для некоторого элемента данных. В отличие от массивов в других языках программирования, таблицы больше похожи на списки, поскольку при инициализации не требуется задавать жесткий размер, а для выделения в таблице дополнительных ячеек не требуется упреждающее резервирование. Элементы таблиц имеют индексы от 1 до n, где n – количество элементов в вашей таблице. Следует отметить, что в большинстве других языков начальное значение индекса таблицы равно 0. Эти подходы называются стилями индексации с нулевым и единичным индексированием. Кроме того, как и любая другая переменная в Lua, таблицы не имеют

типа. Это довольно удобно, так как в других языках вы можете добавлять в массив только значения одного типа. Но это не значит, что можно превращать таблицы в свалку и заваливать ее разнородными данными.

Словари (**Dictionaries**) – это таблицы, в которых индексация выполняется по ключам, а не нумерованным индексам. Размер словаря не ограничен, а элементы в словаре не типизированы. Самое большое преимущество словарей – это возможность индексировать что угодно с помощью удобного ключа. Например, если вы хотите, чтобы все яблоки в **рабочем пространстве** имели один функционал, а все бананы – другой, вы можете использовать название фрукта в качестве ключа словаря, а в качестве значения – соответствующую функцию. Можно даже использовать в качестве ключа экземпляр фрукта, так как ключам не обязательно быть строками. Как мы упоминали ранее, Roblox не слишком удобен для написания кода в объектно-ориентированном стиле, поэтому создание словаря, содержащего разные типы данных с уникальными ключами, позволяет реализовать в Lua нечто вроде класса, что весьма удобно как для организационных, так и для функциональных целей.

Рис. 3.3. Вектор хранит информацию о направлении и величине

В математике **вектором** называется значение, у которого есть направление и величина. Векторы можно использовать для хранения данных о положении или вращении объекта. В Roblox векторы – это пользовательский тип данных, созданный Roblox, а не встроенный в Lua, в отличие от упомянутых ранее. Существует два типа векторов, с которыми вам предстоит работать чаще всего: `Vector3` и `Vector2`. У векторов типа `Vector3` есть компоненты X, Y и Z, которые используются для определения положения и ориентации объектов. Тип `Vector2` обычно используется только при работе с пользовательским интерфейсом или в других двумерных задачах. Этот тип содержит лишь компоненты X и Y. Векторы полезны для выполнения различных вычислений от расчета расстояния между двумя точками до более сложных операций, таких как перекрестные и скалярные произведения.

Тип `CFrame`, или система координат, – это еще один пользовательский тип, похожий на вектор, но с дополнительной информацией. Тип `CFrame` содержит данные о положении, а также девять элементов, объединенных в матрицу 3×3, описывающую вращение `CFrame`. Благодаря этому манипуляции `CFrame` в основном позволяют изменять относительно ориентации самой системы координат.

Экземпляры (**Instances**) – это пользовательский тип, к которому относится все, с чем вы можете взаимодействовать на панели **Explorer** в Studio. Разные типы экземпляров называются **классами**, и каждый класс обладает своими особыми свойствами. Вы можете посмотреть полный список

доступных классов на сайте разработчика: **https://developer.roblox.com/en-us/api-reference/index.**

> **Важное примечание**
>
> Вы уже много узнали о примитивных типах данных языка Lua, но в Roblox есть еще много пользовательских типов данных, которые мы затрагивать не будем. Полный список пользовательских типов данных Roblox можно найти здесь: **https://developer.roblox.com/en-us/api-reference/data-types**.

Теперь пришла пора научиться обрабатывать эти типы данных и присваивать их переменным. Переменные позволяют хранить данные и манипулировать ими, чтобы их можно было использовать в любой программе, которую вы пишете.

Определение переменных и работа с ними

Инициализация и изменение переменных в Lua – это сложный процесс, в котором для достижения желаемого результата используется множество различных операторов. Чтобы узнать значение переменной, вы можете использовать функцию `print()`. Многие программисты начинают карьеру со строки `print("Hello, World!")`. Функция `print()` полезна не только для поиска ошибок, но и для наблюдения за работой кода и его результатами, которые обычно не видны.

Числа

Это один из самых интуитивно понятных типов в Lua. Если вы хотите инициализировать переменную, то должны использовать ключевое слово `local`. Без этого ключевого слова переменная тоже будет работать, но будет видима для всего сценария, что почти никогда не нужно и считается плохим стилем. После слова `local` вы должны указать имя переменной. Имя переменной не может начинаться с небуквенных символов и может содержать только буквы, цифры и символы подчеркивания. Например, если вы хотите завести переменную, в которой будет храниться число 99, ваш код будет выглядеть так:

```
local myNumber = 99
```

Существует много различных операторов и специальных библиотечных функций, с помощью которых можно изменить значение переменной, но для первого примера мы просто увеличим значение на 1, чтобы получить 100. Для этого мы можем присвоить переменной ее же значение плюс 1 с помощью оператора сложения (+):

```
myNumber = myNumber + 1
```

Возможно, вы заметили, что слова `local` перед именем переменной уже нет. Дело в том, что `local` ставится только при первом объявлении переменной, а потом достаточно просто имени переменной. В зависимости от задачи может быть более практичным просто сразу присвоить переменной значение `100`. В этом случае вы можете присвоить переменной значение, как мы уже делали при ее инициализации (без слова `local`).

Lua поддерживает арифметические операторы, стандартные для большинства языков: сложение (+), вычитание (-), умножение (*), деление (/) и остаток (деление по модулю) (%). Для более сложных операций в Lua есть библиотека с той же функциональностью, что и стандартная библиотека `math` в языке C. В этой библиотеке есть тригонометрические функции, преобразования значений и некоторые специальные точные значения. Чтобы использовать эту библиотеку, мы можем применять ключевое слово `math`. В примере ниже мы возьмем из библиотеки `math` значение числа π:

```
myNumber = math.pi
```

Теперь поговорим о логическом типе данных.

Логические типы

Задать логическое значение легко, так как у него два варианта значений, определяемых словами `true` или `false` (истина или ложь). Оператор инициализации логического значения может выглядеть примерно так:

```
local myBool = true
```

Изменение значения этой переменной – это просто выбор между `true` или `false`. Существует простой способ присвоения переменной противоположного значения в одной строке без использования условного выражения. В этом поможет оператор `not`, который мы рассмотрим подробнее, как только перейдем к условным операторам. Оператор `not` возвращает значение, противоположное следующему за ним. Например, если мы хотим сменить значение переменной с `true` на `false`, то можем написать вот такую строку кода:

```
myBool = not myBool
print(myBool) -> false
```

Далее рассмотрим еще один примитивный тип данных, **строки**.

Строки

Чтобы объявить строку, используется та же процедура инициализации переменной, но текст нужно заключить в двойные кавычки. Можно использовать и одинарные кавычки, но двойные кавычки применяются чаще, если только они не содержатся в самой строке:

```
local myString = "Hello"
```

Если ваша строка содержит двойные кавычки, в Lua используется escape-символ обратной косой черты (\). В этом случае любой символ, который обычно является специальным, будет рассматриваться как текст внутри строки. Например, если кто-то в каком-то игровом диалоге говорит от третьего лица, вам нужно поставить двойные кавычки, например:

```
myString = "Он сказал \"Я не люблю яблоки!\""
```

Тот же символ обратной косой черты позволяет наделить обычный текст функционалом. Существует два символа, функциональность которых обеспечивается символом обратной косой черты, – это буквы `n` и `t`. Сочетание `\t` в строке добавляет символ табуляции, причем компьютерные системы считают ее одним символом. Сочетание `\n` добавляет переход на новую строку:

```
myString = "Слово\tотделено\tотступами"
print(myString) -> " Слово отделено     отступами "

myString = "Перенос\nна\nстроку"
print(myString) ->
"Перенос
на
строку"
```

Если строка разделена на несколько строчек, вам не обязательно использовать символ `\n`. Lua, в отличие от некоторых других языков, поддерживает многострочные строки. Для создания новых строк вы можете просто нажать клавишу *Enter*, и, кроме того, можно удобно форматировать строки по абзацам внутри программы. Чтобы заключить строку в абзац, используются двойные скобки, как показано здесь:

```
myString = [[Эта строка
может быть разделена
на несколько строк.]]
```


Один из чаще всего используемых способов изменения строковых переменных – это их конкатенация. Оператор `..` позволяет объединить строки:

```
myString = "Привет,"
myString = myString.. " мир!"
print(myString) -> "Привет, мир!"
```

Возможность добавления чего-то в конец строки особенно полезна, когда вы выводите игроку различную информацию через элемент пользовательского интерфейса. Например, если вы хотите объявить победителя раунда игры, нужно к строке добавить имя игрока:

```
local winnerName = "WinnerWinner"
myString = "Игра окончена! ". winnerName.. " выиграл раунд!"
print(myString) -> "Игра окончена! WinnerWinner выиграл раунд!"
```

Подобно библиотеке `math` для работы с числами, существует библиотека строковых функций для более сложных манипуляций и управления данными. Для доступа к этой библиотеке используется ключевое слово `string`. В ней есть функции для изменения регистра всех букв в строке, возможность разбивать строки в определенных точках и даже выполнять поиск в строке по определенному шаблону, что полезно для, например, создания панелей поиска в игре. Например, преобразуем все буквы в следующей строке в верхний регистр с помощью функции из библиотеки `string`:

```
myString = "эТо ГоРбАтАя СтРоКа."
print(string.upper(myString)) -> "ЭТО ГОРБАТАЯ СТРОКА."
```

Использования строк в числовой арифметике лучше по возможности избегать, но иногда и это бывает уместно. Всякий раз, когда строка используется там, где требуется число, Lua попытается автоматически преобразовать эту строку в число. Например, если вы попытаетесь прибавить строку `"50"` к числу `100`, все сработает правильно, как показано здесь:

```
print("50" + 100) -> 150
```

Но если строка, над которой вы пытаетесь выполнить такую операцию, содержит нечисловые символы, преобразование строки в число завершится ошибкой. Чтобы предотвратить это, вы можете проверить, состоит ли строка только из чисел, с помощью функции `tonumber()`. Если строку, переданную функции, нельзя преобразовать в число, возвращаемое значение будет равно нулю. Обычно ноль означает что-то несуществующее. Если мы попытаемся прибавить строку `"Привет"` к числу `100`, возникнет ошибка:

```
myString = "Привет"
print(tonumber(myString)) -> nil
local myNumber = 100 + myString -> "local myNumber = 100 +
myString:3: attempt to perform arithmetic (add) on number and
string"
```

Теперь рассмотрим **таблицы**, которые также применяются во множестве задач.

Таблицы

Таблицы по сути просты, но чуть сложнее в определении и манипулировании, чем другие типы данных, которые мы рассмотрели ранее, поскольку для большинства манипуляций нужна еще одна библиотека. Чтобы создать

новую пустую таблицу, вы должны указать после имени переменной пару фигурных скобок:

```
local myTable = {}
```

Не обязательно инициализировать именно пустую таблицу. Вы можете сразу добавить в нее данные при первом создании. Отметим, что элементы в таблице перечисляются через разделительный символ: запятую (,) или точку с запятой (;). Если игроку выдается квест на покупку продуктов из списка, вы могли бы инициализировать таблицу продуктов следующим образом:

```
local myTable = {"Сыр", "Молоко", "Бекон"}
```

Создав таблицу, вам нужно иметь возможность обращаться к отдельным элементам в списке. Если не использовать циклы, которые мы тоже рассмотрим позже в этой главе, обращаться к элементам можно только по отдельности. Помните, что в таблицах используется числовое индексирование с единицы. Все элементы из списка продуктов можно присвоить переменным или обратиться к ним напрямую, как показано в коде ниже:


```
local myTable = {"Сыр", "Молоко", "Бекон"}
local firstItem = myTable[1]
print(firstItem, myTable[2], myTable[3]) -> "Сыр Молоко Бекон"
```

Если нужно добавить или удалить элемент из таблицы, вам поможет библиотека `table`, обратиться к которой можно с помощью ключевого слова `table`. Эта библиотека позволяет вам изменять структуру таблиц, сортировать их, менять содержимое и расположение существующих записей в таблицах. Для добавления новых элементов в таблицу следует использовать функцию `table.insert()`. Этой функции требуется как минимум два аргумента, первый из которых – это целевая таблица, а второй – значение, которое вы хотите добавить в таблицу. Если вы передадите три аргумента, то первый аргумент – это целевая таблица, второй – желаемая позиция в таблице, а третий – добавляемое значение. При использовании функции с тремя аргументами важно помнить, что все элементы, начиная с позиции вставки, сдвинутся вправо. Кроме того, на значение индекса не накладывается ограничений, что означает, что индекс может быть отрицательным или лежать за пределами таблицы, но таких случаев лучше избегать. Ниже приведен пример добавления элемента в начало таблицы и элемента без указания позиции, то есть по умолчанию добавление выполняется в конец таблицы:


```
local items = {"Слон", "Полотенце", "Черепаха"}
table.insert(items, 1, "Камень")
table.insert(items, "Кот")
-> items = {"Камень", "Слон", "Полотенце", "Черепаха", "Кот"}
```

Вы не можете убрать все элементы с определенным значением или не подходящие по определенным критериям без использования циклов. Чтобы что-то удалить, вам нужно знать индекс удаляемого значения. Например, если приведенный ниже список должен состоять только из живых существ, нужно удалить элементы `Камень` и `Полотенце`. Удаление можно выполнить с помощью функции `table.remove()`. Важно отметить, что после удаления элемента из таблицы все другие элементы после него сдвинутся влево. Итак, если сначала убрать камень со стола, индексы всех остальных предметов в таблице будут на единицу меньше, чем раньше. Это можно увидеть в следующем коде:

```
items = {"Камень", "Слон", "Полотенце", "Черепаха", "Кот"}
table.remove(items, 1)
-> items = {"Слон", "Полотенце", "Черепаха", "Кот"}

table.remove(items, 2)
-> items = {"Слон", "Черепаха", "Кот"}
```

Чтобы узнать количество элементов в таблице, можно использовать оператор `#` и имя переменной таблицы. Тот же результат может дать функция `table.getn()`, хотя писать ее чуть дольше. Проверим, что эти методы возвращают одинаковый результат, выполнив приведенное ниже сравнение. Подробнее о сравнениях поговорим, когда дойдем до условного оператора:

```
print(#items == table.getn(items)) -> true
```

В следующем разделе мы узнаем о словарях.

Словари

Как мы упоминали ранее, словари – это таблицы, в которых в качестве индексов используются ключи, а не упорядоченные числа. По сути, ввод значений в словарь можно рассматривать как объявление переменной, но ключевое слово `local` не используется. Элементы в словаре можно расположить в виде таблицы, но чаще всего формируется структура, где у каждой записи есть своя строка. Разделительным символом для элементов может быть либо точка с запятой, либо запятая. Если бы в вашей игре было ресторанное меню, вы могли бы создать словарь, где ключом была бы перемена блюд, а значением – название блюда:

```
local menu = {
    appetizer = "Салатик";
    entree = "Бутерброд с ветчиной";
    dessert = "Мороженое";
}
```

Обращаться к этим элементам просто – достаточно прописать путь к желаемому значению. Предположим, что вы хотите сохранить основное блюдо в новой переменной:

```
local meal = menu.entree
print(meal) -> "Бутерброд с ветчиной"
```

Вся работа с элементами выполняется аналогично: задать или изменить элемент словаря можно так же, как и любую другую переменную:

```
menu.entree = "Бутерброд с индейкой"
```

Одно из преимуществ ключевой индексации в Lua заключается в том, что в качестве индексов используются не только строки. С помощью скобок ([]) вы можете применять в качестве индекса для значения любой тип данных. Это особенно полезно, если нужно, чтобы один тип данных имел прямую связь с другим при заданном значении. Например, если вам нужен список пороговых цен с их описательными строками, вы можете использовать число в качестве индекса. Имейте в виду, что для индексации ключей, не являющихся строками, нужно использовать скобки:

```
local prices = {
    [0] = "Бесплатно";
    [5] = "Дешево";
    [20] = "Средне";
    [50] = "Дорого";
}

print(prices[0]) -> "Бесплатно"
```

Есть и менее очевидная возможность – вы можете в качестве ключей использовать значения пользовательских данных. Например, можно связать начало координат рабочего пространства со строкой, числом или другой позицией – никаких ограничений.

Следует отметить, что таблицы тоже могут быть значением другой таблицы. Если некий объект существует внутри иного объекта того же типа, это называется **вложенностью**. Вы можете создавать древовидные структуры, вставляя таблицы друг в друга и обращаясь к элементам по цепочке ключей. Когда мы дойдем до модулей, вы увидите, что вложенные таблицы используются довольно часто для организационных и функциональных целей.

Например, если вы хотите перечислить некоторые основные характеристики неигрового персонажа (NPC) из игры, можно добавить их характеристики в таблицу, вложенную в общую таблицу NPC, и тогда информацию можно будет находить по имени NPC:

```
local units = {
    ["Пехотинец"] = {
        WalkSpeed = 16;
        Damage = 25;
    };

    Scout = {
        WalkSpeed = 25;
        Damage = 15;
    };
}
```

Пришла пора узнать все о векторах, а заодно понять трехмерную среду.

Векторы

Векторы – это объекты, у которых есть направление и величина. В программировании Roblox векторы используются для обозначения положения в трехмерных и двумерных средах, определения ориентации различных экземпляров, отображения направления `CFrame` и вычисления дополнительной информации о взаимном расположении объектов.

Объявление вектора аналогично созданию многих других пользовательских типов данных Roblox. Вы указываете имя типа данных, а затем функцию создания. Чаще всего при работе с векторами используется функция `new`. Для примера возьмем `Vector3`, однако у `Vector2` принцип работы останется таким же, но задается два компонента:

```
local myVector = Vector3.new(0,0,0)
```

Изменение векторных значений происходит не так, как у других рассмотренных типов данных. Это связано с тем, что арифметические операции выполняются над всеми компонентами вектора, причем поведение зависит от операции. Например, сложение двух векторов выполняется поэлементно, то есть компоненты результирующего вектора будут равны сумме компонентов исходных:

```
myVector = Vector3.new(1,3,5) + Vector3.new(2,4,6)
-> Vector3.new(3,7,11)
```

Однако векторная арифметика работает по-другому, если в операции присутствует **скаляр**, то есть любое значение, которое содержит величину, но не направление. Например, векторы можно умножать и делить на скаляры, но нельзя выполнять сложение или вычитание с ними. Единственным исключением из этого является случай, когда скаляр делится на вектор, и в этом случае деление выполняется по компонентам, а скаляр действует как числитель каждого элемента:

```
myVector = Vector3.new(2,4,6) * 2 -> Vector3.new(4,8,12)
myVector = Vector3.new(2,4,6) / 2 -> Vector3.new(1,2,3)
myVector = 2 / Vector3.new(2,4,6) -> Vector3.new(1,0.5,0.333)
```

Помимо работы с целыми векторами, вы можете получить из вектора отдельные значения. Для этого определим сразу три локальные переменные. Как правило, такой формат называется **кортежем**. Это объект, который образуется, когда функция возвращает несколько значений, не связанных в одну структуру. Тогда для хранения результата требуется больше одной переменной. Обращаясь к полям X, Y и Z вектора, мы можем получить числовые значения, которые можно будет использовать в вычислениях:

```
local x,y,z = myVector.X, myVector.Y, myVector.Z
```

Одна из самых часто используемых операций с векторами – это определение расстояния между двумя точками. Вы можете использовать формулу расстояния из библиотеки `math`, но есть более прямолинейный способ. Как мы упоминали ранее, все векторы имеют величину. Ее можно рассчитать и вручную, но в Roblox у всех векторов есть свойство величины, которое можно получить:

```
local magnitude = myVector.Magnitude
```

Чтобы вычислить расстояние между двумя позиционными векторами, их необходимо вычесть друг из друга; а величина нового вектора и будет расстоянием между ними:

```
local vector1 = Vector3.new(1,5,7)
local vector2 = Vector3.new(2,4,6)
local distance = (vector1 - vector2).Magnitude
print(distance) -> 1.73205
```

Подобно векторам, тип `CFrame` используется для хранения данных в трехмерных средах. О нем поговорим в следующем разделе.

CFrame

Пользовательский тип `CFrame` похож на векторы, но у него больше применений, так как в нем содержится больше информации. Чтобы объявить переменную `CFrame`, вы можете передать только данные о положении, как и у вектора. Используется конструктор `new` и координаты X, Y и Z, как показано ниже:

```
local myCFrame = CFrame.new(0,0,0) --CFrame.new() также используется
--для создания пустых CFrame.
```

В отличие от векторов, у CFrame есть матрица информации о вращении, которая описывает ориентацию объекта в пространстве с помощью векторов направления. Скорее всего, вам не придется работать с компонентами матрицы по отдельности, но менять ориентацию CFrame в целом приходится делать постоянно. Типичный способ задать ориентацию CFrame – использовать аргумент LookAt конструктора CFrame.new(). Если вы передадите ей точку начала и целевую точку, новый CFrame будет смотреть в указанном направлении, то есть вектор LookVector (передняя часть объекта) будет направлен в точку LookAt. Такой подход облегчает настройку направления объектов во время движения вперед. Чтобы проверить это, добавьте в рабочее пространство два объекта под названием Part1 и Part2. Расположите Part1 где угодно, а Part2 там, куда Part1 должна смотреть. После этого выполните следующий код, чтобы увидеть, как лицевая сторона Part1 указывает прямо на Part2:

```
local Part1 = workspace.Part1
local Part2 = workspace.Part2
Part1.CFrame = CFrame.new(Part1.Position, Part2.Position)
```

Как упоминалось ранее, одно из преимуществ работы с CFrame заключается в простоте программирования относительного движения. Если вы уже успели побаловаться с позицией объектов, то могли заметить, что позиция задается глобально, и не существует способа подвинуть объект в том направлении, куда он смотрит. Глобальная система координат называется **мировыми координатами**. Используя CFrame, вы можете двигать объекты относительно их ориентации, что важно для работы снарядов, дверей и даже транспортных средств. Тогда мы работаем в **координатах объекта**. Ниже приведен пример, в котором объект перемещается на одно деление в направлении взгляда:

```
myCFrame = myCFrame * CFrame.new(0,0,-1)
```

Обратите внимание, что при перемножении двух CFrame не выполняется умножение их компонентов, а происходящее скорее больше похоже на сложение. Реализовав приведенный выше код в цикле, особенно быстром, вы можете имитировать движение. Этот метод часто используется для программирования снарядов и экономит много ресурсов, поскольку *подвижная* часть закреплена и нет необходимости выполнять физические расчеты. Такое направленное движение используется почти для всех движущихся деталей, причем чаще всего вместо свойства Position.

Это не интуитивно очевидно, но вы должны представлять, какие оси позволяют задавать вращение в трехмерной среде. Как показано на следующем рисунке, оси X, Y и Z дают возможность управлять направлениями вращения не так, как вы могли ожидать.

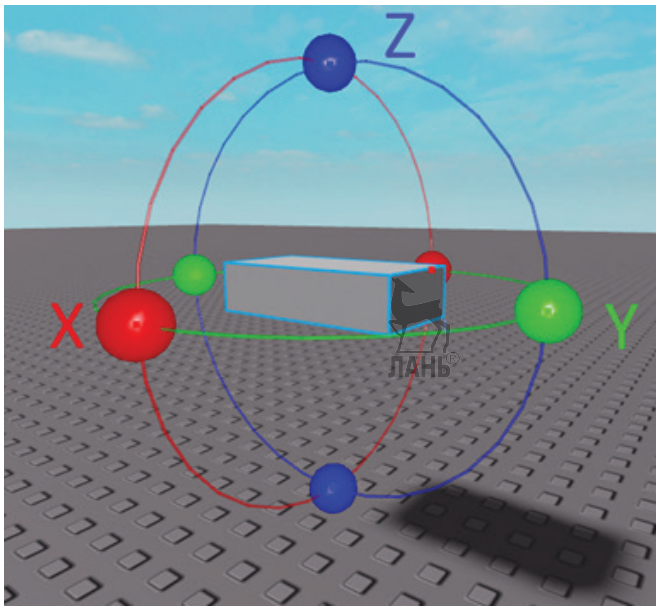


Рис. 3.4. Вид на переднюю часть детали с отображением осей вращения

Вы можете спросить, почему маркеры для оси Y расположены горизонтально, а для оси X – вертикально. Причина в том, что деталь вращается вокруг определенной оси. Поэтому маркеры вращения перпендикулярны осям направления.

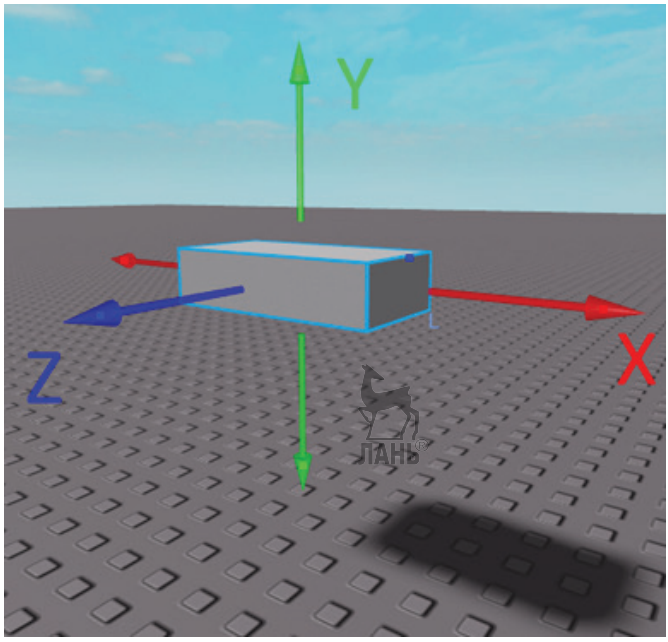


Рис. 3.5. На рисунке видно, что маркеры соответствуют осям вращения

Используя конструктор `CFrame.Angles()`, вы можете напрямую манипулировать ориентацией `CFrame`. Помните, что свойство `Orientation` задается в градусах, но в `CFrame` для работы с матрицей вращения используются радианы. За счет этого умножение `CFrame` на `CFrame.Angles()` по сути приводит к сложению компонентов вращения. Давайте рассмотрим пример, в котором мы развернем `CFrame` на 180°. Для этого нам нужно выполнить вращение вокруг оси Y и использовать радианы, как показано здесь:

```
myCFrame = CFrame.new() --No rotation
myCFrame = myCFrame * CFrame.Angles(0,math.pi,0) --вы также можете
--использовать math.rad(180) для math.pi
```

У `CFrame` есть много встроенных функций, но нет библиотеки, как у других типов Lua, которые мы уже рассмотрели. Существует множество функций и конструкторов (многие из которых даже лишние), у каждого из которых есть собственное применение для выполнения сложных операций и различных глобальных вычислений. Этих функций великое множество, но вы, скорее всего, не будете использовать многие из них, если дело не дойдет до создания систем с зубодробительными математическими вычислениями. Теперь предположим, что мы хотим получить ориентацию `CFrame`, который у нас только что получился. Как говорилось ранее, в `CFrame` для описания вращения используется матрица направленных векторов. Вы не можете просто индексировать ее значение и получить ориентацию. Для применения этой ориентации к деталям и выполнения вычислений нужна специальная функция `ToEulerAnglesYXZ()`. Она возвращает три числа X, Y, Z в виде кортежа. Эти значения – приближенное значение вращения `CFrame` в радианах.

Возьмем значение `CFrame` из предыдущего примера и сохраним его ориентацию в переменную `Vector3` без использования промежуточных переменных.

```
local orientation = Vector3.new(myCFrame:ToEulerAnglesYXZ())
-> Vector3.new(-0,3.1415925,0)
```

Получив такое значение, вы можете применить его самыми разными способами. Например, значение ориентации можно применить к детали в рабочем пространстве, умножив вектор на значение преобразования, чтобы получить ориентацию в градусах:

```
part.Orientation = orientation * (360 / (2 * math.pi))
```

Если вы хотите получить дополнительную информацию о других функциях и конструкторах `CFrame`, их можно найти в справке по API на сайте разработчика: **https://developer.roblox.com/en-us/api-reference/datatype/CFrame**.

Экземпляры

Экземпляры – это пользовательские элементы данных, которые создаются с использованием конструктора `new`. Существуют сотни различных экземпляров, но лишь некоторые из них можно создавать из сценариев из-за ограничений безопасности. Мы гораздо подробнее поговорим об экземплярах в следующей главе, а пока давайте просто создадим в рабочем пространстве новый объект и раскрасим его в яркий голубой цвет. Для этого вызовем метод `Instance.new()` и передадим ему в качестве аргумента имя класса:

```
local part = Instance.new("Part")
part.BrickColor = BrickColor.new("Cyan")
part.Parent = workspace
```

Вы уже знаете, как создавать и изменять различные данные после присвоения переменным, теперь нужно понять, как извлечь из них определенные значения, чтобы определить нужное поведение.

Условные операторы

Условные операторы, или **условные выражения**, используются в коде, когда нужно, чтобы то или иное поведение возникало только при определенных условиях. С их помощью можно извлекать информацию из данных и определять, что ваша программа должна делать для правильной обработки этих данных.

В основе условного выражения лежит `if`. Выражения состоят из трех элементов: ключевого слова `if`, условия, которое должно выполняться для выполнения кода, и ключевого слова `then`, которое завершает условие. Для примера в коде ниже приведено условное выражение, где в качестве условия дано просто слово `true`, то есть код под условием всегда будет выполняться:

```
if true then
    print("Выполнено")
end
```

В коде видно, что условное выражение заканчивается ключевым словом `end`. В языке Lua все, что работает как один блок кода, должно заканчиваться этим словом.

Вычисление условия в выражении подчиняется булевой логике, позволяющей работать с любым типом данных. Результатом вычисления выражения всегда является `true` или `false`. Для проверки условий вы можете использовать различные **логические операторы** и **операторы сравнения**. Как и во многих языках, в Lua два знака равенства (`==`) используются для проверки равенства значений, то есть это **оператор сравнения**. В качестве

примера предположим, что деталь должна иметь возможность сталкиваться только в случае, если ее прозрачность равна 1. Для этого сценария деталь определяется произвольно:

```
if part.Transparency == 1 then
    part.CanCollide = true
end
```

У этого оператора есть противоположный оператор *не равно* (~=). Как и предыдущий, этот оператор отношения используется для выполнения сравнений по явному значению.

Для сравнения конечных или бесконечных диапазонов чисел вы можете использовать операторы *больше* (>) и *меньше* (<). У них также есть варианты, включающие само значение, с которым выполняется сравнение, то есть *больше или равно* (>=) и *меньше или равно* (<=). Очень часто такие операторы применяются, чтобы проверить, жив ли игрок. Для примера предположим, что `char` – это персонаж игрока:


```
local humanoid = char.Humanoid

if humanoid.Health > 0 then
    print("Игрок бодр и весел!")
end
```

Оператор `not` используется для отрицания любого переданного ему значения. Мы уже видели на примере переключения состояния логической переменной, что оператор `not` возвращает противоположное от того, что ему передали. В условных операторах его применение чем-то похоже на `~=`:

```
if not part.Anchored then
    part.Material = Enum.Material.Neon
end
```


Логический оператор `and` используется для сравнения двух значений и требует, чтобы оба переданных ему значения были истинными. В следующем примере мы хотим убедиться, что в обеих переменных хранится значение `Fruit`. Это условие не выполняется, так как одна из переменных равна `Vegetable`, составной оператор даст `false`, и поэтому мы не увидим никакого вывода:

```
local item1 = "Фрукт"
local item2 = "Овощ"

if item1 == "Фрукт" and item2 == "Фрукт" then
    print("Оба - фрукты.") --ничего не выводится, так как условие не выполнилось
end
```

Логический оператор `or` требует, чтобы хотя бы одно из переданных ему значений было истинным. В данном примере переменная `item` равна `Vegetable`. Условие говорит, что `item` должна иметь значение `Fruit` или `Vegetable`, поэтому условие соблюдено и вывод появится:

```
local item = "Овощ"

if item == "Фрукт" or item == "Овощ" then
    print("Это продукт.") --вывод появится, так как одно из условий выполнилось
end
```

В среде Roblox при работе с экземплярами довольно часто используется метод `IsA`. Он позволяет проверить, относится ли экземпляр к указанному типу. В отличие от свойства `ClassName` экземпляра, метод `IsA` также проверяет **базовые классы**. Базовые классы – это некие группы для некоторых экземпляров, имеющих схожие характеристики. Например, классы **`Part`**, **`MeshPart`** и **`UnionOperation`** относятся к базовому классу **`BasePart`**. Теперь, если вам нужно проверить, что экземпляр, прозрачность которого нужно вывести на экран, является `BasePart`, поможет следующий код:

```
if myInstance:IsA("BasePart") then
    print(myInstance.Name.. ": прозрачность равна ".. myInstance.Transparency)
end
```

Как мы упоминали ранее, в одном условном выражении можно проверить несколько случаев, не используя несколько операторов `if`. Ключевое слово `else` расширяет возможности условного выражения и обрабатывает альтернативный случай, если первое условие не было выполнено. Давайте рассмотрим пример, когда для подъема тяжелого предмета требуется 100 силы, а для легкого – 50 силы. Обратите внимание, что переменная `heavy` будет хранить логическое значение `true` или `false`, и, следовательно, нам не нужно использовать оператор равенства:

```
local heavy = true
local strengthRequired = 0

if heavy then
    strengthRequired = 100
else
    strengthRequired = 50
end

print(strengthRequired) -> 100
```

Это мощный инструмент, но он не позволяет нам явно проверять дополнительные случаи, вместо этого задавая действие просто *во всех остальных*

случаях. Ключевое слово `elseif` используется, когда вы хотите проверить дополнительные случаи. Блоков `elseif` может быть сколько угодно, что позволяет создавать цепочки различных условий и случаев. При использовании оператора `elseif` вы можете также добавить оператор `else`, но он должен находиться в конце всего условного оператора. Давайте рассмотрим пример, в котором машине были предоставлены случайные продукты, и мы должны посчитать отдельно фрукты, овощи и любые другие предметы, которые к нам попали:

```
local numFruits = 0
local numVeggies = 0
local notProduce = 0

local item = "Фрукт"

if item == "Фрукт" then
    numFruits = numFruits + 1
elseif item == "Овощ" then
    numVeggies = numVeggies + 1
else
    notProduce = notProduce + 1
end
```

Наконец, существуют неявные условные операторы, реализуемые в виде логических выражений. Это позволяет обрабатывать разные случаи даже без оператора `if`. Цель приведенного ниже кода – присвоить строку переменной `isAnchored` в зависимости от того, закреплена ли деталь. Того же результата можно добиться с помощью оператора `if-else`, проще использовать логику:

```
local isAnchored = Part.Anchored and "Закреплена" or "Не закреплена"
```

Мы видим, что если деталь закреплена, часть оператора после `and` будет истинной и присвоит переменной значение `"Закреплена"`. Здесь используется логика сокращенных вычислений, то есть условное вычисление останавливается после выполнения одного условия. Если деталь не закреплена, присвоится значение `"Не закреплена"`. Видно, что оператор `or` в таких неявных выражениях работает аналогично ключевому слову `else`.

Условные операторы – это ключевой компонент программирования, который, как вы видели, имеет множество применений даже в самых простых системах разработки игр. В следующем разделе мы рассмотрим циклы. Циклы часто идут рука об руку с условными операторами, поскольку они позволяют передавать в условия целые наборы данных или повторять некоторый код многократно для достижения желаемого поведения.

Объявление и использование циклов

Циклы невероятно полезны в программировании, особенно при работе с наборами данных. Конечно, не стоит ожидать от программиста, что он будет руками перебирать тысячу индексов в таблице и извлекать данные в тысячу переменных, чтобы выполнить какую-то операцию. Для реализации подобного поведения и придумали циклы. Циклы после выполнения возвращаются к началу своего блока кода, если заданное условие все еще выполняется, то есть работают, пока дело не сделано.

Циклы for

Циклы `for` в основном используются для перебора наборов данных. В Lua это чаще всего таблицы или числа. В Lua есть два типа циклов `for`: **числовой** и **универсальный**. Основное различие между ними заключается в условии продолжения работы. В числовых циклах `for` некоторой переменной присваивается определенное начальное значение, конечное значение и необязательное значение шага. Если шаг не задан, Lua устанавливает его равным единице. Цикл `for` выполнит свой блок кода указанное количество раз, включая конечные значения диапазона. Кроме того, переменная цикла позволяет понимать, на какой по счету итерации цикл находится в данный момент. Как и в операторе `if-then`, в циклах используется пара `for-do`. В следующем примере выводятся числа от 0 до 10 с шагом 1. Обратите внимание, что шаг в этом случае можно было не задавать, но мы это сделали, чтобы было понятнее:

```
for i = 0, 10, 1 do
    print(i)
end
```

Рассмотрим более практический пример: с помощью числового цикла `for` находим сумму всех целых чисел от 1 до *n*. Проверить работу цикла можно, подставив то же значение *n* в формулу суммы прогрессии:

$$\sum_{i=1}^{n} i = 1 + 2 + 3 + \cdots + n = \frac{n(n+1)}{2}.$$

Рис. 3.6. Формула суммы первых *n* натуральных чисел

В этом примере следует обратить внимание на некоторые детали. Во-первых, поскольку цикл идет с шагом 1, шаг указывать не надо, чтобы сохранить хороший стиль кода. Во-вторых, мы вызываем функцию `tostring()`, которая работает так же, как вышеупомянутая `tonumber()`. Она нужна, поскольку функция `print()` автоматически преобразует другие типы данных в строки, а вы не можете прибавлять к строкам ничего, кроме чисел.

В этом примере мы найдем сумму, напечатаем ее, а затем напечатаем, равняется ли найденная сумма той, что мы вычислим по формуле:

```
local n = 17
local sum = 0

for i = 1, n do
    sum = sum + i
end

print(sum) -> 153
print("Функция работает = ".. tostring(sum == (n * (n + 1)) /
2))
-> "Функция работает = true"
```

Второй тип – универсальные циклы `for`. Из названия может показаться, что они не столь полезны и более специфичны, но вы, вероятно, будете использовать их чаще числовых и вообще любых других типов циклов. Универсальные циклы `for` позволяют перебирать индексы и значения, возвращаемые **функцией итератора**. В программировании функции итератора позволяют программистам перебирать элементы структуры данных, извлекая значения из нее и изолируя их. Эта изоляция, которую также можно увидеть при определении переменной в блоке кода, связана с понятием **области видимости**. Это означает, что на данные, объявленные в блоке, нельзя ссылаться вне этого блока. В следующем примере у нас есть словарь с именем `items`, который содержит три строки. Передав эту структуру данных в функцию `pairs()`, вы можете красиво отобразить каждый индекс и значение, возвращаемые итератором:

```
local items = {
    Animal = "Слон";
    Food = "Яйцо";
    Plant = "Цветок";
}

for index, value in pairs(items) do
    print(index, value)
end
```

Как мы упоминали ранее, индекс и значение, предоставленные итератором, не являются компонентами фактической структуры данных, которая передается в функцию `pairs()`. Этот означает, что вы можете манипулировать ими по своему усмотрению, не рискуя испортить саму обрабатываемую структуру. В следующем нам нужно удвоить попадающиеся нечетные числа, чтобы сделать их четными. Мы могли бы присвоить число, полученное в результате вычисления остатка, новой переменной, но в этом случае можно просто взять значение `value` напрямую. Обратите внимание, что, чтобы фактически изменить элемент таблицы, нужно применить индекс, выдаваемый итератором к самой таблице:

```
local values = {37, 60, 59, 20, 4, 10, 100, 75, 83}

for index, value in pairs(values) do
    value = value % 2

if value == 1 then --Odd number
        values[index] = values[index] * 2
    end
end

-> values = {74, 60, 118, 20, 4, 10 , 100, 150, 166}
```

В последнем примере универсального цикла мы будем перебирать экземпляры, являющиеся дочерними для рабочего пространства. Когда требуется изменять свойства объектов внутри рабочего пространства, вы, скорее всего, будете использовать цикл `for`. В следующем примере мы будем искать и закреплять все `BasePart`. С помощью метода `GetDescendants()` рабочего пространства мы можем рекурсивно перебрать все содержащиеся в нем экземпляры. Не имеет значения, какой именно объект является родительским для искомого. Отметим, что есть еще метод `GetChildren()`, который возвращает таблицу только непосредственных детей экземпляра. Обратите внимание, что переменной индекса мы умышленно назначили неинформативное имя «_». Это стилистический прием, который явно говорит, что в коде переменная не используется и не важна. Здесь мы проверяем, является ли объект `BasePart`, поскольку свойства `Anchored` у других объектов нет:

```
local items = workspace:GetDescendants()

for _, object in pairs(items) do
    if object:IsA("BasePart") then
        object.Anchored = true
    end
end
```

Рассмотрим еще один тип цикла, который будет выполняться всегда, пока выполняется условие.

Цикл while

Цикл `while` работает непрерывно, пока выполняется некоторое заданное условие. Его можно использовать для тех же целей, что и цикл `for`, но лучше всего представить, что это повторяющийся условный оператор. В следующем примере цикл `while` увеличивает значение на 1, пока это значение меньше 10:

```
local num = 0

while num < 10 do
    num = num + 1
end

print(num) -> 10
```


В разработке игр циклы `while` часто используются для запуска раундов игры или активации других событий, которые должны происходить через заданное время. Удобное, но интуитивно неочевидное поведение цикла `while` заключается в том, что ему не обязательно передавать логическое значение. Пока условие неложное (с точки зрения Lua, не `false` и не `nil`), цикл будет выполняться, а если само условие – это функция, то она будет выполняться, и цикл будет работать, если функция возвращает значение. При этом следует не забывать о хорошем стиле, который мы обсудим подробнее в разделе «*Стиль программирования и эффективность кода*».

У цикла `while` есть вариант `while true`. Этот цикл работает бесконечно, поскольку его условие всегда истинно. Бесконечный вариант цикла `while` используется для действий, которые не нужно завершать. В то же время он может привести к сбою сценария, поскольку цикл бесконечно накладывается поверх самого себя. Чтобы избежать этого, мы можем использовать функцию `wait()`. Она позволяет сделать в сценарии паузу и дать нужным вещам время на выполнение. Применение функции `wait()` не дает циклу спотыкаться о собственные ноги и позволяет ему работать с нужной вам скоростью. В следующем цикл `while` увеличивает переменную каждую секунду, отслеживая время, прошедшее с начала цикла. Обратите внимание, что числовой аргумент для функции `wait()` выражается в секундах:

```
local elapsedTime = 0

while true do
    wait(1)
    elapsedTime = elapsedTime + 1
    print(elapsedTime)
end
```


В следующем разделе мы узнаем еще один похожий тип цикла, который тоже можно использовать в некоторых задачах.

Цикл repeat

Цикл `repeat` выполняется до тех пор, пока не будет выполнено некоторое условие. Он очень похож на цикл `while`, но разница в том, что `while` запускается только при выполнении условия, проверяя условие перед запуском.

Цикл `repeat` же всегда выполняется по крайней мере один раз, проверяя условие завершения только после выполнения, подобно циклу `do-while` на других языках. Цикл состоит из слов `repeat` и `until`, где код, который должен быть выполнен, следует после слова `repeat`, а после `until` описывается условие выхода. В следующем цикле числовая переменная уменьшается до тех пор, пока ее значение не станет равным 0:

```
local num = 12

repeat
    num = num - 1
until num == 0

print(num) -> 0
```

Вооружившись циклами, вы можете обрабатывать большие наборы данных и создавать системы, реализующее последовательное, повторяющееся поведение. Далее мы поговорим о новом способе подачи данных в циклы, а также об их сжатии, если они используются часто.

Функции и события

В программировании **функцией** называется многократно вызываемый блок кода, обычно предназначенный для выполнения одной задачи. Функции позволяют сокращать трудозатраты на написание кода и избавиться от повторяющегося кода. В этом разделе вы узнаете о различных способах форматирования функций, а также о том, когда их следует использовать.

Функции в программировании

Как правило, функции применяются для выделения кода, на который впоследствии можно было бы легко ссылаться и многократно выполнять. Во многих языках программирования функции отделены от **процедур** и **подпрограмм**. Разница в том, что функция выполняет код и возвращает данные, а процедура просто выполняет некоторую задачу и не возвращает значение в точку вызова. Приведенная ниже функция предназначена для создания новой таблицы и случайного заполнения ее фруктами, овощами или непроизводственными товарами. Обратите внимание, что функция определена локально, как переменная. Вы должны указать ключевое слово `function`, а затем имя функции с парой круглых скобок. Эта часть функции называется ее **заголовком**. Чтобы выбрать случайный элемент, мы создаем новый случайный объект, используя конструктор `Random.new()`. Он создается в стиле, похожем на Java. Метод `NextInteger()` случайного объекта генерирует случайное целое число в диапазоне между минимальным и максимальным значениями, которое ему передано. Эта функция может быть связана с одним из примеров, приведенных в разделе «*Условные опе-*

раторы». Если вы уже уверенно ориентируетесь в коде, попробуйте создать счетчик продукции, используя условный оператор из предыдущего раздела, цикл и эту функцию:


```
local random = Random.new()

local function fillStoreSupply()
    local storeSupply = {}

    for i = 1, 10 do
        local ranVal = random:NextInteger(1,3)
        local item = (ranVal == 1 and "Fruit") or (ranVal == 2
         and "Vegetable") or "Shoe"
        table.insert(storeSupply, item)
    end

    return storeSupply
end

local supplyTable = fillStoreSupply()
```

При вызове функций для выполнения задачи часто бывает нужно передать им некоторую информацию. Для этого значения должны быть добавлены в оператор вызова функции и определены в строке, где объявлена функция. Значение, переданное в вызов функции, называется **аргументом**. Внутри функции эти данные называются **параметрами**. Приведенная ниже функция вычисляет факториал от предоставленного числа *n*. Факториал – это результат перемножения всех целых положительных чисел вплоть до заданного. Обратите внимание, что при вызове функции мы передаем в качестве аргумента число. Когда функция запускается, это значение автоматически присваивается *n*, и им можно далее манипулировать по мере необходимости:


```
local function factorial(n)
    assert(n == math.floor(n), "n должно быть целым числом.")

    local factorial = 1 --Пустой продукт должен быть равен 1

    while n > 0 do
        factorial = factorial * n
        n = n - 1
    end

    return factorial
```

```
end

print(factorial(12)) -> 479001600
```

Вы могли заметить, что мы используем функцию `assert()`. Подобно функции `throw()` в Java, с ее помощью вы можете выдать ошибку и завершить процесс, если какое-то условие не выполняется. Второй аргумент – это строка, которая отправляется на выход и выглядит как обычное сообщение об ошибке.

Если вы не знаете, сколько аргументов нужно передать функции, можно создать **функцию с переменным числом аргументов**. Функции с переменным числом аргументов похожи на обычные функции, но могут принимать любое количество аргументов в виде кортежа. Приведенная ниже функция возвращает сумму всех переданных ей чисел. Обратите внимание на использование трех точек (...). В эти точки попадают любые аргументы, передаваемые функции, которые затем помещаются в таблицу для будущей обработки. В следующем блоке кода функции передается случайное количество числовых аргументов, а функция вычисляет их сумму.

Параметры складываются, и возвращается одно значение:

```
local function sum(...)
    local args = {...}
    local sum = 0

    for _, number in pairs(args) do
        sum = sum + number
    end

    return sum
end

local num = sum(7, 9, 12, 3, 2, 6, 13)
print(num) -> 52
```

Вы уже могли узнать, что все циклы, которые мы рассмотрели, задерживают работу программы. Цикл может во время запуска приостанавливать текущий поток, и пока цикл не завершится, любой следующий за ним код не выполнится. В программировании для решения этой проблемы используется **многопоточность**. В Lua для создания нового потока в сценарии применяется функция `spawn()`. В приведенном ниже примере цикл `while` срабатывает один раз в секунду и выводит прошедшее время. Цикл работает аналогично второму примеру, представленному в разделе «*Цикл while*», но с другим условием. В таких случаях код после цикла `while` не выполняется, так как нет условий, которые привели бы к завершению цикла.

Тем не менее он находится в функции spawn(), поэтому для цикла или любого другого содержащегося кода создается новый поток, а остальная часть содержимого сценария в основном потоке не будет препятствовать выполнению:

```
local elapsedTime = 0

spawn(function()
    while wait(1) do
        elapsedTime = elapsedTime + 1
        print(elapsedTime)
    end
end)

print("Код после цикла все еще может выполняться!")
```

Отметим, что иногда Lua ведет себя неожиданным образом, если в стек объединены несколько потоков. Чтобы предотвратить это, по возможности избегайте использования функции `spawn()`.

Рекурсия

У функций есть необычайно ценное свойство – они могут вызывать сами себя. Правильно организовав структуру кода, вы можете создать цикл, который называется **рекурсией**. Разница между циклом `while` и рекурсией заключается в том, что цикл возвращается к своему началу, а рекурсия вызывает сама себя, образуя стек. В программировании стеком называют структуру данных или, как в случае рекурсии, просто состояние чего-либо в вашей программе. Стек похож на стопку тарелок или блинов, в которой блин, который положили последним, будет съеден первым.

Чтобы показать, как работает стек, давайте вернемся к функции факториала, которую мы написали ранее. Цикл `while` тоже годится для этой задачи, но можно достичь того же результата, используя рекурсию. Обратите внимание, что в приведенной ниже функции вызов и заголовок функции остаются неизменными. Рекурсивный элемент проявляется в операторе `return`. Оператор `if` в первом случае просто возвращает 1, если *n* меньше 1, поскольку нельзя взять факториал от значений меньше 1. Если равно 0, его факториал также будет равен 1 (таковы правила). Следующий случай является наиболее важным: если число *n* больше 1, то оно умножается на значение, возвращаемое функцией факториала от *n* – 1. Как вы можете заметить, это приводит к тому, что функция складывается до тех пор, пока *n* не сойдется до 1. Это «базовый случай», когда вызов функции не выполняется:

```
local function factorial(n)
    assert(n == math.floor(n), "n must be a whole number.")
```

```
    if n < 1 then
        return 1
    else
        return n * factorial(n - 1)
    end
end

print(factorial(6)) -> 720
```

Чтобы лучше понять, как работает этот процесс, давайте рассмотрим пример предыдущего вызова функции факториала, подставив число 6 вместо *n*. Обратите внимание, что в каждом вызове *n* умножается на возвращаемое значение функции, и рекурсия продолжает работать до тех пор, пока не будет достигнут случай, когда функция не изменяется. Затем каждая функция останавливается и убирается из стека, возвращая свое значение в точку вызова. Когда этот процесс завершается, наша исходная функция возвращает окончательное значение туда, откуда она была вызвана:

```
6 * factorial(5)
6 * (5 * factorial(4))
6 * (5 * (4 * factorial(3)))
6 * (5 * (4 * (3 * (factorial(2)))))
6 * (5 * (4 * (3 * (2 * factorial(1)))))
6 * (5 * (4 * (3 * (2 * 1))))
6 * (5 * (4 * (3 * 2)))
6 * (5 * (4 * 6))
6 * (5 * 24)
6 * 120
720
```

Теперь вы должны лучше понимать концепцию рекурсии. Давайте рассмотрим другой практический пример. Когда мы присваиваем переменной существующую таблицу, вся таблица не копируется в новую переменную. Вместо этого переменной присваивается **ссылка** на таблицу. Ссылки позволяют экономить ресурсы, так как ссылочные переменные являются просто указателями на ранее объявленную таблицу. Этот указатель можно увидеть, выведя таблицу на экран. За счет этого поведения присвоение таблицы переменной и изменение какого-либо элемента в этой таблице изменит ее везде, где на нее ссылаются. Это легко проверить с помощью приведенного ниже кода. Видно, что когда переменной присваивается уже созданная таблица, все переменные содержат одну и ту же ссылку:

```
local function checkEquality(table1, table2)
    print("Variable 1: ".. tostring(table1))
```

```
    print("Variable 2: ".. tostring(table2))
    print("First and second variable same table = "..
tostring(table1 == table2))
end

local group = {"Ашитош", "Хэйден", "Софи"}
local groupClone = group
checkEquality(group, groupClone)
```

Если бы таблицы полностью клонировались при каждом присвоении переменной, библиотеки индексирования или любые другие структуры в виде больших таблиц стали бы отнимать слишком много ресурсов. Однако существуют сценарии, в которых вам может понадобиться клонировать таблицу или словарь. В некоторых случаях для этого можно использовать цикл `for`, но при наличии вложенных таблиц, как показано в конце раздела «*Словари*», потенциально может потребоваться бесконечное число таких циклов. Чтобы обойти это, мы можем еще раз использовать рекурсию. В следующем примере мы создаем копию таблицы элементов путем создания новой таблицы и добавления в нее каждого элемента по индексу и значению. В случае если клонируемое значение представляет собой таблицу, функция рекурсивно использует вложенную таблицу в качестве аргумента. По окончании работы новая таблица возвращается туда, откуда она была вызвана, после чего мы с помощью функции `checkEquality()` проверяем, что это две разные таблицы:

```
local items = {
    Egg = {fragile = true;};
    Water = {wet = true};
}

local function recursiveCopy(targetTable)
local tableCopy = {}

for index, value in pairs(targetTable) do
        if type(value) == "table" then
            value = recursiveCopy(value)
        end

        tableCopy[index] = value
    end

    return tableCopy
end
```

```
local itemsClone = recursiveCopy(items)
checkEquality(items, itemsClone)
```

Как и циклы, вызовы рекурсивных функций задерживают работу программы, но обычно они выполняются достаточно быстро, и на работу потока это не влияет. Если по какой-то причине ваша рекурсивная функция выполняется достаточно долго и вызывает заметные задержки выполнения остальной части сценария, вам, скорее всего, следует пересмотреть свою функцию, а не пытаться создавать новый поток. Дело в том, что любой код в новом потоке будет выполняться до того, как будет возвращено значение, что может привести к проблемам.

События и методы экземпляров

У экземпляров есть не только разные свойства, зависящие от их класса, но и уникальные методы и события, также зависящие от класса. Мы знаем, что базовые классы предоставляют одни и те же свойства своим экземплярам. Методы наследуются точно так же.

В следующих примерах мы продолжим работать с деталью в плейсе. Приведенный ниже метод возвращает массу детали, используя плотность детали из **`PhysicalProperties`** и его размер. Зная массу, мы можем рассчитать количество силы, необходимой для моделирования невесомости для этой детали. Для этого нам понадобится экземпляр `BodyForce`. Базовый класс `BodyMover` и его типы мы рассмотрим в следующих главах. Чтобы это сработало лучше, слегка оторвите деталь от земли и прыгните на нее или примените другую силу, и увидите, что она начнет двигаться. Убедитесь, что деталь не закреплена, иначе она не сможет двигаться:


```
local part = workspace.FloatingPart
local mass = part:GetMass()

local bodyForce = Instance.new("BodyForce")
bodyForce.Force = Vector3.new(0,mass * workspace.Gravity,0)
bodyForce.Parent = part
```

С методами мы уже разобрались. **События** – это некие обнаруживаемые сигналы, которые отправляются, когда в вашей игре что-то происходит. Эти сигналы можно легко связать с функциями, что позволяет каждому событию активировать определенное поведение в легко управляемой причинно-следственной связи. Когда сигнал срабатывает, мы говорим, что он был **запущен**, или **сработал**.

Для примера рассмотрим событие `Touched` объекта **`BasePart`**. Это событие запускается всякий раз, когда другой объект `BasePart` касается `TouchPart`, которая ждет сигнала от события. Для соединения сигналов с функциями используется метод `Connect()`. Синтаксис будет похож на функцию `spawn()`.

Вы можете использовать ключевое слово `function` и пару скобок. Многие события передают информацию в виде аргументов, которые вы захотите включить в свою функцию. Вы можете определить эти параметры так же, как при создании собственной вызываемой функции. Объект, к которому было совершено касание, является аргументом, который передает событие. Если вы хотите его использовать, его нужно определить в заголовке функции:

```
local Part = workspace.TouchPart

Part.Touched:Connect(function(hit)
    print(hit)
end)
```

В предыдущем примере функция была отформатирована так, что получилась встроена в событие. Для многих это предпочтительный метод работы, но если вы хотите, чтобы функция, которая используется в другом месте, также была связана с событием, то можете передать имя функции в метод `Connect()`. В следующем примере мы локально определяем функцию и отдельно объявляем, что функция должна вызываться, когда запускается событие `touched`. Обратите внимание, что первый параметр функции все еще здесь, и нам не нужно передавать его из того места, где возникло событие:

```
local Part = workspace.TouchPart

local function printHitName(hit)
    print(hit)
end

Part.Touched:Connect(printHitName)
```

В этой главе вы получили много полезных знаний, которые будут вам полезны и сейчас, и в будущем. Теперь вы знаете достаточно, чтобы начать писать свои собственные программы. Пора узнать, как писать программы с хорошим стилем и делать код эффективным.

Стиль программирования и эффективность кода

Написание хорошо стилистически оформленного кода повышает качество работы, а также готовит вас к работе в более профессиональной среде с другими людьми. Некоторые правила, которые мы рассмотрим, используются повсеместно в программировании, а некоторые относятся в первую очередь к Roblox.

Общие правила стиля

Читабельность – это важная характеристика хорошего стиля кода. Мало того, что другие люди, которые читают ваш код, должны понимать его, но и вам самим будет удобнее работать, если код будет понятным. Соблюдение чистого стиля кодирования также позволит вам лучше понимать другие факторы стиля, которые будет нужно реализовать. Есть способ сделать код наиболее читабельным – это использовать правильные отступы и соблюдать длину строки. Что касается длины строки, в большинстве курсов по программированию в колледже говорят, что нужно ограничивать строку сотней символов. В Roblox есть номера строк, но нет индикации номера символа в строке. Как правило, длина вашей строки не должна превышать размера окна просмотра, то есть пользователь не должен быть вынужден пользоваться горизонтальной полосой прокрутки. Возможно, следует более тщательно соблюдать правильные отступы, тоже ради удобочитаемости. Roblox Studio должен автоматически делать в коде отступ, когда вы нажимаете клавишу `Enter`. Важно, чтобы ваш код соответствовал примерам кода в этой главе, пока не начнете соблюдать это правило автоматически.

Когда речь идет о работе в команде программистов, комментарии в коде становятся бесценны и позволяют рассказать другим, что делает ваш код. Удобочитаемость для этой цели тоже нужна, но комментарии дают возможность более явно рассказать другим программистам, где используется блок кода, что ему нужно или какого поведения от него ожидать.

Мы недавно рассмотрели неявные условные операторы. Можно сказать, что есть ситуации, когда более целесообразно использовать логику прямо в объявлении переменной, чем использовать явный условный оператор.

Решение в конечном счете зависит от вас, важно лишь применять инструменты целесообразно. Если вам нужно описать более двух или трех случаев, лучше просто использовать условное выражение. Кроме того, не забывайте учитывать ранее упомянутые правила стиля, например длину строки.

Мы уже кратко упоминали об этом, но повторим: не группируйте разнородные данные в таблицах или словарях. Чтобы код был лучше организован, вы должны использовать таблицы для важных целей, а элементы данных в них должны быть хотя бы слабо связаны друг с другом. При необходимости лучше создать дополнительные таблицы и хранить в них другие данные, чем сваливать все в одну.

Специфичные для Roblox принципы

Оптимизация кода крайне важна и позволяет сделать так, чтобы у игроков из вашей потенциальной аудитории не было проблем во время игры. Мы рассмотрим различные способы делать игру максимально эффективной. Вам следует научиться по возможности использовать события, а не циклы. Например, если вы знаете, что выводимое на интерфейсе игроку значение

может быть изменено, нужно использовать событие, а не проверять актуальность информации с помощью цикла. Все дело в том, что ресурсы системы в этом случае используются только при запуске события, а в случае с циклами – постоянно.

У некоторых методов и функций Roblox есть недостатки. Эти методы стали устаревшими, и хоть они и работают, их не рекомендуется использовать в новых проектах. Они даже не отображаются в автозаполнении. В этой книге мы не используем устаревший контент, а при написании собственных программ вам стоит просмотреть документацию по тем инструментам, которые планируете использовать.

Рассмотренные правила помогут вам многократно улучшить свои способности программиста. Члены вашей команды будут более благодарны за вашу работу, а эффективное написание программ в следующих главах позволит вам лучше понять новый материал и использовать его в своих будущих проектах.

Резюме

В этой главе мы рассмотрели конструкции программирования, которые применяются в самых разных языках, а также уникальные инструменты языка Lua. Вооружившись этими знаниями, вы можете начать создавать свои собственные ментальные связи, поэкспериментировать с примерами из этой главы и написать свои собственные программы.

В следующей главе мы займемся программированием, почти полностью специфичным для среды Roblox. После этого соберем воедино все новые знания и начнем создавать первые системы, которые используются в играх, а затем перейдем и к самим играм.

Глава 4

Сценарии программирования в Roblox

Вы уже немало узнали о программировании в общем смысле, и теперь пришла пора изучить сценарии работы, которые касаются почти исключительно разработки игр на Roblox. В этой главе мы рассмотрим программирование в Roblox, сервисы Roblox, методики управления игроком, работы с физикой и другие особенности создания игр.

Здесь мы разберем:

- основы модели «клиент–сервер»;
- работу с сервисами Roblox;
- работу с физикой;
- добавление в игру усложнений.

Поехали!

Технические требования

Как и в предыдущей главе, работать нам предстоит только в Studio. Для эффективности работы в Roblox Studio и доступа к веб-сайту разработчика (который нужен для изучения некоторых вопросов) вам понадобится подключение к интернету. Весь код, использованный в этой главе, можно найти в репозитории книги на GitHub по адресу **https://github.com/PacktPublishing/Coding-Roblox-Games-Made-Easy/tree/main/Chapter04**.

Основы модели «клиент–сервер»

Модель «клиент–сервер» – это распределенная коммуникационная структура, используемая во многих областях компьютерных наук. В разработке игр именно этот тип связи применяется чаще всего, и сервер в этом случае выступает в роли провайдера и верификатора для клиентов, которые отправляют ему запросы.

Различные типы сценариев

Поскольку сервер и клиент – это разные системы с разными целями, во время программирования в Roblox для работы с клиентом и сервером мы должны писать разные сценарии. Кроме того, существует третий тип сценариев, к которому можно получить доступ и локально, и с сервера. Эти типы сценариев и их значки приведены на рис. 4.1.

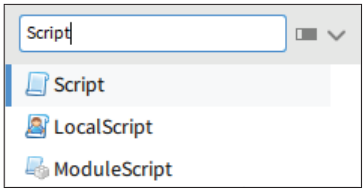


Рис. 4.1. В зависимости от модели данных, с которой вы работаете, можно использовать различные типы сценариев

Далее мы рассмотрим каждый из этих типов и их применение.

Сценарии

Сценарии используются для выполнения кода при работе с сервером, и, вероятно, именно с ними вы работаете в первую очередь. Забегая вперед, отметим, что они должны принадлежать рабочему пространству (`Workspace`) или `ServerScriptService`, как показано на панели **Explorer** (Проводник). По умолчанию в сценарии сразу есть команда `print("Hello World!")`. Поскольку сценарии работают на сервере, они чаще всего используются для общего управления игрой. Сценарий выполняется до тех пор, пока не возникнет ошибка, не будет достигнут конец потока или пока он не окажется в месте, где ему нельзя выполняться. Кроме того, если родитель сценария будет уничтожен, сам сценарий тоже будет завершен. Поскольку отдельные клиенты не могут влиять на сценарии, некоторые сервисы Roblox позволяют им использовать дополнительные ресурсы, но об этом будет рассказано позже в разделе «*Работа с сервисами Roblox*».

Локальные сценарии

Локальные сценарии похожи на серверные, но используются при работе с клиентом. Лишь локальные сценарии позволяют получить локальную информацию об игроке, например `CFrame` камеры, параметры мыши, управление от игрока и многое другое. Они выполняются, только будучи связанными с персонажем, `PlayerGui`, `Backpack`, `PlayerScript` или сервисом `ReplicatedFirst`. Клиентские сценарии в некотором смысле уязвимы, так как создаются и выполняются клиентами через *внедрение сценариев*. Поэтому локальные сценарии не имеют всех тех же разрешений на использование сервисов Roblox, что и серверные сценарии. Поскольку локальные сценарии выполняются на клиенте, они знают, какой клиент их выполняет. Например, если мы хотим получить информацию о мыши клиента, это можно сделать из локального сценария. При этом можно предположить, что переменная `player` уже содержит экземпляр объекта `player`:

```
local mouse = player:GetMouse()
print(mouse) --nil в серверном сценарии
```

Модули

Модуль – это особый тип сценария, доступ к которому должен осуществляться с помощью функции `require()`. Код модуля не будет работать до тех пор, пока не будет вызвана эта функция. Возвращаемый им результат кешируется, то есть код в самом модуле не будет запускаться несколько раз, если функция `require()` вызывается многократно, и в таблице модулей всегда будет одна и та же ссылка. В основном модули используются для хранения функций или другого кода, который должен быть доступен глобально, а не изолирован в отдельном сценарии. Возможно, вы уже видели в определениях переменных или функций глобальный идентификатор (`_G`), но обычно его использование считается дурным тоном. Доступ к модулям может получить как сервер, так и клиент, что делает модули полезными для общего функционала. Вам нужно лишь правильно его организовать. В следующем примере приведен модуль, содержащий простую функцию, и отдельный сценарий, вызывающий эту функцию, что отмечено комментариями в коде. Обратите внимание, что модуль возвращает таблицу, в которой хранится информация. Это должно происходить в модуле, так как упомянутая функция `require()`, по сути, вызывает модуль и возвращает его содержимое:

```
--модуль
local module = {}
module.initialize = function()

    print("Инициализирован")
end

return module

--сценарий
local mod = require(module)
mod.initialize()
```

Модули были задуманы так, чтобы работать в качестве классов, содержащих функции и другой код, но еще они часто используются для хранения таблиц данных, обычно со строковыми ключами. Покажем это на примере свойств NPC. С помощью цикла вы можете настроить каждого NPC в папке, используя их имена в качестве индексов, а затем задать их свойству `Humanoid` соответствующие значения, приведенные в таблице модуля. Обратите внимание, что в следующем примере сценарий обращается к значениям в модуле, как если бы это была таблица непосредственно в сценарии:

```
--модуль
local module = {}

module.Heavy = {
```

```
    MaxHealth = 500;
    Health = 500;
    WalkSpeed = 11;
}

return module

--сценарий
local mod = require(module)
print(mod.Heavy.MaxHealth)
```

Вкладка Script Menu

Теперь, когда вы узнали о различных типах сценариев, необходимо познакомиться с инструментами для работы с ними. В интерфейсе Studio есть вкладка **Script Menu** (Меню сценария), которая появляется только тогда, когда вы находитесь внутри сценария, – см. рис. 4.2.

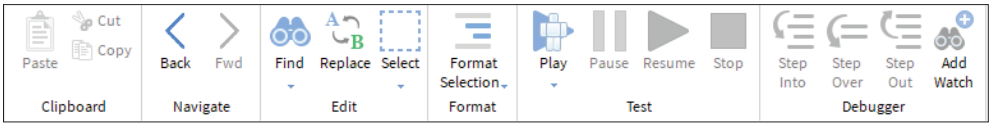


Рис. 4.2. Вкладка **Script Menu** содержит множество инструментов для работы со сценариями

С помощью кнопок в разделе **Navigate** (Навигация) вы можете перелистывать сценарии, открытые в настоящее время в Studio. Также для перехода к сценарию вы можете дважды щелкнуть по открытому сценарию или соответствующей вкладке.

В разделе **Edit** (Правка) находится несколько функций, первая из которых – инструмент **Find** (Найти). Его применение весьма разнообразно, но чаще всего выполняются команды **Find** (Найти) (*Ctrl+F*), **Find All** (Найти все) (*Ctrl+Shift+F*) и **Go To Line** (Перейти к строке) (*Ctrl+G*). Команда **Find** ищет введенную строку в текущем сценарии. Команда **Find All** (Найти все) ищет введенную строку во всех открытых сценариях, а результаты поиска отображаются в новом окне Studio, очень похожем на окно вывода. Наконец, команда **Go To Line** (Перейти к строке) открывает новое окно, куда вы должны ввести номер строки. После ввода редактор перейдет к нужной строке, и окно закроется.

Команда **Replace** (Заменить) открывает похожее на команду **Find** (Найти) окно, но ввести нужно будет две строки. Первая строка – это то, что вы хотите найти в своем сценарии, а вторая – это то, чем вы хотите заменить первую строку. К сожалению, эта функция работает только с одним сценарием, и вы не можете сразу заменить все нужные строки во всех сценариях вашей игры.

В Roblox есть и множество инструментов, используемых для отладки. Для знакомства с ними нам нужно изучить понятие точки останова. Выполнение кода приостановится, как только будет достигнута строка кода с точкой останова. Чтобы добавить точку останова, просто кликните по пустому пространству справа от номера строки в окне сценария, как показано на рис. 4.3.

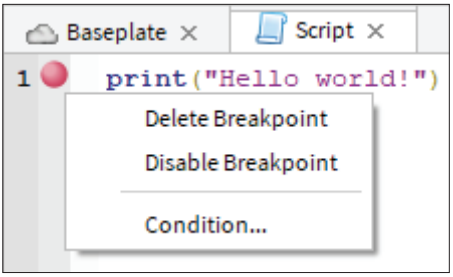


Рис. 4.3. Вы можете добавлять точки останова, чтобы остановить выполнение кода в нужном месте

Когда достигнута точка останова, вы можете продолжить выполнение кода построчно, используя команды **Step Into** (Шаг с входом в подпрограмму), **Step Over** (Шаг без входа в подпрограмму) и **Step Out** (Шаг с выходом из подпрограммы), расположенные в разделе **Debugger** (Отладка) и показанные на рис. 4.4.

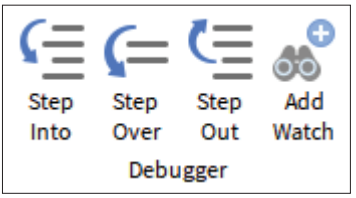


Рис. 4.4. В разделе **Debugger** есть различные действия, которые можно использовать для выполнения кода

Команда **Step Into** продолжит выполнять ваш код построчно, входя в любые функции или контейнеры кода, встреченные в коде. Команда **Step Over** (Шаг без входа в подпрограмму) пропускает блоки кода, если встречает их. Наконец, **Step Out** (Шаг с выходом из подпрограммы) немедленно выводит вас из блока кода, если вы входили в него, и перенесет вас на следующую строку после блока.

На вкладке **View** (Просмотр) находятся окна **Watch** (Наблюдение) и **Call Stack**. Оба эти окна содержат дополнительную информацию о выполняемом во время отладки коде. В окне **Call Stack** (Стек вызовов) вы можете посмотреть, какие процессы в данный момент находятся в стеке. Это позволяет понять, находитесь ли вы в данный момент в функции, просмотреть порядок вызова функций и увидеть, из какой строки эти вызовы выполняются. Данная возможность особенно полезна при работе с рекурсивными функциями, так как вы видите порядок вызовов. Также для анализа выполнения вы можете открыть окно **Watch** (Наблюдение), где перечислены точные значения переменных и типов всех выражений в вашем сценарии. Чтобы оценить достоинства этих инструментов, попробуйте выполнить рекурсивную функцию, аналогичную той, что приведена на рис. 4.5.

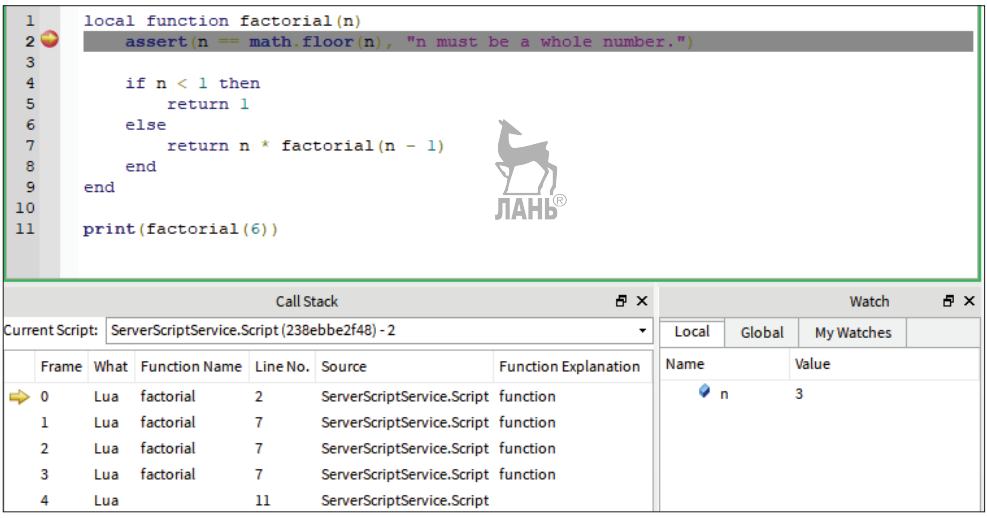


Рис. 4.5. Окна **Watch** и **Call Stack** полезны для анализа сложного кода

Наконец, кнопка **Format Selection** (Выбор форматирования) позволяет вам выровнять стиль и задать правильные отступы для кода. Эта функция появилась недавно и позволяет вам делать отступ только для выбранного в данный момент кода или для всего вашего сценария. Отступ автоматически определяется по расположению ключевых слов контейнеров кода.

Фильтрация

В 2018 году в Roblox внедрили опцию `FilteringEnabled` всегда включенной, чтобы защитить игры от эксплойтов. Эта опция, по сути, служит для создания структуры связи «клиент–сервер», в отличие от ранее используемого и менее проверяемого метода связи и репликации, когда клиенты могли напрямую сообщать серверу, что должно произойти. В результате такого серьезного системного изменения многие существующие игры перестали воспроизводиться, так как такой стиль взаимодействия в них не поддерживается. Сегодня все проекты в Roblox написаны с учетом этой опции и с использованием особых экземпляров объектов, называемых *удаленными*, которые применяются для облегчения общения между клиентом и сервером.

Удаленные события

`RemoteEvent` – это способ связи между клиентом и сервером, который используется, когда на запрос не требуется ответ. Часто клиенты отправляют серверу сигналы, чтобы сервер мог выполнить от их имени некоторую задачу. Для этого клиент должен запустить `RemoteEvent`, используя встроенный метод. Следует отметить, что этот метод может использоваться только тогда, когда вы работаете в клиентском сценарии, поэтому события `RemoveEvent` должны располагаться в месте, откуда они могут быть легко доступны кли-

енту и серверу, например в сервисе ReplicatedStorage, о котором мы поговорим подробнее позже. В следующем примере видно, что клиент вызывает метод FireServer() из предварительно определенного RemoteEvent:

```
RemoteEvent:FireServer()
```

Когда метод FireServer() вызывает запуск удаленного события, сигнал об этом может быть обнаружен только на сервере. Используя RemoteEvent и стандартный метод Connect(), вы можете запрограммировать на сервере любое поведение. Эта функция никогда не заставляет клиента ждать выполнения, даже если используется функция wait(), потому что клиентский и серверный сценарии – это два отдельных потока. Кроме того, клиент, запустивший удаленное событие, передается в качестве аргумента функции события при запуске события. Можно также передать дополнительные аргументы в виде кортежа:

```
RemoteEvent.OnServerEvent:Connect(function(player)
    --Операторы
end)
```

Такой односторонний способ связи через RemoteEvent может работать в обратную сторону, чтобы сервер создавал сигнал, который мог обнаружить только клиент. Клиент получает этот сигнал и что-то делает в ответ. Для этого можно использовать два метода. Вызвав метод FireClient() из RemoteEvent на сервере, вы передаете указанному игроку событие. Также вы можете отправлять событие сразу всем клиентам с помощью метода FireAllClients(). Это часто полезно для рассылки глобальных событий или уведомлений. Аргументы по-прежнему могут быть предоставлены клиенту в виде кортежа:

```
RemoteEvent:FireClient(player) --Обращение к одному клиенту
RemoteEvent:FireAllClients() --Обращение ко всем клиентам
```

Аналогично использованию OnServerEvent при обработке клиентского запроса, клиенты могут находить сигналы в локальных сценариях с помощью OnClientEvent и, аналогично, связывать их с функцией. В следующем примере клиент получает сигнал от RemoteEvent. Из кода видно, что событие отправляет всем клиентам сигнал на выполнение задачи, которая может быть выполнена только локально, а именно отправить уведомление по умолчанию с использованием StarterGui, сервиса Roblox. Подробнее об этом рассказывается в разделе «*Работа с сервисами Roblox*»:

```
--сервер
wait(5)
RemoteEvent:FireAllClients("Игра окончена!", "Синие выиграли!")

--клиент
```

```
local starterGui = game:GetService("StarterGui")
RemoteEvent.OnClientEvent:Connect(function(title, text)
    starterGui:SetCore("SendNotification", {
        Title = title;
        Text = text;
    })
end)
```

Очень часто RemoteEvent используются в случаях, когда игрок наносит урон другому игроку. Такие события, скорее всего, потребуются для всего оружия, используемого в вашей игре, если просчет попаданий выполняется на клиенте. Просчет попаданий в большинстве случаев выполняется именно на клиенте, так как задержка на сервере делает вычисления ненадежными. Клиент может воспользоваться эксплойтом и передать другие аргументы для RemoteEvent. Следовательно, важно по возможности получить обратную информацию от сервера. Не позволяйте клиентам сообщать серверу, сколько урона следует нанести и какое оружие используется в данный момент. Используйте **проверки разумности** переданных значений, чтобы защитить вашу игру в этой ситуации.

Удаленные функции

Удаленные функции (RemoteFunction) похожи на RemoteEvent, но обеспечивают возможность возврата информации через границу «клиент–сервер». Основное использование этого типа экземпляра – получение информации от клиента, которая обычно недоступна серверу, или наоборот. В следующем коде клиент делает запрос, используя метод InvokeServer() экземпляра RemoteFunction:

```
RemoteFunction:InvokeServer()
```


Форматирование функции, принимающей сигнал от RemoteFunction, отличается от случая с RemoteEvent. В RemoteFunction не используется связь сигналов и функций методом Connect(). Вместо этого функция напрямую назначается через **обратный вызов**. В следующем коде сервер получает сигнал от предыдущего запроса клиента с помощью события OnServerInvoke. Обратите внимание, что, как и RemoteEvent, клиент, сделавший запрос к экземпляру, становится параметром назначенной функции:

```
RemoteFunction.OnServerInvoke = function(player)
    return --опционально, для подпрограмм используется RemoteEvent
end
```

Направление связи может меняться, как и у RemoteEvent, что позволяет создавать запрос на клиента с сервера. Для реализации этого поведения в RemoteFunction используется метод InvokeClient():

```
RemoteFunction:InvokeClient(player)
```

В приведенном ниже коде показано, как отправленный запрос может быть получен клиентом. Обратный вызов `OnServerInvoke` экземпляра `RemoteFunction` используется для присвоения функции запросу. Здесь мы видим, что сервер запрашивает `CFrame` камеры клиента, к которой сервер обычно не имеет доступа. Обратите внимание, что функция, назначенная событию, может быть определена в других местах. Важно отметить, что хотя вы можете связать с `RemoteEvent` множество отдельных функций, с `RemoteFunction` можно связать лишь один обратный вызов, а попытка назначить другой просто перезапишет предыдущее назначение:

```
--сервер
local clientCamCFrame = RemoteFunction:InvokeClient(player)

--клиент
local cam = workspace.CurrentCamera

local function getCamCFrame()
    return cam.CFrame
end

RemoteFunction.OnClientInvoke = getCamCFrame
```

Как и проверка того, какие аргументы клиент отправляет в `RemoteEvent`, запрос информации от клиента с помощью `RemoteFunction` также может быть скомпрометирован, если игрок использует внедрение сценариев или любые другие способы взлома. Чтобы решить эту проблему, вы должны использовать проверки, а также запрашивать информацию у клиента только тогда, когда это точно необходимо.

Привязываемые функции и события

События `BindableEvent` и **функции `BindableFunction`** очень похожи на `RemoteEvent` и `RemoteFunction`, с тем лишь отличием, что после запуска их сигналы могут быть обнаружены только тем источником данных, который их запустил, то есть либо сервером, либо клиентом. Это полезно, если вы хотите создавать свои собственные сигналы для подключения функций, или хотите, чтобы сценарии напрямую взаимодействовали друг с другом.

Чтобы продемонстрировать пользу таких экземпляров, взглянем поближе на `BindableFunction`. В приведенном ниже коде мы фактически вызываем функцию, которая существует только в сценарии сервера, но не в том, из которого мы вызываем удаленный экземпляр. Обратите внимание, что с помощью `BindableFunction` мы можем восстановить значение, которое было возвращено функцией, без прямого индексирования функции, которую мы

хотим вызвать. Ключевые слова, использующиеся для вызовов и назначения обратного вызова BindableFunction, почти идентичны рассмотренным ранее, Invoke() и OnInvoke у BindableFunction и Fire() и Event, когда используется BindableEvent. Взгляните на выходной результат, в котором видно, как два серверных сценария взаимодействуют друг с другом для выполнения задачи:


```
--сценарий 1
local function sum(...)
    local sum = 0
    local nums = {...}

    for _, num in pairs(nums) do
        sum = sum + num
    end

    return sum
end

Function.OnInvoke = sum

--сценарий 2
local sum = Function:Invoke(2, 4, 11)
print(sum) → 17
```

Теперь вы знаете, как разные типы сценариев работают во взаимодействии между клиентом и сервером и как облегчить общение в среде с параметром FilteringEnabled. Далее вы узнаете о дополнительных функциях, которые можно добавлять в свои программы, используя сервисы Roblox.

Работа с сервисами Roblox

Пока мы писали только такие сценарии, в которых использовались встроенные функции языка или методы и события экземпляров. В Roblox для программирования более сложного поведения игры нужно использовать сервисы, а также дополнительные события и методы. Сервисы – это библиотеки, которые применяются для работы с различными аспектами вашей игры, а не с конкретными типами данных.

Сервис Players

Практически в каждой игре используется сервис Players. Его можно найти на панели **Explorer** (Проводник), где сохраняются все экземпляры player. Событие PlayerAdded очень часто применяется для определения того, что игрок присоединился к игровому серверу. Следующий код выводит имя

любого игрока, присоединяющегося к игре, и небольшое сообщение. Обратите внимание, что игрок, который присоединяется к игре, передается в параметры функции, связанной с событием:

```
local playersService = game:GetService("Players")

playersService.PlayerAdded:Connect(function(player)
    print(player.Name.. " присоединился к игре.")
end)
```

Существует также аналогичное событие для проверки выхода игрока из игры: событие `PlayerRemoving`. Обратите внимание, что эта функция выполняется до того, как экземпляр удаляется из сервиса, поэтому вы можете успеть получить информацию из экземпляра, который передается в качестве параметра функции:

```
playersService.PlayerRemoving:Connect(function(player)
    print(player.Name.. " покинул игру.")
end)
```

Вы можете и не показывать статистику игрока всем другим игрокам на сервере, но если ее показать, то это может простимулировать конкуренцию и заставить людей играть дольше. Для этого в Roblox есть встроенная система под названием `leaderstats`. Она используется для генерации списков лидеров во многих играх, поскольку для его реализации не требуются дополнительные функции пользовательского интерфейса. На рис. 4.6 с помощью `leaderstats` отображается статистика по количеству золота у игроков. При наличии большего числа игроков их имена будут отображаться по убыванию, начиная от игрока с самой высокой статистикой.

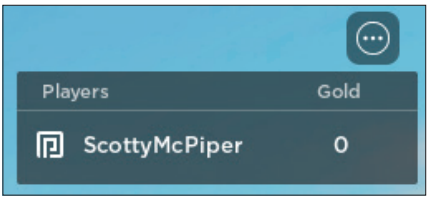


Рис. 4.6. Вы можете отображать информацию в списке игроков с помощью системы `leaderstats`

Давайте попробуем создать собственную таблицу лидеров для нового игрока, который присоединяется к игре. Чтобы система работала, у каждого игрока должна быть папка `leaderstats` для работы системы, поэтому лучше использовать событие `PlayerAdded` и с его помощью добавить необходимые элементы в экземпляр игрока.

Для хранения имени *leaderstats* может использоваться любой экземпляр, но концептуально имеет смысл применять что-то вроде папки. После создания и присвоения имени папка должна быть непосредственно привязана к экземпляру игрока, инициировавшего событие. Затем создаются экземпляры значений в зависимости от того, какой тип данных вы хотите отображать. В следующем примере мы создаем `IntValue`, которое со-

держит количество золота у игрока. В папку `leaderstats` можно добавить сколько угодно значений, но мы ограничимся первыми четырьмя. Кроме того, имена игроков будут отсортированы в списке по первой созданной статистике:

```
playersService.PlayerAdded:Connect(function(player)
    local folder = Instance.new("Folder")
    folder.Name = "leaderstats"
    folder.Parent = player

    local gold = Instance.new("IntValue")
    gold.Name = "Gold"
    gold.Parent = folder
end)
```

Также сервис `Players` часто используется при работе на клиенте. Чтобы определить, какой игрок выполняет локальный сценарий, вы должны проиндексировать свойство `LocalPlayer` сервиса. Если бы вы проиндексировали это свойство внутри сценария сервера, вам вернулось бы `nil`. После получения индекса игрока вам становится доступным множество функций и дополнительной информации:

```
local player = playersService.LocalPlayer --nil в сценарии рег.
print(player.UserId) --ID уникален для каждого пользователя Roblox
```

В последнем примере мы будем использовать метод `GetPlayers()` сервиса `Players` для перебора всех клиентов, находящихся в данный момент в игре. С помощью цикла мы устанавливаем свойство `WalkSpeed` всех игроков равным 30. Отметим, что структурированный таким образом цикл не должен останавливать работу, столкнувшись с неприятностями. Это связано с тем, что игрок может выйти во время выполнения кода, что вызовет проблемы, поскольку клиент все еще будет существовать в таблице игроков, возвращаемой методом `GetPlayers()`, а в игре его уже не будет:

```
local players = playersService:GetPlayers()

for _, player in pairs(players) do
    local char = player.Character

    if char then
        print(char.Name.. " бежит теперь со скоростью 30.")
        char.Humanoid.WalkSpeed = 30
    end
end
```

Сервисы ReplicatedStorage и ServerStorage

ReplicatedStorage и ServerStorage – это удобные места для хранения ресурсов в игре и управления ими. У каждого из них свое разное поведение и оптимизации при добавлении контента, но уникальных методов или свойств у них нет.

Активы в ReplicatedStorage доступны клиенту, но не визуализируются, что позволяет несколько повысить производительность, когда туда перемещаются ненужные активы в режиме реального времени. Как мы упоминали ранее, как локальные, так и серверные сценарии должны быть привязаны к определенным местам, чтобы их можно было запускать. Когда сценарии любого типа связаны со службой хранения, они не будут выполняться до тех пор, пока не будут перемещены в область, где это будет возможно. Такое поведение можно сравнить со свойством Disabled, которое стало равным false.

Когда активы содержатся внутри ServerStorage, они могут быть доступны только серверу. Подобное поведение позволяет использовать ServerStorage для хранения экземпляров вроде модулей (то есть тех, которые не используются клиентом). Кроме того, физические экземпляры из нашего рабочего пространства, которые хранятся в ServerStorage, не загружают память и не визуализируются на клиенте. Это обеспечивает больший прирост производительности по сравнению с ReplicatedStorage, хотя обычно эти сервисы используются для совсем разных целей.

Сервис StarterGui

В сервисе StarterGui собрано множество как основных, так и дополнительных и настраиваемых функций пользовательского интерфейса. Вы можете создавать графические пользовательские интерфейсы (GUI), при этом сервис выступает в качестве контейнера, а экземпляры, связанные с вашим пользовательским интерфейсом, отображаются на экране, если в Studio их вложить в сервис, как показано на рис. 4.7. При запуске игры ваш пользовательский интерфейс копируется отсюда и вкладывается в контейнер PlayerGui клиента. Более подробная информация о пользовательском интерфейсе и его непосредственном влиянии на сценарии будет приведена в главе 6.

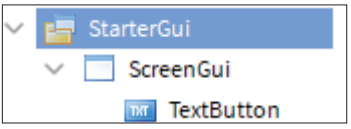


Рис. 4.7. Привязка экземпляров GUI к StarterGui делает их видимыми на экране в Studio

StarterGui также используется в качестве интерфейса для взаимодействия с основными функциями графического интерфейса и управления ими. В следующем примере сервис используется для полного отключения некоторой части основного пользовательского интерфейса с помощью метода SetCoreGuiEnabled(). Чтобы применить его, просто укажите, на какой

именно интерфейс вы хотите повлиять, и передайте логическое значение, соответствующее включению или отключению:

```
local starterGui = game:GetService("StarterGui")
starterGui:SetCoreGuiEnabled(Enum.CoreGuiType.PlayerList,
false)
```

Сервисы StarterPack и StarterPlayer

Сервисы `StarterPack` и `StarterPlayer` позволяют определить поведение игроков при инициализации и параметры их порождения. `StarterPlayer` дает возможность определить свойства по умолчанию для экземпляра `Humanoid`, используемого в качестве игрового персонажа, путем изменения свойств самого сервиса, как показано на рис. 4.8. Следует отметить, что при изменении поведения на вкладке **World** (Мир) в меню **Game Settings** (Настройки игры) физически изменяются следующие значения:

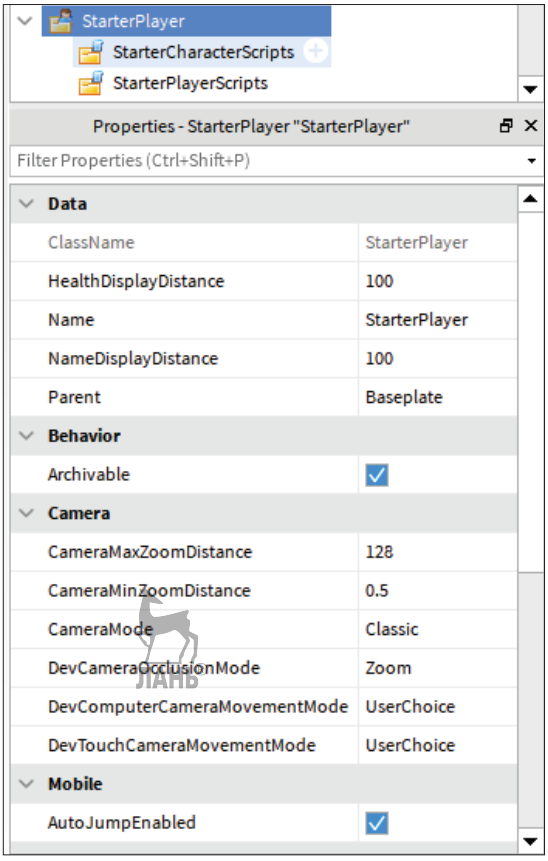


Рис. 4.8. Изменение свойств `StarterPlayer` влияет на поведение проигрывателя по умолчанию

Кроме того, вы можете хранить сценарии в двух местах с разным поведением. Сценарии, вложенные в StarterPlayerScripts, запускаются в момент присоединения игрока к игре без сброса настроек. Это место отлично подходит для *локальных обработчиков*, то есть локальных сценариев, которые управляют всеми последующими модулями. Более подробно мы рассмотрим физическую инфраструктуру сценариев в следующей главе. Когда локальные или серверные сценарии вкладываются в StarterCharacterScripts, они помещаются в модель персонажа игрока при каждом порождении. Экземпляры StarterCharacterScripts и StarterPlayerScripts не имеют никаких дополнительных методов или свойств.

С точки зрения сервиса StarterPack, все вложенные в него элементы помещаются в экземпляр Backpack игрока в момент его порождения. Чаще всего сюда кладутся Tools, которыми игроки должны пользоваться по умолчанию. Любой Tools, который вы хотите добавить позже, можно поместить непосредственно в Backpack игрока. Как и StarterCharacterScripts, StarterPack сбрасывает свое содержимое всякий раз, когда игрок возрождается, и если вы хотите, чтобы сценарий выполнялся при каждом возрождении, то вложите его в StarterPack.

StarterPack сам по себе не имеет каких-либо уникальных свойств или методов.

Сервис PhysicsService

PhysicsService отвечает за управление столкновениями между объектами в вашей игре с помощью **групп коллизий**. В игре может быть до 32 групп коллизий с заданными правилами взаимодействия друг с другом.

В следующем коде мы создаем группу коллизий для игровых персонажей, чтобы игроки не могли пересекаться друг с другом. Это полезно для игр, в которых столкновение игроков приводит к негативным последствиям. Группы столкновений создаются с помощью метода CreateCollisionGroup() сервиса PhysicsService. Этот метод принимает уникальную строку, которая позднее будет использоваться в качестве ключа. Мы не хотим, чтобы игроки могли пересекаться. Все игровые персонажи будут находиться в одной группе коллизий, и мы можем настроить группу так, чтобы у объектов не было пересечений. Для этого мы можем использовать метод CollisionGroupSetCollidable(), который принимает две группы столкновений и логическое значение, определяя, возможна ли коллизия. Далее, создав функцию с помощью события player.CharacterAdded, которое часто используется в событии PlayerAdded, мы переберем в цикле все BaseParts экземпляра. Все части назначаются группе "Players" столкновений с помощью метода SetPartCollisionGroup(), который принимает в качестве аргументов BasePart и имя группы коллизий:

```
local physics = game:GetService("PhysicsService")

physics:CreateCollisionGroup("Players")
```

```
physics:CollisionGroupSetCollidable("Players", "Players",
false)

player.CharacterAdded:Connect(function(char)
    for _, part in pairs(char:GetDescendants()) do
        if part:IsA("BasePart") then
            physics:SetPartCollisionGroup(part, "Players")
        end
    end

    print(player.Name.. " добавлен в группу!")
end)
```

Сервис UserInputService

`UserInputService` – это служба, которая используется в локальных сценариях для обнаружения физического ввода от клиента. Его можно использовать для чтения данных с мыши, клавиатуры или любого другого устройства ввода игрока, а также для определения режимов ввода.

Допустим, нам нужно обнаруживать ввод с клавиатуры от клиента. Вы можете проверить, использует ли игрок клавиатуру, просмотрев значение свойства `UIS.KeyboardEnabled`. Чтобы обнаружить ввод любого типа, вы можете использовать событие `InputBegan`. В нем указано, каким был ввод и взаимодействовал ли пользователь с каким-либо пользовательским интерфейсом в момент ввода. В следующем примере мы создадим функцию взаимодействия, вызываемую нажатием клавиши *E* на клавиатуре игрока. Мы не хотим, чтобы эта функция запускалась, если клиент нажимал клавишу взаимодействия во время написания сообщения в чате, поэтому немедленно выйдем из функции, если клиент взаимодействовал с какими-либо элементами пользовательского интерфейса. Далее, поскольку мы ищем ввод с клавиатуры, можем проверить перечисление `KeyCode` и сравнить ввод с нужным символом:

```
local UIS = game:GetService("UserInputService")

UIS.InputBegan:Connect(function(input, typing)
    if typing then return end

    if input.KeyCode == Enum.KeyCode.E then
        print("Клиент нажал клавишу E!")
    end
end)
```

Мы рассмотрели большинство сервисов, которые могут быть полезны новичку, но вы также можете посетить веб-сайт разработчика, почитать о

других сервисах и получить дополнительную информацию об их приложениях. Найти нужный сервис можно с помощью инструмента поиска вашего браузера: **https://developer.roblox.com/en-us/api-reference/index.**

Сервисы лежат в основе создания любой игры в Roblox, и теперь, когда вы научились их использовать, вы можете создавать гораздо более комплексные системы или даже применять события и методы более низкого уровня, которые без определения нужного сервиса были бы недоступны. В следующих главах, когда вы будете создавать свои собственные игры, эти сервисы будут использоваться повсюду – от управления игроком до игровых механик.

Работа с физикой

В зависимости от того, занимаетесь вы разработкой внешнего или внутреннего интерфейса проекта (то есть работаете с инфраструктурой кода или видимой для игроков частью), работа с физикой может быть частью ваших повседневных задач. Физика напрямую связана с математикой, и в целом бэкенд-программисты больше работают с математикой и CFrames, но фронтенд-программист тоже вынужден работать с физическими силами в рабочем пространстве. Существует несколько подходов к работе с физикой: свойства, ограничения и перемещение тел.

Ограничения

Базовый класс `Constraint` состоит из экземпляров, которые можно использовать для реализации различного физического поведения в игре. Большинству ограничений для работы требуются экземпляры `Attachment`. Они должны быть вложены в экземпляр `BasePart` и являют собой точки в пространстве, на которую могут ссылаться ограничения.

Чтобы упростить процесс настройки ограничений, вы можете использовать подменю **Constraints** (Ограничения) на вкладке **Model** (Модель) в Studio. В этом меню вы можете создавать новые ограничения, не выполняя вложение самостоятельно. Кроме того, здесь вы найдете удобный раскрывающийся список всех экземпляров ограничений, как показано на рис. 4.9. Вы также можете просматривать ограничения в рабочем пространстве, даже те, которые обычно невидимы, включив опцию **Constraint Details** (Детали ограничений).

Ограничение **Rod** (Стержень) используется для сохранения заданного расстояния между объектами и удерживает их на одной линии жесткой осью, проведенной между точками **Attachments** (Присоединения). Объекты, будучи на одной оси друг с другом, могут свободно вращаться вокруг своих точек **Attachments** (Присоединения). Основные свойства этого экземпляра – это **Thickness** (Толщина) и **Length** (Длина), где **Length** (Длина) – это расстояние между двумя объектами, **Thickness** (Толщина) – визуальная толщина ограничения. Вы также можете просмотреть текущее расстояние

между точками, а впоследствии и объектами, с помощью защищенного от записи свойства `CurrentDistance`.

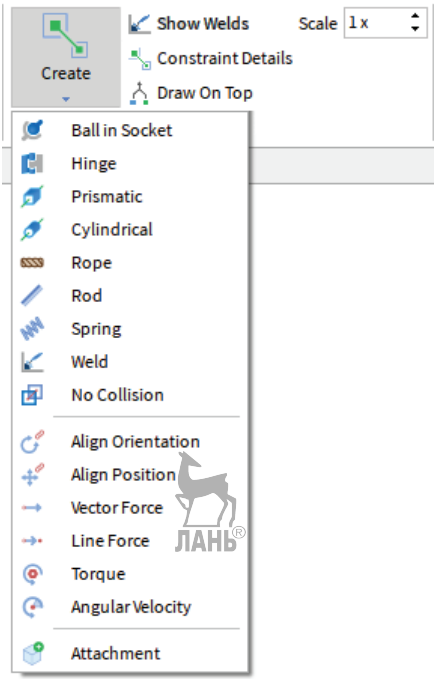


Рис. 4.9. Существует множество ограничений, которые можно использовать для создания уникального физического поведения

Ограничение **Rope** (Веревка) используется для моделирования веревочного соединения между двумя объектами. **Rope** (Веревка) имеет те же свойства, что и **Rod** (Стержень), но дополнительно у **Rope** (Веревка) есть свойство упругости **Restitution** (Восстановление), которое определяет, насколько быстро веревка восстанавливает свою первоначальную длину после приложения к ней силы. По сути, это стержень, но без жесткости, как и настоящая веревка.

Инструмент **Welding** (Сварка) позволяет сделать так, чтобы детали оставались соединенными, но не закрепленными в пространстве. Существует несколько типов экземпляров, которые можно использовать для «сварки», но соединения ломаются, если вы двигаете объекты по отдельности с помощью инструментов построения Studio. Также соединение создает `CFrame` из объектов. Хотя некоторые из этих эффектов можно свести на нет с помощью кода, иногда проще использовать экземпляр ограничения **Weld** (Шов). Используя это ограничение, вы можете «сварить» два объекта вместе, сохраняя их исходное положение и свободу перемещения деталей после их сварки без разрыва соединения между ними. Включите опцию **Show Welds** (Показать швы) на вкладке **Model** (Модель), чтобы просматривать соединения, в Studio, как показано на рис. 4.10.

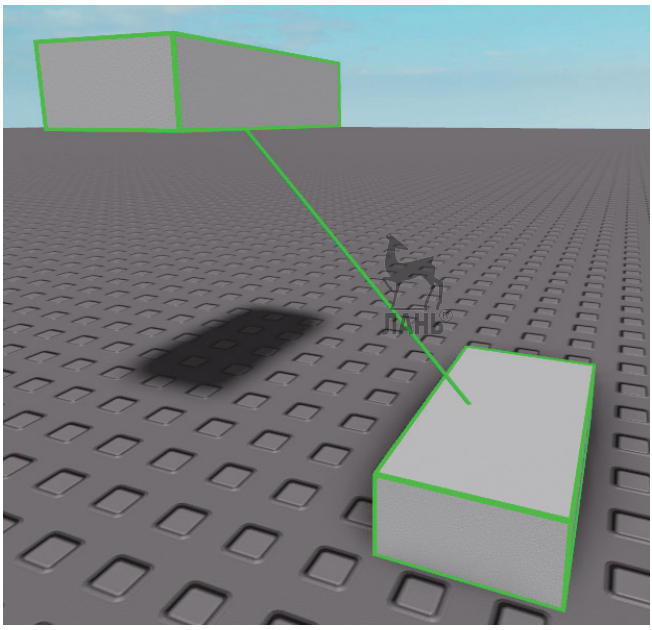


Рис. 4.10. Вы можете просматривать соединения в Studio, переключая опции **Show Welds** и **Constraint Details**

Чтобы узнать больше о том, какие существуют другие типы ограничений и как их использовать, ознакомьтесь со следующей статьей на сайте разработчика: **https://developer.roblox.com/en-us/articles/Constraints**.

Перемещение тел

Экземпляры базового класса `BodyMover` – это удобный способ реализовать приложение сил к `BasePart` напрямую или через заданную работу. `BodyMover` работают только в том случае, если они вложены в экземпляр `BasePart`. Это означает, что если вы хотите, чтобы вся модель двигалась, все ее части должны быть сварены, а `BodyMover` должен находиться в центральной части.

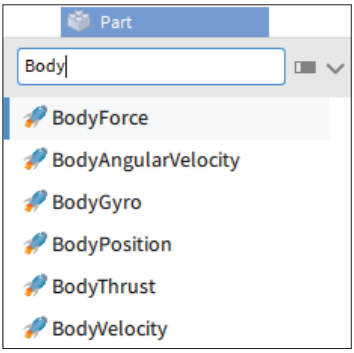


Рис. 4.11. Здесь создано шесть экземпляров `BodyMover`, каждый со своим поведением

`BodyForce` – это экземпляр `BodyMover`, который применяет силу к экземпляру, к которому он относится. Эта сила используется в координатах мирового пространства, то есть не зависит от ориентации самого объекта. Основным свойством данного экземпляра является `Force`, значение типа `Vector3`, которое применяется к центру масс детали. Применение прямой силы имеет много практических примеров, а в следующем коде мы научим объект зависать в воздухе.

Чтобы сделать это, нам нужно базовое понимание физики и метод `GetMass()` экземпляра `BasePart`. Сила, которая нужна для подвешивания детали в воздухе, – это масса тела, умноженная на силу тяжести и приложенная вверх по оси Y:

```
local part = workspace.Part
local bodyForce = Instance.new("BodyForce")
bodyForce.Parent = part
local requiredForce = part:GetMass() * workspace.Gravity
bodyForce.Force = Vector3.new(0,requiredForce,0)
```

`BodyVelocity` – это еще один экземпляр `BodyMover`, который аналогичным образом действует на мировое пространство. Этот экземпляр особенно полезен, когда вы хотите задать постоянную скорость объекта без использования циклов. Два основных свойства этого `BodyMover` – это `MaxForce` и `Velocity`. `MaxForce` – это свойство типа `Vector3`, задающее силу, прикладываемую к детали при заданной скорости. Это означает, что если масса объекта больше `MaxForce` движителя, умноженного на силу тяжести данной оси, объект не сдвинется с места. `Velocity` – это тоже свойство типа `Vector3`, задающее желаемую скорость.

В следующем примере объект движется вперед относительно своей передней грани. Обратите внимание, что мы устанавливаем каждый компонент свойства `MaxForce` равным `math.huge`, что по сути есть бесконечно большое значение. Это сделано для того, чтобы движитель мог воздействовать на деталь независимо от ее массы. По мере роста вашего опыта вы сможете менять значение этого свойства, чтобы ограничить нежелательное поведение, но для приведенных здесь примеров этого не требуется. Умножая желаемую скорость на направленный вперед вектор объекта или компонент `LookVector` его `CFrame`, мы можем задать движение детали с заданной скоростью в мировых координатах. Рекомендуется запускать этот код в поле **Command Bar** (Командная строка) или даже в серверном сценарии, а тест выполнить в режиме **Run** (Выполнить):

```
local Part1 = workspace.Part1
local bodyVelocity = Instance.new("BodyVelocity")
local huge = math.huge
bodyVelocity.MaxForce = Vector3.new(huge,huge,huge)
```

```
bodyVelocity.Parent = Part1

local projectileVel = 5
local moverVel = Part1.CFrame.LookVector * projectileVel
bodyVelocity.Velocity = moverVel
```

Вы можете установить `CFrame` объекта так, чтобы он смотрел вперед, перемещать его приведенным кодом, но можно также задавать направление скорости, не меняя объект напрямую. С помощью ранее настроенного `BodyMover` мы можем вычислить новый направленный вектор, исходная точка которого находится в той части, с которой в настоящее время связан движитель, а конечная – на другом объекте. В следующем коде мы вычислим новый вектор, выполнив простой расчет. Вычитая первый вектор положения из второго, мы получаем новый вектор направления от первого положения ко второму. Обратите внимание, что целевой объект должен быть закреплен. Также обратите внимание на использование `Unit` в `LookVector`. Это делается с целью нормализации вектора, чтобы он имел длину 1, что важно. Компоненты вектора `CFrame` всегда имеют длину 1. Обратите внимание, насколько по-другому ведет себя `BodyMover` без использования нормализованного вектора. Замените все строки после `projectiveVel` в предыдущем примере следующим кодом:

```
local Part2 = workspace.Part2
local lookVector = (Part2.Position - Part1.Position).Unit
--p2 - p1 = вектор направления в p1, смотрящий на p2
local moverVel = lookVector * projectileVel
bodyVelocity.Velocity = moverVel
```

Экземпляр `BodyPosition` работает в мировом пространстве и функционирует несколько не так, как рассмотренные объекты `BodyMover`. `BodyPosition` имеет четыре первичных свойства: `Position`, `Maxforce`, `P` и `D`. Свойства `P` и `D` используются для определения того, насколько агрессивно достигается цель и сколько демпфирования применяется при попытке достичь этой цели. Вы можете оставить эти свойства со значениями по умолчанию, если не знаете, что именно делать. Свойство `Maxforce` ведет себя так же, как у `BodyVelocity`. `Position` – это свойство типа Vector3, которое показывает, куда `BodyPosition` пытается переместить объект, и автоматически используется для расчета необходимых сил. В следующем `Maxforce` снова был равен `math.huge`, а `Position` равно положению в рабочем пространстве, 15 по оси Y. Вы должны увидеть плавный переход части между исходным положением и новой целью:

```
local part = workspace.Part
local bodyPosition = Instance.new("BodyPosition")
local huge = math.huge
```

```
bodyPosition.MaxForce = Vector3.new(huge,huge,huge)
bodyPosition.Parent = part
bodyPosition.Position = Vector3.new(0,15,0)
```

Экземпляр BodyGyro позволяет перевести объект в положение, заданное CFrame. Здесь нам интересны свойства MaxTorque и CFrame. В следующем примере объект будет поворачиваться на 45° вокруг оси Z. Обратите внимание, что MaxTorque тоже равен max.huge, чтобы можно было управлять объектом любой массы:

```
local bodyGyro = Instance.new("BodyGyro")
local huge = math.huge
bodyGyro.MaxTorque = Vector3.new(huge,huge,huge)
bodyGyro.Parent = part
bodyGyro.CFrame = part.CFrame * CFrame.fromOrientation(0,0,
math.pi/4)
```

BodyThrust – это BodyMover, который к объекту применяет силу относительно его направления. Тяга прикладывает силу постоянно, как и экземпляр BodyForce, что полезно для моделирования ускорения или, если добавить математики, постоянной скорости, если известна сила противодействия. Этот экземпляр отличается от BodyForce тем, что работает в пространстве объекта, что удобно для моделирования двигателей, хотя для этого можно использовать экземпляр объекта RocketPropulsion. Ключевые свойства BodyThrust – это Force и Location. Force ведет себя так же, как у BodyForce. Location определяет, в какой точке объекта прикладывается сила, а это означает, что тягу не обязательно прикладывать к центру масс родительского объекта BodyMover. В следующем примере к телу прикладывается сила BodyThrust, в два раза превышающая силу тяжести, в результате чего деталь быстро взлетает вверх:

```
local bodyThrust = Instance.new("BodyThrust")
bodyThrust.Parent = part
local requiredForce = part:GetMass() * workspace.Gravity * 2
bodyThrust.Force = Vector3.new(0,requiredForce,0)
--Сила будет приложена вверх относительно ориентации детали
```

Теперь, когда вы знаете, как взаимодействовать с физикой в Roblox за пределами чисто математической среды, вы можете создавать в игре еще более сложные игровые механики и визуальные эффекты, чтобы обеспечить лучший опыт для вашей аудитории.

Добавление второстепенных аспектов игры

Чтобы игра стала полноценнее, вы должны уметь работать со второстепенными аспектами проекта. Разработчики нередко упускают из внимания

детали, такие как звук, освещение и другие эффекты, и работа с ними позволит вам стать более опытным и, следовательно, более ценным членом команды.

Звук

В разработке игр звук – это очень важный аспект, позволяющий игроку погрузиться в происходящее на сцене или в игре. Хороший звуковой дизайн – это тема для отдельной книги, но вам нужно знать, как работать со звуками в Roblox.

Существует несколько способов найти нужные звуки – поискать аудиофайлы на панели **Toolbox** (Панель инструментов), поискать там же модели, которые могут содержать нужный звук, или загрузить свои собственные аудиофайлы в Roblox. Первый вариант реализуется с помощью **Toolbox** (Панель инструментов) или сайта Roblox. И там, и там вы можете найти подходящие звуки с помощью поиска. При поиске музыки или звуков таким способом может быть полезно знать, что большинство файлов загружены компанией Roblox. У Roblox есть лицензионные соглашения с музыкальными компаниями APM и Monstercat, в рамках которых разработчики могут пользоваться аудиофайлами без дополнительных затрат и без указания авторства. Если вы ищете именно звуковые эффекты, то можете поискать более короткие файлы с помощью фильтра, как показано на рис. 4.12.

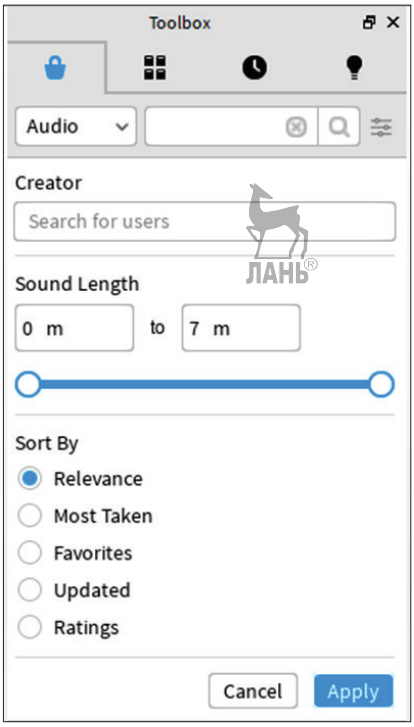


Рис. 4.12. Фильтры облегчают вам поиск нужного файла

Для реализации второго варианта нужно выполнить поиск моделей, которые могут содержать нужные вам звуки. Иногда это более полезно, чем поиск аудио напрямую, потому что названия аудиофайлов часто не отражают их содержимого. Наконец, вы можете загружать свои собственные аудиоресурсы в Roblox за небольшую плату, которая зависит от длины трека.

> **Важное замечание**
>
> Важно понимать, что аудиофайлы, которые вы загружаете, по-прежнему подпадают под действие законов об авторском праве. Например, загрузка вашей любимой песни определенной группы повлечет за собой проверку вашей учетной записи, если Roblox получит запрос DMCA от правообладателя. Однако у Roblox есть несколько систем, предназначенных для вашей защиты. При загрузке аудио бот проверяет материалы, защищенные авторским правом, и блокирует загрузку такого контента. Если вам все же удастся загрузить файл и вы получите запрос DMCA, Roblox применяет систему нарушения авторских прав с тремя предупреждениями. После третьего несоблюдения законов об авторском праве ваша учетная запись может быть временно заморожена или удалена. Соответственно, загружая аудиофайл, вы должны иметь на это право или лицензию.

В конце 2016 года Roblox выпустили долгожданное обновление аудиосистемы, позволяющее управлять звуком в реальном времени. Это дает возможность менять звуковое поведение без необходимости повторной загрузки файлов и изменений кода. Использование эффектов может сделать ваше окружение или игровую систему более захватывающими. Например, вы можете использовать эхо в тоннелях или повышать тон звука при ускорении персонажа (рис. 4.13).

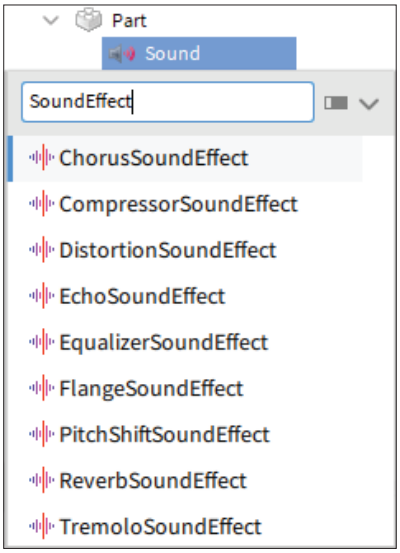


Рис. 4.13. В Roblox есть звуковые модификаторы для изменения свойств звука в режиме реального времени

Подробнее о работе звуковых модификаторов вы можете прочитать в блоге Roblox. В этом посте развернуто описывается работа каждого экземпляра и их свойства: **https://blog.roblox.com/2016/11/1/.**

Освещение

Свет ничуть не менее звука важен для создания красивой игровой среды. Если вы работаете в команде, то этим может заниматься отдельный человек, но знание того, как настроить освещение так, чтобы оно выглядело в игре красиво, – это полезный навык независимо от вашей роли в команде.

Первый способ, которым вы можете настроить освещение для своей игры, – это через свойства сервиса **Lightning** (Освещение) в окне **Explorer** (Проводник). Некоторые свойства приведены на рис. 4.14.

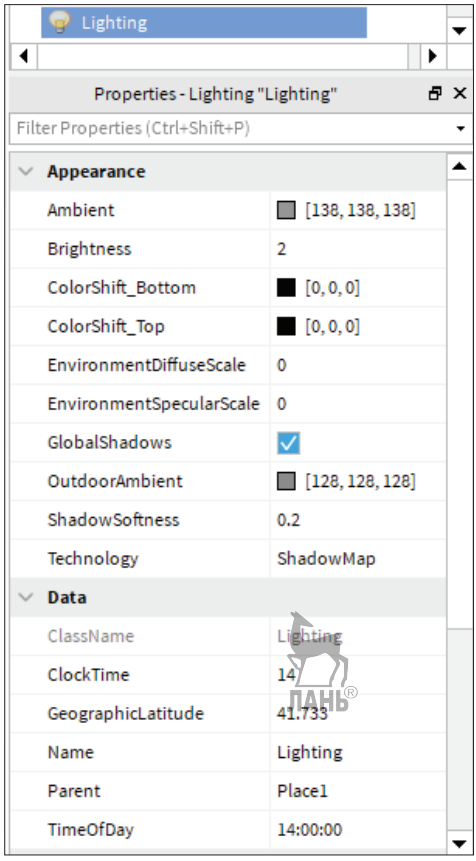


Рис. 4.14. Изменение свойств света наделяет игру особым шармом

Одним из наиболее часто используемых свойств **Lightning** (Освещение) является `ClockTime`. Это свойство принимает значения от 0 до 24 и соответствует часам суток. С помощью данного свойства можно задавать дневные и ночные циклы.

В разделе Technology свойств сервиса **Lightning** (Освещение) вы можете выбрать тип технологии освещения, используемый в игре. Современные технологии освещения называются Compatibility, Voxel, ShadowMap и Future. Каждая технология освещения используется для рендеринга освещения в игре. Рассмотрим подробнее:

- опция Compatibility делает свет похожим на стиль Legacy, который был удален в пользу технологии Future is Bright;
- опция Voxel делает свет таким же, как Compatibility, но для расчета тени используется сетка 4×4×4 (воксель). Эта технология более производительна, чем некоторые другие технологии освещения, поскольку тени получаются менее четкими;
- опция ShadowMap визуализирует тени с высоким разрешением. Однако она более эффективна, чем свойство Future освещения, потому что в большинстве случаев не дает точных отражений;
- Future – это новейшая технология освещения, которая поддерживает усовершенствованные тени с высокой детализацией, а также более реалистичные отражения на поверхностях из различных источников. Единственным недостатком этой технологии является то, что рендеринг этих дополнительных визуальных деталей требует больше ресурсов, снижает частоту кадров и производительность на более слабых устройствах.

Чтобы улучшить гибкость и разнообразие эстетики, в играх Roblox добавили ряд эффектов постобработки и модификаторов освещения. Вы можете спроектировать освещение в своем мире именно так, как вы хотите, чтобы лучше создать атмосферу любыми способами: от солнечных лучей и глубины резкости до общей цветокоррекции (рис. 4.15).

Посетив веб-сайт разработчика, вы можете сами ознакомиться с некоторыми реализациями вышеупомянутых световых эффектов в подробной среде. Там же вы найдете советы и рекомендации относительно вариантов использования: **https://developer.roblox.com/en-us/articles/post-processing-effects.**

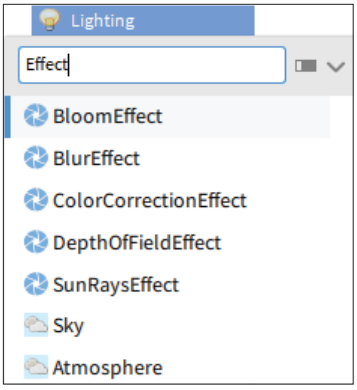


Рис. 4.15. В Roblox есть световые эффекты, позволяющие легко изменить атмосферу игры

Другие эффекты

Помимо освещения и звука, существуют и более прямые визуальные эффекты, с помощью которых можно сделать игру привлекательнее и интерактивнее. Для создания настраиваемых эффектов, которые могут оживить вашу игру, вы можете добавить ParticleEmitters (Генераторы частиц), Beams (Лучи) и Trails (Следы).

Частицы – это генерация 2D-изображений для создания эффектов дыма, огня и даже конфетти. У генераторов есть множество свойств, которые можно использовать для изменения внешнего вида эффектов, – размер частиц, текстура, прозрачность, скорость и частота появления частиц. Генераторы также позволяют менять эти значения в течение жизни частицы. При установке значений некоторых из этих свойств из сценария используются определенные типы данных, с которыми вы еще незнакомы, в том числе `NumberRange` и `NumberSequence`. В качестве примера применения генераторов частиц ознакомьтесь со следующей статьей на веб-сайте разработчика: **https://developer.roblox.com/en-us/articles/Particle-Emitters.**

`Beams` позволяют создавать постоянный визуальный эффект между двумя точками. Кроме того, вы можете добавить траектории эффекта кривизну, создавая электрические дуги или радуги. В свойствах этого экземпляра вы можете изменить цвет, прозрачность, текстуру, ширину и даже скорость перемещения текстуры. На следующем изображении показана дугообразная пунктирная линия, которую можно использовать, например, для прицеливания снарядом. Обратите внимание, что дуга идеально расположена между двумя точками (рис. 4.16).

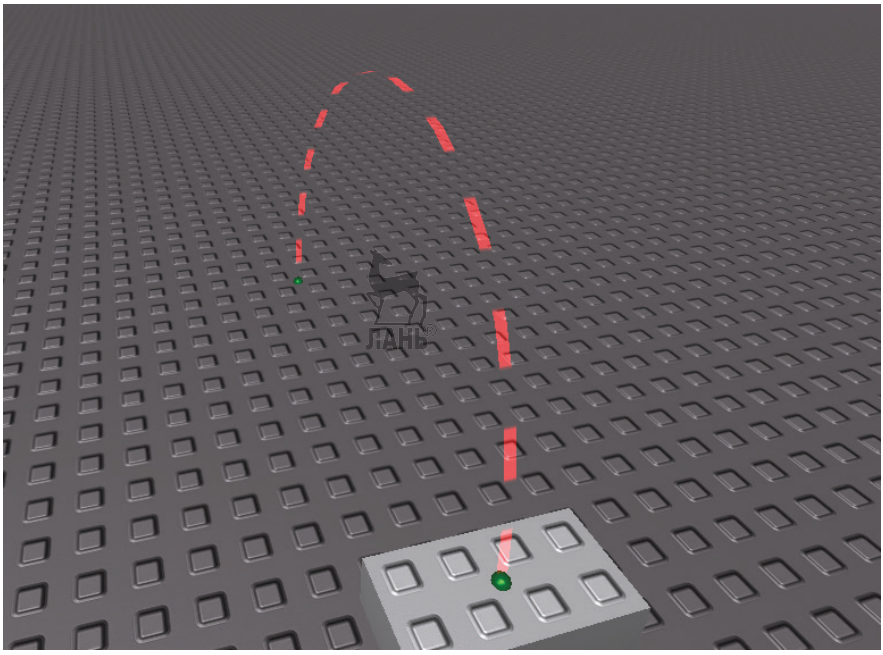


Рис. 4.16. Лучи можно использовать для создания множества эффектов, в том числе с кривизной

`Trails` похожи на `Beams`, так как они тоже создают постоянно видимый эффект между двумя точками, но `Trails` создают эффект, который тянется за объектом и угасает по мере движения. Ширина эффекта зависит от расстояния между точками.

В свойствах `Trails` вы можете изменить цвет, прозрачность, текстуру и время, необходимое для исчезновения эффекта. У этого эффекта есть много применений, в том числе создание трассеров для снарядов, следов шин или просто обозначение движения персонажа. Во многих популярных играх этот эффект делают коллекционным или покупаемым, а дизайн эффекта уникален или является показателем привилегий (рис. 4.17).

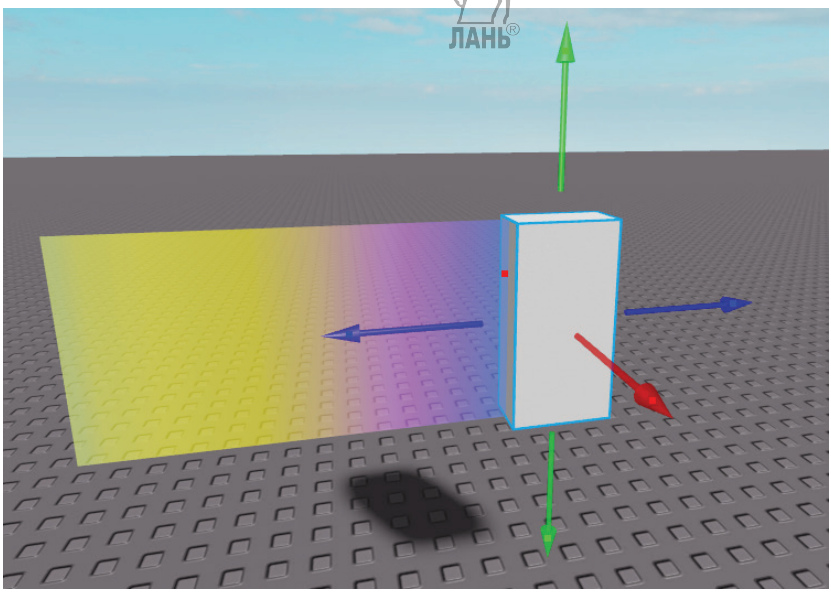


Рис. 4.17. Вы можете использовать `Trails`, чтобы обозначить движение игроков в игровой среде

Для программистов работа с этими аспектами вторична, но они по-прежнему являются важной частью проекта в целом, и умение работать с ними является бесценным навыком независимо от того, работаете вы в одиночку или в команде.

Резюме

В этой главе мы освоили оставшийся материал, о котором вам нужно было знать, чтобы перейти к созданию полноценных игр. Во всей вашей будущей работе вы будете использовать ваши знания о сценариях, фильтрации, сервисах, физике и второстепенных аспектах игры.

Теперь, когда вы умеете программировать в Lua и знаете функциональность различных сервисов и экземпляров, необходимых для создания систем, в следующей главе мы создадим свою первую полноценную игру. Вы увидите полное руководство по созданию обби-игры. Это популярный в Roblox жанр, в который вы можете вдохнуть что-то свое и привлечь новую аудиторию.

Глава 5

Пишем обби-игру

Теперь, когда у вас есть все знания, необходимые для создания полноценной игры, проверим ваши навыки, создав **игру в жанре «полоса препятствий»**, который чаще называется **обби**. В этой главе вы узнаете, как объединить все изученное в предыдущих главах, а именно такое, как движущиеся части, награды, эффекты и управление игроками. К концу этой главы мы объединим все наши знания, протестируем и опубликуем полнофункциональную игру.

Здесь мы рассмотрим следующие основные темы:

- управление данными игрока;
- создание препятствий;
- создание наград;
- добавление магазинов и покупок;
- создание эффектов;
- тестирование и публикация.

Технические требования

В рамках этой главы вы будете работать исключительно в Roblox Studio. Как упоминалось ранее, наличие подключения к интернету улучшит вашу работу в Studio и позволит вам проводить независимые исследования по любым темам.

Весь код этой главы вы можете найти в репозитории данной книги на GitHub – **https://github.com/PacktPublishing/Coding-Roblox-Games-Made-Easy/tree/main/Chapter05.**

Настройка серверной части

Под серверной частью (бэкендом) системы в вычислительной технике подразумевается в первую очередь инфраструктура кода, с которой клиент не будет напрямую взаимодействовать никогда. В этом разделе мы сосредоточимся на управлении данными и игроками, которое осуществляется с сервера. Кроме того, как упоминалось в главе 3, сервер должен получить всю доступную ему информацию в целях безопасности.

В начале главы 4 вы познакомились с модулями и некоторыми из их возможных применений. Становится ясно, что при создании чего-то более масштабного, например полноценного проекта, модульность игровых систем выходит на передний план. Если бы вы использовали в одном диспетчере сценариев отдельные сценарии, а не модули, организация сценариев быстро превратилась бы в нечто совершенно неуправляемое. В этой главе созданные вами системы будут содержаться в модулях, что облегчает доступ и редактирование.

Помните, что код внутри модулей просто сохраняется, но не запускается, если только модуль не запрашивается явно. Приведенный код должен быть добавлен в экземпляр `Script` в `ServerScriptService`. Сценарию не нужно давать никаких особенных имен, но обычно выбирают имя вроде `ServerHandler`. Все создаваемые вами модули должны быть вложены в этот сценарий, поскольку клиентам легко получать к нему доступ. Вы уже должны были достаточно набить руку в программировании в среде Roblox, чтобы изменить путь к индексируемому объекту, если хотите, чтобы он находился в другом месте. Просто помните, что для хранения модулей лучше всего использовать `ServerStorage` и `ServerScriptService`, поскольку клиент-злоумышленник попасть туда не может.

Код в сценарии `ServerHandler` будет простым – каждый модуль работает как изолированная система, а сценарию `ServerHandler` эти модули нужны для того, чтобы их запускать. По-прежнему лучше использовать модули, а не отдельные сценарии, т. к. сценарий `ServerHandler`, вероятно, будет использоваться для более общего управления проектом, а модули содержат только код, относящийся к отдельной системе. В приведенном ниже коде используется цикл `for`, который перебирает модули, вложенные в сценарий. Помните, что ключевое слово **script** указывает на сам экземпляр `Script` на панели **Explorer** (Проводник). Далее мы используем функции `create()` и `resume()` из библиотеки `coroutine` для создания нового потока. Эта функция оборачивает функцию `require()`, которая принимает текущий модуль в качестве аргумента.

В этом примере мы задаем поведение, идентичное поведению функции `spawn()`, так как сами функции `spawn()` могут выдавать странное и нежелательное поведение, и поэтому мы должны использовать сопрограммы. Преимущество создания нового потока здесь заключается в том, что если некоторая модель выдаст ошибку при загрузке, другие модули загрузятся нормально:

```
for _, module in pairs(script:GetChildren()) do
    local loadMod = coroutine.create(function()
        require(module)
    end)

    coroutine.resume(loadMod)
end
```

Теперь, когда мы обсудили структуру бэкенда, пора разобраться, как управлять данными игрока.

Управление данными игрока

Одна из важнейших частей бэкенда игры – это управление данными игрока. Игре не только нужно будет отслеживать количество собранных игроком денег, которые можно будет потратить в магазине, но, кроме того, мы будем отслеживать прогресс игрока, чтобы по возвращении в игру он оказывался там же, где закончил в прошлый раз. Для начала мы создадим новый модуль с именем `Data`, в котором будет лежать код. Как мы сказали ранее, он должен быть вложен в `ServerHandler`, как показано на рис. 5.1.

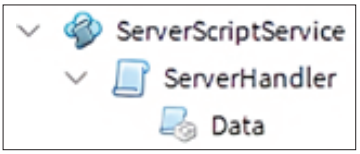


Рис. 5.1. Модуль `Data` связан непосредственно со сценарием `ServerHandler`

Пора двигаться дальше и узнать, как создать систему хранилища данных.

Создание системы хранения данных

Чтобы создать для игры систему хранения данных, нам нужно сначала понять, какие сервисы потребуются для ее настройки. Приведенный ниже код содержится в модуле `Data`. Вам нужно будет добавить в него функции, которые будут приведены в этом разделе, между строками `dataMod` и `return return dataMod`. Обратите внимание, что добавление любого кода после `return dataMod` приведет к ошибке.

В некоторых функциях нам нужен будет сервис `Players`. Сервис `DataStoreService` используется для сохранения данных на серверах Roblox с использованием ключа. Этот сервис и не отображается на панели **Explorer** (Проводник).

Переменная `store` использует метод `GetDataStore()` сервиса `DataStoreService`, чтобы создать новый экземпляр `GlobalDataStore`. Данные, хранящиеся в `GlobalDataStore`, сохраняются с помощью ключа, которым он инициализирован; в нашем случае это ключ `DataStoreV1`. Это означает, что вы можете создавать пустые хранилища данных, вводя ключ, который не использовался в игре раньше.

Переменная `sessionData` содержит словарь с данными игроков, которые в настоящий момент находятся на сервере. `UserId` в нем используются в качестве ключей, а таблицы данных в качестве значений. `UserId` – это номер, который значится в URL-адресе профиля игрока. Он уникален для каждого пользователя и не изменяется при смене имени, поэтому его можно использовать в качестве ключа для сохранения данных:

```
local playerService = game:GetService("Players")
local dataService = game:GetService("DataStoreService")
```

```
local store = dataService:GetDataStore("DataStoreV1")

local sessionData = {}
local dataMod = {}

return dataMod
```

Первой функцией в модуле `Data` будет функция `recursiveCopy()`, которую мы рассматривали в главе 3. Там же, при изучении табличного типа данных, мы говорили, что в таблицах используют ссылки, а не копии переменных. Функция `recursiveCopy()` дает вам некоторую безопасность при работе с таблицами данных, в случае если любой код, который вы добавляете в другом месте, ожидает, что таблица будет уникальной:

```
dataMod.recursiveCopy = function(dataTable)
    local tableCopy = {}

    for index, value in pairs(dataTable) do
        if type(value) == "table" then
            value = dataMod.recursiveCopy(value)
        end

        tableCopy[index] = value
    end

    return tableCopy
end
```


Важное примечание

Обратите внимание, что вспомогательные функции можно хранить и внутри модуля, но функции вроде `recursiveCopy()` могут оказаться полезны и в других сценариях. Следовательно, лучше бы добавить ее в таблицу модулей, чтобы любой сценарий мог к ней обратиться.

В целях наведения порядка вы можете попробовать размещать вспомогательные функции в верхней части вашего `ModuleScript`, а локальные функции упорядочивать в порядке от самых используемых к менее используемым.

В следующем разделе мы добавим функции создания и загрузки данных игроков, которые присоединяются к игре.

Создание и загрузка данных сеанса

Когда игрок присоединяется к игре, ему необходимо добавить свои данные в `sessionData`. Для этого мы добавим в модуль новую функцию под

названием `setupData()`, принимающую в качестве аргумента экземпляр **игрока**. Обратите внимание, что для получения данных мы используем метод `GetAsync()` из экземпляра `GlobalDataStore`, который хранится в переменной `store`. Значение переменной `key` — это `UserId` игрока. Если у игрока есть сохраненные данные, они будут лежать в переменной `data`, будь то таблица или отдельное значение. Если с игроком не связано никаких данных, переменная будет содержать `nil`. В следующей функции мы обработаем оба этих случая.

В первом случае мы копируем таблицу статистики под названием `defaultData`, а затем используем цикл `for` для изменения значений в ней в соответствии с данными игрока. Таблицу лучше именно скопировать, а не менять значения в ней напрямую, так как можно будет позже добавлять новые значения в `defaultData`, и они сразу попадут в данные игрока при следующей загрузке проигрывателя без необходимости дополнительного кодирования.

Во втором случае ничего делать не нужно, так как пустая таблица у нас задана сразу. При желании вы можете сделать дополнительный модуль, в котором бы таблица `defaultData` хранилась, не занимая места в сценарии.

Вы можете заметить, что у нас вызывается функция `set()`, которую мы не определяли. Оставьте эту строку, а функцию мы определим позже. Приведенный ниже код добавьте в модуль без изменений, если вы не уверены в работе с этой системой:

```
local defaultData = {
    Coins = 0;
    Stage = 1;
}

dataMod.load = function(player)
    local key = player.UserId
    local data = store:GetAsync(key)

    return data
end

dataMod.setupData = function(player)
    local key = player.UserId
    local data = dataMod.load(player)

    sessionData[key] = dataMod.recursiveCopy(defaultData)

    if data then
        for index, value in pairs(data) do
            dataMod.set()
```

```
            print(index, value)
                dataMod.set(player, index, value)
            end

            print(player.Name.. " - данные игрока загружены!")
        else
            print(player.Name.. " - наш новый игрок!")
        end
    end
```

Теперь вам нужна возможность вызывать эти функции в нужный момент. Добавьте в модуль функцию события `PlayerEvent`, которая вызывает `setupData()`, передавая ей объект игрока. Мы также реализуем систему `leaderstats`, о которой говорили в главе 4, чтобы отображать текущую статистику игрока. Значения в папке `leaderstats` изначально соответствуют данным в `defaultData`, но в момент загрузки данных вызывается метод `set()`, который обновляет значения в папке `leaderstats`, что избавляет вас от лишней работы:

```
playerService.PlayerAdded:Connect(function(player)
    local folder = Instance.new("Folder")
    folder.Name = "leaderstats"
    folder.Parent = player

    local coins = Instance.new("IntValue")
    coins.Name = "Coins"
    coins.Parent = folder
    coins.Value = defaultData.Coins

    local stage = Instance.new("IntValue")
    stage.Name = "Stage"
    stage.Parent = folder
    stage.Value = defaultData.Stage

    dataMod.setupData(player)
end)
```

Данные игрока создаются и добавляются в `sessionData`, теперь нам нужно будет добавить в модуль функции для управления им.

Управление данными сеанса

Когда игрок присоединился, а его данные заданы, вы должны иметь возможность манипулировать данными в `sessionData` из других сценариев. Добавим все в тот же модуль `Data` некоторые функции, которые позволят задавать, увеличивать значения и извлекать данные из таблицы. Эти три

функции очень просты: у функций `set()` и `increment()` три аргумента – это игрок, которым вы хотите управлять, строка статистики, которую вы хотите изменить, и значение, которое используется функцией:

- функция `set()` устанавливает заданное значение для заданной статистики игрока в `sessionData`;
- функция `increment()` прибавляет к выбранной статистике переданное значение;
- функция `get()` принимает экземпляр игрока и название статистики, которую вы хотите прочитать.

Обратите внимание, что функции `set()` и `increment()` также обновляют соответствующую статистику в системе `leaderstats`:

```
dataMod.set = function(player, stat, value)
    local key = player.UserId
    sessionData[key][stat] = value
    player.leaderstats[stat].Value = value
end

dataMod.increment = function(player, stat, value)
    local key = player.UserId
    sessionData[key][stat] = dataMod.get(player, stat) + value
    player.leaderstats[stat].Value = dataMod.get(player, stat)
end

dataMod.get = function(player, stat)
    local key = player.UserId
    return sessionData[key][stat]
end
```

Теперь данные игрока можно легко изменить, в следующем разделе показано, как сохранить эти данные, когда игрок покидает игру.

Сохранение данных игрока

Сохранить данные игрока ничуть не сложнее, чем загрузить. Функция для сохранения данных будет просто называться `save()` и принимать экземпляр игрока в качестве аргумента.

Как и прежде, в качестве ключа таблицы `sessionData` используется `UserId`. Кроме того, ключ применяется в методе `SetAsync()` экземпляра `GlobalDataStore` в переменной `store` для сохранения копии этой таблицы данных. `SetAsync()` принимает ключ в качестве первого аргумента и данные, которые необходимо сохранить, в качестве второго:

```
dataMod.save = function(player)
    local key = player.UserId
```

```
    local data = dataMod.recursiveCopy(sessionData[key])

    store:SetAsync(key, data)
    print(player.Name.. " - данные игрока сохранены!")
end
```

Когда игрок покидает игру, его таблица данных должна быть удалена из `sessionData`. Если этого не делать, игра будет занимать все больше памяти с каждым игроком, чьи данные загружаются в таблицу, но не удаляются из нее. Добавим функцию `removeSessionData()`, которая принимает в качестве аргумента игрока, чьи данные нужно удалить. Для поиска нужной строки в таблице по-прежнему используем `UserId`:

```
dataMod.removeSessionData = function(player)
    local key = player.UserId
    sessionData[key] = nil
end
```

Аналогично тому, как мы использовали событие `PlayerAdded`, событие для вызова функции `setupData`, мы будем применять `PlayerRemoving`, чтобы определить момент для удаления данных из таблицы. Обратите внимание, что функция события вызывает и метод `save()`, и метод `removeSessionData()` модуля, передав экземпляр `player` как параметр. Помните, что вызов функций из модулей перехватывает управление, поэтому можно не беспокоиться о том, что таблица данных игрока будет удалена до сохранения:

```
playerService.PlayerRemoving:Connect(function(player)
    dataMod.save(player)
    dataMod.removeSessionData(player)
end)
```

В следующем разделе мы поговорим о создании вашей собственной системы хранилища данных и рассмотрим некоторые реализации, с помощью которых мы зададим хранилищу желаемое поведение, добавим дополнительные функции и получим дополнительную второстепенную информацию о природе хранилищ данных.

Обработка троттлинга и граничных случаев

Сервис `DatastoreService` известен своей капризностью и умением случайно терять данные даже в самых совершенных системах. Чтобы сделать вашу систему как можно более совершенной, вы можете реализовать определенные методы, позволяющие гарантировать сохранение данных игрока, даже если это удастся не с первой попытки.

Одна из наиболее распространенных причин потери данных – это троттлинг. Троттлинг происходит, когда ваш запрос на установку или получение данных из Roblox отклоняется сервером. Чтобы предотвратить повреждение данных игрока, надо ввести функцию вроде `pcall()` (сокращение от *protected call*). Эта функция оборачивает выполняемый код и сообщает, успешно ли выполнился код и какая ошибка возникла (если возникла). При загрузке и сохранении данных игрока мы можем применять методы `SetAsync()` и `GetAsync()` функции `pcall()` и просто повторить попытку, если предыдущая попытка сохранить или загрузить данные не удалась.

В приведенном ниже коде мы переопределяем функции `save()` и `load()` модуля `Data`, введя реализацию функции `pcall()`. Обратите внимание, что в обеих функциях мы проверяем, содержит ли переменная `success` значение `true`, и если нет, вызываем функцию снова. Этот цикл будет продолжаться до тех пор, пока выполнение не удастся. Для сохранения мы можем просто вызвать функцию, но для загрузки нам нужно реализовать некоторую рекурсивную логику, поскольку мы получаем некоторые данные. Как вы можете видеть, в случае длительных проблем мы продолжаем вызывать функцию `load()`, пока не будет достигнут успех, возвращая данные туда, откуда произошел первый вызов. Обязательно полностью замените обе имеющиеся функции в вашем модуле на этот код:

```
dataMod.save = function(player)
    local key = player.UserId
    local data = dataMod.recursiveCopy(sessionData[key])

    local success, err = pcall(function()
        store:SetAsync(key, data)
    end)

    if success then
        print(player.Name.. " - данные игрока сохранены!")
    else
        dataMod.save(player)
    end
end

dataMod.load = function(player)
    local key = player.UserId
    local data

    local success, err = pcall(function()
        data = store:GetAsync(key)
    end)

    if not success then
```

```
        data = dataMod.load(player)
    end

    return data
end
```

Важное примечание

С помощью отдельной функции мы также записываем количество повторных попыток и по достижении определенного предела отключаем игрока, сохраняя его данные.

Реализовав систему автосохранения, вы можете сохранять данные клиента через заданный период времени, что снижает потери данных, если вдруг сохранение не выполнится при выходе из игры. В следующей функции используется постоянный период времени, определенный в переменной `AUTOSAVE_INTERVAL`, которая должна быть помещена в верхнюю часть вашего модуля.

С помощью цикла `while` внешний цикл `for` перебирает таблицу `sessionData` и находит игрока по его `UserId` с помощью метода `GetPlayerByUserId()` сервиса `Players`. Как только игрок будет найден, вызывается функция `save()`. Важно, чтобы `AUTOSAVE_INTERVAL` не был слишком мал, так как очень частое сохранение увеличивает риск троттлинга:

```
local AUTOSAVE_INTERVAL = 120

local function autoSave()
    while wait(AUTOSAVE_INTERVAL) do
        print("Автосохранение данных всех игроков")
        for key, dataTable in pairs(sessionData) do
            local player =
                playerService:GetPlayerByUserId(key)
            dataMod.save(player)
        end
    end
end

spawn(autoSave) --инициализация цикла автосохранения
```

Еще один способ защитить данные игрока – это добавить в модуль `Data` функцию `BindToClose()`. Событие `Close` игры срабатывает, когда отключается экземпляр сервера, если все игроки вышли или игра была закрыта вручную разработчиком. Когда это происходит, событие `PlayerRemoving` сервиса

Players может опоздать. Приведенная ниже функция не только делает запрос на сохранение данных каждого игрока посредством функции dataMod.save(), но и задержит закрытие сервера на тридцать секунд, пока не будут выполнены запросы на сохранение:

```
game:BindToClose(function()
    for _, player in pairs(playerService:GetPlayers()) do
        dataMod.save(player)
        player:Kick("Отключение игры. Все данные сохранены.")
    end
end)
```

Когда система хранения готова, предлагаю вам добавить в таблицу defaultData любую статистику, какая вам нравится, и с помощью функций тестирования Studio убедиться, что система работает. Один из способов сделать это – изменить свою статистику в функции события PlayerAdded, после того как ваши данные загрузятся, затем выйти и снова присоединиться, чтобы убедиться, что отображается статистика, с которой вы вышли из сеанса тестирования, а не значения из defaultData.

В следующем разделе мы познакомимся с некоторыми физическими механиками, которые должны быть реализованы в обби-игре.

Управление столкновениями и персонажами игроков

Поскольку столкновения игроков могут помешать игрокам проходить препятствия в игре, введем группу столкновений, чтобы столкновения между игроками были невозможны. Для этого необходимо создать новый модуль с именем Physics и вложить его в сценарий ServerHandler, как и раньше. Нам нужны будут сервисы Players и PhysicsService, как показано в следующем коде:

```
local playerService = game:GetService("Players")
local physicsService = game:GetService("PhysicsService")
local physicsMod = {}

return physicsMod
```

В простом варианте игры нам не обязательно создавать какие-либо функции внутри модуля, но позже они могут понадобиться, если вы будете реализовывать свои собственные системы. Как и в главе 4, нам необходимо создать новую группу столкновений, задав ей уникальный строковый идентификатор с помощью метода CreateCollisionGroup() сервиса PhysicsService.

После этого с помощью метода CollisionGroupSetCollidable() сервиса PhysicsService сделаем так, чтобы BasePart внутри группы не сталкивались

друг с другом. Это позволяет нам написать функцию события `PlayerAdded`, которая имеет функцию события `CharacterAdded` для любого клиента, подключающегося к игре.

Внешний цикл `for` перебирает экземпляры `BasePart` персонажа, который передается в качестве аргумента функции события `CharacterAdded`. Если экземпляр `BasePart` найден, он добавляется в группу столкновений `Players` с помощью метода `SetPartCollisionGroup()` сервиса `PhysicsService`.

Игроки больше не смогут сталкивать друг друга с карты или вставать друг на друга, чтобы попасть в места, в которые не должны:

```
physicsService:CreateCollisionGroup("Players")
physicsService:CollisionGroupSetCollidable("Players", "Players", false)

playerService.PlayerAdded:Connect(function(player)
    player.CharacterAdded:Connect(function(char)
        for _, part in pairs(char:GetDescendants()) do
            if part:IsA("BasePart") then
                physicsService:SetPartCollisionGroup(part, "Players")
            end
        end
    end)
end)
```

Вы должны иметь в виду, что группы столкновений могут использоваться не только для реализации игрового поведения. Вы можете использовать их для создания особых препятствий и головоломок, например этапа, на котором игрок может пройти сквозь стену, а коробка или мяч не проходит. Такого рода головоломки встречаются в серии видеоигр «Портал», например.

В этом разделе мы рассмотрели, как создать полноценную систему хранения данных, которая легко бы редактировалась и обеспечивала бы максимальную безопасность данных игроков. С помощью сервиса `PhysicsService` мы сделали так, чтобы игроки не могли мешать друг другу при прохождении этапов в игре.

Создание этапов для обби-игры

Теперь мы добрались до самой важной части разработки игры: проектирования сцен! Создание и проектирование уровней – это вопрос вашей фантазии, но в этом разделе мы рассмотрим, как запрограммировать различные механизмы, которые можно внедрить в этапы, и тем самым сделать игру интересной и прибыльной.

Начнем с поведения объектов.

Создание поведения объекта

Здесь модульность системы тоже будет иметь решающее значение. Многие новички в программировании, которые не пользуются модулями, пытаются использовать отдельные сценарии для управления системами, состоящими из сотен частей. Такой подход почти сразу наказывает программиста, если нужно будет внести какие-либо изменения, так как код нужно будет менять сразу везде.

Благодаря модулям код, работающий с определенными частями, можно хранить в одном месте, что облегчает его редактирование. В новом модуле, который должен называться `PartFunctions`, потребуется определить лишь несколько служб или модулей, так как мы в основном работаем с методами из класса `BasePart`, как показано в следующем коде:

```
local playerService = game:GetService("Players")
local replicatedStorage = game:GetService("ReplicatedStorage")
local dataMod = require(script.Parent.Data)
local partFunctionsMod = {}

return partFunctionsMod
```

В приведенных ниже функциях мы будем использовать событие `Touched` экземпляра `BasePart`. Если вы помните, экземпляры `BasePart` запускают событие `Touched` при касании с любой другой `BasePart`.

Чтобы ограничить количество избыточного кода, мы создадим вспомогательную функцию с именем `playerFromHit()`, которая принимает экземпляр `BasePart` и проверяет, является ли она потомком персонажа игрока. Если да, то и игрок, и персонаж игрока возвращаются в формате кортежа, а в противном случае оба значения по умолчанию будут нулевыми.

Обратите внимание, что метод экземпляров `FindFirstAncestorOfClass()` будет рекурсивно искать экземпляр, соответствующий указанному типу класса, пока не найдет его или не достигнет экземпляра `DataModel`, который является вершиной **иерархии наследования Roblox**. Независимо от того, будет найден экземпляр `Model` или значение будет равно нулю, оно передается в метод `GetPlayerFromCharacter()` сервиса `Players`.

Этот метод может принимать значение `nil` или любой экземпляр в качестве входных данных, но будет возвращать `nil`, если экземпляр, переданный методу, не является моделью персонажа игрока, присутствующего в игре. Если игрок с переданной моделью найдется, то игрок возвращается функцией `playerFromHit()`, что и сделано в следующем коде:

```
partFunctionsMod.playerFromHit = function(hit)
    local char = hit:FindFirstAncestorOfClass("Model")
    local player =
```

```
        playerService:GetPlayerFromCharacter(char)

    return player, char
end
```

Как упоминалось ранее, эта вспомогательная функция может быть сохранена как локальная функция, но если хранить ее внутри модуля, другим сценариям будет удобно обращаться к ней для создания аналогичного поведения. Первая функция, которую мы добавим в модуль `PartFunctions`, называется `KillParts()`, которая будет уничтожать объекты. Она используется как препятствие, которого игрок должен избегать. При касании игрока к препятствию с таким событием свойство `Health` экземпляра `Humaniod` станет равным 0, то есть игрок погибнет. Экземпляр `Humaniod` есть у всех персонажей Roblox и используется для управления поведением моделей персонажей.

Обратите внимание, что событие `Touched` передает прикоснувшуюся часть тела, которую можно использовать в качестве параметра вызова вновь реализованного метода `playerFromHit()`, возвращающего как игрока, так и персонажа, если часть тела является потомком модели персонажа игрока.

Далее мы используем логику короткого замыкания, которую описывали в главе 3. Если переменная `player` равна нулю, условный оператор закончит работу, так как оператор `and` не выполнился. Упорядочивая наши состояния так, что сначала идет игрок, а затем параметр `Health`, мы избегаем использования нескольких условных операторов и других ошибок. Если параметр `Health` у `Humanoid` больше 0, он становится равен 0, убивая игрока, как показано в этом блоке кода:

```
partFunctionsMod.KillParts = function(part)
    part.Touched:Connect(function(hit)
        local player, char =
            partFunctionsMod.playerFromHit(hit)
        if player and char.Humanoid.Health > 0 then
            char.Humanoid.Health = 0
        end
    end)
end
```

Проверка уровня здоровья позволяет снизить число повторных выполнений некоторого кода, поскольку игрока не нужно убивать, если он уже мертв. Это небольшая оптимизация, которая может дать плоды, если в вашей игре много этапов и игроков.

Следующая функция называется `DamageParts()`, и она используется, когда объект должен лишь наносить урон игроку, а не убивать его мгновенно. Вы можете расположить по уровню побольше таких, чтобы добавить интриги

для игрока, который пытается сохранить как можно больше своего здоровья. Эта функция аналогична `KillParts()`, но вместо убийства игрока она вычитает количество урона из текущего здоровья игрока, при этом урон получается от `IntValue` объекта `Damage`.

Кроме того, так называемый `debounce`. `Debounce` – это значение, которое предотвращает запуск чего-либо до тех пор, пока оно не будет сброшено. В следующем примере мы устанавливаем `debounce` равным `true`, когда игрок получает урон, и введем функцию `delay()`, которая сбрасывает это значение через 0,1 секунды.

Функция `delay()` создается аналогично функции `spawn()`, но перед определением функции используется значение времени. Вы можете спросить: зачем нам функция `delay()`, если есть `wait()`, но дело в том, что `delay()` не задерживает выполнение другого кода, что делает ее подходящей для различных вариантов использования.

Поскольку значение `debounce` сбрасывается через 0,1 секунды, игрок может получить урон не более 10 раз в секунду. При создании собственных систем вы обнаружите, что введение этого значение важно, чтобы ограничить число поступающих событий. Но помните, что злоумышленник может изменить значение `debounce`, поэтому не используйте его в качестве меры безопасности, если общаетесь с сервером:

```
partFunctionsMod.DamageParts = function(part)
    local debounce = false
    local damage = part.Damage.Value

    part.Touched:Connect(function(hit)
        local player, char =
            partFunctionsMod.playerFromHit(hit)

        if player and not debounce then
            debounce = true
            local hum = char.Humanoid
            hum.Health = hum.Health - damage

            delay(0.1, function()
                debounce = false
            end)
        end
    end)
end
```

В данные игрока нужно включить статистику под названием `Stage`, которую можно использовать для сохранения этапа, на котором игрок остановился и где его нужно возрождать. Приведенная ниже функция `SpawnParts()`

будет использоваться с объектами, используемыми в качестве контрольных точек, расположенных в начале каждого этапа. Они должны содержать `IntValue` с именем `Stage` с возрастающими значениями.

Как и раньше, мы определяем касание игрока, а затем проверяем, имеет ли текущая контрольная точка значение на единицу больше, чем последняя пройденная игроком, чтобы убедиться, что он вернулся назад и не проскочил какой-то этап. Если это условие выполнено, статистика `Stage` обновляется новым значением:

```
partFunctionsMod.SpawnParts = function(part)
    local stage = part.Stage.Value

    part.Touched:Connect(function(hit)
        local player, char =
            partFunctionsMod.playerFromHit(hit)
        if player and dataMod.get(player, "Stage") ==
            stage - 1 then
            dataMod.set(player, "Stage", stage)
        end
    end)
end
```

Этот код работает так, как и ожидалось, но игрок не получает уведомлений о своем прогрессе, если не поглядывает в статистику. Этот вопрос будет рассмотрен в разделе «*Создание эффектов*» в данной главе, где мы будем реализовывать частицы и звуки, которые появляются, когда игрок доходит до новой точки возрождения.

Теперь, когда нужные функции созданы, обращу ваше внимание на то, что для их имен не использовался верблюжий стиль. Дело в том, что вы должны создавать экземпляры `Folder` в рабочем пространстве и назвать их так же, как и функции. Это делается для того, чтобы приведенный ниже код в модуле `PartFunctions` мог легко ссылаться на папки объектов и вызывать функции с тем же именем.

После добавления папок в таблицу используются вложенные циклы `for`, которые вызывают функции с тем же именем, что и папка, передавая объекты этим папкам. Вы должны поместить этот код в конец вашего модуля, так как функцию нужно определить, чтобы вызывать:

```
local partGroups = {
    workspace.KillParts;
    workspace.DamageParts;
    workspace.SpawnParts;
}

for _, group in pairs(partGroups) do
```

```
    for _, part in pairs(group:GetChildren()) do
        if part:IsA("BasePart") then
            partFunctionsMod[group.Name](part)
        end
    end
end
```

Теперь все объекты будут работать как положено без какого-либо дополнительного программирования, а нам пришла пора добавить функциональность к объектам в папке `SpawnParts`, которая в настоящее время используется для обновления статистики `Stage` игрока. Чтобы выполнить это, мы создадим новый модуль с именем `Initialize` под сценарием `ServerHandler`, из которого в момент появления игрока будут запускаться более общие задачи, которые не относятся к конкретной системе.

Мы видим, что текущий этап игрока хранится в его данных, но нам нужна еще одна функция, которая позволит найти физическую точку, соответствующую номеру этапа. Чтобы сделать это, мы реализуем функцию с именем `getStage()`, которая в качестве аргумента принимает номер этапа. Эта функция будет перебирать объекты в папке `SpawnParts` и возвращать тот, который соответствует аргументу `stageNum`. После этого `CFrame` в свойстве `PrimaryPart` персонажа становится равен `CFrame` точки вознаграждения со смещением, чтобы игрок не сливался с объектом (см. код):

```
local playerService = game:GetService("Players")
local dataMod = require(script.Parent.Data)
local spawnParts = workspace.SpawnParts
local initializeMod = {}

local function getStage(stageNum)
    for _, stagePart in pairs(spawnParts:GetChildren()) do
        if stagePart.Stage.Value == stageNum then
            return stagePart
        end
    end
end

playerService.PlayerAdded:Connect(function(player)
    player.CharacterAdded:Connect(function(char)
        local stageNum = dataMod.get(player, "Stage")
        local spawnPoint = getStage(stageNum)
        char:SetPrimaryPartCFrame(spawnPoint.CFrame *
            CFrame.new(0,3,0))
    end)
end)

return initializeMod
```

Теперь, когда основные специальные части для вашего обби были созданы, следующий шаг – поддерживать интерес игроков, находя несколько способов их вознаграждения.

Создание наград

Ваша игра обби может привлекать игроков сложностью, тематикой или уникальными головоломками, но чтобы еще больше вовлечь аудиторию в процесс, вы можете добавить награды за достижение контрольных точек или более рискованное прохождение. В данных игрока есть статистика под названием `Coins` – валюта, которую игроки могут обменивать на внутриигровые предметы.

Чтобы изменить статистику `Coins`, мы создадим новую функцию с именем `RewardParts()` в модуле `PartFunctions`. Эта функция должна находиться перед циклом, который применяет функцию к объекту. Как и раньше, вам нужно будет добавить папку с соответствующим именем в таблицу `partGroups`, чтобы объекты в этой папке обрабатывались.

Приведенная ниже функция сначала получает количество монет, которое выдается игроку, из `IntValue` объекта `Reward`. После этого мы напишем для монеты уникальный код, чтобы отслеживать, какие монеты собрал игрок, и не давать ему получать одну и ту же монету многократно.

В функции события `Touched` мы получаем экземпляр игрока, и функция проверяет, есть ли папка с кодами монет под названием `CoinTags` в данных игрока. Если такой папки нет, то она создается. Когда папка создана, мы проверяем, есть ли в папке монетка с определенным кодом, тем самым проверяя, собиралась ли она. Если эта проверка пройдена, сумма вознаграждения выдается игроку, а в папку добавляется код монетки.

Однако объект монетки еще отображается, и ничто не указывает на то, что игрок подобрал монетку. Мы исправим это в разделе «*Создание эффектов*». Имейте в виду, что если игрок присоединится к игре повторно, независимо от сервера, монеты появляются заново. Если такое поведение вас не устраивает, вы можете задать монеткам постоянные коды и сохранять их в данных игрока:

```
local uniqueCode = 0

partFunctionsMod.RewardParts = function(part)
    local reward = part.Reward.Value
    local code = uniqueCode
    uniqueCode = uniqueCode + 1

    part.Touched:Connect(function(hit)
        local player =
            partFunctionsMod.playerFromHit(hit)

        if player then
```

```
            local tagFolder = player:FindFirstChild("CoinTags")
            if not tagFolder then
                tagFolder = Instance.new("Folder")
                tagFolder.Name = "CoinTags"
                tagFolder.Parent = player
            end

            if not tagFolder:FindFirstChild(code) then
                dataMod.increment(player, "Coins", reward)

                local codeTag = Instance.new("BoolValue")
                codeTag.Name = code
                codeTag.Parent = tagFolder
            end
        end
    end)
end
```

Один из способов заставить подогреть азарт игрока – выдавать значки. Если вы хотите награждать игрока значками при прикосновении к некоторому объекту, вы можете реализовать приведенную ниже функцию.

В этой функции мы научим `BadgeParts()` взаимодействовать с ранее созданной системой обработки объектов, а также добавим в начало модуля сервис `BadgeService`. Идентификатор значка, который нужно выдать игроку, должен иметь тип `IntValue` и храниться `BadgeId`. Помните, что нужно создать папку `BadgeParts` и добавить ее в таблицу `partGroups`. Имея `BadgeId`, мы проверяем совершившего касание игрока.

С помощью метода `UserHasBadgeAsync()` сервиса `BadgeService`, который принимает в качестве аргументов `UserId` игрока и `BadgeId`, в окне **Output** (Вывод) мы можем проверить, есть ли у игрока такой значок, чтобы не награждать его дважды. В главе 2 мы упоминали, что значок может быть вручен игроку лишь раз, даже если он удалит значок из своего инвентаря.

Если у игрока нет нужного значка, он вручается игроку с помощью метода `AwardBadge()` сервиса `BadgeService`. Этот метод тоже принимает в качестве аргументов `UserId` и `BadgeId`:

```
local badgeService = game:GetService("BadgeService")

partFunctionsMod.BadgeParts = function(part)
    local badgeId = part.BadgeId.Value

    part.Touched:Connect(function(hit)
        local player = partFunctionsMod.playerFromHit(hit)

        if player then
```

```
            local key = player.UserId
            local hasBadge = badgeService:UserHasBadgeAsync(key,
                badgeId)

            if not hasBadge then
                badgeService:AwardBadge(key, badgeId)
            end
        end
    end)
end
```

Магазины и покупки

Теперь, когда этапы игры функционально готовы, вы можете приступить к созданию монетизации, а именно магазинов, в которых можно будет тратить внутриигровую валюту и робаксы! Добавление в игру экономики позволит вам не только привлечь ваших игроков, но и заработать на внутриигровых покупках.

Премиальные покупки за робаксы

Чтобы игра приносила доход, некоторые предметы должны продаваться только за робаксы. Как упоминалось в главе 1, разработчики Roblox в основном зарабатывают деньги на продаже предметов в играх, а также от Premium Payouts. В этом разделе мы рассмотрим, как создать динамическую систему продаж, а также некоторые специальные бонусы для обби-игры.

Для реализации этой системы мы добавим новый модуль под названием `Monetization`, вложенный в сценарий `ServerHandler`. В модуле будет два сервиса, с которыми мы раньше не работали, – `InserveService` и `MarketplaceSevice`.

`InserveService` используется в основном для загрузки ресурсов с сайта по известному `assetId`. Обычно `assetId` представляет собой число, которое можно увидеть в URL-адресе ассета на сайте Roblox.

`MarketplaceSevice` используется для управления покупками в Roblox, включая обработку данных о покупках в соответствии с правилами Roblox, а также для награждения игроков после приобретения чего-либо в игре.

Для начала добавим в модуль следующий код:

```
local playerService = game:GetService("Players")
local dataService = game:GetService("DataStoreService")
local insertService = game:GetService("InsertService")
local marketService = game:GetService("MarketplaceService")
local dataMod = require(script.Parent.Data)
local monetizationMod = {}

return monetizationMod
```

Первая функция, которую мы добавим в модуль `Monetization`, будет называться `insertItem()`. Она будет загружать объекты `Gears` с сайта с помощью `InsertService` и добавлять их в `Backpack` игрока. Метод `LoadAsset()` сервиса `InsertService` загружает ассет с переданным идентификатором в игру. Ассеты, вставленные с помощью этой службы, для лучшей организации будут вложены в модель. Это означает, что купленный предмет нужно будет извлекать из модели, а модель удалять из игры с помощью метода `Destroy()`.

Метод `Destroy()` полностью удаляет из вашей игры любой экземпляр, и, как и метод `Clone()`, на некоторых экземплярах он использоваться не может. После индексации `tool` вкладывается в `Backpack` игрока:

```
monetizationMod.insertTool = function(player, assetId)
    local asset = insertService:LoadAsset(assetId)
    local tool = asset:FindFirstChildOfClass("Tool")
    tool.Parent = player.Backpack
    asset:Destroy()
end
```

Закончив с функцией `insertTool()`, мы должны написать код, который будет ее вызывать. В следующем примере кода функции добавляются в `monetizationMod` с помощью `gamepassId` или `productId` разработчика, который используется как индекс функции. Важно соблюдать этот формат, так как он позволит нам легко вызывать соответствующую функцию при совершении покупки, используя идентификатор товара.

Помните, что загружать игровые пропуски в игру нужно с помощью страницы `Create`, где нули в индексах нужно будет заменять цифрами из URL-адреса пропуска. Кроме того, мы добавили один валютный товар, который можно покупать многократно:

```
monetizationMod[000000] = function(player)
    --Ускоритель
    monetizationMod.insertTool(player, 99119158)
end

monetizationMod[000000] = function(player)
    --Гравитатор
    monetizationMod.insertTool(player, 16688968)
end

monetizationMod[000000] = function(player)
    --Рация
    monetizationMod.insertTool(player, 212641536)
end

monetizationMod[000000] = function(player)
```

```
    --100 монет
    dataMod.increment(player, "Coins", 100)
end
```

Чтобы определить, когда игрок закончил взаимодействие с подсказкой об игровом пропуске, мы используем событие `PromptGamePassPurchaseFinished` сервиса `MarketplaceService`. Само событие передается в качестве аргументов вашей функции: сюда входят игрок, которому была выдана подсказка, идентификатор отправленной подсказки и информация о том, купил ли игрок товар, который ему предложили. Если игрок купил то, что ему было предложено, мы можем вызвать уже определенную функцию с индексом, соответствующим идентификатору события.

Также добавим новый сервис под названием `CollectionService`. Он будет использоваться для назначения тегов к экземплярам и их проверки. Метод `AddTag()` сервиса принимает в качестве аргументов экземпляр и строку. Из кода видно, что игроку назначается тег со значением идентификатора купленного игрового абонемента (это будет важно позже в данном разделе).

```
local collectionService = game:GetService("CollectionService")

marketService.PromptGamePassPurchaseFinished:
                    Connect(function(player, gamePassId, wasPurchased)
    if wasPurchased then
        collectionService:AddTag(player, gamePassId)
        monetizationMod[gamePassId](player)
    end
end)
```

Для продуктов разработчиков Roblox® не ввели такой же отлаженный процесс. В нашем предыдущем примере `gamepassId` используется в виде индекса функции, для того чтобы реализовать определенное поведение при обработке покупки. Для правильной обработки покупки продукта разработчика и получения робаксов от продажи Roblox требует, чтобы вы использовали функцию события `ProcessReceipt`.

Это событие является частью сервиса `MarketplaceService` и запускается всякий раз, когда в вашей игре производится покупка продукта разработчика. Для регистрации покупок необходимо создать хранилище данных с именем `PurchaseHistory`, и лучшая практика гласит, что вы должны регистрировать `UserId` пользователя и товар, который он купил.

Связующая нить этих двух фрагментов данных используется в качестве ключа для сохранения значения `true` в хранилище данных. И последнее и самое главное – функция возвращает опцию `PurchaseGranted` из перечисления `ProductPurchaseDecision`. Так Roblox узнает, что покупка выполнена успешно, без ложных срабатываний.

В приведенной ниже функции единственный случай, когда покупка может не удастся, происходит, когда игрока больше не существует, то есть он покидает игру до обработки покупки. В этом случае мы возвращаем опцию `NotProceededYet`. Приведенную ниже функцию необходимо вставить в модуль без изменений. Обратим также внимание на функцию события `PromptProductPurchaseFinished`. Это событие запускается всякий раз, когда пользователь закрывает окно покупки, как и событие `PromptGamePassPurchaseFinished` для игровых пропусков.

Эта функция вызывает в модуле функцию с идентификатором продукта, что позволяет вам определить поведение при покупке продуктов аналогично игровому пропуску. Добавьте этот код в модуль без изменений, если не уверены в них на сто процентов:

```
local PurchaseHistory = dataService:GetDataStore("PurchaseHistory")

function marketService.ProcessReceipt(receiptInfo)
    local playerProductKey = receiptInfo.PlayerId .. ":"
        .. receiptInfo.PurchaseId
    if PurchaseHistory:GetAsync(playerProductKey) then
        return
            Enum.ProductPurchaseDecision.PurchaseGranted
    end

    local player =
    playerService:GetPlayerByUserId(receiptInfo.PlayerId)
    if not player then
        return
            Enum.ProductPurchaseDecision.NotProcessedYet
    end

    PurchaseHistory:SetAsync(playerProductKey, true)
    return Enum.ProductPurchaseDecision.PurchaseGranted
end

marketService.PromptProductPurchaseFinished:
  Connect(function(playerId, productId, wasPurchased)
    if wasPurchased then
        local player = playerService:GetPlayerByUserId(playerId)
        monetizationMod[productId](player)
    end
end)
```

Чтобы узнать больше о том, зачем в вашей игре нужна такая функция, вы можете почитать ее описание на сайте разработчика: **https://developer.roblox.com/en-us/api-reference/callback/MarketplaceService/ProcessReceipt.**

Теперь, когда у игроков есть необходимые инструменты, нужно добавить новый вызов функции в коде функции события CharacterAdded модуля Initialize. Эта функция с именем givePremiumTools() проверит, есть ли у игрока какие-либо инструменты игрового пропуска, и добавит их в рюкзак игрока, если они есть.

С помощью метода UserOwnsGamePassAsync() сервиса MarketplaceSErvice мы можем узнать, владеет ли игрок игровым пропуском, передав функции UserId и gamepassId. Этот метод работает не так, как вы могли бы ожидать. В 2018 году он пришел на замену устаревшему методу UserHasPass(), который на какое-то время задерживал работу игры, чтобы выяснить, действительно ли пользователь владеет игровым пропуском.

Новый метод кеширует результаты вызова, то есть будет возвращать один и тот же результат на протяжении всей игровой сессии. Из-за этого, если игрок покупает пропуск во время игры, новый метод об этом так и не узнает.

Для решения данной проблемы мы будем использовать временные теги в функции события PromptGamePassPurchaseFinished, а также в методе UserOwnsGamePassAsync(), чтобы определить, давать ли игроку инструменты при возрождении.

Обратите внимание, что у нас есть таблица идентификаторов игровых пропусков, которая перебирается циклом for с вызовом функции в модуле Monetization, если игрок владеет игровым пропуском. Код для предоставления игрокам их премиум-инструментов выглядит следующим образом:

```
local collectionService = game:GetService("CollectionService")
local marketService = game:GetService("MarketplaceService")
local monetization = require(script.Parent.Monetization)
local toolPasses = {000000, 000000, 000000}

-- Вызовите эту функцию в событии CharacterAdded ---модуля
-- инициализации для использования инструментов игрового пропуска
initializeMod.givePremiumTools = function(player)
    for _, ID in pairs(toolPasses) do
        local key = player.UserId
        local ownsPass = marketService:UserOwnsGamePassAsync(key, ID)
        local hasTag = collectionService:HasTag(player, ID)

        if hasTag or ownsPass then
            monetization[ID](player)
        end
    end
end
```

Теперь, когда обработка покупок выполнена, мы введем новую функцию в модуль PartFunctions. Она будет называться PurchaseParts(), и вам нужно

будет добавить в рабочее пространство папку с тем же именем и добавить ее в таблицу `partGroups`. Цель этой функции – вывести игроку окошко с пропуском или продуктом разработчика при касании с объектом. Такие окошки обычно выглядят так, как показано на рис. 5.2.

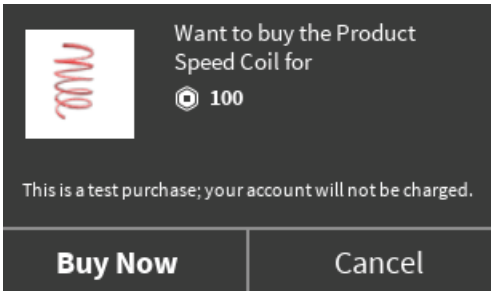


Рис. 5.2. Это стандартный интерфейс, который видят игроки, когда им предлагается совершить покупку

После сигнатуры функции `PurchaseParts()` мы создаем две переменные, которые хранят в объекте значения `PromptId` и `IsProduct`. Переменная `promptId` содержит идентификатор игрового абонемента или продукта разработки, который должен выдаваться при касании с объектом. Переменная `isProduct` содержит логическое значение, определяющее, является ли продукт игровым пропуском или продуктом разработчика.

Эта переменная важна, так как всплывающее окошко у разных типов продукта разное. Задав эти два значения, мы вводим условие в зависимости от типа покупки. Если тип покупки – продукт разработчика, то вызывается метод `PromptPurchase()` сервиса `MarketplaceService`, который получает на вход игрока и `promptId`.

Если приобретается игровой пропуск, вызывается метод `PromptGamePassPurchase()` сервиса `MarketplaceService` с такими же аргументами. Приведенная ниже функция открывает окошко покупки в зависимости от продукта:

```
local marketService = game:GetService("MarketplaceService")

partFunctionsMod.PurchaseParts = function(part)
    local promptId = part.PromptId.Value
    local isProduct = part.IsProduct.Value

    part.Touched:Connect(function(hit)
        local player =
            partFunctionsMod.playerFromHit(hit)
        if player then
            if isProduct then
                marketService:PromptProductPurchase(player,
                  promptId)
```

```
            else
                marketService:PromptGamePassPurchase(player,
                  promptId)
            end
        end
    end)
end
```

Создание магазинов с внутриигровой валютой

Теперь, когда игроки могут совершать в игре покупки, нужно создать полезную для всех систему магазина, в которой можно было бы потратить монеты, запрограммированные нами ранее.

Для примера добавим в магазин один товар, экземпляр `Tool`, который можно будет копить с помощью **Coins**. Чтобы создать свой инструмент, просто добавьте в рабочую область новую деталь с именем `Handle`. Назовем его `Зелье прыгучести`. Этот и любые другие созданные вами инструменты должны храниться в папке `ShopItems` под `ReplicatedStorage`. `Зелье прыгучести` – это расходуемый предмет, увеличивающий высоту прыжка на 30 секунд.

Важно, чтобы игрок после смерти не терял купленный предмет. Для этого сперва добавим в `Tool` сценарий `LocalScript`, который может называться как угодно, например `ToolHandler`. Затем мы должны взять индекс игрока, его персонажа, сам инструмент и мышь игрока. Также мы создадим три переменные, содержащие информацию о том, экипирован ли инструмент, использовал ли игрок инструмент, и значение `JumpPower` персонажа игрока, которое должно быть установлено.

Используя события `Equipped` и `Uneqipped` экземпляров `Tool`, мы можем создать две функции, которые изменяют значение переменной `equipped` для отслеживания состояния инструмента. С помощью события `Button1Down` мыши мы можем определить, когда игрок делает щелчок. Если `Tool` экипирован и игрок совершает щелчок, свойство `JumpPower` должно измениться с 50 на значение переменной `JUMP_POWER`, равное 90. Затем с помощью функции `delay()` программа ждет 30 секунд, после чего возвращает силу прыжка по умолчанию и инструмент. Код `ToolHandler` выглядит следующим образом:

```
local playerService = game:GetService("Players")
local player = playerService.LocalPlayer
local char = player.Character or player.CharacterAdded:Wait()
local tool = script.Parent
local mouse = player:GetMouse()

local equipped
local clicked = false
```

```
local JUMP_POWER = 90

tool.Equipped:Connect(function()
    equipped = true
end)

tool.Unequipped:Connect(function()
    equipped = false
end)

mouse.Button1Down:Connect(function()
    if equipped and not clicked then
        clicked = true
        char.Humanoid.JumpPower = JUMP_POWER

        delay(30, function()
            char.Humanoid.JumpPower = 50
            tool:Destroy()
        end)
    end
end)
```

Создав предмет, который можно дать игроку, давайте вернемся в модуль `PartFunctions` и добавим новую папку и функцию с именами `ShopParts`. Помните, что для объектов вам необходимо добавить новую папку в рабочую область и добавить папку в таблицу `partGroups`.

Сперва нужно создать таблицу продуктов, где имя связанного `Tool` будет использоваться в качестве индекса, а в таблице будет храниться ценная информация. Как и в случае с таблицей `defaultData`, вы можете хранить эту информацию в модуль для удобства организации. Поскольку хранилище `ReplicatedStorage` еще не определено в модуле `PartFunctions`, вам нужно будет сделать для него переменную. Теперь вы должны использовать тот же формат для создания функции `ShopParts()`.

В функции события `Touched` название предмета, который игрок хочет купить, берется из `StringValue` с именем `ItemName`; затем имя элемента используется для индексации информации инструмента из таблицы `items`. Как только игрок, прикоснувшийся к объекту, определен, нужно проверить, что у него достаточно денег для совершения покупки, зная цену товара из таблицы и используя метод `get()` модуля `Data`.

Если у игрока достаточно валюты для совершения покупки, ее цена вычитается из его текущего количества монет с помощью метода `increment()` модуля `Data`.

Зная индекс товара, мы копируем `Tool`, используя метод `Clone()` экземпляра, и связываем его с рюкзаком игрока. Метод `Clone()` создает точную

копию переданного экземпляра, но не является дочерним для других объектов.

Следует помнить, что не каждый экземпляр можно клонировать, особенно если это что-то важное, например сервис (это приведет к ошибке). Как упоминалось в главе 4, `LocalScript` в экземпляре `Tool` будет вложен в рюкзак игрока, а это означает, что программа экземпляра `tool` теперь будет работать как положено без каких-либо дополнительных входных данных из других источников:

```
local items = {
    ["Зелье прыгучести"] = {
        Price = 5;
    } ;
}

local replicatedStorage = game:GetService("ReplicatedStorage")

partFunctionsMod.ShopParts = function(part)
    local itemName = part.ItemName.Value
    local item = items[itemName]

    part.Touched:Connect(function(hit)
        local player = partFunctionsMod.playerFromHit(hit)

        if player and dataMod.get(player, "Coins") >= item.Price then
            dataMod.increment(player, "Coins", - item.Price)
            local shopFolder = replicatedStorage.ShopItems
            local tool = shopFolder:FindFirstChild(itemName):Clone()

            tool.Parent = player.Backpack
        end
    end)
end
```

Борьба с эксплойтами

Конкретно в этой игре особо нечего использовать в качестве эксплойта, но мы все равно будем применять лучшие практики безопасности. По возможности данные нужно получать от сервера, а не брать их у клиента. Кроме того, отслеживая собранные игроком монеты на сервере, клиент не может злоупотреблять этим и запрашивать лишние монеты.

В следующей главе вы будете работать с системами, которые должны получать информацию от клиента, и в таких ситуациях нужно будет исполь-

зовать проверки работоспособности и внедрять меры безопасности. На данный момент вы можете придумать функцию, удаляющую все экземпляры `BodyMover`, которые клиент добавляет персонажу. Это нужно для того, чтобы читеры не могли летать по уровням.

Теперь игроки могут покупать в игре предметы с помощью робаксов и внутриигровой валюты. Вы уже знаете чуть больше о безопасности и кое-что из мер безопасности уже реализовали. В следующих разделах мы создадим интерфейс для игры.

Настройка внешнего интерфейса (фронтенд)

Теперь ваша игра функционально работает и дополнена монетизацией. Интерфейс – это та часть игры, с которой игроки взаимодействуют. В этом разделе мы украсим сцену, оживим игру с помощью эффектов и движений, а также дадим клиенту визуальную и звуковую информацию о происходящем.

Начнем с создания эффектов.

Создание эффектов

При создании эффектов основная часть работы выполняется на клиенте. Все дело в том, что большинство эффектов должны появляться только локально, чтобы не тратить производительность на их воспроизведение. Например, когда вы подбираете монету, звук подбора должен быть слышен только вам.

Для начала создадим в `StarterPlayerScripts` новый сценарий с именем `LocalHandler`, который содержит тот же код, что и `ServerHandler`. Структура систем в целом будет такой же, а код разделен на модуле. Начнем добавление эффектов с создания в `ClientHandler` нового модуля под названием `Effects`. Вам нужно будет `ReplicatedStorage`, потому что нам нужно будет использовать связь «клиент–сервер» экземпляра `RemoteEvent`:

```
local replicatedStorage = game:GetService("ReplicatedStorage")
local effectsMod = {}

return effectsMod
```

После создания модуля для будущих эффектов нужны вспомогательные функции. Первая функция поможет нам работать со звуками. Звуковой дизайн в нашей игре не самая важная часть, но наверняка вы захотите добавить фоновую музыку и некоторые звуковые эффекты при контакте игрока с объектом.

Циклическая фоновая музыка создается в рабочем пространстве с помощью экземпляра `Sound`, у которого параметры `Playing` и `Looped` равны `true`, а вот для настройки звука при активации потребуется немного программи-

рования. В следующем примере код будет воспроизводить экземпляр Sound внутри переданного объекта.

Удобно вводить такой механизм в папке RewardParts, чтобы игрок мог ассоциировать определенный звук с получением награды. Для следующего кода мы используем метод FindFirstChildOfClass(), имеющийся у всех экземпляров, чтобы найти экземпляр Sound переданного объекта. Если звук нашелся, то он возвращается, а затем запускается его метод Play, воспроизводя звук там, где его может слышать только клиент, поскольку вызов был сделан из модуля, выполняемого клиентом:

```
local function playSound(part)
    local sound = part:FindFirstChildOfClass("Sound")

    if sound then
        sound:Play()
    end

    return sound
end
```

Аналогично воспроизведению звука, мы можем ввести функцию, которая может испускать частицы, если у объекта есть ParticleEmitter. Частицы, о которых мы говорили в главе 4, представляют собой простой способ украсить игру и привлечь внимание игроков. В следующей функции, найдя экземпляр ParticleEmitter, мы вызываем метод Emit(), который может испустить несколько частиц.

Помните, что по умолчанию ParticleEmitter должен быть отключен. Более того, метод Emit() выпустит все частицы сразу, поэтому если вы хотите просто включить излучатель, а через некоторое время выключить его, вы можете установить свойство Enabled экземпляра ParticleEmitter равным true и реализовать функцию delay, чтобы отключить его:

```
local function emitParticles(part, amount)
    local emitter = part:FindFirstChildOfClass("ParticleEmitter")

    if emitter then
        emitter:Emit(amount)
    end

    return emitter
end
```

Теперь, как и в модуле PartFunctions, мы создаем функции, соответствующие имени папки, чтобы их легче было вызывать в модуле PartFunctions, куда вы ходите добавить звук или частицу. Для этого вы должны вставить строку, начинающуюся со слова server, после выполнения всех проверок.

Вам также понадобится новый расположенный в ReplicatedStorage экземпляр RemoteEvent под названием Effect, чтобы сервер мог отправлять сигналы клиенту. Эта строка будет посылать клиенту сигнал, передавая в нем объект, с которым столкнулся игрок, чтобы клиенту не нужно было выполнять обнаружение попаданий для всех объектов в игре. Как только сигнал будет отправлен, он обнаружится на стороне клиента в модуле Effects.

Создадим функцию события OnClientEvent, передадим ей имя папки группы объектов от ее родителя и вызовем функцию с соответствующим именем в модуле Effects. Функции в модуле смогут вызывать уже определенные вспомогательные функции, передавая им объект. Для объектов, являющихся родительскими и для RewardParts, и для SpawnParts, мы проверяем и наличие частиц, и наличие звука.

Даже если вы не добавите звук и частицы, функция все равно будет работать. Для проверки в папке SpawnParts дополнительно поменяем материал на неон, а потом снова на гладкий пластик с помощью функции delay():

```
--сервер
-- поместите это в функцию детали после выполнения всех проверок
replicatedStorage.Effect:FireClient(player, part)

--клиент
replicatedStorage.Effect.OnClientEvent:Connect(function(part)
    local folderName = part.Parent.Name
    effectsMod[folderName](part)
end)

effectsMod.RewardParts = function(part)
    part.Transparency = 1
    playSound(part)
end

effectsMod.SpawnParts = function(part)
    playSound(part)
    emitParticles(part, 50)
    part.Material = Enum.Material.Neon

    delay(1, function()
        part.Material = Enum.Material.SmoothPlastic
    end)
end
```

Еще один эффект, который позволит вдохнуть в игру больше жизни, – это движение объекта. Его можно выполнять на сервере, но делать так, как правило, не рекомендуется, поскольку сервер обновляется медленнее, а кли-

ент может не иметь идеального соединения, что приводит к замедлениям или рывкам в эффектах. Для начала составим таблицу объектов в модуле `Effects` так, чтобы объекты из любой группы можно было вращать, если они содержат значение.

Для вращения объектов мы будем использовать специальный цикл в `RunService`. `RunService` используется для создания циклов, которые работают на более высоких скоростях, чем те, которых можно достичь с помощью функции `wait()`. Событие `RenderStepped` доступно только при использовании `RunService` на клиенте и срабатывает при рендеринге каждого кадра. Событие проходит в определенный период времени, обратный количеству кадров в секунду, то есть если игроку отображается 60 кадров в секунду, `dt` равняется 1/60.

После определения папки, связанной с объектом, мы смотрим, содержит ли деталь значение `Vector3Value` с именем `Rotate`. Если значение задано, то объект добавляется в таблицу, где содержатся все объекты, которые нужно повернуть. Тип значения `Vector3` позволяет настраивать скорости вращения деталей. Как вы увидите, скорость задается в градусах в секунду.

Функция события `RenderStepped` запускается каждый кадр, поворачивая все объекты в цикле `for`. У каждого объекта находится экземпляр `Vector3Value`, и его значение присваивается переменной `rot`. Затем этот вектор умножается на значение `dt`, чтобы преобразовать каждый компонент в количество градусов в секунду. Потом этот новый вектор преобразуется в радианы с помощью другого векторно-скалярного умножения, дабы значение можно было использовать для работы с `CFrame`.

Наконец, мы строим новый `CFrame.Angles()`, взяв новый вектор и умножая его на текущий `CFrame`, что дает вращение:

```
local runService = game:GetService("RunService")
local rotParts = {}

local partGroups = {
    workspace.KillParts;
    workspace.DamageParts;
    workspace.SpawnParts;
    workspace.RewardParts;
    workspace.PurchaseParts;
    workspace.BadgeParts;
    workspace.ShopParts;
}

for _, group in pairs(partGroups) do
    for _, part in pairs(group:GetChildren()) do
        if part:IsA("BasePart") then
            if part:FindFirstChild("Rotate") then
```

```
                table.insert(rotParts, part)
            end
        end
    end
end

runService.RenderStepped:Connect(function(dt)
    for _, part in pairs(rotParts) do
        local rot = part.Rotate.Value
        rot = rot * dt
        rot = rot * ((2 * math.pi) / 360)
        rot = CFrame.Angles(rot.X, rot.Y, rot.Z)

        part.CFrame = part.CFrame * rot
    end
end)
```

Заметим также, что, комбинируя `RunService` с другими циклами, можно создать больше эффектов механики с использованием движущихся частей, включая лифты, движущиеся платформы, между которыми игроки могут прыгать, и другие препятствия.

Наконец, нельзя упускать из виду работу с освещением. Чрезмерно яркое или перенасыщенное освещение может оттолкнуть игроков. Многие разработчики игр в жанре обби совершают ошибку, резко увеличивая насыщенность и яркость, чтобы сцены выглядели более детскими или мультяшными, но результат чаще становится раздражающим, чем ярким и веселым. Вопросы освещения остаются на вас, но следите за тем, чтобы карта удобно просматривалась, чтобы вы четко могли видеть всю сцену, на которой находитесь в данный момент.

Теперь наша игра стала более интерактивной и красочной, что позволяет привлечь новых игроков. В следующих разделах мы поговорим о публикации игры и о том, что будем делать в будущем.

Тестирование и публикация

Теперь, когда и фронтенд, и бэкенд игры готовы, важно протестировать ее этапы и убедиться, что их можно пройти и что все их компоненты функционируют так, как и задумано. В случае если вы реализовали еще и собственный функционал, следует дважды проверить, что вся система работает. Убедитесь, что ваши покупки обрабатываются правильно и приобретенные предметы попадают к игроку, чтобы игроки не жаловались на то, что не получили товар, за который заплатили.

Лучший способ искать ошибки – нажать сочетание клавиш *Shift+F9* и открыть консоль разработчика. Эта консоль позволит вам во время игры

просматривать выходные данные, предупреждения и ошибки как на сервере, так и на клиенте, как показано на рис. 5.3. Доступ к панели команд на стороне сервера имеют только те, у кого есть права на редактирование игры (рис. 5.3).

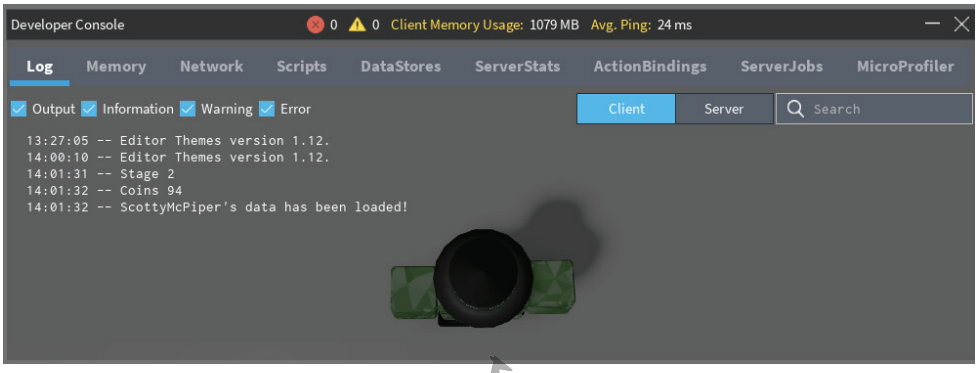


Рис. 5.3. Консоль разработчика используется для вывода информации во время игры

После самостоятельного тестирования хорошо бы заодно получить отзывы от друзей и дать им возможность поиграть, найти какие-нибудь проблемы и дать обратную связь. Если вы готовы выпустить свою игру, то стоит начать с небольшого количества спонсоров или рекламы, чтобы получить больше отзывов от зачастую слишком честной публики и посмотреть, какие изменения они предлагают и какие ошибки им удалось найти. Когда речь идет о поиске ошибок в игре, тестов много не бывает.

Как только ваше тестирование завершится и спонсоры начнут работу, вы можете себя официально поздравить с выпуском первой игры! Нет лучшего чувства, чем смесь счастья, облегчения, а иногда и беспокойства по поводу публикации игры. Лучше всего оставить игру в покое и следить за ее производительностью. Если у вас есть место, где игроки могут оставлять отзывы, будь то канал в Discord, личные сообщения или стена группы, то соберите их отзывы и внимательно проанализируйте их.

Резюме

Мы закончили создание вашей первой полноценной игры на Roblox. Из этой главы вы узнали, как управлять игроками и их данными и создавать системы, добавляющие функциональные возможности вашей игре в целом, а также рассмотрели способы привлекать и удерживать аудиторию. Позже мы создадим еще одну игру, в которой будут использоваться некоторые системы из этой главы и множество новых, позволяющих внедрить в игру более сложную механику.

В следующей главе вы узнаете, как создать игру в жанре «Королевская битва». Этот жанр сегодня чрезвычайно популярен и реализован в том или

ином виде в Fortnite, PUBG, COD: Warzone и других играх. Разработка игры такого типа даст вам больше опыта работы с клиентом и внедрения дополнительных методов безопасности, поскольку мы займемся созданием оружия, работой с пользовательским интерфейсом и будем более активно использовать связь между клиентом и сервером.

Глава 6

Создание игры в жанре «Королевская битва»

В предыдущей главе вы создали вашу первую полноценную игру. Жанр обби популярен отчасти из-за того, что игры в этом жанре легко создавать. В этой главе мы создадим игру в жанре «Королевская битва», в котором игроки телепортируются на поле битвы, где они должны искать оружие и сражаться друг с другом и последний выживший игрок побеждает.

В этом проекте потребуются все знания, которые вы уже получили из книги, и, кроме того, мы изучим новый материал. Мы более глубоко займемся работой с пользовательским интерфейсом, а также методами безопасности для систем вроде оружия и других, в которых клиент общается с сервером.

Здесь мы рассмотрим следующие основные темы:

- настройка серверной части;
- управление данными игрока;
- создание системы раундов;
- создание оружия;
- локальная репликация;
- спаун предметов;
- настройка интерфейса.

Технические требования

Как и в предыдущей главе, вы будете работать исключительно в Studio. Наличие подключения к интернету позволит вам эффективнее работать в Studio и проводить независимые исследования по любым затронутым темам.

Весь код этой главы можно найти в репозитории книги на GitHub: **https://github.com/PacktPublishing/Coding-Roblox-Games-Made-Easy/tree/main/Chapter06**.

Настройка серверной части

Как и при создании обби-игры, мы сделаем серверную часть игры модульной. Как и раньше, мы создадим основной серверный сценарий с именем

ServerHandler, в который будут вложены все серверные модули. Вам не нужно будет возвращаться к предыдущей главе, так как весь код ServerHandler будет приведен здесь:

```
for _, module in pairs(script:GetChildren()) do
    local loadMod = coroutine.create(function()
        require(module)
    end)

    coroutine.resume(loadMod)
end
```

Мы один за одним представим модули, которые нужно будет добавить в сценарий ServerHandler для реализации основных функций. Как и в предыдущей главе, вам нужно будет протестировать каждую созданную вами систему, не учитывая зависимостей от еще не созданных модулей. Выполнение тестирования на протяжении всего процесса разработки не только гарантирует отсутствие ошибок, но и мотивирует!

Управление данными игрока

Управление данными игрока будет легко реализовать, поскольку вы уже создали систему хранения данных. Вам лишь нужно вложить модуль Data, созданный в предыдущей главе, под сценарий ServerHandler. При создании игр в дальнейшем вы тоже можете использовать этот модуль в готовом виде.

> **Важное примечание**
>
> Если вы не выполняли этапы из предыдущей главы, вам следует вернуться к ней и воспроизвести их. Выполнив инструкции из раздела «*Управление данными игрока*», вы можете вернуться сюда.

Для этой игры нужно будет внести в модуль Data небольшие изменения, так как отслеживаемые данные у нас изменятся. В предыдущей главе мы отслеживали только текущий этап игрока и количество монет у него. Для игры в жанре «Королевская битва» нам нужно отслеживать количество монет, количество убитых врагов и количество выигранных игр. Изменим таблицу defaultData, добавив статистики Wins, Kills и Conis. Кроме того, нам нужны будут связанные с ними экземпляры ValueBase для отображения статистик на экране игрока с помощью системы leaderstats:

```
local defaultData = {
    Coins = 0;
```

```
    Wins = 0;
    Kills = 0;
}

playerService.PlayerAdded:Connect(function(player)
    local folder = Instance.new("Folder")
    folder.Name = "leaderstats"
    folder.Parent = player

    local coins = Instance.new("IntValue")
    coins.Name = "Coins"
    coins.Parent = folder
    coins.Value = defaultData.Coins

    local wins = Instance.new("IntValue")
    wins.Name = "Wins"
    wins.Parent = folder
    wins.Value = defaultData.Wins

    local kills = Instance.new("IntValue")
    kills.Name = "Kills"
    kills.Parent = folder
    kills.Value = defaultData.Kills

    dataMod.setupData(player)
end)
```

Важно отметить, что если вы изменили ключ экземпляра `GlobalDataStore`, его можно сбросить, так как данные в разных играх Roblox сохраняются.

Настройка системы раундов

В основе этой игры будет лежать настройка игрового цикла или системы раундов. Эта часть бэкенда позволит вам делать между матчами паузы, добавлять на поле боя игроков, показывать всем, кто победил в матче, и повторять весь процесс.

Перед созданием цикла нам нужно создать лобби, в котором игроки будут находиться между матчами или после смерти в текущем матче. Это лобби проектируется на ваше усмотрение и может, например, включать в себя кнопки для открытия окон покупки предметов вроде тех, что вы создали в предыдущей главе. Игроки, скорее всего, какое-то время проведут в лобби, так что вы должны удерживать их внимание. Чтобы мертвые игроки не скучали, вам нужно создать графический интерфейс, который позволит вам наблюдать за другими игроками (мы рассмотрим, как это сделать).

Кроме того, вы можете сделать простой вариант «обби» без контрольных точек, в котором игроки могли бы получить дополнительные монеты, пока ждут в лобби.

Единственный функционал, который должен быть реализован в лобби, – это несколько экземпляров `SpawnLocation`. Это точки, на которых игроки спаунятся по умолчанию, если какой-либо код не направит их в другое место. Как только вы добавите в лобби экземпляр `SpawnLocation`, свойству `Duration` нужно будет задать малое значение или вообще `0`. Это свойство определяет, как долго у игроков есть бессмертие (экземпляр `ForceField`) после спауна, но для этой игры оно не нужно.

Начнем создание системы раундов с нового модуля с именем `GameRunner` под сценарием `ServerHandler`. В этом модуле будут службы, переменные и объекты, необходимые для создания игрового цикла.

Вы увидите, что в переменных `message` и `remaining` сохраняются пути к двум экземплярам `StringValue` с именами `message` и `remaining` соответственно. Они будут использоваться для вывода клиенту информации в различных частях игрового цикла. Добавьте эти два `StringValue` в `ReplicatedStorage`, задав правильные имена.

Переменные, чьи имена написаны полностью заглавными буквами, называются **константами**. В Lua нет констант как специальной конструкции, как в других языках, но принято писать константы заглавными буквами, как показано в следующем блоке кода:

```
local playerService = game:GetService("Players")
local replicatedStorage = game:GetService("ReplicatedStorage")
local dataMod = require(script.Parent.Data)
local random = Random.new()
local message = replicatedStorage.Message
local remaining = replicatedStorage.Remaining
local gameRunner = {}
local competitors = {}

local MIN_PLAYERS = 2
local INTERMISSION_LENGTH = 15
local ROUND_LENGTH = 300
local PRIZE_AMOUNT = 100

return gameRunner
```

Давайте рассмотрим назначение констант, которые мы объявили в модуле:

- значение константы `MIN_PLAYERS` будет использоваться для определения того, сколько игроков должно быть в матче, чтобы он вообще мог начаться. Это значение по умолчанию равно 2, так как игра с од-

ним игроком заканчивалась бы мгновенной победой, что позволяло игрокам бесконечно получать награды на пустом сервере;

- вторая переменная, `INTERMISSION_LENGTH`, используется для создания паузы между матчами, когда на сервере находится достаточное количество игроков. Это значение должно быть довольно большим, чтобы игроки в лобби могли пообщаться и отдохнуть перед началом нового матча;
- значение `ROUND_LENGTH` будет использоваться аналогичным образом, но уже после запуска матча. Если к концу отведенного времени победитель не обнаружился, то матч заканчивается без победителя. Это значение должно дать игрокам достаточно времени, чтобы сражаться без спешки, но не так много, чтобы два последних живых игрока могли тянуть время слишком долго;
- последнее значение, `PRIZE_AMOUNT`, будет использоваться для награждения победителя раунда монетами. Эти значения вы можете свободно менять, когда внедрите код в модуль.

Как мы уже говорили в предыдущих главах, не каждая вспомогательная функция в вашем модуле должна быть частью возвращаемой таблицы модуля, если используется только с изолированной задачей модуля. Приведенные ниже функции, которые используются для получения и удаления игроков из таблицы `competitors`, реализованы в модуле как локальные функции, поскольку существует не так много практических сценариев, в которых они должны были бы использоваться за пределами модуля. Функция `getPlayerInTable()` получает экземпляр игрока и возвращает индекс игрока в таблице. Он используется функцией `removePlayerFromTable()`, которая будет вызываться, когда игрок умирает или выходит из игры. Этот метод получает индекс игрока, который вышел из игры или умер, прежде чем удалить его из таблицы `competitors`. Вы должны без изменений реализовать следующий код в модуле `GameRunner`:

```
local function getPlayerInTable(player)
    for i, competitor in pairs(competitors) do
        if competitor == player then
            return i, player
        end
    end
end

local function removePlayerFromTable(player)
    local index, _ = getPlayerInTable(player)

    if index then
        table.remove(competitors, index)
```

```
    end
end

playerService.PlayerRemoving:Connect(function(player)
    removePlayerFromTable(player)
end)
```

Когда игрок готовится ворваться на поле битвы, нам нужно подготовить его. Этим займется функция `preparePlayer()`. В ней игроку выдается простое оружие по умолчанию, создание которого мы рассмотрим в разделе «*Создание оружия*» этой главы, а также создается новая функция события на случай смерти игрока. С помощью экземпляра `Humanoid` персонажа игрока мы можем определить смерть игрока через событие `Died`, а затем вызвать функцию `removePlayerFromTable()`. Добавив следующий код в модуль, вы можете закомментировать строки, касающиеся `defaultWeapon`, так как до создания оружия мы еще не добрались:

```
local function preparePlayer(player)
    local char = player.Character or
        player.CharacterAdded:Wait()
    local hum = char:WaitForChild("Humanoid")

    local defaultWeapon = replicatedStorage.Weapons.M1911:Clone()
    defaultWeapon.Parent = player.Backpack

    hum.Died:Connect(function()
        removePlayerFromTable(player)
    end)
end
```

Функция `addPlayersToTable()` будет использоваться для добавления в таблицу `competitors` в начале матча. С помощью метода `GetPlayers()` сервиса `Players` мы получаем таблицу каждого игрока, подключенного в данный момент к игре, и проверяем, что игрок, которого мы хотим добавить, жив. Это нужно для того, чтобы игрок не застрял в лобби, если он возродится уже после того, как код его перемещения на поле боя выполнится. После выполнения этого условия игрок добавляется в таблицу `competitors` и подготавливается с помощью функции `preparePlayer()`. Вставьте приведенную ниже функцию в ваш модуль под другими функциями, которые он должен вызывать:

```
local function addPlayersToTable()
    for _, player in pairs(playerService:GetPlayers()) do
        local char = player.Character or
```

```
            player.CharacterAdded:Wait()

        if char.Humanoid.Health > 0 then
            table.insert(competitors, player)
            preparePlayer(player)
        end
    end
end
```

Теперь нам нужно будет вывести игроков из лобби на поле боя. Для этого добавим в рабочей области папку под названием `Spawns`. Важно, чтобы на каждого игрока была хотя бы одна точка спауна. Точки спауна должны быть обычными экземплярами `Part`, а не `SpawnLocation`, как в лобби.

Создав папку и точки спауна, вы должны реализовать функцию `spawnPlayers()` в модуле `GameRunner`. Эта функция создаст новую таблицу, состоящую из всех точек спауна из `SpawnFolder`, а затем будет перебирать всех игроков в таблице `competitors`. После определения игрока и персонажа вызывается метод `NextInteger()` объекта `Random` для получения случайного индекса в диапазоне от 1 до длины таблицы. Если выпадает индекс точки спауна, то она удаляется из таблицы, чтобы другие игроки не могли появиться в той же точке:

```
local function spawnPlayers()
    local spawnPoints = workspace.Spawns:GetChildren()

    for _, player in pairs(competitors) do
        local char = player.Character or
            player.CharacterAdded:Wait()
        local randomIndex = random:NextInteger(1,
            #spawnPoints)
        local spawnPoint = spawnPoints[randomIndex]
        table.remove(spawnPoints, randomIndex)

        char:SetPrimaryPartCFrame(spawnPoint.CFrame *
            CFrame.new(0,2,0))
    end
end
```

Последняя вспомогательная функция, которую нам необходимо реализовать для этой системы, называется `loadAllPlayers()`, и она будет вызываться по окончании игры, чтобы возродить всех оставшихся игроков обратно в лобби, не убивая их. Это делается простым перебором таблицы `competitors` и вызовом метода `LoadCharacter()` экземпляра `Player`. Код показан здесь:

```
local function loadAllPlayers()
    for _, player in pairs(competitors) do
        player:LoadCharacter()
    end
end
```

Теперь, когда все вспомогательные функции были реализованы в модуле `GameRunner`, мы должны ввести основной цикл, который управляет игрой в целом. Мы реализуем этот цикл как функцию в таблице модулей, чтобы можно было легко использовать функцию `spawn()`. Эта функция будет называться `gameLoop()`, и с помощью цикла `while` можно будет бесконечно запускать игру. В начале каждого цикла по константе `MIN_PLAYERS` проверяется, достаточно ли игроков в игре, а затем запускается отсчет перерыва, а игрокам выводится соответствующее сообщение. По завершении перерыва игрокам сообщается, что пора подготовиться к бою, а затем игроки добавляются в таблицу `competitors` с помощью функции `PlayersToTable()`. Добавленные в таблицу игроки появляются в случайных точках спауна с помощью функции `spawnPlayers()`. После спауна запускается таймер матча, заданный значением `ROUND_LENGTH`, а на экран выводится количество игроков, оставшихся в раунде.

Вы можете заметить, что у цикла `repeat` есть два условия выхода из таймера. Первое условие: если количество игроков таблице `competitors` становится меньше или равно 1, а второе условие: если значение `gameTime` доходит до 0. Как только цикл завершается, условный оператор проверяет, какое из условий сработало. Если количество участников равно 0 или `gameTime` равно 0, это означает, что победитель не определился, так как либо два последних игрока убили друг друга одновременно, либо вышло время, отведенное на раунд. Если же остался один игрок, значит, он и есть победитель. Определить победителя легко, так как он будет первым и единственным элементом в таблице `competitors`. Количество побед у него увеличивается на 1, а к статистике `Coins` добавляется значение `PRIZE_AMOUNT`. Кроме того, в течение 5 секунд отображается имя победителя и короткое сообщение о победе, а затем начнется новый матч.

Я рекомендую после внедрения кода в модуль внимательно изучить код, чтобы понять, как в целом этот цикл заставляет игру работать:

```
gameRunner.gameLoop = function()
    while wait(0.5) do
        if #playerService:GetPlayers() < MIN_PLAYERS then
            message.Value = "Нужно "..MIN_PLAYERS.. " игроков для начала."
        else
            local intermission = INTERMISSION_LENGTH
            repeat
                message.Value = "Ожидание: "..intermission
                intermission = intermission - 1
```

```
                wait(1)
            until intermission == 0

            message.Value = "Приготовьтесь..."
            wait(2)
            addPlayersToTable()
            spawnPlayers()

            local gameTime = ROUND_LENGTH
            repeat
                message.Value = "Осталось времени: "..gameTime
                remaining.Value = #competitors.. " осталось"
                gameTime = gameTime - 1
                wait(1)
            until #competitors <= 1 or gameTime == 0

            loadAllPlayers()
            remaining.Value = ""
            if gameTime == 0 or #competitors == 0 then
                message.Value = "Победителей нет..."
            else
                local winner = competitors[1]
                dataMod.increment(winner, "Wins", 1)
                dataMod.increment(winner, "Coins", PRIZE_AMOUNT)
                message.Value = winner.Name.." выиграл раунд!"
            end

            competitors = {}
            wait(5)
        end
    end
end

spawn(gameRunner.gameLoop)
```

Основной игровой цикл готов, теперь нам нужно добавить некоторые элементы, которые сделают геймплей таким, каким мы его хотим видеть. В следующем разделе мы рассмотрим, как создавать оружие, чтобы игроки могли показать друг другу, кто здесь чемпион.

Создание оружия

Не бывает «Королевской битвы» без оружия. В этом разделе вы узнаете, как создать комплексную и безопасную систему оружия с использованием рейкастинга для обнаружения попаданий.

Начнем с добавления в сценарий `ServerHandler` нового модуля под названием `Weapons`. В этот модуль мы добавим сервисы, переменные и другие не-

обходимые вещи. Из переменных `chitRemote` и `replicateRemote` видно, что нам понадобится два экземпляра `RemoteEvent`, вложенных в `ReplicatedStorage`, с именами `Hit` и `Replicate`. Они будут использоваться для того, чтобы клиент и сервер могли при необходимости взаимодействовать друг с другом. Добавьте следующий блок кода в свой новый модуль:

```
local playerService = game:GetService("Players")
local replicatedStorage = game:GetService("ReplicatedStorage")
local hitRemote = replicatedStorage.Hit
local replicateRemote = replicatedStorage.Replicate
local dataMod = require(script.Parent.Data)
local weapons = {}

return weapons
```

В предыдущей главе у нас была функция `playerFromHit()`. Мы будем использовать ее в модуле `Weapons`, чтобы проверять выстрелы игроков. Она задана в таблице модуля, а не как локальная вспомогательная функция, поскольку она нужна и другим системам. Вы можете добавить эту функцию в таблицу модулей без каких-либо изменений:

```
weapons.playerFromHit = function(hit)
    local char = hit:FindFirstAncestorOfClass("Model")
    local player = playerService:GetPlayerFromCharacter(char)
    return player, char
end
```

Следующей частью этой системы будет создание нового экземпляра `Tool` в сервисе `StarterPack`. К инструменту можно добавить новую `Part` под названием `Handle`. Согласно названию, в этой точке персонаж клиента будет держаться за `Tool`. Кроме того, вы должны добавить в `Tool` новый модуль `Settings`, который будет содержать настройки для вашего оружия. Его код приведен здесь:

```
local gunSettings = {
    fireMode = "SEMI"; --SEMI или AUTO
    damage = 15;
    headshotMultiplier = 1.5;
    rateOfFire = 300; --Количество выстрелов в минуту
    range = 500;
    rayColor = Color3.fromRGB(255, 160, 75);
    raySize = Vector2.new(0.25, 0.25); --Ширина и высота
    debrisTime = 0.05;
}

return gunSettings
```

Давайте посмотрим, как каждое из этих параметров будет влиять на внешний вид и поведение вашего оружия:

- `fireMode` используется для определения того, могут ли игроки стрелять непрерывно, удерживая зажатой кнопку мыши, или выполняют по одному выстрелу за клик;
- значение `damage` вычитается из здоровья игрока, по которому вы попали;
- значение `headshotMultiplier` умножается `damage` при попадании в голову, то есть, по сути, работает как критический выстрел;
- значение `rateOfFire` ограничивает количество выстрелов в минуту. Чтобы определить минимальный период времени между выстрелами, вычислите `60/rateOfFire`;
- значение `range` используется для определения дальности полета снаряда, который вы выпускаете. По умолчанию – 500 единиц;
- значения `rayColor` и `raySize` используются для настройки цвета, ширины и высоты визуального эффекта снаряда;
- наконец, значение `debrisTime` задает время видимости визуального эффекта снаряда.

Теперь, когда мы заложили основу для модуля `Weapons` на сервере и создали модуль `Settings` внутри `Tool` и `StarterPack`, добавим новый `LocalScript` с именем `ToolHandler`. Поскольку было бы полезно иметь возможность использовать систему оружия в разных играх, мы сделаем ее полностью независимой по отношению к коду. Вам не нужно будет разбивать ее на модули, будем работать непосредственно в сценарии.

Добавьте в вашем сценарии следующий код, в котором определены необходимые сервисы и экземпляры (все эти вещи вам уже знакомы). Помимо стандартного набора, мы также создадим две функции обработки событий и переменную, которая отслеживает, используется ли `Tool` в данный момент.

Наконец, обратим внимание на переменные `firePoint` и `gunSettings`. Переменная `gunSettings` содержит таблицу модуля `Settings` внутри `Tool`. Переменная `firePoint` содержит экземпляр, из которого выпускается снаряд. Обычно это `Handle`, так как модели для оружия у вас, скорее всего, нет. Добавьте сюда же любые частицы, которые должны воспроизводиться во время выстрела (для этого мы позже добавим функцию `gunEffects()`). Код модуля:

```
local playerService = game:GetService("Players")
local replicatedStorage = game:GetService("ReplicatedStorage")
local replicateRemote = replicatedStorage.Replicate
local hitRemote = replicatedStorage.Hit
local player = playerService.LocalPlayer
local char = player.Character or player.CharacterAdded:Wait()
```

```
local mouse = player:GetMouse()

local tool = script.Parent
local firePoint = tool:WaitForChild("Handle")
-- откуда летит пуля
local gunSettings = require(tool:WaitForChild("Settings"))
local equipped = false

tool.Equipped:Connect(function()
    equipped = true
end)

tool.Unequipped:Connect(function()
    equipped = false
end)
```

После реализации показанного кода добавьте в рабочую область новую папку с именем `Effects`. *Это важно для обнаружения попаданий вашего оружия, о чем мы поговорим далее в этом разделе.*

> **Важное примечание**
>
> Часто можно добавить оружию красоты с помощью анимации, когда оно находится в руках. В Roblox этот процесс очень прост, а вы можете узнать больше о создании и использовании анимации в следующей статье: **https://developer.roblox.com/en-us/articles/using-animations-in-games**.

Первая функция, которую мы добавим в сценарий `ToolHandler`, называется `castRay()`. Она будет создавать новый `raycast`, возвращая информацию о том, в какой экземпляр игрок попал, где он находится, вектор направления `raycast` и положение его источника. Для начала нужно понять, что вообще такое `raycast` и зачем он нам нужен. `Raycast` представляет собой луч, движущийся от точки начала до столкновения с объектом или пока не достигнет максимальной длины. `Raycast` одномерный, то есть имеет длину, но не имеет ширины или высоты, поэтому при обнаружении попаданий он всегда будет вести себя как лазер, а не как сеть.

Чтобы создать новый `raycast`, нам нужны координаты точки его начала `Vector3`, а также вектор направления. Помните, что получить вектор направления можно путем вычитания двух векторов положения. Например, если вам нужен вектор из точки 1 в точку 2, нужно произвести вычитание координат P1 – P2. После вычитания вам будет нужно свойство `Unit` вновь полученного вектора. Вы получите **единичный вектор**, то есть вектор длиной 1. Вникать в тонкости вам не нужно, но именно этот вектор дол-

жен умножаться на скалярное значение **range**. Имея точку начала и единичный вектор направления, вы можете создать новый Ray (луч) с помощью конструктора Ray.new(), который принимает в качестве аргументов точку начала и направление. Результат нужно будет присвоить отдельной переменной.

Пока мы объяснили лишь первые четыре строки функции castRay(), но все остальное гораздо проще. После определения переменной ray мы используем в рабочей области метод под названием FindPartOnRayWithIgnoreList(). Он принимает четыре аргумента: сам луч, таблицу ignoreList, правила поведения при попадании в ландшафт и флаг игнорирования пересечений с местностью. Название метода происходит из того, что второй аргумент под названием ignoreList содержит таблицу экземпляров, которые, включая всех своих потомков, игнорируются raycast. Очевидно, что вы не можете стрелять, например, в союзников или в визуализаторы снарядов других игроков. Чтобы такие попадания не случались, мы добавляем персонажа клиента и папку Effects в таблицу ignoreList.

При попадании в экземпляр в рабочей области функция FindPartOnRayWithIgnoreList() возвращает четыре значения: экземпляр или ячейку ландшафта, в которую попал выстрел; позицию, в которую он попал; вектор нормали к поверхности в точке попадания и материал BasePart точки попадания. Последние два элемента используются нечасто, а вот первые два будут нам нужны. Важно помнить, что если попадания не произошло, все значения будут просто нулевыми, за исключением точки попадания, которая будет обозначать конец raycast.

После получения первых двух значений, возвращенных raycast, запускается событие RemoteEvent в ReplicatedStorage. Мы передаем ему экземпляр Tool, точку начала луча и точку пересечения. В разделе *«Локальная репликация»* мы рассмотрим, как это событие реплицирует визуализатор снаряда. На данный момент мы создаем визуализатор снаряда, который отображается только локально. Это просто обычный объект Part, у которого свойство anchored равно true, с ним нельзя столкнуться, у него неоновый материал, а его цвет, ширина и высота берутся из модуля Settings. Длину или глубину визуализатора можно вычислить как расстояние между точкой начала и точкой пересечения. Помните, что расстояние между двумя векторами положения вычисляется путем их вычитания и вычисления длины результирующего вектора. Зная длину визуализатора, мы располагаем его посередине между точкой начала и точкой пересечения.

Когда все свойства Part заданы правильно, мы наконец создаем в рабочей области объект в папке Effects. Поскольку этот объект будет временным и создается лишь на краткий миг, мы будем использовать сервис Debris. Этот сервис чаще всего применяется через метод AddItem(), который принимает экземпляр и период времени, по истечении которого переданный экземпляр навсегда удаляется. Из кода видно, что визуализатор находится в папке в течение времени debrisTime, которое хранится в модуле Settings,

после чего удаляется. По умолчанию нужно задать достаточное количество времени, чтобы игрок мог успеть увидеть что-то вроде снаряда. После этого информация, которую мы перечислили ранее, возвращается туда, откуда была вызвана функция. Реализуйте приведенную ниже функцию в ваш сценарий `ToolHandler` без изменений:

```
local ignoreList = {char, workspace.Effects}
local debris = game:GetService("Debris")

local function castRay()
    local origin = firePoint.Position
    local direction = (mouse.Hit.p - firePoint.Position).Unit
    direction = direction * gunSettings.range

    local ray = Ray.new(origin, direction)
    local hit, pos = workspace:FindPartOnRayWithIgnoreList(ray, ignoreList)
    replicatedStorage.Replicate:FireServer(tool, origin, pos)
    local visual = Instance.new("Part")
    local length = (pos - origin).Magnitude
    visual.Anchored = true
    visual.CanCollide = false
    visual.Material = Enum.Material.Neon
    visual.Color = gunSettings.rayColor
    visual.Size = Vector3.new(gunSettings.raySize.X, gunSettings.raySize.Y,
                              length)
    visual.CFrame = CFrame.new(origin, pos) * CFrame.new(0,0,-length/2)
    visual.Parent = workspace.Effects
    debris:AddItem(visual, gunSettings.debrisTime)

    return hit, pos, direction, origin
end
```

Последняя вспомогательная функция, которую вы должны реализовать в сценарии `ToolHandler`, называется `gunEffects()`. Она перебирает все экземпляры, являющиеся родительскими для `firePoint` вашего `Tool`. Он проверяет наличие экземпляров `ParticleEmitter` и `Sound`, вызывая при необходимости методы `Emit()` или `Play()`. Из-за поведения, вызванного опцией `FilteringEnabled`, вам нужно будет убедиться, что свойство `RespectFilteringEnabled` сервиса `SoundService` равно `false`. Если это свойство будет истинным, звуки на клиенте воспроизводиться не будут. Вы можете по своему усмотрению изменить следующий код после реализации:

```
local function gunEffects()
    for _, effect in pairs(firePoint:GetChildren()) do
```

```
            if effect:IsA("ParticleEmitter") then
                effect:Emit(50)
            end

            if effect:IsA("Sound") then
                effect:Play()
            end
        end
    end
```

Теперь, когда мы реализовали все вспомогательные функции в `ToolHandler`, нужно обработать событие нажатия со стороны игрока. Однако перед этим добавим в `Tool` новое логическое значение под названием `Debounce`. Оно будет использоваться для устранения дребезга и видно как серверу, так и клиенту.

Используя события `Button1Down` и `Button1Up` экземпляра `Mouse`, мы можем задать поведение при нажатии и отпускании кнопки. Мы создадим новую функцию с именем `fire()`, которая будет вызывать вспомогательные функции и действия, чтобы оружие стреляло. В зависимости от настроек `fireMode` эта функция будет вызываться в цикле с завершающей переменной, которая управляется состоянием экземпляра `Mouse`. Мы внедрим в функцию условный оператор, чтобы `Tool` можно было взять в руки, а значение `Debounce` внутри `Tool` должно быть равно `false`, обозначая готовность оружия к выстрелу.

Если эти два условия выполнены, мы задаем `Debounce` равным `true` и вызываем функцию `delay`, которая ждет в течение времени `60/rateOfFire` и снова возвращает значение `false` для `Debounce`.

Затем мы вызываем `gunEffects()` и `castRay()`, сохраняя информацию, которую возвращает последняя функция. Потом проверяем наличие попадания. Если оно есть, мы находим относительный `CFrame` между объектом, в который был совершен выстрел, и точкой пересечения между этим объектом и `raycast`.

Относительный `CFrame`, по сути, дает информацию о положении одного объекта относительно другого. Например, если в цилиндр диаметром 2 осуществляется попадание так, что точка пересечения между `raycast` и цилиндром находился на передней поверхности цилиндра с совпадающими значениями по осям `x` и `y`, тогда относительный `CFrame` между ними будет равен половине его диаметра в отрицательном направлении оси `z`. Относительный `CFrame` находится путем умножения в `CFrame BasePart` на обратный `CFrame` той `BasePart`, которая становится точкой отсчета. Обратный `CFrame` вычисляется методом `Inverse()`. Этот сценарий вычисления относительного `CFrame` показан на рис. 6.1, где мы умножаем в `CFrame` точки пересечения на обратный `CFrame` цилиндра.

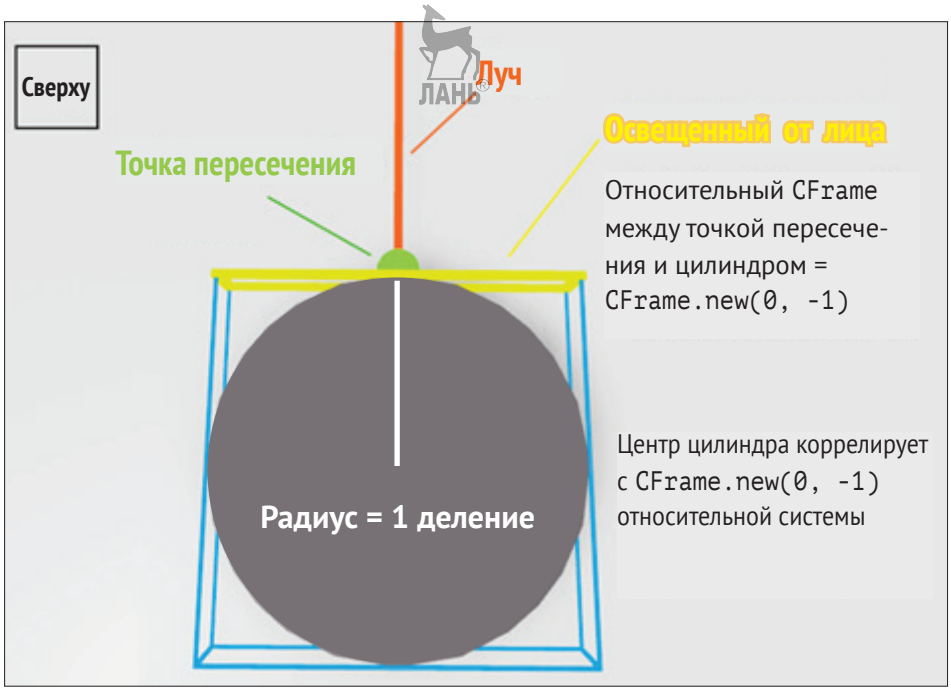


Рис. 6.1. Относительный `CFrame` между точкой пересечения и цилиндром

Когда это значение вычислено, запускается `RemoteFunction` под названием `Hit` в `ReplicatedStorage`. Ей передается `Tool`, цель попадания, направление `raycast`, точка начала `raycast`, относительный `CFrame` точки пересечения и цели попадания, а также точка пересечения. Все они будут использоваться для проверки безопасности на сервере. Вы должны внедрить следующий код в свой сценарий `ToolHandler` без изменений:

```
local doFire = false

local function fire()
    local waitTime = 60/gunSettings.rateOfFire

    repeat
        if equipped and not tool.Debounce.Value then
            tool.Debounce.Value = true

            delay(waitTime, function()
                tool.Debounce.Value = false
            end)

            gunEffects()
            local hit, pos, direction, origin =
```

```
                castRay()

            if hit then
                local relCFrame = hit.CFrame:Inverse()
                    * CFrame.new(pos)
                hitRemote:FireServer(tool, hit,
                    direction, origin, relCFrame)
            end
        end
        wait(waitTime)

    until not equipped or not doFire or
        gunSettings.fireMode ~= "AUTO"
end

mouse.Button1Down:Connect(function()
    doFire = true

    if char.Humanoid.Health > 0 then
        fire()
    end
end)

mouse.Button1Up:Connect(function()
    doFire = false
end)
```

Теперь, когда событие `RemoteEvent` с именем `Hit` запущено, нам необходимо определить поведение для него на сервере. Вернемся в модуль `Weapons` и реализуем новую вспомогательную функцию, называемую `verifyHit()`, которая будет получать цель попадания, направление `raycase` с точки зрения клиента, точку начала `raycast`, относительный `CFrame` и настройки оружия. Сперва применим относительный `CFrame` туда, где по мнению сервера находится цель попадания. Это мера защиты, которая позволяет определить, не запускает ли клиент ложное `RemoteEvent` с неверной информацией о местоположении цели. Как только новый `CFrame` будет найден, мы берем позиционную составляющую `CFrame` и создаем новый вектор направления на точку пересечения и `raycase` с точки зрения сервера. Поскольку величина вектора направления, видимого сервером, не изменилась, взятие длины этого вектора даст нам расстояние между началом луча и точкой пересечения. Если это расстояние больше, чем значение `range` оружия из модуля `Settings`, то пользователь, видимо, пытается сжульничать. В этом случае мы просто выходим из функции.

Далее проверяем, не равна ли длина любого из этих направленных векторов нулю. Шансы на это крайне малы, но если вдруг так случится, то сле-

дующую проверку безопасности выполнить уже не получится, и мы снова вернем нулевое значение. Затем мы находим угол между двумя векторами, используя скалярное произведение векторов и разделив его на произведение длин двух векторов. Получится косинус угла между двумя векторами. Из-за ошибок с плавающей точкой, о которых мы говорили в главе 3, полученное значение может оказаться чуть больше 1 или чуть меньше –1, если угол очень близок к 0 или π радиан. Это проблема, поскольку область определения функции арккосинуса, которую мы должны использовать для получения угла между двумя векторами, лежит в диапазоне от –1 до 1. Чтобы решить эту проблему, введем условный оператор, который выдает угол 0 радиан, если косинус больше 1, π радиан, если косинус меньше –1. В противном случае мы вычисляем арккосинус, поскольку он находится в области определения функции. Наконец, мы преобразуем угол из радианов в градусы, получив, наконец, угол между двумя векторами в градусах.

Нам нужно получить конус возможных расхождений между тем, что видит клиент, и тем, что видит сервер, чтобы скомпенсировать случайные факторы и задержку. Для этого мы создадим в модуле новую константу с именем `SECURITY_ANGLE`, который я задал равным 15°. На рис. 6.2 приведен общий вид такого конуса. Вектор направления сервера и вектор направления клиента могут быть любым вектором, лежащим внутри конуса (рис. 6.2).

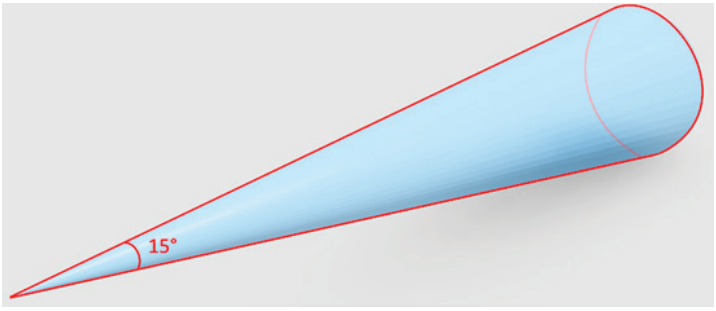


Рис. 6.2. На этом рисунке показан конус безопасности и бесконечное количество векторов в нем

Зная угол, мы используем условный оператор, чтобы убедиться, что угол между двумя векторами направления меньше или равен максимально допустимому отклонению. Таким образом, если вектор направления сервера не более чем на 15° отклоняется от вектора клиента, мы фиксируем попадание как валидное и возвращаем значение `true` в точку вызова. Вставьте этот код в модуль `Weapons` без изменений:

```
local SECURITY_ANGLE = 15

local function verifyHit(hit, direction, origin, relCFrame,
    gunSettings)
```

```
    local target = (hit.CFrame * relCFrame).p
    local serverDirection = target - origin

    if serverDirection.Magnitude > gunSettings.range then
        return end

    if serverDirection.Magnitude == 0 or
        direction.Magnitude == 0 then return end

        local combinedVectors =
            serverDirection:Dot(direction)
        local angle = combinedVectors/(direction.Magnitude
        *serverDirection.Magnitude)

    if angle > 1 then
        angle = 0
    elseif angle < -1 then
        angle = math.pi
    else
        angle = math.acos(angle)
    end
    angle = math.deg(angle)

    if angle <= SECURITY_ANGLE then
        return true
    end
end
```

Когда вспомогательная функция `verifyHit()` готова, мы можем создать функцию события `Hit`. Первым делом событие вызывает функцию `playerFromHit()`, передав ей цель попадания. Обратите внимание, что в идущем далее условии мы проверяем лишь то, что `char` существует, что всегда будет истинно, если `BasePart` является потомком модели, даже если с ней не связан игрок. Так и задумано, потому что мы проверяем, что `char` существует, а затем проверяем наличие в нем `Humanoid`, что позволяет наносить урон не только игрокам, а еще и неигровым персонажам (`NPC`). Дополнительно мы проверяем, что значение `Debounce` внутри `Tool` не находится на перезарядке, чтобы злоумышленники не могли слишком часто отправлять `RemoteEvent` и наносить урон. Если эти условия соблюдены, нам будет нужен модуль `Settings` переданного оружия, затем мы вызываем функцию `verifyHit()`, которая возвращает `true`, если все условия соблюдены.

Предполагая, что функция возвращает значение, значение `Debounce` становится равно `true`, а функция `delay` задерживает следующий выстрел на время `60/rateOfFire`. Наконец, мы определяем экземпляр `Humanoid` цели и

проверяем его здоровье. Если оно уже меньше или равно 0, делать ничего не нужно. Если цель все еще жива, то мы создаем новую переменную для хранения количества урона, которое следует вычесть из ее здоровья.

Если объект, в который совершено попадание, называется `Head`, то произошел выстрел в голову, и мы умножаем значение `damage` на `headshotMultiplier` и кладем результат в переменную функции `Hit`. Затем этот урон вычитается из здоровья цели. Если здоровье цели стало меньше или равно 0, то игрок убил цель, и нужно увеличить его статистику `Kills`. Добавьте следующую функцию в свой модуль без изменений:

```
hitRemote.OnServerEvent:Connect(function(player, weapon,
  hit, direction, origin, relCFrame)
    local otherPlayer, char = weapons.playerFromHit(hit)

    if char and char:FindFirstChildOfClass("Humanoid") and
        not weapon.Debounce.Value then
        local gunSettings = require(weapon.Settings)

        if verifyHit(hit, direction, origin, relCFrame,
            gunSettings) then
            weapon.Debounce.Value = true
            local waitTime = 60/gunSettings.rateOfFire
            delay(waitTime, function()
                weapon.Debounce.Value = false
            end)

            local hum = char:FindFirstChildOfClass("Humanoid")

            if hum.Health > 0 then
                local damage = gunSettings.damage
                if hit.Name == "Head" then
                    damage = damage *
                            gunSettings.headshotMultiplier
                end


                hum.Health = hum.Health - damage

                if hum.Health <= 0 then
                    dataMod.increment(player, "Kills", 1)
                end
            end
        end
    end
end)
```

В этом разделе мы многое узнали о безопасности и применении надежных методов обнаружения попаданий. В следующем разделе вы познакомитесь с концепцией локальной репликации в оружейной системе и запрограммируете поведение события `Replicate`.

Локальная репликация

Локальная репликация – это процесс отправки клиентом сигнала на сервер, который, в свою очередь, отправляет сигнал другим клиентам для воспроизведения событий локально. Например, если клиент локально сгенерировал визуализатор снаряда, никто, кроме этого игрока, не сможет его увидеть. Если бы визуализатор был на сервере, он бы плохо работал из-за низкой частоты обновления сервера, и визуальные эффекты получились бы дерганными. Если клиент, желающий воспроизвести эффект, отправляет необходимую информацию на сервер, чтобы тот передал все нужное клиентам, вы сможете воспроизводить визуальные эффекты практически без потерь.

Чтобы реализовать это, нам нужно будет создать новую функцию события `OnServerEvent` для события `Replicate` в `ReplicatedStorage`. Оно должно было быть еще на этапе создания модуля `Weapons`. Параметрами этой функции события являются игрок, который выстрелил из пульта, само оружие, которое он использовал, точка начала `raycast` и точка пересечения `raycast`. Зная эти значения, сервер вычислит длину и `CFrame` визуализатора, а также обратится к модулю `Settings` оружия, после чего отправит эту информацию всем клиентам через `RemoteEvent`-метод `FireAllClients()`. Код приведен ниже:

```
replicateRemote.OnServerEvent:Connect(function(player,
  weapon, origin, target)
    local length = (target - origin).Magnitude
    local visualCFrame = CFrame.new(origin, target) *
        CFrame.new(0,0,-length/2)
    local gunSettings = require(weapon.Settings)

    replicatedStorage.Replicate:FireAllClients(player,
        gunSettings, visualCFrame, length)
end)
```

Когда сигнал отправлен, мы должны позволить каждому клиенту получить его и соответствующим образом обработать логику локальной репликации. Для этого добавим под `StarterPlayerScripts` новый `LocalScript` под названием `LocalHandler`. Поскольку клиентов будет несколько, этот обработчик должен быть модульным. Ниже приведен его код, который также был включен в сценарий `ServerHandler`:

```
for _, module in pairs(script:GetChildren()) do
    local loadMod = coroutine.create(function()
```

```
        require(module)
    end)

    coroutine.resume(loadMod)
end
```

Сценарий `LocalHandler` создан, теперь нам нужен будет родительский модуль под названием `Replication`. В следующем коде видно, что мы определяем необходимые службы и ссылки, а также `RemoteEvent` под названием `Replicate`. Этот модуль занимается только репликацией, получая сигнал от **`RemoteEvent`**. С помощью события `OnClientEvent` в удаленном событии `Replicate` мы проверяем, что игрок, который использовал свое оружие, – это не тот игрок, у которого выполняется репликация, чтобы не появилось два снаряда. Если это условие выполнено, мы можем создать визуализатор с известной информацией о `CFrame` и его длине из модуля `Settings` оружия, задавая тем самым правильную ширину, высоту, цвет и время, в течение которого визуализатор виден. Код для этого показан здесь:

```
local playerService = game:GetService("Players")
local replicatedStorage = game:GetService("ReplicatedStorage")
local player = playerService.LocalPlayer
local replication = {}
local replicateRemote = replicatedStorage.Replicate

replicateRemote.OnClientEvent:Connect(function(otherPlayer,
  gunSettings, cframe, length)
    if otherPlayer ~= player then
        local visual = Instance.new("Part")
        visual.Anchored = true
        visual.CanCollide = false
        visual.Material = Enum.Material.Neon
        visual.Color = gunSettings.rayColor
        visual.Size = Vector3.new(gunSettings.raySize.X,
            gunSettings.raySize.Y, length)
        visual.CFrame = cframe
        visual.Parent = workspace.Effects
        game.Debris:AddItem(visual,
            gunSettings.debrisTime)
    end
end)

return replication
```

В следующем разделе вам предстоит добавить оружию модели. Если вы хотите использовать более уникальные модели, то вместо использования

готовых моделей из Toolbox попробуйте изучить Blender или загрузите пакет ассетов оружия для своей игры.

Все компоненты модели оружия должны быть приварены Handle, где игрок держится за Tool. Если вам не удается сварить свое оружие вручную, с помощью собственного сценария или панели команд, попробуйте использовать сценарий qPerfectionWeld от Quenty из Toolbox. Этот сценарий автоматически сваривает все компоненты внутри модели, если вложен в нее. Вам следует создать в ReplicatedStorage новую папку под названием Weapons и в дальнейшем помещать все оружие в эту папку. Помните, что одно из оружий в папке должно быть назначено оружием по умолчанию, с которым игроки появляются в вашем игровом цикле. Оружие по умолчанию должно быть не очень мощным, например это может быть пистолет с невысокой скорострельностью.

Теперь система оружия создана, и мы переходим к следующему разделу, в котором узнаем, как создавать точки спауна, на которых игроки смогут находить случайные виды оружия.

Спаун лута

В большинстве игр в жанре «Королевская битва» игроки начинают с простеньким стартовым оружием, а затем бегают по карте в поисках лучшего оружия. В этом разделе мы создадим систему, которая порождает на карте оружие, чтобы игроки могли найти и использовать его. Имейте в виду, что для этой системы у вас должно быть несколько видов оружия, так как большее разнообразие лучше стимулирует игроков к активному поиску.

Для начала добавим в сценарий ServerHandler новый модуль под названием Loot. Добавьте в этот модуль код, определяющий сервисы и другие ссылки, которые нам понадобятся, а также делает объекты точек спауна скрытыми, закрепленными и прозрачными для столкновения. Вы видите, что нам также нужна папка LootSpawns, где будут содержаться точки спауна. По умолчанию модели оружия будут появляться на одну единицу выше положения этой точки, и вам следует помнить об этом при размещении. Следующий код можно добавить в модуль Loot без изменений:

```
local replicatedStorage = game:GetService("ReplicatedStorage")
local lootSpawns = workspace.LootSpawns
local weaponFolder = replicatedStorage.Weapons
local random = Random.new()
local weapons = require(script.Parent.Weapons)
local loot = {}

for _, spawnPoint in pairs(lootSpawns:GetChildren()) do
    spawnPoint.Anchored = true
    spawnPoint.CanCollide = false
```

```
        spawnPoint.Transparency = 1
    end

    return loot
```

После настройки модуля добавим в него новую функцию, которая создает одно случайное оружие в каждой точке спауна. Сперва введем вспомогательную функцию с названием `makeWeaponModel()`. Эта функция берет экземпляр `Tool` и извлекает из него экземпляры `BasePart`, помещая их в новую `Model`. Сохранять сценарии или какие-либо другие типы экземпляров не требуется, поэтому все, что осталось в `Tool`, уничтожается. Эта модель будет визуально отображаться игрокам на карте. Теперь вы должны добавить эту вспомогательную функцию в модуль `Loot`:

```
local function makeWeaponModel(weapon)
    local weaponModel = Instance.new("Model")

    for _, child in pairs(weapon:GetDescendants()) do
        if child:IsA("BasePart") then
            child.Parent = weaponModel
            child.Anchored = true
            child:ClearAllChildren()
        end
    end

    weapon:Destroy()

    return weaponModel
end
```

Сейчас перейдем к реализации основного поведения модуля, добавив новую функцию с именем `spawnWeapons()`. Эта функция будет с помощью цикла `for` перебирать все `Parts` папки `LootSpawns` и добавлять на карту оружие, которое игроки будут подбирать. Мы будем привязывать модель оружия к точке спауна, поэтому сначала нужно проверить, не занята ли точка другим оружием, и удалить его, если оно уже есть. Далее мы создаем пул оружия и присваиваем таблицу переменной `weaponPool`. Он может иметь любой вид, вплоть до того, что вы можете назначить конкретным точкам конкретные виды оружия, но для этого примера мы просто возьмем всю папку `Weapons Folder` из `ReplicatedStorage`.

Создав таблицу оружия, мы генерируем случайный индекс и берем копию оружия с этим индексом из таблицы. Далее вызываем функцию `makeWeaponModel()` для создания модели, которую увидят игроки. Создав модель, мы создаем новый объект `Part`, который станет `PrimaryPart` модели.

Это сделано для того, чтобы мы могли изменить положение Model целиком с помощью метода SetPrimaryPartCFrame(), а также использовать Part в виде хитбокса. Чтобы объект можно было использовать в качестве хитбокса, мы используем метод Model GetBoundingBox(), который возвращает описание CFrame модели, а также наименьший ограничивающий прямоугольник, содержащий модель.

Наконец, создадим новую функцию события Touched, которая проверяет, что игрок коснулся PrimaryPart. Tool, связанный с моделью, отправляется в рюкзак игрока, а модель оружия уничтожается, чтобы его нельзя было подобрать дважды. Добавьте следующий код в модуль Loot, ничего не меняя, если не уверены в том, что делаете:

```
loot.spawnWeapons = function()
    for _, spawnPoint in pairs(lootSpawns:GetChildren())
       do
        local oldModel = spawnPoint:FindFirstChildOfClass("Model")
        if oldModel then
            oldModel:Destroy()
        end

        local weaponPool = weaponFolder:GetChildren()
        local randomIndex = random:NextInteger(1,
            #weaponPool)
        local weapon = weaponPool[randomIndex]:Clone()
        local weaponName = weapon.Name
        local weaponModel = makeWeaponModel(weapon)
        weaponModel.Parent = spawnPoint

        local primaryPart = Instance.new("Part")
        primaryPart.Anchored = true
        primaryPart.CanCollide = false
        primaryPart.Transparency = 1
        primaryPart.CFrame, primaryPart.Size =
            weaponModel:GetBoundingBox()
        primaryPart.Parent = weaponModel

        weaponModel.PrimaryPart = primaryPart
        local newCFrame = CFrame.new(spawnPoint.CFrame.p)
            * CFrame.new(0,1,0)
        weaponModel:SetPrimaryPartCFrame(newCFrame)

        primaryPart.Touched:Connect(function(hit)
            local player, char = weapons.playerFromHit(hit)
```

```
            if player and char then
        local tool =
            weaponFolder:FindFirstChild(weaponName):
             Clone()
                tool.Parent = player.Backpack
                char.Humanoid:EquipTool(tool)
                weaponModel:Destroy()
            end
        end)
    end
end
```

С функциями покончено, теперь обязательно нужно в модуле `GameRunner` определить модуль `Loot` и добавить строку `lootMod.spawnWeapons()` в коде начала раунда, после добавления игроков в таблицу `competitors`. В этом случае новое оружие будет появляться в начале каждого раунда.

Если вы хотите добавить красочности в процесс поиска оружия, то можете добавить к отображаемой модели оружия звуки или частицы, чтобы придать модели свечение либо блеск. Возможно, вы захотите ввести систему редкости. Если у каждого оружия будет своя редкость, вы сможете задать вероятность его появления. Такие системы вводить необязательно, но они помогают воодушевлять игроков, когда те находят нечто редкое.

Настройка внешнего интерфейса

На данный момент серверная часть игры полностью завершена, теперь перейдем к работе над фронтендом, с которым игрокам предстоит взаимодействовать напрямую. В первую очередь это будет работа с пользовательским интерфейсом.

Работа с пользовательским интерфейсом

Мы пока что вообще не делали и не использовали каких-либо интерфейсов. Как часть этой работы мы начали работу с системой `leaderstats`, которая полностью автоматизирована. В этом разделе мы создадим простой пользовательский интерфейс, используя встроенные объекты Roblox, ознакомимся со свойствами, научимся выводить важную информацию для игроков и введем другие интересные функции.

Сообщение от игры и отображение оставшихся игроков

Начнем создание пользовательского интерфейса для игры с того, что найдем сервис `StarterGui` на панели **Explorer** (Проводник). Затем вы должны вложить в него новый экземпляр `ScreenGui` и назвать его `Main`. Обычно свойства экземпляра `ScreenGui` не приходится менять. Эти экземпляры служат контейнерами для пользовательского интерфейса, который отобра-

жается игрокам и с которым они взаимодействуют на своем экране, при этом сам экземпляр никак не отображается. Теперь добавим экземпляр `TextLabel` и назовем его `Message`. На этой метке будет выводиться текст, который сервер будет присваивать строковой переменной `Message` в `ReplicatedStorage`.

После этого добавьте еще один `TextLabel` с именем `Remaining`. Эта метка будет работать так же, как и предыдущая, но значение брать из переменной `Remaining` в `ReplicatedStorage`. Когда вы закончите, сервис `StarterGui` на панели **Explorer** (Проводник) будет выглядеть как на рис. 6.3.

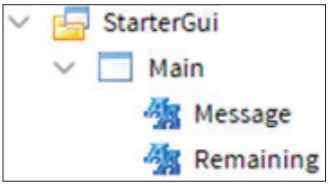


Рис. 6.3. На рисунке показано, как экземпляры `TextLabel` должны быть связаны с экземпляром `ScreenGui`

После добавления объектов в `ScreenGui` нам нужно будет изменить их свойства, чтобы они правильно отображались на экранах игроков. Рассмотрим свойства экземпляров `TextLabel`, которые нам нужно будет изменить.

- Самым важным для `TextLabel` является свойство `Text`. Это строка, которая в этой игре по умолчанию у обеих меток должна быть пустой строкой. Кроме того, следует включить свойство `TextScaled` метки, которое автоматически масштабирует размер шрифта так, чтобы он заполнял все поле. Это свойство расположено в нижней части списка свойств.
- Свойство `ZIndex` используется для определения порядка отображения элементов пользовательского интерфейса. Например, `TextLabel` со значением `ZIndex`, равным `2`, располагается поверх метки с `ZIndex`, равным `1`.
- Свойства `BackgroundColor3` и `BackgroundTransparensy` позволяют задавать настройки фона объектов пользовательского интерфейса, определяя значение типа `Color3` и число соответственно. Мы, в частности, сделаем фон метки `Message` полупрозрачным и серым. Кроме того, поскольку мы будем работать с серым фоном, нам нужно установить `TextColor3` более светлым, ближе к белому.
- Свойства `Size` и `Position` характеристики используются для определения внешнего вида экземпляров пользовательского интерфейса. Эти свойства принимают пользовательский тип данных `UDim2`, который, как и большинство пользовательских типов данных, строится с помощью конструктора `.new()`. `UDim2` состоит из четырех элементов, у которых задается два типа `size` под названием `scale` и `offset` (масштаб и смещение). В этой игре мы будем использовать только масштаб, а это означает, что если мы захотим создать значение `Size`, которое охватывает весь экран, нужно будет вызвать `UDim2.new(1, 0, 1, 0)`.
- Свойство `AnchorPoint` имеется у многих экземпляров пользовательского интерфейса. Это свойство имеет тип `Vector2` и задает точку

привязки элемента интерфейса, а также то, как элемент расширяется или сжимается при изменении размера. Чтобы элемент пользовательского интерфейса был привязан к центру, `AnchorPoint` должно иметь значение `Vector2.new(0.5, 0.5)`, а свойство `Position` — `UDim2.new(0.5, 0, 0.5, 0)`.

На рис. 6.4 видно, что нам нужно будет изменять значения всех перечисленных свойств. Нет нужды задавать эти свойства в сценарии, и даже рекомендуется задать их просто в меню **Properties** (Свойства) в Studio. Свойство `AnchorPoint` метки `Message` должно равняться `Vector2.new(0.5, 0.5)`. Метка будет занимать всю ширину экрана и играть роль фона для обеих меток. Для этого свойства `BackgroundColor3` и `BackgroundTransparensy` зададим равными `Color3.fromRGB(86, 86 ,86)` и `0.5` соответственно. Сделав это, мы изменим значение свойства `Size` метки `Message` на `UDim2.new(1, 0, 0.1, 0)`, а значение свойства `Position` на `UDim2.new(0.5, 0, 0.05, 0)`; так мы охватим всю ширину экрана и расположим метку в верхней центральной части экрана. Метка `Message` готова!

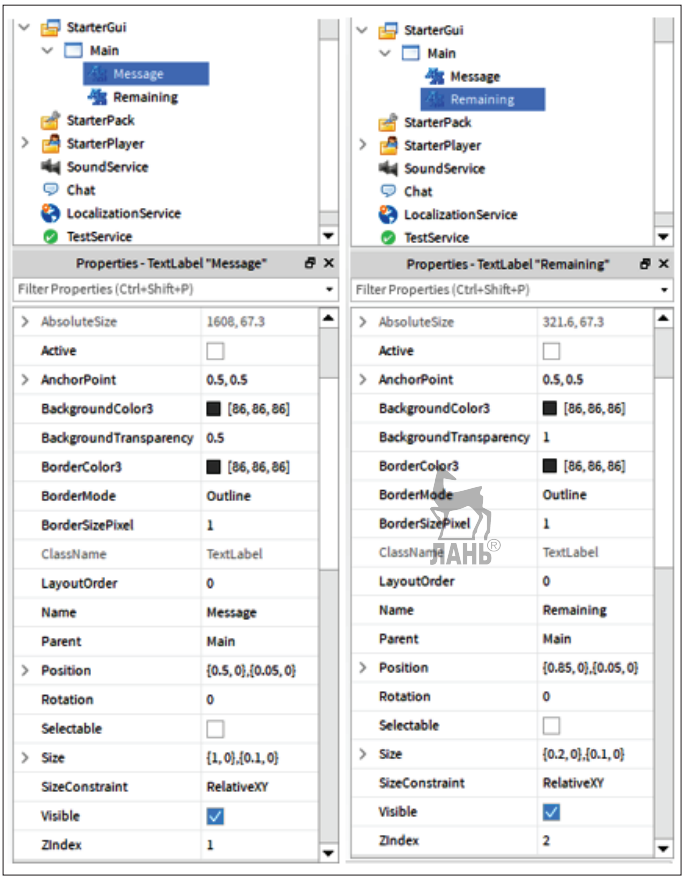


Рис. 6.4. Свойства двух экземпляров `TextLabel` в `ScreenGui`

Теперь займемся меткой Remaining. AnchorPoint у нее будет таким же, как у метки Message. Свойство BackgroundTransparency должно быть равно 1, так как фон берется от метки Message. Затем измените значение ZIndex с 1 на 2, чтобы эта метка отображалась поверх фона. Наконец, нам нужно будет изменить свойства Size и Positiob, чтобы задать красивое смещение от метки Message. Задайте им значения UDim2.new(0.2, 0, 0.1, 0) и UDim2.new(0.85, 0, 0.05, 0) соответственно. Метка Remaining окажется на той же высоте, что и Message, но со смещением к правому краю фона.

В качестве последнего аккорда вы можете установить свойство BorderPixelSize равным 0, чтобы появился видимый контур, но обычно такое не считается особо привлекательным.

Перечисленные свойства показаны на рис. 6.4.

После настройки элементов интерфейса вам нужно будет создать новый LocalScript и добавить его в экземпляр ScreenGui. Назовем этот сценарий UIHandler и, поскольку нам предстоит работать с несколькими системами пользовательского интерфейса, сделаем его модульным.

Добавим в сценарий код, с помощью которого любой модуль можно будет в него вложить:

```
for _, module in pairs(script:GetChildren()) do
    local loadMod = coroutine.create(function()
        require(module)
    end)

    coroutine.resume(loadMod)
end
```

Добавьте в созданный сценарий UIHandler новый модуль с именем Display. В приведенном ниже коде видно, что мы индексируем ReplicatedStorage, ScreenGui, в который вложен UIHandler, а также строковые переменные в ReplicatedStorage. Для краткости мы также зададим псевдонимы для объектов TextLabel, которые будут выводить значения Message и RemainingStrongValue.

Далее с помощью метода экземпляра GetPropertyChangedSignal() мы обновляем свойство Text экземпляра TextLabel новым значением StringValue, когда оно меняется. Мы реализуем эту логику в виде функции события, а не цикла, так как этот вариант более производителен. Таким образом, мы присваиваем значения только при их изменении, а не раз в какое-то время. Добавьте приведенный ниже код в модуль без изменений, если не уверены в изменении на сто процентов:

```
local replicatedStorage = game:GetService("ReplicatedStorage")
local gui = script.Parent.Parent
local message = replicatedStorage.Message
```

```
local remaining = replicatedStorage.Remaining
local display = {}

local messageLabel = gui:WaitForChild("Message")
local remainingLabel = gui:WaitForChild("Remaining")

messageLabel.Text = message.Value
remainingLabel.Text = remaining.Value

message:GetPropertyChangedSignal("Value"):
  Connect(function()
    messageLabel.Text = message.Value
end)

remaining:GetPropertyChangedSignal("Value"):
  Connect(function()
    remainingLabel.Text = remaining.Value
end)

return display
```

Далее мы рассмотрим тему пользовательских интерфейсов, которые позволяют удержать игроков в игре, даже если они постоянно проигрывают.

Создание меню для наблюдателей

Как упоминалось ранее, после смерти и возврата игрока в лобби вам нужно поддержать активность, чтобы он не заскучал и не покинул игру. Хороший вариант – реализовать меню наблюдателя, с помощью которого игроки могли бы просматривать оставшихся участников и переключаться между ними простым нажатием кнопки.

Поскольку нам нужно работать с несколькими элементами пользовательского интерфейса, которые нужно будет одновременно делать видимыми или невидимыми, нам нужно будет реализовать экземпляр `Frame`. Этот экземпляр чаще всего используется в качестве контейнера для других экземпляров пользовательского интерфейса. Нужен он нам потому, что мы можем связать с ним все элементы пользовательского интерфейса, чтобы они появлялись и исчезали одновременно. Это уменьшает объем кода, который нужно будет написать, и улучшает организацию объектов на панели **Explorer** (Проводник). Назовите этот экземпляр `Frame` с именем `Spectate` и разместите его по центру, растяните на весь экран и задайте прозрачный фон. Так вы добьетесь правильного поведения при масштабировании и размещении дочерних элементов. Нужные свойства экземпляра `Frame` показаны на рис. 6.5.

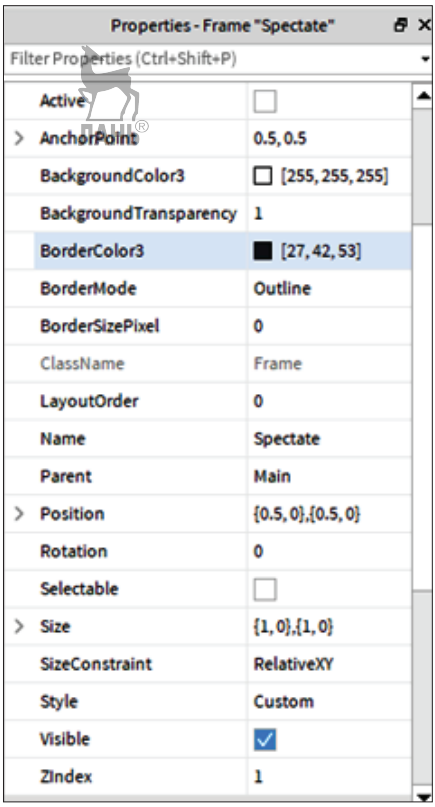


Рис. 6.5. Свойства экземпляра `Frame` с именем `Spectate`

После создания и изменения размера `Frame` вам нужно будет создать три `TextButton` и один `TextLabel`. Кнопки работают аналогично меткам, но умеют определять, когда с ними взаимодействуют через события. Поскольку экземпляр `TextButton` входит в то же семейство объектов, что и `TextLabel`, их свойства почти идентичны. Нам нужна кнопка для переключения в интерфейс **наблюдения**, две кнопки для переключения игроков вперед и назад и одна кнопка для отображения имени игрока, за которым вы сейчас наблюдаете. Этот пользовательский интерфейс эстетически довольно примитивен. Если вы хотите узнать, как украсить свой пользовательский интерфейс, ознакомьтесь со статьями на веб-сайте разработчика:

- **https://developer.roblox.com/en-us/articles/Intro-to-GUIs;**
- **https://developer.roblox.com/en-us/articles/Creating-GUI-Buttons.**

На рис. 6.6 показан настроенный интерфейс. Этот макет неплох по нескольким причинам: он интуитивно понятен для пользователя, на нем четко видны кнопки и их назначение, выводится информация о том, за кем вы наблюдаете. Этот макет хорош, но он слишком прост, а дизайн кнопок примитивен. Создавая собственный проект, вы должны либо сами создать,

либо нанять другого человека для создания красивого пользовательского интерфейса, который будет нравиться игрокам.

Рис. 6.6. Здесь видны экземпляр `Frame` с именем `Spectate`, кнопка переключения и текстовые метки

Для начала нам нужно добавить в сценарий `UIHandler` новый модуль под названием `Spectate`. В приведенном ниже коде мы определяем, какие сервисы будут нужны системе, а также **Frame** и элементы пользовательского интерфейса внутри нее. Кроме того, мы создаем переменные, которые будут использоваться для отслеживания информации при применении системы. Добавьте в модуль следующий код без изменений:

```
local replicatedStorage = game:GetService("ReplicatedStorage")
local playerService = game:GetService("Players")
local player = playerService.LocalPlayer
local cam = workspace.CurrentCamera
local spectate = {}

local gui = script.Parent.Parent
local spectateFrame = gui:WaitForChild("Spectate")
local toggle = gui:WaitForChild("Toggle")
local nameLabel = spectateFrame:WaitForChild("NameLabel")
local nextPlayer = spectateFrame:WaitForChild("NextPlayer")
local lastPlayer = spectateFrame:WaitForChild("LastPlayer")

local competitors = {}
local curIndex = 1
local spectating = false
```

```
spectateFrame.Visible = false

return spectate
```

Перво-наперво система должна понимать, какие игроки еще живы. К счастью, живых игроков найти легко, так как они есть в таблице `competitors` в модуле `GameRunner`. Чтобы передать эту таблицу клиентам с сервера, мы реализуем новую `RemoteFunction`, вложим ее в `ReplicatedStorage` и назовем ее `GetCompetitors`. Кроме того, нам нужно добавить в `ReplicatedStorage` новый `RemoteEvent` с именем `UpdateCompetitors`.

Приведенный ниже код разделен на два блока, названных `client` и `server`. Клиентскую часть кода нужно добавить в модуль `Spectate`, так как она присваивает переменной `competitros` таблицу, возвращенную удаленной функцией, и обновляет таблицу при возникновении `RemoteEvent`. Серверный код идет в модуль `GameRunner` и возвращает таблицу `competitors` туда, откуда произошел вызов. Функция `getCompetitors()` вызывается, когда игрок переключается в интерфейс наблюдения или между оставшимися игроками, обновляя таблицу `competitors`. Мы обработаем эту логику в следующих примерах.

В `RemoteEvent` с именем `UpdateCompetitors` сервер обновляет таблицы `competitors` у всех клиентов в начале игры или после смерти игрока. Для этого обязательно добавьте строку `updateCompetitors:FireAllClients(competitors)` в функции `removePlayerFromTable()` и `gameloop()` модуля `GameRunner`, чтобы она вызывалась на выходе игрока, при появлении игрока или в конце игры, когда таблица очищается. Приведенный ниже код нужно добавить в оба модуля без изменений:

```
--клиент
local getCompetitors = replicatedStorage.GetCompetitors
local updateCompetitors = replicatedStorage.UpdateCompetitors

spectate.getCompetitors = function()
    competitors = getCompetitors:InvokeServer()
end

updateCompetitors.OnClientEvent:Connect(function(list)
    competitors = list
    for _, competitor in pairs(competitors) do
        if competitor == player then
            toggle.Visible = false

            if spectating then
                spectate.toggleSpectate()
            end
            return
```

```
        end
    end

    if spectating then
        spectate.focusCamera(competitors[curIndex])
    end
end)

--сервер
local getCompetitors = replicatedStorage.GetCompetitors
local updateCompetitors = replicatedStorage.UpdateCompetitors

getCompetitors.OnServerInvoke = function()
    return competitors
end
```

В модуле `Spectate` приведенная ниже функция `toggleSpectate()` обрабатывает два случая в зависимости от того, используется ли в данный момент система наблюдения. Если на момент вызова функции система еще не используется, переменная `spectating` становится равной `true`, таблица `competitors` в модуле обновляется, пользовательский интерфейс, связанный с системой, становится видимым, а камера клиента фокусируется на первом игроке в таблице. Если при вызове функции система наблюдения используется, то переменная `spectating` становится равной `false`, интерфейс скрывается, а камера клиента переключается на него самого. Добавьте приведенный ниже код в модуль `Spectate` без изменений:

```
spectate.toggleSpectate = function()
    if not spectating then
        spectating = true
        spectate.getCompetitors()
        spectateFrame.Visible = true
        local targetPlayer = competitors[1]
        spectate.focusCamera(targetPlayer)
    else
        spectating = false
        spectateFrame.Visible = false
        spectate.focusCamera(player)
    end
end
```

Функция `focusCamera()` используется для фокусировки камеры клиента на персонаже переданного ей игрока. Эта функция также проверяет, равно ли количество оставшихся игроков 0, и отключает меню наблюдения,

так как игра в данный момент не идет. Если игроки еще остались, то камера клиента фокусируется на этом игроке, а свойство `Text` метки `NameLabel` обновляется. Добавьте приведенный ниже код в модуль без каких-либо изменений:

```
spectate.focusCamera = function(targetPlayer)
    if #competitors == 0 and spectating then
        spectate.toggleSpectate()
    else
        if targetPlayer then
            cam.CameraSubject = targetPlayer.Character
            nameLabel.Text = targetPlayer.Name
        else
            spectate.getCompetitors()
            local newTargetPlayer = competitors[1]
            spectate.focusCamera(newTargetPlayer)
        end
    end
end
```

Следующий код наделяет функциональностью добавленные вами кнопки пользовательского интерфейса. С помощью события `MouseButton1Click` экземпляра интерфейса вы можете задавать поведение при нажатии кнопки пользовательского интерфейса. Обратите внимание, что нажатие кнопки `toggle` вызывает функцию `toggleSpectate` модуля. Когда обе кнопки назначены переменным `nextPlayer` и `lastPlayer`, мы обновляем и перебираем таблицу `competitors` и проверяем, нужно ли сбросить переменную `curIndex`, если она вышла за пределы значений. Если индекс правильный, камера игрока фокусируется на игроке с заданным индексом в таблице `competitors`.

Код приведен ниже:

```
toggle.MouseButton1Click:Connect(function()
    spectate.toggleSpectate()
end)

nextPlayer.MouseButton1Click:Connect(function()
    spectate.getCompetitors()
    curIndex = curIndex + 1
    if curIndex > #competitors then
        curIndex = 1
    end

    local targetPlayer = competitors[curIndex]
    spectate.focusCamera(targetPlayer)
```

```
end)

lastPlayer.MouseButton1Click:Connect(function()
    spectate.getCompetitors()
    curIndex = curIndex - 1
    if curIndex < 1 then
        curIndex = #competitors
    end

    local targetPlayer = competitors[curIndex]
    spectate.focusCamera(targetPlayer)
end)
```

Создание магазина

Если вам понадобится магазин для игры, теперь вы можете создать для него пользовательский интерфейс или использовать объекты внутри игры, как мы уже делали в обби-игре. Стоит заметить, что в большинстве жанров игр пользователь с большей вероятностью заметит красивую кнопку на экране, чем объект в игре, который игрок вообще может не найти. По этой причине я рекомендую для покупок за робаксы и внутриигровую валюту делать магазин с помощью пользовательского интерфейса. Помните, что для управления покупками за робаксы достаточно взять и слегка модернизировать модуль `Monetization`, который мы сделали в предыдущей главе.

Резюме

В этой главе мы узнали, как реализовать управление игроками в игре с системой раундов, создали оружие с защитой на стороне клиента, изучили концепцию локальной репликации и создали функциональный и симпатичный пользовательский интерфейс. Теперь с помощью богатого инструментария платформы Roblox вы способны создать любую игру, которую только можно вообразить. Прежде чем приступить к созданию собственных проектов, обязательно пробегитесь еще раз по ключевым понятиям из этих глав и их примерам из игр, включая безопасность, взаимодействие и поддержание чистой и организованной кодовой базы.

В следующей и последней главе этой книги мы рассмотрим методы популяризации и поддержки игр, которые вы будете делать в будущем. В частности, вы научитесь продвигать и развивать свои игры, оптимально монетизировать их и реализовывать функции, позволяющие удержать текущую аудиторию и привлечь новую.

Часть III

Логистика производства игр

Данная часть будет посвящена вопросам логистики и деловым навыкам, необходимым для того, чтобы проект продвигался на протяжении всего жизненного цикла. Мы рассмотрим вопросы бюджетирования, управления временем, эффективности разработки и сотрудничества, которые особенно важны в сфере разработки Roblox.

В этой части будет одна глава:

- *глава 7 «Три “М”».*

Глава 7

Три «М»

В предыдущих главах вы научились программировать на Lua и создавать интересные игры в среде Roblox. Благодаря этим навыкам вы сможете создать игру практически из любой фантазии, которая придет вам на ум. Но создать собственную игру – это лишь полдела. Чтобы добиться успеха, вам необходимо немало сделать для оптимизации эффективности вашей игры.

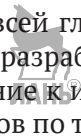
Эффективность игры, как правило, держится на трех китах: Механике, Монетизации и Маркетинге (я называю их «Три “М”»). В этой главе мы познакомимся с лучшими практиками, которые следует соблюдать в отношении каждого из этих аспектов, чтобы игра была максимально приятной для игроков, охватывала максимально большую аудиторию, а доходы росли, но не влияли на пользовательский опыт.

В этой главе мы рассмотрим следующие основные темы:

- механика;
- монетизация;
- маркетинг;
- повторение пройденного.

Технические требования

Эта глава полностью посвящена передовым методам достижения успеха на платформе Roblox. На протяжении всей главы мы будем ссылаться на статьи с сайта разработчика и форума разработчиков (DevForum), поэтому вам будет полезно иметь подключение к интернету для их просмотра, изучения или в целом поиска материалов по теме.



Механика

Первая «М» – это механика. Механика игр в Roblox и на других платформах может сильно различаться. Популярными в Roblox жанрами являются симуляторы, ролевые игры (РП), тайкуны и мини-игры. У всех этих жанров игр есть общие черты в механике – системы многократного вознаграждения, социального взаимодействия или и то, и другое.

В этом разделе мы разберем механики этих часто встречающихся жанров и определим, в чем таится секрет их успеха, чтобы вы могли в будущем интегрировать эти идеи в целые жанры своего собственного творчества.

Симуляторы

В симуляторах игроки начинают игру с простейшими инструментами, оружием или способностями. Со временем, выполняя какие-то простые задачи (вплоть до кликов мышью), игроки получают очки, которые можно конвертировать в валюту для покупки лучших предметов. Это позволяет игрокам быстрее продвигаться вперед в начале игры. Разумеется, возможность быстрого продвижения иллюзорна, поскольку цена на предметы растет в геометрической прогрессии, заставляя игрока много гриндить. Гениальность этого формата заключается в том, что он позволяет новым игрокам довольно быстро стартовать, постоянно получая огромные награды. Когда требования для достижения следующего этапа прогресса растут, игроки стремятся к легкодостижимому успеху, к которому привыкли в начале игры, а это побуждает их продолжать играть.

Взгляните на следующий рисунок.

Рис. 7.1. RussoPlays завершает начальный квест в игре Power Simulator

Несмотря на серьезную способность к захвату и удержанию аудитории, симуляторы не могут жить вечно. Без постоянных обновлений игрокам станет скучно, особенно если они уже приобрели весь доступный контент в игре. Секрет поддержания симулятора в рабочем состоянии заключается в том, что вы должны часто выпускать новый контент в начале, а затем постепенно уменьшать частоту обновлений по мере перехода к другим проектам. С каждым обновлением вы должны добавлять новые разблокируемые предметы и другой бесплатный контент, а также некоторые новые варианты монетизации. В качестве известного примера долгоживущей игры этого

жанра можно привести Bubble Gum Simulator компании Rumble Studios. Она была выпущена в 2018 году и до сих пор поддерживается онлайн десятками тысяч игроков – а все благодаря описанным выше принципам.

RP-игры

RP-игры на платформе Roblox появились давно, но в последнее время в разделе Popular начали доминировать игры с новыми интерпретациями самой концепции жанра. Не следует путать жанр RP с жанром RPG, в котором обычно нужно выполнять квесты, находить предметы и исследовать мир. RP-игры – это игры, в которых пользователи просто отыгрывают какую-то роль, обычно в мирной обстановке. В качестве примера можно привести игры, действие которых происходит в городах или районах, где нужно выполнять шуточные задания или стать членом семьи, например Adopt Me, Meep City, RoCitizens и т. д.

Эти игры особенно успешны, потому что игрокам предлагается множество неповторяющихся действий, а игра сама по себе бесконечна. В таких играх нередко есть что открывать и исследовать, но большая часть игры ориентирована на взаимодействие с другими людьми и улучшение вещей, таких как выращивание домашних животных или украшение дома. Эти игры, как и симуляторы, тоже нужно регулярно обновлять, и в обновлениях должен быть и бесплатный контент, и новые варианты монетизации.

Тайкуны

В последние годы традиционные тайкуны на платформе Roblox стали менее популярны, но они все еще существуют, хоть и в несколько ином формате по сравнению со старыми. Тайкуны – это игры, в которых игроки добавляют различные вещи (обычно машины) на домашнюю базу, чтобы заработать больше валюты. Имея больше валюты, игроки могут покупать больше декоративных и функциональных объектов для своей базы, расширяясь до тех пор, пока не станут доминировать на сервере. Не во всех тайкунах есть механики боя, но почти всегда есть какой-то элемент, который заставляет игроков соревноваться друг с другом, помимо демонстрации того, кто самый богатый игрок на сервере. На рис. 7.2 показан пример типичной игры этого жанра.

Эти игры несколько утратили популярность, потому что после того, как игрок разблокирует все предметы, особого стимула продолжать играть у него уже нет. Обновление таких игр может быть более сложной задачей, в зависимости от того, как она устроена. При обновлении лучше всего добавлять в игру новые предметы, новые возможности карты и новые дополнения к монетизации. Эти игры обычно уходят с пика успеха гораздо быстрее, чем игры других упомянутых жанров.

В современных тайкунах разработчики часто вводят неигровых персонажей (NPC), более сложные способы разблокировки новых предметов, бесконечные возможности расширения, а иногда и потрясающие визуальные

эффекты. Хорошим примером игры этого жанра может быть My Restaurant компании BIG Games.

Рис. 7.2. Пример традиционного тайкуна: Miner's Haven

Мини-игры

Мини-игры, или аркадные игры, основанные на раундах, постоянно пользуются популярностью на платформе, хотя и не имеют такого же успеха, как вышеупомянутые жанры. Аркадные игры легче обновлять, так как для этого достаточно добавлять новые уровни или карты. Кроме того, как и в случае с ролевыми играми, эти игры невероятно социальны, и многие игроки играют с друзьями или находят новых прямо в игре.

Если вы хотите сделать игру на основе раундов, упор должен быть именно на социальный аспект, а монетизация не должна мешать пользователю. Кстати, в следующем разделе вы узнаете, как применять хорошие стратегии монетизации для этого и многих других жанров.

Монетизация

Следующая «М» – это монетизация. Монетизация – один из самых важных аспектов разработки, если вы хотите сделать Roblox своей карьерой или прибыльным хобби. Первое, что вам нужно понять: существуют отличные и ужасные реализации монетизации. Выбранная вами стратегия монетизации может положительно или отрицательно влиять на всех игроков, независимо от того, покупают они что-либо в вашей игре или нет. Начнем с того, чего делать нельзя.

Возможно, самый ужасный случай, о котором часто говорят игроки, – это монетизация по принципу «плати, чтобы выиграть». В таких случаях игроки, у которых есть больше денег, могут покупать очень мощные предметы

или другие вещи, получая уникальное преимущество перед бесплатными игроками. Некоторые игровые компании, такие как Electronic Arts Inc. (EA), печально известны этим стилем монетизации, которой загубили начальную и общую эффективность игр вроде Star Wars: Battlefront II, в которой была система ящиков с добычей. Реализованная таким образом монетизация портит удовольствие от игры всем, кроме игроков, купивших премиум-предметы. Да и то, даже им не доставляет удовольствия играть против других, купивших все то же самое.

Боевые пропуски, дающие преимущества, в какой-то степени приемлемы, но возможности премиальных предметов должны быть где-то в диапазоне от среднего до выше среднего. Это особенно важно соблюдать, если в вашей игре есть какие-либо системы подбора игроков, основанные на умении играть, чтобы новые игроки не могли мгновенно купить себе путь к вершине.

Растеряв значительную часть публики из-за плохих стратегий монетизации, большие и малые компании – разработчики игр сместили акцент в покупках в сторону дополнительного контента или косметических опций. Косметические опции великолепно позволяют избежать ранее упомянутых проблем, поскольку решение о покупке основывается исключительно на личных предпочтениях, а сама покупка не дает никаких преимуществ.

Подход с дополнительным контентом тоже можно хорошо реализовать, особенно в Roblox. Обычно сообщества игроков допускают наличие дополнительного контента, но не до такой степени, чтобы приходилось покупать игру дважды, дабы получить полноценное впечатление. Даже студии уровня AAA в прошлом наступали на эти грабли, что пагубно сказывалось на эффективности игры. Однако игры в Roblox почти всегда бесплатные, за исключением тех немногих, в которых используется система платного доступа, о которой упоминалось еще в главе 2. Поскольку платформа Roblox ориентирована на бесплатные игры, наличие в ней некоторого контента с платным доступом даже более приемлемо, чем где-либо еще. Это удачно применяется в симуляторах, где игрокам дается доступ к области, в которой чуть больше ресурсов и которые чуть лучше оформлены, только при наличии VIP-пропуска. Обе стратегии являются жизнеспособными, вы можете реализовывать их по своему усмотрению и следить за реакцией вашего сообщества.

Что касается варианта монетизации с применением VIP-доступа, иногда бывает более выгодно объединить платные преимущества в подписку, а не пропуск с разовой покупкой. Например, предположим, что VIP-пропуск дает игрокам в вашей игре специальный тег для чата, 1,5-кратное увеличение всех характеристик и доступ к специальной локации. В некоторых играх подобные вещи могут стоить около 400 робаксов. Но данные говорят о том, что превращение платных преимуществ в подписку, которую игроки приобретают для продления VIP-доступа, может значительно увеличить прибыль за аналогичный временной интервал по сравнению с продажей версии Game Pass. В данный момент для создания системы подписки вам необходимо записывать время покупки продуктов и сохранять эту инфор-

мацию в данных игрока. Это не очень сложный процесс, но и не самый удобный. Roblox прислушались к своим разработчикам и планируют разработать новую систему подписки и внедрить ее в свой API, чтобы разработчикам не нужно было самим изобретать велосипед. На момент написания данной книги эта система еще не выпущена в какой-либо форме, но на сайте разработчика упоминаются потенциальные возможности ее использования в будущем: **https://developer.roblox.com/en-us/api-reference/class/MarketplaceService**.

Маркетинг

Ну наконец-то, последняя «М» – это маркетинг. Маркетинг – это процесс, позволяющий сделать игру популярной во всем мире и привлечь тысячи новых игроков. Мы уже касались некоторых маркетинговых концепций, а теперь давайте рассмотрим, что делает маркетинговую кампанию успешной и как привлечь как можно больше игроков с минимальными затратами.

Система продвижения Roblox

Как мы уже говорили, в Roblox есть внутренняя система продвижения, реализованная в двух формах показа рекламы игрокам: User Ads и Sponsors. На рис. 7.3 приведен пример игры, рекламируемой через систему Sponsors, и реклама, сделанная для игры с использованием системы User Ads.

У каждой из этих систем есть свои преимущества и недостатки. При использовании системы Sponsors расширения для веб-браузера, блокирующие рекламу, не мешают показу рекламных материалов, как в случае с User Ads. Это довольно важно, потому что с недавнего времени система User Ads перестала отображать содержимое на странице **Games**, и теперь реклама находится лишь в определенных местах на веб-сайте.

Используя User Ads, вы можете создать для рекламы вашей игры изображение определенного размера. Это может быть особенно полезно, если для игры недостаточно лишь мелкого значка, отображаемого системой Sponsors. Часто разработчики изворачиваются на все лады, создавая от рендеров до мемов и полноценных комиксов, чтобы с помощью этой системы придать своей игре больше характера с первого взгляда по сравнению с мелким значком системы Sponsors.

Но в конце 2020 года на странице **Games** появились изменения, сильно повлиявшие на отклик, который разработчики получали на каждый потраченный робакс. Поскольку у системы User Ads ограничено место, изменение порядка сортировки на странице Games сделало так, что игроки начали видеть только спонсируемые игры в разделах Popular и Most Engaged, а все остальные – гораздо ниже. Изменение возымело серьезные и мгновенные негативные последствия: некоторые разработчики заметили, что после изменений пользователи резко стали меньше кликать по рекламе. Изменение разрушило традиционный путь, по которому разработчики делали свои игры популярными.

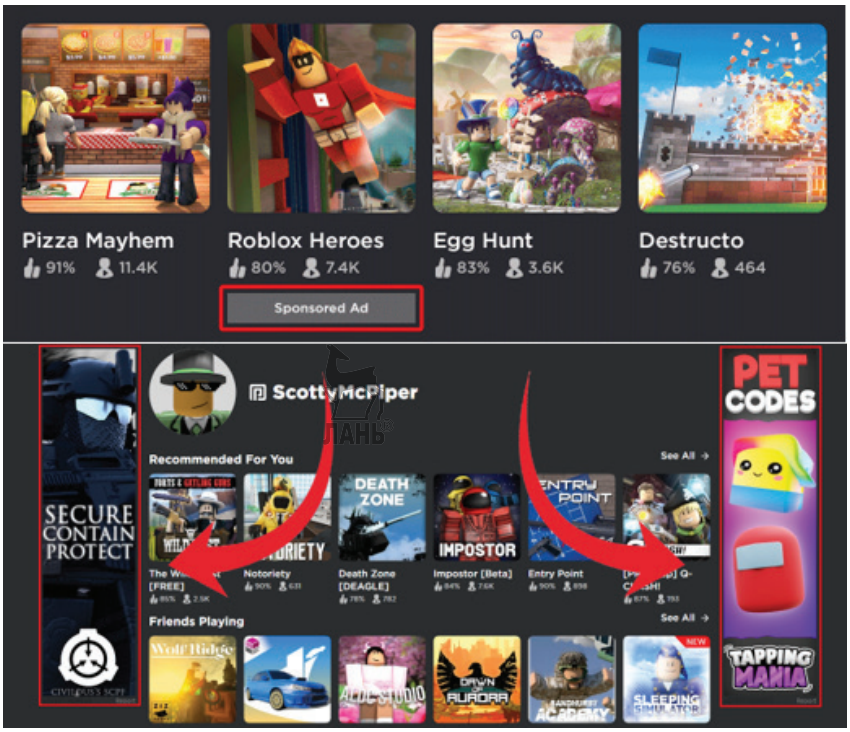


Рис. 7.3. Примеры систем продвижения Sponsore (вверху) и User Ads (внизу)

Еще один фактор, негативно повлиявший на **обнаруживаемость** (то, насколько легко пользователю найти вашу игру), произошел в конце 2020 года, когда Roblox запретили поисковым системам показ материалов со своего веб-сайта. Инструменты поиска на сайте Roblox печально известны своей неточностью. Если игра, которую вы ищете, не слишком популярна, возвращаемые результаты часто получаются совершенно случайными, а названия игр вообще слабо коррелируют с поисковым запросом. Из-за этого многие пользователи для более быстрого поиска игр пользовались Google или Bing. Команда **Developer Relations** отметила, что это временное изменение, и они пытаются улучшить инструменты поиска на своем сайте. Вы можете отследить статус проблемы, понаблюдав за следующей связанной веткой DevForum и другими, подобными ей: **https://devforum.roblox.com/t/no-roblox-games-show-up-in-google-search/841736/33**.

Но Roblox делают и положительные изменения, стараясь улучшить обнаруживаемость. В 2020 году компания внедрила и работает над расширением новой системы, которая позволяет нацеливать рекламные кампании на определенные заданные демографические группы, чтобы вы могли наилучшим образом охватить свою потенциальную аудиторию и ваши усилия не тратились на кого-то, кому ваш контент не интересен. Вы можете узнать больше об этом изменении, перейдя по ссылке на DevForum: **https://devforum.roblox.com/t/we%E2%80%99ve-rebuilt-sponsored-games/832597**.

Чтобы поставить точку в этом разделе, вы должны спросить себя, куда вы хотите распределить свои выделенные на маркетинг деньги, чтобы привлечь как можно больше игроков. Это зависит от того, насколько ваша игра совместима с разными устройствами. В большинстве случаев, чтобы ваша игра была максимально приятной для пользователей мобильных устройств, планшетов и Xbox, вы можете выпустить специальные чехлы для устройств управления, а также дополнительные периферийные функции, такие как изменение внешнего интерфейса на экране. Используя новый функционал Roblox, вы также должны решить, какой демографической группе с большей вероятностью понравятся ваши игры, будь то возраст, пол или другие признаки.

Далее мы обсудим альтернативную форму продвижения, позволяющую достичь гораздо большего, чем вы могли бы сделать с помощью системы продвижения Roblox со всеми ее недостатками.

Ютуберы

Из-за упомянутых ранее факторов и некоторой устарелости платформы Roblox основным компонентом любой маркетинговой кампании становятся ютуберы. Как упоминалось в главе 1, отношения между разработчиками Roblox и пользователями YouTube являются симбиотическими, поскольку разработчики создают игры, создают для ютуберов контент, а подписчики ютубера интересуются игрой и заходят в нее.

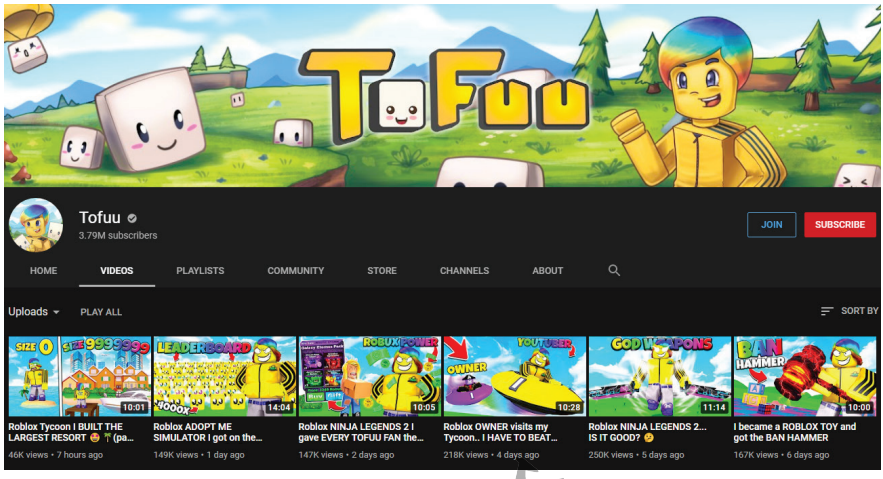


Рис. 7.4. Канал Tofuu – ютубера по тематике Roblox, который часто снимает видео об играх с уникальной механикой

Помимо общего освещения игры, пользователи YouTube иногда ищут разработчиков для создания игр, в которые может играть их аудитория, или для проведения мероприятий. Когда появляются такие возможности, вы должны их использовать. Вы не только сможете порекламировать себя и свои способности к развитию, но и обзаведетесь связями с влиятельными

людьми, которые, быть может, захотят снять видео о ваших будущих играх. Именно так у меня началось партнерство с **Tofuu**, канал которого показан на рис. 7.4. Tofuu хотел найти команду разработчиков и создать игру, и вместе мы создали игру Munching Masters, мою вторую по популярности игру на сегодняшний день, в которой было более 50 млн посещений.

Разумеется, вам не нужно делать с инфлюенсерами полноценный проект, чтобы установить с ними связь. Самый лучший профессиональный совет, который я могу дать в этой книге: просто протяните руку. Вы удивитесь тому, насколько часто простая инициатива написать кому-то первым дает плоды. В любом случае, худшее, что может случиться, – это просто отклонение вашего предложения.

Проверим, чему вы научились

Дойдя до конца главы, вы можете утверждать, что прошли курс по разработке Roblox. Вы продвинулись от полного отсутствия опыта работы с Roblox к владению инструментами, необходимыми для построения собственной карьеры. Давайте рассмотрим некоторые навыки, которые вы приобрели, в хронологическом порядке.

- В *первой главе* вы узнали о том, что из себя представляет платформа Roblox, о том, чего на ней смогли достичь другие, о том, чего вы можете достичь благодаря обучению и практике. Кроме того, мы рассмотрели различные типы разработчиков, которых вы можете взять в свою будущую команду. Наконец, мы поговорили о приобретенных профессиональных навыках, которые вам могут пригодиться в будущем, независимо от вашей карьерной траектории. В целом эта глава была посвящена тому, почему на платформе можно так много заработать и чего другим уже удалось достичь.
- Во *второй главе* вы узнали о Roblox Studio, которая лежит в основе разработки на Roblox. Мы поговорили о том, что значит быть разработчиком Roblox. В этой главе мы поговорили о сайте Roblox, инструментах, которые разработчики используют для создания физической структуры своих игр, а также об общих настройках игры. Эта глава должна была помочь вам встать на рельсы успеха в следующих главах и в вашей карьере разработчика.
- *Третья глава* – это самая важная часть этой книги для вас как разработчика. В этой главе мы рассмотрели множество различных тем, познакомились с языком Roblox Lua, который не только является ключом к созданию игр для Roblox, но и станет отличной базой для изучения любого языка программирования в будущем. Кроме того, мы изучили множество универсальных конструкций в программировании, узнали немало о вычислениях, получили новые в области программирования, применимые ко многим языкам, рассмотрели, как ваш компьютер обрабатывает различные типы данных, принци-

пы отношений «клиент–сервер» и многое другое. Познакомившись с различными типами данных, вы научились ими манипулировать. Узнав о принципах работы с `CFrame` и векторами, вы получили навыки работы в 3D-среде, что невероятно важно на начальном этапе вхождения в сферу STEM.

- В *четвертой главе* мы научились кое-чему большему, чем общее программирование. Вы узнали, как работать с физическими аспектами игрового окружения, как улучшать и добавлять эффекты в различные системы, как разработчики программируют различные варианты поведения игры с помощью сервисов и API-интерфейсов Roblox.
- В *главах 5 и 6* мы занялись непосредственно созданием игр, используя все знания, которые вы получили до этого. В этих главах вы продолжили совершенствовать свои навыки, узнали, как добиться стандартного для многих игр поведения, и заложили основу для того, чтобы вы могли создать любую игру, которую можете себя представить. Например, динамическая структура игр жанров обби и «Королевская битва» позволяет многократно использовать код модулей `Datastore` и `Monetization`, который можно переносить в другие проекты, практически не тратя времени на изменение кода. Всякий раз, когда вас настигают сомнения, вернитесь к этим главам, посмотрите на используемый стиль программирования, принципы использования сервисов и общей организации игры.

Резюме

В последней главе мы рассмотрели передовой опыт, позволяющий любой игре, которую вы придумаете в будущем, завоевать свою аудиторию и стать прибыльной и популярной среди игроков со всего мира. Если вы прислушаетесь к советам трех «М» (механика, монетизация и маркетинг), у вас, без сомнений, получатся игры, способные в кратчайшие сроки выйти на уровень тысяч онлайн-пользователей. Но, несмотря на то что мы делали акцент на зарабатывании денег, не забывайте подходить к работе с азартом и экспериментировать с тем, что вы хотите сделать. Делайте свои игры такими, какими вы хотите их видеть, так как все аспекты разработки, включая те, что обсуждались в этой главе, постоянно развиваются. Не бойтесь выходить за рамки и изобретать новые жанры. Создавайте концепты и визуальные эффекты, о которых раньше никто не задумывался. Станьте лучшим разработчиком, которым только возможно.

Итак, мы подошли к концу. Но на самом деле это не конец. Ваш путь только начался, и впереди у вас множество новых друзей на всю жизнь, популярные игры, профессиональные связи и деньги, позволяющие обеспечить себя в будущем.

Желаю вам удачи в ваших начинаниях!

Предметный указатель

Книги издательства «ДМК ПРЕСС» можно купить оптом и в розницу
в книготорговой компании «Галактика»
(представляет интересы издательств
«ДМК ПРЕСС», «СОЛОН ПРЕСС», «КТК Галактика»).

Адрес: г. Москва, пр. Андропова, 38;
Тел.: **+7(499) 782-38-89**. Электронная почта: **books@alians-kniga.ru**.

При оформлении заказа следует указать адрес (полностью),
по которому должны быть высланы книги;
фамилию, имя и отчество получателя.
Желательно также указать свой телефон и электронный адрес.

Эти книги вы можете заказать и в интернет-магазине:
www.galaktika-dmk.com.

Зандер Брамбо

**Программирование в Roblox.
Сделать игру — проще простого**

Создание игр с помощью Roblox Studio
и языка программирования Lua от «А» до «Я»

Главный редактор *Мовчан Д. А.*
dmkpress@gmail.com
Зам. главного редактора *Сенченкова Е. А.*
Перевод *Райтман М. А.*
Корректор *Синяева Г. И.*
Верстка *Паранская Н. В.*
Дизайн обложки *Бурмистрова Е. А.*

Гарнитура PT Serif. Печать цифровая.
Усл. печ. л. 16,09. Тираж 200 экз.

Веб-сайт издательства: **www.dmkpress.com**

СДЕЛАТЬ ИГРУ – ПРОЩЕ ПРОСТОГО

Roblox — уникальная глобальная виртуальная платформа, предназначенная для создания и прохождения игр. Ее посещает более 150 миллионов активных пользователей в месяц. На Roblox публикуются игры разнообразных жанров, созданные на языке программирования Lua и доступные для запуска всем участникам сообщества Roblox.

Используя данное руководство, вы научитесь пользоваться программой Roblox Studio, создадите свои первые игры, узнаете, как оптимизировать их производительность и сможете успешно монетизировать свои проекты.

Среди рассматриваемых тем:

- знакомство с рабочей средой Roblox Studio;
- программирование на языке Lua;
- создание реалистичных игр в жанрах «Королевская битва» и обби;
- советы по эффективной работе в команде;
- маркетинговые приемы продвижения игр.

Книга предназначена для всех, кто интересуется разработкой игр на платформе Roblox, – как новичков, так и тех, кто уже знаком с Roblox и хочет освоить эффективные приемы разработки и маркетинга.

Для изучения книги не требуется каких-либо знаний о разработке игр.

Интернет-магазин:
www.dmkpress.com

Оптовая продажа:
КТК «Галактика»
books@alians-kniga.ru

Packt›

ДМК ПРЕСС ИЗДАТЕЛЬСТВО

www.дмк.рф

ISBN 978-5-97060-982-8

9 785970 609828 >