Владимир Рубочкин
Юрий Вербиченко

«СОЛОН»



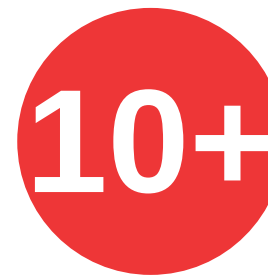
АЗБУКА программирования игр в ROBLOX STUDIO

10+

КНИГА 1

Владимир Рубочкин
Юрий Вербиченко

Азбука программирования игр в Roblox Studio



СОЛОН-Пресс
Москва
2023

УДК 004.4, 621, 374
ББК 32.816, 7426
Р82

Владимир Рубочкин, Юрий Вербиченко
Азбука программирования игр в Roblox Studio 10+. — М.: СОЛОН-Пресс, 2023. — 160 с.

ISBN 978-5-91359-537-9

Давайте спросим себя, почему с 2012-14 г более чем в 30 развитых странах обучают программированию с начальной школы? Ответ прост: программирование является наиболее эффективным средством развития навыков, признанных важнейшими в XXI веке, оно:
— развивает мозг, комплексно задействуя оба его полушария;
— учит действовать, ставя практические цели и достигая их;
— учит анализировать проблемы, экспериментировать, оценивать;
— позволяет самоутверждаться, достигая все более амбициозные цели.

Увлечение компьютерными играми без доступа к их программированию часто ведет к опасной игромании. Программирование игр снимает этот риск и становится сильнейшим стимулом к саморазвитию. И все бы хорошо, но доступ наших школьников к программированию ограничен. В российской школе этого нет. Даже, кружковое программирование в Scratch и Minecraft — это, максимум, до 10 лет. Ведь, если уже поиграл в Roblox-игры, то видишь: это “небо и земля”! И книг по программированию в Roblox для (самого важного) возраста “10+” в России тоже нет — это абсолютно точно!

Великий Михаил Жванецкий как-то сказал: “*Писать, надо тогда, когда уже невтерпеж*.” Вот сейчас примерно такой случай. Авторы этой книги поставили целью сломать имеющиеся барьеры, сделав программирование в Roblox доступным в России любому школьнику “10+”. Соавтором книги является школьник 13 лет. Книга описывает методы и решения, которые сам он уже три года успешно использует в создании игр, получая дипломы на конкурсах. Второй соавтор (имеющий опыт обучения детей программированию) приложил максимум усилий, чтобы книжка была доступна для самостоятельной работы школьникам “10+”. Ведь помощников большинству детей искать просто негде!

Подарите книжку вашему ребенку — и он во всем разберется. А хотите, сами начните программировать игры вместе с вашими детьми!

По вопросам приобретения обращаться:
ООО «СОЛОН-Пресс»
123001, г. Москва, а/я 82
Телефоны:(495) 617-39-64, (495) 617-39-65
E-mail: kniga@solon-press.ru, www.solon-press.ru

ISBN 978-5-91359-537-9
© СОЛОН-Пресс, 2023
© В. Рубочкин, Ю. Вербиченко, 2023

Содержание

Предисловие 5
Полезные подсказки: 6
1. Загрузка Roblox Studio 8
2. Среда программирования 8
3. Строительные блоки Roblox 11
4. Группирование в модели и твердотельное моделирование 16
5. Управление объектами игрового пространства при помощи кода 18
6. Программируем исчезающую платформу 22
7. Программируем смертельную лаву 28
8. Математические и логические действия 31
9. Программируем паркур 33
11. Платформа, через которую можно провалиться 55
12. Начисление баллов игрокам 59
13. Программируем одной программой несколько объектов 68
14. Сохранение данных в хранилище Roblox 72
15. Игра с сохранением баллов игрока между сеансами игры 77
16. Инструменты и стреляющее оружие 87
17. Перезарядка оружия 104
18. Программируем ландшафт 109
19. Как пользоваться Toolbox 117
20. Взрывы: мины и бомбы 118
21. Наклейки с текстурами и картинками 124
22. Излучатели частиц 127

23. Огонь 129
24. Программируем дым 131
25. Лучи смерти 133
26. Источники света 136
27. Программируем “двойной прыжок” 138
28. Программируем сиденья 140
29. Создаем тексты и изображения на игровом поле 144
30. Программируем неигровые персонажи 148
31. Телепортация игроков 151
32. Создание команд игроков 154
33. Разработка игры 157
34. Публикация игры 158
35. Не только игры 159

Предисловие

Многие, кто приходит к программированию в Roblox уже прошли школу Scratch. Это естественно, ведь 3D-мир гораздо интереснее, чем мир плоский. Да и разрешение картинок в Scratch гораздо хуже, чем в Roblox. И это — не говоря о многих недоступных в Scratch спецэффектах и возможностях!

Но, как говорится, лучшее имеет свою цену! Программирование в Roblox Studio сложнее, чем программирование в Scratch. Книг нет, и первое, что приходит на ум — поискать в Youtube, ведь “там есть все”! Но, этот подход к созданию игр имеет свои “минусы”. Конечно же, вы сможете посмотреть ролики в Youtube, скопировать приведенные в них коды и вставить их в свою игру. Открытыми останутся лишь вопросы: “А как же работает код, который вы скачали? Почему этот код устроен именно так?” На эти вопросы многочисленные видеоблогеры ответов не дают: ”Копируй! Вставляй! И все будет работать!” Но, так программистами не становятся, и настоящей серьезной игры вы таким способом не создадите. Это однозначно!

Если вы это понимаете — эта книжка для вас. Одним из ее соавторов является такой же, как вы школьник 13-ти лет. Четыре года назад он тоже начинал в Roblox с поиска подсказок в Youtube, но в 10 лет его папа сказал: “Читай туториал!” И он начал почитывать. А это — трудно, ведь туториалы Roblox написаны на английском. Нужно пользоваться программой перевода с английского. А если вам примерно столько же лет, как и ему, вам не нужно объяснять, что многие термины даже в переводе на русский могут быть абсолютно непонятны! Но, он был упорен и много экспериментировал. Без проб и ошибок в программировании — никуда! Сегодня у моего соавтора много своих Roblox-игр, есть и дипломы конкурсов по программированию. Получал он поощрения и донаты от самого Roblox. Но, он не расслабляется — продолжает упорно “грызть гранит науки”. Roblox Studio и язык программирования Lua — вещи сложные, а “слона нужно есть по частям”!

Наша книга содержит описание инструментов и решений, которые позволят вам не только разработать свои собственные Roblox-игры, но и найти ответы на многие вопросы, на которые не отвечают виде-

облогеры. Мы очень старались, чтобы все, что описано в этой книге было вам понятно.

И наш совет: ищите друзей, которым тоже интересна разработка компьютерных игр, советуйтесь с ними, делитесь с ними полезной информацией и, обязательно, привлекайте их к разработке совместных проектов. Программирование — не для одиночек! Каждому найдется работа по интересам. Кто-то станет дизайнером вашей игры, кто-то разработает классный сценарий, а кто-то наполнит игру крутыми спецэффектами! Желаем вам успехов!

Полезные подсказки:

* при копировании текстов программ из книги **никогда** не ставьте никакие знаки препинания в конце строк кода (даже если они есть в книге);
* набирайте тексты программ **только в английской раскладке** клавиатуры. Каждый символ (даже визуально очень похожие символы кавычек **“ ”**) в русской и английской раскладках имеют разные коды и если вы вместо “английских” используете “русские кавычки”, Roblox воспримет это, как ошибку, а вы не сразу ее обнаружите;
* **регулярно сохраняйте свою работу**, например, через каждые 10 минут. Таким образом, если что-то пойдет не так, по крайней мере, вы не потеряете многого. При сбоях ПК используйте копии ваших программ, которые Roblox Studio автоматически сохраняет в папке **Roblox - AutoSave**;
* не редактируйте программы в то время, как они запущены — ваши изменения не будут действовать — нужно остановить программу, исправить скрипт и запустить программу заново;
* при разработке и отладке программ **всегда** держите открытыми окна **Output** и **Script Analysis,** чтобы контролировать сообщения об ошибках, которые Roblox Studio выводит в эти окна. Эти сообщения выделяются красным цветом шрифта — пытайтесь читать, что написано, в окне **Script Analysis** и **Output**;

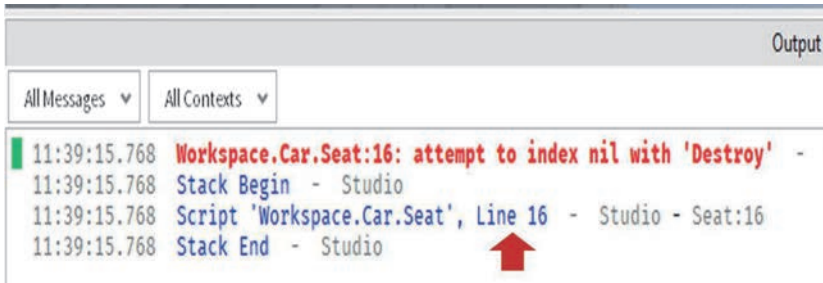


Рис. 0.1 Пример сообщения об ошибке программы в окне Output

В приведенном на рисунке примере показано сообщение об ошибке программы при попытке исполнения строки 16 (**Line 16**) скрипта с названием **Seat**, входящего в состав объекта **Car**, являющегося составной частью **Workspace**;

* при отладке программы не держите открытой программу, которую не отлаживаете. Окно **Output** будет работать только в одной из открытых программ и, вполне возможно, не в той, что нужно;
* для отладки программ регулярно пользуйтесь функцией **print()**, распечатывая те или иные переменная, чье значение должно меняться в ходе программы и, по изменению чьей величины, можно судить о том, правильно ли исполнилась какая-то команда или нет;
* когда создаете **parts** в меню **Part** Roblox Studio, а затем копируете в них заимствованный где-либо или даже в этой книге скрипт, всегда ставьте в скрипт реальные координаты **parts** на вашем игровом поле, а не координаты, которые указаны в заимствованном скрипте, иначе скрипт может не работать.

1. Загрузка Roblox Studio

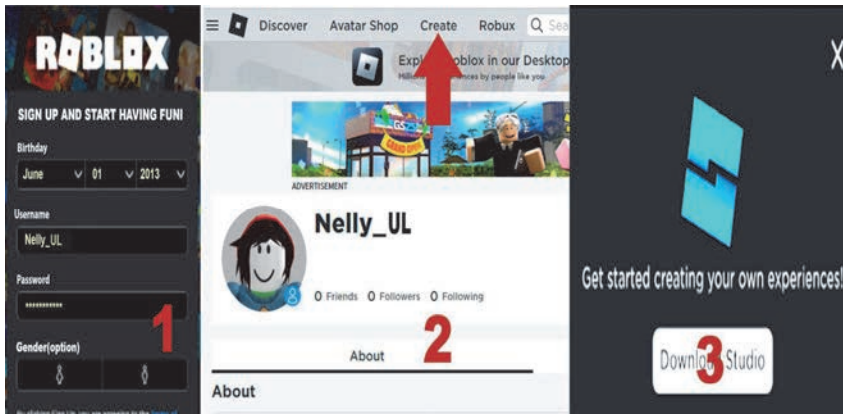


Рис. 1.1 Загрузка Roblox Studio

1. Заведите себе аккаунт в **Roblox** https://www.roblox.com/
2. Войдите в свой аккаунт Roblox и запустите Roblox Studio, кликнув кнопку **Create**.
3. Найдите установщик **Roblox Studio** в истории загрузок браузера и дважды щелкните файл (в Windows это файл RobloxStudio.exe). Войдите в **Roblox Studio**, используя свою учетную запись Roblox.

2. Среда программирования

Открыв Roblox Studio — вы увидите несколько шаблонов (см. рис. 1.1). Это — заготовки для создания новых игр.

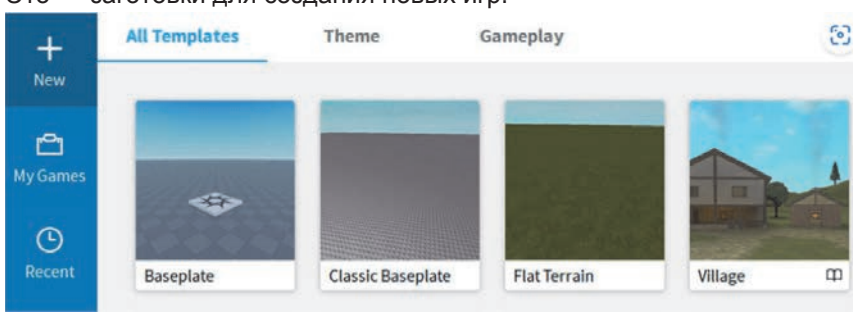


Рис. 2.1 Шаблоны игровых пространств Roblox Studio

Кликните, например, по Baseplate. **Baseplate** — это большая плоская площадка, точно в центре которой (координаты X=0 и Z=0), расположено место спавна **SpawnLocation**, на котором появляется аватар игрока, когда он входит в игру. Но, можете выбрать и заготовку **Flat Terrain** или любую другую. **Flat Terrain** — это тоже большая плоская площадка, но она покрыта травой. Можно выбрать и из других заготовок — на которых уже есть постройки. Но, лучше начните с **Baseplate** — чтобы не отвлекаться на то, что не важно.

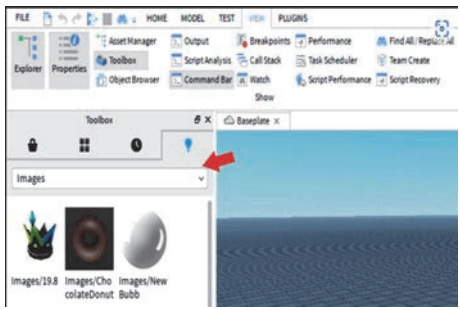


Рис. 2.2 Открытое окно Toolbox

Если вы впервые открываете Roblox Studio, в ней будет открыто несколько окон, которые вам не нужны.

Закройте их, чтобы у вас было больше места на игровом поле для работы над игрой. В частности, закройте панель инструментов **Toolbox**, щелкнув по значку “×” в правой верхней части этого окна. Убедитесь, что вы видите такую картинку.

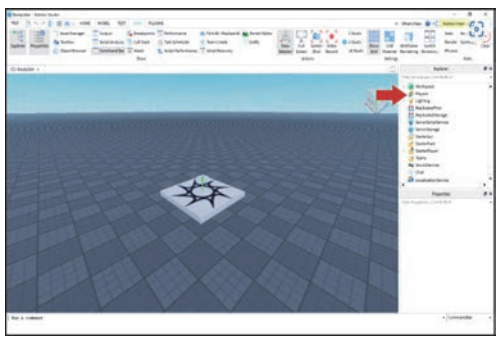


Рис. 2.3 Игровое поле Baseplate и окно Explorer

С правой стороны вы должны всегда видеть открытое окно **Explorer** (Проводника). Кликните по кнопке **View** (Просмотр) и разберитесь с местоположением кнопок вызова/открытия основных окон Roblox Studio, которые нам понадобятся для изучения материала этой книги. Кстати, поставив курсор мыши на границу окон **Properties**, **Explorer** и **любых других**, вы можете изменять (увеличивать или уменьшать) размеры этих окон.

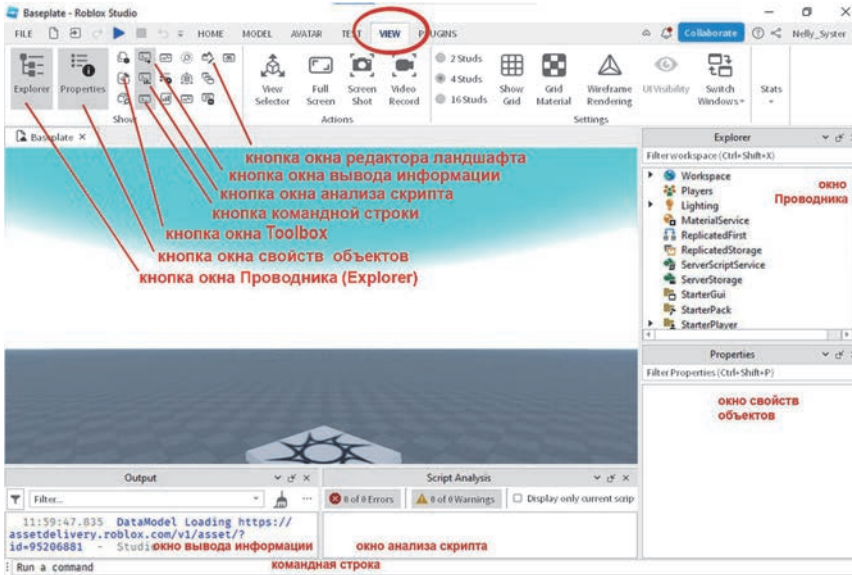


Рис. 2.4 Основные окна Roblox Studio и кнопки их вызова

Заметим, что когда вы вызываете какое-то из окон, оно может появиться в случайном месте экрана, что часто бывает неудобно.

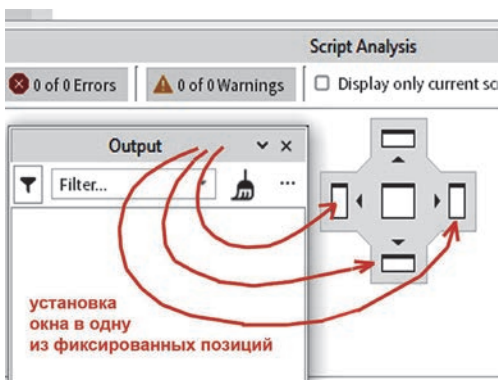


Рис. 2.5 Пример перемещения одного из окон (окна Output) для закрепления его в удобной позиции

Поставьте курсор мыши на “шапку” появившегося окна, нажмите левую кнопку мыши и чуть сдвиньте окно — появится окно с “геймпадом”. Выбрав одно из его положений, вы можете закрепить выделенное окно в той фиксированной позиции, где вам будет удобно.

В окне Проводника (**Explorer**) всякий раз показан полный исходный состав объектов создаваемой вами игры. Левее некоторых перечисленных в **Explorer** объектов стоят направленные вправо стрелки. Если кликнуть по такой стрелке, она развернется острием вниз, а под объектом, рядом с которым она стоит откроется список объектов, вхо-

дящих в этот “родительский” объект. Такие же стрелки будут появляться, когда вы будете добавлять (вкладывать) в имеющиеся объекты Roblox Studio какие-то новые объекты.

Кликнув по любому из объектов **Explorer**, вы подсвечиваете его строку, одновременно, открывается окно свойств (**Properties**) выделенного объекта, а правее названия объекта в **Explorer** появляется кнопка ⊕**,** кликнув по которой, можно открыть “выпадающее” меню, позволяющее дополнить выделенный объект новыми объектами или свойствами.

Кликнув по синей кнопке “**Play**”, (в самой верхней строке меню Roblox Studio) можно запустить программу/игру, в которой ваш аватар может передвигаться по игровому пространству при помощи кнопок клавиатуры **W**, **A**, **S**, **D** или кнопок со стрелками “**вверх**”, “**вниз**”. Аватар способен прыгать, если вы будете нажимать клавишу “**пробел**”. Чтобы осмотреть окрестности игры, нужно нажать правую кнопку мыши и подвигать мышку, а можно использовать кнопки “**влево**”, “**вправо**”. Используйте колесико мыши, чтобы рассматривать удаленные предметы игрового поля или отдельные объекты игрового поля. Останавливается программа/игра нажатием красной (квадратной) кнопки **Stop**.

Сохранить игру можно обычным образом — через меню **File**. Кроме того, Roblox Studio автоматически сохраняет копии программ в папке **Roblox - AutoSave**.

3. Строительные блоки Roblox

Parts (части) — это и элементарные строительные блоки вашего проекта и то понятие, которым мы будем обозначать любые объекты, входящие в создаваемую вами игру. Элементарные строительные блоки вы можете использовать для создания игровой среды. Например, **parts** можно использовать в качестве платформ в паркуре, кирпичей для создания сложных зданий в вашем городе или пластин и других элементов для разных технических объектов типа танков и космических ракет.

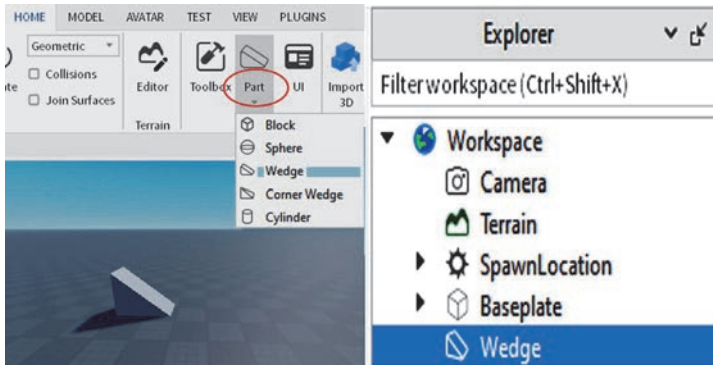


Рис. 3.1 Меню элементарных строительных блоков раздела HOME

На вкладке **Home** (Главная) щелкните по стрелке ниже кнопки **Part** — откроется меню, в котором перечислены доступные варианты элементарных строительных блоков: **кирпич**, **шар**, **клин**, **угловой клин**, **цилиндр**. Кликните по любому из этих элементов — он, сразу же, появится на игровом поле. Одновременно с этим, в окне **Explorer** под объектом **Baseplate** появится новый объект с таким же именем. Если вы выбрали клин (**Wedge**), то и в **Explorer** также появится **Wedge**. Обратите внимание, что если объект подсвечен в меню **Explorer**, то он выделен и на игровом поле. И наоборот, если выделить объект на игровом поле, он выделится в окне **Explorer**.

Нажмите кнопку **MODEL** (Моделирование) Roblox Studio и перейдите в этот режим. Опишем меню режима:

— **Select** (выбрать объект);
— **Move** (двигать объект);
— **Scale** (изменить масштаб объекта);
— **Rotate** (вращать объект);
— **Transform** (трансформировать объект);

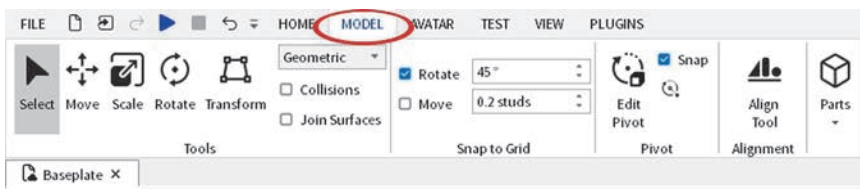


Рис. 3.2 Меню режима MODEL

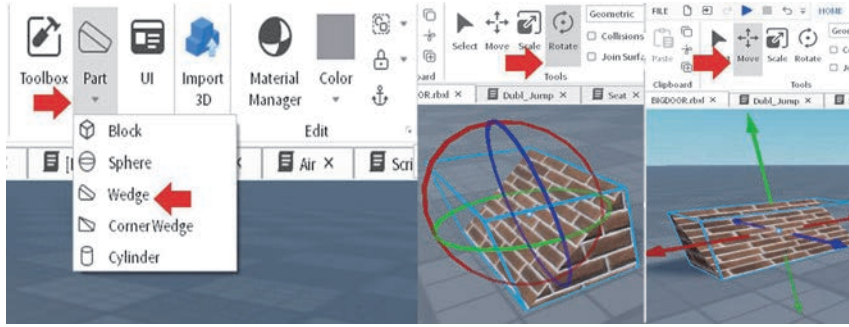


Рис. 3.3. Выбор строительного блока, его вращение и перемещение

При выборе режимов **Select**, **Move**, **Scale** вы увидите цветные точки, стрелки или окружности в непосредственной близости от строительного блока. Это - своеобразные “ручки”, зацепившись за которые курсором мыши, можно изменять размеры строительных блоков, перемещать или вращать их по всем трем осям координат. При этом красная “ручка” соответствует оси координат “X”, синяя “ручка” — оси “Z”, зеленая — оси “Y”. Вы также должны видеть, что оси координат “X” и “Z” расположены в горизонтальной плоскости, а ось “Y” направлена вертикально. И еще одно: в Roblox расстояния измеряются в собственных единицах измерения — в “шипах”.

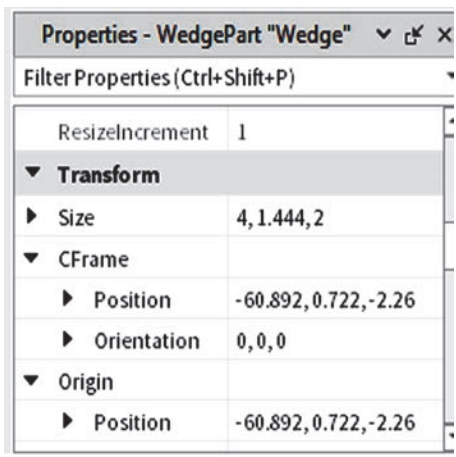
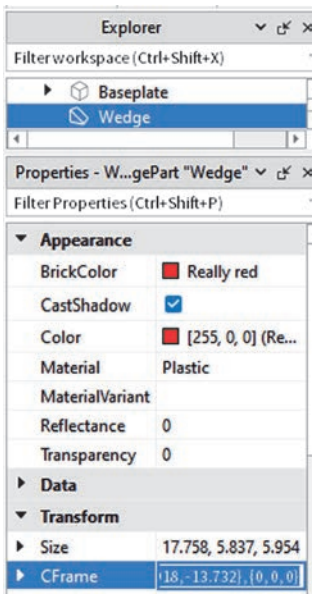


Рис. 3.4 Изменение размеров и положения блока типа Wedge

Найдите в окне **Properties** в списке параметров выбранного вами строительного блока параметр **Size** и, взявши за какую-либо из цветных “ручек” поменяйте размер строительного блока. Вы увидите, что цифры параметра **Size** (Размеры) в окне **Properties** будут изменяться синхронно с движениями “ручки”, за которую вы держались. Это означает, что размеры блоков вы можете изменять обоими способами: либо двигая “ручки”, либо меняя цифры

параметра **Size** в окне **Properties**. Аналогично, можно настраивать местоположение блоков в пространстве игрового поля или поворачивать объекты относительно осей координат.



Если вы будете менять описанные выше параметры строительных блоков с использованием “ручек”, вы должны будете предварительно настроить “шаг” вносимых вами изменений. Обратите внимание, что на рисунке этого меню в разделе **Snap to Grid** (Шаг изменений) стоит “галочка” рядом с **Rotate**, а правее (в окошке) стоит “45град”. Это означает, что при вращении блока, он будет каждый раз поворачиваться только на 45 градусов. Если вам нужен другой шаг изменений, настройте его, используя стрелки “вверх-вниз” этого окошечка.

Рис. 3.5 Окно Properties строительного блока

Окно **Properties** позволяет просто настроить многие параметры блоков:

— **Brick Color** позволяет выбрать цвет;
— **CastShadow** включает тень от объекта;
— **Material** задает материал блока;
— **Transparency** задает прозрачность;
— **Size** задает размеры;
— **CFrame** задает координаты местоположения;
— **Anchored** закрепляет (“якорит”) блок в том месте, куда вы его поставите. Если не поставить “галочку” в этом окне, ваш блок при запуске программы обязательно упадет на **Baseplate**, если он исходно не лежал на ней;
— **CanCollide** задает конструкционную жесткость объекта. Если не поставить “галочку”, задавая это свойство, то объект не сможет сталкиваться с другими объектами, иначе говоря, через него сможет проходить любой другой объект и даже игроки.

Если в меню **MODEL** нажать на стрелку “вниз”, расположенную под кнопкой **Part** появится меню, в котором есть **Material Manager** (Мене-

джер Материалов), позволяющий выбрать материал, из которого вы хотите, чтобы был изготовлен ваш строительный блок. Попробуйте поменять материал вашего строительного блока. Для этого нужно выбрать (выделить) строительный блок, выбрать нужный материал в меню, взять пиктограмму “ведерко с краской” и залить материалом выделенный строительный блок. Далее можно продолжать выбирать другие строительные блоки и заливать их материалом.

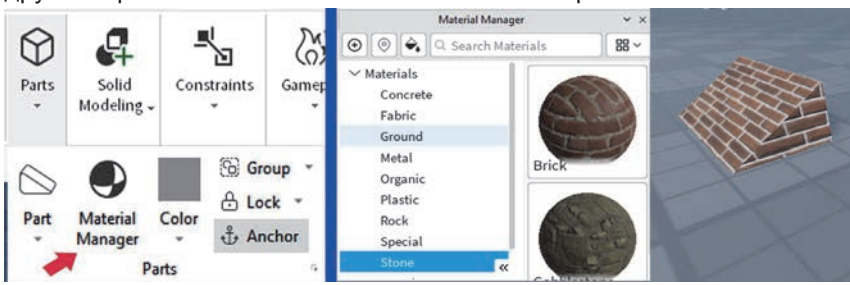
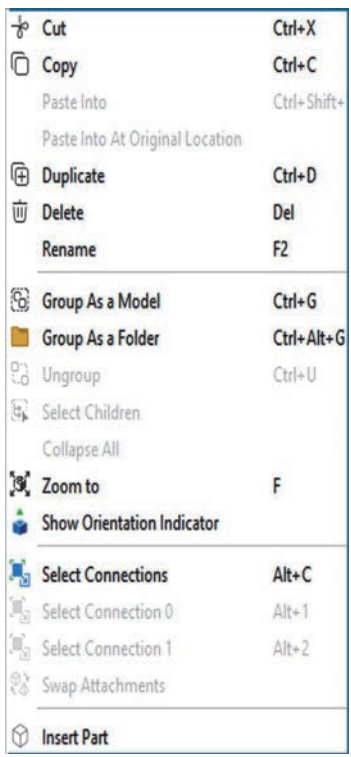


Рис. 3.6 Вызов Material Manager и заливка материалом детали Wedge



В окне **Explorer** кликните правой кнопкой мыши по по подсвеченной строке созданного вами строительного блока (или любого другого объекта) — появится меню управления объектами, входящими в это окно.

Рис. 3.7 Меню управления объектами в окне Explorer

Выбрав соответствующую опцию этого меню, вы можете вырезать (**Cut**), скопировать (**Copy**), задублировать (**Duplicate**), удалить (**Delete**), переименовать (**Rename**) выбранный объект.

Потренируйтесь: сделайте дубликат объекта, переименуйте полученный дубликат, скопируйте его в какой-нибудь другой объект папки **Explorer**, а потом удалите.

Мы не полностью описали это меню. Далее, вы поймете, что для других объектов, входящих в **Explorer** в нем есть и другие команды. Например, для объектов типа **Script** (программа) есть команда **Disable Script**, позволяющая временно отключать исполнение той или иной программы.

4. Группирование в модели и твердотельное моделирование

Подумав, вы скажете, что это не удобно — иметь дело с огромным количеством отдельных строительных блоков (например, кирпичей, из которых построено здание или конструкций из которых состоит танк). Действительно, неудобно! Поэтому в Roblox Studio имеется возможность объединения (группирования) деталей в модели.

Чтобы объединить какие-либо **parts** в модель, достаточно выделить эти **parts** любым из двух способов:
— либо в окне **Explorer** — нажав и удерживая кнопку **Ctrl**, кликнуть по каждой из нужных вам **parts**;
— либо на игровом поле, нажав и удерживая левую кнопку мыши, перемещать мышь так, чтобы в ее окно выделения вошли все необходимые детали.
Затем, кликнув правой кнопкой мыши, вызвать меню управления объектами и в нем кликнуть кнопку **Group As a Model** (Сгруппировать как модель). После этого, можно работать с моделью, а не со многими деталями, из которых она состоит.

Но, и это — не все возможности **Roblox Studio**. А как быть, например, если нужно создать шестеренку или трубу или отверстие в строительном блоке? Ответ прост: для этого существует “твердотельное моделирование”.



Рис. 4.1 Результаты твердотельного моделирования

Твердотельное моделирование — это процесс соединения простых деталей для формирования более сложных. В Roblox Studio доступны 3 операции твердотельного моделирования — найдите их в разделе **MODEL** в подразделе **Solid Modeling**:

— **Union** (Объединение) — соединяет две или более частей вместе;

— **Negate** (Вычитание) — вычитает одну **part** из другой или других;

— **Separate** (Разделение) — разделяет Union обратно на составные части.

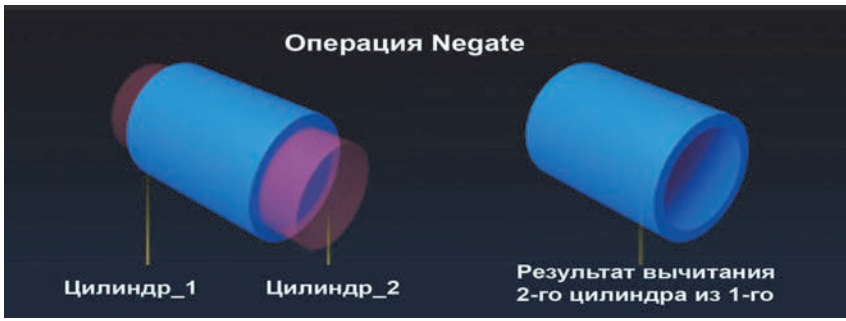


Рис. 4.2 Операция Negate

Инструмент **Union** соединяет две или более деталей вместе, образуя объединение.

Union — это не то же самое, что группа **parts**, объединенная в модель командой **Group As a Model** в меню управления объектами окна **Explorer**. **Union** соединяет все части в единое целое, в то время как, в модели каждая **part** существует самостоятельно, даже если они движутся все одновременно. Вы можете изменить свойства любой отдельной **part** в группе, в то время как изменения свойств влияют на **Union** целиком. Но, **Union** сохраняет свойство цвета каждой из его частей. Чтобы изменить цвет **Union** на общий цвет, включите параметр **PartOperation.UsePart Color** в окне свойств.

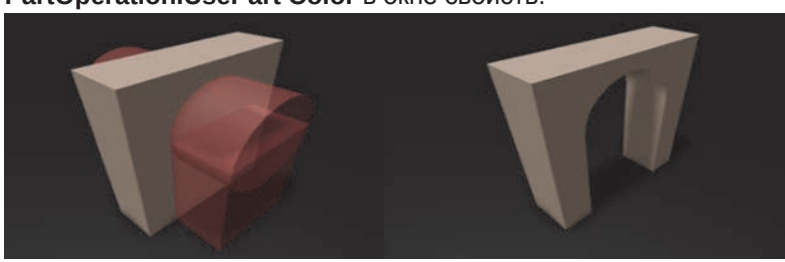


Рис. 4.3. Твердотельное моделирование строительной арки

Чтобы соединить детали в **Union**:
— выберите все части, которые вы хотите соединить вместе (либо нажмите клавишу Shift и щелкните по каждой из **part**, либо нажмите правую кнопку мыши и выделите область, в которую входят все нужные **parts**;
— в меню Roblox Studio выберите вкладку **MODEL** и раздел **Solid Modeling**.
— выберите **Union**.

Инструмент **Negate** позволяет выбрать **parts**, которые будут вычитаться из других. Выбор опции **Negate** для какой-то из **part** изменяет **part**, так что она становится полупрозрачной, что позволяет точно расположить ее и увидеть, что будет удалено, когда части объединятся по команде **Union**.

Инструмент **Separate** снова разделяет **Union** на отдельные части. Поэтому, если что-то объединилось не так, как вам это нужно, разделите **Union** на составные **part** и исправьте недостатки.

Помимо использования инструментов **Union** и **Negate** в разделе **MODEL**, вы также можете использовать эти инструменты, в ваших скриптах. Подобно инструментам **Union** и **Negate**, код **UnionAsync()** объединяет **parts** в **Union**, а код **SubtractAsync()** вычитает одну часть из другой. В разделе, где мы опишем создание паркура, мы покажем как использовать операцию вычитания в коде программирования платформы паркура.

5. Управление объектами игрового пространства при помощи кода

Программы (или код) в Roblox пишут на языке под названием **Lua**. Код хранится и запускается из “скриптов”, которые представляют собой последовательности тех или иных команд. Скрипты можно размещать в различных объектах в Проводнике. Если вы поместите ваш скрипт непосредственно в **part**, Roblox запустит этот код, когда **part** будет загружена в игру.

Выберите существующую **part** или создайте новую. Переименуйте ее в **TestPart**. Наведите в окне **Explorer** указатель мыши на **TestPart** и нажмите на кнопку ⊕, расположенную правее. Затем, в появившемся меню нужно выбрать **Script**. В **Explorer** ниже **TestPart** “вставится” скрипт, который будет иметь такое же название **Script**. Кликните по нему правой кнопкой мыши и переименуйте (выбрав **Rename**), например, в **ChangeColor** (СменаЦвета). Переименование желательно делать каждый раз, создавая новый объект, иначе, впоследствии, вам сложно будет ориентироваться среди многих одинаково названных объектов и, в частности, среди скриптов с одинаковым названием.

Исходный текст скрипта, впервые появляющегося на месте игрового поля, всегда один и тот же: **print ("Hello world!")**. Этот скрипт вам всегда придется заменять своим. Напишите в самой верхней строке окна скрипта комментарий в виде краткого описания того, что будет делать ваш скрипт. Напишите ”- -Изменение цвета детали”. Обязательно начните эту строку двумя расположенными друг за другом символами тире — без этого ваш комментарий будет воспринят, как ошибка кода. А то, что написано за двумя символами “тире” Roblox просто не читает.

Чтобы внести изменения в любую **part** с помощью кода, скрипту сначала нужно знать, с какой **part** ему нужно работать. Используя окно **Explorer**, нам легко найти местоположение **part**: наша деталь **TestPart** находится в **Workspace**. Но, наше знание о конкретном пути, по которому можно найти **part**, нужно преобразовать во что-то, что может понять Roblox Studio:

— под комментарием напишите **game** — это дает Lua доступ к рабочей области игры. Используйте точки для разделения нескольких слов и в той же строке введите **Workspace**, а завершите строку, набрав еще одну точку и название нашей **part**:

```
game.Workspace.TestPart
```

Заметили, что Roblox автоматически дополняет слова по мере ввода, чтобы ускорить процесс кодирования? Когда слова “подсказок” появятся, можно использовать клавиши со стрелками для перемещения вверх-вниз по списку или мышку и выбрать нужный вариант, нажав Enter;

— чтобы получить доступ к свойству цвета нашей **TestPart** введите (через точку) свойство **Color**. Скрипт изменит свойство **Color** детали на новое значение если указать, тип данных **Color3** в котором хранятся цвета:

```
game.Workspace.TestPart.Color = Color3
```

— чтобы изменить цвет детали, используем **RGB**-кодировку цвета. Компьютеры часто используют **RGB**-кодировку (сочетания красного, зеленого и синего цветов **Red-Green-Blue** для создания любых цветов на экране). В **RGB**-кодировке используются три числа — каждое от 0 до 255, разделенные запятыми. Например, черный — это (0, 0, 0), а белый — (255, 255, 255).

```
game.Workspace.TestPart.Color = Color3.fromRGB()
```

— хотя цифры цвета в круглых скобках можно вводить вручную, проще использовать палитру цветов. Кликните внутри круглых скобок — появится цветовой круг — выберите в нем подходящий цвет. Получившийся код должен выглядеть так, как показано ниже, хотя может иметь другое значение RGB:

```
game.Workspace.TestPart.Color = Color3.fromRGB(255, 230, 50)
```

Нажмите **Play**, чтобы запустить программу и проверить, меняет ли цвет ваша **part**.

Все хорошо, цвет меняется. Но, оказывается, что код, который вы использовали до сих пор, не будет работать, если в игре имеется несколько **part** с одним и тем же именем (в частности, с именем **TestPart**). Это означает, что если вы захотите поменять цвета несколько **part**, вам придется создавать отдельные скрипты для каждой из них. Представьте, что вы профессиональный разработчик игр, программирующий огромную игровую среду, состоящую из тысяч **part**. Это же очень долго!

Вместо этого вы можете использовать в скриптах связи типа “родитель - потомок”. **Parent** (“Родитель”) — это любой объект, который имеет встроенные в них другие объекты, например, **part** или скрипты. Все, что расположено внутри ниже “Родителя”, является его **Childs**

(“детьми”). В приведенном примере объект **TestPart** является родительским, а скрипт **ChangeColor** — дочерним.

С прежним скриптом вы можете изменить цвет только одной **part** с именем **TestPart**. Чтобы изменить цвет любой другой **part**, можно использовать отношение Родитель-потомок. Если в скрипте использовать конструкцию **script.Parent**, то этот скрипт прикрепится к любой родительской **part**, в которую он будет встроен. Это сделает возможным использование одного и того же кода в любой **part**, независимо от ее имени.

```
local part_test = script.Parent
part_test.Color = Color3.fromRGB(50, 240, 255)
```

Вставьте в скрипт **TestPart** этот код и сделайте несколько копий **part**, щелкнув по **part** правой кнопкой мыши и выбрав **Duplicate**. Запустите программу и убедитесь, что все **part** изменили свой исходный цвет.

Поясним: скрипты обычно начинают с создания в них “**переменных**”. **Переменная** — это имя, связанное с каким-то значением или “коробочка”, в которую можно в разное время положить разное количество каких-то вещей. После создания переменной ее можно использовать многократно. **Значение** переменных можно менять — по мере необходимости.

Переменные Lua могут содержать **самые разные типы данных**: числовые (целые или дробные), “строковые” (содержащие сочетания букв и цифр), логические (истина/ложь) и другие. В языке Lua переменная создается следующим образом: `local имя = значение`

То есть, написав `local part_test = script.Parent`, мы создали переменную **part_test** и присвоили ей значение **script.Parent**. Это означает, что Родителем скрипта является тот объект, в котором находится скрипт. То есть, Родителями являются все созданные нами **parts**. Термин **local** (ограниченный) означает, что созданная нами переменная будет использоваться только в том блоке скрипта, где она объявлена.

Имена переменных чаще пишутся **со строчной буквы**, но после первой буквы слитно или через символ "_" можно использовать и заглавные буквы и другие символы например: **a1, var_name, _test just_LikeThis**.
Недопустимыми именами переменных являются: **and, break, do, else, elseif, end, false, for, function, if, in, local, nil, not, or, print, repeat, return, then, until, while** и имена из сочетаний чисел и букв, с числом на первом месте, типа **25th**.

Обратите внимание, что вы только что изменили цвет нескольких частей, не назвав их конкретно в тексте программы.

6. Программируем исчезающую платформу

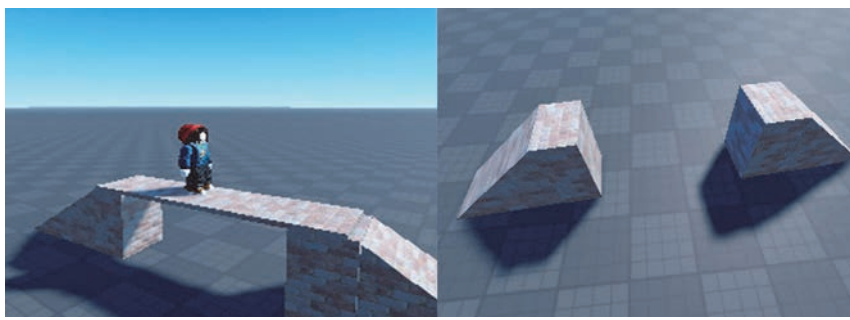


Рис. 6.1 Мост с исчезающей платформой

Вы узнали, как создавать детали (**parts**) и управлять ими в Roblox Studio. Давайте создадим простенький мост, сверху которого положим плоскую платформу и встроим в нее программу так, чтобы платформа периодически исчезала и появлялась. Вы сможете использовать программу в игре, чтобы временно перекрывать пропасть, давая возможность игрокам перебраться на другую сторону.

Уточним, сейчас нам не нужен сложный игровой мир — просто нужен какой-нибудь провал, через который игроки не смогут перепрыгнуть без моста или чего-то подобного. Создайте новую **part** типа "платформа" и переименуйте ее, например, в "**p_Brige**". Скопируйте в нее следующий код, чтобы создать переменную с именем **platform**, имеющую значение **script.Parent**:

```
local platform = script.Parent
```

Создаем функцию

Функции — это **блоки кода**, имеющие имя и предназначенные для выполнения каких-либо определенных действий. Функции можно многократно использовать в разных программах без повторного написания, что создает определенные удобства.

Существуют готовые функции, такие как **print()** (печатать) и **wait()** (ждать). Например, функция **print()** отображает текст на экране компьютера, она очень полезна для отладки программ. Чтобы увидеть какое значение получила та или иная переменная, используйте в своих программах функцию **print()** — тогда вы сможете прямо в процессе тестирования программы увидеть значение переменной в окне **Output** Roblox Studio.

Давайте, создадим в нашем скрипте функцию исчезновения (**disappear**):

```
local function disappear()
    “внутренний код функции”
end
```

Строка 2 скрипта объявляет функцию — указывая начало функции и ее название “**disappear**”. Внутренний код функции должен быть вписан между ее первой строкой и концом функции, обозначенным словом **end**. Скобки после названия функции нужны для включения дополнительной информации — “**параметров**”.

Параметры — это какие-то переменные, которые вы должны сделать доступными для функции и которые используются внутри этой функции. Чтобы указать параметры их перечисляют в круглых скобках. Но, можно и не назначать параметров функции.

Важно, что текст “внутреннего кода функции” должен вводиться с отступом вправо, точно равным интервалу, возникающему при нажатии на клавиатуре кнопки “Таб”. Заметим, что **Roblox Studio** автоматически делает отступ равный одному нажатию кнопки **Tab** внутри каждой новой функции. Но, следите за тем, чтобы в вашем коде были такие

отступы — они помогают указать, где функция начинается и где она заканчивается.

В языке Lua можно создавать и функции без имен (“анонимные функции”), их используют в качестве обработчиков каких-либо событий. Анонимные функции также должны начинаться и заканчиваться ключевыми словами **function** и **end**, но вам не нужно использовать ключевое слово **local** для указания локальной области действия этих функций, поскольку они всегда имеют локальную область действия.

Управление свойствами детали

Когда платформа исчезает, она должна стать невидимой, и пользователи должны проваливаться сквозь нее — но вы не можете просто уничтожить платформу, ведь она должна появляться над пропастью вновь и вновь.

Мы помним, что все **parts** могут иметь много самых различных свойств, и это можно использовать. Нашу платформу можно сделать невидимой, изменив ее свойство **Transparency**. Прозрачность может иметь значение от “0” до “1”, где “1” означает полную прозрачность и, следовательно, невидимость.

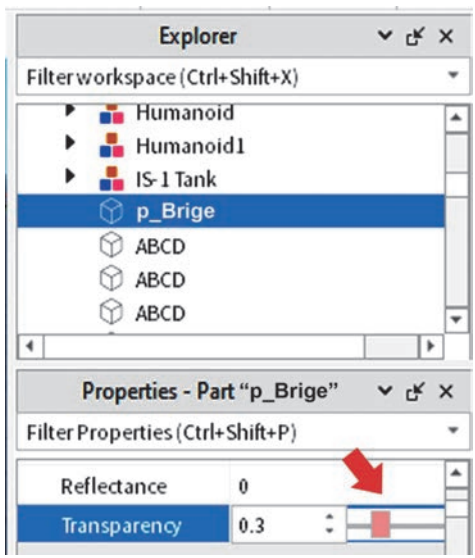


Рис. 6.2. Плавное регулирование свойства Transparency

Кликните мышкой по **p_Brige**, выберите в Roblox Studio меню **View** и кликните по кнопке **Properties** (если это окно не было открыто). Откроется окно **Properties.** Кликните правее установленного в окне числового значения свойства **Transparency.** Появится движок которым можно плавно регулировать свойство от “0” до “1”. Подвигайте регулятор — вы увидите, что прозрачность **p_Brige** будет меняться.

Свойство **CanCollide** определяет, могут ли другие части (и игроки) проходить (проваливаться) прямо через **p_Brige**. Если вы установите значение **false**, игроки будут проваливаться через платформу. Если установить **true**, по платформе можно спокойно пройти пешком.

Свойства **переменных** указываются через точку после их наименования В функции исчезновения установите для свойства **CanCollide** платформы значение **false** (через платформу можно провалиться). А для свойства **Transparency** установите значение "**1**" (платформа прозрачна):

```
1   local platform = script.Parent
2   local function disappear()
3       platform.CanCollide = false
4       platform.Transparency = 1
5   end
```

Внимание: копируя приведенные в книге программы не нужно ставить порядковые номера строк. Мы ставим их только для того, чтобы иметь возможность описать, что делает каждая конкретная строка программ. А если номер строки отсутствует — **пишите текст слитно** с текстом предыдущей строки (без каких-либо пробелов).

Вызов функции

Как только вы объявили и полностью описали функцию, вы можете запустить ее, написав ее имя (вместе со скобками после нее). Например, строка "**disappear()"** запустит функцию исчезновения платформы. Такая строка называется **вызовом функции**.

Вызываем функцию исчезновения в строке 6 нашего скрипта:

```
1   local platform = script.Parent
2   local function disappear()
3       platform.CanCollide = false
4       platform.Transparency = 1
5   end
6   disappear()
```

Протестируйте созданный вами код, нажав кнопку **Play** в меню Roblox Studio. Если ваш код работает, платформа уже должна исчезнуть к тому времени, когда пользователь загрузится в игру.

Создаем еще одну функцию

Теперь вы можете легко заставить платформу снова появиться, написав функцию, которая делает полную противоположность функции исчезновения. Удалите из скрипта последнюю (шестую) строку **disappear()** и объявите новую функцию с именем «**appear**». В теле функции задайте для свойства **CanCollide** значение **true**, а для свойства **Transparency** — значение “**0**”.

```
1    local platform = script.Parent
2    local function disappear()
3        platform.CanCollide = false
4        platform.Transparency = 1
5    end
6    local function appear()
7        platform.CanCollide = true
8        platform.Transparency = 0
9    end
```

Организация циклического исполнения программы

Платформа должна постоянно исчезать и появляться снова, с интервалом в несколько секунд. Для такого рода действий существует специальная команда **while true**. Это — так называемый “вечный цикл”, который выполняет содержащийся в нем “внутренний” код постоянно. Обратите внимание на отступ “внутреннего кода цикла”, он должен быть таким же, как и отступы внутри функций.

```
1    while true do
…        “внутренний код цикла”
N    end
```

Кстати, существует цикл **while**, который исполняется не “вечно”, а только при выполнении какого-либо условия, например, пока некоторая переменная принимает значения меньше или больше заданного ей значения:

```
1    while i<50 do
…        “внутренний код цикла”
N    end
```

В приведенном сейчас цикле “внутренний код” исполняется только, если переменная **i** имеет значения меньшие, чем “50”. Обычно, во “внутреннем коде” таких циклов выполняются какие-то операции, из-

меняющие и значение переменной, величина которой определяет нужно ли выполнять цикл. Например, если исходное значение переменной **i** было равно “0”, а “внутренний код” указанного выше цикла содержит строку `i = i + 1`, то остальные команды, указанные во “внутреннем” коде цикла, будут выполнены 50 раз подряд.

Функция ожидания

Очевидно, что нужно, чтобы было несколько секунд между исчезновением и новым появлением платформы над пропастью. Для этого используют “функцию ожидания” **wait(3)**, в скобках которой указано время ожидания в секундах. Важно: никогда не создавайте цикл **while true** без включения в него “функции ожидания” — иначе ваша игра зависнет.

Конечный вид программы

```
1   local platform = script.Parent
2   local function disappear()
3       platform.CanCollide = false
4       platform.Transparency = 1
5   end
6   local function appear()
7       platform.CanCollide = true
8       platform.Transparency = 0
9   end
10  while true do
11      wait(3)
12      disappear()
13      wait(3)
14      appear()
15  end
```

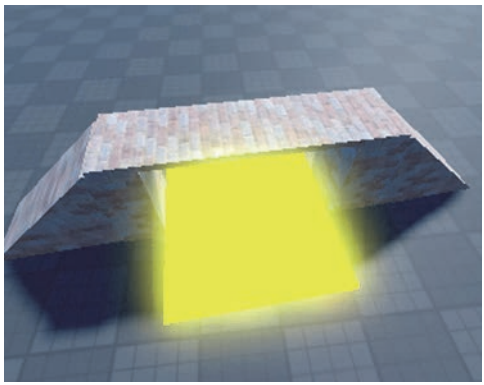
Наша программа содержит:

— привязку созданного скрипта к **p_Brige**;

— описание двух функций;

— вечный цикл **while true**, осуществляющий поочередный вызов созданных функций через 3-секундный интервалы времени.

7. Программируем смертельную лаву



В этой главе мы программируем смертоносную лаву, которая будет убивать игроков, при касании.

Рис. 7.1 Смертельная лава

“Залейте” смертоносную лаву под наш мост — под исчезающей платформой. Вставьте для этого новую **part**-платформу в нужное место. Назовите ее **Lava**. Сделайте ее более похожей на лаву, задав в свойствах **part** для **Material** значение **Neon**(Неон), а для свойства **Color** — **Deep Orange** (ярко-оранжевый цвет).
Вставьте в **Lava** новый скрипт и переименуйте его, например, в **L_kill.** Удалите исходный код, вставьте в новый скрипт определение переменной **lava** и создайте в заготовку функции **kill**:

```
1    local lava = script.Parent
2    local function kill()
…            “встроенный код функции”
N    end
```

Теперь нам необходимо определить, когда игроки касаются лавы.

Индикатор событий типа “Touched”
У каждой переменной, соответствующей какой-то **part,** есть индикатор событий **Touched**, который срабатывает, когда к этой **part** что-то прикасается. Например, если **part A** сталкивается с **part B**, то индикаторы **Touched** срабатывают у обоих **part**. Индикатор события срабатывает только в результате физического движения. Это означает, что по крайней мере одна из задействованных **part** не должна быть **Anchored** во время столкновения.

Можно использовать этот индикатор**,** чтобы он запускал какие-то функции. Для этого нужно указать этот индикатор после имени **part**, поставить знак двоеточие (“**:**”), указать **Connect** (Подключить) и ука-

зать (в скобках) название функции, которую вы хотите подключить к этому индикатору события:

```
…    lava.Touched:Connect(kill)
```

Теперь код, который вы напишете в функции “**kill()**”, будет выполняться всякий раз, когда что-то касается лавы.

Узнаем, кто коснулся лавы

Индикатор **Touched** может сообщить какая другая часть (**other part**) коснулась его, если сделать **other part “**параметром” функции, которую вы подключили к индикатору **Touched.**

Создадим параметр с именем **otherPart** для функции “**kill()**”:

```
1    local lava = script.Parent
2    local function kill(otherPart)
…            “встроенный код функции”
…    end
…    lava.Touched:Connect(kill)
```

Теперь, когда будет вызываться функция **kill()**, объект **otherPart** будет содержать **part**, которая коснулась лавы, и код, который вы напишете в функции **kill()**, сможет использовать эту информацию.

Идентификация игроков и гуманоидов

Roblox может определить конкретную часть тела игрока (левую руку или правую ступню), которая коснулась чего-либо. Эта часть тела является одной из составных частей объекта **Character**(Персонаж) игрока, включающего в себя:

- отдельные части тела игрока (голову, конечности и туловище);
- одежду и аксессуары, которые носит игрок;
- специальный объект **Humanoid**, который содержит множество свойств, связанных с игроком, включая его здоровье **Health**;
- объект **HumanoidRootPart** , который управляет движением игрока.

Зная, как устроен объект **Character**, мы должны найти не руку или ногу, а именно его, а потом в нем найти объект **Humanoid**, содержащий свойство **Health**(Здоровье), чтобы изменить его значение. Для этого, нужно создать переменную **partParent** и сохранить в ней Родителя

otherPart (то есть, сохранить в ней **Character** игрока). Сделаем это в строке 3, задав переменной **partParent** значение **otherPart.Parent**:

```
1   local lava = script.Parent
2   local function kill(otherPart)
3       local partParent = otherPart.Parent
4   end
5   lava.Touched:Connect(kill)
```

Теперь, чтобы убить игрока нам нужно в найденном **Character** найти объект “**Humanoid**”. Это можно сделать с помощью функции поиска в “родительской” модели ее “детей” — **FindFirstChild**. Укажите имя того, что вы ищете, в качестве параметра **FindFirstChild** и она найдет нужный “детский” элемент, который содержится в “родительском” объекте. Поэтому, подключаем к переменной **partParent** функцию поиска **FindFirstChild** с параметром “**Humanoid**” и сохраняем результат в новой переменной с именем **humanoid**:

```
1   local lava = script.Parent
2   local function kill(otherPart)
3       local partParent = otherPart.Parent
4       local humanoid = partParent:FindFirstChild("Humanoid")
5   end
6   lava.Touched:Connect(kill)
```

Является ли то, что коснулось лавы гуманоидом?

Вы можете легко проверить это, используя **оператор if (если)**. Код в приведенный внутри оператора **if - then** будет выполняться только в том случае, если проверяемое в этом операторе условие истинно (**then** означает “тогда”).

```
1   if условие верно then
…       (код, выполняемый если условие верно)
…   end
```

Обратите внимание: код выполняемый, если условие верно, записывается также с отступом вправо, соответствующим одному нажатию кнопки “Tab”.

Конечный вид программы смертельной лавы:

```
1    local lava = script.Parent
2    local function kill(otherPart)
3        local partParent = otherPart.Parent
4        local humanoid = partParent:FindFirstChild("Humanoid")
5        if humanoid then
6            humanoid.Health = 0
7        end
3    end
4    lava.Touched:Connect(kill)
```

Вы видите, что шестая строка нашего скрипта присваивает свойству **Health** переменной **humanoid** значение “0”, убивая аватар игрока.

8. Математические и логические действия

Без математики — просто никуда!

Математические операции	
x + 2	сложение
x - 3	вычитание
x*5	умножение
x/4	деление
Операции сравнения	
a ~= b	проверка неравенства
a > b	a больше b
a < b	a меньше b
a >= b	a больше или равно b
a == b	проверка равенства a и b
a <= b	a меньше или равно b

Для выполнения математических действий с переменными и проверки логических условий используют следующие операторы:

Рис. 8.1 Математические и логические операторы

Ниже приведена тестовая программа, демонстрирующая то, как работают перечисленные в таблице операторы. Создайте какой-нибудь **part**, заведите для него **script** с названием **math_1**, загрузите в этот скрипт тестовую программу, откройте окно вывода информации **Output** и посмотрите в этом окне, каковы будут результаты выполнения программы:

```
a = 1                          -- присваиваем переменной a значение “1”
b = 4                          -- присваиваем переменной b значение “4”
c = b/a                        -- объявляем, чему равна переменная c
print ("a=",tostring(a))       -- печатаем текст “a=”, а за ним значение a
        -- функция tostring(a) преобразует число a в текст - для печати
print ("b=",tostring(b))       -- печатаем текст “b=”, а за ним  значение b
print ("c=b/a=",tostring(c))   -- печатаем “c=b/a=”, а за ним значение c
if a~=b then                   -- проверка условия “a не равно b”
      print ("a не равно b") -- условие выполнено, печатаем “a не равно b”
end                            -- конец оператора if - then
if a<b then                    -- проверяем верно ли условие a < b
      print ("a < b")          -- поскольку условие выполнено печатаем “a<b”
end                            -- конец оператора if - then
a = a + 1                      -- прибавляем “1” к величине переменной a
c = b/a                        -- вычисляем новую величину переменной c
print ("a=a+1=",tostring(a))   -- печатаем текст “a=a+1=”, а за ним значение a
print ("b=",tostring(b))       -- печатаем текст “b=”, а за ним значение b
print ("c=b/a=",tostring(c))   -- печатаем текст “c=b/a=”, а за ним значение c
a = a*2                        -- вычисляем новое значение переменной a
c = b/a                        -- вычисляем новое значение величины c
print ("a=a*2=",tostring(a))   -- печатаем “a=a*2=”, а за ним значение a
print ("b=",tostring(b))       -- печатаем текст “b=”, а за ним значение b
print ("c=b/a=",tostring(c))   -- печатаем текст “c=b/a=”, а за ним значение c
if a==b then                   -- проверяем верно ли условие a == b
      print ("a = b")          -- поскольку условие выполнено печатаем “a=b”
end                            -- конец оператора if - then
a = a - 1                      -- вычисляем новое значение величины a
c = b/a                        -- вычисляем новое значение величины c
print ("a=a-1=",tostring(a))   -- печатаем “a=a-1=”, а за ним значение a
print ("b=",tostring(b))       -- печатаем текст “b=”, а за ним значение b
print ("c=b/a=",tostring(c))   -- печатаем текст “c=b/a”, а за ним значение c
```

Обратите внимание на то, что в правой части окна **Output** имеется движок “лифта” — ведь высота окна ограничена и просмотреть всю его информацию сразу невозможно.

Вы заметили, что мы, вдруг, не поставили порядковые номера строк программы, но в строках появились пояснения-комментарии. Это потому, что и строки кода и комментарии к ним в этой тестовой программе очень короткие. Мы рекомендуем вам также вставлять коммента-

рии в строки ваших программ, тем более, что ширина строк скриптов Roblox Studio гораздо шире, чем ширина строк в книге. Комментарии пригодятся, если вы через месяц-два или еще больший срок захотите модифицировать ранее созданную программу. Тогда, наличие комментариев позволит быстрее вспомнить, как ваша программа работает. Перед каждым комментарием обязательно должны стоять два тире "-- ", иначе, Lua воспримет ваш текст, как ошибочный.

Конечно же, в таблице рис.4.1 приведены не все виды математических операций. Целый ряд операций Lua "упакован" в специальную математическую библиотеку **math**, название которой нужно указывать, когда вы пользуетесь этими функциями. Приведем несколько функций, которые мы используем в этой книге:

math.random (m,n) — функция генерирующая случайное целое число в диапазоне от **m** до **n**;

math.rad(i) — функция пересчитывающая в радианы угол **i**, заданный в градусах;

math.sin(i) — функция рассчитывающая значение синуса для угла **i**, заданного в радианах;

math.cos(i) — функция рассчитывающая значение косинуса для угла **i**, заданного в радианах.

9. Программируем паркур

Вы научились создавать строительные блоки и задавать им нужные вам свойства. Давайте применим полученные навыки для создания игры.

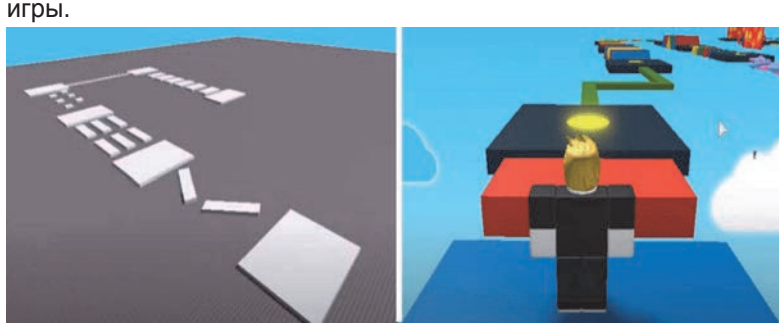


Рис. 9.1 Разработка паркура

Сначала разложите элементы вашего паркура прямо на **Baseplate**. Раскрасьте платформы паркура, чтобы они выглядели привлекательнее. Затем, в меню **Explorer** наведите курсор мыши на **Baseplate**, нажмите правую кнопку мыши и выберите в выпадающем меню опцию **Delete** — **Baseplate** будет удалена, а созданные вами платформы “повиснут” в небе (если, конечно, вы “заякорили” их — иначе, все они упадут в бездну).

Но, то что у вас получилось — очень простой паркур. Можно, конечно, несколько изменить положение и ориентацию в пространстве отдельных его площадок используя свойства **Orientation** в окне **Properties**, но это не слишком увеличит сложность его прохождения. Другое дело, если заставить платформы паркура двигаться прямо в процессе игры. Давайте разбираться, как это сделать.

Первое, что нужно знать: в каждой игре Roblox позиция X=0, Y=0, Z=0 всегда находится в центре игрового пространства и совпадает с местом спавна игрока.

Для определения положении и ориентации любых **parts** в 3D-пространстве игры можно использовать данные типа **Vector3**, которые используют те же координаты: X, Y и Z. Поясним это на примере простой программы, скрипт которой нужно вставить, например, в заранее созданный в **Workplace** объект-платформу **obby_p**:

```
1   local cf_part = script.Parent
2   x,y,z=2,8,2
3   cf_part.Position = Vector3.new(x,y,z)
4   cf_part.Size =Vector3.new(3,0.5,3)
5   cf_part.BrickColor = BrickColor.new (255,0,0)
6   while true do
7       cf_part.Position = Vector3.new(x,y,z)
8       cf_part.Orientation=Vector3.new(x*45,y,z)
9       wait(1)
10      x = x +1
11  end
```

— в строке 1 создаем переменную **cf_part** и указываем ее Родителя **obby_p**;
— в строке 2 задаем исходные значения сразу трем переменным;

— в строке 3 задаем при помощи **Vector3** позицию **obby_p** (указав значение свойства **Position** объекта **cf_part**) — теперь его позиция будет соответствовать тем координатам, которые указаны в строке 2;

— в строке 4 задаем размеры **cf_part** (указав при помощи **Vector3** свойству **Size** размеры 3 x 0,5 x 3 шипа);
— в строке 5 красим **obby_p** в красный цвет;
— цикл **while true** в строках 6…11 меняет не только позицию, но и ориентацию **obby_p** за счет изменения угла ее поворота вокруг оси X.



Ведь через каждую секунду значение переменной X меняется на “1”, а значит, меняется значение координаты X в свойствах **Position** и **Orientation**.

Рис. 9.2 Поворот платформы

Запустите программу, вы увидите, что **obby_p** меняет свое местоположение на игровом поле и, одновременно, поворачивается на 45 градусов при каждом перемещении вдоль оси X. Теперь вы можете, например, используя цикл **while**, заставить некоторые из платформ паркура качаться то в одну, то в другую сторону, да еще наклоняться, усложняя тем самым прохождение паркура. Продублируйте **obby_p** и откорректируйте скрипт новой **part**:
— измените ее цвет;
— измените ее положение (чтобы **parts** не мешали друг другу);
— измените “содержимое” цикла **while true**, а именно:

```
1    local cf_part = script.Parent
2    x,y,z=2,4,2
3    i=0
4    cf_part.Position = Vector3.new(x,y,z)
5    cf_part.Size =Vector3.new(3,0.5,3)
6    cf_part.BrickColor = BrickColor.new(0,200,0)
7    while true do
8        while i<50 do
9            cf_part.Position = cf_part.Position + Vector3.new(0.1,0,0)
10           cf_part.Orientation= cf_part.Orientation + Vector3.new(2,0,0)
11           wait(0.2)
```

```
12                  i = i +1
13              end
14              i=0
15              while i<50 do
16                  cf_part.Position = cf_part.Position + Vector3.new(-0.1,0,0)
17                  cf_part.Orientation= cf_part.Orientation + Vector3.new(-2,0,0)
18                  wait(0.2)
19                  i = i +1
20              end
21              i=0
22      end
```

Запустите игру. Согласитесь, пройти паркур с такими платформами будет гораздо сложнее.

Давайте разберемся с новым скриптом:
— мы видим, что в обоих циклах **while i<50** переменная **i** изменяется от “0” до “50”;
— с каждой итерацией(с каждым шагом) цикла положение **obby_p** по оси X изменяется либо на “0,1”, либо на “-0,1”, а угол поворота **obby_p** вокруг оси X изменяется на “2” либо на “-2”;
— после каждой итерации программа приостанавливается на время равное 0,2 секунды;
— получается, что значения “0.1” и “-0.1” соответствуют понятию “скорость”, а максимальное значение переменной **i** — понятию времени, затраченного на движение платформы. В результате, путь пройденный **obby_p**, можно выразить формулой: S = V*t = 0,1 * 50 = 5 шипов. Зная это, вы всегда можете рассчитать нужный вам диапазон перемещения платформ по оси X;
— аналогично, вы можете заранее рассчитать необходимые вам углы поворота платформ;
— кроме того, меняя “скорость” и “шаг” изменения времени (**i**) мы можем изменять скорость движения и смены ориентации платформ.

А почему мы ограничились изменением положения только по оси X? Можно, аналогично, менять положение и ориентацию платформ паркура по другим осям. В том числе, одновременно. Обязательно попробуйте это сделать.

Чертово колесо

Но и это еще не все. Вспомните как движутся кабинки “чертова колеса”, а что, если сделать, чтобы и некоторые ваши платформы двигались так же?

Давайте, попробуем. Соорудите еще одну платформу, например, **part_sin**, вставьте в нее скрипт, приведенный ниже и проверьте, как это работает. Вы увидите, что цель достигнута.



Рис. 9.3 Вариант кругового движения платформы паркура

```
1    local part_sin = script.Parent
2    i=0
3    x,y,z=12,3,12
4    part_sin.Position = Vector3.new(x,y,z)
5    part_sin.Size =Vector3.new(6,0.5,6)
6    part_sin.BrickColor = BrickColor.new (0,0,250)
7    while true do
8        i_rad=math.rad(i)
9        part_sin.Position = part_sin.Position + Vector3.new(math.sin
  (i_rad)/2,math.cos(i_rad)/2,0)
10       wait(.2)
11       i = i + 10
12   end
```

Поясним скрипт:

— в строке 3 мы изменили исходные координаты платформы;

— в строке 5 увеличили размеры платформы;

— в строке 6 изменили цвет платформы на синий;

— в строке 8 (внутри бесконечного цикла **while true**) мы преобразуем числовое значение переменной **i** (которая у нас будет выполнять роль

величины угла поворота платформы) в радианы, используя для этого уже знакомую математическую функцию **math.rad**;
— в строке 9 мы (при помощи **Vector3.new**) наращиваем исходную позицию платформы **part_sin** на величину, соответствующую:

- по оси координат X — величине синуса угла **i**, деленную на “2”;
- по оси координат Y — величине косинуса угла **i**, деленную на “2”.

Получается, что переменная **i** будто бы бесконечно растет, но вы же представляете, что если крутишь что-либо, угол поворота этой вещи всегда меняется только в пределах от “0” до “360” градусов. Так и здесь. Кстати, вы можете попробовать (и обязательно сделайте это) изменить число “10”, которое прибавляется к прежнему значению переменной **i** и число “2”, на которое делятся синус и косинус. Меняя эти параметры, вы можете заставить платформу вращаться быстрее или медленнее, увеличивать или уменьшать диаметр круга по которому вращается платформа. Можно не только делить значения синуса и косинуса на какой-то коэффициент, но и умножать. Проверьте. Только поднимите платформу повыше, иначе она будет цеплять за **Baseplate** и обратите внимание на параметр **CanCollide**.

Изменяем положение объектов при помощи CFrame

Но, продолжим разбираться с положением и ориентацией **parts**. Вы уже неоднократно заглядывали в окно **Properties** строительных блоков и помните, что в этом окне их положение в игровом пространстве описывает параметр, который называется **CFrame**. Действительно в Lua есть еще один “инструмент”, позволяющий задать в скрипте любой **part** нужное вам положение и ориентацию. Для этого можно использовать выражение:

```
cf_part.CFrame = CFrame.new(X, Y, Z) * CFrame.Angles(math.rad(n1), math.rad(n2), math.rad(n3)),
```

где **X,Y,Z** — нужные вам координаты центра **obby_p**,
Angles указывает на изменение ориентации **obby_p** в пространстве;
n1, n2, n3 — углы поворота **obby_p** в пространстве,
соответствующие осям координат X,Y,Z;
math.rad — функция, преобразующая угол **из градусов в радианы**.

Если же менять ориентацию **part** в пространстве не нужно, то можно использовать более короткое выражение:

```
cf_part.CFrame = CFrame.new(X, Y, Z)
```

Поставьте в строках 7 и 8 скрипта первого из ранее созданных **obby_p** по два тире (как для вставки комментариев), чтобы эти строки перестали действовать. А вместо них вставьте новую строку кода:

```
cf_part.CFrame = CFrame.new(x,y,z) * CFrame.Angles(math.rad(x*45), 0, 0)
```

Запустив программу, вы легко убедитесь, что поведение **obby_p** нисколько не изменилось. То есть, в данном случае конструкции **Vector3** и **CFrame** действуют полностью аналогично. Теперь, *самостоятельно*, попробуйте при помощи CFrame вращать платформу **part_sin**, как в “чертовом колесе”.

Что же лучше использовать **Position** или **CFrame** для управления перемещениями **parts**? В описанных выше скриптах мы перемещали простые **parts**. Но, часто требуется перемещать одновременно группы объектов, скрепленных друг с другом (состоящие из нескольких “сваренных” деталей). Если **Position** одной из сваренных деталей изменять, эта деталь будет двигаться, но ни одна из соединенных с ней деталей не будет двигаться вместе с ней. Напротив, если изменяется **CFrame** одной из деталей, любой объект, приваренный к этой детали, также будет двигаться.

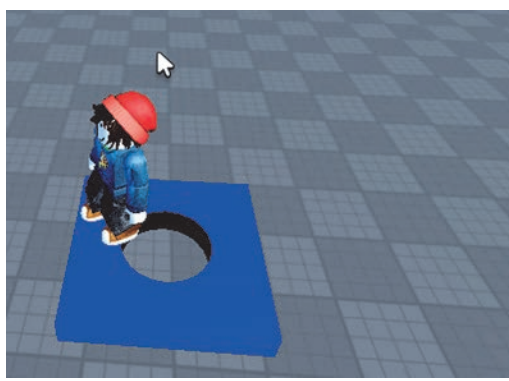


Рис. 9.4 Платформа с отверстием-ловушкой

Что еще можно предложить, чтобы сделать ваш паркур еще круче? А давайте вырежем в некоторых платформах отверстия, через которые игроки могут провалиться при неудачном прыжке. Вот скрипт такой платформы, перемещающейся, как кабинка “чертового колеса”:

```
1    local part_sin = script.Parent
2    i=0
```

```
3       x,y,z=12,6,12
4       part_sin.Position = Vector3.new(x,y,z)
5       part_sin.Size =Vector3.new(6,1,6)
6       part_sin.BrickColor = BrickColor.new (0,0,250)
7       part_sin.Anchored = true
8       local part_2 = Instance.new("Part", workspace)
9       part_2.Shape = "Ball"
10      x1,y1,z1=12,6,12
11      part_2.Position = Vector3.new(x1,y1,z1)
12      part_2.Size =Vector3.new(3,3,3)
13      part_2.Anchored = true
14      new_Union = part_sin:SubtractAsync({part_2})
15      new_Union.Parent = game.Workspace
16      new_Union.Position = part_sin.Position
17      new_Union.Anchored = true
18      part_sin:Destroy()
19      part_2:Destroy()
20      while true do
21              i_rad=math.rad(i)
22              new_Union.Position = new_Union.Position + Vector3.new
   (math.sin(i_rad)/5,math.cos(i_rad)/5,0)
23              wait(.2)
24              i = i + 3
25      end
```

Вы видите, что в строках 1…7 нет никаких изменений в сравнении с прежним скриптом, описывающим **part_sin**. А вот в строке 8 появилось что-то новое. Оказывается, в языке Луа новые (самые разные) объекты можно создавать внутри скрипта с помощью функции **Instance.new()** — где в скобках нужно указывать тип создаваемого объекта.

Именно так в строке 8 (указав тип объекта “**Part**” и указав его Родителя **workspace**) мы создаем объект **part_2**, а далее:
— в строке 9 определяем, что формой этого объекта является шар (**Ball**);
— в строке 10 задаем значения трем переменным x1, y1, z1, которые в следующей строке используем в качестве координат, в которые поместим наш новый объект. Обратите внимание: значения координат совпадают со значением координат x, y, z — значит, центры **part_sin** и **part_2** совпадают. В этом случае, можно было бы использовать в

строках 4 и 11 одни и те же переменные, но мы решили показать, как быть, если вам будут нужны другие координаты;
— в строке 12 задаем размер **part_2** равным 3 x 3 x 3 шипа;
— в строке 13 “якорим” объект;
— в строке 14 создаем объект **new_Union**, представляющий собой “разность” фигур **part_sin** и **part_2**. То есть, вырезаем в платформе **part_sin** отверстие с диаметром равным диаметру шара **part_2**. Обратите внимание на фигурные скобки, в которых указан объект **part_2**. Фигурные скобки означают, что в них может быть указан список. Если бы вам нужно было “вычесть” из **part_sin** не один, а несколько объектов, вы могли бы указать их в этих фигурных списках через запятую;
— в строке 15 указываем Родителя **new_Union;**
— в строке 16 указываем, что положение **new_Union** в игровом пространстве совпадает с положением **part_sin**;
— строка 17 “якорит” новый **Union**. Получается, что мы программным способом создали твердотельную модель, и элементы, из которых она состоит нам уже не нужны;
— в строках 18, 19 уничтожаем ненужные нам **part_sin** и **part_2**, используя функцию **Destroy()**;
— строки бесконечного цикла **while true** 20…25 управляют движением объекта **new_Union**, точно также, как раньше управляли платформой **part_sin**. Теперь игрокам будет труднее, ведь они могут провалиться сквозь отверстие в платформе.



Рис. 9.5 Преграда, качающаяся из стороны в сторону или вращающаяся и мешающая тем самым прыжку игрока с платформы на платформу

Согласитесь, можно вставить в паркур не только платформы, но и преграды, мешающие прыжкам игроков. И теперь вы знаете, как все это сделать.

Чек-пойнты для сложного паркура

Смотрите, вы уже можете создать довольно сложный паркур. Скорее всего, игрокам будет трудно пройти его, и нужно будет предпринимать несколько попыток, чтобы пройти его до конца. И, согласитесь, им бу-

дет обидно по несколько раз проходить уже пройденные участки паркура. Лучше, если бы можно было (после респавна) вернуться, как можно ближе к тому месту, преодолеть которое игроку не удалось. Для этого, нужно вставить в ваш паркур специальные платформы, которые называются **chekpoints** (“чек-пойнтс”) — контрольные точки.

Давайте создадим **part** “чек-пойнт” — платформу, которая будет визуально немного отличаться от всех других. Теперь можно создать несколько копий такой платформы и расставить их в паркуре. А вот скрипт “чек-пойнта”:

```
1   local Players = game:GetService("Players")
2    local spawn = script.Parent
3    spawn.Touched:Connect(function(hit)
4           if hit and hit.Parent and hit.Parent:FindFirstChild("Humanoid") then
5               local player = game.Players:GetPlayerFromCharacter
  (hit.Parent)
6               local checkpointData = game.ServerStorage:FindFirstChild
  ("CheckpointData")
7               if not checkpointData then
8                   checkpointData = Instance.new("Model", game.Server
  Storage)
9                   checkpointData.Name = "CheckpointData"
10              end
11              local checkpoint =
                 checkpointData:FindFirstChild(tostring(player.userId))
12              if not checkpoint then
13                  checkpoint = Instance.new("ObjectValue",
   checkpointData)
14                  checkpoint.Name = tostring(player.userId)
15                  checkpoint.Value = spawn
16                  player.CharacterAdded:Connect(function(character)
17                      wait()
18                      character:WaitForChild("HumanoidRootPart").
   CFrame =  game.ServerStorage.CheckpointData[tostring(player.userId)].
   Value.CFrame + Vector3.new(0, 4, 0)
19                  end)
20              end
21          end
22  end)
```

Мы приводим здесь скрипт “чек-пойнта”, чтобы не разрывать рассказ об игре паркур. Скопируйте и используйте скрипт. А к объяснению скрипта которое мы приводим сейчас, вернитесь еще раз, когда дочитайте книгу до главы 16. Итак, поясняем:

— в строке 1 открываем доступ к имеющемуся в Roblox сервису **Players**. Этот сервис позволит нам получать информацию об игроках, участвующих в игре;

— в строке 2 создаем переменную **spawn** и указываем ее Родителя (наш “чек-пойнт”);

— в строке 3 подключаем к индикатору события **Touched** “чек-пойнта” функцию без имени (строки 3…22) с параметром **hit** (касание), в которой:

— в цикле **if - then** (строки 4…21):

— в строке 4 проверяем:

* что-то коснулось “чек-пойнта”?
* найден Родитель того, что коснулось? Заметим, что если “чек-пойнта” коснулся аватар, то Родителем того, что коснулось “чек-пойнта” должен быть объект **character**;
* можно ли с помощью функции **FindFirstChild** найти в Родителе объект типа “**Humanoid**” (который как раз и находится в **character**)?

И, если все три условия выполняются, то:

— в строке 5 с помощью сервиса **Players** и функции **GetPlayerFrom Character** (ПолучитьИгрокаИзМоделиПерсонажа) определяем игрока, который коснулся “чек-пойнта” и присваиваем все данные этого игрока новой локальной переменной **player**;

— в строке 6 с помощью функции **FindFirstChild**(НайтиПервогоПотомка) ищем данные с именем "**CheckpointData**" в хранилище сервера с именем **ServerStorage**. Эти данные должны хранить информацию о том, касался ли кто-то “чек-пойнтов” или нет;

— если с помощью оператора **if-then** в строке 7, мы обнаружим, что данных с именем "**CheckpointData**"**,** которые программа искала в строке 6 нет, то:

— в строке 8 с помощью **Instance.new** создаем *в серверном хранилище* **ServerStorage** контейнер (“**Model**”) для хранения таких данных и присваиваем его значение переменной **chekpointData**;

— в строке 9 присваиваем свойству **Name**(Имя) созданной переменной **chekpointData** значение “**CheckpointData”**;

— в строке 11 с помощью функции **FindFirstChild**(НайтиПервого Потомка) ищем в контейнере **chekpointData** данные с именем, совпадающим с уникальным номером игрока **player.userId**, который коснулся “чек-пойнта”. А поскольку, имя — это переменная строковая, мы используем функцию **tostring** для перевода уникального номера игрока в строчное значение. Если нужные данные найдены, они присваиваются переменной **chekpoint**;

— но, если при проверке оператором **if-then** в строке 12 окажется, что данные, которые программа искала в строке 11 в контейнере **CheckpointData**, отсутствуют, то:

— в строке 13 создаем в контейнере **CheckpointData** объект **checkpoint**, для хранения таких данных;

— в строке 14 присваиваем свойству **Name** переменной **chekpoint** имя, соответствующее уникальному номеру игрока **player.userId**;

— в строке 15 передаем свойству **Value** переменной **chekpoint** значение переменной **spawn** — включающие в себя координаты **part** “чекпойнта”;

— в строке 16 к событию **player.CharacterAdded** (которое создает игрока, когда игрок возрождается) подключаем функцию без имени с параметром **character** которая содержит две команды;
— в строке 17 дает небольшую задержку;
— в строке 18 с помощью функции **WaitForChild**(ЖдатьПоявленияДитя) ожидает момента, когда в **character** возрождаемого игрока появится объект **HumanoidRootPart**. Как только он появляется, ему при помощи

CFrame задаются координаты “чек-пойнта”, которые теперь хранятся в хранилище сервера в контейнере, соответствующем уникальному номеру игрока.

То есть, теперь, когда будет происходить возрождение игроков, им будут присваиваться координаты “последних” спавнов, которых они касались в игре, и игроки будут появляться гораздо ближе к тому месту, где они потерпели неудачу. Обратите внимание: в строке 18 **Vector3.new(0, 4, 0)** приподнимает игрока над центром “чек-пойнта” на 4 шипа — игрок, как бы, спрыгивает на “чек-пойнт” с небольшой высоты (это, практически, невозможно заметить, но понимать это нужно).

Самые внимательные читатели скажут: “Но, если в хранилище сервера еще не было данных об игроке, то программа только создаст контейнеры для хранения данных игрока и завершится не сообщив событию **CharacterAdded** координат в которых игрок должен возродиться!”

Согласимся с ними, но приведем и свой аргумент: “Игрок обязательно сделает по “чек-пойнту” не один, а хотя бы два шага, то есть, индикатор **Touched** “чек-пойнта” сработает, как минимум, дважды. При первой сработке контейнеры для хранения данных будут созданы, а при второй — программа сохранит в эти контейнеры нужные данные”.

Заметим, что, скорее всего, вам будет трудно сразу понять приведенные пояснения к скрипту — вернитесь к этим пояснениям еще раз и перечитайте, после того, как разберетесь с сохранением данных в хранилище сервера. Подробнее о хранении данных на сервере мы расскажем чуть позже — в другой главе книги.

10. Программируем кнопки управления

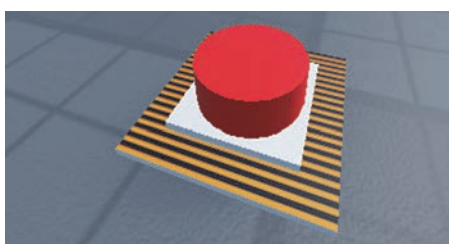


Рис. 10.1 Кнопка управления

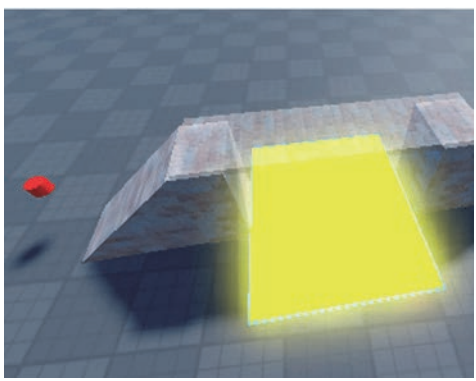
Кнопки удобно использовать, чтобы разблокировать в игре какие-то двери, или открывать сейфы с сокровищами. Всякий раз, когда игрок касается кнопки,

он должен получать “сигнал обратной связи” — изменение ее цвета или звуковой сигнал, или еще что-то подтверждающие нажатие кнопки.

“Висячая” кнопка

Давайте запрограммируем кнопку, активирующую мост над лавой. Если кнопка будет нажата, пусть она станет зеленой и мост станет пригодным для прохода. Если же игрок не нажмет кнопку, то, ступив на мост, он провалится в лаву.

Приступаем. Снимите верхнюю пластину нашего моста и перенесите ее в другое место игрового пространства (пусть полежит - используете ее потом). Создайте вместо нее новую **part** платформы и переименуйте ее в **bridge** (мост). Откройте окно **Properties** моста и измените его параметры, установив:



— Transparency — 0.5;
— Anchored — “V”;
— CanCollide — “галочка” (V) должна быть снята.

Рис. 10.2 Мост с кнопкой

Запустите игру, мост должен выглядеть полупрозрачным, и игрок должен проваливаться, если наступает на платформу моста.

Создайте новую **part** и назовите ее **button** (Кнопка). Установите цвет кнопки красным и заякорите кнопку (**Anchored**). Чтобы кнопка не мешала движению игроков (если она окажется у них на пути) можно отключить ее свойство **CanCollide**. Переместите кнопку так, чтобы она ничего не касалась (чтобы индикатор события **Touched** не срабатывал). Вставьте в **Button** уже знакомый вам скрипт, использующий индикатор касаний **Touched**:

```
1    local button = script.Parent
2    local bridge = game.Workspace.bridge
3    local function buttonPressed()
4        print("button touched")
5        button.Color = Color3.fromRGB(0, 170, 0)
6        bridge.Transparency = 0
```

```
7           bridge.CanCollide = true
8           wait(3)
9           button.Color = Color3.fromRGB(255, 0, 0)
10          bridge.Transparency = 0.5
11          bridge.CanCollide = false
12      end
13      button.Touched:Connect(buttonPressed)
```

Напомним, что в строках 3…12 мы создаем функцию **buttonPressed** (КнопкаНажата), внутри которой:
— в строке 4 организуем печать (в окне **Output** Roblox Studio) текста “кнопка нажата”;
— в строке 5 меняем цвет кнопки на зеленый;
— в строке 6 делаем мост непрозрачным;
— в строке 7 делаем мост твердым (CanCollide);
— в строках 9…11 возвращаем исходные значения параметрам моста;
— в строке 13 подключаем индикатор события **Touched** к функции **buttonPressed**.

Запустите игру и проверьте работу моста с кнопкой управления. Если что-то не так, проверьте соответствуют ли названия **parts** моста и кнопки их названиям, указанным в программе.

Да, мы сделали платформу моста исходно чуть прозрачной. Другой возможный вариант — когда вы сделаете платформу исходно непрозрачной и способной пропускать по себе игроков, но какой-то (не бросающийся в глаза) предмет рядом с входом на мост будет работать кнопкой и, если игрок случайно его заденет, эта кнопка “сломает” платформу и игрок упадет в лаву, ровно через 1 секунду после того, как ступит на нее. Можете сами придумать и другие варианты сценария игры.

Согласитесь, кнопка, висящая в воздухе рядом с мостом — вещь странная. Но, ведь кнопка не обязательно должна иметь явный “кнопочный” вид. В качестве кнопки может работать любой объект: камень, ящик, столбик или даже дерево. Все эти предметы можно расположить на очень малой высоте — всем будет казаться, что они лежат, стоят на земле или даже растут из земли. Или пусть игрок, при отсутствии явно видимой кнопки управления, поищет и догадается, как ему преодолеть препятствие.

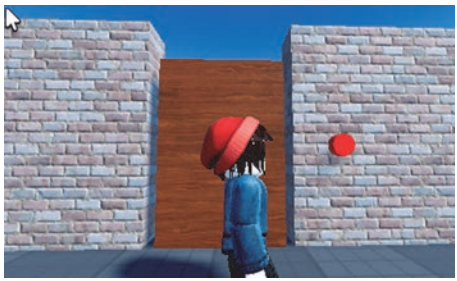


Рис. 10.3 Кнопка открывания двери

Но, вам могут потребоваться и “обычные” кнопки, например, кнопки компьютера, кнопка звонка двери, кодовой панели сейфа или что-то подобное. То есть, такие кнопки, которые обязательно касаются чего-либо. Как быть в этом случае? Вам поможет объект **ClickDetector**, который позволяет скриптам получать информацию о кликах мыши по игровым объектам или даже просто о наведении на эти объекты курсора мыши.

Событие **MouseClick** в **ClickDetector** срабатывает, когда игрок нажимает и отпускает левую кнопку мыши, в то время, когда указатель мыши (курсор) находится на **ClickDetector**. На сенсорных устройствах касание объекта с **ClickDetector** также запускает событие аналогичное щелчку мыши.

Для ограничения расстояния, на котором игрок может находиться от объекта когда **ClickDetector** перестанет быть активным можно использовать свойство **MaxActivationDistance** (МаксимальнаяДистанция Активации). Например, игрок в пределах 10 шипов от объекта с встроенным **ClickDetector** и **MaxActivationDistance**, равной “5”, не сможет кликнуть по **ClickDetector**, потому что он находится вне нужного диапазона. Однако, если бы **MaxActivationDistance** была равна “15”, аватар смог бы щелкнуть по нему.

Если вы хотите использовать событие, когда игрок кликает по объекту с **ClickDetector** правой кнопкой мыши, нужно использовать, соответственно, событие **RightMouseClick**.

Если вы хотите, чтобы **ClickDetector** срабатывал, когда игрок наводит на него указатель мыши или, наоборот, отводит с него курсор мыши, не кликая мышью, вы можете использовать события **MouseHoverEnter** и **MouseHoverLeave**.

Для примера, создадим две стены с дверью **Door** между ними, установим на стену рядом с дверью кнопку и назовем ее **Door_Bell** (КнопкаЗвонка). Эта кнопка будет управлять открыванием и закрыванием двери. Кстати выберите размер кнопки чуть длиннее, чем толщина стены — это позволит вам управлять дверью с обеих сторон стены. В **part** двери вставьте следующий скрипт:

```
1   local door = script.Parent
2   local ClickDetector = Instance.new("ClickDetector")
3   ClickDetector.Parent = game.Workspace.Door_Bell
4   ClickDetector.MaxActivationDistance = 10
5   ClickDetector.MouseHoverEnter:Connect(function()
6           door.Transparency = 0.1
7   end)
8   ClickDetector.MouseHoverLeave:Connect(function()
9           door.Transparency = 0
10  end)
11  ClickDetector.MouseClick:Connect(function()
12          door.Transparency = 0.8
13          door.CanCollide = false
14  end)
15  ClickDetector.RightMouseClick:Connect(function()
16          door.Transparency = 0.1
17          door.CanCollide = true
18  end)
```

Сделаем пояснения к скрипту:
— в строке 1 объявим переменную **door** и укажем ее “Родителя” (**Door**);
— в строке 2 создаем **Click Detector**;
— в строке 3 указываем “Родителя” **Click Detector**, то есть, объект который он должен контролировать (у нас он контролирует **Door_Bell**);
— в строке 4 задаем расстояние от **Door_Bell**, начиная с которого клик по ней будет вызывать сработку **Click Detector**;
— в строках 5…7 описана функция, которая при наведении курсора мыши на кнопку, будет увеличивать прозрачность двери, показывая, что кнопка связана именно с этой дверью, а не с чем-либо еще;
— в строках 8…10 описана функция, которая вновь делает дверь полностью непрозрачной, когда курсор мыши перестает указывать на кнопку;

— в строках 11…14 описана функция, которая при клике по **Door_Bell** левой кнопкой мыши делает дверь прозрачной и способной пропускать через себя любые другие объекты. Теперь через дверь могут продолжать проходить любые объекты, но, прозрачность двери не сохраняется, когда курсор мыши перестает указывать на кнопку — ведь работает функция строк 8…10;
— в строках 15…18 описана функция, которая при клике правой кнопкой мыши опять делает дверь непроницаемой, правда, чуть более прозрачной, пока на нее указывает курсор мыши.

Кнопки с Proximity Prompts
Объекты **Proximity Prompts** (подсказки игрокам о близости к чему-либо) обеспечивают взаимодействие игроков с кнопками, дверями и другими объектами, включаясь, когда игроки приближаются на определенное расстояние к таким объектам. Используя **Proximity Prompts**, вы можете:
— подсказывать игрокам какими объектами они могут управлять;
— подсказывать действие, которое игроки должны выполнить для этого.
Чтобы добавить **Proximity Prompts** к какому-либо объекту, нужно в окне **Explorer** навести курсор мыши на этот объект и нажать кнопку ⊕ — появится выпадающее меню. Выберите в меню опцию **Prompt** — в вашем объекте появится объект **Proximity Prompt**. Переименуйте его так, чтобы было понятно его назначение.

Затем, нужно настроить **Proximity Prompt.** Подсказки о близости должны сообщать о трех вещах:
1/ об объекте, с которым игрок может взаимодействовать;
2/ о том, что произойдет когда игрок произведет подсказанное действие;
3/ о клавише, которую игрок должен нажать и/или удерживать.

Все это можно указать в окне свойств **Proximity Prompt**:

— в свойстве **ObjectText** можно указать объект, с которым игрок может взаимодействовать;

— в свойстве **ActionText** можно подсказать действие, которое должен выполнить игрок;

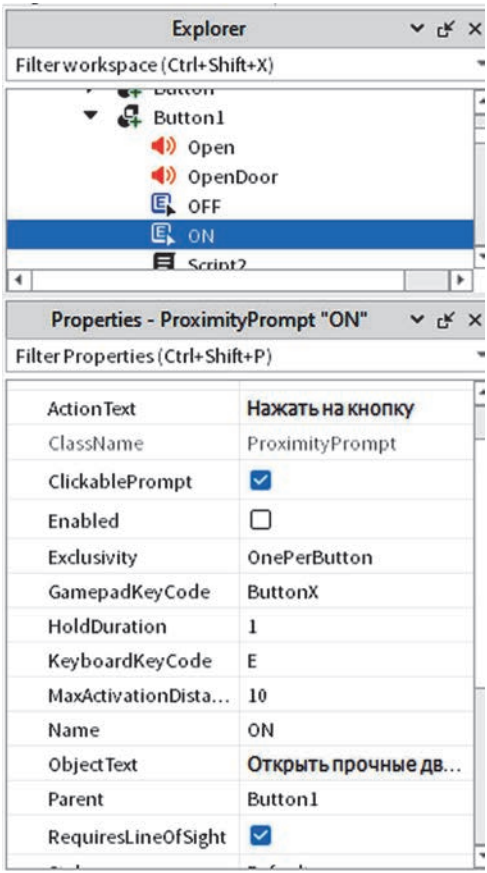


Рис. 10.4 Меню настройки Proximity Prompt

— в свойстве **Keyboardkeycode** нужно указать клавишу клавиатуры, которую пользователь должен нажать и/или удерживать, чтобы запустить действие;

— в свойстве **GamePadkeyCode** кнопку **GamePad**, которую игрок должен нажать и/или удерживать, чтобы запустить действие;

— в свойстве **MaxactivationDistance** можно определить расстояние от игрока, на котором подсказка близости будет активирована;

— если в нескольких подсказках будет использована одна и та же кнопка клавиатуры и в свойстве **Exclusivity** будет указано **OnePerbutton,** то только одна подсказка близости будет видна. Если же несколько подсказок близости используют разные кнопки, то все они будут видны, если их видит камера;
— если в свойстве **Exclusivity** будет указано **OneGlobally,** то только одна подсказка близости будет видна в любой момент времени, независимо от их близости или выбранных кнопок клавиатуры. Будет видна та подсказка, которую “видит” камера. Эту опцию полезно применять, чтобы исключить отвлечение игрока на просмотр нескольких подсказок;

— если в свойстве **Exclusivity** будет указано **AlwaysShow** — подсказка о близости будет видна всегда, если что-то не заслоняет ее от камеры;

— свойство **Holdduration** определяет, сколько секунд необходимо удерживать кнопку, прежде чем ее нажатие приведет к нужному результату. Если указать значение “0”, сработка управления произойдет сразу же;

— свойство **ClickablePrompt** указывает, может ли пользователь кликнуть и удерживать подсказку мышкой, чтобы запустить управление событием (можно выбрать **true** или **false**). Если же пользователь использует телефон или планшет, он всегда может взаимодействовать с подсказкой близости, щелкая по ней независимо от свойства **ClickablePrompt**.

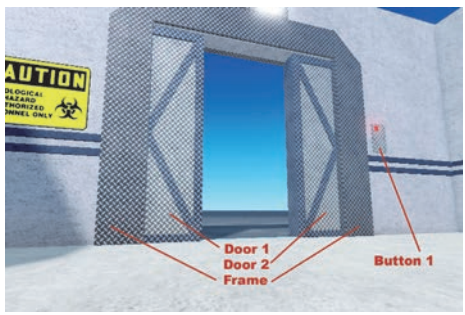


Рис. 10.5 Пример применения *Proximity Prompt* в конструкции раздвижных дверей

Приведем пример использования **Proximity Prompt**. Создайте что-то подобное тому, что изображено на рис. 10.5. Как это выглядит в окне **Explorer** показано на рис. 10.7.



Рис. 10.6. Вид подсказки о близости, включающейся при подходе игрока к кнопке Button1, управляющей открыванием/закрыванием раздвижных дверей (a) и индикатор времени удержания кнопки “E” перед началом открывания/закрывания дверей (b)

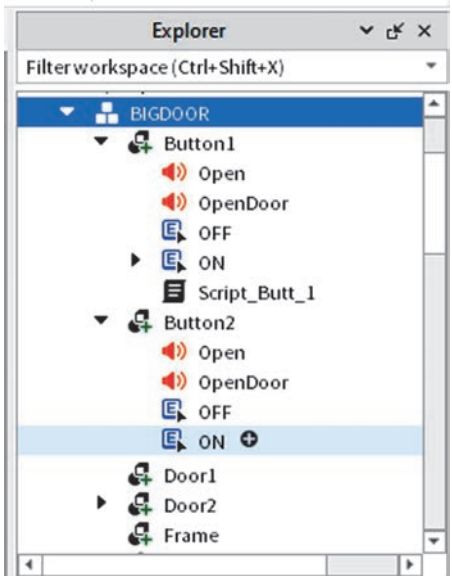


Рис. 10.7 Состав объектов BIGDOOR

Можно видеть, что **BIGDOOR** состоит из большой рамы **Frame**, в которую встроены две раздвижные двери **Door1** и **Door2.** С правой стороны рамы на стене расположена кнопка **Button1**, которая является объектом, содержащим два **Proximity Prompt**. Первый из них называется **ON**, второй — **OFF**. В состав **ON** включен скрипт **Script_Butt_1**, который и управляет работой объекта **BIGDOOR**. Кнопка **Button2** также включает в себя аналогичные **Proximity Prompt**, но этой кнопки на фото не видно — она расположена с другой стороны раздвижных дверей. Скриптов эта кнопка не содержит. Начальная часть (половина) скрипта **Script_Butt_1** приведена ниже:

```
1       i=0
                    -- ОТКРЫВАНИЕ ДВЕРИ (с 1-й стороны)
2       script.Parent.ON.Enabled=true
3       workspace.BIGDOOR.Button1.ON.Triggered:Connect(function(player)
4               script.Parent.ON.Enabled=false
5               script.Parent.Parent.Button2.ON.Enabled=false
6               script.Parent.Open:Play()
7               script.Parent.OpenDoor:Play()
8               i=0
9               while i<70 do
10                      workspace.BIGDOOR.Door1.Position =workspace.
  BIGDOOR.Door1.Position + Vector3.new(0,0,0.1)
11                      workspace.BIGDOOR.Door2.Position =workspace.
  BIGDOOR.Door2.Position + Vector3.new(0,0,-0.1)
12                      wait(0.02)
13                      i=i+1
14              end
```

```
15          wait(5)
16          script.Parent.OFF.Enabled=true
17          script.Parent.Parent.Button2.OFF.Enabled=true
18      end)
                -- ЗАКРЫВАНИЕ ДВЕРИ (с 1-й стороны)
19      workspace.BIGDOOR.Button1.OFF.Triggered:Connect(function(player)
20          script.Parent.OFF.Enabled=false
21          script.Parent.Parent.Button2.OFF.Enabled=false
22          i=0
23          while i<70 do
24              workspace.BIGDOOR.Door1.Position = workspace.
    BIGDOOR.Door1.Position + Vector3.new(0,0,-0.1)
25              workspace.BIGDOOR.Door2.Position = workspace.
    BIGDOOR.Door2.Position + Vector3.new(0,0,0.1)
26              wait(0.02)
27              i=i+1
28          end
29          wait(5)
30          script.Parent.ON.Enabled=true
31          script.Parent.Parent.Button2.ON.Enabled=true
32      end)
```

Разберем как работает этот скрипт:

— первое, что мы делаем в строке 2 — разрешаем работу **Proximity Prompt ON** кнопки **Button1**, указывая **ON.Enabled** = **true**;

— в строке 3, если запускается событие **ON.Triggered** (то есть, если игрок подходит к **Button1** в состав которой входит **Proximity Prompt** с именем **ON** и запускает эту подсказку) мы подключаем к этому событию функцию без имени, которая (см. строки 3…18):

— в строке 4 запрещаем повторную сработку события **ON**;

— в строке 5 запрещаем и сработку события **ON**, входящего в состав кнопки **Button2**, расположенной с другой стороны двери;

— в строках 6 и 7 включаем звук нажатия кнопки и звук открывающихся дверей;

— в строках 9…14 в цикле **while** обеспечиваем открывание раздвижных дверей **Door1** и **Door2**. В этом цикле каждая дверь 70 раз перемещается на шаг равный 0,1 шипа (то есть, в сумме на 7 шипов);

— в строках 16, 17 после того, как двери откроются и завершится задержка строки 15, указываем **OFF.Enabled** = **true** для обеих кнопок

Button1 и **Button2**. Тем самым мы разрешаем работу **Proximity Prompt** с именами **OFF,** которые обеспечивают закрывание дверей.

Строки 19…32 скрипта работают аналогично, с той только разницей, что речь идет о запуске события **OFF.Triggered** кнопки **Button1**.

Вы, конечно, уже догадались, что для полноценного управления этими раздвижными дверями нужно дополнить приведенный выше скрипт, такими же скриптами, но, для кнопки **Button2**, расположенной с другой стороны **BIGDOOR**. Сделайте это самостоятельно.

11. Платформа, через которую можно провалиться

В этой главе покажем, как сделать платформу-мостик, которая после того, как на нее наступает игрок, сначала плавно становится прозрачной, а затем и проницаемой, чтобы зазевавшийся игрок проваливался в пропасть.

Создайте новую деталь, назовите ее **Platform** и переместите на место над пропастью с лавой. Установите нужный размер платформы — чтобы пользователь мог запрыгнуть на нее. Закрепите платформу якорем (**Anchor**). Вставьте в платформу скрипт, переименуйте его и удалите из него исходный скрипт. Создайте переменную для платформы, вставьте в скрипт заготовку функции **fade** (исчезновение), а затем подключите эту функцию к индикатору событий типа **Touched:**

```
1     local platform = script.Parent
2     local function fade()
…          “встроенный код функции”
N     end
…     platform.Touched:Connect(fade)
```

Если бы платформа становилась проницаемой в то же мгновение, когда игрок наступает на нее, это было бы совсем не весело — игроки не могли бы преодолеть пропасть. Достичь плавного повышения прозрачности можно используя **цикл for**, который повторяет нужный код заданное количество раз. Каждый шаг такого кода называется **итерацией**. Число итераций **цикла for** определяется тремя цифрами, разделенными запятыми, например:

```
1     for  i = 1, 10, 1 do
2           “внутренний код цикла”
3     end
```

В этом цикле (в строке 1):
— переменная i выполняет роль **счетчика** шагов-итераций;
— первая “1“ задает **стартовое** значение переменной i;
— “10” задает **конечное** значение переменной i;
— вторая “1” задает шаг изменения показаний счетчика итераций.
То есть, цикл будет выполняться 10 раз — пока переменная i, исходно равная “1”, не станет равной “10”. В качестве “внутреннего кода цикла” можно использовать две строки кода:

```
    platform.Transparency = i/10
    wait(0.2)
```

Этот код обеспечит плавное изменение **Transparency** от значения “0,1” до “1” всего за 2 сек (0,2сек x 10 = 2 сек), а именно:
— после 1-й итерации **Transparency** = 0,1;
— после 2-й итерации **Transparency** = 0,2;
— после 3-й итерации **Transparency** = 0,3;
— после 4-й итерации **Transparency** = 0,4;
— после 5-й итерации **Transparency** = 0,5;
— после 6-й итерации **Transparency** = 0,6;
— после 7-й итерации **Transparency** = 0,7;
— после 8-й итерации **Transparency** = 0,8;
— после 9-й итерации **Transparency** = 0,9;
— после 10-й итерации **Transparency** = 1.

Отметим, что если вместо последней цифры “1”, указанной в цикле **for** указать не “1”, а “2”, то внутренний код цикла выполнит всего 5 (а не 10) шагов-итераций. Ведь счетчик **i** последовательно примет значения 1,3,5,7,9, а значения **Transparency** будет изменяться так:
— после 1-й итерации **Transparency** = 0,1;
— после 2-й итерации **Transparency** = 0,3;
— после 3-й итерации **Transparency** = 0,5;
— после 4-й итерации **Transparency** = 0,7;
— после 5-й итерации **Transparency** = 0,9.

А после того, как будет выполнен цикл изменения прозрачности можно изменить и свойство **CanCollide:**

```
1   local platform = script.Parent
2   local function fade()
3       for i = 1,10,1 do
4           platform.Transparency = i/ 10
5           wait(0.2)
6       end
7       platform.CanCollide = false
8       wait(3)
9       platform.CanCollide = true
10      platform.Transparency = 0
11  end
12  platform.Touched:Connect(fade)
```

Понятие дребезга

В главе “Смертельная лава” вы узнали, что индикатор событий **Touched** запускается каждый раз, когда часть тела игрока соприкасается с лавой. Но, такой результат, очевидно, вызовет проблемы, когда игрок будет шагать по платформе. Ведь функция **fade()** будет запускаться каждый раз, когда игрок будет делать очередной шаг.

Чтобы этого не было, функция **fade()** должна запускаться только один раз — когда игрок впервые касается платформы. Ситуация, когда действие запускается только один, а не несколько раз, называется “**устранением дребезга**”. Чтобы устранить дребезг функции, можно использовать “логическую” переменную, которая может иметь только два значения: **истина** или **ложь**:

— в строке 2 создадим такую логическую переменную с именем **isTouched** (Касается) и присвоим ей исходное значение **false** (ложь), что будет означать, что сразу после запуска программы никто еще не касается нашей платформы.

— весь “внутренний” код функции **fade()** вставим внутрь оператора **if** (строка 4), который будет предварительно проверять: равно ли значение переменной **isTouched** значению **false**.

— в самом начале “внутреннего” кода функции **fade()** (в строке 5) вставим команду, которая изменит значение **isTouched** на **true**. То есть, если игрок касается платформы в первый раз, то значение логической переменной **isTouched** равно **false** и функция **fade()** начнет исполняться. Но, команда строки 5 сразу же изменит значение

isTouched на **true.** Поэтому следующие шаги игрока по платформе не будут приводить к повторному запуску функции **fade()**.

— лишь после того, как будут исполнены все команды функции **fade()**, логической переменной **isTouched** будет вновь присвоено исходное значение **false** (строка 11), что сделает возможным повторный запуск функции **fade()**.

Конечный вид программы

```
1     local platform = script.Parent
2     local isTouched = false
3     local function fade()
4         if isTouched == false then
5             isTouched = true
3             for i = 1,10,1 do
4                 platform.Transparency = i/ 10
5                 wait(0.2)
6             end
7             platform.CanCollide = false
8             wait(3)
9             platform.CanCollide = true
10            platform.Transparency = 0
11            isTouched = false
12        end
13    end
14    platform.Touched:Connect(fade)
```

Протестируйте свой код — вы обнаружите, что платформа плавно исчезает, когда пользователь прыгает на нее, и появляется вновь через три секунды.

Вы можете создать еще одну такую платформу, чтобы обеспечить переход через более широкую пропасть, а также изменить скорости, с которой платформы будут исчезать, чтобы повысить сложность игры.

12. Начисление баллов игрокам

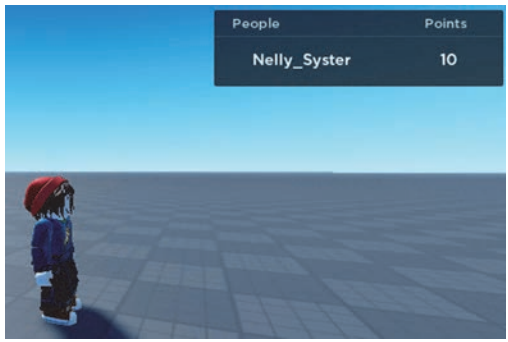


Рис. 12.1 Таблица лидеров

В этой главе мы научимся начислять баллы игрокам за какие-либо достижения в игре.

В Roblox исходно встроена **таблица лидеров** — список игроков с отображением баллов, заработанных этими игроками — она отображается в правой верхней части экрана.

Мы уже говорили, что скрипты можно помещать в разные “контейнеры” игры. Скрипт подсчета очков помещают в **ServerScriptService**. Создайте в **ServerScriptService** скрипт, например, с именем **Points** (Баллы).

Сервис “Players”

В Roblox имеется целый ряд **специальных сервисов** (автоматических программ), которые выполняют разные полезные функции. Например, в сервисе **Players** среди прочей информации об игроках, ведущих игру, имеется **индикатор событий** типа **PlayerAdded** (ДобавленИгрок), который можно использовать для выявления присоединившихся к игре игроков и начисления им баллов.

Чтобы получить доступ к какому-либо сервису, нужно (см. ниже):

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

— создать для этого сервиса переменную, причем, при объявлении переменных, содержащих сервисы, эти переменные **должны быть названы точным именем сервиса** (в нашем случае - **Players**), несмотря на то, что это нарушает ранее упоминавшуюся *традицию* называть переменные с малых, а не с заглавных букв;
— затем, нужно связать созданную переменную с объектом **game**;

— подключить к **game**(через “**:**”) функцию **GetService**(Подключить Сервис);
— указать в качестве *параметра* функции **GetService** название сервиса (в нашем случае - “**Players**”):

Следом, создаем заготовку функции **onPlayerJoin**(Подключение Игрока) с параметром (**player**) — она будет подключать каждого из игроков к таблице лидеров:

```
local function onPlayerJoin(player)
    “внутренний код функции”
end
```

После формального описания функции, подключаем индикатор событий **PlayerAdded** сервиса **Players** к функции **onPlayerJoin**. Тогда, как только сервис **Players**, узнает о подключении нового игрока к игре, он сразу же запустит созданную нами функцию **onPlayerJoin.**

```
Players.PlayerAdded:Connect(onPlayerJoin)
```

Функция **onPlayerJoin** нужна, чтобы каждому новому игроку:
— создать *специальную папку* для хранения его баллов;
— создать *переменную*, соответствующую его баллам и поместить ее в созданную папку;
— задать *исходное (нулевое) значение* созданной переменной.

Вы помните, что в языке Луа новые объекты можно создавать прямо внутри скрипта с помощью функции **Instance.new()** — где в скобках указывается тип создаваемого объекта. Так мы и поступим во “внутреннем коде” функции:

```
2   local function onPlayerJoin(player)
3       local leaderstats = Instance.new("Folder")
4       leaderstats.Name = "leaderstats"
5       leaderstats.Parent = player
6       local points =Instance.new("IntValue")
7       points.Name =”Points”
8       points.Value = 0
9       points.Parent = leaderstats
10  end
11  Players.PlayerAdded:Connect(onPlayerJoin)
```

— в строке 3 с помощью функции **Instance.new** создаем новый объект типа **"Folder"**(Папка) и сохраняем этот объект в новой локальной переменной с именем **leaderstats** (таблица лидеров);

— в строке 4 укажем свойству **Name** переменной **leaderstats** значение “**leaderstats**”;

— в строке 5 укажем свойству **Parent** переменной **leaderstats** значение **player** — то есть, папка будет храниться в модели игрока **player**;

— в строке 6 создаем еще один новый объект, который всегда будет являться целым числом (**IntValue** означает “целое число”) и сохраним этот объект в переменной **points**(баллы);

— в строке 7 укажем свойству **Name** переменной **points** значение **“Points”**;

— в строке 8 укажем свойству **Value**(Величина) переменной **points** исходное значение “**0**”;

— в строке 9 укажем свойству **Parent** переменной **points** значение **leaderstats** (то есть, баллы будут храниться в “родительской” папке **leaderstats**)**.**

Итог: баллы игрока теперь будут храниться в переменной **points**, которая будет храниться в папке **leaderstats**, а сама папка **leaderstats** будет храниться в объекте **player**, откуда с помощью сервиса **Players** можно будет брать данные.

Запустите игру и вы увидите, что в правом верхнем углу появится таблица лидеров с именами находящихся в игре игроков и (пока нулевым) количеством очков рядом с ними.

Начисление баллов игрокам

Для простоты, сделаем так, чтобы каждый игрок зарабатывал один балл за каждую секунду своего существования в игре. Потом, вы сможете установить свои правила начисления баллов. А сейчас, можно использовать цикл **while true** и задержку **wait(1)** чтобы постоянно обновлять значения баллов игроков через каждую секунду.

```
    while true do
        wait(1)
```

Но, чтобы учесть баллы игроков, нужно знать **список** (или **массив**) всех игроков. Этот список можно узнать, подключив к сервису **Players** функцию **GetPlayers** (ПолучитьИгроков). В следующей строке кода получаем игроков и сохраняем их список в переменной **playerList** (списокИгроков):

```
    local playerList = Players:GetPlayers()
```

В списках каждый элемент имеет свой номер, а, значит, и доступ к каждому элементу можно получить по его номеру. Самый первый игрок имеет номер **“1”**, номер последнего обозначают, как “**#playerList**” (никто ведь не знает сколько игроков играет в эту игру в каждый конкретный момент, поэтому невозможно указать какой конкретно номер у игрока, который вошел в игру последним).

Создав цикл **for** с начальным значением “счетчика цикла” равным ”**1**” и конечным значением равным **#playerList**, можно перебрать один за другим каждого из игроков списка и добавить по одному баллу к переменным **points**, хранящимся в их папках **leaderstats**. В скрипте мы назвали счетчик цикла **currentPlayer**(НомерИгрока):

```
12      while true do
13          wait(1)
14          local playerList = Players:GetPlayers()
15          for currentPlayer = 1, #playerList  do
16              local player = playerList[currentPlayer]
17              local points = player.leaderstats.Points
18              points.Value = points.Value + 1
19          end
20      end
```

При исполнении цикла **for** (строки 15…19):
— в строке 16 в переменной с именем **player** сохраняется игрок из списка **playerList**, номер которого соответствует значению **currentPlayer**;

— в строке 17 переменной **points** присваивается количество баллов, которые хранятся в папке **leaderstats** этого игрока;

— в строке 18 количество баллов, которые хранятся в папке **leaderstats** этого игрока увеличивается на один балл:

Конечный код начисления баллов игрокам:

```
1       local Players = game:GetService("Players")
2       local function onPlayerJoin(player)
3           local leaderstats = Instance.new("Folder")
4           leaderstats.Name = "leaderstats"
5           leaderstats.Parent = player
6           local points =Instance.new("IntValue")
7           points.Name ="Points"
8           points.Value = 0
9           points.Parent = leaderstats
10      end
11      Players.PlayerAdded:Connect(onPlayerJoin)
12      while true do
13          wait(1)
14          local playerList = Players:GetPlayers()
15          for currentPlayer = 1, #playerList  do
16              local player = playerList[currentPlayer]
17              local points = player.leaderstats.Points
18              points.Value = points.Value + 1
19          end
20      end
```

Запустите игру - вы увидите, что баллы игроков увеличиваются с течением времени.

Как исключить из статистики погибших игроков

Казалось бы все хорошо. Но, игрокам которые умирают, нужно сбрасывать баллы до “0”. То есть, нужно научиться определять, когда игрок умер.

Когда игрок умирает, встроенный в него объект **Humanoid** автоматически запускает **индикатор события Died**(Умер). То есть, можно использовать этот индикатор, чтобы узнать, когда нужно сбросить баллы

игрока до “0”. Но, как мы знаем, объект **Humanoid** находится внутри объекта **character**(персонаж), который создается внутри модели **player**.

Чтобы находить объекты **Humanoid** в объектах **character** и сбрасывать баллы убитых игроков в “0” создадим еще одну функцию **onCharacterAdded**. Вот заготовка этой функции:

```
1   local Players = game:GetService("Players")
2   local function onCharacterAdded(character, player)
…       “встроенный код функции”
10  end
```

Сервис **Players** уже подключен к функции **onPlayerJoin** (см. строку 11 скрипта). Используем этот сервис и для подключения новой функции к объекту **character** игрока. Для этого создадим в **onPlayerJoin** переменную **player** со свойством **CharacterAdded** (ПерсонажДобавлен) и подключим эту переменную к безымянной функции, которая, в свою очередь, подключит к **character** игрока функцию **onCharacterAdded** и передаст ей два параметра — **character** и **player**:

```
        player.CharacterAdded:Connect(function(character)
                onCharacterAdded(character, player)
        end)
```

Опишем, что же делает функция **onCharacterAdded**:

```
1   local Players = game:GetService("Players")
2   local function onCharacterAdded(character, player)
3       player:SetAttribute("IsAlive", true);
4       local humanoid = character:WaitForChild("Humanoid")
5       humanoid.Died:Connect(function()
6           local points = player.leaderstats.Points
7           points.Value = 0
8           player:SetAttribute("IsAlive", false)
9       end)
10  end
```

— в строке 3 создаем для переменной **player** атрибут “**IsAlive”**(Живой) и присваиваем ему значение **true**.

Атрибуты аналогичны **свойствам** объекта, но, в отличие от свойств, вы можете создавать свои собственные атрибуты для любого объекта. Атрибуты можно создавать, редактировать и удалять непосредственно в окне **Properties** или с помощью скриптов. Атрибуты могут быть числовыми, логическими (**true** или **false** = **правда** или **ложь**) или текстовыми. К атрибутам можно относить: величину убойной силы оружия, время перезарядки оружия, максимальные скорости транспортных средств, цвет окраски объекта, звуки двигателей и выстрелов.

Созданный нами **логический атрибут** нужен нам, чтобы при просмотре списка игроков начислять баллы только тем игрокам, у которых условие “**IsAlive”** (Живой) соответствует значению **true**;

— в строке 4 создаем переменную **humanoid** и присваиваем ей значение объекта **Humanoid** из состава объекта **character** игрока. Мы используем функцию **WaitForChild**(ЖдатьПоявленияПотомка) потому, что объект **Humanoid** появляется в модели **character** не мгновенно. Получив **Humanoid** можно использовать встроенный в него индикатор события **humanoid.Died**;

— строках 5…9 подключаем к индикатору события **humanoid.Died** функцию (без названия и параметра), которая, если индикатор **Died** будет включен, сбросит баллы этого игрока в “**0**” и изменит на **false** его атрибут “**IsAlive”** (Живой).

Конечный код начисления и списания баллов игрока таков:

```
1   local Players = game:GetService("Players")
2   local function onCharacterAdded(character, player)
3       player:SetAttribute("IsAlive", true)
4       local humanoid = character:WaitForChild("Humanoid")
5       humanoid.Died:Connect(function()
6           local points = player.leaderstats.Points
7           points.Value = 0
8           player:SetAttribute("IsAlive", false)
9       end)
10  end
11  local function onPlayerJoin(player)
12      local leaderstats = Instance.new("Folder")
13      leaderstats.Name = "leaderstats"
```

```
14      leaderstats.Parent = player
15      local points = Instance.new("IntValue")
16      points.Name = "Points"
17      points.Value = 0
18      points.Parent = leaderstats
19      player.CharacterAdded:Connect(function(character)
20          onCharacterAdded(character, player)
21      end)
22  end
23  Players.PlayerAdded:Connect(onPlayerJoin)
24  while true do
25      wait(1)
26      local playerList = Players:GetPlayers()
27      for i = 1, #playerList  do
28          local player = playerList[i]
29          if player:GetAttribute("IsAlive") then
30              local points = player.leaderstats.Points
31              points.Value = points.Value + 1
32          end
33      end
34  end
```

Запустите код — вы увидите, что баллы живых игроков растут со временем, а баллы умерших игроков сбрасываются в “0”.

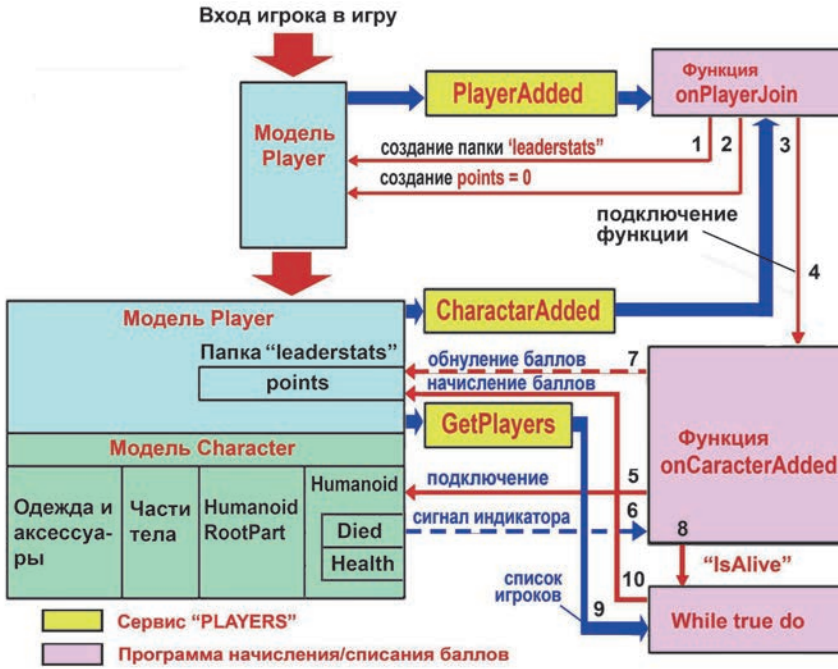


Рис. 12.2 Взаимодействие программных средств при начислении и списании баллов игрокам (цифрами показана последовательность действий во времени)

Обратите внимание, мы поместили функцию **onCharacterAdded**, которую написали позже, чем функцию **onPlayerJoin**, перед ней, а не после нее. Это неспроста, если бы мы разместили функцию **onPlayerJoin** выше, то читая ее код, компьютер нашел бы в строке 19 ссылку о необходимости использовать функцию **onCharacterAdded**, о которой он ничего не знает, и, естественно, посчитал бы наш скрипт ошибочным. Но, теперь, когда мы сначала разместили функцию **onCharacterAdded**, ссылка на нее в функции **onPlayerJoin** воспринимается естественно.

13. Программируем одной программой несколько объектов

В прошлых главах мы писали скрипты для отдельных **parts**. Если бы мы захотели продублировать эти **parts**, нам пришлось бы копировать и их скрипты. Если бы мы захотели как-то обновить скрипты этих **parts,** то потребовалось бы вносить изменения в каждый из скриптов. В этой главе создадим несколько аптечек с единым скриптом, способных восстанавливать здоровье игроков.

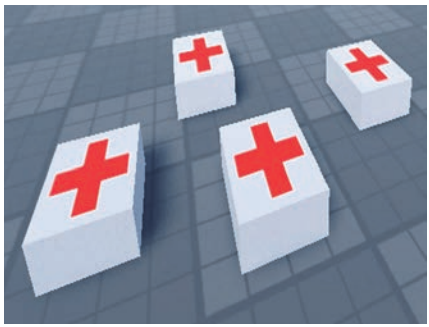


Рис. 13.1 Аптечки

Как выглядит каждая аптечка вы решите сами. Заведите в **Workspace** в папку под названием **H_Up** и поместите в нее несколько **parts** аптечек с одинаковым названием **h_up**.

Для уменьшения здоровья игроков создайте “смертельную лаву” и — для тестирования работы аптечек. Естественно, нужно будет изменить скрипт “смертельной лавы”, чтобы здоровье игрока уменьшалось не до нуля, а, например, всего на 20%.

Как восстановить здоровье игрока

Добавим в папку **ServerScriptService** скрипт с названием **Health**. Наш скрипт должен восстанавливать здоровье игрока. А это — почти то же самое, что делала наша “Смертельная лава” (только наоборот). Создадим функцию **Health_Up**(ПоднятияЗдоровья) с двумя параметрами:

```
1       local function Health_Up(otherPart,h_up)
2               local character = otherPart.Parent
3               local humanoid
=character:FindFirstChildWhichIsA("Humanoid")
4               if humanoid then
5                       humanoid.Health = 100
6               end
7       end
```

Первый параметр функции **otherPart** соответствует тому, что коснулось аптечки. Второй параметр **h_up** соответствует самой аптечке.

Прокомментируем скрипт:
— в строке 2 определяем Родителя **otherPart** и (помня, что для аватара игрока Родителем коснувшейся аптечки части будет **character**) присваиваем его значение локальной переменной **character**;
— в строке 3 при помощи функции **FindFirstChildWhichIsA** (Найти ПервогоПотомкаКоторыйЯвляется…) находим в этом **character** объект **Humanoid** и присваиваем его значение переменной с таким же именем;
— в цикле 4…6, то увеличиваем здоровье гуманоида до максимума.

Обратите внимание: мы применили функцию **FindFirstChildWhichIsA**, она более надежна. Она определяет разыскиваемый объект **по его типу**, в отличие от функции **FindFirstChild,** которая разыскивает объекты **по их имени** (то есть, может найти то, что кто-то случайно назвал **"Humanoid"**).

Особенности доступа к аптечкам
Папка **H_Up**, входящая в **Workspace** и содержащая аптечки, может быть еще не загружена в игру к моменту запуска скрипта из **ServerScriptService**. Поэтому нужно использовать функцию **WaitForChild** (ЖдатьПоявлениеПотомка), и затем, передать значение этой папки локальной переменной **h_Folder** (ПапкаАптечек):

```
local h_Folder = workspace:WaitForChild("H_Up")
local h_array = h_Folder:GetChildren()
```

Затем, подключив к папке **h_Folder** функцию **GetChildren** (Получить Потомков), получаем **список всех объектов** этой папки, то есть, список всех наших аптечек, и передаем его в переменную **h_array**(список):

Организуем контроль касаний каждой из аптечек
В языке Lua есть еще один тип **цикла for**, в котором нет необходимости
указывать с какого начального значения цикл начинается и каким конечным значением он заканчивается. Этот тип цикла используется

для просмотра **каждого элемента массива**. Скрипт этого типа цикла следующий:

```
for _, value in ipairs (array) do
    “встроенный код цикла”
end
```

где: после **for**_ стоит знак подчеркивания;
value — переменная, соответствующая элементу списка;
in ipairs — указание на просмотр каждого элемента списка;
array — список, который будет просматривать цикл **for**.

Чтобы восстанавливать здоровье игроков нам, как раз, необходима проверка касаний каждой из аптечек, которые входят в состав списка **h_array**. Используем для этого описанный выше тип цикла:

```
for _, h_up in ipairs(h_array) do
    “встроенный код цикла”
end
```

Внутрь цикла вставим дополнительную функцию (без имени с параметром **otherPart**), которая будет передавать объект **otherPart** функции **Health_Up**:

```
h_up.Touched:Connect(function(otherPart)
    Health_Up(otherPart, h_up)
end)
```

Промежуточный код программы:

```
1   local h_Folder = workspace:WaitForChild("H_Up")
2   local h_array = h_Folder:GetChildren()
3   local function Health_Up(otherPart,h_up)
4       local character = otherPart.Parent
5       local humanoid
=character:FindFirstChildWhichIsA("Humanoid")
6       if humanoid then
7           humanoid.Health = 100
8       end
9   end
10  for _, h_up in ipairs(h_array) do
11      h_up.Touched:Connect(function(otherPart)
12          Health_Up(otherPart, h_up)
```

```
13          end)
14    end
```

Давайте уточним, как работает скрипт. Да, очень просто! Цикл **for_ in pairs** “просматривает” список всех имеющихся аптечек и подключает индикатор **Touched** каждой из них к функции **Health_Up**. Вот и все. А с функцией **Health_Up** вам должно быть все понятно. Теперь при каждом касании любой аптечки происходит восстановление здоровья.

Протестируйте код, вы обнаружите, что аптечка восстанавливает ваше здоровье. В правом верхнем углу экрана при уменьшении здоровья появляется шкала здоровья, которая исчезнет, когда игрок выздоровеет.

Временное отключение аптечек

Сейчас наши аптечки постоянно готовы лечить любого игрока, который к ним прикоснется. В игре было бы более эффективно, если бы аптечки после их касания отключались на некоторое время, прежде чем их можно будет использовать снова. Измените скрипт аптечек, чтобы это стало возможно (конечный скрипт приведен ниже):

— в строке 16 цикла **for** заведите каждой аптечке атрибут **Enabled** (Готова) **и установите его** значение **true**;

— внутри функции **Health_Up** (в строке 4) введите проверку нового атрибута **Enabled** и если аптечка “не готова”, то объекты **character** и **Humanoid** программа искать не будет**;**

— в строке 8 (сразу после восстановления здоровья игрока) увеличьте прозрачность аптечки до “1”, это сделает ее невидимой;

— в строке 9 измените значение нового атрибута **Enabled** на **false** (сделав аптечку “не готовой” к восстановлению здоровья);

— в строке 10 вставьте задержку на время которой аптечка будет не способна к восстановлению здоровья и невидима игрокам;

— в строке 11 восстановите видимость (непрозрачность) аптечки;

— в строке 12 восстановите значение **true** нового атрибута **Enabled**, что вновь вернет работоспособность этой аптечке:

Конечный код программы аптечек

```
1       local h_Folder = workspace:WaitForChild("H_Up")
2       local h_array = h_Folder:GetChildren()
3       local function Health_Up(otherPart,h_up)
4               if h_up:GetAttribute(“Enabled”) then
4                       local character = otherPart.Parent
5                       local humanoid = character:FindFirstChildWhichIsA
  ("Humanoid")
6                       if humanoid then
7                               humanoid.Health = 100
8                               h_up.Transparency = 1
9                               h_up:SetAttribute(“Enabled”,false)
10                              wait(10)
11                              h_up.Transparency = 0.1
12                              h_up:SetAttribute(“Enabled”,true)
13                      end
14              end
15      end
15      for _, h_up in ipairs(h_array) do
16              h_up:SetAttribute("Enabled", true)
16              h_up.Touched:Connect(function(otherPart)
17                      Health_Up(otherPart, h_up)
18              end)
19      end
```

Напомним:

H_Up — папка в **Workspace**, в которой находятся аптечки **h_up**;
h_Folder — переменная, в которую копируем папку **H_Up**;
h_up — **parts** аптечек;
h_array — список всех аптечек в папке **h_Folder**.

14. Сохранение данных в хранилище Roblox



Рис. 14.1 Банк

Играм часто необходимо хранить некоторые данные между сеансами, например, уровни игроков, их баллы, предметы инвентаря (золото/деньги).

В этой главе расскажем, как создать хранилище данных, сохранять данные и считывать данные обратно в игровую сессию.

Подключение доступа к хранилищам данных

По умолчанию игры, созданные в Roblox Studio, не имеют доступа к хранилищам данных, поэтому такой доступ нужно сначала включить. Для этого вашу игру нужно опубликовать в Roblox. Войдите в меню **File** Roblox Studio и нажмите кнопку **Publish to Roblox**. Введите название и автора игры и сохраните данные нажав кнопку **Create**(Создать).



Рис. 14.2 Кнопка вызова меню Game settings

После того, как загруженная в Roblox игра откроется, выберите раздел **HOME**, а в нем раздел **Game settings** (Настройки). В появившемся выпадающем меню выберите раздел **Security** и в этом разделе включите переключатель **Enable Studio Access to API Service**. Сохраните внесенное изменение (**Save**).

Создание хранилища данных

В этом примере мы организуем хранилище данных с именем **Gold,** которое сохранит золото каждого игрока между сеансами игры.

Создайте в **ServerScriptService** новый скрипт с именем **Gold**. В первой строке скрипта нужно подключиться *к сервису хранения данных* Roblox. Напомним, чтобы получить доступ к какому-либо сервису, нужно:

— создать для него переменную, причем, при объявлении переменных, содержащих сервисы, эти переменные должны быть названы **точным именем сервиса** (в нашем случае - “**DataStoreService**”), несмотря на то, что это нарушает ранее упоминавшееся требование о том, что называть переменные предпочтительнее с малых, а не заглавных букв;

— связать (через знак равенства) созданную переменную с объектом **game**;

— подключить к **game** (через двоеточие) функцию **GetService**;

— указать в качестве параметра функции **GetService** название сервиса (в нашем случае - “**DataStoreService**”):

```
1    local DataStoreService = game:GetService("DataStoreService")
```

В строке 2, вызываем подключенный сервис хранения данных **DataStore Service**, с помощью функции **GetDataStore** (ПолучитьХранилищеДанных) создаем в нем наше хранилище золота **“Gold”** и сохраняем его значение в локальной переменной **goldStore**:

```
2    local goldStore = DataStoreService:GetDataStore("Gold")
```

Сохранение данных в хранилище

Хранилище данных — это таблица, каждая строка которой имеет уникальный ключ, например, уникальный номер игрока **UserId**. Это выглядит примерно так:

UserId	Значение данных
312506082	50
351675979	20
505306092	320

Чтобы сохранить золото **Gold** вашего аватара в хранилище данных Roblox нужно:

— создать переменную с именем **playerUserID** и присвоить ей значение вашего **userID** (скопируйте свой уникальный номер в аккаунте Roblox);

```
3    local playerUserID = ваш UserID
```

— создать переменную **Gold** для хранения золота и задать ей какое-то значение:

```
4    local Gold = 320
```

— подключить к хранилищу **goldStore** функцию передачи **SetAsync** и указать ей в качестве параметров уникальный номер игрока **playerUserID** и название переменной **Gold**:

```
5    local success, errorMessage = pcall(function()
6        goldStore:SetAsync(playerUserID, Gold)
7    end)
8    if not success then
```

```
9           warn(errorMessage)
10      end
```

Безымянная функция, указанная в строке 5, подключает функцию передачи данных к специальной функции **pcall()**, которая обеспечивает проверку успешности выполнения передачи данных.

Поясним. Передача данных в хранилище — это операция в сети Интернет между компьютером одного из игроков и “главным” компьютером (**сервером**) Roblox и она иногда может, завершиться ошибкой. Для обнаружения и обработки таких сбоев используют специальную функцию **pcall()**.Функция **pcall()** контролирует появление ошибок при выполнении передачи данных **SetAsync()**. Если ошибок нет, **pcall()** присваивает переменной **success**(успех) значение **true**. В противном случае **pcall()** присваивает этой переменной значение **false**. Наличие ошибки передачи данных проверяется в цикле **if - then** в строках 8…10, и, если произошел сбой, то в окне **Output** появляется сообщение об ошибке.
Предупредим: не нужно отправлять данные в хранилище данных слишком часто. Такие “посылки” всегда помещаются в очередь, и, если очередь будет заполнена, то дополнительные “посылки” будут просто отброшены. Частая ошибка — обновление данных о золоте игрока каждый раз, когда он находит золотую монету или самородок. Вместо этого нужно хранить золото игрока в специальной локальной переменной и обновлять хранилище данных только изредка, например, при периодическом автосохранении игры и/или когда игрок покидает игру.

Чтение данных из хранилища
Чтобы узнать сколько золота **Gold** лежит у вас в хранилище данных Roblox, подключите к хранилищу **goldStore** функцию **GetAsync** (**Get** - Получить) и укажите в качестве параметра функции свое уникальное имя **playerUserID**:

```
11      local success, Gold = pcall(function()
12          return goldStore:GetAsync(playerUserID)
13      end)
14      if success then
15          print(“Us_ID=”, playerUserID, “ Gold=” Gold)
16      end
```

Чтение данных из хранилища также производят под контролем специальной функции **pcall()**. Если передача данных из хранилища **goldStore** в переменную **Gold** пройдет удачно, то строка 15 распечатает количество принадлежащего вам золота.

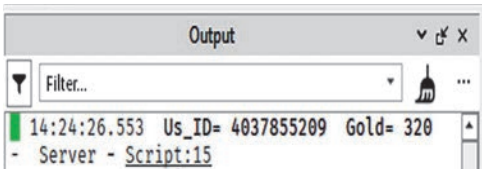


Рис. 14.3 Вывод данных из хранилища

Чтобы протестировать наш скрипт, скопируйте строки 1…16 в скрипт **Gold**:

```
1    local DataStoreService = game:GetService("DataStoreService")
2    local goldStore = DataStoreService:GetDataStore("Gold")
3    local playerUserID = ваш UserID
4    local Gold = 320
5    local success, errorMessage = pcall(function()
6     goldStore:SetAsync(playerUserID, Gold)
7    end)
8    if not success then
9          warn(errorMessage)
10   end
11   local success, Gold = pcall(function()
12         return goldStore:GetAsync(playerUserID)
13   end)
14   if success then
15         print("Us_ID=", playerUserID, " Gold=" Gold)
16   end
```

Откройте окно **Output** и запустите программу. В окне **Output** будет напечатано: “**Us_ID=*ваш ID* Gold = 320”**.

15. Игра с сохранением баллов игрока между сеансами игры

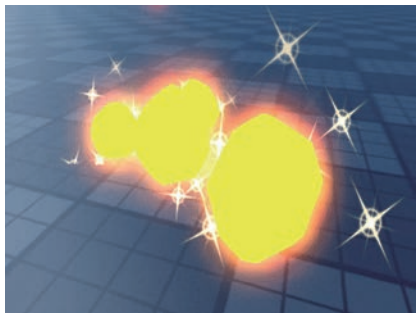


Рис. 15.1 Золотые самородки

Мы с вами научились:
— начислять игрокам баллы;
— определять убитых игроков и списывать их баллы до “0”;
— определять факты, прикосновения игроков к предметам;
— сохранять данные в хранилище Roblox.

Давайте, используя наши знания, запрограммируем игру, чтобы в игре появились разбросанные то тут, то там самородки золота, а игроки прикоснувшись к ним, могли собирать их, увеличивая свой золотой запас. А еще, чтобы накопленное игроками золото не пропадало, а сохранялось, когда они будут повторно входить в игру.

Заготавливаем золото

Создадим в **Workspace** папку с названием **GoldChunks** (Золотые Самородки), в которой создадим **parts** золотого самородка с именем **chunk**. Откроем окно **Properties** самородка **chunk** и введем значения его свойствам:



BrickColor ……. Deep orange
Color …….. 255, 120, 0
DoubleSlided …… V
Material …….. Neon
MeshID ……. rbxassetid://6768819109
Reflectance …….. 1
Archiable …….. V
Size …….. 1.5, 1.5, 1.5
Position …….. 45, 2, 47
Orientation …….. 0, -45, 90
DropCollided …… V

Рис. 15.2 Папка с золотыми самородками

После этого наведем курсор мыши на **chunk**, нажмем правую кнопку мыши и выберем в выпадающем меню **Duplicate**, чтобы получить в папке дубликат самородка. Повторим это действие несколько раз, чтобы получить нужное количество самородков. Создав нужное количество самородков, измените их исходные позиции — разнесите их равномерно по игровой зоне, в которой предстоит собирать самородки. Далее мы будем менять положение самородков автоматически.

Начальная часть программы управления золотом *GoldManager*

Программу управляющую сбором золотых самородков, их накоплением и сохранением данных между сеансами игры назовем, **GoldManager** и разместим в **ServiceScriptService**. Рассмотрим скрипт ее начальной части:

```
1    local Gold_Folder = workspace:WaitForChild("GoldChunks")
2    local ch_array = Gold_Folder:GetChildren()
3    local Players = game:GetService("Players")
4    local DataStoreService = game:GetService("DataStoreService")
5    local goldStore = DataStoreService:GetDataStore("Gold")
6    local playerGold = {}
7    local gold_VAL = 10
8    local points = Instance.new("IntValue")
9    points.Name = "Points"
10   points.Value = 0
```

— в строке 1, используя функцию **WaitForChild**, ждем загрузки в игру папки **GoldChunks**, а затем присваиваем значение (содержимое) этой папки переменной **Gold_Folder**;

— в строке 2, используя функцию **GetChildren()** получаем из переменной **Gold_Folder** список всех самородков, входящих в папку **GoldChunks** и присваиваем его значение (содержимое) переменной **ch_array**;

— в строке 3 подключаем сервис **PLayers**, он предоставит нам данные об игроках, участвующих в игре;

— в строке 4 подключаем **сервис** хранения данных **DataStoreService**;

— в строке 5, вызываем подключенный сервис хранения данных, создаем в нем (функцией **GetDataStore**) “главное” хранилище золота с

именем **“Gold”** и сохраняем его значение в локальной переменной **goldStore**. В это хранилище мы будем записывать данные только изредка — при периодическом автосохранении (через 30 сек) и когда игрок покидает игру;

— в строке 6 создаем еще один локальный “**список-хранилище”** золота **playerGold** (о том, что это список говорят фигурные скобки). В этот список мы будем записывать золото игроков, сразу же, как только это золото будет у них появляться;

— в строке 7 заведем переменную **gold_VAL** (Цена Самородка) и присвоим ей значение 10 единиц. То есть, при нахождении очередного самородка цена золота и баллы игрока в таблице лидеров будут увеличиваться на 10 единиц:

— в строках 8..10 создадим локальную переменную **points**, укажем ее имя и присвоим ей исходное значение “0” баллов. Меняющееся значение этой переменной будет показывать встроенная в Roblox “таблица лидеров”:

Функция контроля касаний GoldTouch

Теперь пора написать функцию **GoldTouch** для контроля касаний золотых самородков игроками:

```
11    local function GoldTouch(chunk, gold_VAL, player)
12        chunk.Parent = nil
13        local playerID = player.UserId
14        if playerGold[playerID] then
15            playerGold[playerID] = playerGold[playerID] + gold_VAL
16            points.Value = points.Value + gold_VAL
17        end
18        wait(2)
19        local next_X =  Random.new(-72,72)
20        local next_Z =  Random.new(-72,72)
21        chunk.CFrame = CFrame.new(next_X, 50, next_Z)
22        chunk.Parent = workspace.GoldChunks
23    end
```

Как вы видите, мы указали для этой функции три параметра: **chunk, gold_VAL, player**. Это значит, что вызывая функцию мы будем долж-

ны всякий раз передавать ей **три значения**, которые функция будет воспринимать в качестве указанных параметров.



Рис. 15.3 Самородок золота

Рассмотрим скрипт функции **GoldTouch**:
— в строке 12 мы указали, что при касании самородка игроком “Родитель” самородка становится равен **nil** (пропадает). Это означает, что мы уничтожаем самородок — он (временно) пропадает с игрового поля;
— в строке 13 мы заводим переменную **playerID** и присваиваем ей значение **уникального номера** игрока, прикоснувшегося к самородку;
— в цикле **if - then** (строки 14…17):

— в строке 15 увеличиваем на 10 единиц количество золота в том элементе списка-хранилища **playerGold**, который соответствует уникальному номеру игрока;

— в строке 16 увеличиваем на 10 единиц, количество баллов (**points**), которые будут высвечиваться этому игроку в “таблице лидеров”;

— строка 18 задает задержку на 2 секунды, после которой запускается скрипт замены временно уничтоженного самородка “новым” (а на самом деле - тем же самым) самородком. При этом, новые координаты X и Z “нового” самородка определяются в строках 19 и 20 как случайные цифры в диапазоне от “-72” до “72” (вы можете указать другие значения, которые подойдут для размеров игровой зоны, которую вы впоследствии создадите);.

— строка 21 размещает “новый” самородок в месте игрового поля, которое соответствует новым координатам X и Z, а координата самородка по оси Y не меняется (всегда равна “50”). То есть, новый самородок всегда будет падать с высоты 50 шипов, что дает ему возможность после падения откатиться немного в сторону — из-за его скошенных граней;

— строка 22 указывает, что родителем “нового” самородка является папка **GoldChunks** игрового мира **workspace.**

Цикл *for in ipairs* для списка самородков

Теперь напишем цикл **for_ in ipairs**, который “просмотрит” весь список самородков и подключит к каждому самородку функцию **GoldTouch**:

```
24    for _, chunk in pairs(ch_array) do
25          chunk.Touched:Connect(function(otherPart)
26                local partParent = otherPart.Parent
27                local player = Players:GetPlayerFromCharacter(partParent)
28                if player then
29                      GoldTouch(chunk, gold_VAL, player)
30                end
31          end)
32    end
```

— в строке 25 индикатор событий **Touced** каждого из самородков **chunk** подключается к функции без имени с параметром **otherPart**;
— внутри этой функции:

— в строке 26 создается локальная переменная, в которую помещается “Родитель” **otherPart**, коснувшейся самородка (то есть, в эту переменную помещается **character** аватара игрока);

— в строке 27 с помощью сервиса **Players** и функции **GetPlayerFrom Character**(ПолучитьИгрокаИзПерсонажа) определяется игрок, который коснулся самородка. Этот игрок помещается в локальную переменную **player**;

— в строках 28..30 условный оператор **if - then** (если игрок определен) будет запускать функцию **GoldTouch**, передавая ей в качестве параметров текущие значения **chunk, gold_VAL** и **player**.

Обратите внимание: мы размещаем в скрипте цикл **for_ in ipairs** после функции **GoldTouch**, ведь в этом цикле мы вызываем функцию **GoldTouch**, и если бы мы разместили скрипт цикла выше, чем скрипт функции **GoldTouch**, то компьютер воспринял бы такую ссылку, как ошибку.

Функция *onCharacterAdded* — контроля жизни игроков

Копируем без всяких изменений то, что мы написали, когда искали убитых игроков, для сброса их баллов в таблице лидеров до нуля:

```
33  local function onCharacterAdded(character, player)
34      player:SetAttribute("IsAlive", true)
35      local humanoid = character:WaitForChild("Humanoid")
36      humanoid.Died:Connect(function()
37          local points = player.leaderstats.Points
38          points.Value = 0
39          player:SetAttribute("IsAlive", false)
40      end)
41  end
```

Функция *onPlayerJoin* — загрузки данных входящих в игру игроков

Эту функцию мы немного изменили:

```
42  local function onPlayerJoin(player)
43      local leaderstats = Instance.new("Folder")
44      leaderstats.Name = "leaderstats"
45      leaderstats.Parent = player
46      points.Parent = leaderstats
47      player.CharacterAdded:Connect(function(character)
48          onCharacterAdded(character, player)
49      end)
50      local playerID = player.UserId
51      local success, Gold = pcall(function()
52          return goldStore:GetAsync(playerID)
53      end)
54      if success then
55          playerGold[playerID] = Gold or 0
56          points.Value = playerGold[playerID]
57          print(playerID.." Gold=".. Gold)
58      end
59  end
```

— в строках 43…45 мы делаем все то, что уже делали **в главе 12** в программе начисления и списания игрокам баллов, а именно: создаем специальную папку для хранения баллов и определяем ее Родителя (то есть, что она будет храниться в **player**);

Но, затем, вы видите, что мы не создаем локальную переменную **points.** Она уже была создана нами в строках 8…10. Мы даже использовали эту переменную в функции **GoldTouch**, и выбор прост: либо заводить (объявлять) такую переменную в каждой функции, где она будет нужна, либо завести ее в самом начале скрипта — тогда ее можно использовать без дополнительного объявления в любом месте скрипта. Именно так мы и поступили. Мы только указываем (в строке 46), что Родителем переменной **points** является папка **leaderstats**, которую мы завели перед этим;

— в строках 47…49, также как и раньше, создаем в функции **onPlayerJoin** переменную **player.CharacterAdded** (ПерсонажДобавлен) и подключаем ее к безымянной функции, которая в свою очередь подключает функцию **onCharacterAdded** к **character** каждого добавленного игрока. Это дает возможность функции **onCharacterAdded** определять убитых игроков и списывать их баллы до “0”. Заметим, что если убитый в игре игрок, зайдет в игру еще раз, его ранее накопленные баллы не будут потеряны.

— в строках 50…58 дополняем скрипт функции **onPlayerJoin** тем, чему мы научились для сохранения данных между сеансами игры, а именно:

— в строке 50 создаем локальную переменную **playerID** и присваиваем ей уникальный номер игрока вошедшего в игру;

— в строках 51…53 передаем переменной **Gold** золото из “главного” хранилища данных **goldStore**, принадлежащее этому игроку (если он ранее играл в эту игру). При успешной передаче золота функция **pcall** присвоит переменной **success** значение **true**;

— оператор **if - then** в строках 54…58 проверяет, равна ли переменная **success** значению **true**, и если это так:

— в строке 55 величина **Gold** сохраняется в “списке-хранилище” **playerGold** под уникальным номером игрока. Если же игрок ранее в игру не играл, то в “списке-хранилище” **playerGold** под уникальным номером этого игрока будет записано количество золота равное “0”;

— в строке 56 в таблицу лидеров **leaderstats** будет записано количество баллов, соответствующее количеству золота в “списке-хранилище”;

— строка 57 распечатывает количество золота принадлежащее игроку.

Функция *saveData* — сохранения данных в “главное” хранилище

Рассмотрим скрипт:

```
60    local function saveData(playerID, Gold)
61            local success, errorMessage = pcall(function()
62                    goldStore:SetAsync(playerID, Gold)
63            end)
64            if not success then
65                    warn(errorMessage)
66            end
67    end
```

У функции указаны два параметра **(playerID, value)** — уникальный номер игрока и золото, соответствующие этому уникальному номеру. В строках 61…63 под контролем функции **pcall** переданное функции золото передается в элемент списка “главного” хранилища **goldStore**, который соответствует уникальному номеру игрока. Если передача данных в хранилище проведена успешно, переменная **success** принимает значение **true**. В противном случае **pcall** выдает предупреждение об ошибке **warn(errorMessage)**, которое можно увидеть в окне **Output**.

Функция сохранения данных при выходе игрока из игры

```
68    local function onPlayerRemoving(player)
69            local playerID = player.UserId
70            if playerGold[playerID] then
71                    saveData(playerID, playerGold[playerID])
72            end
73    end
```

Функция объявляет локальную переменную **playerID** и приравнивает ее значение уникальному номеру игрока (см. строку 69). Далее, оператор **if-then** проверяет наличие в “списке-хранилище” **playerGold** данных, соответствующих игроку с таким уникальным номером. Если та-

кие данные в “списке-хранилище” есть, то вызывается функция **saveData**, которая и сохраняет данные в “главном” хранилище.

Вот и все. Осталось только подключить к сервису **Players** функции **onPlayerJoin** и **onPlayerRemoving**, что мы и сделаем в строках 73, 74:

```
74  Players.PlayerAdded:Connect(onPlayerJoin)
75  Players.PlayerRemoving:Connect(onPlayerRemoving)
```

Запустите игру. Вы можете собирать золотые самородки, игра будет начислять вам за них баллы и ваши баллы будут сохраняться, если вы будете выходить из игры, а затем вновь заходить в нее. А если в процессе игры ваш аватар погибнет, его баллы после того, как он заспавнится вновь, будут сброшены до нуля. Но, если вы выйдете из игры и вновь войдете, ваши ранее накопленные баллы будут восстановлены.

Конечный скрипт программы:

```
1   local Gold_Folder = workspace:WaitForChild("GoldChunks")
2   local ch_array = Gold_Folder:GetChildren()
3   local Players = game:GetService("Players")
4   local DataStoreService = game:GetService("DataStoreService")
5   local goldStore = DataStoreService:GetDataStore("Gold")
6   local playerGold = {}
7   local gold_VAL = 10
8   local points = Instance.new("IntValue")
9   points.Name = "Points"
10  points.Value = 0
11  local function GoldTouch(chunk, gold_VAL, player)
12      chunk.Parent = nil
13      local playerID = player.UserId
14      if playerGold[playerID] then
15          playerGold[playerID] = playerGold[playerID] + gold_VAL
16          points.Value = points.Value + gold_VAL
17      end
18      wait(2)
19      local next_X =  Random.new(-72,72)
20      local next_Z =  Random.new(-72,72)
21      chunk.CFrame = CFrame.new(next_X, 50, next_Z)
22      chunk.Parent = workspace.GoldChunks
```

```
23    end
24    for _, chunk in pairs(ch_array) do
25            chunk.Touched:Connect(function(otherPart)
26                    local partParent = otherPart.Parent
27                    local player = Players:GetPlayerFromCharacter(partParent)
28                    if player then
29                            GoldTouch(chunk, gold_VAL, player)
30                    end
31            end)
32    end
33    local function onCharacterAdded(character, player)
34        player:SetAttribute("IsAlive", true)
35        local humanoid = character:WaitForChild("Humanoid")
36        humanoid.Died:Connect(function()
37            local points = player.leaderstats.Points
38            points.Value = 0
39            player:SetAttribute("IsAlive", false)
40        end)
41    end
42    local function onPlayerJoin(player)
43            local leaderstats = Instance.new("Folder")
44            leaderstats.Name = "leaderstats"
45            leaderstats.Parent = player
46            points.Parent = leaderstats
47            player.CharacterAdded:Connect(function(character)
48                    onCharacterAdded(character, player)
49            end)
50            local playerID = player.UserId
51            local success, Gold = pcall(function()
52                    return goldStore:GetAsync(playerID)
53            end)
54            if success then
55                    playerGold [playerID] = Gold or 0
56                    points.Value = playerGold [playerID]
57            end
58    end
59    local function saveData(playerID, Gold)
60            local success, errorMessage = pcall(function()
61                    goldStore:SetAsync(playerID, Gold)
62            end)
```

```
63              if not success then
64                  warn(errorMessage)
65              end
66      end
67      local function onPlayerRemoving(player)
68              local playerID = player.UserId
69              if playerGold [playerID] then
70                  saveData(playerID, playerGold [playerID])
71              end
72      end
73      Players.PlayerAdded:Connect(onPlayerJoin)
74      Players.PlayerRemoving:Connect(onPlayerRemoving)
```

16. Инструменты и стреляющее оружие



Рис. 16.1 Создание инструмента

Инструменты — это то, что игрок может держать в руке и использовать в игре. Это может быть оружие или, даже, продукты питания. Инструменты можно хранить в игровом мире или раздавать игрокам. В этой главе вы узнаете, как создать лазерный бластер.

Любой инструмент в Roblox исходно называется **Tool**. Давайте создадим его:

— вставьте новый объект в **Workspace**. Для этого кликните по кнопке ⊕ правее **Workspace** и выберите в выпадающем меню **Tool** — появится объект инструмент с таким же названием;

— измените его название на **Blaster**. Кликните по кнопке ⊕ правее **Blaster**, выберите в меню объект **Meshpart** и вставьте его в **Blaster**. Кратко поясним: **Meshpart** — это 3D-модель, разработанная в другой специальной программе, например, в **Unity**, но найти которую можно на сайте Roblox. В нашем случае, мы используем 3D-модель бластера;

— переименуйте **Meshpart** в **Handle**(Рукоятка). Без такой рукоятки игрок не сможет взять в руки никакой инструмент.

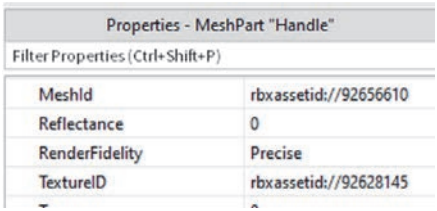


Рис. 16.2 Ввод данных в окно свойств Meshpart “Handle”

— откройте окно **Properties** вставленного **Meshpart** и укажите:
свойству **MeshID** значение **rbxassetid://92656610**, а свойству **Textureid** значение **rbxassetid://92658105**;

Запустите программу. Теперь наш бластер является дочерним элементом **Workspace**, и игрок может взять его, коснувшись бластера. После этого, бластер станет “дочерним элементом” модели персонажа и будет помещен на панель быстрого доступа — в нижней части экрана появится рамочка с названием инструмента и с его порядковым номером. А поскольку других инструментов еще не было создано, наш бластер будет иметь порядковый номер “1”.

Если кликнуть по этой рамочке, то найденный игроком инструмент будет положен в “рюкзак” игрока. Этот “рюкзак” видно не будет, но игрок всегда сможет достать его оттуда в нужный момент, нажав мышкой на рамочку инструмента. Также можно доставать и прятать инструменты в рюкзак, нажимая на клавиатуре кнопку, цифра которой соответствует номеру инструмента. Если же нажать кнопку **Backspace**, игрок просто выбросит инструмент и не сможет пользоваться им, пока повторно не найдет его и не прикоснется к инструменту.

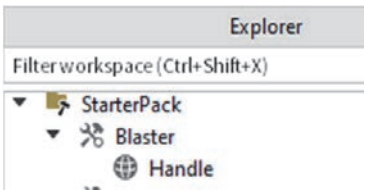


Рис. 16.3 Перемещение инструмента в папку Starter Pack

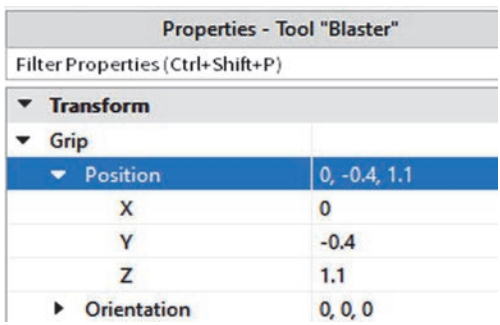
Чтобы каждый раз не искать, где в игровом пространстве лежит наш бластер, наведите на него курсор мыши, нажмите правую кнопку мыши и, не отпуская кнопки, переместите бластер в папку **StarterPack**. Теперь бластер будет всегда появляться в “рюкзаке” вашего игрока при начале игры.

Позиция и ориентация инструмента

Обратите внимание: игрок держит бластер не за ручку, а за центральную точку бластера, поэтому рука игрока, как бы входит внутрь бластера, что не красиво.



Рис. 16.4 Корректировка положения оружия в руке аватара



Положение и ориентацию инструмента можно изменить, изменяя свойства бластера.

Рис. 16.5 Свойства Grip бластера

Откройте окно **Properties** бластера и задайте свойству **Grip**: **Position** значение 0, -0,4, 1,1. Запустите игру — вы увидите, что теперь игрок уже держит бластер за его рукоятку. При необходимости, свойство **Grip: Orientation** может менять и ориентацию инструмента в пространстве.

Можно в рамочке экранного меню инструментов заменить название инструмента на его “картинку”. Для этого в окне свойств **Tool Blaster** нужно указать в качестве свойства **TexturedID** значение **rbxassetid://92628145**.

Каждый инструмент имеет индикаторы трех событий, которые можно использовать в программировании игр:

— индикатор **Equipped** (Выбран) активируется, когда игрок выбирает инструмент на панели быстрого доступа;

— индикатор **Unequipped** (Не выбран) активируется, если инструмент не используется (не выбран на панели быстрого доступа);

— индикатор **Activated** (Действует) активируется, когда игрок щелкает левой кнопкой мыши (например, когда, стреляет из оружия).

Добавление звуков, соответствующих событиям

Чтобы увидеть, как работают индикаторы событий инструмента, можно например, подключить их для воспроизведения звуков, соответствующих тому, или иному событию.

Давайте, создадим два звука, соответствующих событиям **Equip** и **Activated**. Кликните по кнопке ⊕ правее объекта **Handle** и выберите в выпадающем меню объект **Sound**. Повторите эту операцию и вставьте в **Handle** еще один объект **Sound.** Переименуйте один звук в **Equip**, откройте его окно **Properties** и задайте его свойству **SoundId** значение **rbxassetid://282906960**.

Переименуйте второй звук в **Activate** и задайте его свойству **SoundId** значение **rbxassetid://130113322**.

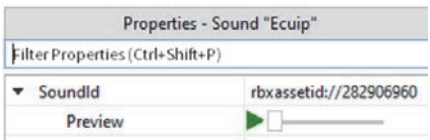


Рис. 16.6 Регулировка громкости

В окне **Properties** можно нажать зеленую кнопку **Prewiew** для предварительного прослушивания звука и отрегулировать нужную громкость звучания (см. параметр **Volume**) и скорость воспроизведения обоих звуков (см. параметр **PlaybackSpeed**).

Если громкость и скорость воспроизведения звуков вас устраивает, кликните по кнопке ⊕ правее объекта **Handle**, выберите в выпадающем меню объект *локальный скрипт* **LocalScript** (а не **Script**) переименуйте его в **BlasterScript** и вставьте в него следующий код:

```
1    local tool = script.Parent
2    local function Equipped()
3        tool.Handle.Equip:Play()
4    end
5    local function Activated()
6        tool.Handle.Activate:Play()
7    end
8    tool.Equipped:Connect(Equipped)
9    tool.Activated:Connect(Activated)
```

Вы поняли? Мы подключили существующие у всех инструментов индикаторы **Equipped** и **Activated** к функциям с аналогичными названиями.

Запустите программу и опробуйте ваш бластер. Работает? Теперь вы можете аналогичным образом создавать другие инструменты.

“Рэйкастинг”: учим бластер стрелять

Бластер, пистолет или пушка могут стрелять, используя для определения пораженных выстрелом целей, функцию **Raycast** (БросанияЛучей). Эта функция создает невидимый луч в заданном направлении на заданную дистанцию. Если луч сталкивается с каким-то объектом на своем пути (с игроком, танком, местностью или др.), он получает информацию об объекте, с которым он столкнулся и о местоположении этого объекта.

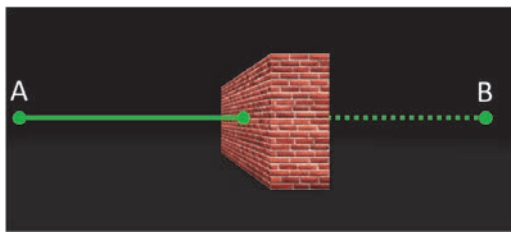


Рис. 16.7 Raycast

Вы можете направить такой луч, а затем получить результаты рэйкастинга **Result**, просто указав с помощью **Vector3** начальные координаты луча **rayOrigin** и конечные координаты луча **rayDestination**:

```
local rayOrigin = Vector3.new(0, 0, 0)
local rayDestination = Vector3.new(0, -100, 0)
local Result = workspace:Raycast(rayOrigin, rayDestination)
```

Заметим, что максимальное расстояние между начальной и конечными точками луча не может превышать 5000 шипов.

Функция **Raycast()** может иметь необязательный объект **RaycastParams**, который позволяет выборочно учитывать или игнорировать определенные **parts** или материалы или даже цвета при рэйкастинге. Иначе, можно, например, случайно попав в **Baseplate**, **SpawnLocation** или **Terrain,** непреднамеренно, уничтожить и их. По сути **RaycastParams** это несколько фильтров (см. табл.1):

Таблица 1. Фильтры рэйкастинга

Тип фильтра	Описание
FilterDescendantsInstances	Список объектов, чьи потомки не будут учитываются при рэйкастинге
FilterType	**Blacklist** (Черный список) — при рэйкастинге будут учтены все **parts**, кроме тех, которые являются потомками объектов, указанных в списке.
FilterType	**Whitelist** (Белый список) — при рэйкастинге будут учитываться только **parts**, которые являются потомками объектов, указанных в списке
IgnoreWater (Игнорировать Воду)	Логическое значение, определяющее, учитывается ли **Water** при рейкастинге

Для организации фильтрации объектов при рэйкастинге нужно:
1/ создать новый луч рэйкастинга, объявив локальную переменную **NewRay** (НовыйЛуч) и присвоить ей значение **RaycastParams.new()**:

```
local NewRay = RaycastParams.new();
```

2/ указать переменную **NewRay** в параметрах результата рэйкастинга **Result**;

```
local Result = workspace:Raycast(rayOrigin, rayDestination, NewRay)
```

3/ чтобы указать объекты, исключенные из рейкастинга, нужно привести их в виде списка **Filter Descendants Instances** (список обозначен фигурными скобками):

```
NewRay.FilterDescendantsInstances = {}
```

Например, чтобы игрок не смог случайно подстрелить самого себя, уничтожить **SpawnLocation**, **Baseplate** или ландшафт (**Terrain**), можно указать все эти объекты в списке:

```
NewRay.Filter Descendants Instances{Player.Character, workspace.SpawnLocation, workspace.Baseplate, workspace.Terrain};
```

4/ если же вы захотите использовать **Blacklist** или **Whitelist**, то нужно указать эти фильтры следующим образом:

```
NewRay.FilterType = Enum.RaycastFilterType.Blacklist
```

5/ если вы хотите определить результат рэйкастинга при стрельбе по воде, нужно указать:

```
NewRay.IgnoreWater = false
```
или
```
NewRay.IgnoreWater = true
```

Определяем координаты цели

Прежде чем выстрелить из бластера, нужно знать, куда же игрок целится. То есть, функция **Raycasting** должна получить в качестве одного из исходных данных координаты цели. Тогда луч столкнется с тем, на что игрок наводит мышь. Естественно, что в Roblox предусмотрен ввод информации от мыши. В частности, есть функция **Player:GetMouse** (ПолучитьСобытияМыши).

Добавьте к скрипту бластера несколько строк, чтобы программа, при каждом щелчке левой кнопкой мыши, печатала координаты мыши и название **parts**, на которые указал курсор мыши:

```
1   local Player = game:GetService("Players").LocalPlayer
2   local Mouse = Player:GetMouse()
3   local tool = script.Parent
4   local function Equipped()
5       tool.Handle.Equip:Play()
6   end
7   local function Activated()
8       tool.Handle.Activate:Play()
9       print(Player.Name,"clicked at position:", tostring(Mouse.Hit.
  Position))
10      if Mouse.Target then
11          print("Clicked part: ", Mouse.Target:GetFullName())
12      else
13          print("No part clicked")
14      end
15  end
16  tool.Equipped:Connect(Equipped)
17  tool.Activated:Connect(Activated)
```

Рассмотрим этот скрипт:

— в строке 1 мы подключили сервис **Players**. Заметили, что в локальном скрипте сервис подключается с добавлением **.LocalPlayer**?**;**

— в строке 2 мы подключили функцию обработки событий и данных мыши. Отметим, что мы специально использовали для размещения программы инструмента **локальный скрипт** потому, что функция **Get.Mouse()**, в отличие от других, работает только в **LocalScript**;

— в строках 9…15 используем данные, полученные от функции **GetMouse()**:

— в строке 9 команда **print** сначала печатает имя локального игрока **Player.Name**, затем, печатает слова “**clicked at position:**” (“кликнул по позиции:”), затем печатает координаты позиции, по которым кликнула мышь.

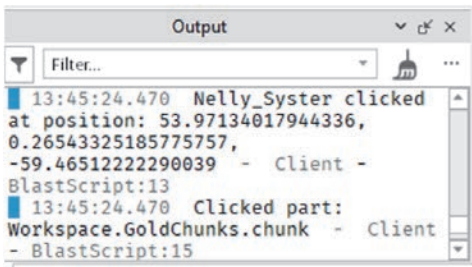


Рис. 16.8 Распечатка координат кликов мышки и имен целей

Функция **print** предназначена для распечатки текстовых (или “строчных”) данных. А координаты курсора мыши, естественно, являются числовыми данными. Поэтому, в функции **print** использован оператор “**tostring(...)**”, который преобразовывает в текст любые числовые данные — в нашем случае он преобразовывает в текст координаты курсора мыши.

— в строках 10…14 функция проверяет кликнула ли мышь по какой-либо цели (**Target**) и если это так, то (в строке 11) печатает полное имя **Target**, а если это не так (см. оператор **else** строки 12), то в строке 13 печатает текст **"No part clicked"** (“Нет кликнутых part”).

Откройте в Roblox Studio окно **Output**, запустите программу, покликайте мышью и посмотрите, что будет печатать Roblox в окне **Output**. Здорово!? Обратите внимание на очень высокую точность с которой Roblox определяет координаты.

Время подумать

Подведем промежуточный итог: мы создали бластер, научили его издавать звуки, поняли как можно указать ему цель. Но, подумаем: наш игрок собирается стрелять в другого игрока, а, как аватар другого игрока, чей компьютер находится в другом городе или даже в другой

стране, должен понять, что в него стреляют и выстрел даже попал в него? Как все происходит? Отвечаем: реализуется все это при помощи **сервера** Roblox.



Рис. 16.9 Клиенты-Сервер

Сервер - это компьютер с высокой вычислительной мощностью, специальным программным обеспечением и круглосуточной бесперебойной работой. Он предназначен для хранения информации и обеспечения доступа к ней с устройств всех игроков (**клиентов** сервера).

В программном обеспечении серверов предусмотрена **синхронизация** (точное по времени совпадение) событий и содержимого нескольких копий игры, запущенных на разных компьютерах. Синхронизация выполняется копированием многих самых разных данных из каждого игрового компьютера на сервер и, наоборот, с сервера — на множество игровых компьютеров. Такое копирование называется **репликацией**. При репликации изменения, сделанные в одной копии игры, передаются в другие копии игры.

Как правило, любые изменения, сделанные на компьютерах игроков (**клиентах** сервера), сохраняются только на самих **клиентах**, но **не реплицируются** на сервер и другие клиенты. Изменения на каком-либо клиенте можно будет увидеть на других клиентах только, если сервер сделает что-то, что перезапишет эти изменения на других клиентах. Например, когда игрок нажимает кнопку “**W”**, чтобы двигаться вперед, данные с его с клавиатуры поступают на его **компьютер-клиент**. Затем этот клиент сообщает об этом серверу, который в ответ перемещает позицию персонажа игрока вперед, и это передвижение видят все клиенты, включенные в игру.

Связь “клиент-сервер”

Для отправки сообщений от **клиента серверу** используют объекты типа **Remote Event** (Удаленное Событие). Создаются такие объекты в

общем сервис-контейнере **ReplicatedStorage** (ХранилищеРепликации), доступном как серверу, так и игровым клиентам.

Каждое удаленное событие **RemoteEvent** должны обслуживать две функции, одна из которых должна быть запущена на клиенте, а другая — на сервере:
— *функция запускаемая на клиенте:*

```
RemoteEvent: FireServer (параметры)
```

— *функция запускаемая на сервере:*

```
RemoteEvent.OnServerEvent: Connect (function (player, параметры))
```

Отметим, что запускаемая на клиенте функция **FireServer** будет работать только если она создана в **LocalScript**. Если функции **FireServer()** передаются какие-то параметры, то на сервере они будут переданы программе-обработчику событий. Первым параметром, который будет использовать обработчик событий сервера всегда является объект **player** клиента, запустившего удаленное событие, остальные параметры должны быть указаны вслед за ним.

Давайте используем то, с чем мы познакомились сейчас:

— чтобы создать *удаленное событие*, соответствующее выстрелу из бластера, наведите в окне **Explorer** указатель мыши на объект **ReplicatedStorage** и нажмите кнопку ⊕. Выберите в выпадающем меню **RemoteEvent** и переименуйте его, например, в **Shoot** (Выстрел), чтобы было понятно назначение этого события;

— внутри локального скрипта **BlasterScript** создайте (см. ниже строки 4…6) *клиентскую функцию* **Pew()** отправки данных на сервер при возникновении удаленного события **Shoot**, а затем, вставьте эту функцию в тело функции **Activated()** (см. строку 11):

```
1    local Player = game:GetService("Players").LocalPlayer
2    local Mouse = Player:GetMouse()
3    local tool = script.Parent
4    local function Pew()
5        game.ReplicatedStorage.Shoot:FireServer(Mouse.Hit.p,
  tool.Handle)
6    end
```

```
7       local function Equipped()
8               tool.Handle.Equip:Play()
9       end
10      local function Activated()
11              Pew()
12              tool.Handle.Activate:Play()
13              print(Player.Name,"clicked at position:", tostring(Mouse.
    Hit.Position))
14              if Mouse.Target then
15                      print("Clicked part: ", Mouse.Target:GetFullName())
16              else
17                      print("No part clicked")
18              end
19      end
20      tool.Equipped:Connect(Equipped)
21      tool.Activated:Connect(Activated)
```

Этот скрипт является *конечным скриптом* **BlasterScript**. Теперь при каждой активации бластера событие **Shoot** будет запускать функцию **FireServer**, передавая на сервер аргументы, указанные внутри ее скобок (координаты курсора мыши и указание на инструмент бластер);

— чтобы создать *серверную функцию* обработки того же удаленного события **Shoot**, наведите в окне **Explorer** указатель мыши на **ServerScriptStorage** и нажмите кнопку ⊕. Выберите в выпадающем меню **Script**. Переименуйте новый скрипт, например, в **Blaster_Handler** (Обработчик_Бластера) и вставьте в этот скрипт приведенный ниже код:

```
1       local RANGE = 500
2       local DAMAGE = 20
3       game.ReplicatedStorage.Shoot.OnServerEvent:Connect(function
(Player, Target, Handle)
4               if Player.Character == nil then
5                       return
6               end
7       end)
```

— в первых двух строках кода указываем, что максимальная дальность стрельбы **RANGE** составляет 500 шипов, а величина урона **DAMAGE**, наносимая выстрелом бластера игроку равна 20% здоровья:

— в строках 3…7 разместим заготовку серверной функции обработки удаленного события **Shoot**;

— внутри серверной функции, в строках 4…6, вставим оператор **if-then**, который будет проверять существует ли модель персонажа игрока и, если модель персонажа еще не сформирована (отсутствует и равна **nil**), будет повторять такую проверку (**return -** возвращаться), пока модель, наконец, не появится;

Приведенный скрипт мы назвали “заготовкой” потому, что ниже строки 6 этого скрипта и записью “**end)**”, обозначающей конец функции, нам нужно будет разместить: скрипт, рисующий трассу выстрела бластера и скрипт рэйкастинга.

Скрипт, рисующий трассу выстрела бластера

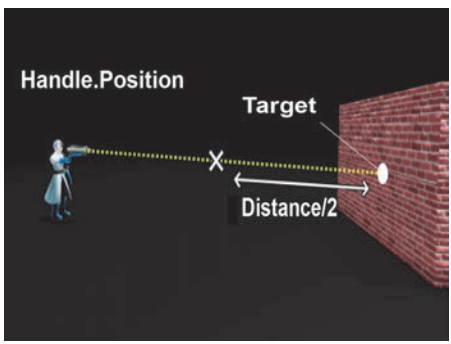


Рис. 16.10 Трасса выстрела

Как написать такой скрипт? — Очень просто! Нужно описать программой **part**, представляющий собой длинную тонкую палку, которая будет появляться при каждом выстреле, начинаться в бластере, кончаться в точке, указанной курсором мыши, иметь подходящий цвет и будет пропадать (уничтожаться) сразу после выстрела. Вот этот скрипт:

```
1    local Distance = (Handle.Position - Target).magnitude
2    local laser = Instance.new("Part", workspace)
3        laser.BrickColor = BrickColor.new("Electric blue")
4        laser.Size = Vector3.new(0.1, 0.1, Distance)
5        laser.Material = "Neon"
6        laser.Transparency = 0.25
```

```
7           laser.Anchored = true
8           laser.CanCollide = false
9           laser.CFrame = CFrame.new( Handle.Position, Target)* CFrame
   .new(0, 0, -Distance/2)
10    game:GetService("Debris"):AddItem(laser, 0.1)
```

— в строке 1 рассчитаем **Distance** — длину нашей “палки”, вычитая из координат центра бластера координаты цели, указанной курсором мыши. Слово **magnitude** здесь обозначает длину этого направленного отрезка прямой;
— в строке 2 создаем в **workspace** объект типа **part** и присваиваем его значение переменной **laser**. В последующих строках задаем этой переменной необходимые свойства:
— в строке 3 — “электрический голубой” цвет;
— в строке 4 — размеры **Size** (толщину, равную 0,1 х 0,1 шага и длину, равную **Distance)**;
— в строке 5 — свечение, присущее материалу **Neon**;
— в строке 6 — полупрозрачность;
— в строке 7 — обязательно “якорим” **part**, чтобы эта **part** не падала;
— в строке 8 — разрешаем не быть **CanCollide**, чтобы наша трасса не была жесткой и не сталкивалась с другими объектами;
— в строке 9 указываем координаты, в которые нужно поместить трассу, при этом используем следующую запись:
CFrame.new(начало, конец)*CFrame.new(0,0, - половина длины “палки”).
Это позволяет поместить центр “палки” точно посредине дистанции между бластером и целью;
— в строке 10 уничтожаем созданный нами **part**, вызывая специальный сервис **Debris**, которому указываем, что он должен уничтожить **part** с именем **laser** через 0,1 секунды.

Скрипт рэйкастинга

Создаем новый луч и определяем параметры фильтрации рэйкастинга:

```
…  local NewRay = RaycastParams.new()
…  NewRay.FilterDescendantsInstances = {Player.Character, workspace.
 SpawnLocation, workspace.Baseplate, workspace.Terrain}
…  local raycastResult = workspace:Raycast(rayOrigin, rayDirection, NewRay)
```

Для рэйкастинга нужны координаты начальной точки и конечной точек луча. В качестве координат начала луча мы используем центр бласте-

ра — то есть, **Handle.Position**. Координаты цели **Target** дает серверу мышь. Значит, можно вычислить **вектор** (“стрелку”) направления стрельбы и указав его максимально возможную длину, присвоить значение этого вектора переменной **rayDirection**:

```
    local rayDirection = (Target - Handle.Position)* RANGE.
```

Ищем результат рэйкастинга и присваиваем его переменной **Result**:

```
    local Result = workspace:Raycast(Handle.Position, rayDirection, NewRay)
```

Если результат рейкастинг существует (**Result = true**), то присваиваем свойство **Instance** объекта **Result** локальной переменной **hitPart (**ПодбитаяPart**)**. Это свойство объекта **Result** соответствует **part**, которую пересек луч рэйкастинга. Затем (в целях отладки программы с использованием окна **Ouput**) печатаем наименование найденного рэйкастингом объекта. Чтобы было “эпичнее” заставляем **hit.part** дымиться, ждем долю секунды и поджигаем ее, а затем, через секунду уничтожаем ее:

```
if Result then
        local hitPart = Result.Instance
        print(hitPart)
        Instance.new("Smoke",hitPart)
        wait(0.1)
        Instance.new("Fire",hitPart)
        wait(1)
        hitPart:Destroy()
```

Согласитесь, стреляя в игрока или в танк вы, очевидно, попадаете не во что-то “монолитное”. Мы уже рассказывали, что части тела являются составными частями объекта **Character** (Персонаж) игрока, включающего в себя:

- отдельные части тела игрока (голову, конечности и туловище);
- любую одежду и аксессуары, которые носит игрок;
- **Humanoid** — специальный объект, который содержит множество свойств, связанных с игроком, включая его здоровье;
- **HumanoidRootPart** — объект, который управляет движением игрока.

Аналогично, обстоит дело и с моделями других объектов. Например, модель того же танка состоит из многих частей. И, поскольку, функция рэйкастинга определяет именно ту часть модели объекта, в которую попал луч, то, чтобы понять является ли эта часть частью танка или игрока, требуется определить, какой модели принадлежит эта часть. Это можно сделать подключив к **hitPart** функцию **FindFirstAncestorOfClass ("Model")** (НайтиПервогоПредкаКласса“Модель”) и затем присвоить найденный результат переменной **model**:

```
    local model = hitPart:FindFirstAncestorOfClass("Model")
```

Затем, если модель найдена (**model = true**) нужно подключить к **model** функцию **FindFirstChild**, чтобы найти в этой модели объект **Humanoid**, включающий в себя здоровье игрока и уменьшить его здоровье на заранее установленную в программе величину:

```
    if model then
        if model:FindFirstChild("Humanoid") then
            model.humanoid.Health -= DAMAGE
        end
    end
```

Кстати, заметили, что написано не ”=”, а “-=”? Так и должно быть: эта форма записи показывает, что нужно уменьшить здоровье гуманоида на величину **DAMAGE**. Хотя, такая запись и не обязательна, можно было записать и по-другому — попробуйте записать именно по-другому.

После этого остается:
— завершить (словом **end**) цикл оператора **if Result then**;
— завершить (словом и скобкой “**end)**”) серверную функцию обработки удаленного события **Shoot**.
Ниже показан *конечный код серверной функции* обработки события **Shoot**, расположенный в скрипте **Blaster_Handler** папки **ServerScriptService**:

```
1   local RANGE = 500
2   local DAMAGE = 20
3   game.ReplicatedStorage.Shoot.OnServerEvent:Connect(function
  (Player, Target, Handle)
```

```
4               if Player.Character == nil then
5                       return
6               end
7               local Distance = (Handle.Position - Target).magnitude
8               local laser = Instance.new("Part", workspace)
9               laser.BrickColor = BrickColor.new("Electric blue")
10              laser.Size = Vector3.new(0.1, 0.1, Distance)
11              laser.Material = "Neon"
12              laser.Transparency = 0.25
13              laser.Anchored = true
14              laser.CanCollide = false
15              laser.CFrame = CFrame.new( Handle.Position, Target) * CFrame
   .new(0, 0, -Distance/2)
16              game:GetService("Debris"):AddItem(laser, 0.1)
17              local NewRay = RaycastParams.new()
18              NewRay.FilterDescendantsInstances = {Player.Character,
   workspace.SpawnLocation, workspace.Baseplate, workspace.Terrain}
19              local rayDirection = (Target - Handle.Position)* RANGE
20              local Result = workspace:Raycast(Handle.Position, rayDirection,
   NewRay)
21              if Result then
22                      local hitPart = Result.Instance
23                      print(hitPart)
24                      Instance.new("Smoke",hitPart)
25                      wait(0.1)
26                      Instance.new("Fire",hitPart)
27                      wait(1)
28                      hitPart:Destroy()
29                      local model = hitPart:FindFirstAncestorOfClass("Model")
30                      if model then
31                              if model:FindFirstChild("Humanoid") then
32                                      model.humanoid.Health -= DAMAGE
33                              end
34                      end
35              end
36      end)
```

Поэкспериментируйте с фильтрами рэйкастинга. Например, если вы укажете в списке **FilterDescendantsInstances** элемент **workspace.red**, то окажется, что бластер не наносит никакого вреда всем предметам

красного цвета. Они становятся, как бы прозрачными для луча рэйкастинга.



Рис. 16.11 Стрельба пулями

Бластер - это хорошо, но многие предпочитают обычное огнестрельное оружие — пистолеты, винтовки и автоматы. А они стреляют не лучом, а пулями. Можно ли доработать скрипт бластера, чтобы было видно, что он стреляет пулями? Конечно, можно.

Поменяйте код в скрипте **Blaster_Handler** папки **ServerScriptService**:

— в строке 10 было: `laser.Size = Vector3.new(0.1, 0.1, Distance);`
— измените строку 10 на `laser.Size = Vector3.new(0.1, 0.1, 2)` этим мы установим длину пуль равной всего 2 шипам;
— в строке 15 было:

```
15      laser.CFrame = CFrame.new( Handle.Position, Target) * CFrame.new(0,
   0, -Distance/2)
```

— оставьте в строке 15 только это:

```
15      laser.CFrame = CFrame.new( Handle.Position, Target);
```

— ниже строки 15 вставьте цикл:

```
1       for i = 0,20,1 do
2               laser.CFrame = laser.CFrame * CFrame.new(0, 0, -4)
3               wait()
4       end
```

Во второй строке цикла цифра “-4” соответствует скорости пули — можно менять ее в небольших пределах (на ваш вкус). На скорость влияют и параметры цикла **for** — поэкспериментируйте, что происходит при их изменении. Попробовали пострелять? Нормально? Ну, и хорошо!

17. Перезарядка оружия

Настоящему оружию требуется периодическая перезарядка. Давайте дополним наш скрипт такой возможностью.



Рис. 17.1 Перезарядка оружия

В самом начале **BlasterScript** (выше строк, в которых описаны функции) объявим переменную **BULLETS** (число патронов в магазине оружия) и зададим ее значение, например, “5”.

В следующей строке создадим переменную RELOAD, и присвоим ей “строковое” значение "**reloadWeapon**". Мы будем использовать лишь имя этой переменной, как обозначение действия, и содержание строки "**reloadWeapon**", в принципе, ни на что не влияет. Создадим также функцию с именем **onRELOAD** и двумя параметрами: **action** (действие) и **input** (тип ввода данных). Эта функция будет осуществлять все действия по перезарядке, а именно:

— увеличение количества патронов в магазине оружия;
— воспроизводить звук перезарядки оружия;
— изменять пиктограмму в меню инструмента, показывая, что идет перезарядка:

```
1    local BULLETS = 5;
2    local RELOAD = "reloadWeapon"
3    local function onRELOAD(action, input)
4        if action == RELOAD and input == Enum.UserInputState.Begin
  then
5            tool.TextureId = "rbxassetid://6593020923"
6            tool.Handle.RELOAD:Play()
7            BULLETS = 5
8            wait(2)
9            tool.TextureId = "rbxassetid://92628145"
10       end
11   end
```

— в строке 4 оператором **if-then** проверяем, соответствует ли переданный функции **onRELOAD** параметр **action** значению **RELOAD**, а параметр **input** значению **UserInputState.Begin**.

Тип значения **UserInputState.Begin** соответствует вводу данных типа “нажатие клавиши” или “нажатие кнопки мыши”. Такая проверка нужна, чтобы функция **onRELOAD** запускалась каждый раз, когда изменяется **input**, но перезарядка происходила только один раз;



Рис. 17.2 Пиктограмма “часы” во время перезарядки оружия

— далее, если оба эти условия выполнены:
— в строке 5, чтобы было явно видно, что игрок перезагружает бластер, на пару секунд изменяем свойство **TextureId** бластера на пиктограмму “часы” «**rbxassetid://6593020923**»;
— в строке 6 включаем звук перезарядки бластера;
— в строке 7 пополняем магазин оружия патронами;
— в строке 8 ждем пару секунд;
— в строке 9 возвращаем исходную пиктограмму “бластер”, присваивая свойству **TextureId** бластера прежнее значение «**rbxassetid://92628145**»;

Чтобы добавить звук перезарядки оружия, заходим в **Explorer** и добавляем объекту **Handle** нашего бластера еще один звук (**sound**) и переименовываем его в **RELOAD.**

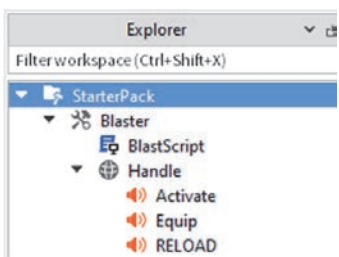


Рис. 17.3 Вставка звука RELOAD

Причем, мы не будем искать новый подходящий звук, а используем для перезарядки звук, который уже использовали в функции **Equip**, но изменим скорость его воспроизведения и громкость, чтобы эти звуки отличались друг от друга (попробуйте - это можно сделать);

Привязка действий к кнопкам клавиатуры

В Roblox имеется сервис **ContextActionService**, связывающий с помощью функции **BindAction** необходимые пользователю действия с конкретными кнопками ввода данных. Функция **BindAction** имеет несколько параметров:

— название действия, которое нужно выполнить;
— название функции, которая будет обрабатывать действия;
— указание о том, должна ли отображаться кнопка сенсорного экрана (это нужно, если игру загружают на смартфоне, а не на компьютере);
— название кнопки клавиатуры (**KeyCodes**), нажатие которой вызывает действие.

Чтобы запустить работу сервиса **ContextActionService** вставьте самой первой строкой **BlasterScript** следующий код. Видите, локальная переменная имеет точно такое же название, как и сервис:

```
local ContextActionService = game:GetService("ContextActionService")
```

Затем, дополните нашу (состоящую из одной строки) функцию **toolEquipped** скрипта **BlasterScript** следующей строкой:

```
    ContextActionService:BindAction(RELOAD, onRELOAD, true,
Enum.Key Code.R)
```

Эта строка:
— вызывает сервис **ContextActionService**;
— подключает к нему функцию **BindAction;**
— передает функции **BindAction** следующие параметры:
— название действия — **RELOAD**;
— название функции, выполняющей нужные действия — **onRELOAD**;
— указание о необходимости создания сенсорной кнопки — **true**;
— название клавиши “R” (**KeyCode.R**), которая должна вызывать перезарядку.

Загрузите программу и убедитесь, что при нажатии кнопки “R” в меню инструментов будет происходить смена картинки “бластер” на картинку “часы”. Однако, вы можете заметить проблему: кнопка “R” будет вызывать смену картинки даже если бластер не выбран игроком. Но, когда игрок убирает бластер в рюкзак, действие перезарядки бластера должно быть отключено.

Чтобы исправить ситуацию, создадим функцию с именем **toolUnequipped** (ИнструментОтложен), вызовем внутри этой функции сервис **ContextActionService**, подключим к нему функцию **UnbindAction** (ОтменаДействия) и передадим ей в качестве параметра имя действия, которое нужно отключить. И, конечно, не забываем, что нужно подключить вновь созданную функцию к индикатору события **Unequipped**, который имеется у каждого инструмента и который мы ранее не использовали:

```
    local function toolUnequipped()
        ContextActionService:UnbindAction(RELOAD)
    end
    tool.Unequipped:Connect(toolUnequipped)
```

Еще раз загрузите игру - теперь, если бластер не взят игроком в руки, уже нельзя включить перезарядку нажатием кнопки “R”. На этом организация процесса перезарядки бластера нами завершена.

Осталось организовать счетчик патронов бластера и каждый раз, когда бластер стреляет вести их учет. Когда боекомплект уменьшится до нуля, мы можем приостановить действие функции **toolActivated** и возобновить ее только после завершения перезарядки. Для этого, первой строкой внутри функции активации бластера **Activated()** вставляем оператор **if-then**, проверяющий не кончились ли патроны, и если патроны кончились, то команда **return** (вернуться к новой проверке условия), записанная в качестве “внутреннего кода” оператора **if**, приостановит работу функции **Activated()** до тех пор, пока количество патронов не станет отличным от нуля.

```
    if BULLETS == 0 then
        return
    end
```

Вот и все. Мы создали полноценный инструмент-бластер, который может укладываться в рюкзак, доставаться из рюкзака нажатием кнопки экранного меню, издавать звуки, соответствующие режимам его использования, стрелять и поражать цели и, как настоящее оружие, перезаряжаться.

Вот конечный вид скрипта **Blaster Script**:

```
1   local ContextActionService = game:GetService("ContextActionService")
2   local RELOAD = "reloadWeapon"
3   local BULLETS = 5
4   local Player = game:GetService("Players").LocalPlayer
5   local Mouse = Player:GetMouse()
6   local tool = script.Parent
7   local function Pew()
8       game.ReplicatedStorage.Shoot:FireServer(Mouse.Hit.p, tool.
  Handle)
9   end
10  local function onRELOAD(action, input)
11      if action == RELOAD and input == Enum.UserInputState.Begin
  then
12          tool.TextureId = "rbxassetid://6593020923"
13          tool.Handle.RELOAD:Play()
14          BULLETS = 5
15          wait(2)
16          tool.TextureId = "rbxassetid://92628145"
17      end
18  end
19  local function toolUnequipped()
20      ContextActionService:UnbindAction(RELOAD)
21  end
22  local function Equipped()
23      ContextActionService:BindAction(RELOAD, onRELOAD, true,
  Enum.KeyCode.R)
24      tool.Handle.Equip:Play()
25  end
26  local function Activated()
27      if BULLETS == 0 then
28          return
29      end
30      Pew()
31      BULLETS = BULLETS - 1
32      tool.Handle.Activate:Play()
33      print(Player.Name,"clicked at position:", tostring(Mouse.Hit.
  Position))
34      if Mouse.Target then
35          print("Clicked part: ", Mouse.Target:GetFullName())
36      else
```

```
37                  print("No part clicked")
38              end
39      end
40      tool.Equipped:Connect(Equipped)
41      tool.Activated:Connect(Activated)
42      tool.Unequipped:Connect(toolUnequipped)
```

18. Программируем ландшафт

Меню ландшафта Roblox Studio позволяют создавать очень реалистичные ландшафты, включающие моря, горы, реки, каньоны и опасные потоки лавы.



Рис. 18.1 Ландшафт

Создавая ландшафт, вы сможете,одновременно, создавать кирпичные стены наподобие Великой китайской стены, неприступные башни, глубокие пропасти, горы с пещерами и многое другое. Для этого вам даже не нужно будет конструировать все это из каких-либо **parts**. Меню ландшафта **Terrain Editor** открывается на вкладке **Home** «Главная».

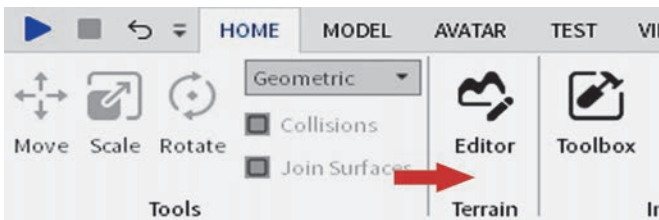


Рис. 18.2 Кнопка вызова меню ландшафта

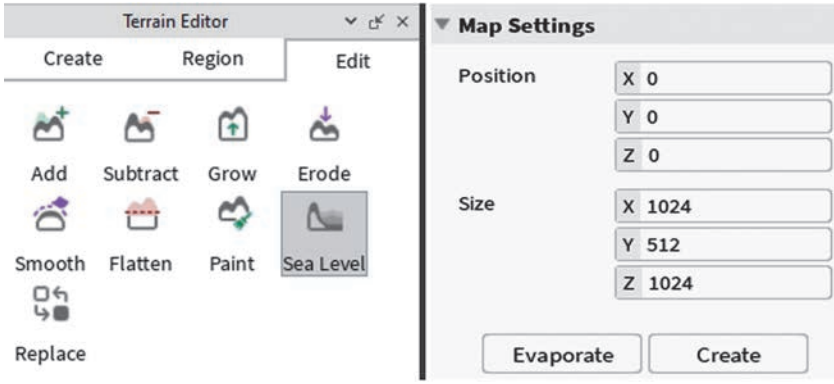


Рис. 18.3 Меню Edit ландшафта (слева) и меню Map Settings установки координат и размеров создаваемых водоёмов (справа)

Давайте, например, создадим небольшой водоем. В появившемся меню редактора ландшафта щелкните вкладку **Edit** (Редактировать) и выберите инструмент **Sea Level** (Уровень моря).

Появится меню **MapSetting**, позволяющее задать центр создаваемого водоема **Position** и его размеры **Size**. Напомним, что координаты X=0, Y=0 и Z=0 соответствуют исходному положению места спавна игрока при входе в игру. Оставьте координаты центра водоема без изменений. Укажите, для начала, небольшие размеры водоема, например. X=100, Y= 4, Z=100. Как вы поняли, “Y = 4” — это глубина водного пространства, но в нашем случае (поскольку вода будет выше отметки Y=0) это будет высота воды над уровнем нашей **Baseplate**. Нажмите кнопку **Create** (Создать).

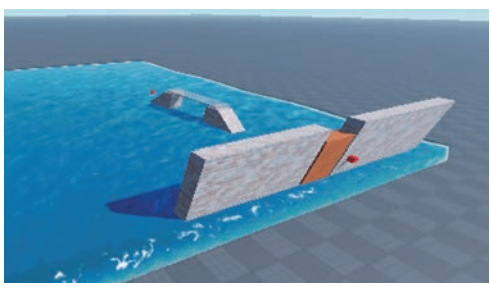


Рис. 18.4 Водоем без берегов, “затопивший” место спавна игроков.

Теперь, при запуске игры, вы увидите интересную картину: выбранное вами пространство залито водой, у которой нет берегов. А ваш аватар спавнится посреди этого водного пространства и после нескольких шагов плывет, а затем выходит на сушу!

Если, в последующем, вас не устроит созданный водоем, вы можете удалить его, повторно войдя в это меню, выбрав те же центр и координаты и нажав кнопку **Evaporate**(Осушить). Чуть позже мы подскажем, как удалить водоем другим способом.

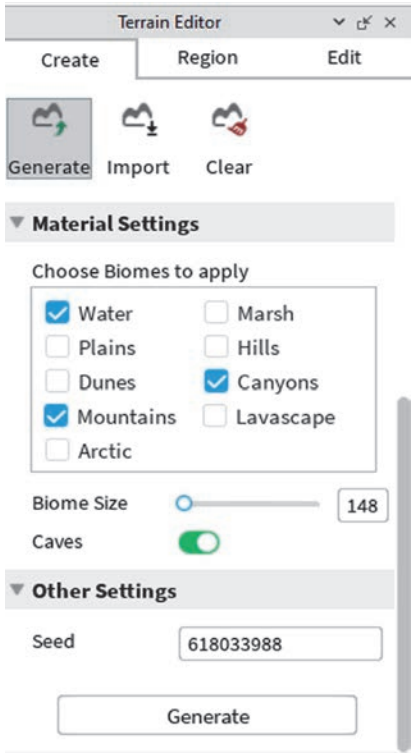


Рис. 18.5 Меню раздела Create

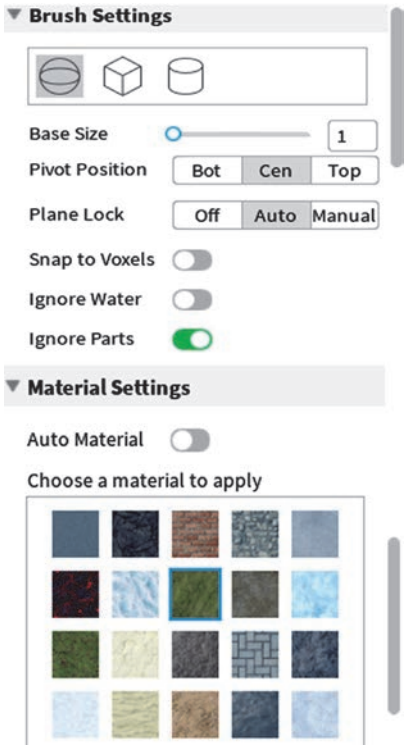
Раздел **Create** (Создать) **Terrain Editor** позволяет автоматически сгенерировать *случайный* ландшафт. Местоположение и размеры ландшафта устанавливаются также, как для водоемов. А ниже окон ввода координат и размеров находится меню выбора составляющих ландшафта, который вы хотите создать. Поставьте “галочки” у тех элементов, которые вы хотите видеть в своем ландшафте:

— водоемы **Water**;
— равнины **Plain**;
— дюны **Dunes**;
— горы **Mountains**;
— арктический ландшафт **Arctic**;
— болота **Marsh**;
— холмы **Hills**;
— каньоны **Canyons**;
— потоки лавы **Lavascape**;
— пещеры **Caves**.

Выбрав то, что нужно, нажмите кнопку **Generate** и дождитесь, когда ландшафт будет создан.

Нужно учесть, что если вы зададите небольшие размеры ландшафта, то вместить в него все, что вы выбрали не получится. Поэтому либо выбирайте более значительные размеры ландшафта, либо ограничьтесь лишь некоторыми из доступных составляющих. Войдите в игру и оцените созданный ландшафт. Если что-то сильно не понравится — войдите в меню **Create** и нажмите кнопку **Clear** (Очистить) — **все со-**

зданные ландшафты будут удалены. Однако, не торопитесь удалять и начинать новую генерацию ландшафта, а просто перейдите в раздел **Edit**.



Выбрав в разделе **Edit** режим **Add** (Добавить) (см. рис. 13.2), вы можете дорисовать то, чего вам не хватает.

Рис. 18.6 Меню параметров кисти режима Add

Для рисования вы можете выбрать три вида кисти: **шар**, **куб**, **цилиндр**, а также установить нужный вам размер кисти при помощи “движка” **Base Size**. И рисовать вы сможете не какими-нибудь красками, а прямо материалами. Поочередно наводите курсор мыши на цветные картинки меню **Material Settings** — рядом с курсором будут появляться названия этих материалов: вода, асфальт, трава, песок, камень, плитка, кирпич и др. Выберите тот материал, что подходит и рисуйте. Например, вы можете нарисовать остров в вашем водоеме или гору, или кирпичную стену, или башню — все, что хотите.

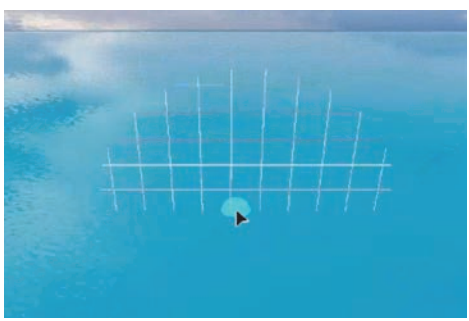


Рис. 18.7 Вертикальное расположение сетки кисти над поверхностью моря

Только нужно понять, что направление рисования ландшафта в режиме **Add** (а также направления иных воздействий на ландшафт в других

режимах меню **Edit**) будет определяться углом, под которым камера смотрит на ландшафт.

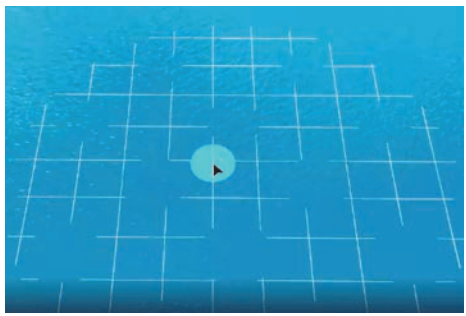


Рис. 18.8 Почти горизонтальное расположение сетки кисти

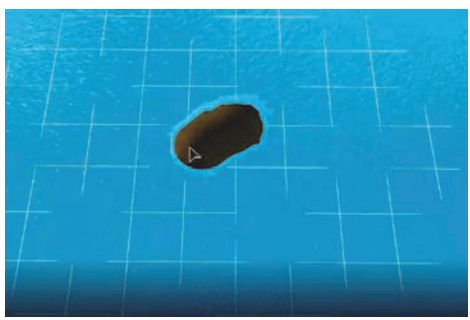


Рис. 18.9 Начало создания острова — при горизонтальном расположении сетки кисти

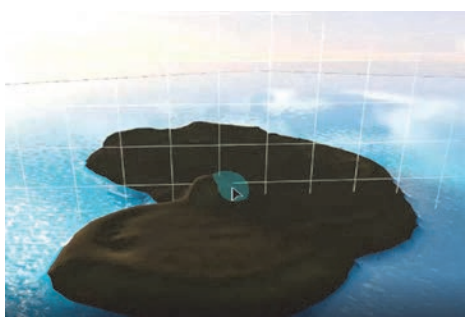


Рис. 18.10 Начало рисования горы на острове — при вертикальном расположении сетки кисти

Угол под которым смотрит камера, вы должны контролировать по углу наклона сетки белого цвета, возникающей при входе в меню **Edit**. Кисть всегда находится в центре этой сетки и рисует (или выполняет другие действия) в плоскости сетки.

Поэтому, чтобы задать правильное направление рисования или других воздействий на ландшафт, меняйте угол под которым смотрит камера. Например, если камера смотрит вертикально вниз, то ваша кисть будет рисовать в горизонтальной плоскости.

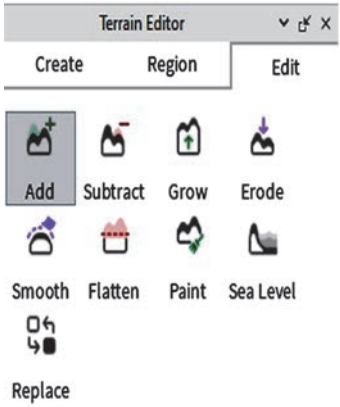


Рис.18.11 Меню Edit Terrain Editor

Второй доступный режим меню **Edit** это режим **Subtract** (Вычитание). Этот режим является аналогом “стерки”. В нем также можно выбрать параметры кисти (за исключением выбора типа материалов — они “стерке” не нужны).

Сходными по результатам рисования с режимами **Add** и **Subtract** являются, соответственно, режимы **Grow** (Растить) и **Erode** (Разъедание/Эрозия). Дополнительно расширяют ваши возможности по коррекции ландшафта режимы **Smooth** (Сглаживание) и **Flaten** (Уплощение).

Третьим разделом **Terrain Editor** является раздел **Region** (Область), который позволяет работать с большими участками ландшафта, выбирая их, копируя, изменяя их размеры, поворачивая или удаляя, практически также, как вы научились работать с **parts** в режиме **MODEL**.

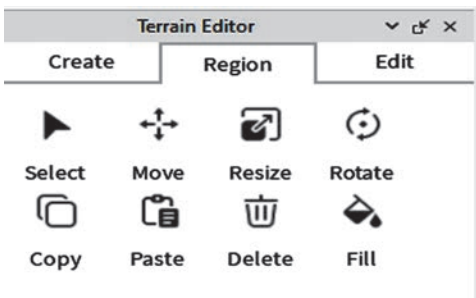


Рис. 18.12 Меню режима Region

Мы познакомили вас с основными инструментами **Terrain Editor**, чтобы их освоить нужна практика. Потренируйтесь. После этого вы самостоятельно сможете разобраться и с остальными инструментами, о которых мы вам не рассказали. Уверен, вы быстро освоите работу с ландшафтом и будете создавать именно то, что будет удивлять других и повышать их интерес к созданным вами играм.

Но, например, если вы создали какой-то летний ландшафт можно ли, “покрасив” его снегом и преобразовав в зимний использовать оба ландшафта в своей игре, меняя их прямо во время игры? Подскажем, как сделать это.



Рис. 18.13 Сообщение об ошибке

Вы можете попробовать подсветить строку **Terrain** в вашей игре, нажать правую кнопку мыши и выбрать **Copy** (Копировать), а затем, выбрав **Workspace**, нажать правую кнопку мыши и выбрать опцию **Paste Into** (Вставить в). Но, Roblox сразу же выдаст вам сообщение об ошибке: *“Невозможно изменить Родителя Terrain. В Workspace уже имеется Terrain”.*

Убедились, что так нельзя. Попробуем по-другому:

— заведите в **ServerScriptService** новый скрипт с именем **Store** и скопируйте в него следующий код:

```
1    workspace.Terrain:CopyRegion(workspace.Terrain.MaxExtents)
2    terrain.Parent = game.Workspace
3    local terrain1 = game.Workspace.TerrainSummer
4    workspace.Terrain:PasteRegion(terrain1, workspace.Terrain.Max
 Extents.Min, true)
```

— поставьте курсор мыши на строку нового скрипта в окне **Explorer**, нажмите правую кнопку мыши и выберите опцию **Disable Scripts** (Отключить Скрипт). Этот скрипт мы использовать не будем, этот скрипт — место временного хранения нашего кода;

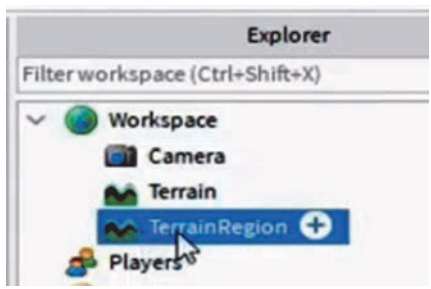


Рис. 18.14 Вставка нового ландшафта

— перейдите в режим **View** и нажмите кнопку вызова окна командной строки (**Command Bar**). Затем, выделите и скопируйте (с помощью клавиш **CTRL+C**) строки 1 и 2 созданного скрипта, вставьте

их (**CTRL+V**) в командную строку и нажмите **Enter**; Вы увидите, что в окне **Explorer** появился новый объект **Terrain Region**. Переименуйте его, например, в **TerrainSummer** (ЛетнийЛандшафт);

— войдите в редактор ландшафта и “покрасив” свой ландшафт снегом, превратите его в зимний. Ну или сделайте любые другие изменения, какие посчитаете нужными;

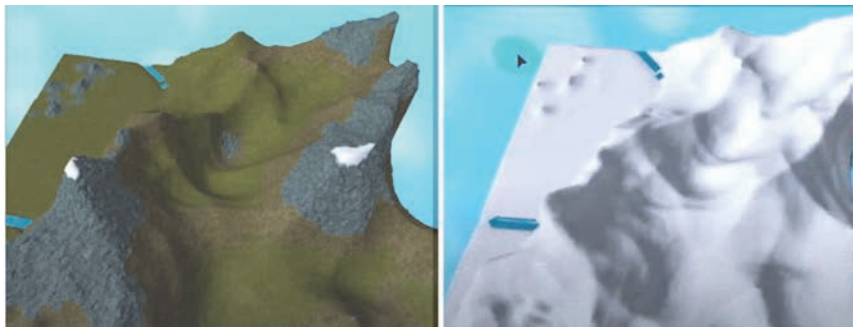


Рис. 18.15 Летний и зимний ландшафты одного и того же игрового поля

— еще раз скопируйте строки 1 и 2 созданного скрипта, вставьте их в командную строку и нажмите **Enter**. В окне Explorer появится еще один **TerrainRegion**. Переименуйте его, например, в **TerrainWinter** (ЗимнийЛандшафт);

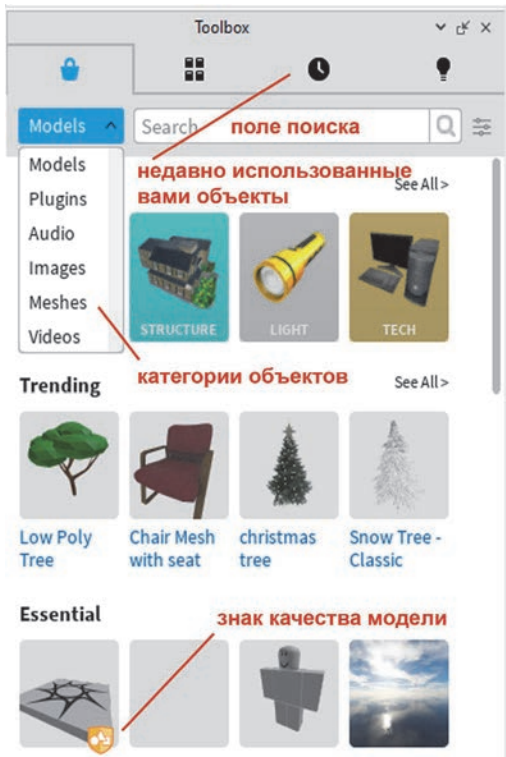
— вернитесь в созданный в **ServerScriptService** скрипт **Store** и проверьте, точно ли совпадают название летнего ландшафта введенное вами в строке 3 кода и название этого ландшафта, введенное в окне **Explorer**, если это не так — устраните неточность. Затем, скопируйте строки 3 и 4 скрипта **Store** в командную строку и нажмите **Enter.** Вы увидите, что на игровом поле ландшафт бывший зимним быстро сменится на летний;

— создайте в **ServerScriptService** еще один скрипт с именем **ChangeTerrain** (СменаЛандшафта) и вставьте в него строки 3 и 4 прежнего скрипта, но, замените в строке 3 название ландшафта на зимний, указанный в окне **Explorer**, а перед этими строками кода вставьте функцию **wait(20)**:

```
1       wait(20)
2       local terrain1 = game.Workspace.TerrainWinter
3       workspace.Terrain:PasteRegion(terrain1, workspace.Terrain.Max
  Extents.Min, true)
```

Запустите игру — через 20 секунд исходно установленный в игре летний ландшафт изменится на зимний. Принцип понятен? Теперь, вы сами можете менять ландшафт так, как и когда этого потребует сценарий вашей игры.

19. Как пользоваться Toolbox



Окно **Toolbox** дает разработчику игр возможность использования того, что создано другими.

Рис. 19.1 Окно Toolbox

Нажав на стрелочку в окошечке под пиктограммой “корзинка” вы откроете меню категорий объектов: моделей, звуков, изображений, моделей и видео. Все это можно найти и использовать.

Выбрав нужную вам категорию впишите (по английски) название того, что конкретно вы ищете в окне поиска. А получив результаты поиска, обращайте внимание в первую очередь на то, что имеет “**знак качества**”. Помеченные таким знаком объекты проверены и одобрены многими пользователями, значит, не создадут вам проблем и будут работать сразу. Правда, объектов со знаком качества

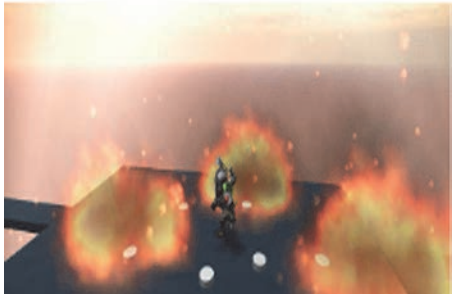
очень немного, поэтому обращайте внимание на **рейтинг объектов** (зеленую пиктограмму в виде кулака с поднятым вверх большим пальцем, рядом с которым стоит процент пользователей, давших положительный отзыв об этом объекте). Если объект понравился — достаточно кликнуть по нему, и он будет помещен в вашу игру.

Если вы ранее уже находили и использовали какие-то объекты, вы сможете быстро найти их кликнув по пиктограмме “часы”.

Казалось бы, все очень удобно, но использование чужих объектов несет в себе риски вирусного заражения вашего компьютера и потери не только робаксов (если они у вас есть), но и всего того, что вы разработали. Кроме того, игры построенные с использованием большого числа заимствованных объектов не пользуются уважением в среде самих разработчиков.

Полезным советом можно считать **обязательную проверку скриптов**, если они входят в состав объекта, который вы хотите заимствовать. Roblox Studio обязательно предупреждает о наличии скриптов в выбранных вами объектах — не отключайте эти предупреждения. Скачав новый объект со скриптами, кликните правой кнопкой мыши, выберите в выпадающем меню **Disable Scripts** (Отключить Скрипты), затем, откройте и просмотрите скрипты. Но, как вы понимаете, для этого нужны знания и опыт. То есть, пока вы не стали более опытны, лучше не используйте объекты **Toolbox** со скриптами.

20. Взрывы: мины и бомбы



Научимся создавать детали (любой формы и цвета), которые будут взрываться в игре.

Рис. 20.1 Программирование взрывов

Создайте в **Workspace**, для начала, одну такую **part** и пусть она называется **ExplosionPart**. Вставь-

те в **ServerScriptService** скрипт управляющий взрывами. В скрипте (см. строка 2) создайте локальную переменную **explosion**. Укажите, что она находится в **Workspace**(см. строку 3) и должна взрываться в том месте, где находятся объект **ExplosionPart**(см. строку 4):

```
1   while true do
2       local explosion = Instance.new("Explosion")
3       explosion.Parent = game.Workspace
4       explosion.Position = game.Workspace.ExplosionPart.Position
5       wait(2)
6   end
```

Запустите программу. Наша мина взрывается через каждые 2 секунды. Но, это только одна мина и то, что происходит выглядит не очень эпично. Дублируйте мину и разместите ее неподалеку от первой, чтобы было удобнее наблюдать за обеими. Запустите программу. — Взрывается только одна мина!

Давайте, изменим наш скрипт:

```
1   local ch_array = game.Workspace:GetChildren()
2   while true do
3       for _, child in ipairs(ch_array) do
4           if child.Name == "ExplosionPart" then
5               if child.BrickColor == BrickColor.Red() then
6                   local explosion = Instance.new("Explosion")
7                   explosion.Position = child.Position
9                   explosion.Parent = game.Workspace
10                  child.BrickColor = BrickColor.White()
11              else
12                  local number = math.random(1,3)
13                  if number == 1 then
14                  child.BrickColor = BrickColor.Red()
15                  end
16              wait(1)
17              end
18          end
19      end
20  end
```

Рассмотрим новый скрипт:
— в строке 1, чтобы взорвались все созданные мины, с помощью функции **GetChildren** (ПолучитьПотомков) находим список всех частей, входящих в **Workspace** и присваиваем его значение переменной **ch_array**;
— в строке 2 создадим бесконечный цикл **while true**;
— в строке 3 используем цикл **for_in ipairs** для просмотра всех потомков (**child**) из списка **ch_array;**
— в цикле **if - then** строки 4 проверяем, является ли имя **child** из списка именем **ExplosionRart**, и, если это так:
— в строке 5 проверяем цвет этого **child**, и если цвет красный, то:
— в строках 6…9 организуем взрыв этого **child;**
— в строке 10 цвет взорвавшегося **child** меняем на белый. Вы удивитесь, но без специальной команды, взорвавшийся **child** никуда не девается;
— если же цвет **child** отличен от красного, то в строке 11 включаем оператор **else**, который:
— в строке 12 присваивает переменной **number** случайное значение (в диапазоне от “1” до “3”);
— в строках 13…15, если величина переменной **number** окажется равной “1” перекрашиваем **child** в красный цвет;

Мы специально сделали, чтобы **ExplosionParts** случайным образом меняли свой цвет на красный и лишь затем взрывались, предупреждая игрока и давая ему шанс увернуться от взрыва!

Но, согласитесь, в скрипте есть некоторые недостатки. Например, вы видите, что после взрыва наши мины вновь появляются целехонькими. Для какой-то игры это вполне может и подойти, например, чтобы создать постоянно действующую “полосу препятствий”. Но, в реальности такого не бывает! Например, сбросили вы бомбу с самолета или вертолета, а она будет постоянно “оживать” и взрываться… Странно? Конечно! Но, эту проблему можно решить очень просто, нужно лишь заменить строку 10 на **child.Destroy()**. Тогда сразу после взрыва взорвавшийся **child** будет уничтожен.

Для полноты информации о взрывах нужно добавить следующее:
— взрыв разрывает связи между составными частями объектов, убивая гуманоидов. Взрыв сразу же уничтожает все остатки взорвавшейся **part**, не нужно ничего очищать от ее осколков;

— взрыв применяет к **part** силу в пределах некоторого радиуса **Blast Radius**:
То есть, если написать в скрипте `explosion.BlastRadius = 60`, то взрыв будет действовать в пределах 60-ти шипов;

— свойство **DestroyJointRadiusPercent** используется для установки той части **BlastRadius** (от 0 до 1), в пределах которой будут разрушены все суставы. Ко всему, что находится за пределами этого диапазона, будет применена только сила взрыва. Например, если для свойства **BlastRadius** установлено значение “60”, а для свойства **DestroyJoint RadiusPercent** установлено значение “0,5”, то все суставы между составными частями гуманоида в радиусе 30 шипов будут сломаны. Но, любые соединения на расстоянии от 30 до 60 шипов разрушены не будут, хотя сила взрыва все равно будет приложена ко всем игровым объектам. Это свойство позволяет разработчикам сделать их взрывы даже совсем «несмертельными», устанавливая **DestroyJointRadiusPercent** равным “0”;
— свойство **BlastPressure** (СилаВзрыва) определяет величину силы, приложенной ко всем **parts**, попавшим в **BlastRadius** независимо от их расстояния от точки взрыва. Все незакрепленные **parts** будут одинаково ускоряться, если они находятся в **BlastRadius**. При этом, сила взрыва не определяет степень разрушения суставов, также, как она не определяет количество урона, наносимого **Terrain**.

— при помощи свойства **ExplosionType** взрывы можно настроить для повреждения **Terrain**. Это свойство определяет, как взрыв будет взаимодействовать с ландшафтом. Свойству **Enum.ExplosionType**, может быть присвоено два значения:
1/ **NoCraters** — взрывы не повреждают ландшафт:

```
explosion.ExplosionType = Enum.ExplosionType.NoCraters
```

2/ **Craters** — взрывы будут создавать кратеры на местности:

```
explosion.ExplosionType = Enum.ExplosionType.Craters
```

радиус кратера при этом, будет примерно равен **BlastRadius**. Кратеры создаются во всех материалах **Terrain**, кроме воды. Размер кратера не зависит от материала, хотя некоторые материалы создают более грубые края, чем другие;

— свойство **Visible** определяет, будет ли отображаться визуальный эффект взрыва. Если для свойства **Visible** установлено значение **false**, взрыв по-прежнему будет влиять на **parts** внутри **BlastRadius**, единственная разница в том, что он не будет виден.

Программируем бомбардировщик

А как быть, если вы хотите, загрузить ваш самолет бомбами и сделать так, чтобы он летал и бомбил врагов? Если запустить приведенный выше скрипт первый же взрыв произойдет в самолете. Последствия понятны.

Давайте построим простенький самолет и превратим его в настоящий бомбардировщик. Вы должны понимать, что все детали самолета должны представлять собой единое целое, например, “твердую модель”.

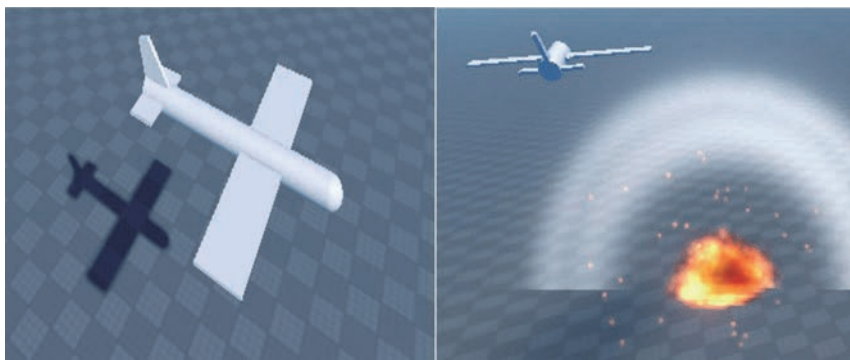


Рис 20.2 Программируем бомбардировщик

Вставьте в модель самолетика **Bomber** два скрипта: **Bombs** и **Motor**. В скрипт **Motor** скопируйте вот такой простой “двигатель”, который заставит самолет двигаться по прямой вдоль оси X:

```
1    local airplane = script.Parent
2    while true do
3        airplane.CFrame = airplane.CFrame +Vector3.new(-1,0,0)
4        wait(.2)
5    end
```

А в скрипт **Bombs** вставьте вот такой код:

```
1    local airplane = script.Parent
```

```
2       while true do
3               local bomb = Instance.new("Part")
4               bomb.Shape ="Cylinder"
5               bomb.Parent = game.Workspace
6               bomb.Position = workspace.Bomber.Position + Vector3.new(0,-2,0)
7               bomb.Size = Vector3.new(2,0.5,1)
8               bomb.Anchored = false
9               wait(2)
10              if bomb.CFrame.Position.Y < 5 then
11                      local explosion = Instance.new("Explosion")
12                      explosion.Position = bomb.Position
13                      explosion.BlastRadius = 5
14                      explosion.Parent = game.Workspace
15                      bomb:Destroy()
16              end
17      end
```

Поясним этот скрипт:
— мы видим, что в строках 2…17 описан бесконечный цикл;
— в строке 3 мы создаем объект типа **Part** и присваиваем его значение переменной **bomb**;
— в строке 4 определяем форму строительного блока — цилиндр;
— в строке 5 указываем Родителя **bomb** (Workspace);
— в строке 6 задаем координаты, в которых будет всегда появляться **bomb** (внутри бомбардировщика). Если бы мы указали, что координаты должны совпадать с центром **union** самолета, то можно было бы заметить, что бомбы не вываливаются из “брюха” самолета, а падают откуда-то с верхней части фюзеляжа самолета. Дело в том, что из-за наличия у самолета высокого киля геометрический центр самолета находится в верхней части фюзеляжа. Поэтому, мы ввели поправку: **Vector3.new(0,-2,0)**, которая опускает на 2 шипа вниз то место, откуда будут появляться бомбы. Для вашей модели вы должны будете сами определить величину такой поправки;
— строка 7 задает размеры бомб;
— строка 8 отключает крепление бомб к самолету и бомбы падают из него одна за другой через интервал времени заданный в строке 9;
— строка 10 открывает цикл **if - then**, в котором, если позиция бомб по оси Y, менее 5 шипов (то есть, если бомба упала), создается объект типа **Explosion** и его значение присваивается переменной **explosion**;
— строка 12 определяет, что взрыв происходит в позиции бомбы;

— строка 13 задает радиус поражения бомбы;
— строка 14 определяет Родителя объекта **Explosion**;
— в строке 15 происходит уничтожение бомб (чтобы после взрыва они не валялись целехонькими по всему игровому полю).

Теперь, как всегда, дело за вами — поэкспериментируйте со скриптами взрывов, разберитесь детально, как работают все их свойства.

21. Наклейки с текстурами и картинками

Объекты типа **Textures** (Текстуры) и **Decal** (Наклейка) — это изображения, которые можно наносить на поверхности разных **parts**. Этих картинок очень много в **Toolbox** и брать их можно безбоязненно.



Рис. 21.1 Текстуры и наклейки

Чтобы приклеить **Textures** или **Decal** нужно подсветить part в меню Explorer, нажать кнопку ⊕ и выбрать в выпадающем меню **Textures** или **Decal**.

Текстура кратно повторяется всей 2D-поверхности **part**. Размер каждого элемента текстуры можно изменить. В отличие от текстур, **Decal** просто растягивается по всей поверхности, то есть ее вид зависит от размера поверхности. **Textures** можно смещать на поверхности **part** с помощью свойств **OffsetStudsU** и **OffsetStudsV**, что дает возможность лучше управлять размещением текстур. Цвет и прозрачность обоих типов изображений также можно менять, меняя их свойства в окне **Properties**.

Сторона **part**, на которую наносится то или иное изображение, зависит от свойства **Face** (можно указать **Front**, **Back**, **Left**, **Right, Top, Bottom**). Размер **Decal** зависит от размера **part**, то есть размер и соотношение сторон картинки можно изменить, изменив соответствующие значения **part.Size** Родителя.

Приведем пример программного управления изображением типа **Decal.** Разместите этот скрипт в **ServerScriptService**:

```
local part = Instance.new("Part")
part.Size = Vector3.new(5, 5, 5)
part.Position = Vector3.new(-10, 2.5, 0)
part.Anchored = true
part.TopSurface = Enum.SurfaceType.Smooth
part.BrickColor = BrickColor.new("Toothpaste")
part.Parent = workspace
local decal = Instance.new("Decal")
decal.Face = Enum.NormalId.Left
decal.Parent = part
local happyFace = "http://www.roblox.com/asset/?id=26424652"
local sadFace = "http://www.roblox.com/asset/?id=147144198"
decal.Texture = happyFace
local sound = Instance.new("Sound")
sound.SoundId = "rbxasset://sounds/uuhhh.mp3"
sound.Parent = part
local debounce = false
part.Touched:Connect(function(hit)
        if not cooldown then
                if hit and hit.Parent then
                        debounce = true
                        sound:Play()
                        decal.Texture = sadFace
                        wait(1)
                        decal.Texture = happyFace
                        wait(0.2)
                        debounce = false
                end
        end
end)
```

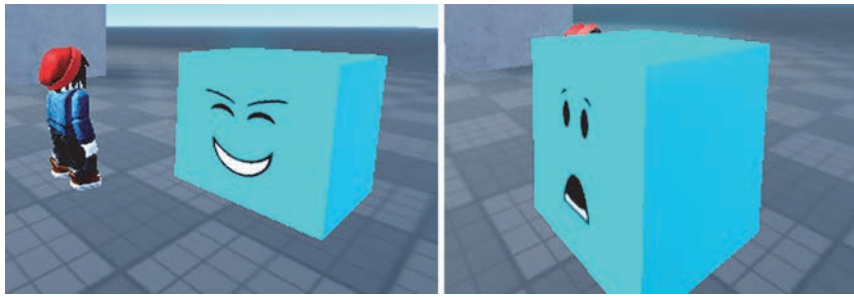


Рис. 21.2 Работа программы управления изображением типа Decal

Поясним код:
— в строках 1…7 создаем **part** типа “кирпич” и задаем его размеры, его местоположение, “якорим”, делаем его поверхность гладкой, определяем его цвет и определяем его Родителя;

— в строках 8…13 создаем **Decal**, указываем, что его нужно приклеить к левой стороне **part**, определяем его Родителя, указываем какие изображения мы будем использовать в **Decal**, а также указываем исходное изображение на **part**;

— в строках 14…16 создаем переменную **sound**, указываем какой звук ей соответствует, и что является источником звука. Громкость звука в Roblox определяется расстоянием между слушателем и положением источника звука. Звук считается «глобальным», если он не связан с **part** или с вложением в **part**. В этом случае звук будет воспроизводиться с одинаковой громкостью во всем игровом пространстве. В нашем случае источник звука — это **part** и звук будет слышен только вблизи этой **part**;

— в строке 17 создаем переменную **debounce**, которую будем использовать для устранения явления “дребезга” при касании **part** аватаром игрока;

— в строке 18 подключаем к индикатору события **Touched** нашей **part** функцию без имени, определяющую действия, которые будут производиться при касании **part**:

— строка 19 проверяет значение переменной **debounce**, и если она равна **false,** то в строке 20 проверяется было ли касание **part**, и если

было, то в строке 21 заменой значения **debounce** на **true** первым делом отключается возможность повторного запуска функции;

— в строках 22, 23 включается звук “Ой!” и картинка **Decal** меняется на “расстроенное лицо”;

— через 1 секунду картинка **Decal** меняется на “счастливое лицо” и вновь меняется значение переменной **debounce** на **false**, разрешая тем самым повторный запуск функции.

22. Излучатели частиц

Излучатели частиц **ParticleEmitter** — это объекты, испускающие 2D-частицы, вид и срок существования которых соответствуют заданным им свойствам. Можно настроить свойства излучателя частиц для создания специальных эффектов, таких как огонь, дым и искры разных форм и размеров.

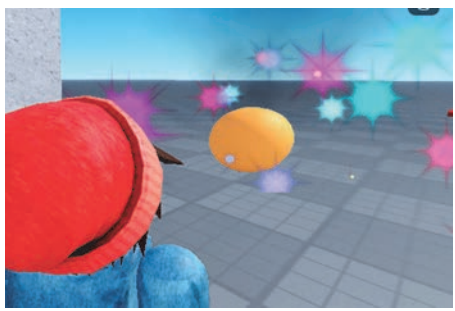


Рис. 22.1 Пример излучения частиц

Когда вы создаете **part**, как источник частиц, частицы зависят от размера **part**. Например, **part** с малыми размерами имеет меньшие по размеру частицы, а часть большого размера создает частицы большего размера. Частицы излучаются из всей “ограничивающей рамки” **part**. Например, если прикрепить излучатель частиц к **part** типа **“**клин”, частицы будут испускаться из всей ограничивающей его рамки, а не только из тех частей, где существует сам клин.

Чтобы создать излучатель частиц, создайте **part**. В окне **Explorer** наведите курсор на **part** и нажмите кнопку ⊕. Появится контекстное меню. Выберите в меню **ParticleEmitter**. Излучатель частиц немедленно начнет испускать частицы в пределах области детали (без запуска программы кнопкой **Play**). Откройте окно **Properties** частиц и настройте свойства этого излучателя частиц.

Свойство **EmissionDirection** (НаправлениеИзлучения) определяет поверхность детали, которая будет излучать частицы. Когда вы меняете это свойство, вы также меняете направление испускания частиц. По умолчанию **EmissionDirection** — это верхняя (**Top**) часть детали, но можно изменить направление излучения на любую другую сторону (**Left, Right, Front, Back, Bottom**). Если **part** во время игры вращается, ориентация ее сторон также меняется, изменяя направление испускания частиц. Вы можете настроить нужное направление излучения частиц в режиме **MODEL** — выберите **Rotate** и разверните **part** так, как вам нужно.

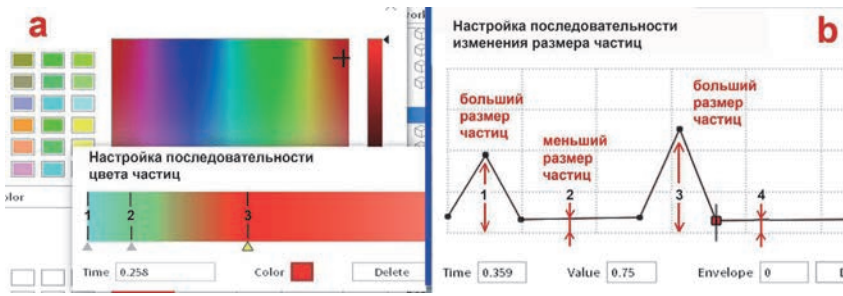


Рис. 22.2 Настройка последовательности цветов частиц (a) и настройка последовательности изменения размеров частиц (b)

По умолчанию частицы имеют белый цвет, но вы можете не только изменить их цвет на любой, но и задать последовательность, в соответствии с которой будет меняться цвет частиц. Выберите в окне **Properties** свойство **Color** и установите нужный вам цвет частиц. Если же хотите, чтобы частицы постепенно меняли свой цвет, нажмите в свойстве **Color** кнопку "...". Появится всплывающее окно управления последовательностью цветов. По умолчанию цветовая последовательность полностью белая в течении всего времени пока частицы существуют. Но, передвигая движки-треугольнички вдоль оси времени и выбирая те или иные цвета, вы можете задавать моменты плавного перехода одного цвета частиц в другой цвет в течение всего времени существования частиц. Для выхода из окна настройки последовательности цветов нажмите "крестик" в его правом верхнем углу.

Свойство **Size** также устанавливает размеры частиц, либо постоянными, либо устанавливает последовательность, в соответствии с которой размеры частиц будут меняться в течении времени их существо-

вания. Чтобы задать изменения размеров частиц, выберите свойство **Size** и нажмите кнопку "…". Во всплывающем окне отобразится график величины частиц во время существования частиц. Исходно график представляет собой прямую линию расположенную вдоль оси времени. Эта линия соответствует одинаковому размеру частиц на протяжении всей их жизни. Прикасаясь курсором мыши к этой прямой, можно поднимать и опускать участки прямой. В моменты времени, в которых линия будет приподнята, размеры частиц будут увеличиваться. Там же, где линия будет опускаться вниз к оси времени, размеры частиц будут уменьшаться. Для выхода из этого окна также нажмите "крестик" в его правом верхнем углу.

У **ParticleEmitter** есть и другие свойства, откройте окно **Properties** и исследуйте их самостоятельно, проводя эксперименты. Естественно, вы можете менять все свойства **ParticleEmitter** не в окне **Properties**, а в своей программе. В том числе, управлять излучением и отключением излучения частиц вы можете задействуя свойство **ParticleEmitter.Enabled = true / false**. Ниже приведен скрипт управления **ParticleEmitter** объекта **Torch**:

```
while true do
	workspace.Torch.ParticleEmitter.Enabled =true
	wait(5)
	workspace.Torch.ParticleEmitter.Enabled =false
	wait(5)
end
```

Конечно же, это не вся информация об излучателях частиц. Вы обязательно должны не только детально исследовать, как действуют описанные выше свойства **ParticleEmitter**, но найти и самостоятельно разобраться с тем, как действуют другие свойства, например, **ParticleEmitter.Lifetime**, а также функции **ParticleEmitter:Emit()** и **ParticleEmitter:Clear().**

23. Огонь

Огонь — это один из нескольких готовых объектов, испускающих частицы. Огонь излучает частицы, когда он связан с конкретной **part** и включен (**Enable**).



Рис. 23.1 Пример огня в игре

Частицы огня исходят из центра **part**, к которой они относятся. Частицы огня обычно испускаются в направлении вверх (+Y), однако, можно использовать и отрицательное (-Y) значение свойства **Fire.Heat**. Когда параметр **Enable** отключен, частицы огня продолжат отображаться до тех пор, пока не истечет установленное в их параметрах время действия. Однако, если **part**, являющаяся для огня Родителем становится равной **nil**, все частицы огня немедленно исчезают.

Огненный объект состоит из двух излучателей. Частицы, испускаемые вторичным излучателем, имеют значительно большее время жизни и поднимаются выше, чем частицы, испускаемые первичным излучателем.
В отличие от настоящего пламени, объект **Fire** не распространяется сам по себе, эффект распространения огня можно создать только при помощи кода.

Создайте на игровом поле какую-нибудь **part**, например шар, назовите его **Torch** (Факел) и вставьте в него скрипт и скопируйте в него приведенный ниже код:

```
for _, child in pairs(workspace:GetChildren()) do
        if child.Name == "Torch" and child:IsA("BasePart") then
                local fire = Instance.new("Fire")
                fire.Parent = child
                fire.Heat = 10
                fire.Color = Color3.new(1, 0.5, 0)
                fire.SecondaryColor = Color3.new(1, 0.7, 0.5)
                fire.Size = math.max(child.Size.X*3, child.Size.Z*3)
                wait(15)
                fire.Size = math.max(child.Size.X*0, child.Size.Z*0)
                fire.Heat = -10
```

```
12          end
13    end
```

Поясним скрипт:
— в строке 1 мы создали цикл просмотра всех объектов игрового мира;
— в строке 2 проверяем имена просматриваемых объектов, и, если имя объекта будет **“Torch”** (Факел), то создается переменная **fire**, соответствующая огню;
— в строках 4…8 задаются параметры огня, а именно:
— строка 4 определяет Родителя огня;
— строка 5 задает скорость истечения частиц огня;
— строка 6 определяет, что цвет основного излучателя огня;
— строка 7 задает цвет вторичных частиц огня;
— строка 8 задает размеры огня;
— строка 9 задает время горения;
— строка 10 “выключает” огонь сокращая его размеры до “0”;
— строка 11 прячет оставшиеся частицы огня, перенаправляя их вниз;
— строка 12 завершает цикл **if - then**;
— строка 13 завершает цикл **for_ in pairs**.

Поэкспериментируйте, меняя один за другим все параметры кода огня.В том числе, меняя время горения или временно отключая ненужные строки кода, напечатав перед ними два тире (”- -”), как перед комментариями.

24. Программируем дым



Рис. 24.1 Дым

Дым является еще одним из объектов, излучающих частицы. Как и другие излучатели частиц объекты **Smoke** излучают частицы, если они связаны с базовой частью (**part**) или с вложением в такую базо-

вую часть. По сравнению с **ParticleEmitter**, в **Smoke** отсутствуют многие свойства и функции, такие как **ParticleEmitter.Lifetime** или **ParticleEmitter:Emit()**. **Smoke** — это способ быстро создать спецэффект.

Когда **Smoke.Enabled** отключен, частицы, испускаемые этим объектом, будут продолжать отображаться до тех пор, пока не истечет их время жизни. Но, когда родительский объект **Smoke** будет установлен на **nil** или применена команда **Instance:Destroy()**, все частицы дыма мгновенно исчезнут. Частицы дыма испускаются только из центра **parts**, к которому они относятся. Но, если привязать **Smoke** к вложению в **parts**, то можно настраивать место излучения частиц дыма. Ниже приведен пример кода, добавляющего дым ко всем горящим объектам игры. Этот скрипт можно просто добавить к коду, имеющемуся в скрипте, которым мы программировали огонь факела **Torch**:

```
1    local function Fire(object)
2        if object:IsA("Fire") and not object.Parent:FindFirstChildOfClass
  ("Smoke") then
3            local smoke = Instance.new("Smoke")
4            smoke.Parent = object.Parent
5            smoke.Opacity = 0.15
6            smoke.RiseVelocity = 4
7            smoke.Size = object.Size / 4
8            smoke.Color = Color3.new(0, 0, 0)
9        end
10       for _, child in pairs(object:GetChildren()) do
11           Fire(child)
12       end
13   end
14   Fire(workspace)
```

Поясним скрипт:
— строки 1…13 описывают функцию Fire;
— строка 14 запускает эту функцию с параметром **workspace**, что означает, что ей передают все объекты игрового мира;

— в строке 2 в цикле **if - then** функция проверяет, является ли переданный ей в процессе поиска **FindFirstChildOfClass** объект объектом типа **Fire** и, одновременно, не является объектом типа **Smoke**, и если

оба этих условия выполняются, то в строке 3 создается объект типа **Smoke**, а его значение присваивается переменной **smoke**;

— в строке 4 цикла указывается, что объекту игрового мира присваивается статус Родителя созданного объекта **Smoke** (то есть, он становится источником дыма);
— в строках 5…8 цикла определяются цвет, плотность, размеры и скорость излучения частиц дыма;

— в любом случае: если оператор **if-then** уже выполнен или же он не выполнялся из-за того, что не выполнялись указанные в нем условия, строки 10…12 запускают цикл **for _, child in pairs**, который "перебирает" один за другим все объекты игрового мира и передает каждый из них функции **Fire.**

Запустите программу и посмотрите, что произойдет. Как и раньше, факел будет гореть несколько секунд, а затем его пламя перестает быть видным. Но, мы же помним, как мы достигли этого? В действительности горение факела продолжается, мы лишь спрятали пламя уменьшив его размер и развернув направление пламени вниз. Поэтому, как только пламя "погаснет" вы увидите клубы дыма. Теперь дело за вами — исследуйте свойства частиц дыма, меняя величину параметров в программе. Ну и, конечно же, попытайтесь модифицировать программу, чтобы создать какие-то новые игровые эффекты с использованием дыма.

25. Лучи смерти

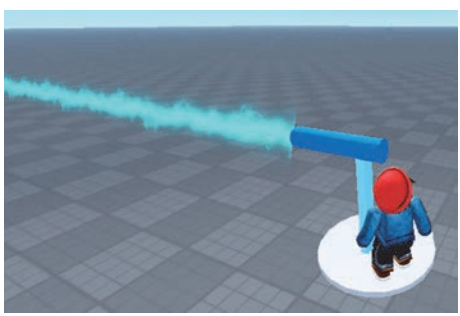


Рис. 25.1 "Луч смерти"

Лучи — это эффект, который можно использовать для создания лазеров, силовых полей и даже водопадов. Этот объект рисует динамически меняющуюся текстуру между двумя точками. Свойства этой текстуры такие как скорость, ширина и кривизна, можно настраивать.

Давайте создадим лазерную пушку, но с лучом более интересного вида. Внешний вид вашей пушки (назовем ее **LaserGun**) вы можете придумать сами. Следующим шагом, нужно добавить **Attachments** — места, где один объект может соединиться с другим. В нашем случае **Attachments** будут использоваться для начальной и конечной точек "луча смерти". В окне **Explorer** наведите курсор мыши на пушку, нажмите кнопку ⊕ и добавьте в **LaserGun** два объекта **Attachment** и переименуйте их в **Start_Attachment** и **End_Attachment**.

Оба объекта **Attechment** должны быть расположены на оси, по которой вы планируете формировать луч. Выберите в окне **Explorer** объект **Start_Attachment** и с помощью инструмента **Move** меню **MODEL** поместите его на конец ствола лазерной пушки. Аналогично, переместите **End_Attachment** туда, где действия "луча смерти" должно прекратиться.

Еще раз кликните по кнопке ⊕, добавьте из выпадающего меню в состав **LaserGun** объект **Beam** (Луч). Откройте его окно **Properties** и найдите в нем свойство **Attachment0**. Щелкните пустое поле справа от этого свойства - появится пиктограмма точки со стрелками, которая будет следовать за курсором мыши. В окне **Explorer** кликните по **Start_Attachment**. Это присоединит начало луча к концу ствола пушки. Аналогично, присоедините **Attachment1** луча к **End_Attachment**.

Теперь нужно настроить другие свойства "луча смерти" в окне **Properties**:

— чтобы луч был виден в любом положении, поставьте "галочку" в свойстве **FaceCamera**, это настроит камеру;

— внешний вид луча формируется при помощи 2D-изображений, которые можно настраивать, определяя цвет, размер, форму и динамику изменений формы луча. Скопируйте один из идентификаторов изображений луча, приведенных ниже:

— **rbxassetid://6060542021**,
— **rbxassetid://6060542158**,
— **rbxassetid://6060542252**

и вставьте его в свойство **Texture**;
— измените свойство **LightEmission** на "0,5";

— измените свойство **Transparency** на “0”;
— сделайте луч шире, установив свойства **Width0** и **Width1** равными “3”;
— анимируйте луч установив свойству **TextureSpeed** значение “3”;
— придайте “лучу смерти” более опасный вид, изменив его свойство **Color**;
— в свойстве **TextureMode** по умолчанию выбран режим **Stretch** (Растянуть), Но, вы можете выбрать вариант **Wrap** (Вращение) или **Static** (Неподвижный);

Скрипт пушки будет проверять, касается ли “луч смерти” игрока. Проблема в том, что **Beams** (Лучи), по умолчанию, не имеют индикаторов обнаружения столкновений. Поэтому, в **LaserGun** нужно создать новую **part** с именем **hidden** (тайная), которая должна иметь длину, равную длине будущего луча, и “заякорить” ее. Осталось добавить в **LaserGun** новый скрипт и скопировать в него приведенный ниже код:

```
1   local LaserGun = script.Parent
2   local hidden = LaserGun:FindFirstChild("hidden")
3   hidden.Transparency = 1
4   local function onTouch(otherPart)
5       local character = otherPart.Parent
6       local humanoid = character:FindFirstChildWhichIsA("Humanoid")
7       if humanoid then
8           humanoid.Health = 0
9       end
10  end
11  hidden.Touched:Connect(onTouch)
```

В строке 2 программа ищет объект с именем “**hidden**” и присваивает его значение переменной **hidden**. Изменяя прозрачность объекта **hidden,** строка 3 делает его невидимым. А функция **onTouch,** убивающая гуманоида, если он прикоснется к объекту **hidden,** вам уже известна.
Запустите программу и проверьте ее работу, направив свой аватар под лазерный луч. Если он работает неправильно, убедитесь, что названия всех объектов в скрипте указаны правильно.

Поэкспериментируйте меняя свойства луча, такие как **CurveSize0** и **CurveSize1**, поищите и импортируйте из **Toolbox** другие текстуры луча.

26. Источники света



Рис. 26.1 Светящаяся поверхность

Светящиеся поверхности **SurfaceLight** — это источники света, которые излучают свет указанного цвета и яркости на заданную дальность.

Чтобы **SurfaceLight** обеспечивал освещение, он должен быть **part** или входящим в нее объектом. Его свет будет исходить только от одной грани детали. Для примера, создайте в меню MODEL столбик с именем **column**, как на рисунке и вставьте в него скрипт, приведенный ниже:

```
1   local plate = Instance.new("Part")
2   plate.Parent = workspace
3   plate.Position = game.workspace.column.Position
4   plate.Anchored = true
5   plate.Size = Vector3.new(4,4,1)
6   local light = Instance.new("SurfaceLight")
7   light.Parent = plate
8   light.Color = Color3.fromRGB(255, 255, 255)
9   light.Brightness = 15
10  light.Range = 16
```

Мы видим, что скрипт создает **plate** (пластину), чей центр совпадает с центром столбика **column**. Строка 5 определяет размеры пластины. Строка 6 создает переменную **light** типа **SurfaceLight**. Строка 7 указывает, что Родителем светящейся поверхности является **plate**. Запустите игру — будет светиться одна грань пластины **plate** и аналогичная грань столбика **column**.



Рис. 26.2. Точечный источник света

В Roblox Studio можно создавать и точечные источники света **PointLight**, излучающие свет из одной точки во все стороны. Чтобы **PointLight** обеспечивал освещение, он должен быть **part** или дочерним элементом **part** (вложением). Если **PointLight** связан с **part**, то свет будет исходить из **part.Position**. Если **PointLight** связан с вложением в **part**, то свет будет исходить из этого вложения.

Пример кода:

```
1   local part = Instance.new("Part")
2   part.Anchored = true
3   part.Position = game.workspace.column.Position + Vector3.new(0,3,0)
4   part.Parent = workspace
5   part.Size = Vector3.new(1,1,1)
6   local light = Instance.new("PointLight")
7   light.Color = Color3.new(1, 1, 1)
8   light.Brightness = 10
9   light.Range = 16
10  light.Parent = part
```

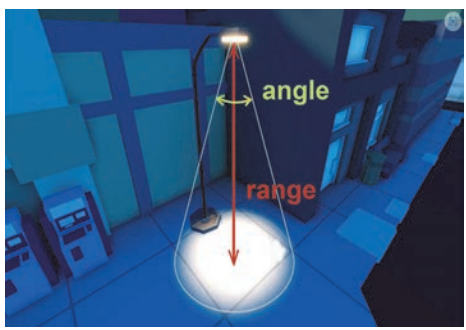


Рис. 26.3 Пример прожектора

Прожекторы **Spotlighs** излучают свет определенного цвета и яркости лучом в форме конуса. Этот тип источников света идеально подходит для фонариков и фар.

Spotlight также должен быть **part** или ее прямым дочерним элементом и имеет следующие свойства:

— свойство **Range** определяет дальность действия светового луча;

— свойство **Angle** определяет угол конуса излучаемого света;

— свойство **SpotLight.Face** определяет поверхность детали, из которой исходит свет: -Z — спереди, +X — справа, +Y — сверху и т. д.

Нужно напомнить, что все три типа источников света доступны для добавления к другим объектам — в выпадающем меню, которое появляется если подсветить объект в окне **Explorer** и нажать кнопку ⊕. Используя полученные в этой главе знания, попробуйте, например самостоятельно, создать инструмент-фонарик и положить его в рюкзак игрока. Это полезно для исследования пещер и подземелий.

27. Программируем “двойной прыжок”



Рис. 27.1 Прыжки

У гуманоидов есть свойство **JumpHeight**, которое определяет, какую высоту он может достичь при прыжке вверх. Начальное значение этого свойства по умолчанию равно 7,2 шипа, хотя, в принципе, высота может быть существенно больше. Это свойство можно использовать для регулировки высоты, на которую может прыгать гуманоид, что позволяет разработчикам игр запрещать прыжки или настраивать высоту прыжка в зависимости от сценария их игры, например, увеличивать высоту прыжка на Луне.

Давайте создадим и запрограммируем **part**, касание которой будет давать игроку суперсилу для “двойного прыжка”. Добавьте на игровое поле **part** и вставьте в него приведенный ниже скрипт:

```
1    local part = script.Parent
2    function d_JUMP(player)
3        local player = player.Parent
4        player = player:FindFirstChildWhichIsA("Humanoid")
5        if player then
6            part:Destroy()
7            local store = player.JumpHeight
8            print("store =", tostring(store))
```

```
9                   player.JumpHeight = 30
10                  print("Height =", tostring(player.JumpHeight))
11                  wait(10)
12                  player.JumpHeight = 7.2
13                  print("Height =", tostring(player.JumpHeight))
14              end
15      end
16      part.Touched:Connect(d_JUMP)
```

Поясним скрипт:

— строки 2…15 описывают функцию **d_JUMP(player)**, дающую гуманоиду возможность совершать “двойные прыжки” в течении установленного времени;

— строка 16 подключает индикатор касания **part** к созданной функции. То есть, при касании **part** функции будет передаваться объект, который ее коснулся (названный нами “**player**”);

— в строке 3 мы объявляем переменную **player** и присваиваем ей значение Родительского объекта, полученного функцией (то есть, объекта **character**);

— в строке 4 с помощью функции **FindFirstChildWhichIsA** ищем в полученном объекте **character** гуманоида;

— в цикле **if - then** (строки 5…14) если гуманоид найден:

- — в строке 6 уничтожаем **part**, которая увеличила высоту прыжка;
- — в строке 7 запоминаем в переменной **store** исходное значение высоты прыжка;
- — в строке 8 распечатываем запомненное в **store** значение (в окне **Output** — для контроля работы программы);
- — в строке 9 присваиваем свойству **JumpHeight** гуманоида новое значение высоты (в 4 раза большее, чем исходное);
- — в строке 10 печатаем присвоенное значение высоты в окне **Output**;
- — в строке 11 задаем время, в течении которого гуманоид будет обладать новой возможностью;
- — в строках 12,13 возвращаем гуманоиду исходные прыжковые возможности и распечатываем их (для контроля);
- — строки 14, 15 завершают цикл **if-then** и функцию **d_JUMP**.

28. Программируем сиденья

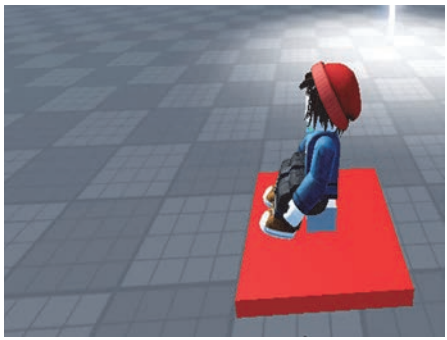


Рис. 28.1 Сиденье

Seat — это **parts**, на которых персонаж игрока может сидеть. Когда персонаж касается объекта **Seat**, он присоединяется к этой **part** “сваркой”, а встроенный скрипт гуманоида показывает анимацию того, как он садится. Персонажа можно усадить на сиденье и с помощью функции **Seat:Sit()**.

Когда аватар, содержащий гуманоида и базовую часть под названием **HumanoidRootPart**, касается сиденья, между сиденьем и **HumanoidRootPart** создается “сварка”. Аватар приваривается к сиденью “сварным швом”, который называется **SeatWeld**. Когда гуманоид сидит, ему устанавливается свойство **Seat.Occupant** (Сидит). Сиденью тоже устанавливается свойство **Humanoid.SeatPart** (Занято Гуманоидом).

У персонажа есть два способа встать со стула. Когда игрок прыгает, он снимается с места. Однако это можно сделать в программе, разрушив сварной шов, например:

```
seat: FindFirstChild ("SeatWeld"):Destroy()
```

Кстати, помните про устранение дребезга? У сидений тоже есть время восстановления (3 секунды). Это означает, что после того, как персонаж встал с места, он не может сесть на то же место в течение 3 секунд.

Сиденья нужны для создания стульев или скамеек, а также для обеспечения возможности персонажам сидеть в транспортных средствах, не сваливаясь из них. Рассмотрим пример такого кода. Создайте простенькую платформу размером (5 x 0,5 x 5 шипов), не забудьте “заякорить” ее, назовите ее **Car** (“автомобильчик”) и вставьте в нее скрипт, приведенный ниже:

```
1   local Players = game:GetService("Players")
2   local Car = script.Parent
3   local seat = Instance.new("Seat")
4   seat.Parent = workspace
5   seat.Size = Vector3.new(1, 1, 1)
6   seat.Position = workspace.Car.Position + Vector3.new(0,0.5,0)
7   local weld = Instance.new("WeldConstraint")
8   weld.Parent = workspace
9   weld.Part0 = seat
10  weld.Part1 = workspace.Car
11  seat_occup = false
12  local function seat_CHANGE()
13          local humanoid = seat.Occupant
14          local i=0
15          seat_occup = true
16          if humanoid then
17                  local character = humanoid.Parent
18                  local player = Players:GetPlayerFromCharacter(character)
19                  if player then
20                          print(player.Name .. " has sat down")
21                          while i<10 do
22                                  i = i +1
23                                  print (tostring(i))
24                                  if  i == 10 and seat_occup ==true  then
25                                          seat: FindFirstChild ("SeatWeld"):Destroy()
26                                          seat_occup = false
27                                  end
28                                  wait(1)
29                          end
30                          return
31                  end
32          end
33  end
34  seat:GetPropertyChangedSignal("Occupant"): Connect(seat_CHANGE)
35  while true do
36          Car.CFrame = Car.CFrame + Vector3.new(-1,0,0)
37  wait(1)
38  end
```

Поясним скрипт:
— в строке 1 вызываем сервис **Players** — чтобы иметь возможность работать с информацией об игроках, ведущих игру;
— в строке 2 создаем переменную **Car** и указываем ее Родителя;

— в строках 3…6 создаем сиденье **seat**, указываем его Родителя, задаем его размеры и указываем, что исходно оно расположено чуть-чуть (на 0.5 шипа) выше созданной нами **part** с именем **Car**;

— в строках 7…10 создаем переменную **weld** (сварной шов), указываем его Родителя, и то, что одним концом он прикреплен к сиденью **seat**, а другим концом — к “автомобильчику” **Car**. Это нужно нам, чтобы сиденье перемещалось по игровому миру вместе с “автомобильчиком”;

— в строке 11 объявляем переменную **seat_occup**, которую будем использовать, чтобы понимать сидит кто-либо на сиденье или нет. Исходно значение **seat_occup** равно **false**, что означает, что сиденье свободно;

— в строках 12…33 описываем функцию **seat_CHANGE**, которая будет обеспечивать и контролировать посадку персонажа игрока на сиденье. В этой функции:

— в строке 13 присваиваем переменной **humanoid** значение **seat.Occupant**. Это запускает показ анимации “усаживания” гуманоида на сиденье и прикрепляет гуманоида к сиденью “сварным швом”;

— в строке 14 объявляем переменную **i** и присваиваем ей значение “0”. Эта переменная будет у нас счетчиком времени, которое гуманоид проводит, сидя на сиденьи;

— в строке 15 присваиваем переменной **seat_occup** значение **true** ведь гуманоид уже сел на сиденье;

— в строке 16 если гуманоид сел на сиденье, то:

— в строке 17 передаем переменной **character** Родителя гуманоида (то есть, передаем переменной **character** одноименную часть модели игрока);

— в строке 18 с помощью сервиса **Players** и функции **GetPlayerFrom Character** (ПолучитьИгрокаИзПерсонажа) находим в объекте **character**
игрока и передаем его в локальную переменную **player**;

— в строке 19 (в цикле **if - then**), когда **player** найден:

— в строке 20 печатаем имя игрока и, следом, печатаем: “сел” (“**has set down**”);

— в строках 21…29 организуем цикл **while i<10**, с каждым шагом которого мы:

— в строке 22 начинаем отсчет времени, наращивая через каждую секунду (см. строку 28) значение переменной **i** на “1”;
— в строке 23 печатаем каждое новое значение переменной **i** (просто для контроля отсчета времени);

— в строках 24..27 в операторе **if-then** проверяем просидел ли гуманоид на сиденье уже 10 секунд и, если уже просидел, то:

— в строке 25 с помощью функции **FindFirstChild** (НайтиПервогоПотомка) с именем **SeatWeld** находим и уничтожаем “сварной шов” между гуманоидом и сиденьем (то есть, поднимаем гуманоида с сиденья);

— в строке 26 меняем значение переменной **seat_occup** на **false** ведь мы уже подняли гуманоида с сиденья;

— команда **return** строки 30 позволяет повторять оператор **if - then** строк 19…31 до тех пор, пока этому оператору не будут переданы данные игрока из сервиса **Players** и он не сделает то, что ему задано выполнить во “внутреннем” коде этого оператора;

— строка 34 подключает сиденье **seat** к функции получения сигнала об изменении свойства гуманоида **"Occupant"** (название этой функции **GetPropertyChangedSignal** переводится, как “ПолучитьСигналОб ИзмененииСвойства”). А эта функция, в свою очередь, подключает к сиденью созданную нами функцию **seat_CHANGE**. То есть, если гуманоид сел, это приводит к изменению его свойства **Seat.Occupant**, что

вызывает появление сигнала об изменении свойства и, сразу же, подключает сиденье к созданной нами функции **seat_CHANGE**;

— строки 35…38 цикла **while true do** обеспечивают передвижение нашего “автомобильчика” вдоль оси координат X.

Запустите программу. Направьте своего аватара к сиденью, и как только он коснется сиденья, вы увидите, что аватар сядет на сиденье и поедет на “автомобильчике”. Однако, через 10 секунд он самостоятельно встает с сиденья, а “автомобильчик” продолжит свою поездку. Повторите посадку аватара на сиденье автомобильчика — через 10 секунд после посадки аватар опять будет автоматически высажен с “автомобильчика”.

Распечатки имени игрока, севшего на сиденье в игре будет не видна. Мы просто показали вам как использовать окно **Ouput** при отладке программ. Обязательно поэкспериментируйте с сиденьями.

29. Создаем тексты и изображения на игровом поле



Рис. 29.1 Тексты на игровом поле

Roblox Studio позволяет создавать разнообразные визуальные подсказки и оповещения в виде текстов и изображений, расположенных внутри игрового пространства.

Чтобы разместить текст на какой-либо **part** игрового поля, подсветите **part** в окне **Explorer**, нажмите кнопку ⊕ и выберите в выпадающем меню **SurfaceGui** (ПоверхностныйGui).

Сокращение (аббревиатура) “**Gui**” означает **GraphicUserInterface** — Графический ИнтерфейсПользователя. Любой интерфейс — это

набор средств, обеспечивающих взаимодействие с пользователям. В нашем случае, речь идет о взаимодействии с игроками путем выдачи им текстовых или визуальных сообщений.

Итак, откройте окно **Properties SurfaceGui** и установите:
— в свойстве **Face**, ту грань **part**, на которой должен появиться текст (**Left, Right, Front, Back, Top, Bottom**). Но, это в том случае, если вы знаете, где какая грань у вашей **part**. В противном случае, вы можете сначала сделать вашу **part** прозрачной — где бы потом не появился текст, вы увидите его;
— свойство **SizingMode** установите в опцию **PixelsPerStud**;
— свойство **PixelsPerStud** сделайте равным “50”.

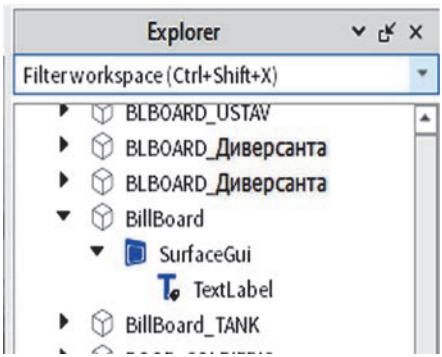


Рис. 29.2 TextLabel в SurfaceGui

Подсветите **SurfaceGui** в окне **Explorer**, нажмите кнопку ⊕ и выберите в выпадающем меню **TextLabel** (Текстовый Ярлык). Этот объект позволяет отображать фоновый прямоугольник с рамкой или без нее и текст. На фоне созданной **part** появится маленький прямоугольник белого цвета, который и определяет границы текста. На прямоугольнике будет виден исходный текст “**Label**”.



Рис. 29.3 Изменение размеров *TextLabel* свойством *Size{}{}*

Основные свойства **TextLabel** доступные для изменения в окне **Properties** перечислены ниже:

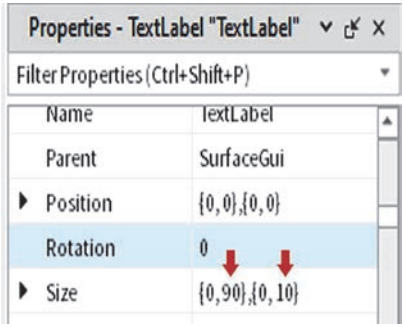


Рис. 29.4 Установка размеров

— **Size {0, ...},{0, ...}** задает размеры прямоугольника, определяющего границы фона и текста. Измените размеры, как вам это нужно;
— **BackgroundColor3** определяет цвет фона текстового сообщения;
— **BackgroundTransparency** определяет степень прозрачности фона;
— **BorderColor3** определяет цвет рамки вокруг фона текста;
— **BorderMode** определяет вид рамки вокруг фона;
— **Visible** управляет видимостью текста;
— **Text** — поле предназначенное для ввода текста (исходно в нем содержится текст “**Label**”);
— **Font Face** позволяет выбрать тип шрифта, которым будет написан текст, а также его стиль (основной, полужирный, жирный, наклонный;
— **LineHeight** определяет расстояние между строками текста;
— **TextColor3** определяет цвет букв текста;
— **TextSize** определяет размер шрифта текста;
— **TextStrokeColor3** и **TextStrokeTransparency** задают цвет и уровень прозрачности обводки букв текста по их контуру;
— **TextWrapped** позволяет отображать текст на нескольких строках;
— **TextXAlignment** и **TextYAlignment** определяют горизонтальное и вертикальное выравнивание текста внутри **Surfasegui**. Возможны: **Left, Right, Center** для горизонтального и **Top**, **Bottom** и **Center** для вертикального выравнивания.



Рис.29.5 Пример текста в игре

Действия по размещению на **part** картинок сходны с размещением текста: сначала нужно создать **part**, затем вложить в нее **SurfaceGui**, а в нее вложить уже не **TextLabel**, а **ImageLabel**.

Некоторые свойства **ImageLabel** сходны со свойствами **TextLabel.** Размещать можно картинки типа **Decal**, загруженные с сайта Roblox.



Рис. 29.6 Пример использования *ImageLabel* для вывода анимации на поверхности одной из part игрового поля

Приведем пример кода, выводящего на поверхность **part** анимированную картинку (вставьте его в скрипт внутрь **ImageLabel**).

```
1     local imageLabel = script.Parent
2     imageLabel.Image = "rbxassetid://848623155"
3     imageLabel.ImageRectSize = Vector2.new(256, 256)
4     local fr_array =  { Vector2.new(0, 0), Vector2.new(1, 0),
  Vector2.new(2, 0), Vector2.new(3, 0), Vector2.new(0, 1),
  Vector2.new(1, 1), Vector2.new(2, 1),Vector2.new(3, 1),
  Vector2.new(0, 2), Vector2.new(1, 2),Vector2.new(2, 2),
  Vector2.new(3, 2) }
5     while true do
6           for _, frame in ipairs(fr_array) do
7                 imageLabel.ImageRectOffset = frame * imageLabel.
  ImageRectSize
8                 wait(0.1)
9           end
10    end
```

— в строке 2 мы загружаем в **imageLabel** картинку анимации с сайта Roblox, состоящую из двенадцати слайдов;
— в строке 3 определяем, что размер картинки **imageLabel** составляет 256 x 256**;**
— строка 4 присваивает переменной **frames** содержимое списка всех используемых в анимированной картинке слайдов;
— в строке 5 создаем бесконечный цикл, обеспечивающий “прокрутку” слайдов картинки одну за другой;
— цикл **for_ in ipairs** обеспечиваем “перебор” одного за другим слайдов, входящих в список.

30. Программируем неигровые персонажи

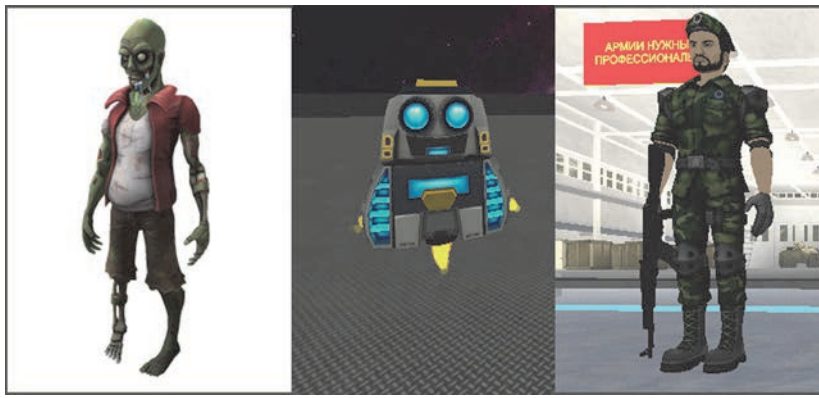


Рис. 30.1 Non-Player Characters (NPC)

Все хорошо, но пока еще разработанные вами игры малоизвестны, и их посещают лишь ваши друзья. Поэтому в них маловато действующих лиц. Но, вы можете значительно оживить вашу игру добавив в нее **Non-Player Characters (NPC**) — неигровых персонажей.
Имеющиеся в Roblox Studio **NPC** имеют разную внешность, а их поведение можно программировать. **NPC-зомби** и **NPC-солдаты** могут даже защищать какие-то области игрового пространства, атакуя игроков или других **NPC**. Вам самим и вашим друзьям будет веселее и интереснее с ними.

Чтобы использовать NPC в своей игре:
— перейдите по одной из указанных здесь ссылок и выберите себе NPC:
* Зомби — https://www.roblox.com/library/3924238625/Drooling-Zombie-Rthro
* Солдаты — https://www.roblox.com/library/3924234975/Soldier-Rthro
* Роботы — https://www.roblox.com/library/3924232032/RO-01-Robot
* Роботы — https://www.roblox.com/library/3924229481/NP-C-9000-Robot
— на выбранной странице NPC нажмите зеленую кнопку **Get** (Получить) и подтвердите свое желание получить NPC бесплатно;
— в Roblox Studio откройте окно **Toolbox**, а в нем откройте окно **Inventory** (Инвентарь). Выбранный вами **NPC** автоматически появится внутри этого окна. Можете сразу же загрузить всех доступных **NPC**;

— выберите в окне **Inventory** нужного **NPC** и кликните по нему, чтобы добавить его в свою игру.

Рассмотрим состав моделей **NPC** в окне **Explorer**. Обычно модель содержит следующие объекты:

— папку **Animations** с анимациями типа **AttackAnimation** (Анимация Атаки);
— папку **Initial Poses** (Исходные позы);
— скрипт **Animate**, который загружает и воспроизводит анимацию;
— аксессуары (такие как головные уборы, оружие и т. д.);
— скрипт **Health**, восстанавливающий здоровье **NPC**;
— папку **Humanoid**, которая управляет внешним видом, здоровьем, скоростью передвижения и другими параметрами;
— скрипт **NPC**, который определяет его поведение (блуждание, атаки);
— скрипт **RbxNpcSounds**, управляющий звуковыми эффектами;
— различные части тела персонажа, модели оружия, и другое.
— объект **Configuration** (Конфигурация), также входящий в состав модели **NPC** является контейнером объектов-свойств, определяющих поведение **NPC**. Некоторые свойства являются логическими, то есть, могут иметь значение **true/false**, либо требуют установки “галочки”. Другие свойства требуют ввода числового значения. Чтобы задать нужное значение каждому из свойств, выберите свойство и измените его значение в окне **Properties:**
* **DestroyOnDeath** — уничтожает NPC после его смерти;
* **PatrolEnabled** — заставляет **NPC** блуждать (или патрулировать);
* **PatrolRadius** — определяет расстояние, на которое **NPC** отходит от его начальной позиции, если **PatrolEnabled** имеет значение **true**;
*** RagdollEnabled** — заставляет **NPC** после смерти не разваливаться на части;
AttackDamage — определяет, величину урона жертвы при атаке NPC;
AttackDelay — задает время между выстрелами;
AttackMode — определяет кого солдат будет атаковать (см.ниже);
AttackRadius — определяет расстояние, на котором **NPC** атакует;
ClipCapacity — определяет, число патронов в магазине оружия;
ReloadDelay — определяет время перезарядки оружия **NPC**.

Для управления агрессией **NPC** по отношению к другим **NPC** или игрокам используют специальные метки (**теги**), которые могут быть назначены, как игрокам, так и NPC.

Например:
— если **NPC-солдату** свойство **AttackMode** установлено равным ”1”, то, чтобы этот **NPC-солдат** атаковал других игроков, они должны быть помечены тегом **SoldierEnemy** (ВражескийСолдат);

— если **AttackMode** солдата установлен на “2”, все объекты не имеющие тега **SoldierFriend** (СолдатСоюзник), будут атакованы;

— если **AttackMode** солдата установлен на “3”, все теги игнорируются, и **NPC-солдат** будет атаковать всех персонажей;

— тегом **ZombieFriend** (ДругЗомби) пользуются для управления зомби. Они становятся послушными персонажу, отмеченному таким тегом.

Чтобы назначить нужный тег какому-то NPC, нужно скопировать приведенную ниже строку кода **в окно командной строки** Roblox Studio и нажать клавишу Enter:

```
game:GetService("CollectionService"):AddTag(model, tag)
```

где: "**CollectionService**"— название специального сервиса, обеспечивающего присвоение или снятие тегов;
AddTag — команда присвоения тега;
RemoveTag — команда отключения ранее присвоенного тега;
model — название главного объекта NPC;
tag — тег, который нужно присвоить.

Чтобы присвоить теги игрокам, нужно добавить в **StarterCharacterScripts** скрипт, вызывающий специальный сервис **CollectionService** присвоения и отключения тегов (см. пример скрипта ниже).

Например, в одной из военных игр сценарием предусмотрена возможность, при входе в игру, выбрать для себя роль диверсанта. А в охране военной базы задействованы **NPC-солдаты**. Поэтому, в игровом пространстве те, кто выбирали роль диверсанта, проходя в указанное им место, наступали на скрытую от их глаз **part**, обеспечивающую автоматическое получение ими соответствующего тега. Вот скрипт такой **part,** имеющей название **pointPart_DIVERSANT**:

```
1       part = workspace.pointPart_DIVERSANT
2       part.Touched:Connect(function(hit)
3               if hit.Parent == script.Parent then
4                       game:GetService("CollectionService"):AddTag(script.Parent,
 "SoldierEnemy")
5                       game:GetService("CollectionService"):RemoveTag(script.
 Parent, "SoldierFriend")
6                       print("Tag added")
7               end
8       end)
```

Поясним скрипт:
— строка 1 определяет **part**, обеспечивающую смену тегов;
— в строке 2 индикатор события **Touched** подключает тех, кто коснулся **part** к функции меняющей их теги;
— цикл **if - then** строк 3…7 определяет факт касания игроком **part**, снимает (в строке 5) тег **SoldierFriend** и присваивает (в строке 4) тег **SoldierEnemy**.

31. Телепортация игроков

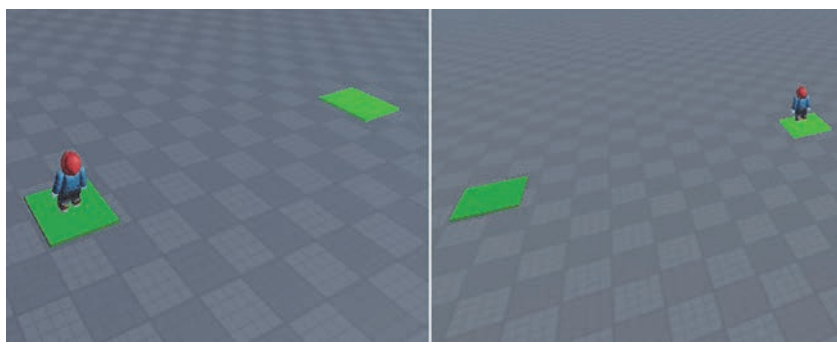


Рис. 31.1 Тест телепортации игроков

В Roblox имеется специальный сервис **TeleportService**, который способен перемещать игроков даже из одной вашей игры в другую вашу игру, но этот сервис достаточно сложен и его нельзя протестировать в Roblox Studio. Если же вы хотите создать в игре возможность мгновенного перемещения игроков из одной части вашего игрового пространства в другую его часть потребуется более простая телепорта-

ция. Например, аватар наступает на какой-то ничем не примечательный предмет и оказывается в подземелье. Или, наоборот, находит ключ и мгновенно выбирается из сложного лабиринта.

Давайте рассмотрим вариант именно такого телепорта. Создайте в **Workplace** две **part,** например, в виде площадок размером 4x1x4 шипа, расположенных на некотором расстоянии друг от друга. Назовите их **Teleport_1** и **Teleport_2**, покрасьте их в разные цвета, чтобы проще было различать во время телепортации, и скопируйте в них приведенные ниже скрипты:

Скрипт_1:

```
1       script.Parent.Touched:Connect(function(hit)
2               if game.Players:GetPlayerFromCharacter(hit.Parent) then
3                       if script.Parent.Locked == false and script.Parent.Parent.
  Teleport_2.Locked == false then
4                               script.Parent.Locked = true
5                               script.Parent.Parent.Teleport_2.Locked = true
6                               hit.Parent:MoveTo(workspace.Teleport_2.Position)
7                               wait(3)
8                               script.Parent.Locked = false
9                               script.Parent.Parent.Teleport_2.Locked = false
10                      end
11              end
12      end)
```

Скрипт_2:

```
1       script.Parent.Touched:Connect(function(hit)
2               if game.Players:GetPlayerFromCharacter(hit.Parent) then
3                       if script.Parent.Locked == false and script.Parent.Parent.
  Teleport_1.Locked == false then
4                               script.Parent.Locked = true
5                               script.Parent.Parent.Teleport_1.Locked = true
6                               hit.Parent:MoveTo(workspace.Teleport_1.Position)
7                               wait(3)
8                               script.Parent.Locked = false
9                               script.Parent.Parent.Teleport_1.Locked = false
10                      end
11              end
12      end)
```

Запустите программу и протестируйте телепорты. Вы увидите, что, как только ваш аватар наступает на одну из площадок, он мгновенно оказывается на другой площадке. Все работает! Это можно использовать очень круто в самых разных играх!

Поясним скрипты:
— строка 1 скриптов подключает индикатор события **Touched** телепорта к безымянной функции и передает ей параметр касание (**hit**);
— внутри **function(hit)**, в строке 2 функция **GetPlayerFromCharacter** проверяет, действительно ли объект, который коснулся телепорта является одним из игроков и если это так, то:
— в строке 3 проводится проверка: не заблокирована ли работа обоих телепортов (**LOCKED** — значит "**блокировка**"), и, если телепорты не заблокированы, то:
— в строках 4, 5 работа обоих телепортов блокируется. Это нужно, чтобы избежать сбоев в работе. Вдруг, какой-то другой игрок мгновением позже тоже коснется какого-то из телепортов и все перепутается. Не поймешь, как отделить одного игрока от другого, ведь они оба уже коснулись телепорта;
— в строке 6 тот игрок, который коснулся телепорта перемещается в позицию (**Position**), соответствующую позиции другого телепорта. Кстати, функцию **Move** способную переместить игрока или NPC в нужное место, можно использовать и в таком виде:

```
game.Players.LocalPlayer:Move(Vector3.new(X, Y, Z ), true)
```

где: X,Y,Z — координаты места, куда нужно направить игрока или NPC.

Если второй аргумент функции **Move** не указан (то есть, равен по умолчанию **false**), то **Vector3** перемещает игрока в координатах игрового поля, если же он равен **true**, то перемещение осуществляется относительно телекамеры игрока Roblox.

— в строках 8, 9 проводится (после задержки в строке 7) разблокировка обоих телепортов, чтобы они вновь могли реагировать на касания игроков.

Обратите внимание на величину задержки и размеры телепортов. При некоторых их величинах (например, при небольшой задержке, например, в 1 сек и больших размерах телепортов игрок не сможет быстро

покинуть телепорт) может происходить серия телепортирований — и игрок будет постоянно перемещаться между телепортами. Разберитесь с этим самостоятельно.

Естественно, приведенные скрипты — не единственно возможный вариант телепорта. Давайте рассмотрим вариант односторонней телепортации: из **Teleport_1** в **Teleport_2**. Тогда, очевидно, что скрипт в **Teleport_2** нам будет не нужен — удалите его. Саму платформу **Teleport_2** можно сделать маленького размера, прозрачной и можно снять “галочку” с ее свойства **CanCollide**. А код первого телепорта можно заменить на такой:

```
1    script.Parent.Touched:Connect(function(hit)
2        if game.Players:GetPlayerFromCharacter(hit.Parent) then
3            hit.Parent.HumanoidRootPart.CFrame = workspace.
  Teleport2.CFrame
4        end
5    end)
```

Вы, конечно, заметили, что строки 2 нового скрипта и прежних совпадают, а строка 3 сразу меняет положение игрока (при помощи **CFrame**). Напомним, что **HumanoidRootPart** — это та самая часть модели гуманоида, которая отвечает за его передвижение.

32. Создание команд игроков

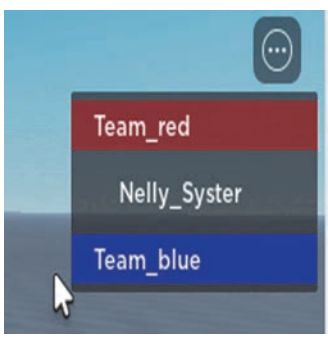


Рис. 32.1 Список игроков в командах

Вы можете разделить игроков в своей игре на несколько команд, используя службу **Teams** (Команды) Roblox Studio. Включение службы **Teams**, автоматически устанавливает цвета имен, которые высвечиваются над каждым из аватаров игрока, чтобы можно было различать тех, кто принадлежит к разным командам.

Вы можете создать новые команды, выполнив следующие действия:
— наведите указатель мыши на объект **Teams** в окне **Explorer** и щелкните кнопку ⊕, чтобы отобразить список добавляемых объектов;
— выберите и вставьте новый объект **Team**. После этого, откройте окно свойств нового объекта **Team** и измените его **Name** (Название команды) и назначьте ему **TeamColor** (Цвет команды);
— повторите действия чтобы определить и вторую команду.

По умолчанию Roblox автоматически распределяет новых игроков, присоединяющихся к игре, в команду с наименьшим количеством участников.

Имея несколько команд игроков, вы можете разделять их полномочия на определенные действия. Например, разрешать проход через дверь игрокам одной из команд, но, запрещать такой проход игрокам из других команд. Вот пример кода для такой двери:

```
1   local Players = game:GetService("Players")
2   local door = Instance.new("Part")
3   door.Anchored = true
4   door.Size = Vector3.new(7, 10, 1)
5   door.Position = Vector3.new(0, 5, 0)
6   door.Parent = workspace
7   local debounce = false
8   door.Touched:Connect(function(hit)
9       if not debounce then
10          debounce = true
11          if hit then
12              local player = Players:GetPlayerFromCharacter
    (hit.Parent)
13              if player and player.TeamColor == BrickColor.new
    ("Bright red") then
14                  door.Transparency = 0.5
15                  door.CanCollide = false
16                  wait(3)
17                  door.Transparency = 0
18                  door.CanCollide = true
19              end
20          end
21          wait(0.5)
```

```
22                  debounce = false
23              end
24      end)
```

Поясним скрипт:
— строка 1 подключает сервис **Players** и присваивает его значение переменной Players;
— строки 2…6 создают дверь и определяют ее свойства;

— строка 7 создает переменную **debounce** и присваивает ей значение **false**. Эта переменная нужна, чтобы иметь возможность устранить явление “дребезга”, которое может произойти при нескольких (одно за другим) касаниях двери аватаром игрока;

— строка 8 подключает с индикатор событий типа **Touched** двери к функции с параметром **(hit)** (касание);

— строка 9 проверяет равно ли значение переменной **debounce** значению **false**, то есть, проверяет: было ли уже касание двери, и, если “нет”, то:
— в строке 10 меняет значение переменной **debounce** на **true**, не разрешая тем самым повторный запуск **function(hit)** до ее завершения;

— строка 11 проверяет было ли касание двери, и, если “да”, то:

— в строке 12 присваивает переменной **player** значение игрока который коснулся двери;

— строка 13 проверяет соответствует ли цвет команды игрока, прикоснувшегося к двери цвету “**Bright red**” и, если соответствует, то:

— строки 14…16 меняют прозрачность двери и разрешают проход через нее на 3 секунды;

— строки 15,16 закрывают проход через дверь;

— строка 22 меняет значение переменной **debounce** вновь разрешая запуск функции **function(hit)**.

33. Разработка игры



Рис. 33.1 Игровой мир

Перед тем, как профессиональные разработчики приступают к разработке новой игры, они, обычно,составляют план, который подробно описывает, как все будет выглядеть и происходить в игре. Этот этап разработки игр называется “**пре-продакшн**”. Его результатом является документ **Game Design Document** (GDD).

Профессиональные **GDD** часто бывают очень объемными, и вам этого не нужно. Но, чтобы ваша работа была более эффективной и успешной, мы рекомендуем и вам начать с выяснения того, что игроки, вошедшие в вашу игру, будут в ней делать и с определения игрового мира, в котором все будет происходить. Четкое видение этого поможет ускорить разработку игры и упростит обсуждение ваших идей с друзьями, чтобы узнать их мнение о вашей игре (стоит ли терять на нее время?).



Откройте текстовый редактор (в нем удобнее делать правки, которые обязательно будут возникать в процессе создания документа) и запишите:

— что игроки увидят, войдя в игру (мистический мир, равнину, покрытую деревьями и реками, космическую станцию, городскую среду, военный полигон…);
— предметы, которые игрокам нужно собирать, чтобы заслужить награду (золотые самородки, блестящие кристаллы, удивительные артефакты украденные космическими пиратами у внеземной цивилизации, потерянных домашних животных, боевые награды, воинские звания…);

— испытания, которые должны пройти игроки в вашей игре (паркуры, стрелковые полигоны, минные поля, подземные лабиринты, трудные трассы, которые нужно преодолевать на танке или бронетранспортере, нападения врагов…);

— доступное игрокам оружие или инструменты, необходимые для сбора нужных предметов или выживания в опасных условиях (ключи, перчатки, ошейники для животных, карты местности…);

— описание макета карты, которое позволит понять: где игроки начинают игру, где будут размещаться собираемые предметы, где будут находиться опасные места или вражеские персонажи…;

— где и какие подсказки игрокам вы планируете разместить по ходу игры;

— описание возможностей командной игры (цель, которую должна достигнуть команда-победитель…)

— описание ролей (если это ролевая игра) и условий, обеспечивающих переход из одной роли в другую.

Только после понимания всего этого можно приступать к созданию ландшафта с помощью Terrain Editor.

34. Публикация игры

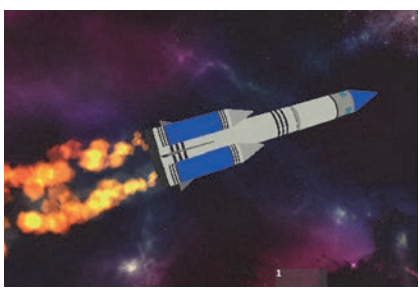


Рис. 34.1. Запуск игры в Roblox

Опубликуйте свою игру, чтобы другие пользователи Roblox могли в нее играть:

— в строке меню выберите **File - Publish to Roblox** — откроется окно, в котором нужно указать:

— название игры;

— краткое описание (что за игра);

— имена разработчиков;

— жанр игры;
— устройства, на которых игра работает;
— нажмите кнопку **Create** (Создать).
— после на вкладке **HOME** нажмите кнопку **Game Settings**;
— на вкладке **Playability** (Разрешения) выберите, какие пользователи могут получить доступ к игре:
— **Friends** (Друзья);
— **Public** (Все желающие);
— **Private** (Только для разработчика).
— в правом нижнем углу нажмите кнопку **Save** (Сохранить).
— в правом верхнем углу Studio нажмите кнопку **Share** (Поделиться).

35. Не только игры

Roblox Studio — отличная среда, которую можно использовать для создания прикладных программ абсолютно любого назначения. Вот пример.


ДИПЛОМ
ВРУЧАЕТСЯ!
Вербиченко Юрий
УЧАСТНИКУ
Всероссийского конкурса естественно-научных и инженерных проектов школьников и студентов «Реактор»
Проект: «Компьютерная программа профилактики близорукости»
Трек: «Создавай!»
Площадка: (Ульяновск)
А. А. Шенфельд
РЕАКТОР
— 2021 —

Рис. 35.1 Не только игры

Врачи фиксируют значительный рост близорукости у подростков и связывают его с повышенной нагрузкой на глаза из-за компьютеров. Снизить нагрузку на глаза можно регулярной гимнастикой для глаз. Программ для отдыха и тренировки цилиарных мышц глаз много. Однако, все они реализованы в 2D-графике, что не позволяет создать ситуацию аналогичную работе глаз на улице - когда им приходится фокусироваться то на близких, то на удаленных объектах, отслеживая их движения, а также быстро переводить взор на разные объекты, находящиеся под случайными углами зрения. Такой режим работы глаз врачи считают самым полезным и (для профилактики близорукости) настоятельно рекомендуют увеличить время прогулок.

Соавтор этой книги эффективно усовершенствовал гимнастику для глаз за счет использования возможностей 3D-графики Roblox. Проводя сеанс гимнастики с его программой каждый почувствует, что мышцы его глаз действительно работают, как на улице. Это улучшает их кровоснабжение, клеточное питание, снабжение кислородом, отвод “шлаков, поддерживает тонус (https://www.youtube.com/watch?v=w7n_6DWOJ20&t=186s).



Подумайте, куда автор программы спрятал аватар своего игрока, найдите ответ и разработайте свою прикладную программу в Roblox Studio.

Владимир Рубочкин, Юрий Вербиченко

Азбука программирования игр в Roblox Studio 10+

Ответственный за выпуск: **В. Митин**
Верстка и обложка: **СОЛОН-Пресс**

ООО «СОЛОН-Пресс»
115487, г. Москва,
пр-кт Андропова, дом 38, помещение № 8, комната № 2.
Формат 60×88/16. Объем 10 п. л. Тираж 1000 экз.