Министерство образования Республики Беларусь

Учреждение образования
«Гомельский государственный технический
университет имени П. О. Сухого»

Кафедра «Информационные технологии»

ИГРОВЫЕ ПЛАТФОРМЫ

УЧЕБНО-МЕТОДИЧЕСКОЕ ПОСОБИЕ
для студентов специальности
1-40 05 01-12 «Информационные системы
и технологии (в игровой индустрии)»
дневной формы обучения

Гомель 2022

УДК 004.9(075.8)
ББК 22.19я73
И27

Рекомендовано научно-методическим советом
факультета автоматизированных и информационных систем
ГГТУ им. П. О. Сухого
(*протокол № 4 от 28.12.20 г.*)

Составитель *Е. В. Комракова*

Рецензент: зав. каф. «Информатика» ГГТУ им. П. О. Сухого
канд. техн. наук, доц. *Т. А. Трохова*

Игровые платформы : учеб.-метод. пособие для студентов специальности
И27 1-40 05 01-12 «Информационные системы и технологии (в игровой индустрии)» днев. формы обучения / сост. Е. В. Комракова. – Гомель : ГГТУ им. П. О. Сухого, 2022. – 316 с. – Систем. требования: PC не ниже Intel Celeron 300 МГц ; 32 Mb RAM ; свободное место на HDD 16 Mb ; Windows 98 и выше ; Adobe Acrobat Reader. – Режим доступа: https://elib.gstu.by. – Загл. с титул. экрана.

Предназначено для теоретической и практической подготовки по дисциплине «Игровые платформы». Дисциплинарный курс знакомит студентов с основами теории и практики, необходимыми для разработки и визуального отражения компьютерных игр различных жанров.

Для студентов специальности 1-40 05 01-12 «Информационные системы и технологии (в игровой индустрии)» дневной формы обучения.

УДК 004.9(075.8)
ББК 22.19я73

Введение

Учебно-методическое пособие, предназначенное для студентов специальности 1-40 05 01-12 «Информационные системы и технологии (в игровой индустрии)». В пособии последовательно излагается материал, посвящённый теории и практике, необходимыми для разработки и визуального отражения компьютерных игр различных жанров.

В первом разделе описано создание двумерных игр различных жанров, таких как: простая аркадная игра; игра, учитывающая законы физики; шутер; карточная игра; игра в слова и приключенческая игра.

Во втором описывается создание трехмерной игры, как от первого лица, так и от третьего лица. В третьем разделе рассматриваются мобильные и мультиплатформенные игры. В четвертом приведено описание оптимизации игр в Unite. Рассмотрены приемы разработки сценарием, пакетная обработка, разгон физического движка и динамическая графика.

Введение в игровые платформы

Вместе с созданием первых игр программисты пришли к тому, что каждая игра содержит общие компоненты, даже несмотря на различие аппаратных платформ. А первые игры имели место на игровых автоматах размером с холодильник.

Общая для игр функциональность – графические решения, игровые механики, расчет физики и другое – стала выделяться в отдельные библиотеки, но, для того чтобы быть «игровым движком» было еще далеко. Во многом это было связано с серьезным различием программно-аппаратных платформ и неопределенности в самих играх. Ведь жанры и типы игр еще предстояло изобрести, при том, что многие первые игры были текстовыми.

Собственно, именно для ранних адвенчур и платформеров и стали возникать игровые движки, особенно с развитием графики – хорошим примером можно назвать Adventure Game Interpreter (AGI). При разработке King's Quest в далеком 1984 году, программисты Sierra On-Line столкнулись с неудобством низкоуровневой разработки столь сложной и перспективной по графике в те времена игры – и разработали набор решений, которым и стал AGI. Всего на нем было выпущено 14 различных игр за 5 лет на 7 различных платформах, поэтому понятие «кроссплатформенность» было важным уже тогда.

Однако, движки того времени редко выходили за пределы изначальной компании-разработчика и, как правило, были достаточно узкоспециализированными под конкретный жанр игры.

Начало

Ситуация начала меняться в 1993-м году после выхода игры Doom от компании id Software. Хотя при ее разработке использовались наработки движка Wolfenstein 3D, с точки зрения возможностей и модульности в ней был совершен настоящий технологический прорыв. В то время видеопроцессоры были не способны эффективно работать с трехмерной графикой, поэтому Джон Кармак (ведущий программист движка) выполнял все необходимые математические вычисления, служащие для манипуляции с трехмерными объектами, светом, затенением, наложением текстур и прочего самостоятельно. В результате, изображение выглядело трехмерным, на самом деле таковым не являясь. Поэтому Doom engine (первая версия id Tech) был не истинно трехмерным, а псевдотрехмерным. Но важно то, что техни-

ческая составляющая этой игры задала стандарт для того, что могло называться игровым движком. А именно, движок Doom был модульным, представлял из себя набор подсистем, в нем каждый четко отделенный программный слой отвечал за обработку своей порции данных. В результате, использовать его для различных игр (Hexen, Heretic, Strife) и силами сторонних разработчиков (Raven Software и Rogue Entertainment) стало намного проще. Поэтому появление игровых движков относят к середине 90-х годов 20-го века, то есть тогда окончательно сформировалось определение игрового движка в современном смысле.

Игровой движок представляет своеобразную узкоспециализированную операционную систему, поскольку включает все модули последней. В него входят: система управления памятью, графическая подсистема, система ввода, аудио подсистема, искусственный интеллект, физическая подсистема, сетевая подсистема, редактор игровых уровней и другое. Кроме того ядро движка может предоставлять особый подход к работе с файлами – файловую (ресурсную) систему, а так же отличающиеся от основной операционной системы средства работы с многопоточностью. Современный игровые движки вдобавок включают интерпретатор скриптового языка, заточенного для описания игровой логики, а нередко и полностью визуальный ее редактор. Его использование позволяет абстрагироваться от описания низкоуровневых команд и инструкций, а сконцентрироваться на геймплее. На этом составляющие движок компоненты не ограничиваются, их может быть как больше, так и меньше.

Цели

Игровой движок в первую очередь создается в целях упрощения и ускорения разработки. Поэтому включает средства для создания игрового мира – level-моделинга, импорта объектов, текстурирования, загрузки и анимации персонажей, создания визуальных эффектов, настройки физики и прочего.

Второй значительной целью разработки движка является кроссплатформенность или платформонезависимость разрабатываемой игры. То есть возможность ее запуска с минимально возможными изменениями. Совсем без изменений на другой платформе осуществить запуск игры не удастся из-за аппаратных различий, в том числе: размеров экрана, средств и способов управления и др.

Развитие игровых движков происходит вместе или под влиянием развития аппаратных и программных платформ, вместе с появлением новых игровых жанров и изменениями вкусов пользователей. Коротко говоря, развитием игровой индустрии в целом.

Генезис графических систем

В середине 90-х после появления видеопроцессоров, способных обрабатывать трехмерную графику стали появляться программные интерфейсы, упрощающие ее разработку. Вслед за кроссплатформенным OpenGL на сцену в составе DirectX вышел Direct3D для Windows. Эти 2 визуализатора на много лет вперед определили способы графического вывода в играх.

В 1996-м году вышла игра Quake на Quake Engine. Этот движок оказал колоссальное влияние на игровую индустрию.

Почти до конца десятилетия на рынке промежуточного программного обеспечения для игр (другими словами, игровых движков) практически единол ично ритм задавала id Software. Однако в 1998-м году компания Epic Games выпустила успешную игру Unreal на одноименном движке – с настоящим технологическим прорывом по уровню графики. Ведущим программистом движка стал основатель Epic Тим Суини. Тим наравне с Кармаком является наиболее значимой фигурой в истории движков игровой индустрии – и Unreal Engine в его 3 и 4 версиях очень популярен и сейчас. Год спустя от Epic вышла ставшая еще более популярной игра Unreal Tournament.

В это же самое время конкурирующая компания-разработчик – id Software выпустила мультиплеерную игру Quake 3 Arena (на движке id Tech 3), ровно как Unreal Tournament включающую сетевые баталии.

Эти две игры стали флагманами индустрии, определив ее развитие на годы вперед.

На рынке было не так много игроков. Поэтому их продукция была очень дорога, и флагманские движки лицензировались только достаточно крупными разработчиками,

Ситуация начала коренным образом меняться примерно в середине первого десятилетия 21-го века. Тогда на рынке и в свободном доступе стало появляться большое количество средств для разработки игр. Бизнес промежуточного ПО (middleware) стал набирать обороты. Сначала рынок заполнился графическими фреймворками: Ogre, DarkGDK и др., предоставляющие программисту высокоуровневую

прослойку над графическим API. В то же время отличающиеся от игровых движков полным отсутствием внутриигровых редакторов.

Затем на рынок пришли полноценные игровые движки по ценам, уместным для небольшой инди-команды разработчиков, среди них: Torque 3D, Unity 3D, и многие другие. Даже стартовавшие как флагманские движки – например, CryEngine от Crytek и ранее упомянутый Unreal Engine – стали использовать намного более доступную ценовую политику и стали доступны даже начинающим разработчикам.

Важным трендом игровой индустрии стали казуальные игры. Эти, по своей сути, незамысловатые, но красочные, не требующие бешеного взаимодействия с клавиатурой и мышкой головоломки с технической точки зрения были проще трехмерных хардкорных шутеров, поэтому для их разработки не понадобилось сильной модификации универсальных движков. Но, зато, в индустрии появились новые игроки, такие как: Torque Game Builder, HGE и другие.

В это же время, благодаря World of Warcraft, в игровой индустрии стали очень популярны MMORPG – а параллельно многие жанры делали все большую ставку на мултиплеер. Целый ряд движков не смог предоставить пользователям новую функциональность для клиент-серверных приложений, поэтому они ушли в небытие. Другие движки были адаптированы для мультиплеерного мира путем разработки для них серверных решений, так для Unity 3D были разработаны Photon и SmartFox. Третий тип универсальных движков, изначально являясь клиент-серверным, не почувствовал изменений. К нему относится Torque 3D. Также на рынке появились новые движки, предназначенные для глобальных многопользовательских игр, например HeroEngine, BigWorld, объединяющие масштабируемое под тысячи игроков серверное решение и доступный конкретному игроку клиент.

На рынке еще с 90х существовали браузерные игры, а затем второе рождение им дали социальные сети. необходимость эффективно создавать игры для браузера не осталась незамеченной. Разработчики универсальных движков, например Torque 2D/3D, Unity 3D отреагировали на это довольно оперативно, выпустив плагины для браузеров, которые позволили отображать графику прямо в окне последних. Сначала популярность завоевал визуализатор на основе технологии Flash, но по целому ряду причин эта технология все больше теряет свою долю на рынке. Поэтому сейчас для визуализации в вебе

часто используется библиотека для языка JavaScript – WebGL, которая позволяет создавать интерактивную 3D-графику. Однако, из-за недостатков языка, таких как отсутствие многопоточности, библиотека не может полноценно удовлетворить потребности игроделов. Ей на смену консорциумом W3C (куда входят: Microsoft, Google, Mozilla и др.) разрабатывается новый низкоуровневый бинарный компилируемый формат WebAssembly.

Под конец первого десятилетия 21-го века очень быстро развивались мобильные технологии. Как гром среди ясного неба появились мобильные устройства по мощности сопоставимые с ПК средней ценовой категории и способные запускать мощные игровые приложения со всеми спецэффектами, которыми обладали низкоуровневые графические интерфейсы. На что разработчики игровых движков ответили в некоторых случаях созданием специализированных конверторов, создающих нативный для конкретного оборудования код (как, например, Unity 3D), а в других – модернизировали свои продукты для кроссплатформенности (к примеру, Torque 2D, Cocos 2DX). Также, на рынке появились новые игроки, предлагающие кроссплатформенные движки для всего парка мобильных устройств, выполняющиеся со скоростью нативного кода. Примеры подобных средств: Corona SDK, Marmalade SDK, AGK (App Game Kit).

Также, возник целый ряд кроссплатформенных движков, позволяющих разработать игру при минимальном знании программирования. Примерами можно назвать Construct 2 и GameMaker Pro. Используя готовые решения и визуальные редакторы, можно быстро – иногда в течение нескольких часов – создавать простые игры. Это оказалось особенно распространенным на мобильном рынке, где распространение free2play модели и короткая игровая сессия сделали «простые» игры вполне успешным жанром.

Низкоуровневые программные интерфейсы: OpenGL, DirectX развиваются в соответствии с видеоадаптерами. Раз в 1 – 2 года появляются новые версии, которые поддерживают и дают прикладным программистам (разработчикам движков) реализовать всю функциональность железа. DirectX уже достиг 12-й версии. С другой стороны на смену OpenGL пришел Vulkan – новый кроссплатформенный графический api, разрабатываемый консорциумом Khronos Group, куда входят производители железа и софта.

Последний на текущий момент тренд игровой индустрии – виртуальная/дополненная реальность. Подавляющее большинство совре-

менных игровых движков уже обзавелись поддержкой данной технологии, среди них: Torque 3D, Unity 3D, Unreal Engine 4. Разработано и множество сторонних расширений, таких как Vuforia Unity Extension. Чтобы реализовать поддержку очков VR разработчикам движков надо не только добавить визуализацию на второй экран (для второго глаза) с отличным от первого содержимым (так как, первый и второй глаза могут видеть отличающиеся сцены), но и так же добавить поддержку управления с новых устройств ввода, которые различны для разных гарнитур VR и пока не стандартизированы.

1. Создание 2D игры

1.1. Создание простой аркадной игры

Прототип 1: Apple Picker
Сценарий игры Apple Picker

Игрок управляет тремя корзинами в нижней части экрана и может перемещать их мышью влево и вправо. Яблоня в случайном порядке раскачивается взад-вперед и сбрасывает яблоки, и игрок должен ловить их в корзины, не давая упасть на землю. За каждое пойманное яблоко игроку начисляются очки, но если хотя бы одно яблоко упадет на землю, все другие яблоки исчезают, и вместе с ними исчезает одна корзина. Когда игрок теряет все три корзины, игра завершается.

Игровые объекты GameObject в Apple Picker

A. Корзины: управляются игроком. Корзины могут двигаться влево и вправо, следуя за движениями мыши. Когда Корзина сталкивается с яблоком, яблоко считается пойманным, и игроку начисляются очки.

B. Яблоки: сбрасываются с Яблони и падают вниз. Если яблоко сталкивается с одной из трех Корзин, оно считается пойманным и исчезает с экрана (а игрок получает несколько очков). Если яблоко достигает нижней границы экрана, оно исчезает, и вместе с ним исчезают все яблоки, имеющиеся на экране. Также исчезает одна Корзина (самая верхняя). После этого Яблоня вновь начинает сбрасывать яблоки.

C. Яблоня: раскачивается влево и вправо и сбрасывает яблоки. Яблоки сбрасываются через регулярные интервалы, поэтому единственный элемент случайности – это движение влево-вправо.

Список действий игровых объектов в Apple Picker

В этом анализе не рассматривается возрастающая сложность уровней в оригинальной игре Kaboom. Вместо этого мы сосредоточимся на действиях, выполняемых каждым игровым объектом

Действия Корзины

Корзина выполняет следующие действия:

– перемещается влево и вправо, следуя за движениями мыши;

– если Корзина сталкивается с ялоком, это ялоко считается пойманным.

Действия яблока

Яблоко выполняет следующие действия:

– падает вниз;

– если достигает нижнего края экрана, завершает текущий раунд игры.

Действия яблони

Яблоня выполняет следующие действия:

– раскачивается по экрану влево и вправо;

– сбрасывает яблоки через каждые 0,5 секунды.

А теперь приступим к созданию Apple Picker.

Начало: художественные ресурсы

Как прототип, эта игра не нуждается в фантастической графике, она просто должна работать. Искусство, которое вы будете творить в процессе чтения книги, известно как искусство программирования, искусство подготовки почвы для фактического игрового искусства, создаваемого художниками. Цель этого искусства, как и почти всего другого в прототипе, – как можно быстрее перейти от идеи к действующему прототипу Хорошо, если вы уже владеете искусством программирования, хотя это совершенно необязательно.

Яблоня

1. В главном меню Unity выберите пункт GameObject->3D Object ->Cylinder. Этот игровой объект будет служить стволом яблони. Переименуйте Cylinder в Trunk, выбрав его в иерархии и щелкнув на поле с именем объекта в верхней части панели Inspector (Инспектор).

2. Теперь выберите пункт меню GameObject > 3D Object > Sphere (Игровой объект > 3D объект > Сфера). Переименуйте объект Sphere в Leaves и настройте его компонент Transform, как показано ниже:

Leaves (Sphere) P:[0, 0.5, 0] R:[0, 0, 0] S:[3, 2, 3].

Вместе объекты Leaves и Trunk должны (немного) напоминать дерево, но сейчас это два отдельных объекта. Вы должны создать пустой игровой объект, который будет играть роль их родителя и объединять в один объект.

3. Выберите пункт меню GameObject > Create Empty (Игровой объект > Создать пустой). В результате будет создан пустой игровой объект. Настройте его компонент Ttransform, как показано ниже:

GameObject (Empty) P:[0, 0, 0] R:[0, 0, 0] S:[1, 1, 1]

Пустой игровой объект включает только компонент Transform и может служить прекрасным контейнером для других игровых объектов.

4. В панели иерархия измените имя игрового объекта на AppleTree

5. Перетащите по отдельности игровые объекты Trunk и Leaves в AppleTree

Теперь, когда игровые объекты Trunk и Leaves подчинены AppleTree, если попробовать передвинуть, масштабировать или повернуть AppleTree, вместе с ним передвинутся, изменят масштаб или повернутся оба объекта – Trunk и Leaves. Попробуйте поэкспериментировать с параметрами компонента Transform объекта AppleTree.

6. Поиграв с настройками компонента Transform объекта AppleTree, установите их, как показано ниже:

AppleTree P:[0, 0, 0] R:[0, Q, 0] S:[2, 2, 2]

7. Добавьте компонент твердого тела Rigidbody в AppleTree, выделив его в иерархии и выбрав в главном меню Unity пункт Component > Physics > Rigidbody

8. В инспекторе для компонента Rigidbody объекта AppleTree снимите флажок Use Gravity. Если оставить его, яблоня будет падать вниз при проигрывании сцены.

Простые материалы для AppleTree

Давайте добавим немного красок в сцену.

1. Выберите пункт меню Assets > Create > Material (Ресурсы > Создать > Материал). В результате в панели Project (Проект) появится новый материал.

A. Переименуйте его в Mat_Wood.

B. Перетащите материал Mat_Wood на объект Trunk в сцене или в панели Hierarchy (Иерархия).

C. Выберите Mat_Wood в панели Project (Проект).

D. В инспекторе для Mat_Wood, в разделе Main Maps, выберите в поле Albedo коричневый цвет, какой вам понравится.

2. Проделайте то же самое и создайте материал с именем Mat_Leaves.

A. Перетащите материал Mat_Leaves на объект Leaves в сцене или в панели Hierarchy (Иерархия).

B. Выберите в поле Albedo для материала Mat_Leaves зеленый цвет, напоминающий цвет листьев.

3. Перетащите AppleTree из панели Hierarchy (Иерархия) в панель Project (Проект), чтобы создать шаблон.

4. Установите координаты, поворот и масштаб элемента Directional Light в иерархии, как показано ниже:

Directional Light P:[0, 20, 0] R:[50, -30, 0] S: [1, 1, 1]

5. Чтобы сместить AppleTree вверх, выберите объект AppleTree в иерархии и измените его координаты на P:[0, 10, 0].

Apple

После создания яблони AppleTree можно переходить к созданию шаблона яблок, которые будут падать с яблони.

1. Выберите пункт меню GameObject > 3D Object > Sphere (Игровой объект > 3D объект > Сфера). Переименуйте созданный объект в Apple и настройте его компонент Transform, как показано ниже:

Apple (Sphere) P:[0, 0, 0] R:[0, 0, 0] S:[1, 1, 1].

2. Создайте новый материал с именем Mat_Apple и выберите в поле Albedo красный цвет.

3. Перетащите Mat_Apple на Apple в иерархии.

Добавление физических характеристик яблока

1. Выберите Apple в панели Hierarchy (Иерархия). В главном меню Unity выберите пункт Component > Physics > Rigidbody (Компонент > Физика > Твердое тело).

2. Щелкните по кнопке Play (Играть), и Apple упадет вниз.

3. Щелкните по кнопке Play (Играть) еще раз, чтобы остановить воспроизведение, и Apple вернется в начальную позицию.

Добавление тега «Apple» в объект Apple

Позднее вам понадобится получить массив всех игровых объектов Apple на экране, а чтобы упростить эту задачу, снабдим их специальным тегом.

1. Выберите Apple в иерархии, щелкните по кнопке в инспекторе рядом с надписью Tag и в открывшемся списке выберите Add Tag.

2. Щелкните по значку +, чтобы добавить новый тег.

3. Введите в поле New Tag Name (Имя нового тега) текст Apple (C) и щелкните на кнопке Save (Сохранить). В списке Tags (Теги) появится элемент Apple (D).

4. Щелкните по объекту Apple в иерархии, чтобы вернуть отображение настроек этого объекта в инспекторе.

5. Щелкните по кнопке рядом с надписью Тад еще раз; теперь в списке вы увидите тег Apple. Выберите пункт Apple в списке тегов.

Преобразование Apple в шаблон

Выполните следующие шаги, чтобы преобразовать игровой объект Apple в шаблон:

1. Перетащите Apple из панели Hierarchy (Иерархия) в панель Project (Проект), чтобы создать шаблон1.

2. После создания шаблона Apple в панели Project (Проект) щелкните по экземпляру Apple в панели Hierarchy (Иерархия) и удалите его (выбрав пункт Delete (Удалить) в контекстном меню – щелчком правой кнопки мыши – или нажав комбинацию клавиш Command-Delete (просто Delete в Windows) на клавиатуре).

Basket

Искусство программирования корзины, как и других объектов, не содержит ничего сложного.

1. Выберите в главном меню Unity пункт GameObject > 3D Object > Cube (Игровой объект > 3D объект > Куб). Переименуйте в Basket и настройте его компонент Ttransform, как показано ниже:

Basket (Cube) P:[0, 0, 0] R:[0, 0, 0] S:[4, 1, 4].

2. Создайте новый материал с именем Mat_Basket, окрасьте его в ненасыщенный желтый цвет (например, соломенный) и примените материал к корзине.

3. Добавьте в Basket компонент Rigidbody. Выделите Basket в иерархии и выберите в главном меню Unity пункт Component > Physics > Rigidbody (Компонент > Физика > Твердое тело).

A. В инспекторе для компонента Rigidbody объекта Basket снимите флажок Use Gravity.

B. Там же установите флажок Is Kinematic.

4. Перетащите Basket из панели Hierarchy (Иерархия) в панель Project (Проект), чтобы создать шаблон, и удалите оставшийся экземпляр Basket из иерархии (как мы проделали это с экземпляром Apple).

5. Сохраните сцену.

Настройка камеры

Одним из важнейших аспектов любой игры является выбор правильного местоположения камеры. В Apple Picker нам нужно, чтобы камера отображала игровое поле приличного размера. Поскольку игра по сути является двумерной, также нужно настроить камеру, чтобы она давала ортографическую проекцию вместо перспективной.

Настройка камеры для Apple Picker

Теперь настроим камеру для прототипа Apple Picker:

1. Выберите Main Camera в панели Hierarchy (Иерархия) и настройте компонент Ttransform, как показано ниже:

Main Camera (Camera) P:[0, Q, -10] R:[0, 0, 0] S:[1, 1, 1].

2. В инспекторе для компонента Camera установите следующие значения:

A. В поле Projection выберите значение Orthographic.

B. В поле Size установите значение 16.

В результате AppleTree получит оптимальные размеры в панели Game (Игра), и останется достаточно места для падения яблок, которые игрок должен ловить в корзины.

Настройка панели Game (Игра)

Еще одним важным фактором игры является соотношение сторон панели Game (Игра).

1. Вверху панели Game (Игра) имеется раскрывающийся список, в данный момент отображающий Free Aspect. Это список выбора соотношения сторон.

2. Щелкните на список и выберите пункт 16:9. Это стандартное соотношение сторон широкоформатных телевизионных экранов и компьютерных мониторов, поэтому данный выбор даст оптимальное изображение, если запустить игру в полноэкранном режиме.

Программирование прототипа Apple Picker

Теперь займемся реализацией программного кода, который будет приводить в движение прототип этой игры. Дважды щелкните по сценарию AppleTree в панели Project (Проект), чтобы открыть его.

Нам понадобятся некоторые переменные для хранения настроек; чтобы добавить их, откройте класс AppleTree в MonoDevelop и введите следующий код:

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;
public class AppleTree : MonoBehaviour {
 [Header("Set in Inspector")]
// Шаблон для создания яблок
 public GameObject applePrefab;

// Скорость движения яблони
public float speed = If;

// Расстояние, на котором должно изменяться направление движения яблони public float leftAndRightEdge = 10f;
public float chanceToChangeDirections = 0.1f;
// Частота создания экземпляров яблок public float secondsBetweenAppleDrops = If;
void Start () {
// Сбрасывать яблоки раз в секунду }
void Update () {
// Простое перемещение // Изменение направления }
}
```

Сохраните сценарий AppleTree в MonoDevelop и вернитесь в Unity.

2. Чтобы этот код действительно что-то делал, его нужно подключить к игровому объекту AppleTree.

A. Перетащите сценарий AppleTree из панели Project (Проект) на шаблон AppleTгее в той же панели Project (Проект).

B. Щелкните по экземпляру AppleTree в панели Hierarchy (Иерархия); в результате сценарий добавится не только в шаблон AppleTree, но и во все его экземпляры.

C. Выберите AppleTree в иерархии; в инспекторе, в разделе с компонентом AppleTree (Script), должны появиться все только что объявленные переменные.

3. Попробуйте переместить AppleTree в сцене, изменяя координаты X и Y в компоненте Transform в инспекторе, чтобы определить наилучшее местоположение AppleTree по высоте (position.y) и уточнить левую и правую границы для перемещения.

A. Установите координаты AppleTree в P:[0, 12, 0].

В. В переменную leftAndRightEdge компонента AppleTree (Script) в инспекторе введите значение 20.

Простое перемещение

Теперь добавим перемещение яблони.

1. Добавьте в метод Update() сценария AppleTree строки из следующего листинга.

```
public class AppleTree : MonoBehaviour {
void Update () {
// Простое перемещение
Vector3 pos = transform.position;
pos.x += speed * Time.deltaTime;
transform.position = pos;
// Изменение направления
```

2. Сохраните сценарий, вернитесь в Unity и щелкните по кнопке Play (Играть). Теперь вы должны увидеть, как яблоня AppleTree очень медленно перемещается по экрану.

Изменение направления

Теперь, когда яблоня AppleTree перемещается с приличной скоростью, она быстро достигнет края экрана и исчезнет за ним. Давайте организуем изменение направления движения, когда яблоня отдаляется от центра на расстояние leftAndRightEdge. Измените сценарий AppleTree, как показано ниже:

```
public class AppleTree : MonoBehaviour {
void Update () { // Простое перемещение
// Изменение направления
If (pos.x<-leftAndRightEdge)
Speed=Math.Abs(speed)
} else if ( pos.x > leftAndRightEdge ) {
// c speed = -Mathf.Abs(speed); // Начать движение влево }
}
}
```

Сохраните сценарий, вернитесь в Unity и щелкните по кнопке Play (Играть), чтобы увидеть, что из этого получилось.

Случайное изменение направления с заданной вероятностью

Чтобы добавить случайное изменение направления перемещения яблони, выполните следующие шаги.

1. Добавьте в сценарий строки:

```
public class AppleTree : MonoBehaviour {

void Update () {
// Простое перемещение
// Изменение направления
if ( pos.x < -leftAndRightEdge )
{ speed = Mathf.Abs(speed); /
/ Начать движение вправо }
else if ( pos.x > leftAndRightEdge )
{ speed = -Mathf.Abs(speed);
// Начать движение влево }
else if ( Random.value < chanceToChangeDirections )
{ //a speed *= -1; // Change direction  }
}
}
```

2. Щелкнув по кнопке Play (Играть), вы увидите, что для значения по умолчанию 0,1 в переменной chanceToChangeDirections направление меняется слишком часто. Измените в инспекторе значение chanceToChangeDirections на 0,02, и теперь ситуация должна выглядеть намного лучше.

В каждом кадре есть вероятность 2 %, что AppleTree изменит направление движения. На быстром компьютере эта вероятность может проверяться до 400 раз в секунду, тогда как на медленном компьютере вероятность будет проверяться лишь 30 раз в секунду.

3. Чтобы исправить этот недостаток, перенесем код, управляющий сменой направления из Update() (который вызывается со скоростью отображения кадров) в FixedUpdate().

```
}
public class AppleTree : MonoBehaviour {
void Update () {
// Простое перемещение

// Изменение направления
if ( pos.x < -leftAndRightEdge ) {
speed = Mathf.Abs(speed); // Начать движение вправо }
else if ( pos.x > leftAndRightEdge )
{ speed = -Mathf.Abs(speed); // Начать движение влево } // a
```

```
void FixedUpdate() {
// Теперь случайная смена направления привязана ко времени,

// потому что выполняется в FixedUpdateQ
if ( Random.value < chanceToChangeDirections ) { // b
speed *= -1; // Change direction } }
}
```

Теперь AppleTree будет случайно менять направление в среднем 1 раз в секунду (50 вызовов FixedUpdate в секунду х вероятность 0,02 = в среднем 1 раз в секунду).

Сбрасывание яблок

Пришел черед сбрасывать яблоки.

1. Выберите AppleTree в иерархии и посмотрите на параметры компонента AppleTree(Script) в инспекторе. Полют applePrefab нужно присвоить шаблон Apple из панели Project (Проект).

2. Вернитесь в MonoDevelop и добавьте код из следующего листинга в класс AppleTree:

```
public class AppleTree : MonoBehaviour {
void Start () {
 // Сбрасывать яблоки раз в секунду
Invoke( "DropApple", 2f ); //a }
void DropApple() {
// b GameObject apple = Instantiate<GameObject>( applePrefab ); //
c apple.transform.position = transform.position; // d
Invoke( "DropApple", secondsBetweenAppleDrops ); // e 1
void Update (){...} // f
}
```

3. Сохраните сценарий AppleTree, вернитесь в Unity, щелкните по кнопке Play (Играть) и посмотрите, что получится.

4. В компоненте Rigidbody игрового объекта AppleTree установите флажок Is Kinematic в инспекторе.

5. Снова щелкните по кнопке Play (Играть), и вы увидите, что проблема с яблоней исчезла, но с яблоками осталась.

Настройка физических слоев

Сначала создадим несколько новых физических слоев.

1. Щелкните по AppleTree в иерархии и в раскрывающемся списке Layer, в инспекторе, выберите пункт Add Layer.

2. Дайте слою 8 имя AppleTree, слою 9 – имя Apple и слою 10 – имя Basket.

3. В главном меню Unity выберите пункт Edit > Project Settings > Physics (Правка > Параметры проекта > Физика).

4. Экземпляры Apple должны обнаруживать столкновения с объектами Basket, но не с объектом AppleTree и другими экземплярами Apple.

5. После настройки матрицы Layer Collision Matrix можно заняться распределением важных игровых объектов по физическим слоям.

A. Щелкните по шаблону Apple в панели Project (Проект) и выберите пункт Apple в раскрывающемся списке Layer, в верхней части панели Inspector (Инспектор).

B. Щелкните по шаблону Basket в панели Project (Проект) и выберите пункт Basket в списке Layer.

C. Щелкните по шаблону AppleTree в панели Project (Проект) и выберите пункт AppleTree в списке Layer.

Если теперь щелкнуть по кнопке Play (Играть), яблоки должны просто падать с яблони, как задумывалось.

Остановка падения яблок ниже заданного уровня

Если запустить игру и оставить ее воспроизводиться в течение некоторого времени, можно заметить появление большого количества яблок в иерархии. Причина в том, что каждую секунду код создает новое яблоко, но не удаляет упавшие яблоки.

1. Откройте сценарий Apple и добавьте код, уничтожающий яблоки, когда они упадут ниже уровня transform, position .y == -20 (то есть скроются за нижним краем экрана):

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;
public class Apple : MonoBehaviour {
public static float bottomY = -20f;
void Update(){
if ( transform.position.y < bottomY ) {
 Destroy( this.gameObject ); // b }
}
}
```

2. Сохраните сценарий Apple.

3. Подключите сценарий Apple к шаблону Apple в панели Project (Проект), чтобы задействовать этот код в игре.

A. Выберите Apple в панели Project (Проект).

B. Прокрутите содержимое в панели Inspector (Инспектор) до самого низа и щелкните по кнопке Add Component (Добавить компонент).

C. В открывшемся меню выберите Scripts > Apple (Сценарии > Apple).

Если теперь щелкнуть по кнопке Play (Играть) в Unity и уменьшить масштаб сцены, можно увидеть, как яблоки падают вниз, достигают уровня Y, равного -20, и исчезают.

Это все, что имеет отношение к яблокам.

Создание экземпляров корзин

1. Подключите сценарий ApplePicker к главной камере Main Camera в иерархии.

2. Откройте сценарий ApplePicker в MonoDevelop, введите следующий код и сохраните его:

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;
public class ApplePicker : MonoBehaviour {
[Header("Set in Inspector")]
public GameObject basketPrefab;
public int numBaskets = 3;
public float basketBottomY = -14f;
public float basketSpacingY = 2f;
void Start () { f
or (int i=0; i<numBaskets; i++) {
GameObject tBasketGO = Instantiate<GameObject>( basketPrefab );
Vector3 pos = Vector3.zero;
pos.y = basketBottomY + ( basketSpacingY * i );
tBasketGO.transform.position = pos; } }
}
```

3. В Unity щелкните по Main Camera в панели Hierarchy (Иерархия) и в инспекторе выберите для поля basketPrefab шаблон Basket из панели Project (Проект).

Перемещение корзин мышью

Далее нам нужно написать код, перемещающий каждую корзину (экземпляр Basket) вслед за указателем мыши.

1. Подключите сценарий Basket к шаблону Basket в панели Project (Проект).

2. Откройте сценарий Basket в MonoDevelop, введите следующий код и сохраните его:

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

public class Basket : MonoBehaviour {
void Update () {
 // Получить текущие координаты указателя мыши на экране из Input
Vector3 mousePos2D = Input.mouseposition; // a
// Координата Z камеры определяет, как далеко в трехмерном пространстве
 // находится указатель мыши
mousePos2D.z = -Camera.main.transform.position.z; // b
// Преобразовать точку на двумерной плоскости экрана в трехмерные
// координаты игры
Vector3 mousePos3D = Camera.main.ScreenToWorldPoint( mousePos2D ); // c
// Переместить корзину вдоль оси X в координату X указателя мыши Vector3 pos = this.transform.position; pos.x = mousePos3D.x;
 this.transform.position = pos;
}
}
```

Теперь, после щелчка по кнопке Play (Играть) в Unity корзины будут перемещаться вслед за указателем мыши, и вы сможете ловить ими яблоки, хотя на самом деле яблоки пока не ловятся.

Ловля яблок

Перейдем к реализации ловли яблок:

Добавьте в сценарий Basket строки из следующего листинга:

```
public class Basket : MonoBehaviour {

void Update () { ... }

void OnCollisionEnter( Collision coll ) { //a
// Отыскать яблоко, попавшее в эту корзину
GameObject collidedWith = coll.gameObject; // b
if ( collidedWith.tag == "Apple" ) { // c
Destroy( collidedWith ); } }
}
```

2. Сохраните сценарий Basket, вернитесь в Unity и щелкните по кнопке Play (Играть).

Теперь игра действует очень похоже на классическую игру Kaboom!

ГИП и управление игрой

Последнее, что мы добавим в нашу игру, – графический интерфейс пользователя (ГИП) и управление игрой, которые сделают этот прототип более похожим на настоящую игру. Далее мы добавим один элемент ГИП – счетчик очков и элементы управления – уровни и жизни.

Счетчик очков

Счетчик очков дает игроку возможность видеть его достижения в игре.

1. Откройте сцену _Scene_0, дважды щелкнув на ней в панели Project (Проект).

2. В меню Unity выберите пункт GameObject > UI > Text (Игровой объект > ПИ > Текст).

3. Дважды щелкните по Canvas в иерархии, чтобы уменьшить масштаб и увидеть весь объект целиком. В списке дочерних объектов Canvas вы увидите игровой объект Text.

4. Выберите игровой объект Text в иерархии и в панели Inspector (Инспектор) измените его имя на HighScore.

5. Выполните следующие инструкции, чтобы настроить объект HighScore в инспекторе:

6. Щелкните правой кнопкой мыши по HighScore в иерархии и в контекстном меню выберите пункт Duplicate (Дублировать)1.

7. Выделите новый игровой объект HighScore (1) и измените его имя на ScoreCounter.

8. Измените настройки компонентов RectTransform и Text (Script) игрового объекта ScoreCounter в инспекторе. Не забудьте изменить значения Anchors и Pivot в компоненте RectTransform, а также значение Alignment в компоненте Text.

Добавление очков за пойманное яблоко

1. Откройте сценарий Basket в MonoDevelop и добавьте строки из следующего листинга:

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;
using UnityEngine.UI; //
Эта строка подключает библиотеку для работы с ГИП // a

public class Basket : MonoBehaviour {
 [Header("Set Dynamically")]
public Text scoreGT; // a

void Start() {
ScoreCounter GameObject scoreGO = GameObject.Find("ScoreCounter"); // b
// Получить компонент Text этого игрового объекта
scoreGT = scoreGO.GetComponent<Text>(); // c
// Установить начальное число очков равным 0 scoreGT.text = "0"; }
void Update () { ... }
void OnCollisionEnter( Collision coll ) {
// Отыскать яблоко, попавшее в эту корзину
GameObject collidedWith = coll.gameObject;
if ( collidedWith.tag == "Apple” ) {
 Destroy( collidedWith );
// Преобразовать текст в scoreGT в целое число
 int score = int.Parse( scoreGT.text ); // d
// Добавить очки за пойманное яблоко
score += 100;
// Преобразовать число очков обратно в строку и вывести ее на экран scoreGT.text = score.ToString();
```

Уведомление сценария ApplePicker, что яблоко не было поймано.

Другой аспект, делающий прототип Apple Picker более похожим на игру, – завершение раунда и удаление корзины, если яблоко не было поймано. В данный момент яблоки уничтожаются автоматически, и это здорово, но они должны уведомлять сценарий ApplePicker об этом, чтобы можно было завершить раунд и уничтожить остальные яблоки. Для этого необходимо, чтобы один сценарий вызывал функцию в другом сценарии.

1. Для начала в MonoDevelop внесите следующие изменения в сценарий Apple:

```
public class Apple : MonoBehaviour {
[Header("Set in Inspector'')]
public static float bottomY = -20f;
void Update () {
if ( transform.position.y < bottomY ) { //a
Destroy( this.gameObject );
// Получить ссылку на компонент ApplePicker главной камеры Main Camera ApplePicker apScript = Camera.main.GetComponent<ApplePicker>();// b
// Вызвать общедоступный метод AppleDestroyed() из apScript apScript.AppleDestroyed(); // c
}
}
}
```

2. Нам нужно добавить общедоступный метод AppleDestroyed() в сценарий ApplePicker, поэтому откройте этот сценарий в MonoDevelop и добавьте следующие строки:

```
void Start () { ... }
public void AppleDestroyed() { //a
// Удалить все упавшие яблоки
GameObject[] tAppleArray=GameObject.FindGameObjectsWithTag("Apple"); // b foreach ( GameObject tGO in tAppleArray ) {
Destroy( tGO ); } }
}
```

Сохраните ВСЕ сценарии в MonoDevelop. После определения метода AppleDest- royed() снова появится возможность запустить игру в Unity.

Уничтожение корзины после потери яблока

Последний фрагмент кода, который мы добавим в эту сцену, удаляет одну корзину при потере яблока и останавливает игру после удаления последней корзины. Добавьте следующие изменения в сценарий ApplePicker:

```
using System.Collections;
 using System.Collections.Generic;
using UnityEngine;
using UnityEngine.SceneManagement;

public class ApplePicker : MonoBehaviour {
 [Header("Set in Inspector")]
Public GameObject basketPrefab;
 public int numBaskets = 3;
public float basketBottomY = -14f;
public float basketSpacingY = 2f;
public List<GameObject> basketList;

void Start () {
 basketList = new List<GameObject>(); // c
for (int i=0; icnumBaskets; i++) {
GameObject tBasketGO = Instantiate<GameObject>( basketPrefab );
Vector3 pos = Vector3.zero;
pos.y = basketBottomY + ( basketSpacingY * i );
 tBasketGO.transform.position = pos;
basketList.Add( tBasketGO ); // d
}
}
public void AppleDestroyed() {
// Удалить все упавшие яблоки
GameObject[] tAppleArray=GameObject.FindGameObjectsWithTag("Apple"); foreach ( GameObject tGO in tAppleArray ) {
Destroy( tGO ); }

// Удалить одну корзину // e
// Получить индекс последней корзины в basketList
int basketindex = basketList.Count-1;
// Получить ссылку на этот игровой объект
```

```
Basket GameObject tBasketGO = basketList[basketindex];
// Исключить корзину из списка и удалить сам игровой объект basketList.RemoveAt( basketindex ); Destroy( tBasketGO );
}
}
```

Добавление высшего достижения

Теперь задействуем игровой объект HighScore, созданный ранее.

1. Создайте новый сценарий на C# с именем HighScore и подключите его к игровому объекту HighScore в панели Hierarchy (Иерархия).

2. Откройте сценарий HighScore в MonoDevelop и введите следующий код:

```
using System.Collections;
using System.Collections.Generic; u
sing UnityEngine; using UnityEngine.UI;
// Напомню, эта библиотека нужна для работы с ГИП
public class HighScore : MonoBehaviour {
static public int score = 1000; // a
void Update () { // b
Text gt = this.GetComponent<Text>();
 gt.text = ''High Score: "+score; }
}
```

3. Откройте сценарий Basket и добавьте следующие строки, выделенные жирным, которые используют статическую переменную score:

```
public class Basket : MonoBehaviour {

void OnCollisionEnter( Collision coll ) {

if ( collidedWith.tag == "Apple" ) {

// Преобразовать число очков обратно в строку и вывести ее на экран scoreGT.text = score. ToStringQ;
// Запомнить высшее достижение
 if (score > HighScore.score) {
 HighScore.score = score; }
}
```

}
}

Теперь переменная HighScore.score будет обновляться всякий раз, когда количество очков, заработанное пользователем, превысит ее текущее значение.

4. Откройте сценарий ApplePicker и добавьте следующие строки, возвращающие игру в исходное состояние, когда игрок потеряет все корзины. Этот код предотвращает появление исключения IndexOutOf Range, упомянутого выше.

public class ApplePicker : MonoBehaviour {
public void AppleDestroyed() {
// Исключить корзину из списка и удалить сам игровой объект basketList.RemoveAt(basketindex); Destroy(tBasketGO);
// Если корзин не осталось^ перезапустить игру
if (basketList.Count == 0) {
SceneManager.LoadScene("_Scene_0”); //a }
}
}

5. К данному моменту мы изменили несколько сценариев. Не забыли сохранить их? Если еще не сохранили – или не помните, как это часто случается со мной, – выберите в меню MonoDevelop пункт File > Save АП (Файл > Сохранить все), чтобы сохранить все изменившиеся сценарии. Если пункт Save АН (Сохранить все) отображается серым цветом, как неактивный, тогда все в порядке – все ваши сценарии уже сохранены.

Теперь у вас есть прототип игры, очень похожий на классическую игру Kaboom! компании Activision. В нашей игре не хватает, например, последовательного увеличения сложности, начального и конечного экранов, но вы можете добавить все это самостоятельно, когда у вас появится больше опыта.

1.2. Создание игры, учитывающей законы физики

Прототип 2: Mission Demolition

Игры, учитывающие законы физики, являются одними из наиболее популярных, и именно эта черта обеспечивает широкую известность таким играм, как Angry Birds.

Идея прототипа игры

В этой игре игрок будет стрелять из рогатки по замку, стараясь разрушить его. В каждом замке имеется особая зона, попав в которую игрок переходит на следующий уровень.

Вот как выглядит желаемая последовательность событий:

1. Когда игрок наводит указатель мыши на рогатку, она должна подсвечиваться.

2. Если игрок нажимает левую кнопку мыши (в Unity это кнопка с номером 0), когда рогатка подсвечена, в области указателя мыши должен создаваться снаряд.

3. При перемещении указателя мыши с нажатой левой кнопкой снаряд должен следовать за указателем, но оставаясь в пределах сферического коллайдера рогатки.

4. От каждого рога рогатки к снаряду должна тянуться белая линия, чтобы рогатка в игре выглядела как настоящая.

5. Когда игрок отпускает левую кнопку мыши (с номером 0), снаряд должен запускаться из рогатки.

6. Цель игрока – разрушить замок, находящийся в нескольких метрах, и попасть в особую область внутри него.

7. Игроку дается право выполнить три выстрела, чтобы поразить цель. Каждый сделанный выстрел оставляет след на экране, чтобы игрок мог сориентироваться и скорректировать следующий выстрел.

Художественные ресурсы

Прежде чем приступить к программному коду, нужно создать несколько ресурсов художественного оформления.

Земля

Выполните следующие шаги, чтобы создать землю.

1. Откройте сцену _Scene_0. Проверьте, что содержимое сцены _Scene_0 видно в панели Hierarchy (Иерархия): в списке должны присутствовать главная камера Main Camera и источник направленного освещения Directional Light

2. Создайте куб (выберите в меню пункт GameObject > 3D Object > Cube (Игровой объект > 3D объект > Куб)). Переименуйте куб в Ground. Чтобы превратить куб в сплошной прямоугольник, вытянутый во всю ширину сцены вдоль оси X, установите параметры компонента Transform, как показано ниже:

Ground (Cube) P:[0, -10, 0] R:[0, 0, 0] S:[100, 1, 4].

3. Создайте новый материал (Assets > Create > Material (Ресурсы > Создать > Материал)) и дайте ему имя Mat_Ground.

A. Выберите в поле Albedo для материала Mat_Ground коричневый цвет.

B. Установите ползунок Smoothness в позицию 0 (земля не должна блестеть).

C. Подключите Mat_Ground к игровому объекту Ground в иерархии (в предыдущей главе рассказывалось, как это сделать).

4. Сохраните сцену.

Направленное освещение

В последних версиях Unity источник направленного освещения Directional Light добавляется в сцену по умолчанию, но мы должны правильно настроить его для данного проекта.

1. Выберите Directional Light в панели Hierarchy (Иерархия). Переместите источник в сторону, установив параметры его компонента Transform, как показано ниже:

Directional Light P:[-10, 0, 0] R:[50, -30, 0] S:[1, 1, 1].

2. Сохраните сцену.

Настройка камеры

Шаги для настройки камеры:

1. Выберите главную камеру Main Camera в иерархии и переименуйте ее в _MainCamera.

2. Установите следующие параметры компонента Transform камеры _MainCamera (обязательно установите координату Y в разделе Position равной 0):

_MainCamera P:[0, 0, -10] R:[0, 0, 0] S:[1, 1, 1].

3. Установите следующие параметры компонента Camera камеры _MainCamera:

A. В раскрывающемся списке Clear Flags выберите пункт Solid Color.

B. В поле Background выберите цвет, близкий к небесно-голубому

C. В поле Projection выберите значение Orthographic.

D. В поле Size введите число 10.

5. Сохраните сцену. Никогда не забывайте сохранять свои сцены.

Рогатка

Теперь создадим простую рогатку из трех цилиндров:

1. Создайте пустой игровой объект (GameObject > Create Empty (Игровой объект > Создать пустой)). Измените его имя на Slingshot и установите параметры его компонента Transform, как показано ниже:

Slingshot (Empty) P:[0, 0, 0] R: [0, 0, 0] S:[1, 1, 1].

2. Создайте новый цилиндр (GameObject > 3D Object > Cylinder (Игровой объект > 3D объект > Цилиндр)) и измените его имя на Base.

Base (Cylinder) P: [0, 1, 0] R: [0, 0, 0] S: [0.5, 1, 0.5]

3. Не снимая выделения с объекта Base, щелкните по ярлыку с изображением шестеренки рядом с компонентом Capsule Collider в инспекторе и в открывшемся меню выберите пункт Remove Component

4. Создайте новый материал с именем Mat_Slingshot, выберите в поле Albedo светло-желтый цвет и установите ползунок Smoothness в позицию 0. Перетащите материал Mat_Slingshot на объект Base, чтобы применить к нему материал.

5. Выберите объект Base в панели Hierarchy (Иерархия) и создайте его копию, нажав комбинацию Command-D на клавиатуре (Ctrl+D в Windows) или выбрав в главном меню пункт Edit > Duplicate (Правка > Дублировать).

6. Измените имя новой копии с Base (1) на LeftArm:

LeftArm (Cylinder) P:[0, 3, 1] R:[45, 0, 0] S:[0.5, 1.414, 0.5]

Этот объект будет служить левым рогом рогатки.

7. Выберите объект LeftArm в иерархии и создайте его копию. Переименуйте этот экземпляр в RightArm. Установите параметры компонента Transform объекта RightArm, как показано ниже:

RightArm (Cylinder) P:[0, 3, -1] R:[-45, 0, 0] S:[0.5, 1.414, 0.5].

8. Выберите объект Slingshot в иерархии. Добавьте в него компонент сферического коллайдера (Component > Physics > Sphere Collider (Компонент > Физика > Сферический коллайдер)). Установите параметры компонента Sphere Collider (Is Trigger = true, Center = [0, 4, 0], Radius = 3).

Slingshot (Empty) P:[-10, -10, 0] R:[0, -15, 0] S:[1, 1, 1]

10. Добавьте точку выстрела рогатки, чтобы определить место, откуда будут вылетать снаряды. Щелкните правой кнопкой по Slingshot в иерархии и выберите в контекстном меню пункт Create Empty (Создать пустой), чтобы создать новый, пустой игровой объ-

ект, вложенный в Slingshot. Переименуйте этот игровой объект в LaunchPoint. Установите параметры компонента Transform объекта LaunchPoint, как показано ниже:

LaunchPoint (Empty) P:[0, 4, 0] R:[0, 15, 0] S:[1, 1, 1].

11. Сохраните сцену.

Снаряды

Теперь перейдем к снарядам.

1. Создайте сферу в сцене (GameObject > 3D Object > Sphere (Игровой объект > 3D объект > Сфера)) и дайте ей имя Projectile.

2. Выберите игровой объект Projectile в иерархии и присоедините к нему компонент Rigidbody (Component > Physics > Rigidbody (Компонент > Физика > Твердое тело)).

А. В разделе Rigidbody в инспекторе введите число 5 в поле Mass.

3. Создайте новый материал с именем Mat_Projectile. Выберите в поле Albedo для Mat_Projectile темно-серый цвет. Установите ползунок Metallic в значение 0,5 и ползунок Smoothness в значение 0,65, чтобы придать снаряду вид металлического шарика. Примените материал Mat_Projectile к игровому объекту Projectile в иерархии.

4. Перетащите Projectile из панели Hierarchy (Иерархия) в панель Project (Проект), чтобы создать шаблон. Удалите экземпляр Projectile, оставшийся в панели Hierarchy (Иерархия).

5. Сохраните сцену.

Программный код прототипа

Теперь, когда все художественные ресурсы на месте, можно начинать добавлять в проект программный код.

Создание класса Slingshot

Выполните следующие шаги, чтобы создать класс Slingshot.

1. Создайте новый сценарий на C# с именем Slingshot (Assets > Create > C# Script (Ресурсы > Создать > Сценарий C#)). Подключите его к игровому объекту Slingshot в иерархии и откройте в MonoDevelop. Введите следующий код и удалите все лишние строки, которые добавляются по умолчанию:

```
using UnityEngine;
using System.Collections;
```

```
public class Slingshot : MonoBehaviour {

void OnMouseEnter()
{ print("Slingshot:OnMouseEnter()");
void OnMouseExit() {
print("Slingshot:OnMouseExit()"); }
}
```

2. Сохраните сценарий Slingshot в MonoDevelop и вернитесь в Unity.

3. Щелкните по кнопке Play (Играть) и подвигайте указателем мыши в пределах коллайдера Sphere Collider игрового объекта Slingshot в панели Game (Игра).

Демонстрация активного состояния рогатки

Далее добавим подсветку, чтобы игрок мог видеть, когда рогатка находится в активном состоянии.

1. Выберите LaunchPoint в иерархии. Добавьте в LaunchPoint компонент Halo (Component > Effects > Halo (Компонент > Эффекты > Гало)), который создаст эффект светящегося ореола вокруг LaunchPoint. Введите число 3 в поле Size эффекта гало и выберите светло-серый цвет в поле Color, чтобы сделать ореол видимым.

2. Теперь добавьте в сценарий Slingshot следующий код. Кстати, сейчас самый удобный момент закомментировать инструкции print(), добавленные для тестирования на предыдущем шаге:

```
public class Slingshot : MonoBehaviour {
 public GameObject launchPoint;

void Awake() {
Transform launchPointTrans = transform.Find("LaunchPoint"); // a
launchPoint = launchPointTrans.gameObject;
 launchPoint.SetActive( false ); // b }

void OnMouseEnter() {
 //print("Slingshot:OnMouseEnter()");
 launchPoint.SetActive( true ); // b }

void OnMouseExit() { /
/print("Slingshot:OnMouseExit()");
launchPoint.SetActive( false ); // b }
```

```
}
```

Создание снаряда

Следующий шаг – создание снаряда, когда пользователь нажмет кнопку мыши с номером 0.

1. Добавьте в Slingshot следующий код:

```
public class Slingshot : MonoBehaviour {
// поля, устанавливаемые в инспекторе Unity
[Header(”Set in Inspector")] // a
public GameObject prefabProjectile;

// поля, устанавливаемые динамически
 [Header("Set Dynamically")]
public GameObject launchPoint;
public Vector3 launchPos; П b
 public GameObject projectile; // b
 public bool aimingMode; H b

void Awake() {
Transform launchPointTrans transform.FindChild("LaunchPoint");
launchPoint.SetActive( false ); launchPos = launchPoint-
Trans.position;
}
// c
void OnMouseEnter() { . .. } // Не изменяйте OnMouseEnter()

void OnMouseExit() { .. . } //Не изменяйте OnMouseExit()

void OnMouseDown() { // d
/ Игрок нажал кнопку мыши, когда указатель находился над ро-
гаткой
aimingMode = true;
// Создать снаряд
 projectile = Instantiate( prefabprojectile ) as GameObject;
// Поместить в точку launchPoint
projectile.transform.position = launchPos;
// Сделать его кинематическим
projectile.GetComponent<Rigidbody>().isKinematic = true; }
}
```

2. Сохраните сценарий и вернитесь в Unity. Выберите Slingshot в панели Hierarchy (Иерархия) и в поле prefabProjectile выберите шаблон Projectile из панели Project (Проект), щелкнув по пиктограмме с изображением мишени, справа от prefabProjectile, или перетащив шаблон Projectile из панели Project (Проект) на поле prefabProjectile в инспекторе.

3. Щелкните по кнопке Play (Играть), наведите указатель мыши на область рогатки и щелкните по ней – на экране появится экземпляр Projectile.

4. Не будем останавливаться на достигнутом и продолжим. Добавьте в класс Slingshot следующее поле и метод Update():

```
public class Slingshot : MonoBehaviour {
// поля, устанавливаемые в инспекторе Unity
[Header("Set in Inspector”)]
public GameObject prefabProjectile;
public float velocityMult = 8f; // a

// поля, устанавливаемые динамически
[Header("Set Dynamically”)]

public bool aimingMode;

private Rigidbody projectileRigidbody;

void Awake() { ... }

void OnMouseDown() {
}
// Сделать его кинематическим
projectileRigidbody = projectile.GetComponent<Rigidbody>();
projectileRigidbody.isKinematic = true;

void Update() {
 // Если рогатка не в режиме прицеливания, не выполнять этот код
 if (’aimingMode) return;
Vector3 mousePos2D = Input.mousePosition;
mousePos2D.z = -Camera.main.transform.position.z;
```

```
Vector3 mousePos3D = Camera.main.ScreenToWorldPoint( mouse-
Pos2D );
// Найти разность координат между launchPos и mousePos3D
Vector3 mouseDelta = mousePos3D-launchPos;
// Ограничить mouseDelta радиусом коллайдера объекта
Slingshot // d

float maxMagnitude = this.GetComponent<SphereCollider>().radius;
if (mouseDelta.magnitude > maxMagnitude)
{ mouseDelta.Normalize(); mouseDelta *= maxMagnitude; }
// Передвинуть снаряд в новую позицию
Vector3 projPos = launchPos + mouseDelta;
projectile.transform.position = projPos;
 if ( Input.GetMouseButtonUp(0) ) {
// Кнопка мыши отпущена
aimingMode = false; projectileRigidbody.isKinematic = false;
projectileRigidbody.velocity = -mouseDelta * velocityMult; projec-
tile = null; }
}
```

5. Щелкните по кнопке Play (Играть) и посмотрите, как действует рогатка.

6. Сохраните сцену.

Слежение за полетом снаряда

Наша следующая задача – заставить главную камеру _MainCamera сопровождать полет снаряда, но решить ее немного сложнее, чем кажется.

Выполните следующие шаги.

1. Создайте новый сценарий на C# (Assets > Create > C# Script (Ресурсы > Создать > Сценарий C#)) с именем FollowCam.

2. Перетащите сценарий FollowCam на главную камеру _MainCamera в панели Hierarchy (Иерархия), чтобы сделать его компонентом камеры.

3. Щелкните дважды по сценарию FollowCam, чтобы открыть его, и введите следующий код:

```
using UnityEngine;
using System.Collections;
```

```
public class FollowCam : MonoBehaviour {
 static public GameObject POI; // Ссылка на интересующий объект
// a

[Header("Set Dynamically")]
 public float camZ; // Желаемая координата Z камеры

void Awake() {
camZ = this.transform.position.z; }

void FixedUpdate () {
// Однострочная версия if не требует фигурных скобок
 if (POI == null) return; // выйти, если нет интересующего объекта
// b

// Получить позицию интересующего объекта
Vector3 destination = POI.transform.position;
// Принудительно установить значение destination.z равным
camZ, чтобы // отодвинуть камеру подальше
destination.z = camZ;
// Поместить камеру в позицию destination
 transform.position = destination;
}
}
```

Сразу бросаются в глаза следующие проблемы.

A. Если достаточно уменьшить масштаб, можно заметить, что снаряд улетает за край земли.

B. Если выстрелить в землю, снаряд не отскочит от нее и не прекратит полет, когда попадет в нее. Если приостановить игру сразу после выстрела, выбрать снаряд в иерархии (чтобы выделить его и вывести маркеры ориентации) и затем возобновить игру, вы увидите, что снаряд катится по земле без остановки.

C. После выстрела камера сразу же подпрыгивает на уровень снаряда, что создает неприятное ощущение.

D. Когда снаряд окажется на определенной высоте – или пересечет край земли, – вы будете видеть только небо, и вам будет сложно понять, насколько высоко летит снаряд.

Выполните следующие шаги, чтобы исправить описанные проблемы друг за другом (расположены в порядке от простых в исправлении к сложным).

Сначала устраним проблему A, определив параметры компонента Transform объекта Ground, как: P:[100, -10,0] R:[0,0,0] S:[400,1,4]. Благодаря этому прямоугольник, представляющий землю, будет простираться далеко вправо.

Чтобы устранить проблему B, нужно добавить ограничения в компонент Rigidbody и физические свойства материала шаблона снаряда Projectile.

1. Выберите шаблон Projectile в панели Project (Проект).

2. В компоненте Rigidbody выберите в раскрывающемся списке способов определения столкновений Collision Detection значение Continuous, соответствующее непрерывному определению. Чтобы получить дополнительную информацию о способах определения столкновений, щелкните по кнопке вызова справки в правом верхнем углу раздела компонента Rigidbody.

3. Также в компоненте Rigidbody объекта Projectile.

A. Щелкните на пиктограмме с изображением треугольника рядом с названием подраздела Constraints, чтобы раскрыть его.

B. Установите флажок Freeze Position Z.

C. Установите флажки Freeze Rotation X, Y и Z.

Флажок Freeze Position Z предотвратит приближение снаряда к камере или удаление от нее (удержит снаряд на том же расстоянии от камеры, на каком находятся также земля и замок, который мы добавим далее). Флажки Freeze Rotation X, Y и Z предотвратят вращение снаряда.

4. Сохраните сцену, щелкните по кнопке Play (Играть) и попробуйте выстрелить еще раз.

5. В меню Unity выберите пункт Assets > Create > Physic Material (Ресурсы > Создать > Физический материал).

6. Дайте этому физическому материалу имя PMat_Projectile.

7. Щелкните по PMat_Projectile и в инспекторе введите в поле bounciness число 1.

8. В панели Project (Проект) перетащите PMat_Projectile на шаблон Projectile, чтобы применить его к Projectile.SphereCollider.

9. Сохраните сцену, щелкните по кнопке Play (Играть) и попробуйте выстрелить снова.

Выбрав Projectile, вы должны увидеть в инспекторе, что PMat_Projecti1e назначен как материал коллайдера Sphere Collider. Теперь после выстрела снаряд подскакивает и останавливается, а не просто скользит по поверхности земли.

Проблема С имеет два решения: замедление реакции камеры посредством интерполяции и добавление ограничения на ее местоположение. Применим оба.

1. Чтобы замедлить реакцию камеры, добавьте в FollowCam следующие строки:

```
public class FollowCam : MonoBehaviour {
 static public GameObject POI;
// The static point of interest

public float easing = 0.05f;

[Header("Set Dynamically")]

void FixedUpdate () {
// Однострочная версия if не требует фигурных скобок if (POI == null) return; // выйти, если нет интересующего объекта
// Получить позицию интересующего объекта V
ector3 destination = POI.transform.position;
// Определить точку между текущим местоположением камеры и destination
destination = Vector3.Lerp(transform.position, destination, easing);
// Принудительно установить значение destination.z равным camZ, чтобы // отодвинуть камеру подальше
destination.z = camZ;
// Поместить камеру в позицию destination transform.position = destination;
}
}
```

2. Добавьте ограничения на позицию камеры, включив в FollowCam следующие строки:

```
public class FollowCam : MonoBehaviour {

[Header("Set in Inspector")]
public float easing = 0.05f;
public Vector2 minXY = Vector2.zero;
```

```
[Header("Set Dynamically")]

void FixedUpdate () {
 // Однострочная версия if не требует фигурных скобок
 if (POI == null) return; // выйти, если нет интересующего объекта

// Ограничить X и Y минимальными значениями
destinations = Mathf.Max( minXY.x, destinations );
destination.y = Mathf.Max( minXY.y, destination.y );
// Определить точку между текущим местоположением камеры и destination
}
}
```

Чтобы устранить проблему D, требуется динамически корректировать размер проекции orthographicSize камеры.

1. Добавьте в сценарий FollowCam следующие строки,:

```
public class FollowCam : MonoBehaviour {
void FixedUpdate () {
// Поместить камеру в позицию destination
transform.position = destination;
// Изменить размер orthographicSize камеры., чтобы земля
// оставалась в поле зрения Camera.main.orthographicSize = destination.y + 10;
}
}
```

2. Щелкните дважды по объекту Ground в иерархии, чтобы уменьшить масштаб в панели Scene (Сцена) и отобразить в ней объект Ground целиком.

3. Выберите _MainCamera, щелкните по кнопке Play (Играть) и произведите выстрел вертикально вверх. В панели Scene (Сцена) вы увидите, как постепенно расширяется поле зрения камеры с увеличением высоты полета снаряда.

4. Сохраните сцену.

Строительство замка

В игре Mission Demolition игрок должен что-то разрушать, поэтому построим замок, который будет служить этой цели.

1. Отрегулируйте панель Scene (Сцена) так, чтобы видеть сцену сзади в изометрической проекции, щелкнув на стрелке напротив оси Z, на значке с изображениями осей.

2. Кроме того, сейчас самое время избавиться от скайбокса (Skybox – изображение неба) в панели Scene (Сцена).

3. Щелкните дважды по _MainCamera в иерархии, чтобы отрегулировать масштаб панели Scene (Сцена), удобный для строительства замка.

Изготовление стен и перекрытий

Начнем с создания шаблонов игровых объектов, из которых будет строиться замок.

1. Создайте дубликат материала Mat_Cloud и дайте ему имя Mat_Stone1.

A. Выберите Mat_Cloud в панели Project (Проект).

B. Выберите в меню Unity пункт Edit > Duplicate (Правка > Дублировать).

C. Измените имя Mat_Cloud 1 на Mat_Stone.

D. Выделите Mat_Stone и в поле Main Color выберите 25 % серый цвет (RGBA: [64, 64, 64, 255]).

2. Создайте новый куб (GameObject > 3D Object > Cube (Игровой объект > 3D объект > Куб)) и переименуйте его в Wall.

A. Настройте компонент Transform объекта Wall: P:[0,0,0] R:[0,0, 0] S:[1, 4,4].

B. Добавьте в объект Wall компонент Rigidbody (Component > Physics > Rigidbody (Компонент > Физика > Твердое тело)).

C. Ограничьте координату Z объекта Wall, установив флажок FreezePosition Z в настройках компонента Rigidbody.

D. Ограничьте поворот, установив флажки FreezeRotation X и У в настройках компонента Rigidbody.

E. Введите в поле Rigidbody.mass число 4.

F. Перетащите материал Mat_Stone на объект Wall, чтобы окрасить его в серый цвет.

3. Создайте в папке__Scripts новый сценарий с именем RigidbodySleep и введите следующий код:

```
using UnityEngine;
public class RigidbodySleep : MonoBehaviour {
void Start () {
```

```
Rigidbody rb = GetComponent<Rigidbody>();
 if (rb '= null) rb.Sleep(); } }
```

4. Подключите сценарий RigidbodySleep к объекту Wall.

5. Перетащите Wall в панель Project (Проект), чтобы создать шаблон (поместите его в папку _Prefabs), и после этого удалите экземпляр Wall из панели Hierarchy (Иерархия).

6. Выберите шаблон Wall в папке -Prefabs в панели Project (Проект) и создайте его копию.

A. Переименуйте копию Wall 1 в Slab.

B. Выберите Slab в папке -Prefabs и настройте масштаб в компоненте Transform: S:[4, 0.5, 4].

Строительство замка из стен и перекрытий

Теперь построим замок из стен и перекрытий.

1. Создайте пустой игровой объект, который будет служить корневым узлом замка (GameObject > Create Empty (Игровой объект > Создать пустой)).

A. Дайте ему имя Castle.

B. Настройте компонент Transform: P:[0, -9.5, 0] R:[0, 0, 0] S:[1, 1, 1]. Эти настройки поместят объект в удобное для строительства место с нижней границей точно на объекте Ground.

2. Перетащите Wall на объект Castle, чтобы сделать его дочерним по отношении к нему.

3. Создайте три копии Wall и определите для них следующие местоположения:

Wall P:[-6, 2, 0] Wall (1) P:[-2, 2, 0] Wall (2) P:[2, 2, 0] Wall (3) P:[6, 2, 0].

4. Перетащите Slab из папки _Prefabs в панели Project (Проект) в иерархию на объект Castle, чтобы сделать их дочерними по отношению к Castle.

5. Создайте две копии Slab и определите для них следующие местоположения:

Slab P:[-4, 4.25, 0] Slab (1) P:[0, 4.25, 0] Slab (2) P:[4, 4.25, 0].

6. Чтобы построить второй этаж замка, выберите мышью три соседних стены Wall и два перекрытия Slab над стенами. Создайте их копии (Command-D или Ctrl+D) и, удерживая нажатой клавишу Command (Ctrl на PC), перетащите их вверх, сформировав второй

этаж1. Вам потребуется вручную настроить их местоположения; окончательные координаты новых стен и перекрытий должны быть следующими:

Wall (4) P:[-4, 6.5, 0] Wall (5) P:[0, 6.5, 0] Wall (6) P:[4, 6.5, 0] Slab (3) P:[-2, 8.75, 0] Slab (4) P:[2, 8.75, 0].

7. Повторите прием с копированием, чтобы построить третий и четвертый этажи, добавив еще три вертикальных стены и одно горизонтальное перекрытие:

Wall (7) P:[-2, 11, 0] Wall (8) P:[2, 11, 0] Slab (5) P:[0, 13.25, 0] Wall (9) P:[0, 15.5, 0].

Одно из основных преимуществ строительства замка из шаблонов, как в этом примере, состоит в том, что вы легко сможете изменить внешний вид всех перекрытий Slab сразу, изменив шаблон Slab.

8. Выберите шаблон Slab в панели Project (Проект) и введите в поле transform, scale.x число 3.5. Изменение должно отразиться на всех экземплярах Slab, присутствующих в замке.

Создание области прицеливания

Последним игровым объектом, который мы должны добавить в замок, является область прицеливания, которую должен поразить игрок.

1. Создайте куб с именем Goal.

A. Сделайте его дочерним по отношению к объекту Castle.

B. Настройте компонент Transform объекта Goal: P:[0, 2, 0] R:[0, 0, 0] S:[3, 4, 4].

C. Установите флажок BoxCollider.isTrigger в настройках объекта Goal в инспекторе.

D. Перетащите Goal в папку -Prefabs в панели Project (Проект), чтобы создать шаблон.

2. Создайте в папке -Materials новый материал с именем Mat_Goal.

A. Перетащите Mat.Goal на шаблон Goal в папке -Prefabs в панели Project (Проект).

B. Выделите Mat_Goal в панели Project (Проект) и выберите шейдер Legacy Shaders > Transparent > Diffuse.

C. В поле Main Color для материала Mat_Goal выберите ярко-зеленый цвет с непрозрачностью 25 % (в формате RGBA в цветовой палитре Unity он определяется как [0, 255, 0, 64]).

Тестирование замка

Для тестирования замка выполните следующие шаги:

1. Настройте компонент Transform объекта Castle: P:[50, -9.5, 0] и затем щелкните по кнопке Play (Играть). Возможно, вам придется перезапустить игру несколько раз, но вы должны поразить замок снарядом.

2. Сохраните сцену.

Возврат для другого выстрела

Теперь, когда у нас есть замок для разрушения, пришло время добавить дополнительную игровую логику. После приземления снаряда камера должна вернуть фокус на рогатку.

1. Прежде чем сделать что-то еще, добавьте тег Projectile в шаблон снаряда.

A. Выберите шаблон Projectile в панели Project (Проект).

B. В инспекторе выберите пункт Add Tag (Добавить тег) в раскрывающемся списке Тад. В результате в инспекторе откроется диспетчер тегов и слоев Tags & Layers.

C. Щелкните по значку + справа внизу в пустом списке тегов Tags.

D. Введите в поле New Tag Name (Имя нового тега) текст Projectile и щелкните по кнопке Save (Сохранить).

E. Снова выберите шаблон Projectile в панели Project (Проект).

F. Присвойте ему тег Projectile, выбрав пункт Projectile в раскрывающемся списке Тад в инспекторе.

2. Откройте сценарий FollowCam в MonoDevelop и измените следующие строки:

```
public class FollowCam : MonoBehaviour {
void FixedUpdate () {
//-- // Однострочная версия if не требует фигурных скобок // a
//-- if (POI == null) return; // выйти, если нет интересующего объекта
 //-//-- // Получить позицию интересующего объекта
 //-- Vector3 destination = POI.transform.position;
Vector3 destination;
// Если нет интересующего объекта, вернуть P:[ 0, 0, 0 ]
if (POI == null) {
destination = Vector3.zero; }
```

```
else {
// Получить позицию интересующего объекта
destination = POI.transform.position;
// Если интересующий объект - снаряд, убедиться, что он остановился
if (POI.tag == "Projectile") {
//Если он стоит на месте (то есть не двигается)
if ( POI.GetComponent<Rigidbody>().IsSleepingQ ) {
// Вернуть исходные настройки поля зрения камеры
POI = null;
//в следующем кадре
 return; } } }
 // Ограничить X и Y минимальными значениями
destination.x = Mathf.Max( minXY.x, destination.x );
}
}
```

3. Скорректируйте значение Sleep Threshold в диспетчере физики PhysicsManager.

A. Откройте диспетчер физики PhysicsManager (Edit > Project Settings > Physics (Правка > Параметры проекта > Физика)).

B. Измените значение 0,005 в поле Sleep Threshold на 0,02.

4. Сохраните сцену. Если теперь попробовать сыграть в игру, камера будет возвращаться в исходную позицию, и вы сможете произвести новый выстрел.

Добавление следа, остающегося за снарядом

В Unity уже есть встроенный эффект визуализации следа Trail Renderer, но он нам не подойдет, потому что нам требуется больший контроль, чем может дать встроенный эффект. Поэтому будем использовать Line Renderer, на котором основан Trail Renderer.

1. Сначала создайте пустой игровой объект (GameObject > Create Empty (Игровой объект > Создать пустой)) с именем ProjectileLine.

A. Добавьте в него компонент Line Renderer (Component > Effects > Line Renderer (Компонент > Эффекты > Визуализатор линии)).

B. В настройках ProjectileLine в инспекторе распахните раздел Materials. Настройте компонент Line Renderer.

2. Создайте сценарий на C# (Assets > Create > C# Script (Ресурсы > Создать > Сценарий C#)) в папке__Scripts. Дайте ему имя

ProjectileLine и подключите к игровому объекту ProjectileLine. Откройте сценарий ProjectileLine в MonoDevelop и введите следующий код:

```
using System.Collections; using System.Collections.Generic; using UnityEngine;
public class ProjectileLine : MonoBehaviour {

[Header("Set in Inspector")]
 public float minDist = 0.1f;
private LineRenderer line; _
private GameObject poi;
private List<Vector3> points;

void Awake() {
 S = this;
// Установить ссылку на объект-одиночку
// Получить ссылку на LineRenderer
line = GetComponent<LineRenderer>();
// Выключить LineRenderer, пока он не понадобится
line.enabled = false;
// Инициализировать список точек
 points = new List<Vector3>(); }

// Это свойство (то есть метод, маскирующийся под поле)
public GameObject poi {
 get { return( _poi ); }
set { _poi = value;
if ( _Poi •'= nuH ) {
// Если поле _poi содержит действительную ссылку,
// сбросить все остальные параметры в исходное состояние
 line.enabled = false;
points = new List<Vector3>();
AddPoint(); } } }

// Этот метод можно вызвать непосредственно, чтобы стереть линию
public void Clear() {
_poi = null;
 line.enabled = false;
```

```
 points = new List<Vector3>(); }

public void AddPoint() {
// Вызывается для добавления точки в линии
Vector3 pt = _poi.transform.position;
if ( points.Count > 0 && (pt - lastPoint).magnitude < minDist ) {
// Если точка недостаточно далека от предыдущей, просто выйти
return; }
if ( points.Count == 0 ) {
 // Если это точка запуска...
Vector3 launchPosDiff = pt - Slingshot.LAUNCH_POS;
// Для определения // ...добавить дополнительный фрагмент линии, // чтобы помочь лучше прицелиться в будущем
points.Add( pt + launchPosDiff );
// Установить первые две точки
line.SetPosition(0, points[0] );
line.SetPosition(l, points[l] );
// Включить LineRenderer line.enabled = true; }
else {
// Обычная последовательность добавления точки
points.Add( pt );
line.positioncount = points.Count;
line.SetPosition( points.Count-1, lastPoint );
 line.enabled = true; }
}

// Возвращает местоположение последней добавленной точки
 public Vector3 lastPoint {
get {
 if (points == null) {
// Если точек нет, вернуть Vector3.zero
 return( Vector3.zero ); }
 return( points[points.Count-l] ); } }

 void FixedUpdate () {
if ( poi == null ) {
// Если свойство poi содержит пустое значение, найти интересующий
 // объект
```

```
if (FollowCam.POI != null) {
 if (FollowCam.POI.tag == "Projectile") {
poi = FollowCam.POI; }
else {
return;
// Выйти, если интересующий объект не найден
} } else { return;
// Выйти, если интересующий объект не найден } }
// Если интересующий объект найден,
// попытаться добавить точку с его координатами в каждом FixedUpdate
AddPoint();
if ( FollowCam.POI == null ) {
 // Если FollowCam.POI содержит null, записать nulll в poi poi = null; } }
}
```

3. В сценарий Slingshot нужно также добавить свойство LAUNCH_POS, чтобы позволить методу AddPoint () ссылаться на начальную launchPoint:

```
public class Slingshot MonoBehaviour {
static private Slingshot S;
 [Header(”Set in Inspector")]
private Rigidbody projectileRigidbody;

static public Vector3 LAUNCH_POS {
get {
 if (S == null) return Vector3.zero;
return S.launchPos; } }

void Awake() {
S = this;
Transform launchPointTrans = transform.FindChild("LaunchPoint");
}
}
```

А. Это скрытый статический экземпляр Slingshot, который будет играть

Сохраните сцену.

Поражение области прицеливания

Область прицеливания в замке должна реагировать на попадание в нее снаряда.

1. Создайте новый сценарий на C# с именем Goal и подключите его к шаблону Goal в папке _Prefabs в панели Project (Проект). Затем введите в сценарий Goal следующий код:

```
using UnityEngine; using System.Collections;
public class Goal : MonoBehaviour {
// Статическое поле, доступное любому другому коду
static public bool goalMet = false;

void OnTriggerEnter( Collider other ) {
// Когда в область действия триггера попадает что-то,
// проверить, является ли это "что-то" снарядом
 if ( other.gameObject.tag == "Projectile" ) {
 // Если это снаряд, присвоить полю goalMet значение true
Goal.goalMet = true;
// Также изменить альфа-канал цвета, чтобы увеличить непрозрачность
Material mat = GetComponent<Renderer>().material;
Color c = mat.color;
 c.a = 1;
mat.color = c; } }
}
```

Если теперь поразить область прицеливания снарядом, она окрасится в ярко-зеленый цвет.

2. Сохраните сцену.

Добавление в сцену пользовательского интерфейса

Выполните следующие шаги, чтобы добавить пользовательский интерфейс в сцену.

1. Добавьте в сцену элемент пользовательского интерфейса UI Text (GameObject > UI > Text (Игровой объект > ПИ > Текст)) и дайте ему имя UIText_Level.

2. Создайте второй элемент UI Text с именем UIText_Shots.

3. Настройте оба элемента.

4. Создайте UI Button (GameObject > UI > Button (Игровой объект > ПИ > Кнопка)). Дайте кнопке имя UIButton_View.

5. Настройте компонент Rect Transform кнопки UIButton_View.

6. В иерархии щелкните по пиктограмме с изображением треугольника рядом с UIButton_View, выберите дочерний объект Text и настройте компонент Text (Script). Не меняйте ничего за пределами раздела Character. По завершении сохраните сцену.

Нам удалось создать игру, учитывающую законы физики подобно Angry Birds, которую вы можете самостоятельно продолжить, развивать и расширять.

1.3. Создание игры-шутера

Прототип 3: SPACE SHMUP

К жанру SHMUP (или «shoot 'em up» – игра-стрелялка, шутер) относятся такие игры, как классические Galaga и Galaxian из 1980-х, и современный шедевр Ikaruga.

Настройка сцены

Выполните следующие шаги, чтобы настроить сцену.

1. Выберите источник направленного света Directional Light в иерархии и настройте его компонент Transform:

P:[0, 20, 0] R:[50, -30, 0] S:[1, 1, 1]

2. Переименуйте главную камеру Main Camera в _MainCamera (о чем уже говорилось выше, во врезке с инструкциями по настройке проекта). Выберите главную камеру J4ainCamera и настройте ее компонент Transform:

P:[0, 0, -10] R: [0, 0, 0] S: [1, 1, 1]

3. В компоненте Camera главной камеры _MainCamera настройте следующие поля, а потом сохраните сцену:

• в поле Clear Flags выберите значение Solid Color;

• в поле Background выберите черный цвет (со значением 255 в альфа-канале; RGBA:[O, O, 0, 255]);

• в поле Projection выберите значение Orthographic;

• в поле Size введите число 40 (после настройки поля Projection);

• в поле Near Clipping Plane введите число 0.3;

• в поле Far Clipping Plane введите число 100.

4. Поскольку наша будущая игра – это шутер с вертикальной ориентацией игрового поля, для панели Game (Игра) нужно установить соотношение сторон с книжной ориентацией. Для этого в панели Game (Игра) щелкните в раскрывающемся списке выбора соотношения сторон, в котором в данный момент должен отображаться текст Free Aspect. Найдите внизу списка кнопку с символом +. Щелкните на ней, чтобы добавить новое предопределенное соотношение сторон.

Создание космического корабля игрока

Чтобы создать космический корабль игрока, выполните следующие шаги:

1. Создайте пустой игровой объект с именем _Hero (GameObject > Create Empty (Игровой объект > Создать пустой)). Настройте его компонент Transform: P:[O, O, O J R:[0, 0, 0] S:[1,1,1].

2. Создайте куб (GameObject > 3D Object > Cube (Игровой объект > 3D объект > Куб)) и перетащите его на объект _Hero, сделав его дочерним объектом по отношению к _Hero. Дайте кубу имя Wing и настройте его компонент Transform: P:[0, -1,0 1 R:[0, 0,45] S:[3, 3, 0.5].

3. Создайте пустой игровой объект с именем Cockpit и сделайте его дочерним по отношению к _Hero.

4. Создайте куб и сделайте его дочерним по отношению к Cockpit (для этого можно щелкнуть правой кнопкой мыши на Cockpit и выбрать в контекстном меню пункт 3D Object > Cube (3D объект > Куб)). Настройте его компонент Transform: P:[O, O, 0]R:[315, 0, 45]S:[1,1,1].

5. Снова выберите объект Cockpit и настройте его компонент Transform: P:[0, 0, 0] R:[0, 0,180] S:[1, 3,1]. В данном случае используется тот же трюк, с которым вы познакомились в главе 27 «Объектно-ориентированное мышление», позволяющий быстро создать корабль вытянутой формы.

6. Выберите объект _Hero в иерархии и щелкните по кнопке Add Component (Добавить компонент) в инспекторе. Выберите в открывшемся меню пункт New Script (Новый сценарий).

7. Добавьте в _Hero компонент Rigidbody, выбрав Hero в иерархии, и затем выберите в меню пункт Add Component > Physics > Rigidbody (Добавить компонент > Физика > Твердое тело), щелкнув на кнопке Add Component (Добавить компонент) в инспекторе. Настройте следующие поля в компоненте Rigidbody объекта _Hero:

• Снимите флажок Use Gravity;

• Установите флажок isKinematic;

• В разделе Constraints: установите флажки Freeze Position Z и Freeze Rotation X, YhZ.

Позднее мы еще вернемся к объекту _Hero, но пока этих настроек достаточно.

8. Сохраните сцену! Не забывайте сохранять сцену после любых изменений в ней.

Метод Update() в сценарии Него

Метод Update() в следующем листинге сначала читает из InputManager значения на горизонтальной и вертикальной осях, сохраняет полученные в диапазоне между -1 и 1 в переменные xAxis и yAxis. Вторая половина метода Update () выполняет перемещение корабля с учетом значения скорости speed и реального времени.

Откройте сценарий Hero в MonoDevelop и введите следующий код:

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;
public class Hero : MonoBehaviour
{ static public Hero S; // Одиночка

[Header("Set in Inspector")]
// Поля, управляющие движением корабля
public -Float speed = 30;
public float rollMult = -45;
public float pitchMult = 30;

[Header("Set Dynamically")]
public float shieldLevel = 1;

void Awake() {
if (S == null) {
S = this; // Сохранить ссылку на одиночку // a
} else {
Debug.LogError("Hero.Awake() - Attempted to assign second Hero.S!"); } }
```

```
void Update () {
// Извлечь информацию из класса Input
float xAxis = Input.GetAxis("Horizontal"); // b
float yAxis = Input.GetAxis("Vertical"); // b
// Изменить transform.position, опираясь на информацию по осям
 Vector3 pos = transform.position; pos.x += xAxis ♦ speed ♦ Time.deltaTime;
pos.y += yAxis ♦ speed * Time.deltaTime; transform.position = pos;

// Повернуть корабль, чтобы придать ощущение динамизма // c
transform.rotation = Quaternion.Euler(yAxis*pitchMult,xAxis*rollMult,0);
}
}
```

Попробуйте сыграть в игру и подвигать корабль клавишами со стрелками или буквами WASD, чтобы увидеть, как он себя ведет.

Защитное поле для космического корабля игрока

Защитное поле для _Hero формируется как комбинация прозрачного текстурированного квадрата и сферического коллайдера Sphere Collider (для обработки столкновений):

1. Создайте новый квадрат (GameObject > 3D Object > Quad (Игровой объект > 3D объект > Квадрат)) с именем Shield и сделайте его дочерним по отношению к _Hero. Настройте компонент Transform объекта Shield: P:[0, 0, 0], R:[0, 0, 0], S:[8,8,8].

2. Выберите объект Shield в иерархии и удалите из него компонент Mesh Collider, щелкнув в инспекторе по пиктограмме с шестеренкой справа от имени Mesh Collider и выбрав в открывшемся меню пункт Remove Component (Удалить компонент). Добавьте компонент Sphere Collider (Component > Physics > Sphere Collider (Компонент > Физика > Сферический коллайдер)).

3. Создайте новый материал (Assets > Create > Material (Ресурсы > Создать > Материал)) с именем Mat_Shield и поместите его в папку -Materials в панели Project (Проект). Перетащите Mat_Shield на объект Shield в _Hero в иерархии, чтобы связать его с объектом квадрата Shield.

4. Выберите Shield в иерархии, теперь материал Mat_Shield появится в инспектонастройками объекта Shield.

5. Щелкните по кнопке Select (Выбрать) в правом нижнем углу квадратика с изображением текстуры и выберите текстуру с именем Shield. Щелкните по цветовой шкале в поле Main Color и выберите ярко-зеленый цвет (RGBA:[0, 255,0, 255]). Затем введите значения в поля:

- 0,2 BTiling.x.
- 0,4 в Offset.x,
- В поле Tiling.y оставьте значение 1,0.
- В поле Offset.y оставьте значение 0.

6. Создайте новый сценарий на C# с именем Shield (Assets > Create > C# Script (Ресурсы > Создать > Сценарий C#)). Перетащите его в папку__Scripts в панели Project (Проект) и затем перетащите его же на объект Shield в иерархии, чтобы создать компонент сценария в игровом объекте Shield.

7. Откройте сценарий Shield в MonoDevelop и введите следующий код:

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

public class Shield : MonoBehaviour {
 [Header("Set in Inspector")]
 public float rotationsPerSecond = 0.1f;

[Header("Set Dynamically")]
public int levelShown = 0;

// Скрытые переменные, не появляющиеся в инспекторе
Material mat; // a

void Start() {
mat = GetComponent<Renderer>().material; // b }

 void Update () {
 // Прочитать текущую мощность защитного поля из объекта-одиночки Hero
 int currLevel = Mathf.FloorToInt( Hero.S.shieldLevel ); // c
 // Если она отличается от levelShown...
```

```
if (levelShown != currLevel) {
levelShown= currLevel
// Скорректировать смещение в текстуре, чтобы отобразить поле
с другой // мощностью
mat.mainTextureOffset = new Vector2( 0.2f♦levelShown, 0 ); // d
}
 // Поворачивать поле в каждом кадре с постоянной скоростью
float rZ = -(rotationsPerSecond*Time.time*360) % 360f; // e
transform.rotation = Quaternion.Euler( 0, 0, rZ );
}
}
```

Сохранение корабля _Hero на экране

Теперь движение корабля _Hero выглядит достаточно привлекательно, а вращающееся защитное поле создает красивый эффект, но в данный момент легко можно вывести корабль за границы экрана.

1. Выберите _Hero в иерархии и, щелкнув по кнопке Add Component (Добавить компонент) в инспекторе, выберите в открывшемся меню пункт New Script (Новый сценарий). Дайте сценарию имя BoundsCheck и щелкните по кнопке Create and Add (Создать и добавить). Перетащите сценарий BoundsCheck в папку__Scripts в панели Project (Проект).

2. Откройте сценарий BoundsCheck и добавьте в него следующий код:

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;
public class BoundsCheck : MonoBehaviour { //a
 [Header("Set in Inspector'*)]
public float radius = If;

[Header("Set Dynamically")]
public float camWidth;
public float camHeight;

void Awake() {
camHeight = Camera.main.orthographicSize;
camWidth = camHeight * Camera.main.aspect; }
```

```
void LateUpdate () { // d
Vector3 pos = transform.position;
if (pos.x > camWidth - radius) {
pos.x = camWidth - radius; }
if (pos.x < -camWidth + radius) {
pos.x = -camWidth + radius; }
if (pos.y > camHeight - radius) {
pos.y = camHeight - radius; }
if (pos.y < -camHeight + radius) {
pos.y = -camHeight * radius; }
transform.position = pos;
}
// Рисует границы в панели Scene (Сцена) с помощью OnDraw-
Gizmos()
void OnDrawGizmos () { // e
Gizmos,DrawWireCube(Vector3.zero, boundsize);
У
}
```

3. Щелкните на кнопке Play (Играть) и попробуйте поперемещать корабль по экрану.

Добавление вражеских кораблей

Графические изображения вражеских кораблей

Чтобы создать графический ресурс с изображением Enemy_0, выполните следующие шаги:

1. Создайте пустой игровой объект с именем Enemy _0 и настройте его компонент Transform: P:[-20, 10, 0], R:[0, 0, 0], S:[1, 1, 1]. Местоположение выбрано так, чтобы этот корабль не перекрывался кораблем игрока _Него, пока мы будем его конструировать.

2. Создайте сферу с именем Cockpit, сделайте ее дочерней по отношению к Enemy _0 и настройте ее компонент Transform: P:[0, 0, 0], R:[0, 0, 0], S:[2, 2, 1].

3. Создайте вторую сферу с именем Wing, сделайте ее дочерней по отношению к Enemy _0 и настройте ее компонент Transform: P:[0,0,0], R:[0,0,0], S:[5,5,0.5].

4. Следуя этому формату создайте четыре других вражеских корабля.

Enemy_1

Enemy_l (Empty) P:[-10, 10, 0] R:[0, 0j 0] S:[1, 1, 1]
Cockpit (Sphere) P:[0, 0, 0] R:[0j 0, 0] S:[2, 2, 1]
Wing (Sphere) P:[0, 0, 0] R:[0, 0, 0] S:[6, 4, 0.5]
Enemy_2
Enemy_2 (Empty) P:[0, 10, 0] R:[0, 0j 0] S:[1, 1, 1]
Cockpit (Sphere) P:[-1.5, 0, 0] R:[0, 0л 0) S:[1, 3, 1]
Reactor (Sphere) P:[2, 0, 0] R:[0, 0, 0] S:[2, 2, 1]
Wing (Sphere) P:[0, 0, 0] R:[0, 0, 0] S:[6, 4, 0.5]

Enemy_3
Enemy_3 (Empty) P:[10, 10, 0] R:[0, 0, 0] S:[1, 1, 1]
CockpitL (Sphere) P:[-1, 0, 0] R:[0, 0, 0] S:[1, 3, 1]
CockpitR (Sphere) P:[1, 0, 0] R:[0, 0, 0] S:[1, 3, 1]
Wing (Sphere) P:[0, 0.5, 0] R:[0, 0, 0] S:[5, 1, 0.5]

Enemy_4
Enemy_4 (Empty) P:[20, 10, 0] R:[0, 0, 0] S:[1, 1, 1 1
Cockpit (Sphere) P:[0, 1, 0] R:[0, 0, 0] S:[1.5, 1-5, 1-5]
Fuselage (Sphere) P:[0, 1, 0] R:[0, 0, 0] S:[2, 4, 1]
WingL (Sphere) P:[-1.5, 0, 0] R:[0, 0, -30] S:[5, 1, 0.5]
WingR (Sphere) P:[1.5, 0, 0] R:[0, 0j 30] S:[5, 1, 0.5]

5. Добавьте компонент Rigidbody во все игровые объекты вражеских кораблей (то есть Enemy_0, Enemy_l, Enemy_2, Enemy_3 и Enemy_4). Чтобы добавить Rigidbody, выполните следующие шаги.

A. Выделите Enemy_0 в иерархии и выберите в меню пункт Component > Physics > Rigidbody (Компонент > Физика > Твердое тело).

B. В настройках компонента Rigidbody снимите флажок Use Gravity.

C. Установите флажок isKinematic.

D. Щелкните на пиктограмме с треугольником рядом с именем раздела Constraints, установите флажки Freeze Position Z и Freeze Rotation X, Y и Z.

6. Скопируйте компонент Rigidbody из объекта Enemy_0 во все другие объекты вражеских кораблей. С этой целью для каждого вражеского корабля выполните следующие действия:

A. Выделите Enemy_0 в иерархии и щелкните на кнопке с шестеренкой в правом верхнем углу, в разделе с настройками компонента Rigidbody.

B. В открывшемся меню выберите пункт Copy Component (Копировать компонент).

C. Выберите в иерархии объект вражеского корабля, в который хотите добавить Rigitbody

D. Щелкните по кнопке с шестеренкой в правом верхнем углу, в разделе с настройками компонента Transform.

E. В открывшемся меню выберите пункт Paste Component As New (Вставить компонент как новый).

7. Перетащите каждый вражеский корабль в папку _Prefabs в панели Project (Проект), чтобы создать из них шаблоны.

8. Удалите все экземпляры вражеских кораблей из иерархии, кроме Enemy_0.

Сценарий Enemy

Чтобы создать сценарий Enemy, выполните следующие шаги:

1. Создайте новый сценарий на C# с именем Enemy и поместите его в папку__Scripts.

2. Выберите Enemy_0 в панели Project (Проект) – не в иерархии. В инспекторе щелкните по кнопке Add Component (Добавить компонент) и в открывшемся меню выберите пункт Scripts > Enemy (Сценарии > Enemy). Если после этого щелкнуть по объекту Enemy_0 в панели Project (Проект) или Hierarchy (Иерархия), вы должны увидеть подключенный компонент Enemy (Script).

3. Откройте сценарий в MonoDevelop и введите следующий код:

```
using System.Collections; // Необходимо для доступа к массивам и другим коллекциям //
using System.Collections.Generic; // Необходимо для доступа к спискам и словарям
using UnityEngine; // Необходимо для доступа к Unity

public class Enemy : MonoBehaviour {
[Header("Set in Inspector: Enemy")]
public float speed = 10f; // Скорость в м/с
public float fireRate = 0.3f; // Секунд между выстрелами (не используется)
public float health = 10;
public int score = 100; // Очки за уничтожение этого корабля
```

```
// Это свойство: метод, действующий как поле public Vector3 pos
{ // a
get
return( this.transform.position );
} set { this.transform.position = value; }
}

void Update()
{ Move(); }


public virtual void Move() { // b
Vector3 tempPos = pos;
 tempPos.y -= speed * Time.deltaTime;
pos = tempPos; }
}
```

4. В Unity щелкните по кнопке Play (Играть), и экземпляр Enemy_0 в сцене должен начать движение вниз.

5. Чтобы подключить сценарий BoundsCheck к экземпляру Enemy_0, выберите его в иерархии (не в панели Project (Проект)). В инспекторе щелкните по кнопке Add Component (Добавить компонент) и в открывшемся меню выберите пункт Scripts > BoundsCheck (Сценарии > BoundsCheck).

6. Примените изменения в экземпляре Enemy_0 к его шаблону, щелкнув на кнопке Apply (Применить) в верхней части инспектора. Теперь выберите шаблон Enemy_0 в панели Project (Проект), чтобы убедиться, что сценарий был подключен к нему

7. Выберите экземпляр EnemyGO в иерархии и в разделе BoundsCheck, в инспекторе, введите в поле radius число -2.5.

8. Щелкните на кнопке Play (Играть), и вы увидите, что экземпляр Enemy_0 остановился сразу, как только оказался за нижней границей экрана. Однако нам нужно, чтобы Enemy_O не оставался за границей экрана, на самом деле мы должны определить момент, когда он вышел из видимой области, и уничтожить его.

9. Для этого внесите в сценарий BoundsCheck следующие изменения.

```
/// <summary>
/// Предотвращает выход игрового объекта за границы экрана
```

```
///Важно: работает ТОЛЬКО с ортографической камерой Main Camera в [ 0, 0, 0 ].
/// </summary>
public class BoundsCheck : MonoBehaviour {
 [Header(”Set in Inspector”)]
public float radius = If;
public bool keepOnScreen = true; // a

[Header("Set
public bool Dynamically”)]
public float isOnScreen = true;
public float camWidth; camHeight;
// b
void Awake() { ... } // Напомню: многоточие означает, что метод не изменился.

void Latellpdate () {
Vector3 pos = transform.position;
isOnScreen = true;
if ( pos.x > camWidth - radius ) {
pos.x = camWidth - radius;
isOnScreen = false; }
if ( pos.x < -camWidth + radius ) {
pos.x = -camWidth + radius;
isOnScreen = false; }
if ( pos.y > camHeight - radius ) {
pos.y = camHeight - radius;
isOnScreen = false; }
if ( pos.y < -camHeight + radius ) {
pos.y = -camHeight + radius;
 isOnScreen = false; }
if ( keepOnScreen && !isOnScreen ) {
 transform.position = pos; isOnScreen = true; }
}
```

Удаление вражеского корабля после выхода за границы экрана

1. Снимите флажок keepOnScreen в компоненте BoundsCheck (Script) шаблона Enemy_0, находящегося в папке -Prefabs в панели Project (Проект).

2. Чтобы изменения достигли экземпляра Enemy_0 в иерархии, выберите его и щелкните по кнопке с шестеренкой справа от названия компонента BoundsCheck (Script) в инспекторе. В появившемся меню выберите пункт Revert to Prefab (Вернуть из шаблона), чтобы скопировать настройки из шаблона в экземпляр в иерархии.

3. Добавьте в сценарий Enemy следующий код:

```
public class Enemy : MonoBehaviour {
public int score = 100; // Очки за уничтожение этого корабля
private BoundsCheck bndCheck; И
void Awake() { // b
bndCheck = GetComponent<BoundsCheck>(); }

void Update() {
 Move();
if
}
( bndCheck != null && IbndCheck.isOnScreen ) { // c
 // Убедиться, что корабль вышел за нижнюю границу экрана
if ( pos.y < bndCheck.camHeight - bndCheck.radius ) { // d
// Корабль за нижней границей, поэтому его нужно уничтожить
Destroy( gameObject ); }
```

Теперь сценарий действует в точности как мы хотели, но кажется немного странным выполнять одно и то же сравнение pos.y с camHeight и radius здесь и в BoundsCheck.

4. Измените сценарий BoundsCheck, добавив следующие строки,:

```
public class BoundsCheck : MonoBehaviour {
public float camHeight;
[Hideininspector]
public bool offRight, offLeft, offUp, offDown;

void Start() { ... }

void LateUpdate () {
Vector3 pos = transform.position;
 isOnScreen = true;
offRight = offLeft = offUp = offDown = false;
if ( pos.x > camWidth - radius )
{ pos.x = camWidth - radius;
```

```
offRight = true;
} if ( pos.x < -camWidth + radius ) {
pos.x = -camWidth + radius
offLeft = true; // c
}
if ( pos.y > camHeight - radius ) {
pos.y = camHeight - radius;
offUp = true; // c }
if ( pos.y < -camHeight + radius ) {
pos.y = -camHeight + radius; offDown = true; / }
isOnScreen = !(offRight || offLeft || offUp || offDown);
if ( keepOnScreen && !isOnScreen ) {
 transform.position = pos;
isOnScreen = true;
offRight = offLeft = offUp = offDown = false;  }
}
}
```

5. Теперь внесите в сценарий Enemy следующие изменения, выделенные жирным, чтобы воспользоваться улучшениями в сценарии BoundsCheck:

```
public class Enemy : MonoBehaviour {

void Update() {
Move();
if ( bndCheck != null && bndCheck.offDown ) {
// Корабль за нижней границей, поэтому его нужно уничтожить
 Destroy( gameObject ); }
}
}
```

А. Теперь достаточно проверить только bndCheck. off Down, чтобы определить, что экземпляр Enemy вышел за нижний край экрана.

В. В этих двух строках убрано по одному отступу, потому что теперь проверка выполняется одной инструкцией if вместо двух.

Реализация класса Enemy заметно упростилась – сейчас он использует возможности компонента BoundsCheck и может делать свою работу, не дублируя его функциональность.

Случайное создание вражеских кораблей

Теперь, закончив работу над вражеским кораблем, можно реализовать случайное создание произвольного количества объектов Enemy_0.

Подключите сценарий BoundsCheck к главной камере _MainCamera и снимите в его настройках флажок keepOnScreen.

Создайте новый сценарий с именем Main и подключите его к _MainCamera.

using System.Collections; // Необходимо для доступа к массивам и другим

// коллекциям

using System.Collections.Generic; // Необходимо для доступа к спискам и словарям

using UnityEngine; // Необходимо для доступа к Unity

using UnityEngine.SceneManagement; // Для загрузки и перезагрузки сцен

public class Main : MonoBehaviour {

static public Main S; // Объект-одиночка Main

[Header(''Set in Inspector")]

public GameObject[] prefabEnemies; // I Массив шаблонов Enemy

public float enemySpawnPerSecond = 0.5f; // Вражеских кораблей в секунду

public float enemyDefaultPadding = 1.5f

//

Отступ для // позиционирования

private BoundsCheck bndCheck;

void Awake() {

S = this;

// Записать в bndCheck ссылку на компонент BoundsCheck этого игрового // объекта

bndCheck = GetComponent<BoundsCheck>();

// Вызывать SpawnEnemy() один раз (в 2 секунды при значениях по умолчанию)

Invoke("SpawnEnemy",

If(enemySpawnPerSecond); //a }

public void SpawnEnemy() {

// Выбрать случайный шаблон Enemy для создания

```
int ndx = Random.Range(0, prefabEnemies.Length); // b
GameObject go = Instantiate<GameObject>( prefabEnemies[ ndx ] );
// c
// Разместить вражеский корабль над экраном в случайной позиции x
float enemyPadding = enemyDefaultPadding; // d
if (go.GetComponent<BoundsCheck>() != null) { // e
enemyPadding = Mathf.Abs( go.GetComponent<BoundsCheck>().radius ); }

// Установить начальные координаты созданного вражеского корабля // f
Vector3 pos = Vector3.zero;
float xMin = -bndCheck.camWidth + enemyPadding;
float xMax = bndCheck.camWidth - enemyPadding;
pos.x = Random.Range( xMin, xMax );
pos.y = bndCheck.camHeight + enemyPadding; go.transform.position = pos;
// Снова вызвать SpawnEnemyO
Invoke( "SpawnEnemy", If/enemySpawnPerSecond ); // g
}
}
```

3. Сохраните сценарий, закончив ввод, вернитесь в Unity и выполните следующие шаги:

A. Удалите экземпляр Епету_0 из иерархии (не трогая при этом шаблон в панели Project (Проект)).

B. Выберите _MainCamera в иерархии.

C. Раскройте раздел prefabEnemies в компоненте Main (Script) объекта_MainCamera, щелкнув на пиктограмме с треугольником, и в разделе prefabEnemies введите 1 в поле Size.

D. Перетащите Enemy_0 из панели Project (Проект) в поле Element 0 в массиве prefabEnemies.

4. Запустите сцену.

Настройка тегов, слоев и физики

В этой игре имеется несколько видов игровых объектов, которые можно разместить в разных слоях и по-разному организовать взаимодействие между ними:

– Слой для корабля игрока (Hero): корабль _Hero должен сталкиваться с вражескими кораблями, вражескими снарядами и кубиками-бонусами, но не должен сталкиваться со своими снарядами;

– Слой для снарядов, выпускаемых кораблем игрока (ProjectileHero): снаряды, выпускаемые кораблем _Hero, должны сталкиваться только с вражескими кораблями;

– Слой для вражеских кораблей (Enemy): вражеские корабли должны сталкиваться только с _Hero и снарядами, выпускаемыми им, но не должны сталкиваться с кубиками-бонусами;

– Слой для снарядов, выпускаемых вражескими кораблями (ProjectileEnemy): снаряды, выпускаемые вражескими кораблями, должны сталкиваться только с .Hero;

– Слой для бонусов (PowerUp): бонусы должны сталкиваться только с .Hero.

Чтобы создать эти слои, а также некоторые теги, которые пригодятся нам позже, выполните следующие шаги:

1. Откройте диспетчер тегов и слоев Tags & Layers в панели Inspector (Инспектор), выбрав в меню пункт Edit > Project Settings > Tags and Layers (Правка > Параметры проекта > Теги и слои).

2. Раскройте раздел Tags со списком тегов. Щелкая на значке + в конце списка, введите имена тегов. Вот имена этих тегов: Hero, Enemy, ProjectileHero, ProjectileEnemy, PowerUp и PowerllpBox.

3. Раскройте раздел Layers. Начиная с восьмого пользовательского слоя User Layer 8, введите имена слоев. Имена слоев: Hero, Enemy, ProjectileHero, ProjectileEnemy и PowerUp.

Откройте диспетчер физики PhysicsManager (Edit > Project Settings > Physics (Правка > Параметры проекта > Физика)) и настройте матрицу столкновений слоев Layer Collision Matrix.

Повреждение корабля игрока вражескими кораблями

Теперь, когда вражеские корабли и корабль игрока находятся в слоях, которые могут сталкиваться, нужно заставить их реагировать на столкновения.

1. Щелкните по треугольнику рядом с именем _Hero в иерархии и выберите дочерний объект Shield. В инспекторе установите флажок Is Trigger в настройках коллайдера Sphere Collider, чтобы превратить его в триггер. Нам не нужно, чтобы другие объекты отскакивали от Shield; мы лишь хотим определять момент, когда происходит столкновение.

2. Добавьте следующий метод, выделенный жирным, в конец сценария Hero:

```
public class Hero : MonoBehaviour {

void Update() {
}
void OnTriggerEnter(Collider other) {
print("Triggered: ”+other.gameObject.name); }
}
```

3. Запустите сцену и попробуйте столкнуться с вражескими кораблями.

4. В сценарии Hero замените код в методе OnTriggerEnter () строками, выделенными жирным в следующем листинге:

```
public class Hero : MonoBehaviour {

void OnTriggerEnter(Collider other) {
Transform rootT = other.gameObject.transform.root;
GameObject go = rootT.gameObject;
 print("Triggered: "+go.name); }
}
```

5. Измените метод OnTriggerEnter() класса Hero, чтобы каждое столкновение с вражеским кораблем уменьшало уровень защиты корабля игрока на 1 и уничтожало вражеский корабль:

```
public class Hero : MonoBehaviour {
public float shieldLevel = 1;
private GameObject lastTriggerGo=null;
void OnTriggerEnter(Collider other) {
Transform rootT = other.gameObject.transform.root;
GameObject go = rootT.gameObject; //
print(”Triggered: ”+go.name);

// Гарантировать невозможность повторного столкновения с тем же объектом
if (go == lastTriggerGo) {
return; } lastTriggerGo = go;
if (go.tag == ”Enemy") {
// Если защитное поле столкнулось с вражеским
```

```
// кораблем
shieldLevel--;
// Уменьшить уровень защиты на 1
Destroy(go); // ...
и уничтожить врага И е
} else {
}
}
print("Triggered by non-Enemy: ”+go.name); И f
}
```

6. Запустите сцену и попробуйте столкнуться с несколькими вражескими кораблями. В какой-то момент, после достаточно большого числа столкновений, можно заметить странное поведение защитного поля. Оно полностью истощается и вновь начинает набирать мощность.

Чтобы исправить этот недостаток, преобразуем поле shieldLevel в свойство, защищающее и ограничивающее новое скрытое поле с именем _shieldLevel.

7. В классе Hero измените имя общедоступной переменной shieldLevel на _shieldLevel, замените спецификатор области видимости public на private и добавьте строку с атрибутом [SerializeField]:

```
public class Hero : MonoBehaviour {

[Header("Set Dynamically”)]
[SerializeField]
private float -ShieldLevel = 1; // Обратите внимание на символ
// подчеркивания
// Эта переменная хранит ссылку на последний столкнувшийся
игровой объект
}
```

8. Добавьте свойство shieldLevel в конец класса Hero:

```
public class Hero : MonoBehaviour {

void OnTriggerEnter(Collider other) {
}

public float shieldLevel {
 get {
```

```
return( _shieldLevel ); }
set {
-ShieldLevel = Mathf.Min( value, 4 );
if (value < 0) { // c Destroy(this.gameObject); }
}
}
}
```

Перезапуск игры

Поиграв в игру, можно заметить, что игра теряет смысл после уничтожения корабля игрока _Hero. Теперь мы изменим оба класса, Hero и Main, добавив вызов метода, который после разрушения _Hero будет ждать 2 секунды и перезапускать игру.

1. Добавьте поле gameRestartDelay в начало определения класса Hero:

```
public class Hero : MonoBehaviour {
static public Hero S; // Одиночка
[Header("Set in Inspector'')]

public float pitchMult = 30;
public float gameRestartDelay = 2f;
[Header("Set Dynamically'')]
}
```

2. Добавьте следующие строки в определение свойства shieldLevel в классе Hero:

```
public class Hero : MonoBehaviour {

public float shieldLevel {
get { ... }
set {
if (value < 0) {
Destroy(this.gameObject);
// Сообщить объекту Main.S о необходимости перезапустить игру
Main.S.DelayedRestart( gameRestartDelay ); //a
}
}
}
```

Добавьте следующие методы в класс Main, чтобы реализовать перезапуск.

```
public class Main : MonoBehaviour {

public void SpawnEnemy() { ... }

public void DelayedRestart( float delay ) {
// Вызвать метод Restart() через delay секунд
Invoke( "Restart", delay ); }



public void Restart() {
// Перезагрузить _Scene_0, чтобы перезапустить игру
 SceneManager.LoadScene( "_Scene_0" ); }
}
```

4. Щелкните поа кнопке Play (Играть), чтобы протестировать игру Теперь, после уничтожения корабля игрока, игра будет ждать пару секунд и затем перезапускаться перезагрузкой сцены.

Стрельба (наконец)

Теперь, когда вражеские корабли могут наносить повреждения кораблю игрока, пришло время дать игроку возможность защищаться.

ProjectileHero, снаряды для Hero

Выполните следующие шаги, чтобы создать снаряд для стрельбы по вражеским кораблям.

1. Создайте в иерархии куб с именем ProjectileHero и следующими настройками компонента Transform:

ProjectileHero (Cube) P:[10, 0, 0] R:[0, 0, 0] S:[0.25, 1, 0.5]

2. В раскрывающихся списках Tag и Layer для ProjectileHero выберите значение ProjectileHero.

3. Создайте новый материал с именем Mat_Projectile, поместите его в папку-Materials в панели Project (Проект), выберите для него шейдер ProtoTools > UnlitAlpha и присвойте игровому объекту ProjectileHero.

4. Добавьте в игровой объект ProjectileHero компонент Rigidbody со следующими настройками:

• Снимите флажок Use Gravity.

• Снимите флажок isKinematic.

• В поле Collision Detection выберите значение Continuous.

• В разделе Constraints установите флажки Freeze Position Z и Freeze Rotation X, Y и Z.

5. В компоненте Box Collider игрового объекта ProjectileHero установите значение Size.Z равным 10. Это гарантирует поражение снарядами объектов, которые могут находиться вне плоскости XY (то есть Z=0).

6. Создайте новый сценарий на C# с именем Projectile и подключите к ProjectileHero. Мы займемся им позже.

7. Сохраните сцену.

8. Подключите компонент сценария BoundsCheck к ProjectileHero. Снимите флажок keepOnScreen и введите -1 в поле radius. Величина radius в BoundsCheck не влияет на столкновения с другими игровыми объектами, она лишь определяет расстояние, на которое ProjectileHero должен выйти за край экрана, чтобы считаться покинувшим экран.

9. Преобразуйте ProjectileHero в шаблон, перетащив его из иерархии в папку -Prefabs в панели Project (Проект). Затем удалите экземпляр, оставшийся в иерархии.

10. Сохраните сцену

Реализация стрельбы

Теперь дадим игроку возможность стрелять снарядами.

```
public class Hero : MonoBehaviour {
public float gameRestartDelay = 2f;
public GameObject projectilePrefab;
public float projectileSpeed = 40;
void Update () {
transform.rotation = Quaternion.Euler(yAxis*pitchMult, xAxis*rollMult,0);

// Позволить кораблю выстрелить
if ( Input.GetKeyDown( KeyCode.Space ) ) {
TempFire();
}
void TempFire()
```

```
{ GameObject projGO = Instantiate<GameObject>( projectileprefab
);
projGO.transform.position = transform.position;
Rigidbody rigidB = projGO.GetComponent<Rigidbody>();
rigidB.velocity = Vector3.up * projectileSpeed; }
void OnTriggerEnter(Collider other) { ... }
}
```

2. В Unity выберите _Hero в иерархии и в поле projectilePrefab сценария Hero (в инспекторе) перетащите шаблон ProjectileHero из панели Project (Проект).

3. Сохраните сцену и щелкните по кнопке Play (Играть). Если теперь нажать пробел, корабль будет стрелять снарядами, но пока они не наносят ущерба вражеским кораблям и продолжают свой полет бесконечно, даже когда покинут экран.

Управление снарядами

Чтобы снаряды могли поражать врага, выполните следующие шаги.

1. Откройте сценарий Projectile и добавьте следующие строки, выделенные жирным. Сейчас мы реализуем только уничтожение снаряда после выхода за границы экрана

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;
public class Projectile : MonoBehaviour {
private BoundsCheck bndCheck;

void Awake ()
{ bndCheck = GetComponent<BoundsCheck>(); }

void Update () {
 if (bndCheck.offUp) { //a
Destroy( gameObject ); } }
}
```

И снова не забудьте сохранить.

Повреждение вражеских кораблей снарядами

Нам нужно также добавить возможность повреждения вражеских кораблей снарядами.

Откройте сценарий Enemy и добавьте в конец следующий метод:

```
public class Enemy : MonoBehaviour {
public virtual void Move() (...)

void OnCollisionEnter( Collision coll ) {
GameObject otherGO = coll.gameObject; // a
if ( otherGO.tag == "ProjectileHero” ) { // b
Destroy( otherGO );
// Уничтожить снаряд
Destroy( gameObject );
// Уничтожить игровой объект Enemy
} else {
print( "Enemy hit by non-ProjectileHero: " + otherGO.name ); } }
J
```

2. Получите игровой объект, которому принадлежит коллайдер, столкнувшийся с вражеским кораблем.

3. Если otherGO имеет тег ProjectileHero, тогда уничтожьте его и данный экземпляр Enemy.

4. Если otherGO не имеет тега ProjectileHero, выведите его имя в консоль для отладки

Если теперь щелкнуть по кнопке Play (Играть), экземпляры Enemy_0 начнут движение по экрану сверху вниз, и вы сможете расстреливать их снарядами.

1.4. Создание карточной игры

Прототип 4: PROSPECTOR SOLITAIRE

Мы создадим свою первую карточную игру – цифровую версию популярного пасьянса Tri-Peaks (Три вершины).

Графические ресурсы, которые мы будем использовать, созданы на основе общедоступного комплекта карт Vectorized Playing Cards 1.3 Криса Агиляра.

Откройте сцену__Prospector_Scene_0 и проверьте настройки - MainCamera.

-MainCamera (Camera) P:[0, 0, -40] R:[0, 0, 0] S:[1, 1, 1] Projection: Orthographic Size: 10

Импортирование изображений в виде спрайтов

Далее нужно правильно импортировать изображения, чтобы их можно было использовать как спрайты. Спрайт – это двумерное изображение, которое может перемещаться по экрану, масштабироваться и поворачиваться. Они часто используются в двумерных играх:

1. Откройте папку .Sprites в панели Project (Проект) и выберите все изображения в ней.

2. В разделе Texture 2Ds Import Settings (Настройки импортирования двумерной текстуры), в панели Inspector (Инспектор), выберите в раскрывающемся списке Texture Type (Тип текстуры) пункт Sprite (2D and UI) (Спрайт (2D и ПИ)). Щелкните по кнопке Apply (Применить), и Unity повторно импортирует все изображения с правильными соотношениями сторон.

3. Выберите в панели Project (Проект) изображение с именем Letters.

4. В разделе Letters Import Settings (Настройки импортирования Letters), в панели Inspector (Инспектор), выберите в раскрывающемся списке Sprite Mode (Режим спрайта) пункт Multiple (Множественный) и щелкните на кнопке Apply (Применить).

5. Щелкните по кнопке Sprite Editor (Редактор спрайтов), чтобы открыть редактор. Вы увидите изображение Letters в редакторе, окруженное синей рамкой, определяющей границы спрайта Letters.

6. Щелкните в редакторе спрайтов по маленькой пиктограмме с изображением радуги или буквы A, чтобы переключиться между просмотром фактического изображения и его альфа-канала. Поскольку Letters содержит изображения белых букв на прозрачном фоне, вам проще будет видеть происходящее, глядя на альфа-канал.

7. Щелкните по раскрывающемуся списку Slice (Нарезать) в левом верхнем углу редактора спрайтов.

A. В раскрывающемся списке Type (Тип) замените значение Automatic (Автоматически) на Grid by Cell Size

B. В поле Pixel size (Размер в пикселах) установите значения X:32 и Y:32.

C. Щелкните по кнопке Slice (Нарезать). В результате изображение Letters будет разделено по горизонтали на 16 спрайтов с размерами 32 x 32 пиксела.

D. Щелкните по кнопке Apply (Применить) в правом верхнем углу редактора спрайтов, чтобы сгенерировать спрайты в панели Project (Проект). Теперь в панели Project (Проект) вместо одного спрайта Letters внутри текстуры Letters появятся 16 спрайтов с именами от Letters_O до Letters_15. Спрайты Letters_l – Letters_13 будут использоваться в этой игре для изображения достоинства карт (от туза до короля). Теперь все спрайты на месте и готовы к использованию.

8. Сохраните сцену. В действительности мы еще ничего не изменили в сцене, но сохранять ее постоянно – хорошая привычка. Возьмите за правило сохранять сцену после любого изменения.

Конструирование карт из спрайтов

Один из самых важных аспектов этого проекта – конструирование всей колоды карт программно с использованием 21 импортированного изображения. Это поможет уменьшить окончательный размер сборки и увидеть, как работать с файлами XML.

Использование XML в программном коде

1. Для первой части этого проекта создайте три сценария на C# с именами Card, Deck и Prospector. Поместите их в папку__Scripts.

Card: класс, представляющий каждую карту в колоде, хранит информацию о позициях всех спрайтов на карте каждого достоинства) и Decorator.

Deck: класс, интерпретирующий информацию из файла DeckXML.xml и на ее основе создающий колоду карт.

Prospector: класс, управляющий всей игрой. Если класс Deck обрабатывает создание карт, то класс Prospector включает эти карты в игру. Prospector собирает карты в разные стопки (например, стопка свободных карт и стопка для сброшенных карт) и управляет логикой игры.

2. Откройте сценарий Card и введите следующий код. Эти маленькие классы в Card, cs предназначены для хранения информации, которую класс Deck получает при чтении XML-файла.

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;
```

```
public class Card : MonoBehaviour {
// Будет определен позже }

[System.Serializable] // Сериализуемый класс доступен для правки в инспекторе
public class Decorator {
 // Этот класс хранит информацию из DeckXML о каждом значке на карте
public string type; // Значок, определяющий достоинство карты, имеет
 // type = "pip"
public Vector3 loc; // Местоположение спрайта на карте
public bool flip = false; // Признак переворота спрайта по вертикали
public float scale = If; // Масштаб спрайта }

[System.Serializable]
public class CardDefinition {
// Этот класс хранит информацию о достоинстве карты
public string face; // Спрайт, изображающий лицевую сторону карты
public int rank; // Достоинство карты (1-13)
public List<Decorator> pips = new List<Decorator>(); // Значки // a
}
```

3. Откройте сценарий Deck и введите следующий код:

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

public class Deck : MonoBehaviour {

[Header("Set Dynamically")]
public PT_XMLReader xmlr;

// InitDeck вызывается экземпляром Prospector, когда будет готов
public void InitDeck(string deckXMLText) {
ReadDeck(deckXMLText); }
```

```
// ReadDeck читает указанный XML-файл и создает массив экземпляров CardDefinition
public void ReadDeck(string deckXMLText) {
xmlr = new PT_XMLReader(); // Создать новый экземпляр PT_XMLReader
xmlr.Parse(deckXMLText); // Использовать его для чтения DeckXML

// Вывод проверочной строки, чтобы показать, как использовать xmlr.
string s = ”xml[0] decorator[0] ";
 s += "type="+xmlr.xml["xml"][0]["decorator"][0].att("type");
s += " x="+xmlr.xml["xml"][0]["decorator"][0].att("x");
s += " y="+xmlr.xml["xml"][0]["decorator"][0].att("y");
s += " scale="+xmlr.xml["xml"][0]["decorator"][0].att("scale");
print(s);
}
}
```

4. Теперь откройте сценарий Prospector и введите следующий код:

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;
using UnityEngine.SceneManagement; // Будет использоваться позже
using UnityEngine.UI; // Будет использоваться позже

public class Prospector : MonoBehaviour {
static public Prospector S;

[Header("Set in Inspector")] p
ublic TextAsset deckXML;

[Headerf’Set Dynamically")]
public Deck deck;

void Awake() {
```

```
S = this; // Подготовка объекта-одиночки Prospector }


Void Start() {

deck = GetComponent<Deck>(); // Получить компонент Deck
 deck.InitDeck(deckXML.text); // Передать ему DeckXML
}
}
```

5. Обязательно сохраните все эти сценарии перед возвратом в Unity. В меню MonoDevelop выберите пункт File > Save АП (Файл > Сохранить все). Если пункт Save АП (Сохранить все) неактивен, значит, вы уже сохранили их.

6. Теперь вернитесь в Unity и подключите оба сценария – Prospector и Deck – к -MainCamera. (Перетащите их по очереди из панели Project (Проект) на объект -MainCamera в панели Hierarchy (Иерархия).) Выберите -MainCamera в иерархии. Вы должны увидеть, что оба сценария подключены как компоненты Script.

7. Перетащите DeckXML из папки Resources в панели Project (Проект) на поле deckXML TextAsset компонента Prospector (Script) в инспекторе.

8. Сохраните сцену и щелкните по кнопке Play (Играть). В консоли должна появиться строка:

xml[0] decorator[0] typesetter x=-1.05 y=1.42 scale=l.25

Связывание спрайтов, составляющих карты

Теперь, когда содержимое XML-файла благополучно читается в списке, можно приступать к конструированию карт. Для начала получим ссылки на все спрайты, созданные выше:

1. Добавьте следующие поля в начало класса Deck для хранения ссылок на спрайты:

```
public class Deck : MonoBehaviour {
[Header("Set in Inspector'')]
// Масти
public Sprite suitClub;
public Sprite suitDiamond;
 public Sprite suitHeart;
public Sprite suitSpade;
```

```
public Sprite[] faceSprites;
public Sprite[] ranksprites;


public Sprite cardBackGold;
public Sprite cardFront;
public Sprite cardFrontGold;
// Шаблоны public GameObject public GameObject
prefabCard; prefabsprite;
[Header("Set Dynamically")]
}
```

2. Перетащите текстуры Club, Diamond, Heart и Spade из папки -Sprites в панели Project (Проект) в соответствующие поля компонента Deck (suitClub, suitDiamond, suitHeart и suitSpade). Unity автоматически запишет переменные ссылки на спрайты (а не на текстуры Texture2D).

3. Следующий шаг чуть сложнее. Заблокируйте инспектор для -MainCamera, выбрав -MainCamera в иерархии и затем щелкнув по маленькому значку с изображением замка вверху на панели Inspector (Инспектор). Блокировка панели Inspector (Инспектор) гарантирует, что ее содержимое не изменится при выборе чего-то еще.

4. Свяжите все спрайты, начиная с FaceCard_, с элементами массива faceSprites компонента Deck (Script) в инспекторе:

A. Выберите FaceCard_llC в папке -Sprites панели Project (Проект), нажмите клавишу Shift и, не отпуская ее, щелкните по FaceCard_13S. В результате должны быть выделены все 12 спрайтов FaceCard_.

B. Перетащите эту группу из панели Project (Проект) на имя массива faceSprites компонента Deck (Script) в инспекторе. Когда указатель мыши окажется над именем массива faceSprites, рядом с ним должен появиться значок + и слово <multiple> (на PC может появиться только значок +).

C. Отпустите кнопку мыши, и если все было сделано правильно, размер массива faceSprites должен увеличиться до 12, а его элементы – заполниться ссылками на спрайты FaceCard_. Если что-то не получилось, добавьте спрайты по одному.

5. Щелкните на пиктограмме с изображением треугольника рядом со списком текстур Letters в папке -Sprites в панели Project (Проект). Повторите процедуру, описанную на предыдущем шаге, чтобы выбрать текстуры с Letters_O по Letters_15.

6. Перетащите спрайты Card_Back, Card_Back_Gold, Card_Front и Card_Front_Gold из панели Project (Проект) в соответствующие им переменные компонента Deck (Script) в инспекторе.

7. Разблокируйте панель Inspector (Инспектор). Сохраните сцену! Будет обидно, если всю эту работу придется повторить.

Игра Prospector

Пасьянс Prospector основан на классическом карточном пасьянсе Tri-Peaks. Правила обеих игр совпадают, кроме двух аспектов:

– по сюжету Prospector игрок добывает золото, тогда как в Tri-Peaks он пытается покорить три вершины;

– цель игрока в Tri-Peaks – собрать все карты в одну колоду. Цель игрока в Prospector – зарабатывать очки, собирая как можно более длинные цепочки карт, и каждая золотая карта в цепочке удваивает количество очков всей цепочки.

Правила игры в Prospector

Для пробы возьмите обычную колоду игральных карт (то есть колоду настоящих карт, не виртуальных, которые мы только что создали). Уберите из колоды джокеры и перетасуйте оставшиеся 52 карты.

1. Разложите 28 карт следующим образом. Карты в трех нижних рядах положите рубашкой вверх, а карты в верхнем ряду – лицевой стороной вверх. Карты не должны перекрывать друг друга с боков, но карты в верхних рядах должны перекрывать карты в нижних рядах. Это начальная раскладка, изображающая «шахту», которую должен выкопать старатель.

2. Остальные карты из колоды образуют стопку свободных карт. Положите эту стопку над верхним рядом рубашкой вверх.

3. Снимите верхнюю карту со стопки свободных карт и положите ее в центре над верхним рядом. Это – целевая карта.

4. Вы можете перенести на нее любую карту из раскладки шахты, которая на ступень старше или младше целевой, и таким образом объявить новую целевую карту Для тузов и королей действует правило циклического перехода старшинства, то есть туз можно положить на короля и наоборот.

5. Если карта, повернутая рубашкой вверх, не перекрывается картами из верхнего ряда, ее можно перевернуть лицевой стороной вверх.

6. Если ни одну из карт в раскладке, лежащих лицевой стороной вверх, нельзя положить на целевую карту, снимите новую карту со стопки свободных карт.

7. Если вам удалось переложить все карты из раскладки до исчерпания стопки свободных карт, вы победили! (Обсуждение правил подсчета очков и появления золотых карт в цифровой версии игры я отложу на потом.)

Программная реализация Prospector

Как можно заметить из предыдущего описания, Prospector– очень простая игра, но достаточно увлекательная.

Раскладка карт

В цифровой версии Prospector нам нужно реализовать ту же раскладку карт, имитирующую шахту, которую мы видели выше, когда пробовали сыграть с настоящими картами.

1. В Unity откройте файл LayoutXML.xml, находящийся в папке Resources, чтобы увидеть информацию о раскладке. Обратите внимание, что комментарии в разметке XML заключены в<!--и--> (по аналогии с /* и */ в коде на C#).

<xml>

<!-- Этот файл хранит информацию о раскладке карт в карточной игре Prospector. -->

<!-- Элемент multiplier имеет атрибуты x и y с множителями. -->

<!-- Множители определяют, насколько свободной или плотной будет раскладка. --> <multiplier x="1.25" y="1.5” />

<!-- В разметке XML ниже атрибут id определяет номер карты -->

<!-- x и y определяют позицию в раскладке -->

<!-- если faceup имеет значение 1, карта повернута лицом вверх -->

<!-- layer определяет номер слоя, необходимого для правильного перекрытия -->

<!-- hiddenby - номер карты, перекрывающей эту -->

<!-- Layer®, самый нижний ряд карт. -->

<slot ids"®" x="-6" y="-5” faceup=”0” layer="0" hiddenby="3,4" />

```
<slot id="l" x="0" y="-5" faceup="0" layer=”0" hiddenby="5,6" />
<slot id="2"x="б" y=’’-5" faceup="0" layer=”0” hiddenby="7,8" />

<!-- Layerl, второй ряд снизу. -->
<slot id=”3" x="-7" y="-4" faceup="0’' layer="l’’ hiddenby="9,10"
/>
<slot id="4" x="-5'• y="-4'" faceup=”0’’ layer="l" hid-
denby="10,11'1 />
<slot id="5" x="-l’• y="-4’’ faceup=”0" layer="l’’ hiddenby="12,13"
/>
<slot id="6" x="l" y="-4" faceup=”0” layer="l" hiddenby="13,14"
/>
<slot id="7" x="5" y="-4" faceup="0" layers"1" hiddenby="15,16"
/>
<slot id="8" x="7" y="-4" faceup=”0” layer="l" hiddenby="16,17"
/>

<!-- Layer2, третий ряд снизу. -->
<slot id="9" :x="-8" y="-3" faceup="0” layer="2" hiddenby="18,19"
/>
<slot id="10” x=”-6" y="-3’’ faceup=”0’' layer="2" hid-
denby="19,20’’ />
<slot id="ll" x="-4" y="-3" faceup="0" layer="2'1 hiddenby="20,21’’
/>
<slot id="12" x="-2" y="-3’’ faceup="0" layers'^'1 hiddenby="21,22‘'
/>
<slot id="13" x="0" y="-3" faceup=”0" layer="2" hiddenby="22,23"
/>
<slot id="14" x="2" y=”-3” faceup=”0" layers"2" hiddenby="23,24"
/>
<slot id="15" x=”4" y="-3" faceup=”0” layer="2" hiddenby="24,25"
/>
<slot id="16" x="6" y="-3" faceup="0" layer="2" hiddenby="25,26"
/>
<slot id="17" x="8" y="-3” faceup=”0" layers"2" hiddenby="26,27"
/>
```

```
<!-- Layer3, верхний ряд. • ->
 <slot id="18" x="-9" y="-2'' faceup="l'' layers"3'' />
<slot id="19" x="-7" y="-2'' faceup="l'' layer="3'' />
<slot id="22" x="-Г• y="-2" faceup=="Г" layer=:"3" />
<slot id="23" x="l" y="-2" faceup='' 1" layer='' 3" />
<slot id="24" x="3" y="-2" faceup=*'1" layer='•3" />
<slot id="25" x="5" y="-2" faceup=*'Г' layer='•3" />
 <slot id="26" x="7" y="-2" faceup='' 1" layer='' 3" />
 <slot id="27" x="9" y="-2" faceup='*1" layers'' 3" />
<!-- Позиция стопки свободных карт и их смещение друг относительно друга -->
<slot types"drawpile" x="6" y="4" xstagger=,'0.15" layer="4'7>


<!-- Позиция стопки сброшенных карт и целевой карты -->
 <slot type="discardpile" x="0" y="1" 1ayer="5"/>
 </xml>
```

2. Реализуем анализ этого файла LayoutXML и сохранение информации для дальнейшего использования. Создайте в папке__Seri pts новый сценарий на C# с именем Layout и введите в него следующий код:

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

// Класс SlotDef не наследует MonoBehaviour, поэтому для него не требуется
// создавать отдельный файл на C#.
 [System.Serializable] // Сделает экземпляры SlotDef видимыми в инспекторе Unity

public class SlotDef
{ public float x;
public float y;
public bool faceup = false;
public string layerName = "Default";
public int layerlD = 0;
```

```
public int id;
public List<int> hiddenBy s new List<int>();
 public string type = "slot";
public Vector2 stagger;

public class Layout : MonoBehaviour {
public PT_XMLReader xmlr; // Так же, как Deck, имеет PT_XMLReader
public PTXMLHashtable xml; // Используется для ускорения доступа к xml
public Vector2 multiplier; // Смещение от центра раскладки
// Ссылки SlotDef
public List<SlotDef> slotDefs; // Все экземпляры SlotDef для рядов 0-3
public SlotDef drawPile;
public SlotDef discardPile;
//Хранит имена всех рядов
public string[]sortingLayerNames = new string!] < "Row0”, "Rowl”, "Row2", "Row3", "Discard", "Draw" };

public void ReadLayout(string xmlText) {
xmlr = new PT_XMLReader();
xmlr.Parse(xmlText);
// Загрузить XML
xml = xmlr.xml["xml"][0];
// И определяется xml для ускорения доступа к XML

// Прочитать множители, определяющие расстояние между картами
multiplier.x = float.Parse(xml["multiplier"][0].att("x"));
multiplier.y = float.Parse(xml["multiplier"][0].att("y"));

// Прочитать слоты
SlotDef tSD;
// slotsX используется для ускорения доступа к элементам <slot>
PT_XMLHashList slotsX = xml["slot"];

for (int i=0; icslotsX.Count; i++) {
tSD = new SlotDef(); // Создать новый экземпляр SlotDef
```

```
if (slotsX[i].HasAtt("type")) {
// Если <slot> имеет атрибут type, прочитать его
tSD.type = slotsX[i].att("type");
} else { /
/ Иначе определить тип как "slot"; это отдельная карта в ряду
tSD.type = "slot"; }
// Преобразовать некоторые атрибуты в числовые значения
 tSD.x = float.Parse( slotsX[i].att("x") );
 tSD.y = float.Parse( slotsX[i].att("y") );
tSD.layerlD = int.Parse( slotsX[i].att("layer”) );
II Преобразовать номер ряда layerlD в текст layerName
 tSD.layerName = sortingLayerNames[ tSD.layerlD ];


switch (tSD.type) {
// прочитать дополнительные атрибуты, опираясь на тип слота
case "slot":
tSD.faceup = (slotsX[i].att("faceup") == "1");
tSD.id = int.Parse( slotsX[i].att("id") );
if (slotsX[i].HasAtt("hiddenby")) {
string[] hiding = slotsX[i].att("hiddenby").Split(',');
foreach( string s in hiding ) {
tSD.hiddenBy.Add ( int.Parse(s) ); } }
SlotDefs.Add(tSD);
 break;
case "drawpile":
tSD.stagger.x = float.Parse( slotsXfi].att("xstagger") );
drawPile = tSD;
break;
case "discardpile":
 discardPile = tSD; break;
}
}
}
```

3. Откройте сценарий Prospector и добавьте следующие строки:

```
public class Prospector : MonoBehaviour {
static public Prospector S;
```

```
[Header("Set in Inspector'')]
public TextAsset deckXML;
 public TextAsset layoutXML;
[Header("Set Dynamically")]
public Deck deck;
 public Layout layout;

void AwakeO (
S = this; // Подготовка объекта-одиночки Prospector )

void Start () {
deck = GetComponent<Deck>(); // Получить компонент Deck
deck.InitDeck(deckXML.text); // Передать ему DeckXML
Deck.Shuffle(ref deck.cards); // Перемешать колоду, передав ее по ссылке

// Этот фрагмент нужно закомментировать; сейчас мы создаем фактическую раскладку И Card c; //
for (int cNum=0; cNumcdeck.cards.Count; cNum++) {
// c = deck.cards[cNum];
//          c.transform.localPosition          =          new          Vec-tor3((cNum%13)*3,cNum/13*4,0); П }

layout = Getcomponent<Layout>(); // Получить компонент Layout
layout.ReadLayout(layoutXML.text); // Передать ему содержимое LayoutXML
}
}
```

4. Сохраните все сценарии в MonoDevelop и вернитесь в Unity.

5. В Unity выберите _MainCamera в иерархии. В главном меню выберите пункт Component > Scripts > Layout (Компонент > Сценарии > Layout), чтобы подключить сценарий Layout к-MainCamera (это еще один способ подключения сценария объекту).

6. Найдите компонент Prospector (Script) главной камеры _MainCamera. Вы увидите, что в нем появились новые общедоступные поля layout и layoutXML. Щелкните на пиктограмме с изображением мишени рядом с полем layoutXML и выберите LayoutXML на вкладке Assets. (Возможно, вам придется щелкнуть на вкладке Assets в верхней части диалога Select TextAsset (Выбор текстового ресурса).)

7. Сохраните сцену.
8. Щелкните на кнопке Play (Играть).

CardProspector – наследник класса Card

Прежде чем поместить карты в раскладку, нужно расширить класс Card, подготовив его для использования в игре Prospector. Так как классы Card и Deck проектировались с возможностью повторного использования в других карточных играх, вместо изменения самого класса Card создадим класс CardProspector, наследующий Card.

1. Создайте в папке__ Scripts новый сценарий на C# с именем CardProspector и введите следующий код:

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

// Перечисление, определяющее тип переменной, которая может принимать
 // несколько предопределенных значений
public enum eCardstate {
drawpile,
tableau,
target,
 discard }

public class CardProspector : Card { // CardProspector должен расширять Card
[Header(”Set Dynamically: CardProspector”)]
// Так используется перечисление eCardState
public eCardState state = eCardState.drawpile;
// hiddenBy - список других карт, не позволяющих перевернуть эту лицом вверх
public List<CardProspector> hiddenBy = new List<CardProspector>();
// layoutID определяет для этой карты ряд в раскладке
public int layoutID;
// Класс SlotDef хранит информацию из элемента <slot> в LayoutXML
public SlotDef slotDef; }
```

Теперь, определив подкласс, нужно изменить тип карт в колоде с Cards на CardProspectors.

2. Для этого добавьте следующий код в класс Prospector:

```
public class Prospector : MonoBehaviour {

void Awake() { ... }

[Header("Set Dynamically")]
public Deck deck;
public Layout layout;
 public List<CardProspector> drawPile;


void Start () {
layout = GetComponent<Layout>(); // Получить компонент Layout
layout.ReadLayout(layoutXML.text); // Передать ему содержимое
LayoutXML drawpile = ConvertListCardsToListCardProspectors( deck.cards );
}
List<CardProspector> ConvertListCardsToListCardProspectors(
List<Card> 1CD) { List<CardProspector> 1CP = new List<CardProspector>();
CardProspector tCP;
foreach( Card tCD in 1CD ) {
tCP = tCD as CardProspector; // a
lCP.Add( tCP ); } return( 1CP ); }
}
```

3. Сохраните все сценарии в MonoDevelop и вернитесь в Unity.

4. Запустите игру и посмотрите на поле drawPile компонента Prospector (Script) главной камеры -MainCamera в панели Inspector (Инспектор).

5. Чтобы провернуть этот трюк, выберите шаблон PrefabCard в панели Project (Проект), после чего он появится в инспекторе с компонентом Card (Script).

6. Щелкните по кнопке Add Component (Добавить компонент) и в открывшемся меню выберите пункт Scripts > CardProspector (Сценарии > CardProspector). Это действие добавит компонент CardProspector (Script) в шаблон PrefabCard.

7. Чтобы удалить ненужный компонент Card (Script), щелкните на кнопке с изображением шестеренки в правом верхнем углу в разделе Card (Script) и в открывающемся меню выберите удалить компонент

8. Выберите -MainCamera в иерархии и запустите сцену; вы увидите, что теперь вместо null все элементы в drawPile хранят ссылки на экземпляры CardProspector.

9. Сохраните сцену. Вы знаете правила.

Позиционирование карт в раскладке

Теперь у нас есть все что нужно, и мы можем добавить в класс Prospector код, который фактически создает раскладку карт в игре:

```
public class Prospector : MonoBehaviour {
static public Prospector S;
[Header("Set in Inspector")]
public TextAsset deckXML;
public TextAsset layoutXML;
public float xOffset = 3;
public float yOffset = -2.5f;
public Vector3 layoutcenter;

[Header("Set Dynamically")]
public Deck deck;
public Layout layout;
public List<CardProspector> drawPile;
public Transform layoutAnchor;
public CardProspector target;
public List<CardProspector> tableau;
public List<CardProspector> discardPile;

void Awake() (...)

void Start () {
drawPile = ConvertListCardsToListCardProspectors( deck.cards );
LayoutGame();
)
```

```
List<CardProspector>          ConvertListCardsToListCardProspec-
tors(List<Card> 1CD) {
)
// Функция Draw снимает одну карту с вершины drawPile и воз-
вращает ее
CardProspector Draw() {
 CardProspector cd = drawPile[0]; /
/ Снять 0-ю карту CardProspector
drawPile.RemoveAt(0);
// Удалить из Listo drawPile
return(cd); // И вернуть ее


// LayoutGameQ размещает карты в начальной раскладке - "шах-
те"
 void LayoutGameQ {
// Создать пустой игровой объект} который будет служить цен-
тром раскладки // a
if (layoutAnchor == null) {
GameObject tGO = new GameObject("-LayoutAnchor");
 // Л Создать пустой игровой объект с именем -LayoutAnchor в
иерархии
layoutAnchor = tGO.transform; // Получить его компонент
Transform
layoutAnchor.transform.position = layoutcenter; // Поместить в
центр }

CardProspector cp;
// Разложить карты
foreach (SlotDef tSD in layout.slotDefs) {
// Л Выполнить обход всех определений SlotDef в layout.slotDefs
cp = Draw(); // Выбрать первую карту (сверху) из стопки
drawPile
cp.faceup = tSD.faceUp; // Установить ее признак faceup И в со-
ответствии с определением в SlotDef
cp.transform.parent = layoutAnchor; // Назначить layoutAnchor ее
родителем
// Эта операция заменит предыдущего родителя:
deck.deckAnchor, который
```

```
// после запуска игры отображается в иерархии с именем _Deck.
cp.transform.localPosition = new Vector3(
 layout.multiplier.x * tSD.x,
layout.multiplier.y ♦ tSD.y,
 -tSD.layerlD );
// Л Установить localPosition карты в соответствии
// с определением в SlotDef
 cp.layoutID = tSD.id;
cp.slotDef = tSD;
// Карты CardProspector в основной раскладке имеют состояние
// Cardstate.tableau cp.state = eCardstate.tableau;
tableau.Add(cp); // Добавить карту в список tableau } } }
```

Сохраните сценарий и вернитесь в Unity. Запустите игру, и вы увидите, что карты действительно разместились в раскладке, представляющей шахту, которая описана в файле LayoutXML.xml, при этом карты в верхнем ряду правильно отображаются лицевой стороной вверх, а остальные – лицевой стороной вниз.

Реализация логики игры

Прежде чем начать двигать карты, определим, что может происходить в игре:

A. Если целевая карта замещается любой другой картой, прежняя целевая карта перемещается в стопку сброшенных карт.

B. Верхнюю карту в стопке свободных карт можно открыть и сделать целевой.

C. Открытую карту в основной раскладке, которая на единицу старше или младше целевой, можно сделать новой целевой картой.

D. Если карта, лежащая лицевой стороной вниз, не перекрывается другими картами, она поворачивается лицевой стороной вверх.

E. Игра завершается, когда убирается последняя карта из основной раскладки (победа) или опустошается стопка свободных карт и нет возможных ходов (проигрыш).

Пункты B и C в этом списке связаны с активными действиями – перемещением карт игроком. A пункты A, D и E определяют пассивные события, происходящие в результате действий в пунктах B и C.

Добавление в карты реакции на щелчок мыши

Поскольку все активные действия инициируются щелчком мыши на карте, первым делом добавим возможность реагировать на щелчок.

1. Реакция карт на щелчок мыши необходима в любой карточной игре, поэтому добавим следующий метод в конец класса Card:

```
public class Card : MonoBehaviour {

public bool faceUp {
get { ... }
set { ... } }

virtual public void OnMouseUpAsButton() {
 print (name); /
/ По щелчку эта строка выведет имя карты }
}
```

Если теперь запустить сцену и щелкнуть на какой-нибудь карте, в консоли появится ее имя.

2. Однако в игре Prospector щелчок на карте должен выполнять нечто большее, поэтому добавим в конец класса CardProspector следующий метод:

```
public class CardProspector : Card {
// CardProspector должен расширять Card

// Класс SlotDef хранит информацию из элемента <slot> в LayoutXML
public SlotDef slotDef;

// Определяет реакцию карт на щелчок мыши
override public void OnMouseUpAsButton() {
 // Вызвать метод CardClicked объекта-одиночки
Prospector Prospector.S.CardClicked(this);
// а также версию этого метода в базовом классе (Card.cs)
base.OnMouseUpAsButton(); // a }
}
```

3. Дополнительно мы должны добавить в сценарий Prospector метод CardClicked (именно из-за его отсутствия только что введенный код подчеркнут красной волнистой линией), но сначала напишем не-

сколько вспомогательных функций. Добавьте в конец класса Prospector методы MoveToDiscard(), MoveToTargetQ и UpdateDrawPile().

```
public class Prospector : MonoBehaviour {
void MoveToDiscard(CardProspector cd)
// Установить состояние карты как discard (сброшена)
cd.state = eCardstate.discard;
discardPile.Add(cd);
// Добавить ее в список
 discardPile cd.transform.parent = layoutAnchor;
// Обновить значение transform.parent

// Переместить эту карту в позицию стопки сброшенных карт
cd.transform.localPosition = new Vector3(
layout.multiplier.x ♦ layout.discardPile.x,
layout.multiplier.y * layout.discardPile.y,
-layout.discardPile.layerID+0.5f );
cd.faceUp = true;
// Поместить поверх стопки для сортировки по глубине
cd.SetSortingLayerName(layout.discardPile.layerName);
cd.SetSortOrder(-100+discardPile.Count);
}

// Делает карту cd новой целевой картой
void MoveToTarget(CardProspector cd) {
// Если целевая карта существует, переместить ее в стопку сброшенных карт
if (target != null) MoveToDiscard(target);
target = cd;
// cd - новая целевая карта
cd.state = eCardState.target;
cd.transform.parent = layoutAnchor;
// Переместить на место для целевой карты
cd.transform.localPosition = new Vector3(
layout.multiplier.x ♦ layout.discardPile.x,
layout.multiplier.y * layout.discardPile.y,
-layout.discardPile.layerlD );
cd.faceUp = true; // Повернуть лицевой стороной вверх
```

```
 // Настроить сортировку по глубине
cd.SetSortingLayerName(layout.discardPile.layerName);
cd.SetSortOrder(0);
}
```

// Раскладывает стопку свободных карт, чтобы было видно, сколько карт осталось

```
void UpdateDrawPile() {
CardProspector cd;
// Выполнить обход всех карт в drawPile
for (int i=0; icdrawPile.Count; i++) {
cd = drawPile[i];
cd.transform.parent = layoutAnchor;

// Расположить с учетом смещения layout.drawPile.stagger
Vector2 dpStagger = layout.drawPile.stagger;
cd.transform.localPosition = new Vector3(
layout.multiplier.x * (layout.drawPile.x + i*dpStagger.x),
layout.multiplier.y ♦ (layout.drawPile.y + i*dpStagger.y),
-layout.drawPile.layerID+0.If*i );
cd.faceUp = false;
// повернуть лицевой стороной вниз
cd.state = eCardState.drawpile;
// Настроить сортировку по глубине
cd.SetSort i ng L aye rName(layout.d rawP ile.laye rName);
 cd.SetSortOrde r(-10* i);
}
```

4. Добавьте следующий код в конец Prospector. LayoutGameQ, чтобы нарисовать начальную целевую карту и разложить стопку свободных карт. В этом листинге, в конце класса Prospector, также определена начальная версия метода CardClicked() для обработки щелчков мыши на всех картах CardProspector.

```
public class Prospector : MonoBehaviour {

// LayoutGame() размещает карты в начальной раскладке - "шахте"
void LayoutGameQ {

foreach (SlotDef tSD in layout.slotDefs) {
```

```
tableau.Add(cp); // Добавить карту в список tableau
}

// Выбрать начальную целевую карту
MoveToTarget(Draw ());

// Разложить стопку свободных карт
UpdateDrawPile();
}
// Перемещает текущую целевую карту в стопку сброшенных карт void MoveToDiscard(CardProspector cd) { ... }

void MoveToTarget(CardProspector cd) { ... }

void UpdateDrawPileQ { ... }

// CardClicked вызывается в ответ на щелчок на любой карте
public void CardClicked(CardProspector cd) {
// Реакция определяется состоянием карты
switch (cd.state) {
 case eCardState.target: /
/ Щелчок на целевой карте игнорируется
break;

case eCardState.drawpile:
// Щелчок на любой карте в стопке свободных карт приводит
 // к смене целевой карты
MoveToDiscard(target);
// Переместить целевую карту в discardpile
MoveToTarget(Draw());
// Переместить верхнюю свободную карту
// на место целевой
UpdateDrawPileQ; /
/ Повторно разложить стопку свободных карт
break;

case eCardState.tableau: /
/ Для карты в основной раскладке проверяется возможность // ее перемещения на место целевой
```

```
 break;
}
}
```

5. Сохраните все сценарии в MonoDevelop, вернитесь в Unity и запустите сцену

Теперь вы сможете щелкнуть по стопке свободных карт (в правом верхнем углу экрана), чтобы вытянуть новую целевую карту Сейчас мы уже близки к завершению игры!

Проверка карт из основной раскладки

Чтобы можно было переместить выбранную карту из основной раскладки, мы должны убедиться, что она на одну ступень старше или младше целевой (конечно же, с учетом циклического переноса старшинства туз/король).

1. Добавьте в метод CardClicked() класса Prospector следующие строки:

```
public class Prospector : MonoBehaviour {
// CardClicked вызывается в ответ на щелчок на любой карте
public void CardClicked(CardProspector cd) {
// Реакция определяется состоянием карты
switch (cd.state) {

case eCardstate.tableau: // Для карты в основной раскладке проверяется возможность
// ее перемещения на место целевой
bool validMatch = true;
if (Icd.faceUp) {
// Карта, повернутая лицевой стороной вниз, не может
// перемещаться
validMatch = false; 1

if (!AdjacentRank(cd, target)) {
// Если правило старшинства не соблюдается,
// карта не может перемещаться
validMatch = false; }

if (!validMatch) return;
// Выйти, если карта не может перемещаться
// Мы оказались здесь: Ура! Карту можно переместить,
```

```
tableau.Remove(cd);
 // Удалить из списка tableau
MoveToTarget(cd); // Сделать эту карту целевой
break;
}
}
// Возвращает true, если две карты соответствуют правилу старшинства
// (с учетом циклического переноса старшинства между тузом и королем)
public bool AdjacentRank(CardProspector c0, CardProspector cl) {
// Если любая из карт повернута лицевой стороной вниз, И правило старшинства не соблюдается.
if (!c0.faceUp || Icl.faceUp) return(false);
if (Mathf.Abs(c0.rank - cl.rank) == 1) {
return(true); }
// Если одна карта - туз, а другая - король, правило старшинства
 // соблюдается
 if (cO.rank == 1 && cl.rank == 13) return(true);
 if (cO.rank == 13 && cl.rank == 1) return(true);
// Иначе вернуть false return(false);
}
}
```

2. Сохраните сценарий в MonoDevelop и вернитесь в Unity.

3. Для этого добавьте в класс Prospector следующий код:

```
public class Prospector : MonoBehaviour {

// LayoutGame() размещает карты в начальной раскладке - "шахте”
void LayoutGame() {

CardProspector cp;
// Разложить карты
 foreach (SlotDef tSD in layout.slotDefs) {
tableau.Add(cp); // Добавить карту в список tableau
}

// Настроить списки карт, мешающих перевернуть данную
foreach (CardProspector tCP in tableau) {
```

```
        foreach( int hid in tCP.slotDef.hiddenBy ) {
         cp = FindCardByLayoutlD(hid);
        tCP.hiddenBy.Add(cp); } }
        // Выбрать начальную целевую карту
        MoveToTarget(Draw ());

        // Разложить стопку свободных карт
        UpdateDrawPile();
        }

        foreach (CardProspector tCP in tableau) {
        // Поиск по всем картам в списке tableau
        if (tCP.layoutID == layoutID) {
        // Если номер слота карты совпадает с искомым, вернуть ее
        return( tCP ); } } // Если ничего не найдено, вернуть null return(
null );
        }

        // Поворачивает карты в основной раскладке лицевой стороной
вверх или вниз
        void SetTableauFaces() {
        foreach( CardProspector cd in tableau ) {
        bool faceUp = true;
        // Предположить, что карта должна быть
         // повернута лицевой стороной вверх
        foreach( CardProspector cover in cd.hiddenBy ) {
         // Если любая из карт, перекрывающих текущую, присутствует
        // в основной раскладке
        if (cover.state == eCardState.tableau) {
         faceUp = false;
        // повернуть лицевой стороной вниз } }
         cd.faceUp = faceUp;
        // Повернуть карту так или иначе } }
        // Перемещает текущую целевую карту в стопку сброшенных
карт

         void MoveToDiscard(CardProspector cd) { ... }
```

```
// Делает карту cd новой целевой картой
oid MoveToTarget(CardProspector cd) { ... }

// Раскладывает стопку свободных карт, чтобы было видно, сколько карт осталось
void UpdateDrawPileQ (...)

// CardClicked вызывается в ответ на щелчок на любой карте
public void CardClicked(CardProspector cd) {
// Реакция определяется состоянием карты
switch (cd.state) {

case eCardState.tableau:


// Мы оказались здесь: Ура! Карту можно переместить,
tableau.Remove(cd); // Удалить из списка tableau
MoveToTarget(cd); // Сделать эту карту целевой
SetTableauFacesQ; // Повернуть карты в основной раскладке // лицевой стороной вниз или вверх
break;
}
}
// Возвращает true, если две карты соответствуют правилу старшинства // (с учетом циклического переноса старшинства между тузом и королем) public bool AdjacentRank(CardProspector c0, CardProspector cl) { }
```

Теперь, сохранив сценарии и вернувшись в Unity, вы сможете сыграть полноценный раунд!

1.5. Создание простой игры в слова

Прототип 5 : Word Game
Об игре Word Game

Это классическая форма игры в слова. В числе коммерческих версий этой игры можно назвать Word Whomp студии Pogo.com, Jumbline 2 студии Branium, Pressed for Words студии Words and Maps и многие другие. Игроку предлагается перемешанный набор букв, из которых можно составить не менее одного слова определенной длины

(обычно шесть букв), и он должен найти все слова, которые можно создать перестановкой букв.

Парсинг списка слов

Эта игра использует видоизмененный список слов 2of12inf, созданный Аланом Билом. Вы свободно можете использовать этот список в будущем, при соблюдении условий авторских прав Алана Била и Кевина Аткинсона, как определено в сноске. Был изменен список, преобразовав все буквы в верхний регистр и заменив заключительные символы строк \r\n (возврат каретки и перенос строки, которые стандартно используются в текстовых файлах в операционной системе Windows) на \n (единственный символ перевода строки, стандартно используемый в текстовых файлах в macOS).

В таких играх, как Scrabble или Letterpress, игрок получает набор плиток с буквами и может выбирать, какие слова записать, используя эти буквы. Но в этой игре в зачет игроку идут только слова, присутствующие в списке.

1. Создайте в папке__Scripts новый сценарий на C# с именем WordList и введите следующий код:

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

public class WordList : MonoBehaviour {
private static WordList S;

[Header(’’Set in Inspector")]
public TextAsset wordListText;
public int numToParseBeforeYield = 10000;
public int wordLengthMin = 3;
public int wordLengthMax = 7;

[Header("Set Dynamically’*)]
public int currLine = 0;
public int totalLines;
public int longWordCount;
public int wordcount;
```

```
// Скрытые поля
private stringf] lines;
 private List<string> longWords;
private List<string> words;

void Awake() {
S = this; // Подготовка объекта-одиночки WordList }

void Start () {
lines = wordListText.text.Split('\n');
totalLines = lines.Length;
Startcoroutine( ParseLines() );
}

Public IEnumerator ParseLines(){
string word;
// Инициализировать список для хранения длиннейших слов
// из числа допустимых
longWords = new List<string»(); // f
words = new List<string»();

for (currLine = 0; currLine < totalLines; currLine++) {
word = linesfcurrLine];
// Если длина слова равна wordLengthMax...
if (word.Length == wordLengthMax) {
longWords.Add(word); // ...сохранить его в longWords }

// Если длина слова между wordLengthMin и wordLengthMax...
if ( word.Length>=wordLengthMin &&
word.Length<=wordLengthMax ) {
words.Add(word); // ...добавить его в список допустимых слов }

// Определить, не пора ли сделать перерыв
if (currLine % numToParseBeforeYield == 0) {
// Подсчитать слова в каждом списке, чтобы показать,
// как протекает процесс анализа
longWordCount = longWords.Count; wordCount = words.Count;
```

```
// Приостановить выполнение сопрограммы до следующего кад-
ра
yield return null; // i
// Инструкция yield приостановит выполнение этого метода,
// даст возможность выполниться другому коду и возобновит
// выполнение сопрограммы с этой точки, начав следующую
 // итерацию цикла for.
}
} longWordCount = longWords.Count;
wordCount = words.Count;
}
// Эти методы позволяют другим классам
// обращаться к скрытым полям List<string» // j
static public List<string» GET_WORDS() {
return( S.words ); }

static public string GET_WORD(int ndx) {
return( S.words[ndx] ); }

static public List<string» GET_LONG_WORDS() {
return( S.longWords ); }

static public string GET_LONG_WORD(int ndx) {
return( S.longWords[ndx] );

static public int WORD_COUNT {
get { return S.wordCount; } }

static public int LONG_WORD_COUNT {
get { return S.longWordCount; } }

static public int NUM_TO_PARSE_BEFORE_YIELD {
get { return S.numToParseBeforeYield; } }
static public int WORD_LENGTH_MIN {
get { return S.wordLengthMin; } }

static public int WORD_LENGTH_MAX {
get { return S.wordLengthMax; } }
}
```

2. Сохраните сценарий и вернитесь в Unity.

3. Подключите сценарий WordList к главной камере _MainCamera.

4. Выберите _MainCamera в иерархии и в инспекторе присвойте переменной wordListText компонента WordList (Script) ссылку на файл 2ofl2inf, который вы найдете в папке Resources в панели Project (Проект).

5. Щелкните по кнопке Play (Играть).

Вы увидите, как начнут увеличиваться значения currLine, longWordCount и wordCount с шагом 10 000.

Настройка игры

Наша следующая цель – определить класс WordGame, который будет управлять игрой, но перед этим мы должны внести несколько изменений в WordList. Во-первых, перенесем запуск анализа списка слов из метода Start() в метод Init(), который будет вызываться другим классом. Во-вторых, сценарий WordList должен уведомить будущий сценарий WordGame, когда закончит анализ слов. Для этого организуем отправку сообщения из WordList объекту _MainCamera с использованием метода SendMessage(). Как сценарий WordGame будет интерпретировать это сообщение, вы увидите чуть позже.

1. Измените имя метода void Start() в сценарии WordList на public void Init() и добавьте следующие строки:

```
public class WordList : MonoBehaviour {

void Awake() { ... }

public void Init () { // Эта строка заменила "void Start()"
lines = wordListText.text.Split('\n');
totalLines = lines.Length;
StartCoroutine( ParseLines() );
}

static public void INIT () { //
S.lnit(); }
```

```
// Все сопрограммы возвращают значение типа IEnumerator.
public IEnumerator ParseLines() {
for (currLine = 0; currLine < totalLines; currLine++) {

} longWordCount = longWords.Count;
wordCount = words.Count;

// Послать игровому объекту gameObject сообщение об окончании анализа
gameObject.SendMessage("WordListParseComplete");
}

// Эти методы позволяют другим классам
// обращаться к скрытым полям List<string> s
tatic public List<string> GET_WORDS() { ... }
}
```

2. Создайте в папке__Scripts сценарий на C# с именем WordGame и подключите его к-MainCamera. Введите следующий код, чтобы воспользоваться изменениями, только что произведенными в WordList:

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;
 using System.Linq; // Мы будем использовать LINQ

public enum GameMode {
preGame., // Перед началом игры
makeLevel, // Создается отдельный WordLevel
levelPrep, // Создается уровень с визуальным представлением
inLevel // Уровень запущен
}
public class WordGame : MonoBehaviour {
static public WordGame S; // Одиночка

[Header("Set Dynamically")]
public GameMode mode = GameMode.preGame;
```

```
void Awake() {
S = this; // Записать ссылку на объект-одиночку }

void Start () {
mode = GameMode.loading;
// Вызвать статический метод INIT() класса WordList
WordList.INIT(); }
// Вызывается методом SendMessage() из WordList
public void WordListParseComplete() {
 mode = GameMode.makeLevel; }
}
```

3. Выберите _MainCamera в панели Hierarchy (Иерархия) и раскройте содержимое компонента WordGame (Script) в инспекторе. Щелкните по кнопке Play (Играть), и вы увидите, как значение preGame в поле mode сменится на loading.

Создание уровня с помощью класса WordLevel

Теперь пришла пора на основе слов из WordList создать уровень. Класс WordLevel будет включать:

– Длинное слово, на котором основывается уровень. (Если maxWordLength равно 6, это будет шестисимвольное слово, буквы из которого будут использоваться для конструирования других слов.)

– индекс этого слова в массиве longWords класса WordList;

– номер уровня в int levelNum. В этой главе при запуске каждого нового сеанса игра будет выбирать случайное слово1;

– словарь Dictionaryc, > с символами в слове и количеством раз их использования. Словари являются частью System. Collections. Generic вместе со списками;

– список List о всех других слов, которые можно сформировать из символов в словаре, описанном в предыдущем пункте.

Словарь Dictionaryc,> – это обобщенный тип коллекции, хранящей множество пар ключ-значение. На каждом уровне создается словарь Dictionaryc, > с символьными ключами и целочисленными значениями, соответствующими количеству раз использования символа в длинном слове.

Класс Word Level имеет также два удобных статических метода:

– MakeCharDict(): на основе полученной строки заполняет словарь charDict.

– CheckWordInLevel(): проверяет, можно ли составить заданное слово из символов, имеющихся в словаре charDict.

1. Создайте в папке__ Scripts новый сценарий на C# с именем WordLevel и введите следующий код. Обратите внимание, что класс WordLevel не наследует MonoBehaviour, поэтому его нельзя подключить ни к какому игровому объекту в виде компонента сценария и он не имеет функций StartCoroutineQ, SendMessage() и многих других, характерных для Unity:

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;
[System.Serializable] // Экземпляры WordLevel можно просматривать/изменять в инспекторе
public class WordLevel {// WordLevel НЕ наследует MonoBehaviour
public int levelNum;
public int longWordlndex;
public string word;
// Словарь со всеми буквами в слове
public Dictionary<char,int> charDict;
// Все слова, которые можно составить из букв в charDict p
ublic List<string> subWords;

// Статическая функция, подсчитывает количество вхождений символов в строку
// и возвращает словарь Dictionary<char,int> с этой информацией
static public Dictionary<char,int> MakeCharDict(string w) {
Dictionary<char,int> diet = new Dictionarycchar, int>();
char c;
if (diet.ContainsKey(c)) {
 dict[c]++;
} else {
 diet.Add (c,l); }
} return(dict);
}

// Статический метод, проверяет возможность составить слово
 // из символов в level.charDict
```

```
public static bool CheckWordInLevel(string str, WordLevel level) {
Dictionary<char,int> counts = new Dictionaryechar, int>();
for (int i=0; i<str.Length; i++) {
char c = str[i];
// Если charDict содержит символ c
if (level.charDict.ContainsKey(c)) {
 // Если counts еще не содержит ключа с символом c
 if (!counts.ContainsKey(c)) { /
/ ...добавить новый ключ со значением 1 counts.Add (c,1);
 } else {
// В противном случае прибавить 1 к текущему значению
counts[c]++; }

// Если число вхождений символа c в строку str
// превысило доступное количество в level.charDict
if (counts[c] > level.charDict[c]) {
// ... вернуть false
return(false); }
} else {
// Символ c отсутствует в level.word, вернуть false
return(false); }
} return(true);
}
}
```

2. Чтобы задействовать класс WordLevel, внесите следующие изменения в сценарий WordGame:

```
public class WordGame : MonoBehaviour {
static public WordGame S; // Одиночка

[Header("Set Dynamically")]
public GameMode mode = GameMode.preGame;
public WordLevel currLevel;

public void WordListParseComplete() {
currLevel = MakeWordLevel();
}
```

```
public WordLevel MakeWordLevel(int levelNum = -1) {
//a WordLevel level = new WordLevel();
if (levelNum == -1) {
 // Выбрать случайный уровень
level.longWordlndex=Random.Range(0,WordList.LONG_WORD_
COUNT);
} else {
// Код в эту ветку будет добавлен далее в этой главе
} level.levelNum = levelNum;
level.word = WordList.GET_LONG_WORD(level.longWordlndex);
level.charDict = WordLevel.MakeCharDict(level.word);
Startcoroutine( FindSubWordsCoroutine(level) );
return( level );
}
// Сопрограмма, отыскивающая слова, которые можно составить
на этом уровне
public IEnumerator FindSubWordsCoroutine(WordLevel level) {
level.subWords = new List<string>();
 string str;

List<string> words = WordList.GET_WORDS(); //

// Выполнить обход всех слов в WordList
for (int i=0; i<WordList.WORD_COUNT; i++) {
str = words[i];
// Проверить, можно ли его составить из символов в
level.charDict
if (WordLevel.CheckWordInLevel(str, level)) {
level.subWords.Add(str); }
// Приостановиться после анализа заданного числа слов в этом
кадре
 if (iXWordList.NUM-TO_PARSE_BEFORE_YIELD == 0) {
// приостановиться до следующего кадра
yield return null; } }

level.subWords.Sort ();
level.subWords = SortWordsByLength(level.subWords).ToList();
```

```
// Сопрограмма завершила анализ, поэтому вызываем
SubWordSearchComplete() SubWordSearchComplete();
}
public static IEnumerable<string> SortWordsBy-
Length(IEnumerable<string> ws) {
ws = ws.OrderBy(s => s.Length);
return ws; }

public void SubWordSearchComplete() {
mode = GameMode.levelPrep; }
}
```

3. Сохраните сцену! Если до сих пор вы ни разу не сохранили сцену и сделали это, только увидев этот пункт, значит, вам нужно чаще напоминать себе о необходимости сохранять сцену.

Компоновка экрана

После создания информационного представления уровня пришло время заняться созданием визуальных элементов на экране, представляющих большие плитки с буквами, которые можно использовать для составления слов, и маленькие – для отображения самих слов. Сначала создадим шаблон PrefabLetter, из которого будут создаваться экземпляры плиток с буквами

Создание PrefabLetter

Выполните следующие шаги, чтобы создать шаблон PrefabLetter:

1. В главном меню выберите пункт GameObject > 3D Object > Quad (Игровой объект > 3D объект > Квадрат). Переименуйте вновь созданный объект квадрата в PrefabLetter.

2. В главном меню выберите пункт Assets > Create > Material (Ресурсы > Создать > Материал). Дайте новому материалу имя LetterMat и поместите его в папку Materials & Textures.

3. Перетащите LetterMat на объект PrefabLetter в иерархии. Щелкните на PrefabLetter и выберите шейдер Unlit > Transparent для материала LetterMat.

4. Выберите текстуру Rounded Rect 256 для материала LetterMat.

5. Дважды щелкните по PrefabLetter в иерархии, и вы увидите в сцене квадрат с закругленными углами.

6. Щелкните правой кнопкой по PrefabLetter в иерархии и в контекстном меню выберите пункт 3D Object > 3D Text (3D объект > 3D Текст). В результате будет создан игровой объект New Text, вложенный в PrefabLetter.

7. Измените имя объекта New Text на 3D Text (с пробелом после «3D»).

8. Выберите 3D Text в иерархии и настройте его. Если буква W отобразится не в центре квадрата, значит, вы случайно ввели символ табуляции после W в поле Text (именно это случилось со мной, когда я работал над этим прототипом).

9. Перетащите PrefabLetter из иерархии в папку -Prefabs в панели Project (Проект) и удалите оставшийся экземпляр PrefabLetter из иерархии. Сохраните сцену.

Сценарий Letter

Теперь создадим для шаблона PrefabLetter отдельный сценарий на C#, который будет обслуживать настройки отображения символа, его цвет и многое другое.

1. Создайте в папке__Scripts новый сценарий на C# с именем Letter и подключите его к PrefabLetter.

2. Откройте сценарий в MonoDevelop и введите следующий код:

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

public class Letter : MonoBehaviour {
[Header("Set Dynamically")]
public TextMesh tMesh; // TextMesh отображает символ
public Renderer tRend; // Компонент Renderer объекта 3D Text. Он
// будет определять видимость символа
public bool big = false // Большие и малые плитки действуют
// по-разному
private char _c; // Символ, отображаемый на этой плитке
private Renderer rend;

void Awake() {
 tMesh = GetComponentInChildren<TextMesh>();
```

```
tRend = tMesh.GetComponent<Renderer>();
rend = GetComponent<Renderer>();
visible = false; }

// Свойство для чтения/записи буквы в поле _c, отображаемой объектом 3D Text
public char c {
get { return( _c ); }
set { _
c = value;
tMesh.text = _c.ToString(); }

// Свойство для чтения/записи буквы в поле _c в виде строки
public string str {
get { return( _c.ToString() ); }
set { c = value[0]; } }

// Разрешает или запрещает отображение 3D Text, что делает букву

// видимой или невидимой соответственно.
 public bool visible {
get { return( tRend.enabled ); }
 set { tRend.enabled = value; } }

// Свойство для чтения/записи цвета плитки
public Color color {
get { return(rend.material.color); }
set { rend.material.color = value; } }

// Свойство для чтения/записи координат плитки
 public Vector3 pos {
 set { transform.position = value; // Далее мы добавим дополнительный код } }
 }
```

Класс Wyrd: коллекция плиток с буквами

Класс Wyrd будет действовать как коллекция плиток с буквами, а его имя записано через у, чтобы отличить его от других экземпляров слова word, часто встречающихся в этом коде и в тексте книги. Wyrd – это еще один класс, который не наследует MonoBehaviour и который нельзя подключить к игровому объекту, но он содержит списки классов, подключенных к игровым объектам.

1. Создайте в папке__Scripts новый сценарий на C# с именем Wyrd.

2. Откройте сценарий Wyrd в MonoDevelop и введите следующий код:

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

public class Wyrd { // Wyrd не наследует MonoBehaviour
public string str; // Строковое представление слова
public List<Letter» letters = new List<Letter>();
public bool found = false; // Получит true, если игрок нашел это слово

// Свойство, управляющее видимостью компонента 3D Text каждой плитки Letter
public bool visible {
 get {
 if (letters.Count == 0)
return(false);
return(letters[0).visible); }
set {
foreach(Letter l in letters)
1.visible = value;
}
}
}

// Свойство для назначения цвета каждой плитке Letter
public Color color {
get {
 if (letters.Count == 0)
```

```
return(Color.black); return(letters[0].color); }
 set {
foreach( Letter 1 in letters) {
1.color = value; } } }

// Добавляет плитку в список letters
public void Add(Letter 1) {
 letters.Add(l);
 str += l.c.ToString(); }
```

Метод WordGame.Layout()

Метод Layout () сгенерирует экземпляры Wyrd и Letter, а также экземпляры больших плиток Letter, которые игрок сможет использовать для составления слов. Начнем с маленьких плиток, и на этом этапе разработки прототипа все буквы на плитках будут видимы (но в окончательной версии мы их скроем).

1. Добавьте в класс WordGame следующие строки:

```
public class WordGame : MonoBehaviour {
 static public WordGame S; // Объект-одиночка

[Header(’’Set in Inspector”)]
public GameObject prefabLetter;
public Rect wordArea = new Rect( -24, 19, 48, 28 );
public float letterSize = 1.5f;
public bool showAHWyrds = true;

 [Header("Set Dynamically")]
public GameMode mode = GameMode.preGame;
public WordLevel >  currLevel;
public List<Wyrd> wyrds;

private Transform letterAnchor, bigLetterAnchor;

void Awake() {
bigLetterAnchor = new GameObject("BigLetterAnchor").transform;
}

public void SubWordSearchComplete() {
```

```
 mode = GameMode.levelPrep;
Layout(); // Вызвать Layout() один раз после выполнения WordSearch }

void Layout() {
// Поместить на экран плитки с буквами каждого возможного
// слова текущего уровня
wyrds = new List<Wyrd>();

// Объявить локальные переменные, которые будут использоваться методом
GameObject go;
Letter lett;
string word;
Vector3 pos;
float left = 0;
 float columnwidth =3;
char c;
Color col;
Wyrd wyrd;

// Определить, сколько рядов плиток уместится на экране
 int numRows = Mathf.RoundToInt(wordArea.height/letterSize);

// Создать экземпляр Wyrd для каждого слова в level.subWords
for (int i=0; iccurrLevel.subWords.Count; i++) {
wyrd = new Wyrd();
word = currLevel.subWords[i];

// если слово длиннее, чем columnwidth, развернуть его
columnwidth = Mathf.Max( columnwidth, word.Length );

// Создать экземпляр PrefabLetter для каждой буквы в слове
for (int j=0; j<word.Length; j++) {
c = word[j]; // Получить j-й символ слова
go = Instantiate<GameObject>( prefabLetter );
go.transform.SetParent( letterAnchor );
lett = go.GetComponent<Letter>();
lett.c = c; // Назначить букву плитке Letter
```

```
// Установить координаты плитки Letter
pos = new Vector3(wordArea.x+left+j*letterSize, wordArea.y, 0);

// Оператор % помогает выстроить плитки по вертикали
pos.y -= (i%numRows)*letterSize;

lett.pos = pos;
wyrd.Add(lett);
}
if (showAHWyrds) wyrd.visible = true;
wyrds.Add(wyrd);

// Если достигнут последний ряд в столбце, начать новый столбец
if ( i%numRows == numRows-1 ) {
 left += ( columnwidth + 0.5f ) ♦ lettersize; }
}
}
}
```

2. Перед тем как щелкнуть по кнопке Play (Играть), перетащите шаблон PrefabLetter из панели Project (Проект) на поле prefabLetter компонента WordGame (Script) в объекте -MainCamera.

Добавление больших плиток с буквами

Следующим шагом добавим в Layout () размещение больших плиток с буквами вдоль нижнего края экрана.

1. Для этого добавьте следующий код:

```
public class WordGame : MonoBehaviour {
static public WordGame S; // Объект-одиночка

[Header("Set in Inspector”)]

public bool showAHWyrds = true;
public float bigLetterSize = 4f;
public Color bigColorDim = new Color( 0.8f, 0.8f, 0.8f );
public Color bigColorSelected = new Color( If, 0.9f, 0.7f );
public Vector3 bigLetterCenter = new Vector3( 0, -16, 0 );
```

```
 [Header("Set Dynamically")]

public List<Wyrd> wyrds;
 public List<Letter> bigLetters;
public List<Letter> bigLettersActive;

void Layout() {

// Создать экземпляр Wyrd для каждого слова в level.subWords
for (int i=0; iccurrLevel.subWords.Count; i++) {
}
// Поместить на экран большие плитки с буквами
 // Инициализировать список больших букв
bigLetters = new List<Letter>();
 bigLettersActive = new List<Letter>();

// Создать большую плитку для каждой буквы в целевом слов
e for (int i=0; iccurrLevel.word.Length; i++) {
// Напоминает процедуру создания маленьких плиток
c = currLevel.word[i];
go = Instantiate<GameObject>( prefabLetter );
go.transform.SetParent( bigLetterAnchor );
lett = go.GetComponent<Letter>();
lett.c = c;
go.transform.localscale = Vector3.one*bigLetterSize;

// Первоначально поместить большие плитки ниже края экрана
 pos = new Vector3( 0, -100, 0 );
lett.pos = pos;
lett.color = col;
 lett.visible = true; // Всегда true для больших плиток
lett.big = true; bigLetters.Add(lett);
} // Перемешать плитки
 bigLetters = ShuffleLetters(bigLetters);

// Вывести на экран
ArrangeBigLetters();
```

```
// Установить режим mode -- "в игре"
mode = GameMode.inLevel;
}
// Этот метод перемешивает элементы в списке List<Letter> и возвращает результат
List<Letter> ShuffleLetters(List<Letter> letts) {
 List<Letter> newL = new List<Letter>();
 int ndx;
while(letts.Count > 0) {
ndx - Random.Range(0^letts.Count);
newL.Add(letts[ndx]);
letts.RemoveAt(ndx); }
return(newL); }

// Этот метод выводит большие плитки на экран
void ArrangeBigLetters() {
// Найти середину для вывода ряда больших плиток с центрированием
// по горизонтали
float halfWidth = ( (float) bigLetters.Count )/2f - 0.5f;
Vector3 pos;
for (int i=0; i<bigLetters.Count; i++) {
pos = bigLetterCenter;
pos.x += (i-halfWidth)*bigLetterSize;
bigLetters[i].pos = pos; }
// bigLettersActive
 halfWidth = ( (float) bigLettersActive.Count )/2f - 0.5f;
for (int i=0; i<bigLettersActive.Count; i++) {
pos = bigLetterCenter;
pos.x += (i-halfWidth)*bigLetterSize;
pos.y += bigLetterSize*1.25f; bigLettersActive[i].pos = pos; } }
}
```

2. Теперь, кроме маленьких плиток, внизу на экране должен также появиться ряд больших плиток с буквами из целевого слова, следующими в случайном порядке.

Добавление интерактивности

В этой игре игрок должен иметь возможность собрать слово из имеющихся букв набольших плитках, нажимая клавиши с алфавитными символами, и подтвердить окончание ввода слова нажатием Return/Enter. Он также должен иметь возможность нажать Backspace/Delete, чтобы удалить букву в конце конструируемого слова, и перемешать оставшиеся невыбранные буквы нажатием клавиши пробела.

Взаимодействие с игроком реализовано в функции Update() класса WordGame и основывается на Input, inputstring, строке с клавишами, нажатыми в этом кадре.

1. Добавьте в WordGame метод Update() и вспомогательные методы:

```
public class WordGame : MonoBehaviour {

[Header(”Set Dynamically")]

public List<Letter> bigLettersActive;
 public string testWord;
private string uppercase = "ABCDEFGHIJKLMNOPQRSTUVWXYZ";

void ArrangeBigLetters() { ... }

void Update() {
/ Объявить пару вспомогательных переменных
 Letter ltr;
char c;

switch(mode{
case GameMode.inLevel;
// Выполнить обход всех символов, введенных игроком в этом кадре
foreach (char clt in Input.inputstring) {
 // Преобразовать clt в верхний регистр
c = System.Char.ToUpperlnvariant(cIt);
```

```
// Проверить, есть ли такая буква верхнего регистра
 if (uppercase.Contains(c)) { // Любая буква верхнего регистра
 // Найти доступную плитку с этой буквой в bigLetters
ltr = FindNextLetterByChar(c);
// Если плитка найдена
 if (ltr != null) {
// ... добавить этот символ в testWord и переместить
// соответствующую плитку Letter в bigLettersActive
testword += c.ToString(); /
/ Переместить из списка неактивных в список активных
// плиток
bigLettersActive.Add(ltr);
bigLetters.Remove(ltr);
 ltr.color = bigColorSelected;
// Придать плитке И активный вид ArrangeBigLetters();
// Отобразить плитки
 } }

if (c == ’\Ь’) { // Backspace
// Удалить последнюю плитку Letter из bigLettersActive
if (bigLettersActive.Count == 0) return;
if (testWord.Length > 1) {
 // Удалить последнюю букву из testword
testWord = testword.Substring(0,testWord.Length-1);
 } else {
 testWord = }

ltr = bigLettersActive[bigLettersActive.Count-l];
// Переместить из списка активных в список неактивных
// плиток
bigLettersActive.Remove(ltr);
 bigLetters.Add (ltr);
ltr.color = bigColorDim;
// Придать плитке неактивный вид
ArrangeBigLetters(); // Отобразить плитки
}

if (c == '\п' || c == ’\r') { // Return/Enter macOS/Windows
 // Проверить наличие сконструированного слова в WordLevel
```

```
Checkword(); }

if (c == ' ') { // Пробел
 // Перемешать плитки в bigLetters
bigLetters = ShuffleLetters(bigLetters);
 ArrangeBigLetters(); }

}
}
// Этот метод отыскивает плитку Letter с символом c в bigLetters.
// Если такой плитки нет, возвращает null.
Letter FindNextLetterByChar(char c) { /
/ Проверить каждую плитку Letter в bigLetters
foreach (Letter ltr in bigLetters) {
 // Если содержит тот же символ, что указан в c
if (ltr.c == c) { /
/ ... вернуть ее r
eturn(ltr); } }
return( null ); // Иначе вернуть null }

public void Checkword() {
// Проверяет присутствие слова testword в списке level.subWords
string subWord;
bool foundTestWord = false;

// Создать список List<int> для хранения индексов других слов,
// присутствующих в testword
 List<int> containedWords = new List<int>();

// Обойти все слова в currLevel.subWords
for (int i=0; i<currLevel.subWords.Count; i++) {
 // Проверить, было ли уже найдено Wyrd
 if (wyrds[ij.found) { //a
continue; }

subWord = currLevel.subWords[i];
 // Проверить, входит ли это слово subWord в testword
if (string.Equals(testword, subWord)) { // b
HighlightWyrd(i);
```

```
foundTestWord = true;
} else if (testword.Contains(subWord)) {
containedWords.Add(i); }
}

if (foundTestWord) { // Если проверяемое слово присутствует в списке
 // ...подсветить другие слова, содержащиеся в testword
int numContained = containedWords.Count;
int ndx;
 // Подсвечивать слова в обратном порядке
for (int i=0; i<containedWords.Count; i++) {
ndx = numContained-i-1;
 HighlightWyrd( containedWords[ndx] ); } }

// Подсвечивает экземпляр Wyrd
void HighlightWyrd(int ndx) {
// Активировать слово
wyrds[ndx].found = true; // Установить признак, что оно найдено
// Выделить цветом
wyrds[ndx].color = (wyrds[ndx].color+Color.white)/2f;
wyrds[ndx].visible = true; 11 Сделать компонент 3D Text видимым }

// Удаляет все плитки Letters из bigLettersActive
void ClearBigLettersActive() {
testword = // Очистить testword
foreach (Letter ltr in bigLettersActive) {
 bigLetters.Add(ltr); // Добавить каждую плитку в bigLetters
ltr.color = bigColorDim; // Придать ей неактивный вид
bigLettersActive.Clear(); // Очистить список
ArrangeBigLetters(); // Повторно вывести плитки на экран
}
}
```

2. Сохраните сценарий WordGame и вернитесь в Unity.

3. В инспекторе сбросьте флажок showAHWyrds в компоненте WordGame (Script) объекта _MainCamera и щелкните на кнопке Play (Играть).

1.6. Создание приключенческой игры

Прототип 7: DUNGEON DELVER

Игра Dungeon Delver, которую мы создадим, является упрощенным вариантом оригинальной игры Legend of Zelda для игровой приставки Nintendo Entertainment System. В своей преподавательской деятельности я не раз убеждался, что воссоздание старых игр очень помогает дизайнерам в их учебе, и The Legend Of Zelda всегда была одним из лучших образцов для повторения.

Настройка камер

Это первый проект, где мы будем использовать несколько камер. Первая, главная камера Main Camera, будет отображать основной игровой процесс, а вторая, камера пользовательского интерфейса GUI Camera, будет отображать графический пользовательский интерфейс (ГПИ) игры. Это позволит управлять каждой камерой независимо, что упростит реализацию ГПИ.

Панель Game (Игра)

Прежде чем заняться настройками камер, нужно подготовить панель Game (Игра).

В раскрывающемся списке соотношения сторон, в верхней части панели Game (Игра), выберите пункт 1080p (1920x1080). Если у вас этот пункт отсутствует в списке, выполните следующие шаги:

1. Щелкните по раскрывающемуся списку выбора соотношения сторон в верхней части панели Game (Игра) – второй раскрывающийся список слева – и щелкните по кнопке + внизу этого списка.

2. В открывшемся диалоге:

• в поле Label введите значение 1080p;

• в списке Type выберите Fixed Resolution;

• в поле W введите 1920;

• в поле H введите 1080.

3. Щелкните по кнопке ОК, чтобы сохранить настройки.

4. Выберите пункт 1080p (1920x1080) в раскрывающемся списке соотношения сторон.

Главная камера

Выберите Main Camera в иерархии и в инспекторе:

О Настройте компонент Transform: P:[23.5, 5, -10] R:[0, 0, 0] S:[1, 1,1].

В компоненте Camera:

• В поле Clear Flags выберите Solid Color;

• В поле Background установите черный цвет (RGBA:[0, 0, 0, 255]);

• В поле Projection выберите Orthographic;

• В поле Size введите 5.5;

• В поле Viewport Rect введите значения X:0, Y:0, W:0.8, H:l.

Камера пользовательского интерфейса

Выполните следующие шаги, чтобы создать камеру с именем GUI Camera.

1. Создайте новую камеру, выбрав в меню пункт GameObject > Camera (Игровой объект > Камера).

2. Переименуйте созданный объект Camera в GUI Camera.

3. Выберите GUI Camera в иерархии и в инспекторе:

• В поле Тад выберите Untagged – это гарантирует, что Camera. main будет продолжать ссылаться на Main Camera.

• Настройте компонент Transform: P:[-100, 0, -10] R:[0, 0, 0] S:[1,1,1].

• В компоненте Camera:

• В поле Clear Flags выберите Solid Color.

• В поле Background пока установите серый цвет (RGBA:[128,128,128,255]).

• В поле Projection выберите Orthographic.

• В поле Size введите 5.5.

• В поле Viewport Rect введите значения X:0.8, Y:0, W:0.2, H: 1.

Audio Listener – в сцене может присутствовать только один компонент Audio Listener, и в этой сцене он находится в Main Camera.

• Щелкните по кнопке с шестеренкой справа от компонента Audio Listener и в отвыберите (Удалить компонент).

В результате этих настроек панель Game (Игра) будет поделена на две меньших панели. Main Camera заполнит большую часть экрана, а GUI Camera – примерно 20%.

Добавление героя

Героем этой игры будет Дрей (Dray), рыцарь в доспехах. Для имитации 8-битной технологии оригинальной игры Legend of Zelda для героя предусмотрено отображение с четырех сторон, и его пере-

движение будет реализовано с применением анимационного эффекта. Позднее мы добавим также позу атакующего удара мечом или другим оружием.

Анимация героя

Чтобы создать первую анимацию, выполните следующие шаги:

1. Создайте в панели Project (Проект) новую папку с именем -Animations (Assets > Create > Folder (Ресурсы > Создать > Папку)).

2. В панели Project (Проект) выберите Dray_Walk-Oa и Dray_Walk_Ob в списке спрайтов Dray, в папке -Images. (Для этого щелкните на первом спрайте, нажмите клавишу Shift и, удерживая ее, щелкните на втором спрайте.)

3. Перетащите их вместе из панели Project (Проект) в панель Hierarchy (Иерархия) и отпустите кнопку мыши. Откроется диалог, в котором вам будет предложено ввести имя анимации.

4. Введите имя Dray_Walk_O.anim и сохраните анимацию в папке Animations.

В результате будут созданы:

В папке -Animations, в панели Project (Проект):

Dray_Walk_O

• Dray_Walk_Oa: аниматор, хранящий несколько разных анимаций и управляющий порядком их отображения.

В иерархии:

• Игровой объект с именем Dray_Walk_Oa с подключенным компонентом аниматора Dray_Walk_Oa. Это главный игровой объект, представляющий героя, которым будет управлять игрок.

Прежде чем продолжить, нужно внести кое-какие изменения.

5. Выберите игровой объект Dray_Walk_Oa в иерархии и переименуйте его в Dray.

6. Выберите аниматор Dray_Walk_Oa в папке -Animations в панели Project (Проект) и переименуйте его в Dray_Animator.

7. Щелкните дважды по Dray_Animator в панели Project (Проект).

8. Выберите объект Dray в иерархии и настройте координаты в компоненте Transform: P:[23.5, 5, 0]. В результате объект Dray переместится в центр поля зрения камеры Main Camera в панели Game (Игра).

9. Щелкните по кнопке Play (Играть).

Unity отобразит панель Game (Игра), и вы увидите, как Дрей быстро бежит на месте в центре открытой комнаты1. Кроме того, в панели Animator (Аниматор) вы должны увидеть, как заполняется и опустошается синий индикатор в оранжевом прямоугольнике Dray_Walk_0, отражающий ход выполнения. Это показывает, что анимация Dray_Walk_0 в настоящее время воспроизводится снова и снова.

10. Остановите игру и сохраните сцену.

Организация спрайтов по слоям

1. Выберите объект Dray в иерархии.

2. В его компоненте Sprite Renderer щелкните по раскрывающемуся списку Sorting Layer, определяющем слой сортировки, где в данный момент отображается значение Default, и выберите пункт Add Sorting Layer (Добавить слой сортировки). Откроется диспетчер тегов и слоев Tags & Layers с раскрытым разделом Sorting Layers (Слои сортировки).

3. Щелкните по кнопке 4- в правом нижнем углу списка Sorting Layers (Слои сортировки) и дайте вновь созданному слою имя Dray.

4. Повторите эту процедуру и создайте еще три слоя: Ground, Enemies и Items.

5. Разместите слои в таком порядке, сверху вниз: Ground, Default, Enemies, Dray, Items. При таком размещении слой Ground всегда будет отображаться позади всех остальных, Enemies – между слоями Default и Dray, Dray – поверх всех слоев, кроме Items, и Items – поверх всех других слоев.

6. Выберите объект Dray в иерархии и в его компоненте Sprite Renderer, в инспекторе, выберите в раскрывающемся списке Sorting Layer пункт Dray. (Не пытайтесь изменить значение в списке Layer (Слой) в верхней части панели Inspector (Инспектор), – он определяет физический слой.)

7. Выберите шаблон Tile в папке -Prefabs, в панели Project (Проект), и в его компоненте Sprite Renderer, в инспекторе, выберите в раскрывающемся списке Sorting Layer пункт Ground. В результате все экземпляры Tile всегда будут отображаться позади любых других спрайтов.

8. Сохраните сцену

По мере добавления других объектов в игру мы также будем определять для них свой слой сортировки.

Настройка анимации Dray_Walk_O

Прямо сейчас Дрей слишком быстро перебирает ногами и пока может бежать только вправо. Исправим оба этих недостатка в панели Animation (Анимация).

1. Щелкните по кнопке с тремя маленькими горизонтальными полосками в правом верхнем углу панели Game (Игра). В открывшемся меню выберите пункт Add Tab > Animation (Добавить вкладку > Анимация), чтобы открыть панель Animation (Анимация) как вкладку в панели Game (Игра).

2. Выберите Dray в иерархии и раскройте раздел Dray : Sprite в панели Animation (Анимация), щелкнув на пиктограмме с треугольником рядом с названием раздела.

3. Введите в поле Samples число 10 и нажмите Return/Enter. Согласно этому значению, Дрей будет делать примерно 5 шагов в секунду.

4. Чтобы увидеть, как это будет выглядеть на экране:

• Перейдите в панель Scene (Сцена), щелкнув на вкладке Scene (Сцена), чтобы отобразить ее поверх панели Animator (Аниматор).

• Щелкните дважды по объекту Dray в иерархии, чтобы переместить его в центр сцены.

• Щелкните по кнопке Animation Play (Запустить анимацию), которая находится в панели Animation (Анимация), непосредственно над переключателем анимаций.

5. Щелкните по кнопке Animation Play (Запустить анимацию), чтобы остановить воспроизведение анимации.

Состояния анимаций в аниматоре

Если вернуться в панель Animator (Аниматор), выбрав в главном меню Unity пункт Window > Animator (Окно > Аниматор), можно увидеть четыре состояния объекта Dray. Это компонент Animator игрового объекта Dray в иерархии, и он объединяет игровой объект Dray с четырьмя анимациями Dray_Walk_#.

Перемещение Дрея

Прежде чем начать перемещать объект Drey, добавим в него компонент твердого тела Rigidbody.

1. Выберите игровой объект Dray в иерархии.

2. Подключите к нему компонент Rigidbody (Component > Physics > Rigidbody (Компонент > Физика > Твердое тело)).

Снимите флажок Use Gravity.

В разделе Constraints:

• Установите флажок Freeze Position Z.

• Установите флажок Freeze Rotation X, Y и Z.

3. Сохраните сцену.

Теперь добавим в объект Dray сценарий, управляющий движением.

4. Создайте в папке__Scripts новый сценарий на C# с именем Dray и подключите его к игровому объекту Dray в иерархии.

5. Откройте сценарий Dray в MonoDevelop и введите следующий код:

```
using System.Collections;
using System.Collections.Generic;
 using UnityEngine;
public class Dray : MonoBehaviour {
 [Header("Set in Inspector")]
public float speed =5;

[Header("Set Dynamically")]
public int dirHeld = -1; // Направление3 соответствующее
 // удерживаемой клавише
 private Rigidbody rigid;
void Awake () {
rigid = GetComponent<Rigidbody>(); }

void Update () {
dirHeld=-1;
if ( Input.GetKey(KeyCode.LeftArrow ) ) dirHeld = 2;
if ( Input.GetKey(KeyCode.DownArrow ) ) dirHeld = 3;

Vector3 vel = Vector3.zero;
switch (dirHeld) {
```

```
case 0:
vel = Vector3.right;
break;
case 1:
vel = Vector3.up;
break;
case 2: v
el = Vector3.1eft;
break;
case 3:
vel = Vector3.down;
break; }
rigid.velocity = vel * speed;
}
}
```

6. Сохраните сценарий Dray, вернитесь в Unity и щелкните по кнопке Play (Играть). Теперь вы сможете перемещать Дрея по сцене, нажимая клавиши со стрелками.

Анимация движения Дрея

Снова откройте сценарий Dray в MonoDevelop и добавьте в него следующий код:

```
public class Dray : MonoBehaviour {

private Rigidbody rigid;
private Animator anim;

void Awake () {
 rigid = GetComponent<Rigidbody>();
 anim = GetComponent<Animator>(); 1

void Update () {

rigid.velocity = vel * speed;
// Анимация
if (dirHeld == -1) {
 anim.speed = 0;
 } else {
```

```
anim.CrossFade( "Dray_Walk_"+dirHeld, 0 ); anim.speed = 1; }
}
}
```

Функция anim.CrossFade() требует от аниматора anim переключиться на новую анимацию и дает на переход 0 секунд. Если anim уже отображает требуемую анимацию, данный вызов не произведет никакого эффекта.

Сохраните сценарий Dray и вернитесь в Unity. Запустите игру и посмотрите, как при движении Дрея в разных направлениях воспроизводятся разные анимационные эффекты.

Создание анимации атаки

Чтобы создать анимацию атаки, выполните следующие шаги:

1. Выберите объект Dray в иерархии и перейдите в панель Animation (Анимация), выбрав в меню Unity пункт Window > Animation (Окно > Анимация).

2. В списке переключателя анимаций выберите пункт Create New Clip (Создать новый клип) и сохраните новый клип с именем Dray_Attack_O в папке Animations.

3. Выберите Dray_Attack_O в списке переключателя анимаций.

4. Введите в поле Samples число 10 и нажмите Return/Enter.

5. В панели Project (Проект) выберите спрайт Dray_Attack_O в изображении Dray, в папке -Images.

6. Перетащите спрайт Dray_Attack_O из панели Project (Проект) в область плана- графика в панели Animation (Анимация).

7. Повторите шаги со 2-го по 6-й и создайте анимации с именами Dray_Attack_l, Dray_Attack_2 и Dray_Attack_3, использовав соответствующие спрайты.

Теперь в панели Animator (Аниматор) должно появиться четыре новые анимации атаки.

Программирование анимаций атаки

Откройте сценарий Dray в MonoDevelop и добавьте в начало новые строки с перечислением и определениями полей.

```
public class Dray : MonoBehaviour {
public enum eMode { idle, move, attack, transition }

[Header("Set in Inspector")]
public float speed = 5;
```

```
public float attackDuration = 0.25f; // // Продолжительность атаки в секундах
public float attackDelay = 0.5f; // Задержка между атаками

 [Header("Set Dynamically")]
 public int dirHeld = -1; // Направление, соответствующее // удерживаемой клавише
public int p facing = 1; // Направление движения Дрея
ublic eMode mode = eMode.idle

private float timeAtkDone = 0;
private float timeAtkNext = 0;

private Rigidbody rigid;
private Animator anim;
}
```

2. Появление новых полей существенно меняет реализацию метода Update(), поэтому в следующем листинге он приводится целиком. Он реализует те же идеи, что были описаны выше, но немного иначе.

```
void Update () {
 //-–Обработка ввода с клавиатуры и управление режимами eMode
dirHeld = -1;
for (int i=0; i<4; i++) { i
f ( Input.GetKey(keys[i]) ) dirHeld = i; }

// Нажата клавиша атаки
if (Input.GetKeyDown(KeyCode.Z) && Time.time >= timeAtkNext) {
mode = eMode.attack;
 timeAtkDone = Time.time + attackDuration;
timeAtkNext = Time.time + attackDelay; }

// Завершить атаку, если время истекло
 if (Time.time >= timeAtkDone) {
mode = eMode.idle; }
```

```
// Выбрать правильный режим, если Дрей не атакует
 if (mode •= eMode.attack) {
if (dirHeld == -1) {
mode = eMode.idle;
} else {
facing = dirHeld;
mode = eMode.move; } }

//-–Действия в текущем режиме–
Vector3 vel = Vector3.zero;
switch (mode) {
 case eMode.attack:
anim.CrossFade( "Dray_Attack_"+facing, 0 );
anim.speed = 0; b
reak;
case eMode.idle:
anim.CrossFade( "Dray_Walk_"+facing, 0 );
anim.speed = 0;
break;
case eMode.move:
vel = directions[dirHeld]; anim.CrossFade( "Dray_Walk_”+facing, 0
);
 anim.speed = 1; break;
}

rigid.velocity = vel ♦ speed;
}
```

Меч Дрея

Главное оружие Дрея – меч, которым он может наносить удары в любом направлении. Текстура со спрайтами меча была импортирована в составе пакета, в начале этой главы.

1. Выберите объект Dray в иерархии. Щелкните по нему правой кнопкой мыши и выберите в контекстном меню пункт Create Empty (Создать пустой). Переименуйте вновь созданный пустой игровой объект в SwordController.

2. Настройте компонент Transform объекта SwordController: P:[0, 0, 0] R:[0, 0, 0] S:[1,1,1].

3. Раскройте текстуру Swords в папке -Images, щелкнув на пиктограмме с треугольником.

4. Перетащите спрайт Swords_0 из панели Project (Проект) в панель Hierarchy (Иерархия). Вложите его в объект SwordController (который сам вложен в объект Dray).

В иерархии переименуйте экземпляр Swords_0 в Sword.

Настройте компонент Transform объекта Sword: P:[0.75, 0, 0] R:[0, 0, 0] S:[1, 1,1].

В раскрывающемся списке Sorting Layer компонента Sprite Renderer объекта Sword выберите слой сортировки Enemies

5. Добавьте в Sword компонент Box Collider (Component > Physics > Box Collider (Компонент > Физика > Коробчатый коллайдер)). Он по умолчанию должен приобрести оптимальные размеры, но если это не так, введите в поля раздела Size значения [1, 0.4375, 0.2].

Установите флажок Is Trigger в коллайдере.

6. Выберите SwordController в иерархии.

Щелкните по кнопке Add Component (Добавить компонент) в инспекторе.

В открывшемся меню выберите пункт New Script (Новый сценарий).

Дайте новому сценарию имя SwordController.

В панели Project (Проект) перетащите сценарий SwordController в папку Scripts.

7. Откройте сценарий SwordController и введите следующий код:

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

public class SwordController : MonoBehaviour {
private GameObject sword;
private Dray dray;

void Start () {
sword = transform.Find("Sword")«gameObject;
dray = transform.parent.GetComponent<Dray>();
// Деактивировать меч
sword.SetActive(false); }
```

```
void Update () {
transform.rotation = Quaternion.Euler( 0, 0, 90*dray.facing );
sword.SetActive(dray.mode == Dray.eMode.attack); // d }
}
```

8. Сохраните все сценарии в MonoDevelop1, вернитесь в Unity и щелкните по кнопке Play (Играть).

Враги: скелеты

Скелеты – основные враги Дрея. Подобно скелетам в The Legend of Zelda, в этой игре скелеты тоже будут хаотически блуждать по комнатам, в которых они находятся. Скелеты могут проходить друг сквозь друга и наносить урон Дрею.

Изображения скелетов

Чтобы получить изображения скелетов:

1. Выберите два спрайта с именами Skeletos_0 и Skeletos.l (вы найдете их в панели Project (Проект) > _ Images > Skeletos).

2. Перетащите их оба в иерархию. В результате будет создана новая анимация. Сохраните ее с именем Skeletos.anim в папке Animations.

3. В иерархии переименуйте игровой объект Skeletos_0 в Skeletos.

4. Настройте компонент Transform объекта Skeletos: P:[19, 7, 0].

5. Подключите к объекту Skeletos компонент Rigidbody (Component > Physics > Rigidbody (Компонент > Физика > Твердое тело)).

Снимите флажок Use Gravity.

В разделе Constraints:

• Установите флажок Freeze Position Z;

• Установите флажок Freeze Rotation X, Y и Z.

6. Добавьте в Skeletos сферический коллайдер (Component > Physics > Sphere Collider (Компонент > Физика > Сферический коллайдер)).

7. В раскрывающемся списке Sorting Layer компонента Sprite Renderer объекта Skeletos выберите слой сортировки Enemies.

8. Выберите объект Skeletos в иерархии и перейдите в панель Animation (Анимация), выбрав в меню Unity пункт Window > Animation (Окно > Анимация).

9. Введите в поле Samples число 5 и нажмите Return/Enter.

10. Сохраните сцену.

Щелкните по кнопке Play (Играть), и в комнате с Дреем вы должны увидеть скелет, бегущий на месте.

Базовый класс врагов Enemy

Все враги в игре Dungeon Delver бу дут наследовать общий базовый класс Enemy.

1. Создайте в папке__Scripts новый сценарий на C# с именем Enemy.

2. Откройте его в MonoDevelop и введите следующий код:

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;
public class Enemy : MonoBehaviour {
protected static Vector3[] directions = new Vector3[] {
Vector3.right, Vector3.up, Vector3.1eft, Vector3.down };

[Header("Set in Inspector: Enemy”)]
public float maxHealth =1; //

[Header("Set Dynamically: Enemy")]
public float health;

protected Animator anim;
protected Rigidbody rigid;
protected SpriteRenderer sRend;

protected virtual void Awake() {
health = maxHealth;
anim = GetComponent<Animator>();
rigid = GetComponent<Rigidbody>();
sRend = GetComponent<SpriteRenderer>(); }
}
```

Skeletos – подкласс Enemy

Создайте подкласс Skeletos, выполнив следующие шаги:

1. Создайте в папке__Scripts новый сценарий на C# с именем Skeletos.

2. Подключите его к игровому объекту Skeletos в иерархии.
3. Откройте сценарий в MonoDevelop и введите следующий код:

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

public class Skeletos : Enemy {
[Header("Set in Inspector: Skeletos")]
public int speed = 2;
public float timeThinkMin = If;
public float timeThinkMax = 4f;

 [Header("Set Dynamically: Skeletos")]
public int facing = 0;
public float timeNextDecision = 0;

void Update () {
if (Time.time >= timeNextDecision) { //
DecideDirection(); }
// Поле rigid унаследовано от класса Enemy и инициализируется
Enemy.Awake() rigid.velocity = directions[facing] * speed; }

void DecideDirection() { // d
facing = Random.Range(0,4);
timeNextDecision = Time.time + Ran-
dom.Range(timeThinkMin,timeThinkMax);
}
```

4. Сохраните все сценарии в MonoDevelop

Вы должны увидеть скелет, блуждающий по комнате и свободно проходящий сквозь стены! Нам нужен сценарий, который будет удерживать скелеты и других врагов в их комнатах.

Сценарий InRoom

Сценарий InRoom реализует несколько вспомогательных служб. Чтобы определить попытку скелета выйти из комнаты, нужно знать, где в комнате он находится, а поскольку все комнаты в подземелье имеют одинаковые размеры, сделать это легко.

1. Создайте в папке__Scripts новый сценарий на C# с именем InRoom.

2. Откройте его в MonoDevelop и введите следующий код:

```
public class InRoom : MonoBehaviour {
using System.Collections;
using System.Collections.Generic;
 using UnityEngine;
static public float ROOM_W = 16;
static public float ROOM_H = 11;
static public float WALL.T = 2;

// Местоположение этого персонажа в локальных координатах комнаты
public Vector2 roomPos {
get { Vector2 tPos = transform.position;
tPos.x %= ROOM_W;
tPos.y %= ROOM_H;
return tPos; }
set {
Vector2 rm = roomNum;
rm.x *= ROOM_W;
rm.y *= ROOM.H;
rm += value; transform.position = rm; } }

// В какой комнате находится этот персонаж?
public Vector2 roomNum {
get {
Vector2 tPos = transform.position;
tPos.x = Mathf.Floor( tPos.x / ROOM.W );
 tPos.y = Mathf.Floor( tPos.y / ROOM_H );
return tPos;
} set {
Vector2 rPos = roomPos;
Vector2 rm = value;
 rm.x *= ROOM_W;
rm.y *= ROOM_H;
 transform.position = rm + rPos; }
```

```
}
}
```

Удержание игровых объектов внутри комнат

Как упоминалось выше, нам нужно удержать скелет внутри комнаты. Для этого добавим метод LateUpdate(), который будет проверять, не вышел ли скелет за пределы основной области комнаты. Метод LateUpdate() вызывается в каждом кадре после вызова метода Update() всех игровых объектов1. Метод LateUpdate() отлично подходит для выполнения заключительных операций, таких как возврат увлекшихся персонажей в комнату, где они должны находиться.

1. Подключите сценарий InRoom к объекту Skeletos в иерархии.

2. Откройте сценарий InRoom в MonoDevelop и добавьте следующие строки:

```
public class InRoom : MonoBehaviour {
 static public float ROOM_W = 16;
static public float ROOM_H = 11;
static public float WALL_T = 2;

[Header("Set in Inspector")]
public bool keepInRoom = true;
 public float gridMult = 1;

void LateUpdate() {
if (keepInRoom) {
Vector2 rPos = roomPos;
rPos.x = Mathf.Clamp( rPos.x, WALL_T, ROOM_W-1-WALL_T );
 rPos.y = Mathf.Clamp( rPos.y, WALL_T, ROOM_H-1-WALL_T );
roomPos = rPos; } }

// Местоположение этого персонажа в локальных координатах комнаты
public Vector2 roomPos (...)
}
```

3. Сохраните сценарий In Room и вернитесь в Unity, чтобы протестировать его. Теперь скелет все так же будет блуждать по комнате, но уже не сможет проникать сквозь стены, как прежде.

Столкновения с плитками

Информация о местоположениях всех плиток в подземелье хранится в текстовом файле DelverData, изображения хранятся в файле DelverTiles.

Чтобы воспользоваться информацией из DelverCollisions, добавим коллайдер в шаблон Tile.

1. Выберите игровой объект Tile в папке _Prefabs в панели Project (Проект).

2. Подключите к нему коллайдер Box Collider (Component > Physics > Box Collider (Компонент > Физика > Коробчатый коллайдер)).

Определение столкновений с плитками

Далее, добавим несколько строк в сценарии TileCamera и Tile, которые будут использовать информацию из DelverCollisions.

1. Откройте сценарий TileCamera в MonoDevelop и добавьте следующие строки:

```
public class TileCamera : MonoBehaviour {
static public int W, H;
static private int[,] MAP;
static public Sprite[] SPRITES;
static public Transform TILE_ANCHOR;
static public Tile[j] TILES; s
tatic public string COLLISIONS;

[Header("Set in Inspector")]

void Awake() {
COLLISIONS = Utils.RemoveLineEndings( mapCollisions.text );
LoadMap(); }
}
```

2. Сохраните сценарий TileCamera.

3. Откройте сценарий Tile и добавьте следующие строки:

```
public class Tile : MonoBehaviour {

[Header("Set Dynamically")]
public int x;
 public int y;
 public int tileNum;
```

```
private BoxCollider bColl

void Awake() {
bColl = GetComponent<BoxCollider>();
}
Public void SetTile(…)
GetComponent<SpriteRenderer>().sprite = TileCam-
era.SPRITES[tileNum];
SetCollider();

// Настроить коллайдер для этой плитки void SetCollider() {
// Получить информацию о коллайдере из Collider
DelverCollisions.txt
bColl.enabled = true;
char c = TileCamera.COLLISIONS[tileNum];
switch (c) {
case 'S': // Вся плитка
bColl.center = Vector3.zero;
bColl.size = Vector3.one;
break;
case 'W’: // Верхняя половина
bColl.center = new Vector3( 0, 0.25f, 0 );
bColl.size = new Vector3( 1, 0.5f, 1 );
break;
case ’A’: // Левая половина
bColl.center = new Vector3( -0.25f, 0, 0 );
 bColl.size = new Vector3( 0.5f, 1, 1 );
break; case *D': // Правая половина
bColl.center = new Vector3( 0.25f, 0, 0 );
 bColl.size = new Vector3( 0.5f, 1, 1 );
break;

}
```

4. Сохраните сценарий Tile и вернитесь в Unity.

Добавления коллайдера в объект Dray

Наконец, чтобы увидеть результаты столкновений с отдельными плитками, добавим коллайдер в объект Dray.

1. Выберите объект Dray в иерархии.

2. Добавьте в него сферический коллайдер (Component > Physics > Sphere Collider (Компонент > Физика > Сферический коллайдер)).

Введите в поле Radius коллайдера значение 0.4.

Сохраните сцену и запустите ее. Подвигайте главного персонажа и посмотрите, что получится, если заставить его пройти сквозь стену. В отличие от скелетов, которые принудительно удерживаются в комнате, главный персонаж должен иметь возможность выходить в дверные проемы. Однако наблюдаются две проблемы:

Довольно сложно попасть прямо в дверной проем.

Фактически главный персонаж не переходит в другие комнаты.

Решим их по очереди.

Интерфейс IFacingMover

На самом деле нам нужно, чтобы не только Дрей, но и все создания в Dungeon Delver выравнивались по сетке в процессе движения, и в конечном счете мы должны применить будущий сценарий GridMove ко всем. И Дрей, и скелеты имеют некоторые сходные черты, но единственным общим предком для классов Dray и Skeletos является класс MonoBehaviour.

Интерфейс IFacingMover, который мы реализуем здесь, очень прост и его легко можно применить к обоим классам, Skeletos и Dray.

1. Создайте в папке__Scripts новый сценарий на C# с именем IFacingMover (обычно интерфейсам даются имена, начинающиеся с буквы I).

2. Откройте сценарий IFacingMover в MonoDevelop и введите следующий код. Обратите внимание, что интерфейс IFacingMover не наследует MonoBehaviour и интерфейс – это не класс:

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;
}
public interface IFacingMover {
 int GetFacing();
bool moving { get; }
float GetSpeed();
float gridMult { get; }
Vector2 roomPos { get; set; }
Vector2 roomNum { get; set; }
```

Vector2 GetRoomPosOnGrid(float mult = -1); II f

Реализация интерфейса IFacingMover в классе Dray

Выполните следующие шаги, чтобы реализовать интерфейс IFacingMover в классе Dray:

1. Откройте сценарий Dray и добавьте следующий код, реализующий интерфейс IFacingMover.

```
// apublic class Dray : MonoBehaviour, IFacingMover {

private Rigidbody rigid;
private Animator anim;
private InRoom inRm;

void Awake () {
rigid = GetComponent<Rigidbody>();
 anim = GetComponent<Animator>();
inRm = GetComponent<InRoom>(); }

void Update () { ... }


// Реализация интерфейса IFacingMover
public int GetFacing() {
 return facing; }

public bool moving {
get {
return (mode == eMode.move); 1 }

public float GetSpeed() {
return speed; }


public float gridMult {
get { return inRm.gridMult; } }

public Vector2 roomPos {
get { return inRm.roomPos; }
set { inRm.roomPos value; }
```

```
public Vector2 roomNum {
get { return inRm.roomNum; }
set { inRm.roomNum = value; } }

public Vector2 GetRoomPosOnGrid(
float mult = -1 ) {
return inRm.GetRoomPosOnGrid( mult ); }
}
```

2. Сохраните сценарий Dray и вернитесь в Unity, чтобы убедиться, что компиляция выполнилась без ошибок.

Может показаться, что мы проделали массу лишней работы, но посмотрите, что получится после реализации того же интерфейса в классе Skeletos.

3. Откройте сценарий Skeletos и добавьте следующие строки.

```
public class Skeletos : Enemy, IFacingMover {

public float timeNextDecision = 0;
private InRoom inRm;
protected override void Awake () {
base.AwakeO;
 inRm = GetComponent<InRoom>(); }

void Update () { ... }

void DecideDirection() { ... }

public int GetFacing() { facing;
}
public bool moving { get { return true; } }
public float GetSpeed() {
return speed; }

public float gridMult {
 get { return inRm.gridMult; } }

public Vector2 roomPos {
get { return inRm.roomPos; }
```

```
 set { inRm.roomPos = value; } }

public Vector2 roomNum {
get { return inRm.roomNum; }
set { inRm.roomNum = value; } }

public Vector2 GetRoomPosOnGrid( float mult = -1 ) {
return inRm.GetRoomPosOnGrid( mult ); }
}
.
```

4. Сохраните все сценарии в MonoDevelop и вернитесь в Unity.

Теперь экземпляры обоих классов, Dray и Skeletos, можно обрабатывать одним и тем же кодом, как экземпляры IFacingMover, и не нужно писать код отдельно для каждого класса. Чтобы убедиться в этом, реализуем сценарий Grid Move.

Переход из комнаты в комнату

Теперь, когда Дрей без труда попадает в створ дверей, пришла пора направить его в другие комнаты подземелья. Так как Дрей – единственный персонаж, который может переходить из комнаты в комнату, большую часть необходимого для этого кода можно включить в класс Dray; однако глобальная информация о комнатах, такая как местоположение дверей и общий размер карты, все еще должна находиться в классе In Room.

1. Откройте сценарий InRoom в MonoDevelop и введите следующий код:

```
public class InRoom : MonoBehaviour {
static public float ROOM_W = 16;
static public float ROOM_H = 11;
static public float WALL_T = 2;

static public int MAX_RM_X = 9;
 static public int MAX_RM_Y = 9;

static public Vector2[] DOORS = new Vector2[] {
new Vector2(14j 5), Vector2(7.5f 9),
new Vector2(7.5f, 1)
```

```
[Header("Set in Inspector”)]
public bool keepInRoom = true;
}
```

2. Сохраните сценарий InRoom.

3. Откройте сценарий Dray в MonoDevelop и введите следующий код:

```
public class Dray : MonoBehaviour, IFacingMover {

[Header("Set in Inspector")]

public float attackDelay = 0.5f; // Задержка между атаками
public float transitionDelay = 0.5f; // Задержка перехода между // комнатами

[Header("Set Dynamically")]

private float p timeAtkDone = 0;
rivate float timeAtkNext = 0;

private float transitionDone = 0;
private Vector2 transitionPos;
private Rigidbody rigid;

void Update () {
if ( mode == eMode.transition )
rigid.velocity = Vector3.zero;
anim.speed = 0;
roomPos = transitionPos; // Оставить Дрея на месте
if (Time.time < transitionDone) return;
 // Следующая строка выполнится, только если Time.time >= transitionDone mode = eMode.idle; }
//-–Обработка ввода с клавиатуры и управление режимами eMode–
dirHeld = -1
}
void LateUpdate()
// Получить координаты узла сетки, с размером ячейки
// в половину единицы, ближайшего к данному персонажу
Vector2 rPos = GetRoomPosOnGrid( 0.5f ); // Размер ячейки в пол-
```

```
// Персонаж находится на плитке с дверью?
int doorNum;
for (doorNum=0; doorNum<4; doorNum++) {
if (rPos == InRoom.DOORS[doorNum]) {
 break; } }

if ( doorNum > 3 || doorNum != facing ) return;
// Перейти в следующую комнату
Vector2 rm = roomNum;
switch (doorNum) {
case 0:
rm.x += 1;
break;
case 1:
rm.y += 1;
break;
case 2:
rm.x -= 1;
break;
case 3:
rm.y -= 1;
break; }

// Проверить, можно ли выполнить переход в комнату rm
 if (rm.x >= 0 && rm.x <= InRoom.MAX_RM_X) {
 if (rm.y >=0 && rm.y <= InRoom.MAX_RM_Y) {
roomNum = rm;
transitionPos = InRoom.DOORS[ (doorNum+2) % 4 ];
 roomPos = transitionPos;
mode = eMode.transition;
 transitionDone = Time.time + transitionDelay; } }
}
// Реализация интерфейса IFacingMover
public int GetFacingO { ... }
```

4. Сохраните все сценарии в MonoDevelop, вернитесь в Unity и щелкните на кнопке Play (Играть).

Перемещение камеры вслед за Дреем

Теперь, когда Дрей получил возможность переходить из комнаты в комнату, мы должны заставить камеру следовать за ним.

1. Создайте в папке__Scripts новый сценарий на C# с именем CamFollowDray.

2. Подключите сценарий CamFollowDray к главной камере Main Camera в иерархии.

3. Откройте CamFollowDray в MonoDevelop и введите следующий код:

```
using System.Collections;
 using System.Collections.Generic;
using UnityEngine;

public class CamFollowDray : MonoBehaviour {
 static public bool TRANSITIONING = false;

public InRoom draylnRm;
public float transTime = 0.5f;

private Vector3 P0, Pl;

private InRoom inRm;
private float transStart;

void Awake() {
inRm = GetComponent<InRoom>(); }

void Update () {
if (TRANSITIONING) {
float u = (Time.time - transStart) / transTime;
if (u >= 1) {
u = 1;
TRANSITIONING = false; }
 transform.position = (l-u)*p0 + u*pl;
} else {
if (draylnRm.roomNum != inRm.roomNum) {
TransitionTo( draylnRm.roomNum ); }
```

```
void TransitionTo( Vector2 rm ) {
p0 = transform.position;
inRm.roomNum = rm;
pl = transform.position + (Vector3.back ♦ 10);
transform.position = p0;

transStart = Time.time;
TRANSITIONING = true;
}
}
```

4. Сохраните сценарий CamFollowDray и вернитесь в Unity.

5. Выберите главную камеру Main Camera в иерархии.

6. Подключите к ней сценарий InRoom.

В инспекторе снимите флажок keepInRoom.

7. В инспекторе присвойте полю draylnRoom компонента CamFollowDray главной камеры Main Camera ссылку на игровой объект Dray. Это позволит компоненту CamFollowDray получить ссылку на компонент InRoom, подключенный к объекту Dray.

8. Сохраните сцену и щелкните на кнопке Play (Играть).

Теперь должна появиться возможность перемещаться между тремя нижними комнатами в подземелье, но теперь встает проблема запертой двери в средней комнате.

Отпирание дверей

Интерфейс IKeyMaster

1. Создайте в папке__Scripts новый сценарий на C# с именем IKeyMaster.

2. Откройте сценарий IKeyMaster в MonoDevelop и введите следующий код:

```
using System.Collections;
using System.Collections.Generic;
 using UnityEngine;
public interface IKeyMaster {
int keyCount { get; set; }
int GetFacingQ; }
```

3. Сохраните сценарий IKeyMaster и откройте сценарий Dray. Добавьте в него следующие строки,.

```
public class Dray : MonoBehaviour, IFacingMover, IKeyMaster {
[Header("Set Dynamically")]
```

```
 public int dirHeld = -1; // Направление, соответствующее
 // удерживаемой клавише
public int facing = 1; // Направление, куда смотрит Дрей
public eMode mode = eMode.idle;
public int numKeys = 0;

private float timeAtkDone = 0;

// Реализация интерфейса IFacingMover
public int GetFacingO {
return facing; }

public Vector2 GetRoomPosOnGrid(
float mult = -1 ) {
 return inRm.GetRoomPosOnGrid( mult );
}

// Реализация интерфейса IKeyMaster
public int keyCount {
 get { return numKeys; }
 set { numKeys = value; } }
}
```

4. Сохраните сценарий Dray. Теперь можно реализовать класс Gatekeeper.

Класс GateKeeper

Класс GateKeeper отпирает двери, заменяя плитки с запертыми дверьми плитками с отпертыми дверьми.

Реализация простой защиты для TileCamera.MAP

Прямые манипуляции статическим общедоступным массивом MAP в TileCamera из класса Tile – не самое лучшее решение. Именно поэтому массив TileCamera. MAP объявлен скрытым (private), а для работы с ним добавлены методы GET_MAP() и SET_MAP(), которые обеспечивают безопасную обработку значений eX и eУ за пределами MAP и тем самым предотвращают появление исключений IndexOutOfRangeException.

Изменение TileCamera.MAP в Tile.SetTile()

Откройте сценарий Tile в MonoDevelop и внесите в метод SetTile() следующие изменения, выделенные жирным:

```
public class Tile : MonoBehaviour {

public void SetTile(int eX, int eY, int eTileNum = -1) {

if (eTileNum == -1) {
eTileNum = TileCamera.GET_MAP[x,y];
} else {
TileCamera.SET_MAP(Xj y, eTileNum); // Заменить плитку, если необходимо } tileNum = eTileNum;
}
}
```

Реализация сценария Gatekeeper

Создайте сценарий Gatekeeper, выполнив следующие шаги:

1. Создайте сценарий Gatekeeper
2. Подключите его к игровому объекту Dray в иерархии.
3. Откройте сценарий GateKeeper в MonoDevelop и введите следующий код:

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;
public class GateKeeper : MonoBehaviour {
// Следующие константы зависят от файла изображения по умолчанию DelverTiles.
const int lockedR = 95;
const int lockedUR = 81;
const int lockedUL = 80;
const int lockedL = 100;
const int lockedDL = 101;
const int lockedDR = 102;

//.....Индексы плиток с открытыми дверьми
const int openR = 48;
const int openUR = 93;
const int openUL = 92;
const int openL = 51;
```

```
const int openDL = 26;
const int openDR = 27;

private IKeyMaster keys;

void Awake() {
keys = GetComponent<IKeyMaster>(); }

void OnCollisionStay( Collision coll ) {
// Если ключей нет, можно не продолжать
if (keys.keyCount < 1) return;

// Интерес представляют только плитки
Tile ti = coll.gameobject.GetComponent<Tile>();
 if (ti == null) return;

// Открывать, только если Дрей обращен лицом к двери
// (предотвратить случайное использование ключа)
int facing = keys.GetFacing();
// Проверить, является ли плитка закрытой дверью
Tile ti2;
switch (ti.tileNum) {
case lockedR:
if (facing != 0) return;
ti.SetTile( ti.x, ti.y, openR );
break;

case lockedUR:
ti.SetTile( ti.x, ti.y, openUR );
 ti2 = TileCamera.TILES[ti.x-l, ti.y];
ti2.SetTile( ti2.x, ti2.y, openUL );
 break;
case lockedUL:
if (facing != 1) return;
ti.SetTile( ti.x, ti.y, openUL );
ti2 = TileCamera.TILES[ti.x+l, ti.y];
ti2.SetTile( ti2.x, ti2.y, openUR );
 break;
```

```
case lockedl.:
if (facing != 2) return;
 ti.SetTile( ti.x, ti.y, openL );
 break;

case lockedDL:
if (facing != 3) return;
ti.SetTile( ti.x, ti.y, openDL );
ti2 = TileCamera.TILES[ti.x+l, ti.y];
 ti2.SetTile( ti2.x, ti2.y, openDR );
break;
case lockedDR:
if (facing != 3) return;
ti.SetTile( ti.x, ti.y, openDR );
ti2 = TileCamera.TILES[ti.x-l, ti.y];
ti2.SetTile( ti2.x, ti2.y, openDL );
break;

default:
return; // Выйти, чтобы исключить уменьшение счетчика ключей
}
 keys.keyCount--;
}}
```

4. Сохраните все сценарии в MonoDevelop и вернитесь в Unity.

5. Щелкните по кнопке Play (Играть) в Unity и попробуйте перейти в комнату справа. Если попробовать выйти в запертую дверь на севере, у вас ничего не получится (пока).

6. Пока Unity находится в режиме игры, выберите объект Dray в иерархии и введите в поле numKeys компонента Dray (Script) число 6 (этого количества ключей достаточно, чтобы обойти все подземелье). Если теперь подойти к запертой двери, вы сможете миновать ее и исследовать остальное подземелье1!

Пользовательский интерфейс для отображения количества ключей и уровня здоровья

Иаши игроки не смогут увидеть количество имеющихся у них ключей, заглянув в инспектор Unity, значит, мы должны добавить несколько элементов пользовательского интерфейса.

1. Создайте новый объект Canvas, выбрав в главном меню Unity пункт GameObject > UI > Canvas (Игровой объект > ПИ > Холст). В результате на верхнем уровне иерархии будут созданы объекты Canvas и Event System.

2. Выберите Canvas в иерархии.

3. Выполните следующие настройки в инспекторе:

В раскрывающемся списке Render Mode выберите пункт Screen Space – Camera.

Щелкните на маленьком изображении мишени справа от поля Render Camera и в появившемся диалоге на вкладке Scene (Сцена) выберите GUI Camera.

4. Перетащите объект DelverPanel из папки _Prefabs на объект Canvas в иерархии, чтобы вложить DelverPanel в Canvas. После этого справа на экране, в изображении, которое формируется камерой пользовательского интерфейса GUI Camera, должна появиться панель. Теперь напишем сценарий, управляющий этим пользовательским интерфейсом.

Поддержка уровня здоровья

В настоящий момент класс Dray хранит количество ключей, но не имеет никакой информации об уровне здоровья и не предусматривает возможности получения ранений главным героем. Исправим первый из этих недостатков.

1. Откройте сценарий Dray в MonoDevelop и введите следующий код:

```
public class Dray : MonoBehaviour, IFacingMover, IKeyMaster {
public enum eMode { idle, move, attack, transition }

[Header("Set in Inspector")]
public float speed = 5;
public float attackDuration = 0.25f; Продолжительность атаки в секундах
public float attackDelay = 0.5f; // Задержка между атаками
public float transitionDelay = 0.5f; // // Задержка перехода между комнатами
public int maxHealth = 10;
```

```
 [Header("Set Dynamically")]
public int dirHeld = -1; // Направление, соответствующее
// удерживаемой клавише
public int facing = 1; // Направление, куда смотрит Дрей
 public eMode mode = eMode.idle;
public int numKeys = 0;

 [SerializeField]
 private int -health;

public int health {
get { return -health; }
 set { -health = value; } }

private float timeAtkDone = 0;
private float timeAtkNext = 0;

void Awake () {
rigid = GetComponent<Rigidbody>();
anim = GetComponent<Animator>();
inRm = GetComponent<InRoom>();
health = maxHealth; // d }
}
```

2. Сохраните все сценарии в MonoDevelop и вернитесь в Unity.

Подключение пользовательского интерфейса к объекту Dray

Мы должны написать сценарий, отображающий значения свойств health и numKeys экземпляра Dray.

1. Создайте в папке__Scripts новый сценарий на C# с именем GuiPanel.

2. Подключите его к игровому объекту DelverPanel (вложен в объект Canvas в иерархии).

3. Откройте сценарий GuiPanel в MonoDevelop и введите следующий код:

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine; using
UnityEngine.UI;
```

```
public class GuiPanel : MonoBehaviour
 [Header("Set in Inspector")]
public Dray dray;
public Sprite healthEmpty;
public Sprite healthHalf;
public Sprite healthFull;

Text keyCountText;
 List<Image> healthimages;

void Start () {
// Счетчик ключей
 Transform trans = transform.Find("Key Count");
keyCountText = trans.GetComponent<Text>();

// Индикатор уровня здоровья
Transform healthPanel = transform.Find("Health Panel");
healthimages = new List<Image>();
 if (healthPanel != null) {
for (int i=0; i<20; i++) {
trans = healthPanel.Find(”H_"+i);
if (trans == null) break;
healthimages.Add( trans.GetComponent<Image>() );
void Update () {
 // Показать количество ключей
keyCountText.text = dray.numKeys.ToString();

// Показать уровень здоровья
int health = dray.health;
for (int i=0; i<healthlmages.Count; i++) {
if (health > 1) {
healthlmages[i].sprite = healthFull;
} else if (health == 1) { healthlmages[i].sprite = healthHalf;
} else {
healthlmages[i].sprite = healthEmpty; }
 health -= 2; }
}
}
```

4. Сохраните сценарий.

5. Выберите DelverPanel в иерархии и в инспекторе настройте компонент GuiPanel (Script):

Перетащите в поле dray объект Dray из иерархии.

В поле healthEmpty выберите спрайт Health_0 из изображения Health в папке -Images в панели Project (Проект).

В поле healthHalf выберите спрайт Health.1 из изображения Health в папке -Images в панели Project (Проект).

В поле healthFull выберите спрайт Health_2 из изображения Health в папке -Images в панели Project (Проект).

6. Щелкните на кнопке Play (Играть), и вы увидите, как индикатор уровня здоровья заполнится доверху.

7. Выберите Dray в иерархии, не останавливая игру в Unity, и в инспекторе попробуйте вводить разные значения в поля numKeys и .health компонента Dray (Script). Эти изменения должны отражаться в панели пользовательского интерфейса. Сохраните сцену.

Реализация нанесения ущерба Дрею врагами

Настал момент добавить опасностей в игровой мир, дав врагам возможность наносить раны Дрею в момент столкновения с ними. Кроме того, столкновение с врагом будет отбрасывать Дрея немного назад и на короткое время делать его неуязвимым.

Реализация DamageEffect

Сценарий DamageEffect будет использоваться для определения степени вреда, наносимого врагом Дрею, и того, должен ли контакт с врагом вызывать отбрасывания Дрея. Позднее этот же сценарий мы применим к оружию Дрея, чтобы определить эффект воздействия оружия на врагов.

1. Создайте в папке__Scripts новый сценарий на C# с именем DamageEffect.

2. Подключите его к игровому объекту Skeletos в иерархии.

3. Откройте сценарий DamageEffect в MonoDevelop и введите следующий код:

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;
```

```
public class DamageEffect : MonoBehaviour {
 [Header("Set in Inspector”)]
public int damage = 1;
public bool knockback = true; }
```

4. Сохраните сценарий DamageEffect в MonoDevelop и вернитесь в Unity. Теперь в компоненте DamageEffect (Script) объекта Skeletos, в инспекторе, должны появиться два поля.

Изменения в классе Dray

Нам также нужно изменить класс Dray и использовать в нем сценарий DamageEffect, только что подключенный к объекту Skeletos.

1. Откройте сценарий Dray в MonoDevelop и добавьте следующие строки.

```
public class Dray : MonoBehaviour, IFacingMover, IKeyMaster {
public enum eMode { idle, move, attack, transition, knockback }

 [Header("Set in Inspector”)]
public float attackDelay = 0.5f; // Задержка между атаками
public float transitionDelay = 0.5f; // Задержка перехода между
// комнатами public
 int maxHealth = 10;
public float knockbackSpeed = 10;
public float knockbackDuration = 0.25f;
public float invincibleDuration = 0.5f;

[Header("Set Dynamically")]
public int numKeys = 0;
 public bool invincible = false;

[SerializeField]
private int _health;
private float transitionDone = 0;
private Vector2 transitionPos;
private float knockbackDone =0;
private float invincibleDone = 0;
 private Vector3 knockbackVel;
private SpriteRenderer sRend;
```

```
 private Rigidbody rigid;

void Awake () {
sRend = GetComponent<SpriteRenderer>();
rigid = GetComponent<Rigidbody>();
}

Void Update()
if (invincible && Time.time > invincibleDone) invincible = false;
sRend.color = invincible ? Color.red : Color.white;
if ( mode == eMode.knockback ) {
rigid.velocity = knockbackVel;
if (Time.time < knockbackDone) return; }
if ( mode == eMode.transition ){...}
}

void LateUpdateQ { ••• }

void OnCollisionEnter( Collision coll ) {
 if (invincible) return; // Выйти, если Дрей пока неуязвим
DamageEffect dEf =
coll.gameObject.GetComponent<DamageEffect>();
 if (dEf == null) return; // Если компонент DamageEffect отсутствует - выйти
health -= dEf.damage;// Вычесть величину ущерба из уровня здоровья
invincible = true; // Сделать Дрея неуязвимым
invincibleDone = Time.time + invincibleDuration;

if (dEf.knockback) { // Выполнить отбрасывание
// Определить направление отбрасывания
Vector3 delta = transform.position - coll.transform.position;
if (Mathf.Abs(delta.x) >= Mathf.Abs(delta.y)) {
// Отбрасывание по горизонтали
delta.x = (delta.x > 0) ? 1 : -1;
delta.y = 0;
} else {
// Отбрасывание по вертикали delta.x = 0;
delta.y = (delta.y > 0) ? 1 : -1; }
```

```
// Применить скорость отскока к компоненту Rigidbody
knockbackVel = delta * knockbackSpeed;
rigid.velocity = knockbackVel;

// Установить режим knockback и время прекращения отбрасывания
mode = eMode.knockback;
knockbackDone = Time.time + knockbackDuration;
}
}
// Реализация интерфейса IFacingMover
public int GetFacingO (...)
}
```

2. Сохраните все сценарии в MonoDevelop, вернитесь в Unity и щелкните на кнопке Play (Играть).

Если теперь Дрей столкнется со скелетом, он получит повреждение, отскочит назад и станет неуязвимым.

Реализация нанесения урона врагам

Сейчас Дрей уже может взмахнуть мечом, атакуя врага; теперь пришло время придать мечу разящую силу.

1. Выберите объект Sword в иерархии (вложен в объект SwordController внутри объекта Dray).

2. Подключите к нему сценарий DamageEffect.

> В поле damage компонента DamageEffect (Script) введите число 2. Меч должен быть достаточно мощным оружием.

3. Сохраните сцену.

Изменения в классе Enemy

1. Откройте сценарий Enemy в MonoDevelop и добавьте следующие строки:

```
public class Enemy : MonoBehaviour {

[Header("Set in Inspector: Enemy”)]
public float maxHealth = 1;
public float knockbackSpeed = 10;
public float knockbackDuration = 0.25f;
public float invincibleDuration = 0.5f;
```

```
[Header("Set Dynamically: Enemy”)]
public float health;
public bool invincible = false;
public bool knockback = false;

private float invincibleDone = 0;

protected virtual void Awake() { ... }

protected virtual void Update() {
// Проверить состояние неуязвимости и необходимость выполнить отскок
if (invincible && Time.time > invincibleDone)
invincible = false;
sRend.color = invincible ? Color.red : Color.white;
if ( knockback ) {
 rigid.velocity = knockbackVel;
if (Time.time < knockbackDone) return; }
anim.speed =1;
knockback = false;
}

void OnTriggerEnter( Collider colld ) {
if (invincible) return; // Выйти, если Дрей пока неуязвим
DamageEffect dEf = colld.gameobject.GetComponent<DamageEffect>();
 if (dEf == null) return; // Если компонент DamageEffect отсутствует – выйти

health -= dEf.damage; // Вычесть величину ущерба из уровня здоровья
if (health <= 0) Die();
invincible = true; // Сделать Дрея неуязвимым
invincibleDone = Time.time + invincibleDuration;
if (dEf.knockback) { // Выполнить отбрасывание
// Определить направление отскока
Vector3 delta = transform.position - colld.transform.root.position;
if (Mathf.Abs(delta.x) >= Mathf.Abs(delta.y)) {
// Отбрасывание по горизонтали
```

```
 delta.x = (delta.x > 0) ? 1 : -1;
delta.y = 0;
} else {
// Отбрасывание по вертикали
delta.x = 0;
delta.y = (delta.y > 0) ? 1 : -1; }

// Применить скорость отбрасывания к компоненту Rigidbody
knockbackVel = delta * knockbackSpeed;
rigid.velocity = knockbackVel;

// Установить режим knockback и время прекращения отбрасывания
 knockback = true;
knockbackDone = Time.time + knockbackDuration;
anim.speed = 0;
}
}
void Die() {
Destroy(gameObject);
```

2. Сохраните сценарий Enemy.

3. Откройте сценарий Skeletos в MonoDevelop и внесите следующие небольшие изменения:

```
public class Skeletos : Enemy, IFacingMover {

protected override void Awake () { ... }

override protected void Update () {
base.Update();
if (knockback) return;

if (Time.time >= timeNextDecision) {
 DecideDirection(); }

// Поле rigid унаследовано от класса Enemy и инициализируется
// в Enemy.Awake() r
igid.velocity = directions[facing] * speed;
}
}
```

4. Сохраните все сценарии в MonoDevelop и вернитесь в Unity.

5. Выберите объект Skeletos в иерархии и в инспекторе введите значение 4 в поле maxHealth компонента Skeletos (Script). В результате, чтобы убить скелета, Дрею придется нанести два удара мечом (каждый удар которым уменьшает уровень здоровья врага на 2).

6. Сохраните сцену и щелкните на кнопке Play (Играть).

Теперь Дрей может атаковать скелетов, наносить им ущерб и заставлять отскакивать назад.

2. Создание 3D игры

2.1 Создание 3D-ролика

Начало проекта: размещение объектов

Итак, начнем с создания объектов и размещения их в сцене. Первыми будут статические объекты – пол и стены. Затем выберем место для источников света и камеры. Последним создадим игрока – это будет объект, к которому вы добавите сценарии, перемещающие его по сцене.

Декорации: пол, внешние и внутренние стены

В расположенном в верхней части экрана меню GameObject наведите указатель мыши на строчку 3D Object, чтобы открыть дополнительное меню. Выберите в нем вариант Cube, так как для нашей сцены требуется куб.

Передвиньте и отмасштабируйте куб, чтобы создать пол.

В разной степени растянутый по разным осям объект теряет кубическую форму

При этом мы его слегка опускаем вниз для компенсации высоты, но объект остается центрированным

Повторите описанную последовательность для создания внешних стен комнаты. Можно каждый раз задействовать новый куб, а можно копировать и вставлять существующий объект, указывая стандартные сокращения. Двигайте, поворачивайте и масштабируйте стены.

Коллайдер и точка наблюдения игрока

В этом проекте игрока будет представлять обычный примитив. В меню GameObject выберите вариант Capsule. Появится цилиндрическая фигура со скругленными концами – это и есть наш игрок. Сместите объект вверх, сделав его координату Y равной 1.1. Теперь наш игрок может произвольным образом перемещаться вдоль осей X и Z при условии, что он остается внутри комнаты и не касается стен. Присвойте объекту имя Player.

На панели Inspector вы увидите, что этому объекту назначен капсульный коллайдер. Вместо капсульного коллайдера мы назначим объекту контроллер персонажа.

Для завершения настройки игрока осталось сделать всего один шаг – присоединить к нему камеру. Как уже упоминалось в разделе,

посвященном созданию пола и стен, вы можете перетаскивать объекты и размещать их друг на друге на вкладке Hierarchy. Проделайте эту операцию, перетащив камеру на капсулу, чтобы присоединить ее к игроку. Затем расположите ее таким образом, чтобы она соответствовала глазам игрока.

Двигаем объекты: сценарий, активирующий преобразования

Чтобы заставить игрока перемещаться по сцене, нам потребуются сценарии движения, которые будут присоединены к игроку. Именно они будут реагировать на клавиатурный ввод и манипуляции мышью, но для начала мы заставим игрока поворачиваться на месте.

Написание кода

Создайте новый сценарий C#.

Приводим объект во вращение:

```
using UnityEngine;
using System.Collections;

public class Spin : MonoBehaviour public float speed = 3.0f;

void Update() {
transform.Rotate(0, speed, 0);
}
}
```

Компонент сценария для осмотра сцены: MouseLook

Фреймворк MouseLook с перечислением для преобразования поворота

```
using UnityEngine;
using System.Collections;

public class MouseLook : MonoBehaviour { public enum RotationAxes {
MouseXAndY = 0,
MouseX = 1,
MouseY = 2
}
public RotationAxes axes = RotationAxes.MouseXAndY;
```

```
void Update() {
if (axes == RotationAxes.MouseX) {
// это поворот в горизонтальной плоскости
}
else if (axes == RotationAxes.MouseY) {
// это поворот в вертикальной плоскости
}
else {
// это комбинированный поворот
}
}
}
```

Удалите компонент Spin и вместо него присоедините к игроку новый сценарий.

Горизонтальное вращение, следящее за указателем мыши

Первая и наиболее простая ветка соответствует вращению в горизонтальной плоскости.

Поворот по горизонтали, пока не связанный с движениями указателя:

```
...
public RotationAxes axes = RotationAxes.MouseXAndY;
public float sensitivityHor = 9.0f;
void Update() {
if (axes == RotationAxes.MouseX) {
transform.Rotate(0, sensitivityHor, 0);
}
...
```

Команда Rotate, реагирующая на движения указателя мыши

```
...
transform.Rotate(0, Input.GetAxis("Mouse X") * sensitivityHor, 0);
```

... Метод GetAxis() получает данные, вводимые с помощью мыши.

Щелкните по кнопке Play и подвигайте мышь в разные стороны. Объект начнет поворачиваться вправо и влево вслед за указателем.

Поворот в вертикальной плоскости для сценария MouseLook:

```
public float sensitivityHor = 9.0f; public float sensitivityVert = 9.0f;

public float minimumVert = -45.0f; public float maximumVert = 45.0f;

private float _rotationX = 0;

void Update() {
if (axes == RotationAxes.MouseX) {
transform.Rotate(0, Input.GetAxis("Mouse X") * sensitivityHor, 0);
}
else if (axes == RotationAxes.MouseY) {
_rotationX -= Input.GetAxis("Mouse Y") * sensitivityVert;
_rotationX = Mathf.Clamp(_rotationX, minimumVert, maximum-Vert);
float rotationY = transform.localEulerAngles.y;

}
transform.localEulerAngles = new Vector3(_rotationX, rotationY, 0);
```

Метод Rotate() просто увеличивает угол поворота, в то время как на этот раз мы его задаем в явном виде.

Одновременные горизонтальное и вертикальное вращения

В последнем фрагменте кода метод Rotate() также не применяется, так как нам снова требуется ограничить диапазон углов поворота по вертикали. Это означает, что горизонтальный поворот тоже нужно вычислять вручную.

Сценарий MouseLook, поворачивающий объект одновременно по горизонтали и по вертикали:

```
...
else {
_rotationX -= Input.GetAxis("Mouse Y") * sensitivityVert;
_rotationX = Mathf.Clamp(_rotationX, minimumVert, maximum-Vert);
```

```
float delta = Input.GetAxis("Mouse X") * sensitivityHor;
float rotationY = transform.localEulerAngles.y + delta;

}
transform.localEulerAngles = new Vector3(_rotationX, rotationY, 0);
```

Поместите в сценарий MouseLook вот такой метод Start():

```
...
void Start() {
Rigidbody body = GetComponent<Rigidbody>();
if (body != null)  Проверяем, существует ли этот компонент.
body.freezeRotation = true;
}
...
```

После запуска этого сценария вы получите возможность смотреть во всех направлениях, просто перемещая указатель мыши.

Компонент для клавиатурного ввода

Возможность смотреть по сторонам в ответ на перемещения указателя мыши относится к важным фрагментам элементов управления персонажем в играх от первого лица, но это только половина дела. Кроме этого, игрок должен перемещаться по сцене в ответ на клавиатурный ввод. Поэтому в дополнение к компоненту для элемента управления мышью напишем компонент для клавиатурного ввода; создайте новый компонент C# script с именем FPSInput и присоедините его к игроку. При этом мы на время ограничим компонент MouseLook только горизонтальным вращением.

Код вращения из первого листинга с парой небольших изменений:

```
using UnityEngine;
using System.Collections;
public class FPSInput : MonoBehaviour { public float speed = 6.0f;
void Update() {
transform.Translate(0, speed, 0);
```

Необязательный элемент на случай, если вы захотите увеличить скорость.

Меняем метод Rotate() на метод Translate().

```
}}
```

Реакция на нажатие клавиш

Код перемещения в ответ на нажатия клавиш аналогичен коду вращения в ответ на движение указателя мыши. Мы снова используем метод GetAxis(), причем аналогичным образом.

Изменение положения в ответ на нажатие клавиш:

```
...
void Update() {float deltaX = Input.GetAxis("Horizontal") * speed; float deltaZ = Input.GetAxis("Vertical") * speed; transform.Translate(deltaX, 0, deltaZ);
}
```

Компонент CharacterController для распознавания столкновений

Назначенные объекту напрямую преобразования не затрагивают такой аспект, как распознавание столкновений. В результате персонаж начинает ходить сквозь стены. Поэтому нам требуется компонент CharacterController. Именно он обеспечит естественность перемещений нашего персонажа.

Перемещение компонента CharacterController вместо компонента Transform

```
...
private CharacterController _charController;

void Start() {
_charController = GetComponent<CharacterController>();
}

void Update() {
float deltaX = Input.GetAxis("Horizontal") *  speed; float deltaZ = Input.GetAxis("Vertical") *  speed; Vector3 movement = new Vector3(deltaX, 0, deltaZ); movement = Vector3.ClampMagnitude(movement, speed);

movement *= Time.deltaTime;
movement = transform.TransformDirection(movement);
_charController.Move(movement);
}
```

Ходить, а не летать

Теперь, когда распознавание столкновений работает, в сценарий можно добавить силу тяжести, чтобы игрок стоял на полу. Объявим переменную gravity и воспользуемся ее значением для оси Y, как показано в следующем листинге.

Добавление силы тяжести в код движения:

```
...
public float gravity = -9.8f;
void Update() {
...
movement = Vector3.ClampMagnitude(movement, speed);
movement.y = gravity;      Используем значение переменной gravity вместо нуля.
...
```

Теперь на игрока действует постоянная сила, тянущая его вниз. К сожалению, она не всегда направлена строго вниз, так как изображающий игрока объект может наклоняться вниз и вверх, следуя за движениями указателя мыши. К счастью, у нас есть все, чтобы исправить этот недочет, достаточно слегка поменять настройки компонента по отношению к игроку. Первым делом ограничьте компонент MouseLook только горизонтальным вращением. Затем добавьте этот компонент к камере и ограничьте его только вертикальным вращением. В результате на перемещения указателя мыши у вас будут реагировать два разных объекта!

2.2. Добавляем в игру врагов и снаряды

Начнем мы с написания сценариев, превращающих нашего игрока в стрелка. После этого мы заполним сцену врагами, добавив в числе прочего код, позволяющий бесцельно перемещаться по сцене и реагировать на попадание. В заключение мы дадим врагам возможность отстреливаться, кидая в игрока огненные шары.

Вот план нашей будущей работы:

– написать код, позволяющий игроку стрелять;
– создать статичные цели, реагирующие на попадание;
– заставить цели перемещаться по сцене;
– вызвать автоматическое появление новых блуждающих целей;
– дать возможность целям/врагам кидать в игрока огненные шары.

Стрельба путем бросания лучей

Первой дополнительной функциональной возможностью, которую мы добавим в наш демонстрационный ролик, станет стрельба. Умение оглядываться по сторонам и перемещаться в шутере от первого лица является, без сомнения, решающим, но игра не начнется, пока игроки не смогут влиять на окружающее пространство и показывать свое мастерство. Стрельба в трехмерных играх реализуется несколькими способами, наиболее важным из которых является бросание лучей.

Имитация стрельбы командой ScreenPointToRay

Итак, мы реализуем стрельбу проецированием луча, начинающегося в месте нахождения камеры и распространяющегося по центральной линии ее поля зрения. Проецирование луча по центральной линии поля зрения камеры представляет собой частный случай действия, которое называется выбором с помощью мыши.

Сценарий RayShooter, присоединяемый к камере:

```
using UnityEngine;
using  System.Collections;

public class RayShooter : MonoBehaviour { private Camera
_camera;

void Start() {
_camera = GetComponent();
}

void Update() {
if  (Input.GetMouseButtonDown(0)) {
Vector3 point = new Vector3(
_camera.pixelWidth/2,  _camera.pixelHeight/2,  0); Ray ray =
_camera.ScreenPointToRay(point); RaycastHit hit;
if (Physics.Raycast(ray, out hit)) {
Debug.Log("Hit " + hit.point);
}
}      Загружаем координаты точки,
}      в которую попал луч.
}
```

Испущенный луч заполняет информацией переменную, на которую имеется ссылка.

Вектор создается, чтобы определить для луча экранные координаты. Параметры pixelWidth и pixelHeight дают нам размер экрана. Определить его центр можно, разделив эти значения пополам. Хотя координаты экрана являются двумерными, то есть у нас есть только размеры по вертикали и горизонтали, а глубина отсутствует, вектор Vector3 все равно создается, так как метод ScreenPointToRay() требует данных этого типа. Вызванный для этого набора координат метод ScreenPointToRay() дает нам объект Ray.

Затем луч передается в метод Raycast(), причем это не единственный передаваемый в этот метод объект. Есть также структура данных RaycastHit; она представляет собой набор информации о пересечении луча, в том числе о точке, в которой возник луч, и об объекте, с которым он столкнулся.

В конце мы вызываем метод Physics.Raycast(), который проверяет место пересечения рассматриваемого луча, собирает информацию об этом пересечении и возвращает значение true в случае столкновения луча с препятствием. Так как возвращаемое значение принадлежит типу Boolean, метод можно поместить в инструкцию проверки условия, совсем как метод Input.GetMouseButtonDown() чуть раньше.

Добавление визуальных индикаторов для прицеливания и попаданий

Теперь нам нужно добавить в сцену два вида визуальных индикаторов: точку прицеливания в середине экрана и метку в месте столкновения луча с препятствием.

Сценарий RayShooter после добавления индикаторных сфер:

```
using UnityEngine;
using  System.Collections;

public class RayShooter : MonoBehaviour { private Camera
_camera;
void Start() {
_camera = GetComponent();
}

void Update() {
```

```
if (Input.GetMouseButtonDown(0)) { Vector3 point = new Vector3(
_camera.pixelWidth/2, _camera.pixelHeight/2, 0); Ray ray = _camera.ScreenPointToRay(point); RaycastHit hit;
if (Physics.Raycast(ray, out hit)) {
StartCoroutine(SphereIndicator(hit.point));
}
}  }
private IEnumerator SphereIndicator(Vector3 pos) {
GameObject sphere = GameObject.CreatePrimitive(PrimitiveType.Sphere); sphere.transform.position = pos;
yield return new WaitForSeconds(1);Destroy(sphere);
}
}
```

Как следует из ее имени, функция StartCoroutine() запускает сопрограмму. После этого она начинает работать до завершения выполнения, периодически делая паузы.

Визуальный индикатор для точки прицеливания:

```
...
void Start() {
_camera = GetComponent();
Cursor.lockState = CursorLockMode.Locked; Cursor.visible = false;
}
void OnGUI() { int size = 12;
float posX = _camera.pixelWidth/2 - size/4; float posY = _camera.pixelHeight/2 - size/2; GUI.Label(new Rect(posX, posY, size, size), "*");
}
...
```

Итак, код, управляющий поведением игрока, готов. Фактически мы завершили работу над взаимодействиями игрока со сценой, но у нас все еще отсутствуют цели.

Создаем активные цели

Возможность стрелять – это замечательно, но в данный момент стрелять нашему игроку не во что. Поэтому нам нужно создать целе-

вой объект и присоединить к нему сценарий, программирующий реакцию на попадание.

Определяем точку попадания

Первым делом нам нужен объект, который послужит мишенью. Создайте куб и поменяйте его размер по вертикали, введя в поле Y для преобразования Scale значение 2. В полях X и Z оставьте значение 1. Поместите новый объект в точку с координатами 0, 1, 0, чтобы он стоял на полу в центре комнаты, и присвойте ему имя Enemy. Создайте сценарий с именем ReactiveTarget и присоедините его к объекту.

Вернитесь к сценарию RayShooter.cs и отредактируйте код испускания луча в соответствии с показанным листингом.

Распознавание попаданий в цель...

```
if (Physics.Raycast(ray, out hit)) {
GameObject hitObject = hit.transform.gameObject;
ReactiveTarget target = hitObject.GetComponent<ReactiveTarget>();
if (target != null) { Debug.Log("Target hit");
} else { StartCoroutine(SphereIndicator(hit.point));
}
}
...
```

Уведомляем цель о попадании

В коде нужно поменять всего одну строку, как показано в следующем листинге.

Отправка сообщения целевому объекту:

```
...
if (target != null) {
target.ReactToHit();
```

Вызов метода для мишени вместо генерации отладочного сообщения.

```
} else { StartCoroutine(SphereIndicator(hit.point));
}
...
```

Теперь отвечающий за стрельбу код вызывает связанный с мишенью метод, который нам нужно написать. Введите в сценарий ReactiveTarget код следующего листинга.

Сценарий ReactiveTarget, реализующий смерть врага при попадании:

```
using UnityEngine;
using System.Collections;

public class ReactiveTarget : MonoBehaviour {

public void ReactToHit() {      Метод, вызванный сценарием стрельбы.
StartCoroutine(Die());
}

private IEnumerator Die() {      Опрокидываем врага, ждем 1,5 секунды и уничтожаем его.
this.transform.Rotate(-75, 0, 0);

yield return new WaitForSeconds(1.5f);

Destroy(this.gameObject);
}
}
```

Теперь наша мишень реагирует на попадание! Но больше она ничего не делает. Давайте добавим ей дополнительные модели поведения, чтобы превратить ее в настоящего врага.

Базовый искусственный интеллект для перемещения по сцене

Статичная цель не очень интересна, поэтому давайте напишем код, который заставит врагов перемещаться по сцене. Такой код является одним из простейших примеров искусственного интеллекта (Artificial Intelligence, AI). Этот термин относится к сущностям, поведение которых контролируется компьютером. В данном случае такой сущностью служит враг в нашей игре, но в реальной жизни это может быть, например, робот.

Базовый сценарий WanderingAI:

```
using UnityEngine;
using System.Collections;
ublic class WanderingAI : MonoBehaviour { public float speed =
3.0f;
public float obstacleRange = 5.0f;
void Update() {
transform.Translate(0, 0, speed * Time.deltaTime);
Ray ray = new Ray(transform.position, transform.forward);
RaycastHit hit;
if (Physics.SphereCast(ray, 0.75f, out hit)) {
if (hit.distance <  obstacleRange)  { float angle = Random.Range(-
110, 110); transform.Rotate(0, angle, 0);
}}    }      }
```

Слежение за состоянием персонажа

Текущее поведение врага имеет один недостаток. Движение вперед продолжается даже после попадания в него пули. Ведь метод Translate() запускается в каждом кадре вне зависимости от обстоятельств. Внесем в код небольшие изменения, позволяющие следить за тем, жив персонаж или мертв. Говоря техническим языком, мы хотим отслеживать «живое» состояние персонажа.

Сценарий WanderingAI после добавления «живого» состояния

```
...
private bool  _alive;  void Start() {
_alive = true;
}

void Update() {
if(_alive) { transform.Translate(0, 0, speed * Time.deltaTime);

...
}
}
public void SetAlive(bool alive) {
_alive = alive;
}
...
```

Сценарий ReactiveTarget сообщает сценарию WanderingAI, когда наступает смерть:

```
...
public void ReactToHit() {
WanderingAI behavior = GetComponent();
if (behavior != null) { behavior.SetAlive(false);
}
StartCoroutine(Die());
}
...
```

Увеличение количества врагов

В настоящее время в сцене присутствует всего один враг, и после его смерти сцена становится пустой. Давайте заставим игру порождать врагов, сделав так, чтобы после смерти существующего врага сразу появлялся новый. В Unity это легко делается с помощью механизма шаблонов экземпляров (prefabs).

Экземпляры невидимого компонента SceneController

Хотя сам по себе шаблон в сцене отсутствует, нам нужен некий объект, к которому будет присоединяться код, порождающий врагов. Поэтому мы создадим пустой игровой объект и добавим к нему сценарий, при этом сам объект останется невидимым.

Выберите в меню GameObject команду Create Empty и присвойте новому объекту имя Controller. Убедитесь, что он находится в точке с координатами 0, 0, 0. Создайте сценарий SceneController, показанный в следующем листинге.

Сценарий SceneController, порождающий экземпляры врагов:

```
using UnityEngine;
using System.Collections;

public class SceneController : MonoBehaviour { [SerializeField] private GameObject enemyPrefab; private GameObject _enemy;
Void Update() {

if (_enemy == null) {
_enemy = Instantiate(enemyPrefab) as GameObject;
```

```
_enemy.transform.position = new Vector3(0, 1, 0); float angle = Random.Range(0, 360);
_enemy.transform.Rotate(0, angle, 0);
}
}
}
```

Присоедините этот сценарий к объекту-контроллеру, и на панели Inspector появится поле для шаблона врага с именем Enemy Prefab. Оно работает аналогично общедоступным переменным, но есть и важное отличие.

Стрельба путем создания экземпляров

Хорошо, добавим врагам еще немного способностей. Пойдем по тому же пути, что и при работе над игроком. Первым делом мы научили врагов двигаться, теперь дадим им возможность стрелять!

Шаблон снаряда

Если раньше мы стреляли без применения реальных снарядов, то теперь они нам понадобятся. Стрельба приемом испускания лучей, по сути, мгновенна, и попадание регистрируется в момент щелчка кнопкой мыши, враги же будут бросать «огненные шары», летающие по воздуху. Разумеется, хотя они двигаются довольно быстро, у игрока будет возможность уклониться. Фиксировать попадания мы будем не методом испускания лучей, а методом распознавания столкновений.

Код будет порождать огненные шары тем же способом, каким порождаются враги, – созданием экземпляров из шаблона. Так как нам требуется огненный шар, выберите в меню GameObject команду 3D Object и в дополнительном меню команду Sphere. Присвойте появившейся сфере имя Fireball. Теперь создайте новый сценарий с таким же именем и присоедините его к объекту. Чтобы он напоминал огненный шар, ему нужно присвоить ярко-оранжевый цвет. Такие свойства поверхностей, как цвет, контролируются при помощи материалов.

Стрельба и столкновение с целью

Отредактируем шаблон врага, наделив его способностью кидать огненные шары. Чтобы код распознавал игрока, потребуется новый сценарий, поэтому создадим сценарий с именем PlayerCharacter и присоединим его к игроку. А теперь откроем сценарий WanderingAI и введем в него код следующего листинга.

Сценарий WanderingAI с возможностью кидать огненные шары:

```
...
[SerializeField] private GameObject fireballPrefab; private GameObject _fireball;
...
if (Physics.SphereCast(ray, 0.75f, out hit)) {
GameObject hitObject = hit.transform.gameObject;
if (hitObject.GetComponent<PlayerCharacter>()) {
if (_fireball == null) {
_fireball = Instantiate(fireballPrefab) as GameObject;
_fireball.transform.position =
transform.TransformPoint(Vector3.forward * 1.5f);
_fireball.transform.rotation = transform.rotation;
}
}      else if (hit.distance < obstacleRange) { float angle = Random.Range(-110, 110); transform.Rotate(0, angle, 0);
}
}
```

Как только новый код окажется на своем месте, на панели Inspector появится новое поле Fireball Prefab, точно так же как в свое время для компонента SceneController появилось поле Enemy Prefab. Щелкните по шаблону врага на вкладке Project, и на панели Inspector появятся компоненты этого объекта, как если бы вы выделили его в сцене.

Теперь, как только игрок окажется непосредственно перед врагом, в него полетит огненный шарПопробуйте выполнить воспроизведение – перед врагом появится огненная сфера, но никуда не полетит, потому что мы пока не написали соответствующий сценарий. Давайте сделаем это сейчас. Код сценария Fireball приведен в следующем листинге.

Сценарий Fireball, реагирующий на столкновения

```
using UnityEngine;
using System.Collections;


public class Fireball : MonoBehaviour { public float speed = 10.0f;
public int damage = 1;
```

```
void Update() {
transform.Translate(0, 0, speed * Time.deltaTime);
}
void OnTriggerEnter(Collider other) {
PlayerCharacter player = other.GetComponent<PlayerCharacter>();
if (player  != null) {       Debug.Log("Player hit");
}
Destroy(this.gameObject);
}
}
```

Огненному шару также требуется компонент Rigidbody, относящийся к системе моделирования законов физики.

Повреждение игрока

Ранее мы создали сценарий PlayerCharacter, но оставили его пустым. Теперь введите в него из следующего листинга код реакции на попадание.

Игрок может получать повреждения:

```
using UnityEngine;
using System.Collections;

public class PlayerCharacter : MonoBehaviour { private int _health;

void Start() {
_health  =  5;
}

public void Hurt(int damage) {
_health  -=  damage
Debug.Log("Health: " + _health);
}
}
```

Теперь нужно вернуться к сценарию Fireball, чтобы вызвать для игрока метод Hurt(). Вставьте вместо отладочной строчки в сценарии Fireball строку player. Hurt(damage), которая будет сообщать игроку о попадании. Всё. Последний фрагмент кода встал на свое место!

2.3. Работа с графикой

Создание геометрической модели сцены

Разговор о моделировании сцен мы начнем с рассмотрения процесса создания геометрической модели сцены (whiteboxing). Обычно это первый шаг моделирования уровня с помощью компьютера (следующий за разработкой этого уровня на бумаге).

Назначение геометрической модели

Составление сцены из примитивов практикуется по двум причинам. Во-первых, это позволяет быстро получить «набросок», который со временем будет постепенно совершенствоваться. Эта деятельность близко связана с проектированием игровых уровней.

Расставляем примитивы в соответствии с планом

Построение геометрической модели сцены в соответствии с имеющимся планом включает в себя позиционирование и масштабирование множества параллелепипедов, которые будут играть роль стен.

Заставьте объект, представляющий игрока или камеру, двигаться по сцене. Теперь можно походить по сделанной из примитивов сцене, чтобы протестировать получившийся уровень! Все очень просто, но пока у вас есть только чистая геометрия. Декорируем ее с помощью двумерных изображений.

Наложение текстур

Пока что наш уровень представляет собой грубый набросок. Он уже доступен для игры, но очевидно, что над внешним видом сцены требуется еще долго работать. Следующим шагом по совершенствованию уровня будет наложение текстур.

В трехмерной графике текстуры имеют разное применение, но наиболее простым является отображение их на поверхности трехмерных моделей.

Импорт файла изображения

Начнем с создания/подготовки наших будущих текстур. Все изображения, используемые в качестве текстур, обычно являются бесшовными, что позволяет многократно повторять их на больших поверхностях.

Получить бесшовное изображение можно различными способами: например, обработать фотографию или нарисовать собственный вариант картинки.

Скачайте выбранные вами изображения и подготовьте их к использованию в качестве текстур. С технической точки зрения ничто не мешает задействовать их сразу, но в исходном виде они далеки от идеала. Разумеется, они являются бесшовными, но рисунок имеет некорректный размер, а файл – не тот формат, который нам нужен. Размер текстуры должен выражаться в степенях двойки. Графические процессоры показывают максимальную эффективность при обработке изображений, размер которых выражается числом 2N: 4, 8, 16, 32, 64, 128, 256, 512, 1024, 2048 (следующее в этом ряду – число 4096, но это слишком большое изображение, чтобы использовать его в качестве текстуры). В графическом редакторе (это может быть Photoshop, GIMP или любой другой вариант из перечисленных в приложении Б) отмасштабируйте скачанные изображения до размера 256 × 256 и сохраните их в формате PNG.

Назначение текстуры

С технической точки зрения наложить текстуру непосредственно на геометрию невозможно. Текстуры должны входить в состав материалов, которые, собственно, и назначаются объектам. Кроме этого удобного метода автоматического создания материалов существует еще и «корректный» способ через подменю, которое появляется после выбора в меню Assets команды Create; новый ресурс появляется на вкладке Project. Перетащите полученный материал со вкладки Project на объект сцены. Попробуйте проделать все описанное с текстурой для стены: создайте новый материал, перетащите в него текстуру и назначьте его стене. Вы увидите, что на поверхности пола и стен появились изображения камня и кирпичей, но они выглядят растянутыми и размытыми. Как получилось, что единственное изображение оказалось растянутым на весь пол? Мы же хотели, чтобы оно повторялось на поверхности несколько раз. Такой эффект дает свойство Tiling: выделите материал на вкладке Project и измените числа в полях Tiling на панели Inspector. Проверьте, что вы задаете повторение основной, а не вторичной карты. По умолчанию число повторений равно 1; присвойте этому параметру, к примеру, значение 8 и посмотрите, как изменится вид пола. Подберите и для второго материала кратность, обеспечивающую оптимальный вид. Итак, пол и стены нашей комна-

ты обзавелись текстурами! Но вы можете назначить текстуру и небу; давайте посмотрим, как это делается.

Создание неба с помощью текстур

Текстуры камня и кирпича придали стенам и полу намного более естественный вид. Но небо пока выглядит пустым и ненатуральным, мы же хотим придать сцене реалистичность. Чаще всего эта задача решается при помощи специальных текстур с изображениями неба.

Создание нового материала для скайбокса

Сначала создайте новый материал. Его параметры отобразятся на панели Inspector. Первым делом нам нужно поменять шейдер материала.

Каждый материал имеет шейдер, который определяет его вид. Новому материалу по умолчанию назначается шейдер Standard. Выделите строку Skybox и выберите в появившемся дополнительном меню вариант 6 Sided.

Теперь в настройках материала появилось шесть ячеек для текстур. Эти шесть текстур соответствуют шести сторонам куба. Они должны совпадать друг с другом в местах стыка, чтобы картинка получилась бесшовной.

Импортируйте в Unity изображения для скайбокса тем же способом, которым импорти ровалась текстура кирпича: перетащите файлы на вкладку Project или щелкните правой кнопкой мыши по вкладке Project и выберите команду Import New Asset.

Теперь можно перетащить изображения на ячейки для текстур. Имена изображений должны совпадать с именами ячеек. Как только все текстуры окажутся на своих местах, можно использовать материал для скайбокса.

Собственные трехмерные модели

Для моделирования трехмерных объектов широко применяются такие приложения, как Maya от Autodesk и 3ds Max. Но это дорогие коммерческие инструменты, поэтому воспользуемся приложением с открытым исходным кодом, которое называется Blender.

Экспорт и импорт модели

При импорте моделей желательно сразу же поменять несколько параметров. Unity масштабирует импортируемые модели до очень маленьких размеров, поэтому введите в поле Scale Factor значение 100, чтобы частично скомпенсировать параметр File Scale, равный 0,01. Можно также установить флажок Generate Colliders, но это необязательно; просто без коллайдера персонажи смогли бы проходить сквозь скамейку. Затем перейдите на вкладку Animation в параметрах импорта и сбросьте флажок Import Animation – ведь эту модель мы анимировать не будем.

Системы частиц

Кроме двумерных изображений и трехмерных моделей у нас остался еще один тип визуального содержимого – системы частиц. Системы частиц применяются при создании таких эффектов, как огонь, дым или водяные брызги. Если большинство других ресурсов создается во внешних приложениях и импортируется в проект, системы частиц генерируются непосредственно в Unity благодаря гибким и мощным инструментам для создания различных эффектов.

Присоединение эффектов частиц к трехмерным объектам

Создайте сферу. Создайте новый сценарий с именем BackAndForth, показанный в следующем листинге, и присоедините его к сфере.

Движение объекта взад и вперед по прямой:

```
using UnityEngine;
using System.Collections;

public class BackAndForth : MonoBehaviour { public float speed =
3.0f;
public float maxZ = 16.0f; Объект движется между этими точками.
public float minZ = -16.0f;

private int _direction = 1; В каком направлении объект движется в данный момент?
```

```
void Update() {
transform.Translate(0, 0, _direction * speed * Time.deltaTime);

bool bounced = false;
if (transform.position.z > maxZ || transform.position.z < minZ) {
_direction = -_direction;   Меняем направление на противопо-
ложное.
bounced = true;
}
if (bounced) {
transform.Translate(0, 0, _direction * speed * Time.deltaTime);
}       Делаем  дополнительное движение
}       в этом кадре, если объект поменял
}       направление.
```

Запустите этот сценарий, и сфера начнет двигаться взад и вперед по центральному коридору уровня. Теперь можно сделать систему частиц дочерней по отношению к сфере, и огонь начнет перемещаться вместе с ней. Точно так же, как вы поступали со стенами, на вкладке Hierarchy перетащите объект Particle system на объект Sphere.

Теперь система частиц двигается вместе со сферой, но пламя при этом не отклоняется, что выглядит неестественно. Это связано с тем, что по умолчанию частицы перемещаются корректно только в локальном пространстве собственной системы. Для завершения модели горящей сферы найдите в настройках системы частиц параметр Simulation Space и измените его значение с Local на World.

2.4. Двумерный GUI для трехмерной игры

Любая игра требует не только места, где происходит действие, но и средств отображения абстрактных взаимодействий и информации. Без этого невозможно представить себе ни двумерную, ни трехмерную игру, ни шутер от первого лица, ни головоломку. Эти средства отображения абстрактных взаимодействий называют пользовательским интерфейсом (UI) или, точнее, графическим интерфейсом пользователя (GUI).

Инструменты создания UI в Unity будут рассматриваться на примере шутера от первого лица, которым мы занимались ранее. Перед нами стоят следующие задачи:

– спланировать интерфейс;
– расположить на экране UI-элементы;
– запрограммировать взаимодействия с UI-элементами;
– запрограммировать GUI-реакции на события в игре;
– запрограммировать реакции сцены на действия с GUI.

Система IMGUI позволяет легко поместить на экран интерактивную кнопку. Остается только присоединить сценарий из этого листинга к любому объекту сцены. Можно вспомнить еще один пример UI непосредственного режима.

Реализация кнопки в системе IMGUI

```
using UnityEngine;
using System.Collections;

public class BasicUI : MonoBehaviour void OnGUI() {
if (GUI.Button(new Rect(10, 10, 40, 20), "Test")) {
Debug.Log("Test button");
}       Параметры: положение по X,
}       положение по Y, ширина, высота,
}       текстовая подпись.
```

Основа листинга – метод OnGUI(). Все представители класса MonoBehaviour автоматически отвечают как на методы Start() и Update(), так и на метод OnGUI(). Он запускается в каждом кадре после визуализации трехмерной сцены, предоставляя место для команд рисования GUI.

Новая система UI функционирует в режиме удержания, поэтому вся графика задается один раз и рисуется в каждом кадре без постоянных повторных определений компоновки. Размещение графических элементов UI при этом выполняется в редакторе Unity, что дает два преимущества перед UI непосредственного режима. Во-первых, вид будущего интерфейса можно оценить в процессе выкладывания его элементов, во-вторых, для такой системы можно использовать собственную графику.

Выбор компоновки

В большинстве игр проекционный дисплей состоит из нескольких многократно повторяющихся UI-элементов. Поэтому понять про-

цесс создания UI несложно. Кнопка будет вызывать всплывающее окно с текстовым полем и ползунком.

Импорт изображений для элементов UI

Нашему пользовательскому интерфейсу требуется графика для отображения таких элементов, как, к примеру, кнопки.

Импорт изображений (если нужно, вручную определите их как спрайты).

Перетаскивание спрайтов в сцену.

Первым делом перетащите все изображения на вкладку Project, чтобы импортировать их в редактор, а затем на панели Inspector убедитесь, что у каждого из них параметр Texture Type имеет значение Sprite (2D And UI).

Это изображения кнопок, индикатора счета и всплывающего окна, то есть элементов интерфейса, который мы собираемся создать. Теперь, когда они добавлены в проект, расположим графику на экране.

Кнопки, изображения и текстовые подписи

Холст задает область, которая будет отображаться как пользовательский интерфейс, но туда следует добавить спрайты с изображениями отдельных элементов. В разделе UI меню GameObject есть команды, позволяющие создать изображение (image), текст (text) или кнопку (button). Создайте по одному элементу каждого вида.

Между объектами на холсте также можно сформировать иерархические связи для удобства их размещения. Например, если перетащить текст на изображение, надпись будет перемещаться вместе с картинкой. Более того, у кнопки, созданной с параметрами по умолчанию, есть дочерний по отношению к ней текстовый объект; в данном случае мы не будем ничего на ней писать, поэтому просто удалите его. Перетащите элементы интерфейса на предназначенные им места.

Если выделить объект UI, на панели Inspector, в верхней части свитка Image вы увидите поле Source Image. Назначьте спрайт с изображением противника объекту Image, а спрайт с изображением шестерни объекту Button. Чтобы спрайты приобрели корректный размер, щелкните на кнопке Set Native Size.

Программирование интерактивного UI

Для взаимодействия с пользовательским интерфейсом нужен указатель мыши. Если помните, в этой игре его настройки редактируются в методе Start() сценария RayShooter. Изначально мы блокировали и скрывали его. Такое поведение прекрасно подходит для элементов управления в шутере от первого лица, но не позволяет работатьс элементами интерфейса. Удалите эти строчки из сценария RayShooter.cs, чтобы появилась возможность щелкать на элементах проекционного дисплея.

Кроме того, в сценарий RayShooter.cs нужно добавить строки, блокирующие в процессе взаимодействия с GUI возможность стрелять. Вот новая версия кода.

Добавление взаимодействий с GUI в код сценария RayShooter.cs:

```
using UnityEngine.EventSystems; ...
void Update() {
if (Input.GetMouseButtonDown(0) &&
!EventSystem.current.IsPointerOverGameObject()) {
Vector3 point = new Vector3(
camera.pixelWidth/2, camera.pixelHeight/2, 0);
...
```

Теперь в процессе игры можно щелкать на кнопках, хотя пока это не дает результатов. Меняется только оттенок кнопки при наведении на нее указателя мыши и при щелчке. За скорость возвращения кнопки к обычному цвету отвечает параметр Fade Duration в свитке Button. Попробуйте уменьшить его до 0.01 и посмотрите, что получится.

Щелчок по кнопке пока ни к чему не приводит, так как она не связана с каким-либо кодом. Давайте исправим этот недостаток.

Программирование невидимого объекта UIController

Как правило, программирование взаимодействия с элементами интерфейса сводится к стандартной, общей для всех элементов последовательности:

– в сцене создается UI-объект;

– пишется сценарий, который будет вызываться при обращении к этому элементу интерфейса;

– сценарий присоединяется к объекту в сцене;

– элементы интерфейса (например, кнопки) связываются с объектом, к которому присоединен этот сценарий.

Кнопка у нас уже есть, осталось создать контроллер, который будет с ней связываться. Создайте сценарий UIController и перетащите его на объект-контроллер в сцене.

Сценарий UIController, предназначенный для программирования кнопок

```
using UnityEngine;
using UnityEngine.UI;      Импортируем фреймворк для работы с кодом UI.
using System.Collections;
public class UIController : MonoBehaviour { [SerializeField] private Text scoreLabel;

void Update() {
scoreLabel.text = Time.realtimeSinceStartup.ToString();
}

public void OnOpenSettings() { Метод, вызываемый кнопкой настроек.
Debug.Log("open settings");
}
}
```

Перетащим объекты на ячейки компонентов, чтобы связать их друг с другом. Перетащите на поле Score Label объекта UIController текстовый объект, предназначенный для отображения счета. Текст данной подписи начнет определять код сценария UIController. Снабдим кнопку элементом OnClick, чтобы добавить ее к объекту-контроллеру.

Всплывающее окно

Кнопка, входящая в состав нашего интерфейса, должна открывать окно диалога. Но окно сначала нужно создать. Его роль сыграет новый объект Image с присоединенными к нему элементами управления (кнопками и ползунками). Первым делом получим новое изображение, поэтому выберите в меню GameObject команду UI, а затем ко-

манду Image. Как и раньше, на панели Inspector вы найдете ячейку Source Image. Перетащите на нее спрайт с именем popup.

Всплывающее окно готово, можно писать для него код. Создайте сценарий SettingsPopup и перетащите его на наше окно.

Сценарий SettingsPopup для всплывающего окна

```
using UnityEngine;
using System.Collections;

public class SettingsPopup : MonoBehaviour { public void Open() {
gameObject.SetActive(true); Активируем этот объект, чтобы открыть окно.
}
public void Close() {
gameObject.SetActive(false);  Деактивируем объект, чтобы закрыть окно.
}
}
```

Теперь откройте сценарий UIController.cs и добавьте в него содержимое следующего листинга.

Возможность работы с окном для сценария UIController

```
...
[SerializeField] private SettingsPopup settingsPopup; void Start() {
settingsPopup.Close(); Закрываем всплывающее окно в момент начала игры.
}
...
public void OnOpenSettings() {
settingsPopup.Open(); Заменяем отладочный текст методом всплывающего окна.
}
...
```

Чтобы кнопка закрывала окно, ей нужно сопоставить событие OnClick. Щелкните по кнопке + поля OnClick, перетащите всплывающее окно на ячейку объекта и выберите в списке функций вариант Close(). Запустите воспроизведение игры и убедитесь, что кнопка действительно закрывает всплывающее окно.

Задание значений с помощью ползунка и поля ввода

Процедура добавления элементов управления к всплывающему окну настроек состоит из двух этапов, почти как знакомая вам процедура создания кнопок. Вы генерируете присоединенные к холсту элементы интерфейса и связываете их со сценарием. Создадим ползунок, текстовое поле и текстовую подпись к ползунку. Выберите в меню GameObject команду UI > Text, чтобы получить текстовый объект, затем – команду UI > InputField, чтобы получить текстовое поле, а потом – UI > Slider для получения ползунка

Кнопка закрытия располагается в верхнем углу, в то время как текстовая подпись находится над ползунком. Сделайте все три объекта потомками всплывающего окна, перетащив их на вкладке Hierarchy, а затем расположите их в соответствии с рисунком, выровняв по центру всплывающего окна. Присвойте текстовому элементу значение Speed, формируя подпись к ползунку. Параметрам Content Type и Line Type можно оставить значения по умолчанию.

Методы для элементов управления вводом всплывающего окна

Этот метод срабатывает в момент начала:

```
...
public void OnSubmitName(string name) { Debug.Log(name);
ввода данных в текстовое поле.
}
public void OnSpeedValue(float speed) { Debug.Log("Speed: " +
speed);
}
...
```

Аналогичным образом происходит инициализация ползунка с использованием сохраненного значения. Добавьте в сценарий SettingsPopup следующий код:

```
using UnityEngine.UI;
...
[SerializeField] private Slider speedSlider; void Start() {
speedSlider.value = PlayerPrefs.GetFloat("speed", 1);
}
...
```

Обратите внимание, что команде get предоставляется как значение переменной speed, так и значение по умолчанию на случай, если сохраненная скорость отсутствует.

Наши элементы управления генерируют отладочные сообщения, но пока никак не влияют на происходящее в игре.

2.5. Игра от третьего лица: перемещения и анимация игрока

Импортируйте персонаж. На этот раз мы воспользуемся моделью, имеющей вид человека, так как в игре

Вот будущая последовательность действий:

– импорт модели персонажа в сцену;

– настройка элементов управления камеры и ее нацеливание на персонаж;

– написание сценария, позволяющего персонажу бегать по поверхности;

– добавление в сценарий движения возможности прыгать;

– воспроизведение анимации на модели на основе ее движений.

Корректировка положения камеры

Перед тем как приступить к написанию кода, управляющего перемещениями персонажа, следует поместить персонаж в сцену и настроить камеру на отслеживание его положения. Импортируем в проект антропоморфную модель без лица и расположим камеру сверху таким образом, чтобы она смотрела на эту модель под углом.

Сцена полностью готова, осталось добавить в нее модель персонажа.

Импорт персонажа

Импортируйте в проект файл FBX, перетащив его на вкладку Project или щелкнув по этой вкладке правой кнопкой мыши и выбрав в открывшемся меню команду Import New Asset. Теперь обратите внимание на панель Inspector, где нужно скорректировать параметры импорта модели. Первым делом присвойте параметру Scale Factor значение 10, чтобы получить модель корректного размера.

Ниже располагается параметр Normals. Он контролирует вид света и теней на модели, используя для этого такое математическое понятие, как нормали.

Перетащите модель персонажа со вкладки Project в сцену. Введите в поля Position значения 0, 1.1, 0, чтобы персонаж оказался в центре помещения. Первый шаг к созданию игры от третьего лица сделан!

Облет камеры вокруг персонажа

В демонстрационном ролике от первого лица мы связывали камеру с объектом-персонажем на вкладке Hierarchy, обеспечивая их совместное вращение. В игре же от третьего лица персонаж будет поворачиваться независимо от камеры. Поэтому связывать камеру с персонажем на вкладке Hierarchy мы на этот раз не будем. Вместо этого напишем код, который будет менять положение камеры вместе с положением персонажа, поворачивая ее независимо от последнего.

Первым делом выберем положение камеры относительно персонажа.

После этого нужно создать сценарий OrbitCamera и добавить в него код следующего листинга. Присоедините сценарий к камере в виде нового компонента, а затем перетащите объект player на ячейку Target. Запустите воспроизведение сцены, чтобы посмотреть, как работает код камеры.

Сценарий вращения нацеленной на объект камеры вокруг объекта

```
using UnityEngine;
using System.Collections;

public class OrbitCamera : MonoBehaviour { [SerializeField] private
Transform target;

public float rotSpeed = 1.5f; private float _rotY;
private Vector3 _offset;
void Start() {
_rotY = transform.eulerAngles.y;
_offset = target.position – transform.position;
}

void LateUpdate() {
float horInput = Input.GetAxis("Horizontal"); if (horInput != 0) {
_rotY += horInput * rotSpeed;
```

```
} else {
_rotY += Input.GetAxis("Mouse X") * rotSpeed * 3;
}
Quaternion rotation = Quaternion.Euler(0, _rotY, 0); transform.position = target.position - (rotation * _offset);
transform.LookAt(target);
}
}
```

Элементы управления движением, связанные с камерой

Теперь, когда мы импортировали в Unity модель персонажа и написали код, управляющий видом с камеры, нужны элементы управления персонажем. Напишем код для связанных с камерой элементов управления, которые будут перемещать персонаж по сцене при нажатии клавиш со стрелками, а также поворачивать его лицом в нужную сторону.

Поворот персонажа лицом в направлении движения

Напишем код, разворачивающий персонаж в направлении клавиш со стрелками. Создайте сценарий RelativeMovement, перетащите его на объект player и свяжите камеру со свойством Target компонента Script. После этого при нажатии управляющих клавиш персонаж будет поворачиваться в разные стороны, выбирая направление относительно камеры, а при его вращении с помощью мыши останется статичным.

Поворот персонажа относительно камеры

```
using UnityEngine;
using System.Collections;

public class RelativeMovement : MonoBehaviour { [SerializeField] private Transform target;

void Update() {
Vector3 movement = Vector3.zero;

float horInput = Input.GetAxis("Horizontal"); float vertInput = Input.GetAxis("Vertical"); if (horInput != 0 || vertInput != 0) {
movement.x = horInput; movement.z = vertInput;
```

```
Quaternion tmp = target.rotation;
target.eulerAngles = new Vector3(0, target.eulerAngles.y, 0);movement = target.TransformDirection(movement); target.rotation = tmp;
transform.rotation = Quaternion.LookRotation(movement);
}
}      }
```

Сейчас персонаж меняет ориентацию скачками, мгновенно поворачиваясь лицом в направлении, заданном клавишей. Но плавное движение смотрится куда лучше. Здесь на помощь приходит линейный алгоритм интерполяции. Добавьте в сценарий переменную:

```
public float rotSpeed = 15.0f;
```

Затем замените строчку transform.rotation... в конце листинга 7.2 следующим кодом:

```
...
Quaternion direction = Quaternion.LookRotation(movement);
transform.rotation = Quaternion.Lerp(transform.rotation, direction, rotSpeed * Time.deltaTime);
}
}
}
```

В результате вместо привязки непосредственно к значению, возвращаемому методом LookRotation(), это значение начнет использоваться в качестве целевого положения, в которое объект будет постепенно поворачиваться. Метод Quaternion.Lerp() выполняет плавный поворот из текущего положения в целевое.

Движение вперед в выбранном направлении

Выделите объект player и выберите в меню Components команду Physics > Character Controller. На панели Inspector уменьшите параметр Radius до 0,4, остальным параметрам оставьте значения по умолчанию, так как они вполне подходят для нашей модели персонажа.

Следующий листинг содержит код, который нужно добавить в сценарий

RelativeMovement.

Код, меняющий положение персонажа

```
using UnityEngine;
using System.Collections;

[RequireComponent(typeof(CharacterController))] public class RelativeMovement : MonoBehaviour {
...
public float moveSpeed = 6.0f;
private CharacterController _charController; void Start() {
_charController = GetComponent<CharacterController>();
}
void Update() {
...
movement.x = horInput * moveSpeed; movement.z = vertInput * moveSpeed;
movement = Vector3.ClampMagnitude(movement, moveSpeed);
...
}

movement *= Time.deltaTime;
_charController.Move(movement);
}
}
```

Этот код обеспечивает перемещения в горизонтальном направлении, нам же нужно, чтобы персонаж мог перемещаться еще и по вертикали.

Прыжки

Код, который мы добавим, будет обрабатывать как прыжки, так и падения. Точнее, он будет включать в себя силу тяжести, все время тянущую персонаж вниз, а в момент прыжка его будет толкать вверх.

Но сначала добавим в сцену пару платформ. Ведь пока персонажу некуда запрыгивать и неоткуда падать! Создайте несколько кубов, поменяйте их размер по собственному вкусу и расположите в сцене.

Вертикальная скорость и ускорение

Следующий листинг добавит к этому вектору движение по вертикали.

Добавление движения по вертикали в сценарий RelativeMovement:

```
...
public float jumpSpeed = 15.0f; public float gravity = -9.8f;
public float terminalVelocity = -10.0f; public float minFall = -1.5f;
private float _vertSpeed;
...
void Start() {
_vertSpeed = minFall;
...
}

void Update() {
...
if (_charController.isGrounded) {
if (Input.GetButtonDown("Jump")) {
_vertSpeed = jumpSpeed;
} else {
_vertSpeed = minFall;
}

} else {
_vertSpeed += gravity * 5 * Time.deltaTime;
if (_vertSpeed < terminalVelocity) {
_vertSpeed = terminalVelocity;
}
}
movement.y = _vertSpeed;
movement *= Time.deltaTime;
_charController.Move(movement);
}
}
```

Распознавание поверхности методом бросания лучей

```
...
private  ControllerColliderHit  _contact;
...
bool hitGround = false; RaycastHit hit;
if (_vertSpeed  <   0  &&       Проверяем, падает ли персонаж.
```

```
Physics.Raycast(transform.position, Vector3.down, out hit)) {
float check =
(_charController.height + _charController.radius) / 1.9f;
hitGround = hit.distance <= check;
}
if (hitGround) {
if (Input.GetButtonDown("Jump")) {
_vertSpeed = jumpSpeed;
} else {
_vertSpeed = minFall;
}
} else {
_vertSpeed += gravity * 5 * Time.deltaTime;
if (_vertSpeed < terminalVelocity) {_vertSpeed = terminalVelocity;
}
if (_charController.isGrounded) {
if (Vector3.Dot(movement, _contact.normal) < 0) { movement =
_contact.normal * moveSpeed;
} else {
movement += _contact.normal * moveSpeed;
}
}
}
movement.y = _vertSpeed;

movement *= Time.deltaTime;
_charController.Move(movement);
}

void OnControllerColliderHit(ControllerColliderHit hit) {
_contact = hit;
}
}
```

Этот листинг содержит практически тот же самый код, что и предыдущий; новый код перемешан с существующим сценарием движения, и крупные фрагменты предыдущего листинга включены с целью задания контекста. Первая строка добавляет новую переменную в верхнюю часть сценария RelativeMovement. Эта переменная используется для хранения данных о столкновении между функциями.

Анимация персонажа

Кроме более сложной формы, которая определяется сеточной геометрией, антропоморфный персонаж нуждается еще и в анимации. Персонажу следует назначить анимацию, которая заставит его руки и ноги двигаться взад и вперед.

В Unity встроена сложная система управления анимацией моделей. Она называется Mecanim. Имя Mecanim указывает, что это более новая, усовершенствованная система анимации, добавленная как замена старой версии. Последняя до сих пор доступна через настройку Legacy, но есть вероятность, что в следующих версиях Unity ее уже не будет.

Анимация, с которой мы будем работать, включена в тот же самый файл FBX, что и модель персонажа. Но система Mecanim дает возможность применять анимацию из других файлов FBX. Например, для всех противников можно использовать один набор анимационных клипов. Такой подход имеет ряд преимуществ, так как дает возможность хранить данные структурированно и экономить время за счет одновременной анимации.

Щелкните по кнопке Apply в нижней части панели Inspector, чтобы применить выбранные настройки к импортированной модели. Теперь можно продолжить работу над определением анимационных клипов.

Анимационные клипы для импортированной модели

Первым шагом по созданию анимации для персонажа будет задание отдельных клипов. У персонажа, напоминающего человека, разные движения совершаются в разное время. Наш персонаж будет то бегать по сцене, то запрыгивать на платформы, то просто стоять с опущенными руками. Каждое такое движение представляет собой независимый «клип».

Контроллер для анимационных клипов

Теперь для персонажа нужно создать контроллер-аниматор. На этом этапе мы определяем состояния анимации и создаем между ними переходы. В каждом состоянии воспроизводится свой набор клипов, а сценарии вызывают переключение контроллера от одного состояния к другому.

Применение дополнительного средства может показаться странным. Зачем между кодом и воспроизведением анимации нужна абст-

ракция в виде контроллера? Скорее всего, раньше вы работали с системами, в которых анимация воспроизводилась непосредственно из кода; кстати. Но у непрямого способа есть преимущество. Если раньше воспроизводить анимацию можно было только для той модели, с которой она была связана, теперь мы можем назначать одну и ту же анимацию разным моделям.

Контроллер-аниматор представляет собой дерево связанных узлов (именно этим объясняется рисунок на значке данного ресурса), просмотр и управление которыми осуществляются на вкладке Animator.

Изначально мы видим только два существующих по умолчанию узла: Entry и Any State. Узел Any State использоваться не будет. Создадим новые узлы, перетаскивая анимационные клипы.

2.6. Интерактивные устройства и элементы

Двери и другие устройства

Игровые уровни, как правило, состоят из статичных стен и предметов обстановки, к которым добавляются функциональные устройства. Многообразие устройств ограничено только вашим воображением, но практически все они пользуются одинаковым кодом, позволяющим персонажу активировать их. В этой главе мы рассмотрим пару примеров, что даст вам возможность адаптировать данный код к устройствам других видов.

Открывающиеся и закрывающиеся двери

Первый вид устройства, который нам предстоит запрограммировать, – это открывающаяся и закрывающаяся дверь. Начнем мы с управления дверью путем нажатия клавиши. В игру можно поместить множество устройств с разными способами управления. Из них дверь распространена шире всего, а управление с клавиатуры – наиболее очевидный подход, поэтому мы начнем именно с них.

Сценарий, по команде закрывающий и открывающий дверь

```
using UnityEngine;
using System.Collections;

public class DoorOpenDevice : MonoBehaviour {
[SerializeField]
```

```
private Vector3 dPos;
private bool _open;

public void Operate() {
```
Булева переменная для слежения за открытым состоянием двери.
```
if (_open)  {
```
Открываем или закрываем дверь в зависимости от ее состояния.
```
Vector3 pos = transform.position - dPos; transform.position = pos;
} else {
Vector3 pos = transform.position + dPos; transform.position = pos;
}
_open = !_open;
}
}
```

Проверка расстояния и направления

Создайте сценарий с именем DeviceOperator. Следующий код реализует управляющую клавишу для управления расположенными поблизости устройствами.

Клавиша управления устройством
```
using:
UnityEngine;
using System.Collections;

public class DeviceOperator : MonoBehaviour { public float radius =
1.5f;

void Update() {
if (Input.GetButtonDown("Fire3")) {
Collider[]  hitColliders  =  Physics.OverlapSphere(transform.

position, radius);
foreach (Collider hitCollider in hitColliders) {
hitCollider.SendMessage("Operate",
SendMessageOptions.DontRequireReceiver);
}
}
```

Осталась одна маленькая деталь. На данном этапе положение персонажа не имеет значения, главное, чтобы он располагался близко к двери. Но можно сделать так, чтобы дверь открывалась, только если персонаж стоит к ней лицом. Вот код, который нужно добавить в сценарий DeviceOperator.

Код, позволяющий открывать дверь, только стоя к ней лицом

```
...foreach (Collider hitCollider in hitColliders) {
Vector3 direction = hitCollider.transform.position - transform.position;
if (Vector3.Dot(transform.forward, direction) > .5f) { hitCollider.SendMessage("Operate",
SendMessageOptions.DontRequireReceiver);
}
}
```

Сообщение отправляется только при корректной ориентации персонажа.

Монитор, меняющий цвет

Итак, мы создали закрывающуюся и открывающуюся дверь. Схема управления, с которой вы познакомились в рамках этого упражнения, применима к устройствам любого вида. Для демонстрации создадим еще одно устройство; на этот раз – меняющий цвет монитор, прикрепленный к стене. Создайте новый куб и подвиньте его к стене практически до соприкосновения.

Сценарий устройства, которое может менять цвет

```
using UnityEngine;
using System.Collections;

public class ColorChangeDevice : MonoBehaviour public void Operate() {
Color random = new Color(Random.Range(0f,1f), Random.Range(0f,1f), Random.Range(0f,1f));
GetComponent<Renderer>().material.color = random;
}
```

Здесь мы объявили функцию с тем же самым именем, что и в сценарии управления дверью. Управляющий устройствами код всегда

содержит функцию с именем Operate – она нужна для активации. В данном случае ее код присваивает материалу объекта случайный цвет.

Итак, мы рассмотрели первый подход к управлению устройствами в играх и даже создали пару демонстрационных устройств. Но взаимодействовать с элементами сцены можно и другим способом. Например, врезаясь в них. Давайте посмотрим, как это делается.

Взаимодействие с объектами через столкновение

Существует и такой крайне эффективный подход, как реакция на столкновение с персонажем. При этом большую часть операций выполняет Unity благодаря встроенным в игровой движок механизмам распознавания столкновений и модели физической среды. Но Unity помогает только распознать столкновение, а запрограммировать реакцию объекта не сможет никто, кроме вас.

Мы рассмотрим три варианта реакции на столкновение, встречающиеся в играх:

• толчок и падение;

• срабатывание устройства;

• исчезновение в момент контакта (при сборе элементов);

• столкновение с препятствиями, обладающими физическими свойствами

Давайте создадим набор поставленных друг на друга коробок, которые при столкновении с персонажем будут разлетаться по сцене. Моделирование подобных вещей требует сложных физических расчетов, но эту задачу берут на себя встроенные функции Unity, которые и обеспечат нам реалистичный вид разлетающихся коробок.

По умолчанию объекты в Unity лишены физических свойств. Эта функция включается при добавлении компонента Rigidbody. Щелкните по кнопке Add Component и найдите строку Rigidbody в разделе Physics.

Создайте куб и добавьте к нему компонент Rigidbody. Нам требуется несколько таких кубов, положенных друг на друга.

Кубы готовы реагировать на приложенные к ним силы. Чтобы источником силы стал персонаж, внесите дополнение из следующего листинга в присоединенный к персонажу сценарий RelativeMovement.

Добавление физической силы в сценарий RelativeMovement

```
...public float pushForce = 3.0f; ...
void OnControllerColliderHit(ControllerColliderHit hit) {
```

_contact = hit;

Rigidbody body = hit.collider.attachedRigidbody; if (body != null && !body.isKinematic) {
body.velocity = hit.moveDirection * pushForce;
} }

Запустите игру и заставьте персонаж побежать прямо на кучу коробок. В момент столкновения куча вполне реалистично рассыплется. Как видите, имитация физического явления не потребовала от вас особых усилий! Благодаря встроенному в Unity механизму имитации физики, не приходится писать объемный код. Мы легко заставили объекты двигаться в ответ на столкновение, но возможен и другой вариант реакции – генерация события срабатывания. Давайте воспользуемся таким событием для управления дверью.

Сбор предметов

В играх часто встречаются предметы, которые может подбирать персонаж. Это оборудование, пакеты для восстановления здоровья и бонусы. Базовый механизм сбора путем столкновения очень прост; сложности начинаются после завершения этого процесса, но об этом мы поговорим чуть позже.

Создайте сферу и поместите ее в произвольном месте сцены на уровне талии персонажа. В настройках коллайдера установите флажок Is Trigger, в настройках слоя выберите вариант Ignore Raycast и создайте новый материал, чтобы присвоить объекту яркий цвет. Из-за малых размеров объекта мы не будем делать его полупрозрачным, поэтому ползунок, отвечающий за альфа- канал, трогать не нужно.

Создайте новый сценарий с именем CollectibleItem.

Сценарий, удаляющий элемент при контакте с персонажем

using UnityEngine;
using System.Collections;

public class CollectibleItem : MonoBehaviour { [SerializeField] private string itemName;

void OnTriggerEnter(Collider other) { Debug.Log("Item collected: " + itemName); Destroy(this.gameObject);
}

}

Это очень короткий и простой сценарий. Присвойте элементу значение name, чтобы в сцену можно было поместить несколько элементов. Метод OnTriggerEnter() вызывает исчезновение сферы. При этом на консоль выводится отладочное сообщение, которое мы впоследствии заменим кодом.

Перетаскивая шаблоны, расположите элементы на открытых участках уровня; для тестирования создавайте по несколько экземпляров одного элемента. Запустите игру и «соберите» элементы.

2.7. Звуковые эффекты и музыка

Когда речь заходит о видеоиграх, основное внимание уделяется графике, хотя существует и такой важный аспект, как звуковое сопровождение. В большинстве игр играет фоновая музыка и присутствуют звуковые эффекты. Соответственно, в Unity присутствует функциональность для их создания. Можно импортировать в Unity и воспроизводить аудиофайлы различных форматов, регулировать громкость звука и даже обрабатывать звуки, исходящие из определенной точки сцены.

Начнем мы с рассмотрения звуковых эффектов. Они представляют собой короткие аудиоклипы, воспроизводимые во время определенных действий. Звуковые клипы с музыкой имеют большую продолжительность, а их воспроизведение не привязано к конкретным событиям в игре. В конечном счете все сводится к одному виду аудиофайлов и одинаковому коду их воспроизведения, но тот факт, что файлы с музыкой обычно намного продолжительнее клипов со звуковыми эффектами, заслуживает, чтобы их выделили в отдельный раздел.

Импорт аудиофайлов

Собранную вами коллекцию аудиофайлов нужно импортировать в Unity.

Простой механизм импорта работает со всеми видами ресурсов: вы перетаскиваете файлы из папки на компьютере на вкладку Project в Unity.

Звуковые эффекты

Теперь нужно сделать так, чтобы добавленные к проекту звуковые файлы зазвучали. Код активации звуковых эффектов понять несложно, но аудиосистема в Unity состоит из фрагментов, которые должны работать согласованно.

Система воспроизведения: клипы, источник, подписчик

Возможно, вы ожидаете, что для проигрывания звука в Unity достаточно указать, какие клипы нужно воспроизводить, но на самом деле для этого нужно задать три компонента: AudioClip, AudioSource и AudioListener.

Зацикленный звук

Аудиоклипы уже импортированы, у используемой по умолчанию камеры есть компонент AudioListener, так что остается назначить только компонент AudioSource. Добавим треск огня к шаблону Enemy – это персонаж, который хаотично перемещается по сцене.

Установите флажки Play On Awake и Loop. Флажок Play On Awake заставляет источник звука начать воспроизведение сразу после загрузки сцены. Флажок Loop заставляет источник звука играть непрерывно, повторяя клип снова и снова.

Теперь проверьте, включен ли звук у вас в колонках, и запустите игру. Вы услышите исходящее от врага потрескивание, которое ослабевает по мере его удаления от персонажа, так как в сцене используется источник 3D-звука.

Активация звуковых эффектов из кода

Настройка компонента AudioSource на автоматическое воспроизведение хорошо подходит для циклических звуков, но в большинстве случаев нужно, чтобы звуковые эффекты возникали в ответ на команды кода. Для этого все равно требуется компонент AudioSource, но воспроизведение звука запускается программно.

Добавьте компонент AudioSource к объекту player. Добавьте в отвечающий за стрельбу сценарий RayShooter код следующего листинга.

Добавление в сценарий RayShooter звуковых эффектов

```
...
[SerializeField]
```

```
private AudioSource soundSource;
[SerializeField]
private AudioClip hitWallSound;
[SerializeField]
 private AudioClip hitEnemySound;
...
if (target != null) { target.ReactToHit();
 soundSource.PlayOneShot(hitEnemySound);
} else {
StartCoroutine(SphereIndicator(hit.point)); soundSource.PlayOneShot(hitWallSound);
}
...
```

Интерфейс управления звуком

Продолжая доработку, добавим туда диспетчер AudioManager. На этот раз будет создан диспетчер управления звуком, который мы тоже добавим в список. Этот центральный модуль позволит регулировать громкость звука в игре и даже выключать звук совсем.

Центральный диспетчер управления звуком

Для настройки диспетчера AudioManager первым делом нужно добавить в проект фреймворк Managers.

Создайте новый сценарий AudioManager, на который может ссылаться код сценария Managers.

Заготовка кода для сценария AudioManager

```
using UnityEngine;
using System.Collections;
using System.Collections.Generic;

public class AudioManager : MonoBehaviour, IGameManager {
public ManagerStatus status {get; private set;} private NetworkService _network;
public void Startup(NetworkService service) { Debug.Log("Audio manager starting...");

_network = service;
```

```
status = ManagerStatus.Started;
}
}
```

Сценарий Managers после добавления сценария AudioManager

```
using UnityEngine;
using System.Collections;
using System.Collections.Generic; [RequireComponent(typeof(AudioManager))]
public class Managers : MonoBehaviour {
public static AudioManager Audio {get; private set;} private List<IGameManager> _startSequence;
void Awake() {
Audio = GetComponent<AudioManager>(); В этом проекте в списке
AudioManager, а не PlayerManager, и т. п.
_startSequence = new List<IGameManager>();
_startSequence.Add(Audio);

StartCoroutine(StartupManagers());
}

private IEnumerator StartupManagers() { NetworkService network = new NetworkService();
foreach (IGameManager manager in _startSequence) { manager.Startup(network);
}

yield return null;

int numModules = _startSequence.Count; int numReady = 0;
while (numReady < numModules) { int lastReady = numReady; numReady = 0;
foreach (IGameManager manager in _startSequence) { if (manager.status == ManagerStatus.Started) {
numReady++;
}
}
```

```
if (numReady > lastReady)
Debug.Log("Progress: " + numReady + "/" + numModules);
yield return null;
}

Debug.Log("All managers started up");
}
}
```

Создайте пустой объект, который будет играть роль диспетчера, и свяжите с ним сценарии Managers и AudioManager. При воспроизведении игры на консоли появятся сообщения о запуске диспетчеров, но диспетчер управления звуком пока не несет никакой функциональной нагрузки.

Интерфейс для управления громкостью

Создаем в сцене холст. Устанавливаем для холста флажок Pixel Perfect. Присваиваем объекту имя HUD Canvas и переключаемся в режим работы с двумерной графикой. Создаем связанное с холстом изображение. Присваиваем новому объекту имя Settings Popup. Перетаскиваем на ячейку Source Image этого объекта спрайт popup. В раскрывающемся списке Image Type выбираем вариант Sliced и устанавливаем флажок Fill Center. Помещаем изображение всплывающего окна в точку с координатами 0, 0. Меняем размеры всплывающего окна до 250 по ширине и 150 по высоте.

Создаем кнопку. Делаем кнопку дочерним объектом по отношению к всплывающему окну. Помещаем кнопку в точку с координатами 0, 40. Раскрываем иерархический список кнопки, чтобы выделить связанную с ней текстовую метку. Меняем текст на Toggle Sound.

Создаем ползунок. Делаем ползунок дочерним объектом всплывающего окна и помещаем его в точку с координатами 0, 15. Присваиваем параметру Value ползунка (в нижней части панели Inspector) значение 1.

Теперь напишем для всплывающего окна код.

Добавление в сценарий AudioManager средств регулировки громкости звука:

```
...
public float soundVolume {Свойство для громкости с функцией
```

чтения и функцией доступа.

```
get {return AudioListener.volume;} set {AudioListener.volume = value;}
}

public bool soundMute { get {return AudioListener.pause;} set {AudioListener.pause = value;}
}
public void Startup(NetworkService service) { Debug.Log("Audio manager starting...");

_network = service;

soundVolume = 1f;

status = ManagerStatus.Started;
}
...
```

В сценарий AudioManager добавлены свойства soundVolume и soundMute. Функция чтения и задающая функция для этих свойств реализованы с помощью глобальных переменных класса AudioListener. Класс AudioListener может регулировать громкость всех звуков, получаемых всеми экземплярами компонента AudioListener. Задание свойства soundVolume в сценарии AudioManager оказывает такой же эффект, как задание громкости в компоненте AudioListener.

После добавления этих методов в сценарий AudioManager можно написать сценарий для всплывающего окна.

Сценарий SettingsPopup с элементами управления громкостью:

```
using UnityEngine;
using System.Collections;

public class SettingsPopup : MonoBehaviour {

public void OnSoundToggle() {
Managers.Audio.soundMute = !Managers.Audio.soundMute;
}
```

public void OnSoundValue(float volume) { Managers.Audio.soundVolume = volume;

В этом сценарии мы видим два метода, влияющие на свойства объекта AudioManager: метод OnSoundToggle() задает свойство soundMute, а метод OnSoundValue() – свойство soundVolume.

Элементы управления теперь работают, но в проект нужно внести небольшие коррективы. Сейчас экран все время закрывает всплывающее окно. Сделаем так, чтобы оно открывалось только при нажатии клавиши M.

Сценарий UIController, вызывающий и скрывающий всплывающее окно

```
using UnityEngine;
using System.Collections;

public class UIController : MonoBehaviour { [SerializeField] private SettingsPopup popup;
void Start() { popup.gameObject.SetActive(false);
}

void Update() {
if (Input.GetKeyDown(KeyCode.M)) {
bool isShowing = popup.gameObject.activeSelf; popup.gameObject.SetActive(!isShowing);

if (isShowing) {
Cursor.lockState = CursorLockMode.Locked; Cursor.visible = false;
} else {
Cursor.lockState = CursorLockMode.None; Cursor.visible = true;
```

Фоновая музыка

Вставим в игру фоновую музыку. Для этого ее нужно добавить в диспетчер AudioManager.

Во-первых, для музыкальных треков, как правило, требуется много компьютерной памяти, и этот параметр необходимо оптимизировать. Нужно отслеживать два аспекта: загрузка музыки в память до того, как она начнет использоваться, и потребление слишком большого объема памяти при загрузке.

Сейчас нас интересует загрузка музыкальных клипов по требованию. Без этого музыка займет слишком много места в памяти, причем еще до того, как ею начнут пользоваться.

Загрузка по требованию (lazy-loading) означает, что загрузка файла откладывается до момента, когда этот файл понадобится. Заранее загруженные данные реагируют быстрее, но загрузка по требованию экономит память в ситуациях, когда быстрая реакция не имеет особого значения.

Второй способ решения проблемы с потреблением памяти связан с потоковой передачей музыки с диска.

В конечном счете подготовка к воспроизведению музыки делится на несколько этапов, в том числе включающих в себя оптимизацию потребления памяти.

Музыкальные циклы

Процесс воспроизведения музыки представляет собой уже знакомую последовательность шагов:

– импортируем аудиоклипы;

– настроим компонент AudioSource для использования диспетчером AudioManager;

– напишем код для воспроизведения аудиоклипа в диспетчере AudioManager;

– добавим элементы управления музыкой в пользовательский интерфейс.

Для звуковых эффектов следует использовать несжатые аудиофайлы, а для фоновой музыки – сжатые, но в качестве исходного материала в обоих случаях можно брать файлы формата WAV, так как Unity умеет сжимать импортированный звук. Для аудиоклипов возможен как 2D-звук, всегда воспроизводимый одинаково, так и 3D-звук, громкость которого зависит от положения слушателя.

Громкость звуковых эффектов легко регулируется на глобальном уровне с помощью компонента AudioListener. Можно по отдельности регулировать громкость отдельных источников звука.

2.8. Объединение фрагментов в готовую игру

Рассмотрим в качестве проекта ролевой боевик. В таких играх камера располагается сверху и смотрит четко вниз. Возможно, вы знакомы с игрой Diablo, которая представляет собой как раз ролевой боевик.

Сборка ресурсов и кода из разных проектов

Первым делом следует обновить фреймворк диспетчеров и добавить в проект противников, управляемых компьютером.

Обновление фреймворка диспетчеров

Проще всего обновить диспетчеры, поэтому эту задачу мы решим первой.

Отредактированный интерфейс IGameManager

public interface IGameManager { ManagerStatus status {get;}

void Startup(NetworkService service);
}

Изменения в сценарии Managers

...
private IEnumerator StartupManagers() { Исправления в начале метода.
NetworkService network = new NetworkService();

foreach (IGameManager manager in _startSequence) { manager.Startup(network);
}
...

Напоследок отредактируем сценарии InventoryManager и PlayerManager с учетом внесенных в интерфейс изменений. Следующий листинг демонстрирует исправления в сценарии InventoryManager; аналогичные правки, но с другими именами, вносятся и в сценарий PlayerManager.

Изменения в сценарии InventoryManager с учетом модификаций в IGameManager

...
private NetworkService _network;

public void Startup(NetworkService service) { Debug.Log("Inventory manager starting...");
_network = service;

_items = new Dictionary<string, int>();

После этих небольших изменений игра должна функционировать так же, как и раньше. Мы скорректировали внутренние особенности, а игровой процесс остался без изменений.

Копирование противников, оснащенных искусственным интеллектом

Первым делом скопируйте все сценарии. Заодно импортируем в проект материал Flame и шаблоны Fireball и Enemy.

К объекту-контроллеру добавьте сценарий SceneController.cs и перетащите шаблон Enemy на одноименную ячейку компонента панели Inspector. Кроме того, свяжите сценарий PlayerCharacter.cs с объектом player, чтобы противники начали атаковать игрока.

Запустите игру и посмотрите, как двигается противник. Огненные шары летят в персонажа, пока не причиняя особого вреда; выделите шаблон Fireball и присвойте его параметру Damage значение 10.

Ранее здоровье персонажа фигурировало только в качестве тестового атрибута. Но теперь в игре есть диспетчер персонажа, а значит, можно отредактировать сценарий PlayerCharacter, добавив в него средства для управления здоровьем.

Добавляем в сценарий PlayerCharacter возможность использовать здоровье в диспетчере игрока

using UnityEngine;
using System.Collections;

public class PlayerCharacter : MonoBehaviour { public void Hurt(int damage) {
Managers.Player.ChangeHealth(-damage);
}}

Итак, мы собрали из фрагментов ранее выполненных проектов демонстрационный ролик. В сцене появился противник, что сделало игру более захватывающей. Но элементы управления и угол обзора до сих пор такие же, как и в демонстрационном ролике от третьего лица. Нужно создать для нашей ролевой игры элементы управления с помощью мыши (point-and-click controls).

Элементы управления с помощью мыши

Демонстрационному ролику требуется камера, нацеленная сверху вниз, и управление перемещениями персонажа с помощью мыши.

В настоящее время мышь управляет камерой, в то время как перемещения персонажа контролируются с клавиатуры, то есть это диаметрально противоположно тому, что нам нужно. Кроме того, мы заставим реагировать на щелчки мыши меняющий цвета монитор. Огромного количества правок в обоих случаях код не требует, поэтому давайте просто возьмем и внесем необходимые коррективы в сценарии движения и устройства.

Обзор сцены сверху вниз

Первым делом присвойте координате Y камеры значение 8, чтобы поднять ее над сценой. Кроме того, давайте уберем из сценария OrbitCamera управление с помощью мыши и оставим в качестве элементов управления только клавиши со стрелками.

Удаление средств управления с помощью мыши из сценария OrbitCamera

```
...
void LateUpdate() {
_rotY -= Input.GetAxis("Horizontal") * rotSpeed;        Меняем
направление на обратное.
Quaternion rotation = Quaternion.Euler(0, _rotY, 0); trans-
form.position = target.position - (rotation * _offset); trans-
form.LookAt(target);
}
...
```

Код движения

Основная идея этого кода сводится к автоматическому перемещению персонажа в указанную точку. Эта точка задается щелчком мыши. При этом код, перемещающий персонажа, напрямую на мышь не реагирует, движение персонажа косвенно управляется при помощи щелчков.

Создайте сценарий PointClickMovement и скопируйте в него код сценария RelativeMovement.

Новый код движения в сценарии PointClickMovement

```
public class PointClickMovement : MonoBehaviour {   Испра-
вим имя после вставки кода.
...
```

```
public float deceleration = 25.0f; public float targetBuffer = 1.5f;
private float _curSpeed = 0f;
private Vector3 _targetPos = Vector3.one;
...
void Update() {
Vector3 movement = Vector3.zero;

if (Input.GetMouseButton(0)) {
Ray ray = Camera.main.ScreenPointToRay(Input.mousePosition);
RaycastHit mouseHit;
if (Physics.Raycast(ray, out mouseHit)) {
_targetPos = mouseHit.point; _curSpeed = moveSpeed;
}
}
if (_targetPos != Vector3.one) {
if (_curSpeed > moveSpeed * .5f) {
Vector3 adjustedPos = new Vector3(_targetPos.x, trans-
form.position.y,
_targetPos.z);
Quaternion targetRot = Quaternion.LookRotation(adjustedPos -
transform.position);
transform.rotation = Quaternion.Slerp(transform.rotation, targetRot,
rotSpeed * Time.deltaTime);
}
movement = _curSpeed * Vector3.forward;
movement = transform.TransformDirection(movement);

if (Vector3.Distance(_targetPos, transform.position) < targetBuffer)
{_curSpeed -= deceleration * Time.deltaTime;
if (_curSpeed <= 0) {
_targetPos = Vector3.one;
}
}
_animator.SetFloat("Speed", movement.sqrMagnitude);
...
```

Управление устройствами с помощью мыши

Для этого создадим сценарий, от которого будут наследовать все устройства; именно туда мы поместим процедуру управления по-

средством мыши. Присвойте новому сценарию имя BaseDevice и скопируйте в него код следующего листинга.

Сценарий BaseDevice, срабатывающий по щелчку мыши

```
using UnityEngine;
using System.Collections;

public class BaseDevice : MonoBehaviour { public float radius =
3.5f;

void OnMouseDown() {        Функция, запускаемая щелчком.
Transform player = GameObject.FindWithTag("Player").transform;
if (Vector3.Distance(player.position, transform.position) < radius) {
Vector3 direction = transform.position - player.position;
if (Vector3.Dot(player.forward, direction) > .5f) {
Operate();
}
}
}
public virtual void Operate() {
// здесь код поведения конкретного устройства
}
}
```

Большая часть операций выполняется внутри метода OnMouseDown(), так как именно его вызывает класс MonoBehaviour после щелчка по объекту. Первым делом проверяется расстояние до персонажа, а затем с помощью скалярного произведения определяется, повернут ли он в сторону устройства. Метод Operate() пока представляет собой пустую оболочку, которая будет заполняться кодом устройств, наследующих данный сценарий.

Добавляем в сценарий ColorChangeDevice код наследования от сценария BaseDevice:

```
using UnityEngine;
using System.Collections;
public class ColorChangeDevice : BaseDevice {
public override void Operate() {
```

```
Color random = new Color(Random.Range(0f,1f), Random.Range(0f,1f), Random.Range(0f,1f));
GetComponent<Renderer>().material.color = random;
}
}
```

Теперь устройство управляется щелчками мыши. Кроме того, у персонажа был удален компонент сценария DeviceOperator, так как этот сценарий задает управление устройством с клавиатуры.

К сожалению, новый вариант управления устройством конфликтует с элементами управления перемещениями.

Корректировка кода, обрабатывающего щелчки мышью, в сценарии:

```
PointClickMovement
...
Ray ray = Camera.main.ScreenPointToRay(Input.mousePosition); RaycastHit mouseHit;
if (Physics.Raycast(ray, out mouseHit)) {
GameObject hitObject = mouseHit.transform.gameObject; if (hitObject.layer == LayerMask.NameToLayer("Ground")) {
_targetPos = mouseHit.point;
_curSpeed = moveSpeed;
}
}
...
```

Запустите игру и убедитесь, что щелчки на меняющем цвет мониторе не приводят персонажа в движение.

3. Мобильные и мультиплатформенные игры

3.1. Создание двумерной игры

«Колодец с сокровищами»

Дизайн игры

Игровой процесс Колодца с сокровищами простейший. Игрок управляет спуском гномика в колодец на веревочке. На дне колодца находятся сокровища. Сложность в том, что колодец наполнен ловушками, которые убивают гномика, если он касается их.

Ниже перечислены шаги, которые нужно сделать, чтобы создать игру.

Сначала создадим гномика, используя временные схематические изображения. Мы подготовим куклу и подключим спрайты.

Далее мы создадим веревку. При этом мы напишем первый значительный фрагмент кода, потому что веревка будет генерироваться во время выполнения, и нам потребуется организовать возможность ее удлинения и укорочения.

Когда веревка будет готова, мы перейдем к созданию системы ввода. Эта система будет принимать информацию о наклоне устройства и делать ее доступной для других частей игры (в частности, для гномика). Одновременно мы займемся пользовательским интерфейсом игры и создадим кнопки для удлинения и укорочения веревки.

Закончив с гномиком, веревкой и системой ввода, мы займемся созданием самой игры: реализуем ловушки и сокровище и попробуем поиграть.

Создание гномика

Поскольку гномик будет состоять из нескольких объектов, движущихся независимо, сначала нужно создать объект, который послужит контейнером для каждой части. Этому объекту также надо присвоить тег Player, потому что механизму определения столкновений, который используется для определения касаний гномиком ловушек, сокровища и выхода из уровня, нужно знать, что объект является специальным объектом Player. Чтобы создать гномика, выполните следующие действия.

Создайте объект Prototype Gnome. Создайте новый пустой игровой объект, открыв меню GameObject (Игровой объект) и выбрав пункт Create Empty (Создать пустой).

Дайте новому объекту имя «Prototype Gnome», а затем присвойте ему тег Player, выбрав Player в раскрывающемся списке Tag (Тег), находящемся в верхней части инспектора.

Установите спрайты как дочерние объекты для объекта Prototype Gnome. В панели иерархии выберите все только что добавленные спрайты и перетащите их на пустой объект Prototype Gnome.

Добавьте компоненты Rigidbody 2D в части тела. Добавьте коллайдеры в части тела. Выберите все спрайты, кроме тела. Добавьте компонент HingeJoint2D во все выбранные спрайты. Настройте сочленения.

Добавьте ограничения для сочленений. Обновите опорные точки сочленений. Добавьте сочленение с веревкой. Добавьте в него пружинное сочленение.

Запустите игру. Сразу после запуска гномик повиснет в середине экрана.

Веревка

Веревка – первый элемент игры, для реализации которого потребуется писать программный код. Это работает так: веревка является коллекцией игровых объектов, каждый из которых имеет твердое тело и пружинное сочленение. Каждое пружинное сочленение соединено со следующим объектом Rope Segment, который в свою очередь соединен со следующим и так далее до начала веревки вверху. Самый верхний объект Rope Segment соединен с неподвижным твердым телом. Другой конец веревки соединен с одним из компонентов гномика: объектом Leg Rope.

Создайте шаблон, используя этот объект. Удалите исходный объект Rope Segment. Создайте новый пустой игровой объект и дайте ему имя Rope. Добавьте твердое тело. Добавьте визуализатор (renderer).

Код реализации веревки

```
using UnityEngine;
using System.Collections;
using  System.Collections.Generic;
public class Rope : MonoBehaviour {

GameObject ropeSegmentPrefab;
```

```
List<GameObject> ropeSegments = new List<GameObject>();
public bool isIncreasing { get; set; } public bool isDecreasing { get;
set; }

public Rigidbody2D connectedObject;
public float maxRopeSegmentLength = 1.0f;
public float ropeSpeed = 4.0f;
LineRenderer lineRenderer; void Start() {
lineRenderer = GetComponent<LineRenderer>();
ResetLength();
}
public void ResetLength() {
foreach (GameObject segment in ropeSegments) {
Destroy (segment);
}
ropeSegments = new List<GameObject>();
isDecreasing = false; isIncreasing  = false;
CreateRopeSegment();
}
void CreateRopeSegment() {
GameObject segment = (GameObject)Instantiate(
ropeSegmentPrefab, this.transform.position, Quaternion.identity);
segment.transform.SetParent(this.transform, true);
Rigidbody2D  segmentBody  =  segment
.GetComponent<Rigidbody2D>();
SpringJoint2D                segmentJoint                =                seg-
ment.GetComponent<SpringJoint2D>();
if (segmentBody == null || segmentJoint == null) {

Debug.LogError("Rope segment body prefab has no " + "Rigid-
body2D and/or SpringJoint2D!");
return;
}
ropeSegments.Insert(0, segment);
if (ropeSegments.Count == 1) {
SpringJoint2D connectedObjectJoint =
connectedObject.GetComponent<SpringJoint2D>();

connectedObjectJoint.connectedBody
```

```
= segmentBody;
connectedObjectJoint.distance = 0.1f;
segmentJoint.distance = maxRopeSegmentLength;
} else {
GameObject nextSegment = ropeSegments[1];
SpringJoint2D nextSegmentJoint = nextSeg-
ment.GetComponent<SpringJoint2D>();
nextSegmentJoint.connectedBody = segmentBody;
segmentJoint.distance = 0.0f;
}
segmentJoint.connectedBody =
this.GetComponent<Rigidbody2D>();
}
void RemoveRopeSegment() {
if (ropeSegments.Count < 2) {
return;
}
GameObject topSegment = ropeSegments[0]; GameObject nextSeg-
ment = ropeSegments[1];
SpringJoint2D nextSegmentJoint = nextSeg-
ment.GetComponent<SpringJoint2D>();
nextSegmentJoint.connectedBody =
this.GetComponent<Rigidbody2D>();
ropeSegments.RemoveAt(0); Destroy (topSegment);
}
void Update() {
GameObject topSegment = ropeSegments[0]; SpringJoint2D top-
SegmentJoint =
topSegment.GetComponent<SpringJoint2D>(); if (isIncreasing) {
if (topSegmentJoint.distance >= maxRopeSegmentLength) {
CreateRopeSegment();
} else {
topSegmentJoint.distance += ropeSpeed * Time.deltaTime;
}}
if (isDecreasing) {
if (topSegmentJoint.distance <= 0.005f) { RemoveRopeSegment();
} else {
topSegmentJoint.distance -= ropeSpeed *
Time.deltaTime;
```

```
}}
if (lineRenderer != null) {
lineRenderer.positionCount
= ropeSegments.Count + 2;
lineRenderer.SetPosition(0, this.transform.position);
for (int i = 0; i < ropeSegments.Count; i++) { lineRen-
derer.SetPosition(i+1,
ropeSegments[i].transform.position);
}
SpringJoint2D connectedObjectJoint = connectedOb-
ject.GetComponent<SpringJoint2D>();
lineRenderer.SetPosition( ropeSegments.Count + 1, connectedOb-
ject.transform.
TransformPoint(connectedObjectJoint.anchor)
);}}}
```

Unity Remote

Для быстрой передачи ввода в редактор разработчики Unity разместили в онлайн- магазине приложений App Store мобильное приложение под названием Unity Remote. Unity Remote соединяется с Unity Editor посредством кабеля для телефона; когда редактор проигрывает игру, на экране телефона отображается копия окна Game (Игра) и вашему сценарию посылаются обратно все касания и информация с датчиков. Это позволяет тестировать игру без необходимости ее сборки – достаточно лишь запустить приложение на телефоне, и можно играть в игру, как если бы она была установлена.

Добавление управления наклоном

Поддержка управления игрой наклоном устройства будет реализована в двух сценариях: InputManager (читающий информацию с датчика акселерометра) и Swinging (получающий входные данные от InputManager и применяющий боко- вую силу к телу – в данном случае к телу гномика).

Создание класса Singleton

Диспетчер ввода InputManager – это объект-одиночка. То есть в сцене всегда будет присутствовать единственный объект InputManager, а все другие объекты будут обращаться к нему. Позднее нам также понадобится добавлять в игру другие виды объектов-

одиночек, поэтому есть смысл создать класс, который потом мы сможем повторно использовать в разных сценариях. Для создания класса Singleton, который будет использован для создания диспетчера ввода InputManager, выполните следующие действия.

Добавьте код в сценарий Singleton. using UnityEngine;

```
using System.Collections;
public class Singleton<T> : MonoBehaviour where T : MonoBehaviour {
private static T _instance;
public static T instance { get {
if (_instance == null)
{
_instance = FindObjectOfType<T>();
if (_instance == null) {
Debug.LogError("Can't find " +
typeof(T) + "!");
}}
return _instance;
}}}
```

Реализация объекта-одиночки InputManager

Теперь создадим класс InputManager, наследующий Singleton. Создайте игровой объект InputManager. Создайте и добавьте сценарий InputManager. Добавьте код в сценарий InputManager.cs. Откройте только что созданный файл

InputManager.cs и добавьте в него следующий код:

```
using UnityEngine;
using System.Collections;
public class InputManager : Singleton<InputManager> {
private float _sidewaysMotion = 0.0f;
public float sidewaysMotion { get {
return _sidewaysMotion;
}}
void Update () {
Vector3 accel = Input.acceleration;
_sidewaysMotion = accel.x;
}}
```

Выберите объект Body тела гномика.

Создайте и добавьте новый сценарий на C# с именем Swinging. Добавьте в файл сценария Swinging.cs следующий код:

```
using UnityEngine;
using System.Collections;
public class Swinging : MonoBehaviour {
public float swingSensitivity = 100.0f;
void FixedUpdate() {
if (GetComponent<Rigidbody2D>() == null) { Destroy (this);
return;
}
float swing = InputManager.instance.sidewaysMotion;
Vector2 force =
new Vector2(swing * swingSensitivity, 0);
GetComponent<Rigidbody2D>().AddForce(force);
}}
```

Управление веревкой

Теперь добавим кнопки, удлиняющие и укорачивающие веревку. Для этого нам понадобятся две кнопки графического интерфейса Unity; нажимая и удерживая кнопку Down, пользователь будет сообщать, что веревка должна удлиниться, а когда пользователь перестает удерживать кнопку, веревка должна перестать удлиняться. Кнопка Up действует аналогично и управляет началом и концом укорачивания веревки.

Добавьте событие Pointer Up и заставьте его записывать false в свойство isIncreasing объекта Rope.

Теперь, если удерживать нажатой кнопку Down, гномик будет опускаться на веревке все ниже и ниже и, наконец, исчезнет из поля зрения. Чтобы этого не произошло, камера должна следовать за гномиком.

Для этого создадим сценарий, который будет подключаться к камере и корректировать ее координату Y в соответствии с положением другого объекта. Назначив этим другим объектом объект Gnome, мы обеспечим синхронное следование камеры за гномиком. Сценарий будет подключаться к камере и настраиваться на следование за телом гномика. Чтобы создать сценарий, выполните следующие действия.

Добавьте сценарий CameraFollow.

Добавьте следующий код в файл сценария CameraFollow.cs:

```
public class CameraFollow : MonoBehaviour {
public Transform target;
public float topLimit = 10.0f;
public float bottomLimit = -10.0f;
public float followSpeed = 0.5f;
void LateUpdate () {
if (target != null) {
Vector3 newPosition = this.transform.position;
newPosition.y = Mathf.Lerp (newPosition.y, target.position.y, followSpeed);
newPosition.y = Mathf.Min(newPosition.y, topLimit);
newPosition.y  = Mathf.Max(newPosition.y, bottomLimit);
transform.position = newPosition;
}}
void OnDrawGizmosSelected() { Gizmos.color = Color.yellow;
Vector3 topPoint =
new Vector3(this.transform.position.x, topLimit, this.transform.position.z);
Vector3 bottomPoint =
new Vector3(this.transform.position.x, bottomLimit, this.transform.position.z);
Gizmos.DrawLine(topPoint, bottomPoint);
}}
```

Код реализации поведения гномика

Настал момент заняться самим гномиком. Гномик должен хранить некоторую информацию о своем состоянии и уметь определять, что с ним происходит.

Создайте файл BodyPart.cs. Создайте новый сценарий на языке C# с именем BodyPart.cs. Добавьте в него следующий код:

```
[RequireComponent (typeof(SpriteRenderer))] public class BodyPart : MonoBehaviour {
public Sprite detachedSprite;
public Sprite burnedSprite;
public Transform bloodFountainOrigin;
bool detached = false;
public void Detach() { detached = true;
this.tag = "Untagged"; transform.SetParent(null, true);
}
```

```
public void Update() {
if (detached == false) { return;
}
var rigidbody = GetComponent<Rigidbody2D>(); if (rigid-
body.IsSleeping()) {
foreach (Joint2D joint in GetComponentsInChildren<Joint2D>()) {
Destroy (joint);
}
foreach (Rigidbody2D body in GetComponentsInChil-
dren<Rigidbody2D>()) { Destroy (body);
}
foreach (Collider2D collider in GetComponentsInChil-
dren<Collider2D>()) { Destroy (collider);
}
Destroy (this);
}}
public void ApplyDamageSprite( Gnome.DamageType dam-
ageType) {
Sprite spriteToUse = null; switch (damageType) {
case Gnome.DamageType.Burning: spriteToUse = burnedSprite;
break;
case Gnome.DamageType.Slicing: spriteToUse = detachedSprite;
break;
}
if (spriteToUse != null) {
GetComponent<SpriteRenderer>().sprite = spriteToUse;
}}}
```

Создайте сценарий Gnome. Создайте новый файл Gnome.cs сценария на языке C#.

Добавьте код для компонента Gnome. Добавьте следующий код в файл Gnome.cs:

```
public class Gnome : MonoBehaviour {
public Transform cameraFollowTarget; public Rigidbody2D rope-
Body;
public Sprite armHoldingEmpty; public  Sprite armHoldingTreasure;
public  SpriteRenderer holdingArm;
public GameObject deathPrefab; public GameObject flameDeathPre-
fab; public GameObject ghostPrefab;
```

```
public float delayBeforeRemoving = 3.0f;
public float delayBeforeReleasingGhost = 0.25f;
public GameObject bloodFountainPrefab;
bool dead = false;
bool _holdingTreasure = false; public bool holdingTreasure {
get {
return _holdingTreasure;
}
set {
if (dead == true) { return;
}
_holdingTreasure  =  value;
if (holdingArm != null) { if (_holdingTreasure) {
holdingArm.sprite = armHoldingTreasure;
} else { holdingArm.sprite =
armHoldingEmpty;
}}}}
public enum DamageType { Slicing,
Burning
}
public void ShowDamageEffect(DamageType type) { switch (type) {
case DamageType.Burning:
if (flameDeathPrefab != null) { Instantiate(
flameDeathPrefab,cameraFollowTarget.position,  cameraFollowTar-
get.rotation
);}
break;
case DamageType.Slicing:
if (deathPrefab != null) { Instantiate(
deathPrefab,  cameraFollowTarget.position,  cameraFollowTar-
get.rotation
);}
break;
}}
public void DestroyGnome(DamageType type) { holdingTreasure =
false;
dead = true;
foreach (BodyPart part in GetComponentsInChildren<BodyPart>())
{
```

```
switch (type) {
case DamageType.Burning:
bool shouldBurn = Random.Range (0, 2) == 0; if (shouldBurn) {
part.ApplyDamageSprite(type);
}
break;
case DamageType.Slicing:
part.ApplyDamageSprite (type);
break;
}
bool shouldDetach = Random.Range (0, 2) == 0; if (shouldDetach) {
part.Detach ();
if (type == DamageType.Slicing) {
if (part.bloodFountainOrigin != null && bloodFountainPrefab !=
null) {
GameObject fountain = Instantiate(
bloodFountainPrefab, part.bloodFountainOrigin.position,
part.bloodFountainOrigin.rotation
) as GameObject;
fountain.transform.SetParent( this.cameraFollowTarget, false
);}}
var allJoints = part.GetComponentsInChildren<Joint2D>();
foreach (Joint2D joint in allJoints) { Destroy (joint);
}}}
var remove = gameObject.AddComponent<RemoveAfterDelay>();
remove.delay = delayBeforeRemoving;
StartCoroutine(ReleaseGhost());
}
IEnumerator ReleaseGhost() {
if (ghostPrefab == null) { yield break;
}
yield return new WaitForSeconds(delayBeforeReleasingGhost);
Instantiate(ghostPrefab, transform.position, Quaternion.identity
);}}
```

Добавьте компонент сценария BodyPart во все части тела гномика. Добавьте контейнер для фонтанов крови.

Создайте сценарий RemoveAfterDelay. Создайте новый сценарий на C# в файле с именем RemoveAfterDelay.cs. Добавьте в него следующий код:

```
public class RemoveAfterDelay : MonoBehaviour {
public float delay = 1.0f;
void Start () {
StartCoroutine("Remove");
}
IEnumerator Remove() {
yield return new WaitForSeconds(delay); Destroy (gameObject);
}}
```

Подготовка диспетчера игры

Диспетчер игры – это объект, отвечающий за управление игрой в целом. Он создает гномика в начале игры, обрабатывает события касания гномиком важных объектов, таких как ловушки, сокровище или событие выхода из уровня, и имеет дело со всем, что существует дольше отдельно взятого гномика.

Перед тем как приступить к реализации диспетчера игры, мы должны добавить класс, от которого он зависит: класс Resettable.

Нам нужен универсальный способ запуска кода, когда потребуется сбросить игру в исходное состояние. Один из таких способов основан на событиях Unity – мы создадим сценарий Resettable.cs с полем Unity Event, который можно будет подключить к любому объекту, требующему переустановки в исходное состояние. Когда потребуется сбросить игру, диспетчер отыщет все объекты с компонентом Resettable и вызовет Unity Event.

При таком подходе можно настраивать отдельные объекты так, что они сами будут приводить себя в исходное состояние, без необходимости писать код для каждого из них.

Создайте сценарий Resettable. Добавьте новый сценарий на C# с именем

Resettable.cs и поместите в него следующий код:

```
using UnityEngine.Events;
public class Resettable : MonoBehaviour {
public UnityEvent onReset;
public void Reset() {
onReset.Invoke();
}}
```

Теперь можно создать диспетчер игры – объект Game Manager.
Создайте объект Game Manager.
Создайте и добавьте в него код GameManager.

```
public class GameManager : Singleton<GameManager> {
public GameObject startingPoint;
public Rope rope;
public CameraFollow cameraFollow;
Gnome currentGnome;
public GameObject gnomePrefab;
public RectTransform mainMenu;
public RectTransform gameplayMenu;
public RectTransform gameOverMenu;
public bool gnomeInvincible { get; set; }
public float delayAfterDeath = 1.0f;
public AudioClip gnomeDiedSound;
public AudioClip gameOverSound;

void Start() {
Reset ();
}
public void Reset() {
if (gameOverMenu)
gameOverMenu.gameObject.SetActive(false);
if (mainMenu)
mainMenu.gameObject.SetActive(false);
if (gameplayMenu)
gameplayMenu.gameObject.SetActive(true);
var resetObjects = FindObjectsOfType<Resettable>();
foreach (Resettable r in resetObjects) {
r.Reset();
}
CreateNewGnome();
Time.timeScale = 1.0f;
}
void CreateNewGnome() {
RemoveGnome();
GameObject newGnome =(GameObject)Instantiate(gnomePrefab,
startingPoint.transform.position, Quaternion.identity);
currentGnome = newGnome.GetComponent<Gnome>();
```

```
rope.connectedObject = currentGnome.ropeBody;
rope.ResetLength();
cameraFollow.target = currentGnome.cameraFollowTarget;
}
void RemoveGnome() {
if (gnomeInvincible) return;
cameraFollow.target = null;
if (currentGnome != null) {
currentGnome.holdingTreasure = false;
currentGnome.gameObject.tag = "Untagged";
foreach (Transform child in currentGnome.transform) {
child.gameObject.tag = "Untagged";
}
currentGnome = null;
}}

void KillGnome(Gnome.DamageType damageType) {
var audio = GetComponent<AudioSource>(); if (audio) {
audio.PlayOneShot(this.gnomeDiedSound);
}
currentGnome.ShowDamageEffect(damageType);
if (gnomeInvincible == false) {
currentGnome.DestroyGnome(damageType);
RemoveGnome();
StartCoroutine(ResetAfterDelay());
}}
IEnumerator ResetAfterDelay() {
yield return new WaitForSeconds(delayAfterDeath); Reset();
}
public void TrapTouched() { KillGnome(Gnome.DamageType.Slicing);
}
public void FireTrapTouched() { KillGnome(Gnome.DamageType.Burning);
}
public void TreasureCollected() {
currentGnome.holdingTreasure = true;
}
public void ExitReached() {
```

```
if (currentGnome != null && currentGnome.holdingTreasure == true) {
var audio = GetComponent<AudioSource>(); if (audio) {
audio.PlayOneShot(this.gameOverSound);
}
Time.timeScale = 0.0f;
if (gameOverMenu) {
gameOverMenu.gameObject.SetActive(true);
}
if (gameplayMenu) {
gameplayMenu.gameObject.SetActive(false);
}
}
}
public void SetPaused(bool paused) {
if (paused) { Time.timeScale = 0.0f;
mainMenu.gameObject.SetActive(true); gameplay-
Menu.gameObject.SetActive(false);
} else {
Time.timeScale = 1.0f; mainMenu.gameObject.SetActive(false);
gameplayMenu.gameObject.SetActive(true);
}}
public void RestartGame() {
Destroy(currentGnome.gameObject);
currentGnome = null;
Reset();
}}
```

Главная задача диспетчера игры Game Manager – создание новых гномиков и подключение других систем к правильным объектам. Когда появляется новый гномик, к его ноге нужно привязать веревку Rope и подсказать сценарию CameraFollow, за каким гномиком он должен следить. Кроме того, диспетчер игры отвечает за отображение меню и обработку нажатий на кнопки в меню. (Меню мы реализуем позже. Теперь, написав код, его можно подключить к сцене.

Реализация игрового процесса с ловушками и целями

Теперь, когда основы игрового процесса настроены, можно начинать добавлять в игру такие элементы, как ловушки и сокровище. С

этого момента дальнейшее развитие игры в основном будет касаться ее дизайна.

Простые ловушки

Большая часть этой игры заключается в обработке касаний разных объектов – ловушек, сокровища, точки выхода и т. д. Учитывая, насколько важно определение момента касания конкретных объектов, создадим общий сценарий, генерирующий событие Unity Event, когда любой объект с тегом «Player» касается их. Это событие будет затем по-разному настраиваться для различных объектов: ловушки будут сообщать диспетчеру игры Game Manager, что гномик получил повреждение, сокровище будет сообщать, что гномик подобрал сокровище, а точка выхода – что гномик достиг выхода.

Теперь создайте новый сценарий на C# в файле с именем SignalOnTouch.cs и добавьте в него следующий код:

```
using UnityEngine.Events;
 [RequireComponent (typeof(Collider2D))] public class SignalOnTouch : MonoBehaviour {
public UnityEvent onTouch;
из AudioSource.
public bool playAudioOnTouch = true;
void OnTriggerEnter2D(Collider2D collider) {
SendSignal (collider.gameObject);
}
void OnCollisionEnter2D(Collision2D collision) {
SendSignal (collision.gameObject);
}
void SendSignal(GameObject objectThatHit) {
if (objectThatHit.CompareTag("Player")) {
if (playAudioOnTouch) {
var audio = GetComponent<AudioSource>();
if (audio && audio.gameObject.activeInHierarchy) audio.Play();
}
onTouch.Invoke();
}}}
```

Теперь, имея класс SignalOnTouch, можно добавить первую ловушку.

Сокровище и выход

После успешного добавления ловушки, убивающей гномика, самое время добавить возможность выиграть в игре. Для этого добавим два новых элемента: сокровище и точку выхода.

Сокровище – это спрайт на дне колодца, который обнаруживает касание гномика и посылает сигнал диспетчеру игры Game Manager. Когда это происходит, диспетчер игры сообщает гномику, что тот ухватил сокровище, после чего спрайт с изображением пустой руки гномика заменяется спрайтом с изображением руки, удерживающей сокровище.

Точка выхода – это еще один спрайт, находящийся в верхней части колодца. Подобно сокровищу, он обнаруживает касание гномика и извещает об этом диспетчер игры. Если в этот момент гномик держит сокровище, игроку присуждается победа в игре.

Основную работу в этих двух объектах выполняет компонент SignalOnTouch – когда гномик достигает точки выхода, должен вызываться метод ExitReached диспетчера игры, а когда гномик касается сокровища, должен вызываться метод TreasureCollected.

Сокровище работает так по умолчанию объект Treasure отображает спрайт с изображением сокровища. Когда гномик касается его, вызывается метод TreasureCollected диспетчера игры Game Manager и на месте сокровища отображается другой спрайт, показывающий, что сокровище подобрано. Когда гномик погибает, объект Treasure возвращается в исходное состояние и вновь отображает спрайт с изображением сокровища.

Так как смена спрайтов в игре будет выполняться довольно часто – вы убедитесь в этом, когда мы займемся улучшением графики, – имеет смысл создать универсальный класс смены спрайтов и задействовать его в объекте сокровища.

Создайте новый сценарий на C# с именем SpriteSwapper.cs. Добавьте в него следующий код:

```
public class SpriteSwapper : MonoBehaviour {
public Sprite spriteToUse;
public SpriteRenderer spriteRenderer;
private Sprite originalSprite;
public void SwapSprite() {
if (spriteToUse != spriteRenderer.sprite) {
originalSprite =  spriteRenderer.sprite;
```

```
spriteRenderer.sprite = spriteToUse;
}}
public void ResetSprite() {
if (originalSprite != null) {
spriteRenderer.sprite = originalSprite;
}}}
```

Теперь можно создать и настроить объект Treasure.

Добавьте спрайт с изображением сокровища. Добавьте коллайдер для сокровища. Добавьте и настройте сценарий смены спрайта. Добавьте и настройте компонент отправки сигнала в ответ на касание Добавьте и настройте компонент Resettable.

Добавление фона

В настоящее время действие игры разворачивается на унылом синем фоне, заданном в Unity по умолчанию.

Слои

Перед добавлением изображений мы должны сначала определить их порядок в сцене. При создании двумерной игры важно, но иногда бывает непросто обеспечить правильный порядок отображения спрайтов, когда одни должны появляться поверх других. К счастью, в Unity имеется встроенное решение, упрощающее эту задачу: слои сортировки (sorting layers).

В игре всегда присутствует хотя бы один слой – с названием «Default». И все новые объекты помещаются в этот слой, если вы не укажете иной.

Создание фона

Закончив создание слоев, можно приступать к конструированию самого фона. Фон имеет три темы оформления, а именно коричневую, синюю и красную. Каждая тема состоит из нескольких спрайтов: задняя стена, боковая стена и затененная версия боковой стены.

Дно колодца

Осталось только добавить последнюю вещь: дно колодца.

Создайте объект-контейнер для спрайтов с изображением дна. Добавьте спрайт, изображающий дно колодца.

Настройка камеры

Чтобы вписать обновленный фон в игру, осталось только скорректировать настройки камеры. Нам потребуется внести следующие изменения: во-первых, камеру нужно настроить так, чтобы игрок мог видеть весь уровень на всю ширину, а во-вторых, необходимо изменить сценарий, ограничивающий перемещение камеры, чтобы учесть изменившиеся размеры уровня. Для настройки камеры вы- полните следующие действия.

Измените размер камеры.

Измените ограничения, влияющие на позиционирование камеры.

Звуки

Подключите звук Death By Static Object к новому компоненту Audio Source. Сни- мите флажки Loop (Бесконечно) и Play On Awake (Проигрывать автоматически).

Протестируйте игру.

3.2. Создание трехмерной игры

Метеоритный дождь – трехмерный космический симулятор. Игры этого вида были популярны в середине 1990-х, когда симуляторы, такие как Star Wars: X-Wing (1993) и Descent: Freespace (1998), дали игрокам возможность свободно перемещаться в открытом космосе, уничтожать врагов и взрывать пространство. Игры такого типа очень близки к симуляторам полетов, но так как никто не ожидает реалистичной реализации физики полета, разработчики игр могут позволить себе больше сосредоточиться на игровых аспектах.

Проектирование игры

Приступая к проектированию игры, мы должны определить несколько ключевых ограничений.

Сеанс игры не должен превышать пары минут.

Элементы управления должны быть очень простыми и включать только элементы управления «перемещением» и «стрельбой».

Игрок должен решать несколько краткосрочных задач, а не одну большую. То есть он должен вести бой с несколькими мелкими врагами.

Сцена

Начнем с подготовки сцены. Мы создадим новый проект Unity, а затем добавим космический корабль, способный летать в пределах сцены.

Корабль

Сам объект Ship, представляющий космический корабль, будет невидимым объектом, содержащим только сценарии; к нему будет подключено множество дочерних объектов, решающих определенные задачи отображения на экране.

Создайте объект Ship. Теперь добавим модель корабля. Добавьте в корабль компонент Box Collider. Добавьте сценарий ShipThrust.

Затем откройте файл ShipThrust.cs и добавьте в него следующий код:

```
public class ShipThrust : MonoBehaviour { public float speed = 5.0f;
void Update () {
var offset = Vector3.forward * Time.deltaTime * speed;
this.transform.Translate(offset);
}}
```

Протестируйте игру.

Космическая станция

Процесс создания космической станции, которой угрожают летящие астероиды, идет по тому же шаблону, что и процесс создания корабля: мы должны создать пустой игровой объект и подключить к нему модель. Однако космическая станция все же немного проще космического корабля, потому что она никуда не движется: она остается на месте и подвергается ударам астероидов. Выполните следующие действия для ее создания.

Создайте контейнер для космической станции. Добавьте модель в качестве дочернего объекта. Сбросьте местоположение объекта модели Station. Выберите объект модели и раскройте его, чтобы увидеть перечень дочерних объектов. Модель станции состоит из нескольких поверхностей (мешей); главная из них – с именем Station. Выберите ее.

Небо

В настоящее время в проекте по умолчанию используется скайбокс, предназначенный для игр, события в которых развиваются на поверхности планеты. Замена его на скайбокс, специально предназна-

ченный для изображения космического про странства, совершенно необходима для создания правильного восприятия игры.

Ввод и управление полетом

Ввод

В этой игре присутствуют два устройства ввода: виртуальный джойстик, позволяющий игроку управлять направлением полета корабля, и кнопка, сигнализирующая о моментах, когда пользователь ведет стрельбу из лазерных пушек.

Добавление джойстика

Начнем с создания джойстика. Джойстик состоит из двух видимых компонентов: большого квадрата «контактной панели» в левом нижнем углу холста и маленькой «площадки» в центре этого квадрата. Когда пользователь помещает палец в границы контактной панели, джойстик меняет свое местоположение так, чтобы площадка оказалась точно под пальцем и одновременно в центре панели. Когда пользователь будет двигать пальцем, вместе с ним будет двигаться площадка. Чтобы приступить к созданию системы ввода, выполните следующие действия.

Создайте контактную панель. Добавьте изображение в контактную панель. Создайте площадку для пальца.

Добавьте сценарий VirtualJoystick.

Откройте файл и добавьте в него следующий код:

```
using UnityEngine.EventSystems;
using UnityEngine.UI;
public class VirtualJoystick : MonoBehaviour, IBeginDragHandler, IDragHandler, IEndDragHandler {
public RectTransform thumb;
private Vector2 originalPosition;
private Vector2 originalThumbPosition;
public Vector2 delta;
void Start () {
originalPosition
= this.GetComponent<RectTransform>().localPosition; originalThumbPosition = thumb.localPosition;
thumb.gameObject.SetActive(false);
delta = Vector2.zero;
}
```

```
public void OnBeginDrag (PointerEventData eventData) {
Vector3 worldPoint = new Vector3(); RectTransformUtility.ScreenPointToWorldPointInRectangle(this.transform as RectTransform, eventData.position, eventData.enterEventCamera,out worldPoint);
this.GetComponent<RectTransform>().position
= worldPoint;
thumb.localPosition = originalThumbPosition;
}
public void OnDrag (PointerEventData eventData) {
Vector3 worldPoint = new Vector3(); RectTransformUtility.ScreenPointToWorldPointInRectangle(this.transform as RectTransform, eventData.position, eventData.enterEventCamera,out worldPoint);
thumb.position = worldPoint;
var size = GetComponent<RectTransform>().rect.size; delta = thumb.localPosition;
delta.x /= size.x / 2.0f; delta.y /= size.y / 2.0f;
delta.x = Mathf.Clamp(delta.x, -1.0f, 1.0f); delta.y = Mathf.Clamp(delta.y, -1.0f, 1.0f);
}
public void OnEndDrag (PointerEventData eventData) {
this.GetComponent<RectTransform>().localPosition
= originalPosition;
delta = Vector2.zero;
}
}
```

Чтобы завершить создание системы ввода, выполните следующие действия.

Настройте джойстик.

Протестируйте джойстик.

Диспетчер ввода

Теперь, закончив настройку джойстика, организуем передачу информации от джойстика космическому кораблю, чтобы тот мог изменять направление полета.

Мы могли бы напрямую связать корабль с джойстиком, но возникает одна проблема. В процессе игры корабль может быть уничтожен, и будет создан новый корабль. Чтобы такое стало возможным, корабль придется превратить в шаблон – в этом случае диспетчер иг-

ры сможет создать множество его копий. Однако шаблоны не могут ссылаться на объекты в сцене, то есть вновь созданный объект корабля не будет иметь ссылки на джойстик.

Лучшее решение – создать объект-одиночку (singleton) диспетчера ввода, всегда присутствующий в сцене и всегда имеющий доступ к джойстику. Так как этот объект не является экземпляром шаблона, нам не придется беспокоиться о потере ссылки при его создании. Когда будет создаваться новый корабль, он сможет использовать диспетчер ввода (доступный программно) для получения информации из джойстика.

Создайте сценарий Singleton. Создайте новый сценарий на C# с именем

Singleton.cs в папке Assets. Откройте этот файл и введите следующий код:

```
// Этот класс позволяет другим объектам ссылаться на единственный
// общий объект. Его используют классы GameManager и InputManager.

// Чтобы воспользоваться этим классом, унаследуйте его:
// public class MyManager : Singleton<MyManager> { }

// После этого вы сможете обращаться к единственному общему
// экземпляру класса так:
// MyManager.instance.DoSomething();

public class Singleton<T> : MonoBehaviour where T : MonoBehaviour {
// Единственный экземпляр этого класса.
private static T _instance;

// Метод доступа. При первом вызове настраивает _instance.
// Если требуемый объект не найден,
// выводит сообщение об ошибке в журнал.
public static T instance { get {
// Если свойство _instance еще не настроено..
if (_instance == null)

{
```

```
// ...попытаться найти объект.
_instance = FindObjectOfType<T>();

// Записать сообщение об ошибке в случае неудачи.
if (_instance == null) {
Debug.LogError("Can't find "
+ typeof(T) + "!");
}
}

// Вернуть экземпляр для использования!
return _instance;
}
}
}
```

Создайте диспетчер ввода. Создайте новый пустой игровой объект с именем Input Manager.

```
public class InputManager : Singleton<InputManager> {

// Джойстик, используемый для управления кораблем.
public VirtualJoystick steering;

}
```

Настройте диспетчер ввода.

Управление полетом

В данный момент корабль может двигаться только вперед. Чтобы изменить направление движения корабля, мы должны изменить его направление «вперед». Для этого мы будем извлекать информацию из виртуального джойстика и в соответствии с ней изменять ориентацию корабля в пространстве.

В каждом кадре корабль использует направление, определяемое джойстиком, и значение, задающее скорость поворота корабля, чтобы получить новую величину поворота. Затем эта информация объединяется с текущей ориентацией корабля, и в результате получается его новое направление «вперед».

Добавьте сценарий ShipSteering. Выберите объект Ship и добавьте новый сценарий на C# с именем ShipSteering.cs. Откройте файл и добавьте следующий код:

```
public class ShipSteering : MonoBehaviour {
public float turnRate = 6.0f;
public float levelDamping = 1.0f; void Update () {
var steeringInput = InputManager.instance.steering.delta;
var rotation = new Vector2();
rotation.y = steeringInput.x; rotation.x = steeringInput.y;
rotation *= turnRate;
rotation.x = Mathf.Clamp(
rotation.x, -Mathf.PI * 0.9f, Mathf.PI * 0.9f);
var newOrientation = Quaternion.Euler(rotation);
transform.rotation *= newOrientation;
var levelAngles = transform.eulerAngles; levelAngles.z = 0.0f;
var levelOrientation = Quaternion.Euler(levelAngles);
transform.rotation = Quaternion.Slerp( transform.rotation, levelOrientation, levelDamping * Time.deltaTime);
}}
```

Протестируйте управление.

Создание элементов пользовательского интерфейса

Для начала создадим объект внутри холста, который будет действовать как контейнер для всех индикаторов. Затем мы создадим индикатор и превратим его в шаблон для многократного использования.

Добавьте код. Добавьте в объект прототипа Indicator новый сценарий на C# с именем Indicator.cs и введите в него следующий код:

```
using UnityEngine.UI;
public class Indicator : MonoBehaviour {
public  Transform target;
public Transform showDistanceTo;
public Text distanceLabel;
public int margin = 50;
public Color color { set {
GetComponent<Image>().color = value;
}
get {
return GetComponent<Image>().color;
}}
```

```
void Start() {
distanceLabel.enabled = false;
GetComponent<Image>().enabled = false;
}
void Update()
{
if (target == null) { Destroy (gameObject); return;
}
if (showDistanceTo != null) {
distanceLabel.enabled = true;
var distance = (int)Vector3.Magnitude(
showDistanceTo.position - target.position);
distanceLabel.text = distance.ToString() + "m";
} else {
distanceLabel.enabled = false;
}
GetComponent<Image>().enabled = true;
var viewportPoint =
Camera.main.WorldToViewportPoint(target.position);
if (viewportPoint.z < 0) {
viewportPoint.z = 0;
viewportPoint = viewportPoint.normalized;
viewportPoint.x *= -Mathf.Infinity;
}
var screenPoint = Camera.main.ViewportToScreenPoint(viewportPoint);
screenPoint.x = Mathf.Clamp( screenPoint.x,margin,Screen.width - margin * 2);
screenPoint.y = Mathf.Clamp( screenPoint.y,margin,Screen.height - margin * 2);
var localPosition = new Vector2(); RectTransformUtility.ScreenPointToLocalPointInRectangle(transform.parent.GetComponent<RectTransform>(), screenPoint,Camera.main,out localPosition);
var rectTransform = GetComponent<RectTransform>(); rectTransform.localPosition = localPosition;
}}
```

В настройке индикаторов осталось выполнить один заключительный шаг. Завершив создание прототипа индикатора, его нужно преобразовать в шаблон.

Подключите надпись, отображающую расстояние. Перетащите дочерний объект Text в поле Distance Label.

Преобразуйте прототип в шаблонный объект (prefab). Перетащите объект Position Indicator в панель обозревателя проекта. В результате будет создан новый шаблонный объект (prefab), используя который можно создать множество индикаторов во время выполнения.

Добавление оружия и прицеливания

Теперь, реализовав управление космическим кораблем, добавим в общую картину еще один игровой элемент. Сначала мы добавим оружие, а после этого реализуем механизм прицеливания.

Оружие

Каждый раз, когда игрок приводит в действие оружие, лазерные пушки космического корабля выстреливают шарами плазмы, которые летят вперед, пока не поразят какую-либо цель или не истечет время их существования. Если плазменный шар попадает в какой-то объект и этот объект может получить повреждение, информация о попадании должна передаваться такому объекту.

Реализовать это можно, создав объект с коллайдером, который движется вперед с определенной скоростью (подобно космическому кораблю). Есть несколько вариантов отображения снаряда – можно создать трехмерную модель ракеты, использовать эффект частиц или спрайт.

Для отображения снаряда мы используем визуализатор Trail Renderer. Этот визуализатор создает постепенно исчезающий светящийся след, оставляемый движущимся объектом. Этот эффект особенно хорош для представления таких движущихся объектов, как качающиеся клинки или летящие снаряды.

Создайте снаряд. Добавьте сценарий Shot. Добавьте в объект новый сценарий на C# с именем Shot. Откройте файл Shot.cs и добавьте в него следующий код:

```
public class Shot : MonoBehaviour {
public  float  speed  = 100.0f;
public float life = 5.0f;
void Start() {
```

```
Destroy(gameObject, life);
}
void Update () {
transform.Translate(
Vector3.forward * speed * Time.deltaTime);
}}
```

Оружие корабля

Чтобы игрок смог открыть огонь из лазерных пушек корабля, мы должны предусмотреть какой-то механизм создания объектов Shot. Механика стрельбы из лазерных пушек немного сложнее, чем простое выбрасывание снаряда, когда игрок нажимает кнопку Fire (Огонь); нам нужно, чтобы снаряды вылетали с постоянной частотой, когда игрок нажимает и удерживает кнопку.

В таких случаях необходимо также решить, как будут выстреливаться снаряды. Можно выстреливать два снаряда сразу с обеих сторон или попеременно – то слева, то справа.

Добавьте сценарий ShipWeapons в объект Ship. Выберите объект Ship, добавьте новый сценарий на C# с именем ShipWeapons.cs и введите в него следующий код:

```
public class ShipWeapons : MonoBehaviour {

public GameObject shotPrefab;
public Transform[] firePoints;
private int firePointIndex;
public void Fire() {
if (firePoints.Length == 0) return;
var firePointToUse = firePoints[firePointIndex];
Instantiate(shotPrefab, firePointToUse.position, firePoint-
ToUse.rotation);
firePointIndex++;
if (firePointIndex >= firePoints.Length) firePointIndex = 0;
}}
```

Кнопка «Fire»

Теперь добавим кнопку, коснувшись которой пользователь может открыть огонь из оружия корабля и прекратить его, отпустив палец.

Вот как будет работать диспетчер ввода: так как в каждый конкретный момент времени в игре будет только один корабль, то будет присутствовать только один экземпляр ShipWeapons. В момент появления сценарий ShipWeapons будет получать ссылку на объект-одиночку диспетчера ввода и сообщать ему, что является текущим сценарием ShipWeapons. Диспетчер ввода запомнит это и будет использовать сценарий как часть системы стрельбы.

Наконец, кнопка Fire будет подключаться к объекту диспетчера ввода и посылать ему сигнал «огонь открыт» в момент касания ее пользователем и сигнал «огонь прекращен», когда пользователь уберет палец с кнопки. Диспетчер ввода будет передавать эти сообщения текущему сценарию ShipWeapons, благодаря чему тот будет вести огонь.

Сначала реализуем поддержку регистрации текущего экземпляра ShipWeapons в классе InputManager; затем добавим код в ShipWeapons, который регистрирует сценарий как текущий в момент появления и аннулирует регистрацию, когда компонент удаляется (когда корабль разрушается).

Добавьте в сценарий диспетчера ввода управление сценарием ShipWeapons, включив следующие свойства и методы в класс InputManager:

```
public class InputManager : Singleton<InputManager> {
public VirtualJoystick steering;
public float fireRate = 0.2f;
private ShipWeapons currentWeapons;
private bool isFiring = false;
public void SetWeapons(ShipWeapons weapons) {
this.currentWeapons = weapons;
}
public void RemoveWeapons(ShipWeapons weapons) {
if (this.currentWeapons == weapons) {
this.currentWeapons = null;
}}
public void StartFiring() {
StartCoroutine(FireWeapons());
}
IEnumerator FireWeapons() {
isFiring = true;
while (isFiring) {
```

```
if (this.currentWeapons != null) {
currentWeapons.Fire();
}
yield return new WaitForSeconds(fireRate);
}}
public void StopFiring() {
isFiring = false;
}}
```

Далее нам нужно реализовать в ShipWeapons взаимодействие с диспетчером ввода, добавив следующие методы в класс ShipWeapons:

```
public class ShipWeapons : MonoBehaviour {
public GameObject shotPrefab;
public  void  Awake() {
InputManager.instance.SetWeapons(this);
}
public void OnDestroy() {
if (Application.isPlaying == true) {
InputManager.instance
.RemoveWeapons(this);
}}
public Transform[] firePoints;
private int firePointIndex;
public void Fire() {
if (firePoints.Length == 0) return;
var firePointToUse = firePoints[firePointIndex];
Instantiate(shotPrefab, firePointToUse.position, firePoint-
ToUse.rotation);
firePointIndex++;
if (firePointIndex >= firePoints.Length) firePointIndex = 0;
}}
```

Теперь создадим кнопку Fire, предписывающую диспетчеру ввода открыть или прекратить огонь. Так как нам требуется извещать диспетчер ввода о начале и конце удержания кнопки, мы не можем использовать стандартное поведение, когда кнопка посылает сообщение только после «щелчка». Вместо этого для отправки отдельных сообщений Pointer Down и Pointer Up нам понадобится использовать компоненты Event Trigger.

Сначала создайте саму кнопку. Дайте новой кнопке имя Fire Button. Установите точки привязки и опорную точку кнопки справа внизу.

Астероиды и повреждения

Астероиды

Сейчас у нас есть корабль, летящий в космосе, индикаторы на экране, а также мы можем прицеливаться и стрелять из лазерных пушек. Единственное, чего у нас нет, – это объектов для стрельбы. Пришло время исправить это упущение. Мы создадим астероиды, не очень большие, но способные летать. Также мы добавим систему, создающую эти астероиды и направляющую их к космической станции.

Сначала создадим прототип астероида. Каждый астероид будет состоять из двух объектов: объекта высокоуровневой абстракции, содержащего коллайдер и всю необходимую логику, и дополнительного «графического» объекта, отвечающего за визуальное представление астероида на экране.

Создайте объект. Добавьте в него модель астероида. Добавьте в объект Asteroid твердое тело и сферический коллайдер. Затем снимите флажок Use Gravity в компоненте Rigidbody и установите радиус сферического коллайдера равным 2.

Добавьте сценарий Asteroid. Добавьте в игровой объект Asteroid новый сцена- рий на C#, дайте ему имя Asteroid.cs и введите в него следующий код:

```
public class Asteroid : MonoBehaviour {
public float speed = 10.0f;
void Start () {
GetComponent<Rigidbody>().velocity= transform.forward * speed;
var indicator = IndicatorManager.instance.AddIndicator(gameObject,
Color.red);
}}
```

Создание астероидов

Теперь, получив действующий астероид, мы можем перейти к реализации системы создания астероидов. Это объект, периодически создающий новые объекты астероидов и запускающий их к цели. Астероиды будут создаваться в случайных точках на поверхности невидимой сферы и настраиваться так, что их направление «вперед» будет нацелено на объект в игре. Кроме того, система создания астероидов

будет использовать одну из возможностей Unity под названием Gizmos, позволяющую отображать дополнительную информацию в представлении сцены для визуализации объема космического пространства, в котором появляются астероиды.

Преобразуйте астероид в шаблон. Создайте объект Asteroid Spawner. Далее добавьте новый сценарий на C# с именем AsteroidSpawner.cs и введите в него следующий код:

```
public class AsteroidSpawner : MonoBehaviour {
public float radius = 250.0f;
public Rigidbody asteroidPrefab;
public float spawnRate = 5.0f; public float variance = 1.0f;
public Transform target;
public bool spawnAsteroids = false;
void Start () {
StartCoroutine(CreateAsteroids());
}
IEnumerator CreateAsteroids() {
while (true) {
float nextSpawnTime= spawnRate + Random.Range(-variance, variance);
yield return new WaitForSeconds(nextSpawnTime);
yield return new WaitForFixedUpdate();
CreateNewAsteroid();
}}
void CreateNewAsteroid() {
if (spawnAsteroids == false) { return;
}
var asteroidPosition = Random.onUnitSphere * radius;
asteroidPosition.Scale(transform.lossyScale);
asteroidPosition += transform.position;
var newAsteroid = Instantiate(asteroidPrefab);
newAsteroid.transform.position = asteroidPosition;
newAsteroid.transform.LookAt(target);
}
void OnDrawGizmosSelected() {
Gizmos.color = Color.yellow;
Gizmos.matrix = transform.localToWorldMatrix;
Gizmos.DrawWireSphere(Vector3.zero, radius);
}
```

```
public void DestroyAllAsteroids() {
foreach (var asteroid in FindObjectsOfType<Asteroid>()) { Destroy
(asteroid.gameObject);
}}}
```

Нанесение и получение повреждений

Теперь корабль может летать вокруг космической станции и между астероидами, летящими к ней, но выстреливаемые игроком шары плазмы фактически не причиняют никакого вреда. Нам нужно добавить возможность нанесения и получения повреждений.

Какие-то объекты будут способны получать повреждения, какие-то – наносить повреждения. А некоторые, например, астероиды, будут способны на то и на другое – они могут получать повреждения от попадания шаров плазмы, а также могут наносить повреждения объектам, сталкивающимся с ними, таким как космическая станция.

Чтобы реализовать это, мы создадим два отдельных сценария: DamageTaking и DamageOnCollide.

Сценарий DamageTaking контролирует очки целостности объекта, к которому он подключен, и удаляет объект из игры, когда это число достигает нуля. Сценарий DamageTaking также экспортирует метод TakeDamage, который будет вызываться другими объектами для нанесения повреждений.

Сценарий DamageOnCollide выполняется, когда объект, к которому он подклю- чен, сталкивается с любым другим объектом или входит в область действия триггера. Если объект, с которым сталкивается данный объект, имеет компонент DamageTaking, сценарий DamageOnCollide вызывает его метод TakeDamage.

Сценарий DamageOnCollide будет добавляться в объекты Shot и Asteroid, а сценарий

DamageTaking – в объекты Space Station и Asteroid.

Сначала реализуем возможность нанесения повреждений астероидам.

Добавьте в астероиды сценарий DamageTaking.

```
public class DamageTaking : MonoBehaviour {
public int hitPoints = 10;
public GameObject destructionPrefab;
public bool gameOverOnDestroyed = false;
public void TakeDamage(int amount) {
```

```
Debug.Log(gameObject.name + " damaged!");
if (hitPoints <= 0) {
Debug.Log(gameObject.name + " destroyed!");
Destroy(gameObject);
if (destructionPrefab != null) {
Instantiate(destructionPrefab, transform.position, transform.rotation);
}}}}
```

Звуки, меню, разрушения и взрывы!

Меню

На данный момент в игру можно играть, только если запускать ее из редактора Unity, щелкнув по кнопке Play (Играть). Когда игра запускается, тут же начинается игровое действие, и если космическая станция окажется разрушена, игру придется остановить и запустить снова.

Чтобы дать игроку возможность управлять игрой, нужно добавить меню. В частности, мы должны добавить самую главную кнопку: New Game (Новая игра). Мы должны дать игроку возможность начать игру сначала, когда космическая станция окажется разрушенной.

Добавление структуры меню имеет большое значение для создания ощущения законченности игры. Далее мы добавим в меню четыре компонента.

Элементы управления игрой

На этом экране будет отображаться джойстик, индикаторы, кнопка Fire (Огонь) и все остальное, что игрок видит в процессе игры.

Первым шагом на этом пути является объединение элементов управления игрой в один общий объект для управления им как единым целым.

Главное меню

Главное меню имеет очень простую организацию – оно включает текстовую надпись с названием игры («Rockfall») и кнопку для запуска новой игры.

По аналогии с элементами управления игрой, заключим все компоненты главного меню в общий объект-контейнер в виде дочерних объектов.

Создайте контейнер Main Menu.

Создайте объект надписи с названием игры. Создайте кнопку.

Экран паузы

Экран паузы отображает надпись «Paused» (Приостановлено) и кнопку для возоб новления игры. Чтобы создать его, выполните те же действия, как при создании главного меню, но со следующими изменениями:объекту-контейнеру присвойте имя Paused; в объект Title введите текст «Paused»; объекту кнопки присвойте имя Unpause ; надпись на кнопке должна содержать текст «Unpause».

Экран завершения игры

Экран завершения игры отображает текст «Game Over» (Конец игры) и содержит кнопку, запускающую новую игру. Он будет появляться после разрушения космической станции, означающего конец игры.

И снова выполните те же действия, как при создании главного меню и экрана паузы, но со следующими изменениями: объекту-контейнеру присвойте имя Game Over; в объект Title введите текст «Game Over»; объекту кнопки присвойте имя New Game; надпись на кнопке должна содержать текст «New Game».

3.3. Дополнительные возможности создания игр

Внешний вид объектов в Unity определяется подключенным к нему материалом. Материал состоит из двух компонентов: шейдера (shader) и данных, которые ис пользуются шейдером.

В Unity поддерживаются два основных типа шейдеров: поверхностные шейдеры (surface shaders) и фрагмент-вершинные шейдеры (fragment-vertex shaders).

Если у вас нет каких-то особых требований, предпочтительнее использовать поверхностные шейдеры.

Чтобы создать такой эффект, выполните следующие действия.

Создайте новый проект. Создайте новый шейдер. Дважды щелкните по нему. Замените его содержимое следующим кодом:

```
Shader "Custom/SimpleSurfaceShader" { Properties {
_Color ("Color", Color) = (0.5,0.5,0.5,1)
_MainTex ("Albedo (RGB)", 2D) = "white" {}
_Smoothness ("Smoothness", Range(0,1)) = 0.5
_Metallic ("Metallic", Range(0,1)) = 0.0
}
```

```
SubShader {
Tags { "RenderType"="Opaque" } LOD 200
// Цвет оттенка отраженного света
fixed4 _Color;
half _Smoothness; half _Metallic;
struct Input {
float2 uv_MainTex;
};
void surf (Input IN, inout SurfaceOutputStandard o) {
fixed4 c =
tex2D (_MainTex, IN.uv_MainTex) * _Color; o.Albedo = c.rgb;
o.Metallic = _Metallic; o.Smoothness = _Smoothness;
}
}
o.Alpha = c.a;
FallBack "Diffuse"
}
```

Создайте новый материал с именем SimpleSurface.

Выберите новый материал и откройте меню Shader в верхней части инспектора. Создайте новую капсулу. Перетащите материал SimpleShader Сейчас объект выглядит почти так же, как при отображении с использованием стандартного шейдера. Давайте теперь добавим освещение сзади!

Первые две характеристики являются униформными (uniform), то есть их значения применяются ко всем пикселам объекта. Третья характеристика имеет переменный (varying) характер, то есть ее значение зависит от направления взгляда; угол между направлением визирования камеры и нормалью к поверхности зависит от того, смотрите вы в середину цилиндра или на его край.

Измените раздел Properties шейдера, добавив следующий код:

```
Properties {
_Color ("Color", Color) = (0.5,0.5,0.5,1)
_MainTex ("Albedo (RGB)", 2D) = "white" {}
_Smoothness ("Smoothness", Range(0,1)) = 0.5
_Metallic ("Metallic", Range(0,1)) = 0.0
_RimColor ("Rim Color", Color) = (1.0,1.0,1.0,0.0)
_RimPower ("Rim Power", Range(0.5,8.0)) = 2.0
}
```

Добавьте в шейдер следующий код:

```
half _Smoothness; half _Metallic;
float4 _RimColor;
float _RimPower;
```

Далее нужно дать шейдеру возможность определять направление визирования камеры. Все переменные значения, используемые шейдером, включены в структуру Input, следовательно, направление визирования камеры нужно добавить туда.

Добавьте следующий код в определение структуры Input:

```
struct Input {
float2 uv_MainTex;
float3 viewDir;
};
```

Теперь у нас есть вся информация, необходимая для вычисления параметров контура, создаваемого подсветкой сзади; осталось сделать последний шаг – реализовать фактические вычисления и добавить их результаты в информацию о поверхности.

Добавьте следующий код в функцию surf:

```
void surf (Input IN, inout SurfaceOutputStandard o) {
fixed4 c = tex2D (_MainTex, IN.uv_MainTex) * _Color; o.Albedo = c.rgb;
o.Metallic = _Metallic; o.Smoothness = _Smoothness;
o.Alpha = c.a;
half rim =
1.0 - saturate(dot (normalize(IN.viewDir), o.Normal));
o.Emission = _RimColor.rgb * pow (rim, _RimPower);
}
```

Сохраните шейдер и вернитесь в Unity.

Фрагмент-вершинные шейдеры (без освещения)

Для демонстрации особенностей работы фрагмент-вершинных шейдеров создадим простой шейдер, отображающий равномерно окрашенные одноцветные объекты. Затем мы преобразуем его, чтобы он отображал градиентную заливку, в зависимости от местоположения объекта на экране.

Создайте новый шейдер. Выполните двойной щелчок по нему, чтобы открыть. Замените содержимое файла следующим кодом:

```
Shader "Custom/SimpleUnlitShader"
{
Properties
{
_Color ("Color", Color) = (1.0,1.0,1.0,1)
}
SubShader
{
Tags { "RenderType"="Opaque" } LOD 100
Pass
{
CGPROGRAM
#pragma vertex vert
#pragma fragment frag
#include "UnityCG.cginc" float4 _Color;
struct appdata
{
float4 vertex : POSITION;
};
struct v2f
{
float4 vertex : SV_POSITION;
};
v2f vert (appdata v)
{
v2f o;
o.vertex = UnityObjectToClipPos(v.vertex);
return o;
}
fixed4 frag (v2f i) : SV_Target
{
fixed4 col;
col = _Color;
return col;
}
ENDCG
}}}
```

Создайте новый материал. Выберите новый материал и измените его шейдер. Создайте в сцене сферу,

Далее попробуем использовать этот шейдер для воспроизведения анимационного эффекта. Для этого не придется писать сценарий; все необходимое мы реализуем внутри шейдера.

Добавьте следующий код в функцию frag:

```
fixed4 frag (v2f i) : SV_Target
{
fixed4 col;
col = _Color;
col *= abs(_SinTime[3]);
return col;
}
```

Вернитесь в Unity

Как видите, фрагмент-вершинные шейдеры позволяют получить полный контроль над внешним видом объектов.

Глобальное освещение

Когда объект освещается, шейдер, отвечающий за его отображение, должен выполнить ряд сложных вычислений, определить количество света, получаемого объектом, и на основе этой величины рассчитать цвет, который видит камера. Это нормально, но некоторые величины очень сложно рассчитать во время выполнения.

Например, если сфера покоится на белой поверхности и освещается прямым солнечным светом, она должна подсвечиваться снизу светом, отраженным от белойповерхности. Однако шейдеру известно только направление на солнце, и, как результат, он не отображает эту подсветку. Конечно, такую подсветку можно рассчитать, но это сразу превращается в очень сложную задачу, чтобы решать ее в каждом кадре.

Лучшее решение заключается в использовании глобального освещения (global illumination) и карт освещенности (lightmapping).

Применение методов глобального освещения позволяет получить очень правдоподобную картину, но требует значительных вычислительных ресурсов. По этой причине вычисления освещенности часто выполняют заблаговременно в редакторе Unity. Результаты также могут быть сохранены в так называемой карте освещенности

(lightmap), которая определяет количество света, получаемое каждым фрагментом каждой поверхности в сцене.

Использование карт освещенности может значительно увеличить производительность создания реалистичного освещения в сцене, так как вычисления выполняются заранее, а их результаты сохраняются в текстурах. Однако чтобы использовать эти текстуры для отображения, их нужно загружать в память. Это может стать проблемой для сложных сцен, использующих много сложных текстур. Сгладить эту проблему можно за счет уменьшения разрешения карт освещенности, но при этом снизится визуальное качество освещения.

Создайте новую сцену в Unity.

Как только объекты в сцене будут объявлены статическими, Unity немедленно приступит к вычислению информации об их освещенности. Через несколько мгновений картина освещенности немного изменится. Наиболее заметным изменением станет появление на белом кубе светового пятна, отбрасываемого зеленым кубом.

Зонды освещения

Добавим нестатический объект в сцену, а затем добавим несколько зондов освеще- ния, чтобы увидеть их влияние на освещенность.

Добавьте в сцену новую капсулу. Добавьте несколько зондов освещения, Расположите зонды

После этого капсула должна получить дополнительный зеленый свет, отраженный зеленым кубом

Размышления о производительности

Настройки освещенности, используемые в игре, могут существенно влиять на производительность пользовательского устройства; то есть наряду с глобальным освещением и картами появляются и другие проблемы с производительностью, вызванные объявлением объектов статическими.

Однако производительность игр не всегда зависит от графики: время, на которое сценарии занимают центральный процессор, тоже имеет большое значение.

Профайлер

Профайлер – это инструмент, записывающий данные в процессе выполнения игры.

Окно профайлера разбито пополам. Верхняя половина делится на несколько горизонтальных панелей – по одной для каждого регистратора данных из перечисленных выше.

Запустите профайлер, открыв меню Window (Окно) и выбрав пункт Profiler (Профайлер).

Запустите игру.

Спустя несколько мгновений остановите или приостановите игру.

Для разных регистраторов отображается разная информация. Если выбран регистратор CPU Usage (Использование процессора), для него внизу по умолчанию отображается иерархия вызовов – список всех методов, вызывавшихся в кадре и отсортированных в порядке уменьшения времени, затраченного на вызов. В списке можно щелкать по треугольникам слева, чтобы раскрыть ту или иную строку и увидеть информацию о вызовах методов, соответствующих этой строке.

Получение данных с устройства

Однако в редакторе игра имеет иную производительность, чем на устройстве. Персональные компьютеры обычно оснащены намного более быстрыми процессорами и обладают большим объемом оперативной памяти, чем мобильные устройства. Соответственно, результаты, полученные в профайлере, будут отличаться, и оптимизация узких мест, наблюдаемых при выполнении игры под управлением редактора, может не улучшить производительность на устройстве конечного пользователя.

Чтобы решить эту проблему, профайлер можно настроить на сбор данных при выполнении игры на устройстве. Для этого выполните следующие действия.

Соберите и установите игру на свое устройство

Убедитесь, что ваши устройство и компьютер находятся в одной сети Wi-Fi. Запустите игру на устройстве. Запустите профайлер и откройте меню Active Profiler (Активный профайлер).

Создание графических интерфейсов пользователя в Unity

По сути, графический пользовательский интерфейс в Unity мало чем отличается от других видимых объектов в сцене. Графический интерфейс – это меш (mesh), конструируемый движком в процессе выполнения, с наложенной на него текстурой; кроме того, элементы графического интерфейса содержат сценарии, вызываемые в ответ на перемещение указателя мыши, события от клавиатуры и касания, для

обновления и изменения этого меша. Меши отображаются с помощью камеры.

Система графического пользовательского интерфейса в Unity состоит из нескольких элементов, взаимодействующих друг с другом. Основу графического пользовательского интерфейса составляют несколько объектов с компонентами RectTransform, которые рисуют свое содержимое и откликаются на события. Все они находятся внутри объекта Canvas.

Canvas

Canvas – это объект, отвечающий за отображение всех элементов пользовательского интерфейса на экране. Он также служит единым пространством, в котором отображается холст.

Объект Canvas может использоваться в одном из трех режимов – Screen Space - Overlay (Пространство экрана – Перекрытие), Screen Space - Camera (Пространство экрана – Камера) и World Space (Пространство игрового мира).

RectTransform

Unity – это трехмерный игровой движок, все объекты которого обладают компонентом Transform, определяющим их координаты, поворот и масштаб в трехмерном пространстве. Однако пользовательский интерфейс в Unity имеет двумерную природу. То есть все элементы пользовательского интерфейса являются двумерными прямоугольниками, характеризующимися местоположением, шириной и высотой.

Управление элементами пользовательского интерфейса осуществляется посредством компонента RectTransform. RectTransform представляет собой прямоугольник, в котором отображается элемент пользовательского интерфейса.

Элементы управления

Для использования в сценах доступно несколько элементов управления. От простых и распространенных, как, например, кнопки и поля ввода, до сложных, таких как прокручиваемые представления.События и метод выпускания лучей

Когда пользователь касается кнопки на экране, он ожидает, что та выполнит некоторое действие, назначенное ей. Для этого система

пользовательского интерфейса должна иметь возможность узнать, какой объект был использован.

Система, поддерживающая такую возможность, называется системой событий. Действие системы событий основано на методе выпускания лучей (raycasts).

Метод выпускания лучей предназначен для использования в трехмерном пространстве, поэтому так же, как и вся остальная часть движка, система событий способна работать и с двумерными, и с трехмерными интерфейсами. Когда происходит событие, такое как касание пальцем или щелчок мышью, каждый источник лучей (raycaster) в сцене выпускает свой луч. Существует три типа источников, испускающих лучи: графические источники лучей (graphic raycasters), двумерные физические источники (2D physics raycasters) и трехмерные физические источни- ки лучей (3D physics raycasters).

Обработка событий

При создании своего пользовательского интерфейса часто бывает удобно иметь возможность добавлять нестандартное поведение в элементы интерфейса. Обычно под этим понимается обработка событий ввода, таких как щелчки и перетаскивания.

Чтобы сценарий мог откликаться на эти события, класс должен унаследовать определенные интерфейсы и реализовать их методы. Например, класс, наследующий интерфейс IPointerClickHandler, обязан реализовать метод с сигнатурой:

public void OnPointerClick (PointerEventData event Data). Этот метод вызывается, когда система событий обнаруживает, что текущий указатель (указатель мыши или палец, коснувшийся экрана) выполнил «щелчок», то есть в пределах изображения была нажата и отпущена кнопка мыши или опущен и поднят палец.

Для демонстрации ниже приводится краткое руководство, как обработать щелчок по объекту пользовательского интерфейса.

В пустой сцене создайте новый объект Canvas. Создайте новый объект Image. Добавьте в объект Image новый сценарий на C# с именем EventResponder.cs и введите в файл следующий код:

```
using UnityEngine.EventSystems;
public class EventResponder : MonoBehaviour, IPointerClickHandler {
public void OnPointerClick (PointerEventData eventData)
```

```
{
Debug.Log("Clicked!");
}}
```

Запустите игру. После щелчка по изображению в консоли должно появиться слово «Clicked!».

Переходы между экранами

Большинство пользовательских интерфейсов в играх делится на два типа: меню и интерфейс в игре. Меню – это интерфейс, с которым взаимодействует игрок, готовясь начать игру, – то есть выбирает между началом новой игры или продолжением предыдущей, настраивает параметры или просматривает список участников перед присоединением к многопользовательской игре. Интерфейс в игре накладывается сверху на игровой мир.

Интерфейс в игре обычно редко изменяет свою структуру и часто служит для отображения важной информации: количество стрел в колчане игрока, уровень здоровья и расстояние до следующей цели. Меню, напротив, меняются очень часто; главное меню обычно сильно отличается от экрана с настройками, потому что к ним предъявляются разные требования.

Поскольку пользовательский интерфейс – это всего лишь объект, отображаемый камерой, в Unity в действительности отсутствует понятие «экрана» как совокупности отображаемых элементов. Есть лишь текущая коллекция объектов на холсте. Если вам понадобится сменить один экран на другой, вы должны будете или изменить содержимое холста, который отображает камера, или направить камеру на что-то другое.

Прием изменения содержимого холста хорошо подходит для случаев, когда требуется изменить ограниченное подмножество элементов интерфейса. Например, если желательно оставить декоративное оформление, но заменить некоторые элементы пользовательского интерфейса, тогда имеет смысл не трогать камеру, а вы- полнить необходимые изменения в холсте. Но если требуется произвести полную замену элементов интерфейса, изменение направления визирования камеры может оказаться более эффективным решением.

При этом важно помнить, что для независимого изменения положения камеры в режиме работы холста требуется выбрать режим World Space (Пространство игрового мира); в двух других режимах –

Screen Space - Overlay (Пространство экрана – Перекрытие), Screen Space - Camera (Пространство экрана – Камера) – пользовательский интерфейс всегда находится прямо перед камерой.

Расширения редактора

При создании игр в Unity приходится работать с большим количеством игровых объектов и управлять всеми их компонентами. Инспектор уже избавляет от множества хлопот, автоматически отображая все переменные в сценариях в виде простых в использовании текстовых полей ввода, флажков и слотов для сброса ресурсов и объектов из сцены, помогая намного быстрее конструировать сцены.

Но иногда возможностей инспектора оказывается недостаточно. Unity разрабатывался так, чтобы максимально упростить создание двух- и трехмерных окружений, но разработчики Unity не в состоянии предвидеть все, что может вам потребоваться в играх.

Пользовательские расширения позволят вам получить контроль над самим редактором. Это могут быть очень маленькие вспомогательные окна, автоматизирующие типичные задачи в редакторе, или даже полностью переделанный инспектор.

Существует несколько способов расширения Unity. Чтобы приступить к работе с примерами в этой главе, создадим новый проект.

Создайте новый проект с названием «Editor Extensions».

Когда Unity загрузится, создайте новую папку внутри папки Assets.

Создание своего мастера

Для начала создадим свой мастер. Мастер – это самый простой способ показать окно, в котором можно получить ввод пользователя и на его основе создать что-нибудь в сцене. Типичным примером может служить создание объекта, зависящего от параметров, введенных пользователем.

Лучший способ понять, чем мастеры могут помочь в повседневной практике использования Unity, – создать один такой.

Чтобы получить подобный объект, требуется вручную создать объект Mesh. Обычно такие объекты импортируются из файлов, например, из файла .blend, однако их можно также создавать программно.

С помощью Mesh можно создать объект, отображающий меш. Для этого сначала нужно создать новый игровой объект GameObject, а

затем подключить к нему два компонента: MeshRenderer и MeshFilter. После этого объект будет готов для ис- пользования в сцене.

Эти шаги легко автоматизировать, а это значит, что данная процедура прекрасно подходит для мастера.

Создайте в папке Editor новый сценарий на C# с именем Tetrahedron.cs и добавьте в него следующий код:

```
using UnityEditor;
public class Tetrahedron : ScriptableWizard {
}
```

Добавьте следующий код в класс Tetrahedron:

```
using UnityEngine;
using System.Collections;
using System.Collections.Generic;
 [MenuItem("GameObject/3D Object/Tetrahedron")]
static void ShowWizard() {
ScriptableWizard.DisplayWizard<Tetrahedron>( "Create Tetrahedron", "Create");
}
```

Вернитесь в Unity и откройте меню GameObject (Игровой объект).

Добавьте в класс Tetrahedron следующую переменную, представляющую разме- ры тетраэдра:

```
public Vector3 size = new Vector3(1,1,1);
```

Вернитесь в Unity.

Теперь мастер позволяет вводить данные, но пока никак их не обрабатывает.

Вызывая метод DisplayWizard, мы передали ему две строки. Первая – для отображения в заголовке окна, а вторая – для отображения на кнопке Create (Создать). Когда пользователь щелкнет по этой кнопке, Unity вызовет метод OnWizardCreate класса мастера, сообщая тем самым, что данные готовы к использованию; когда OnWizardCreate вернет управление, Unity закроет окно.

Теперь реализуем метод OnWizardCreate, который выполнит основную задачу мастера. Используя переменную Size, он создаст объект Mesh и сконструирует игровой объект, отображающий этот меш.

Добавьте следующий метод в класс Tetrahedron:

```
void OnWizardCreate() {
```

```
var mesh = new Mesh();
Vector3 p0 = new Vector3(0,0,0); Vector3 p1 = new Vector3(1,0,0); Vector3 p2 = new Vector3(0.5f,0,Mathf.Sqrt(0.75f)); Vector3 p3 = new Vector3(0.5f,Mathf.Sqrt(0.75f), Mathf.Sqrt(0.75f)/3);
p0.Scale(size); p1.Scale(size); p2.Scale(size); p3.Scale(size);
mesh.vertices = new Vector3[] {p0,p1,p2,p3};
mesh.triangles = new int[] { 0,1,2,
0,2,3,
2,1,3,
0,3,1
};
mesh.RecalculateNormals(); mesh.RecalculateBounds();
var gameObject = new GameObject("Tetrahedron");
var meshFilter = gameObject.AddComponent<MeshFilter>();
meshFilter.mesh = mesh;
var meshRenderer= gameObject.AddComponent<MeshRenderer>();
meshRenderer.material= new Material(Shader.Find("Standard"));
}
```

Вернитесь в Unity, закройте и вновь откройте окно мастера.

4. Оптимизация игр в Unity

4.1. Приемы разработки сценариев

Разработка сценариев занимает довольно много времени, поэтому стоит познакомиться с некоторыми передовыми приемами их создания.

Кэширование ссылок на компоненты

Распространенной ошибкой в сценариях Unity является чрезмерное использование метода GetComponent(). Для примера рассмотрим следующий фрагмент, проверяющий состояние здоровья (значение health) персонажа и выполняющий отключение ряда компонентов, если здоровье падает ниже нулевой отметки (0), чтобы подготовиться к анимации смерти:

```
void TakeDamage() {
if (GetComponent<HealthComponent>().health < 0) { GetComponent<Rigidbody>().enabled = false;
GetComponent<Collider>().enabled = false; GetComponent<AIControllerComponent>().enabled = false; GetComponent<Animator>().SetTrigger("death");
}
}
```

При каждом вызове этот метод будет запрашивать ссылки на пять разных компонентов Component. Это хорошо с точки зрения потребления памяти, но это не очень дружелюбно по отношению к центральному процессору. И выглядит особенно проблематичным, если этот метод вызывается из метода Update(). Даже если это не так, его вызов все равно может совпасть по времени с другими важными событиями, такими как создание эффектов частиц, подмена объекта объектом ragdoll и т. д. Такой стиль программирования может на первый взгляд показаться безобидным, но в перспективе способен вызвать множество проблем и выполнение массы бесполезных операций.

Кэширование этих ссылок для дальнейшего использования потребует очень небольшого объема. То есть если память не является чрезвычайно ограниченным ресурсом, имеет смысл получить все ссылки во время инициализации и хранить их, пока они необходимы:

```
private HealthComponent _healthComponent; private Rigidbody _rigidbody;
private Collider _collider;
private AIControllerComponent _aiController; private Animator _animator;
void Awake() {
_healthComponent = GetComponent<HealthComponent>();
_rigidbody = GetComponent<Rigidbody>();
_collider = GetComponent<Collider>();
_aiController = GetComponent<AIControllerComponent>();
_animator = GetComponent<Animator>();
}
void TakeDamage() {
if (_healthComponent.health < 0) {
_rigidbody.detectCollisions = false;
_collider.enabled = false;
_aiController.enabled = false;
_animator.SetTrigger("death");
}
}
```

Кэширование ссылок на компоненты предотвращает их многократное извлечение, сохраняя при этом ресурсы центрального процессора за счет небольшого дополнительного использования памяти.

Самый быстрый метод получения ссылок на компоненты

Существует несколько вариантов метода GetComponent(), поэтому благоразумнее выбрать наиболее быстрый из них. Имеются три перегруженные версии метода: GetComponent(string), GetComponent<T>() и GetComponent(typeof(T)). Однако скорость их работы зависит от ис- пользуемой версии Unity.

В Unity 4 быстрее всех работает метод GetComponent(typeof(T)). Докажем это с помощью простого теста:

```
int numTests = 1000000; TestComponent test;
using (new CustomTimer("GetComponent(string)", numTests)) { for (var i = 0; i < numTests; ++i) {
test = (TestComponent)GetComponent("TestComponent");
}
}
```

```
using (new CustomTimer("GetComponent<ComponentName>", numTests)) { for (var i = 0; i < numTests; ++i) {
test = GetComponent<TestComponent>();
}
}
using (new CustomTimer("GetComponent(typeof(ComponentName))", numTests)) { for (var i = 0; i < numTests; ++i) {
test = (TestComponent)GetComponent(typeof(TestComponent));
}
}
```

Этот тест вызывает каждую версию метода GetComponent() миллион раз.

Версия GetComponent(string) не должна использоваться из-за своей медлительности и сохранена только из соображений полноты.

Как видите, версия GetComponent<T>() лишь незначительно быстрее, чем GetComponent(typeof(T)), в то время как GetComponent(string) оказался почти в 30 раз медленнее своих альтернатив (интересно, что он стал даже медленнее, чем в Unity 4). Многократное повторение тестов, вероятно, приведет к некоторым колебаниям в результатах, но в любом случае в версии Unity 5 можно использовать любую версию метода GetComponent(), основанную на типах, и результат будет примерно одинаковым.

```
int numTests = 1000000; Rigidbody test;
using (new CustomTimer("Cached reference", numTests))
{
for (var i = 0; i < numTests; ++i) { test = gameObject.rigidbody;
}
}
```

Удаление пустых объявлений обратных вызовов

При создании нового файла сценария с реализацией MonoBehaviour в Unity 4 или 5 редактор Unity автоматически создаст заготовки двух методов:

```
// Используется для инициализации
void Start () {
}
```

```
// Вызывается один раз в каждом кадре
void Update () {
}
```

Движок Unity определяет присутствие этих методов на этапе инициализации и добавляет их в список методов обратного вызова. Но если оставить пустые объявления в коде, это приведет к появлению небольших накладных расходов на их вызов движком.

Метод Start() вызывается только один раз, в момент создания игрового объекта, когда производится загрузка сцены или создается новый игровой объект. Однако этот метод добавляет ненужную нагрузку при любом вызове метода GameObject. Instantiate(), что обычно происходит во время ключевых событий, и может усугубить и без того сложную ситуацию, когда одновременно происходит много событий.

С другой стороны, метод Update() вызывается всякий раз, когда сцена перерисовывается. Если сцена содержит тысячи игровых объектов с компонентами, содержащими такие пустые методы Update(), на их обработку будет потрачено множество тактов процессорного времени, что отрицательно скажется на частоте кадров.

Проверим это утверждение с помощью простого теста. Создадим тестовую сцену с игровым объектом, имеющим компоненты двух видов – содержащие объявления пустых методов Update() и не содержащие:

```
public class CallbackTestComponent : MonoBehaviour { void Update () {}
}
public class EmptyTestComponent : MonoBehaviour {
}
```

Ниже представлены результаты тестирования 32 768 компонентов каждого вида. Если включить все объекты, в которых отсутствуют пустые методы, профилировщик не покажет какого-либо увеличения потребления процессора. Вы сможете заметить некоторые изменения в потреблении памяти и прочую активность VSync (Вертикальная синхронизация), но в целом картина останется в пределах нормы.

Чтобы исправить эту проблему, достаточно удалить объявления пустых методов. Движок Unity не обнаружит их и не будет вызывать. Иногда поиск объявлений пустых методов в большом проекте являет-

ся довольно сложной задачей, и решить ее вам помогут простые регулярные выражения.

Следующее регулярное выражение позволит найти все объявления пустых методов Update() в коде:

void\s*Update\s*?\(\s*?\)\s*?\n*?\{\n*?\s*?\}

С помощью этого регулярного выражения можно найти все стандартные объявления метода Update(), включающие избыточные пробелы и символы новой строки, разбросанные по всему объявлению метода.

Компоненты-одиночки

Недостаток решения на основе статического класса заключается в необходимости наследования самого низкоуровневого класса – Object. Это означает, что статические классы не могут наследовать класс MonoBehaviour и, соответственно, использовать его инструменты, имеющие отношение к Unity, включая обработку событий и сопрограммы. Кроме того, из-за отсутствия объекта для выбора теряется возможность увидеть данные в инспекторе во время выполнения. Однако эти инструменты доступны в глобальных классах-одиночках. Типичное решение этой проблемы заключается в реализации класса «компонента-одиночки», порождающего игровой объект и предоставляющего статические методы для глобального доступа.

Ниже приведено определение класса SingletonAsComponent:

```
public class SingletonAsComponent<T> : MonoBehaviour where T : SingletonAsComponent<T> {
private static T Instance;
protected static SingletonAsComponent<T> _Instance { get {
if(! Instance) {
T [] managers = GameObject.FindObjectsOfType(typeof(T)) as T[];
if (managers != null) {
if(managers.Length == 1) {
   Instance = managers[0]; return Instance;
} else if (managers.Length > 1) { Debug.LogError("You have more than one " +
typeof(T).Name + " in the scene. You only need 1, it's a singleton!");
for(int i = 0; i < managers.Length; ++i) { T manager = managers[i];
Destroy(manager.gameObject);
}
}
```

```
}
GameObject go = new GameObject(typeof(T).Name, typeof(T));
   Instance = go.GetComponent<T>(); DontDestroyOnLoad( In-
stance.gameObject);
}
return Instance;
}
set {
   Instance = value as T;
}
}
}
```

Поскольку объект должен быть глобальным и постоянным, сразу после создания игрового объекта требуется вызвать функцию DontDestroyOnLoad(). Эта специальная функция сообщает Unity, что объект должен сохраняться при смене сцен, вплоть до завершения выполнения приложения. При загрузке новой сцены объект не будет уничтожен и сохранит все свои данные.

Это определение класса предполагает два момента. Во-первых, поскольку в определении поведения используются обобщенные шаблоны, для создания конкретного класса требуется унаследовать конкретные типы. Во-вторых, должен быть определен метод, присваивающий значение переменной _Instance и осуществляющий необходимое приведение типов.

Например, ниже показано, что требуется для успешного создания нового класса MySingletonComponent, наследующего класс SingletonAsComponent:

```
public class MySingletonComponent : SingletonAsCompo-
nent<MySingletonCompone
nt> {
public static MySingletonComponent Instance {
get { return ((MySingletonComponent)_Instance); } set { _Instance
= value; }
}
}
```

Правильная очистка компонента-одиночки может вызывать сложности из-за особенностей уничтожения сцен в Unity. Метод OnDestroy() объекта вызывается во время выполнения, когда этот

объект должен быть уничтожен. Тот же метод вызывается во время завершения работы приложения, причем каждый компонент, присоединенный к каждому игровому объекту, имеет свой метод OnDestroy(), вызываемый Unity. Кроме того, приложение завершается при выходе из режима воспроизведения и возврата в режим редактирования. При этом уничтожение объектов происходит в случайном порядке и не гарантируется, что компонент-одиночка будет уничтожен в последнюю очередь.

Возможность обращения некоторых объектов к компоненту-одиночке при уничтожении объясняется тем, что такие компоненты часто реализуют шаблон «Наблюдатель», позволяя другим объектам подписываться/отписываться с его помощью на определенные задания, подобно тому как в Unity применяются методы обратного вызова, но в менее автоматизированной манере.

Для решения этой проблемы нужно внести три изменения. Во-первых, добавить дополнительный флаг в компонент-одиночку для проверки его состояния и выключения в соответствующие моменты, к которым относятся момент самоуничтожения компонента, а также завершение работы приложения (OnApplicationQuit() – еще один полезный метод обратного вызова для наследников MonoBehaviour, который будет вызван в этот момент):

```
private bool _alive = true;
void OnDestroy() { _alive = false; }
void OnApplicationQuit() { _alive = false; }
```

Во-вторых, необходимо реализовать проверку текущего состояния компонента для внешних объектов:

```
public static bool IsAlive {
get {
if ( Instance == null) return false;
return Instance._alive;
}
}
```

И наконец, любой объект, который пытается обратиться к компоненту-одиночке из метода OnDestroy(), должен перед вызовом Instance проверить состояние компонента с помощью свойства IsAlive. Например:

```
public class SomeComponent : MonoBehaviour {
void OnDestroy() {
```

```
if (MySingletonComponent.IsAlive) { MySingletonComponent.Instance.SomeMethod();
}
}
}
```

Это гарантирует, что никто не попытается получить доступ к Instance во время уничтожения. Однако если в дальнейшем будет решено использовать сразу несколько таких управляющих классов или разделить их поведение для удобства, придется изменить массу кода.

Имеются также другие альтернативы, например использование встроенного моста Unity между кодом сценария и интерфейсом инспектора.

Отключение неиспользуемых сценариев и объектов

Сцены бывают очень загруженными, особенно когда создаются большие открытые миры. Чем больше объектов вызывает код в методе Update(), тем хуже масштабирование и медленнее работает игра. Однако многие объекты обрабатываются впустую, поскольку находятся вне поля зрения игрока или просто слишком далеко. Отказ от обработки таких объектов невозможен в больших градостроительных симуляторах, где постоянно должно выполняться полное моделирование, но в играх от первого лица или гонках, где игрок перемещается по достаточно большой области, отключение невидимых объектов не окажет заметного влияния на процесс игры.

Отключение невидимых объектов

Иногда необходимо отключать компоненты или игровые объекты, когда они не видны. Unity имеет встроенные механизмы отключения объектов, которые не видны на камере, и объектов, скрытых за другими объектами, но эти механизмы не воздействуют на неотображаемые компоненты, такие как сценарии искусственного интеллекта. Их поведение необходимо контролировать вручную.

Эту проблему легко решить с помощью методов обратного вызова OnBecameVisible() и OnBecameInvisible() класса MonoBehaviour, которые вызываются, когда объект становится видимым или невидимым для всех камер в сцене. Кроме того, в сцене с несколькими камерами методы вызываются, только когда объект становится видимым для любой одной камеры и невидимым для всех камер. Это означает, что методы будут вызываться именно в нужные моменты.

Для включения и отключения отдельных компонентов по событиям изменения видимости можно использовать следующие методы:

void OnBecameVisible() { enabled = true; } void OnBecameInvisible() { enabled = false; }

А включение или отключение всего игрового объекта, к которому подключен компонент, можно реализовать так:

void OnBecameVisible() { gameObject.SetActive(true); } void OnBecameInvisible() { gameObject.SetActive(false); }

Отключение отдаленных объектов

В других ситуациях желательно отключать компоненты или игровые объекты, оказавшиеся достаточно далеко от игрока и потому едва заметны. Хорошими кандидатами для отключения в таких случаях являются блуждающие создания с искусственным интеллектом, которые должны быть видимыми на большом расстоянии, но, находясь там, они ничего не должны делать.

Ниже приводится простая сопрограмма, которая периодически проверяет расстояние до целевого объекта и отключает себя, если он находится слишком далеко:

```
[SerializeField] GameObject _target; [SerializeField] float _maxDistance;
 [SerializeField] int _coroutineFrequency;
void Start() {
StartCoroutine(DisableAtADistance());
}
IEnumerator DisableAtADistance() { while(true) {
float distSqrd = (Transform.position - _target.transform.position).sqrMagnitude;
if (distSqrd < _maxDistance * _maxDistance) { enabled = true;
} else {
enabled = false;
}
for (int i = 0; i < _coroutineFrequency; ++i) { yield return new WaitForEndOfFrame();
}
}
}
```

Нужно присвоить в инспекторе полю _target объект игрока, задать максимальное расстояние в поле _maxDistance и изменить частоту вызова сопрограммы с помощью свойства _coroutineFrequency. routineFreque удалится от объекта в свойстве _target дальше чем на расстояние _maxDistance, он будет отключен. Он будет снова включен при приближении на расстояние меньше порогового.

Метод Update, сопрограммы и метод InvokeRepeating

Метод Update вызывается в каждом кадре, но иногда неумелые попытки добиться уменьшения частоты вызовов метода Update приводят к более частому вызову пустого метода:

```
void Update() {
_timer += Time.deltaTime;
if (_timer > _aiUpdateFrequency) { ProcessAI();
_timer -= _aiUpdateFrequency;
}
}
```

Такое определение этой функции приводит к вызову пустой функции практически в каждом кадре. На самом деле все гораздо хуже, потому что выполняется проверка логического значения, которая почти всегда возвращает false.

Это наглядный пример функции, которую нужно преобразовать в сопрограмму, чтобы воспользоваться возможностью задерживать вызовы:

```
void Start() {
StartCoroutine(UpdateAI());
}
IEnumerator UpdateAI() { while (true) {
yield return new WaitForSeconds(_aiUpdateFrequency); ProcessAI();
}
}
```

Однако этот подход имеет свои недостатки. С одной стороны, сопрограмма вносит дополнительные накладные расходы в сравнении с вызовом обычной функции, а так- же использует кучу для сохранения своего текущего состояния до следующее вызова. Во-вторых, после инициализации выполнение сопрограммы не зависит от метода Update() игрового объекта и будет вызываться независимо от того, активен объект или нет.

Однако в определенных ситуациях преимущества отсутствия вызовов в большинстве кадров перевешивают дополнительные затраты на вызов в каждом кадре. На самом деле, когда в операторах yield не выполняется ничего сложного, часто используют более простой вариант этого подхода, основанный на функции InvokeRepeating(), которая имеет меньшие накладные расходы:

```
void Start() {
InvokeRepeating("ProcessAI", 0f, _aiUpdateFrequency);
}
```

Обратите внимание, что функция InvokeRepeating также не зависит от метода Update() игрового объекта и будет продолжать вызываться, даже когда объект отключен. Независимо от выбранного подхода существует дополнительный риск вызова слишком большого количества методов в одном и том же кадре. Представьте себе тысячи объектов, которые одновременно инициализируются при инициализации сцены. Каждый раз, по истечении _aiUpdateFrequency секунд, они все вместе будут запускать метод ProcessAI() в одном и том же кадре, что вызовет огромный всплеск потребления центрального процессора.

Эта проблема имеет следующие возможные решения:

– выполнять задержку на случайный интервал времени при каждом срабатывании таймера или запуске сопрограммы;

– распределить инициализацию сопрограмм по кадрам;

– делегировать ответственность за вызов обновлений некоторому управляющему классу, который ограничит количество вызовов в каждом кадре.

Уменьшив количество излишних определений метода Update с помощью простой сопрограммы, можно ликвидировать массу ненужных накладных расходов, поэтому такое преобразование следует выполнять всегда, когда это возможно. Можно также пересмотреть решение оригинальной проблемы, чтобы предотвратить одновременное появление массы событий.

Другой подход к оптимизации обновлений – отказ от использования метода Update(), точнее вызывать его, но только один раз. При вызове метода Update() происходит передача управления между низкоуровневой реализацией игрового объекта и его управляемым представлением, что является затратной задачей.

То есть накладные расходы можно уменьшить, ограничив количество передач управления между управляемым и неуправляемым ко-

дами. Сделать это можно с помощью вездесущего класса, вызывающего пользовательские методы обновления во всех пользовательских компонентах. На самом деле многие Unity-разработчики используют этот подход в своих проектах изначально, поскольку он позволяет им полностью контролировать обновления в системе, включая такие моменты, как паузы при выводе всплывающих меню и приостановки при манипуляции эффектами.

Все объекты, подлежащие интегрирации с такой системой, должны иметь общую точку входа. Этого можно достичь с помощью интерфейсного класса.

В данном случае достаточно реализовать следующий интерфейс, который требует определить единственный метод:

```
public interface IUpdateable {
void OnUpdate(float dt);
}
```

Далее определим класс MonoBehaviour, реализующий этот интерфейс:

```
public class UpdateableMonoBehaviour : MonoBehaviour, IUpdateable
{
public virtual void OnUpdate(float dt) {}
}
```

Обратите внимание, что метод получил имя OnUpdate(), а не Update().

Так мы сохранили смысл оригинальной идеи в пользовательской версии и избежали конфликта имен со стандартным методом Update().

Метод OnUpdate() в классе UpdateableMonoBehaviour извлекает текущее время (dt), чтобы избавить нас от массы ненужных вызовов метода Time.deltaTime. Кроме того, функция отмечена спецификатором virtual, чтобы ее можно было переопределить в производных классах. Как известно, Unity автоматически определяет присутствие методов с именем Update() и вызывает их, а так как пользовательские методы обновления имеют другое имя, необходимо реализовать некий везде- сущий класс «GameLogic», который будет вызывать эти методы.

При инициализации таких компонентов необходимо сообщать объекту GameLogic об их создании и уничтожении, чтобы он знал, когда нужно или не нужно вызывать функцию OnUpdate().

В следующем примере считается, что класс GameLogic является компонентом-одиночкой, и имеет статические функции для подписки и отказа от подписки

Наиболее подходящими для подключения объекта класса MonoBehaviours к системе являются методы Start() и OnDestroy():

```
void Start() {
GameLogic.Instance.RegisterUpdateableObject(this);
}
void OnDestroy() {
GameLogic.Instance.DeregisterUpdateableObject(this);
}
```

Выбор метода Start() объясняется просто: к моменту его вызова для всех других компонентов уже будут вызваны их методы Awake().

Обратите внимание, что метод Start() определен в базовом классе MonoBehaviour, поэтому, если определить метод Start() в производном классе, он фактически переопределит метод базового класса и Unity будет использовать метод производного класса. Поэтому предпочтительнее реализовать виртуальный метод Initialize(), чтобы производные классы могли переопределить его для настройки инициализации без вмешательства в действия базового класса, такие как уведомление объекта GameLogic о существовании компонента.

И наконец, реализуем класс GameLogic. Его реализация во многом подобна реализации компонента-одиночки, независимо от использования MessagingSystem в нем. В любом случае, объект класса UpdateableMonoBehaviour должен оформить подписку или отказаться от нее, как объект IUpdateableObject, а класс GameLogic должен использовать свой метод Update() для обхода всех зарегистрированных объектов и вызова их методов OnUpdate().

Определение класса для системы GameLogic:

```
public class GameLogic : SingletonAsComponent<GameLogic> {
public static GameLogic Instance {
get { return ((GameLogic)_Instance); } set { _Instance = value; }
}
List<IUpdateableObject> _updateableObjects = new
List<IUpdateableObject>(); public void RegisterUpdateableOb-
ject(IUpdateableObject obj) {
if (!_Instance._updateableObjects.Contains(obj)) {
_Instance._updateableObjects.Add(obj);
}
```

```
}
public void DeregisterUpdateableObject(IUpdateableObject obj) { if
(_Instance._updateableObjects.Contains(obj)) {
_Instance._updateableObjects.Remove(obj);
}
}
void Update() {
float dt = Time.deltaTime;
for(int i = 0; i < _Instance._updateableObjects.Count; ++i) {
_Instance._updateableObjects[i].OnUpdate(dt);
}
}
}
```

Если все пользовательские классы MonoBehaviours будут наследовать класс UpdateableMonoBehaviour, мы фактически заменим N вы- зовов функции Update() единственным вызовом метода Update() и N вызовами виртуальных функций. Это предотвратит заметное снижение производительности, поскольку подавляющее большинство операций обновления будет выполняться внутри управляемого кода и передача управления между управляемым и неуправляемым кодами будет осуществляться реже.

Ускорение проверки отсутствия ссылки на игровой объект

Оказывается, что сравнение ссылки на объект Unity со значением null вызывает передачу управления между управляемым и неуправляемым кодом, что ожидаемо привносит нежелательные затраты:

```
if (gameObject != null) {
// делаем что-то с объектом игры
}
```

Существует простая альтернатива, позволяющая выполнить ту же проверку, но почти в два раза быстрее:

```
if (!System.Object.ReferenceEquals(gameObject, null)) {
// делаем что-то с объектом игры
}
```

Это относится и к игровым объектам, и к компонентам, и к другим объектам Unity, имеющим управляемое и неуправляемое представления. Однако элементарное тестирование показывает, что оба

способа выполняются лишь несколько наносекунд на процессоре Intel Core i5 3570K.

4.2. Пакетная обработка. Аудио и видео

Преимущества пакетной обработки

В 3D-графике и играх понятие пакетной обработки представляет собой весьма общий термин, описывающий процесс группировки большого количества непостоянных частей данных и их совместной обработки как единого большого блока данных. Целью этого процесса является сокращение времени вычислений, часто с помощью параллельной обработки, или уменьшение накладных расходов, когда целые пакеты рассматриваются как отдельные элементы. В некоторых случаях пакетная обработка применяется к мешам, множествам вершин, граней, UV-координатам и т. д., используемым для представления трехмерного объекта. Однако тот же термин относится и к пакетной обработке аудиофайлов, спрайтов, текстур и других больших наборов данных.

Поэтому, чтобы избежать путаницы при упоминании пакетной обработки в Unity, под ней понимаются два главных механизма обработки файлов мешей: статическая и динамическая пакетная обработка. Пакетная обработка позволяет повысить производительность приложения, но только когда она используется разумно. Ее использование связано с массой нюансов и запутанных условий, которые важны для улучшения производительности. На самом деле в некоторых случаях пакетная обработка может даже ухудшать производительность, если используется для обработки наборов данных, непригодных для пакетирования.

Системы пакетной обработки в Unity фактически являются черными ящиками, о внутренней структуре которых компания Unity Technologies не раскрывает подробной технической информации. Но, основываясь на поведении, результатах профилирования и перечне требований к условиям использования, можно многого добиться.

Материалы и шейдеры

Шейдеры – это короткие программы, которые определяют, как графический процессор должен отображать входные данные, описы-

вающие вершины и пиксели. Шейдеры сами по себе не обладают необходимыми сведениями о состоянии, чтобы сделать что-то ценное.

Важной обязанностью системы материалов в Unity является предоставление этой информации шейдерам. Это значит, что шейдер может использоваться для отображения объекта только вместе с материалом. Даже вновь импортированным мешам, вводимым в сцену без назначенных материалов, автоматически присваивается материал по умолчанию, который придает им базовый диффузный шейдер и белый цвет.

Эти две системы охватывают большинство переменных состояния визуализации. То есть если свести к минимуму количество материалов, используемых для визуализации сцены, автоматически уменьшится количество необходимых изменений состояния визуализации и, соответственно, количество времени, затрачиваемого центральным процессором на подготовку кадров для графического процессора.

Начнем с простой демонстрации поведения материалов и пакетной обработки. Но прежде отключим несколько глобальных параметров механизма визуализации, чтобы не отвлекаться:

– выберите в меню пункт Edit – Project Settings –Quality (Правка – Параметры проекта – Качество) и установите значение Disable Shadows (Отключить тени) в параметре Shadows (Тени) или установите уровень качества по умолчанию Fastest (Быстрый);

– выберите в меню пункт Edit – Project Settings –Player (Правка – Параметры проекта – Проигрыватель), откройте вкладку Other Settings (Прочие параметры) и сбросьте флажки Static Batching (Статическая пакетная обработка), Dynamic Batching (Динамическая пакетная обработка) и GPU Skinning (Скининг графическим процессором).

Далее создадим сцену с единственным направленным источником света и восемью мешами – четыре куба и четыре сферы, – каждый из которых имеет уникальный материал, позицию, поворот и масштаб

Взгляните на параметр Batching (Пакетная обработка) во всплывающей панели GameView’s Stats (Статистика представления игры), и вы увидите, что число пакетов равно девяти (это же число стоит в параметре Draw Calls (Вызовы системы визуализации) во всплывающей панели Stats (Статистика) в Unity 4). Если в параметре Clear Flags (Флаги очистки) камеры выбран вариант Don’t Clear (Не очищать),

один пакет будет использован для рисования фона. Это может быть объект Skybox сцены или прямоугольник, заполняющий экран пикселями, окрашенными в цвет, определяемый свойством камеры Background (Фон).

Прочие восемь пакетов используются для рисования восьми объектов. В любом случае, каждый вызов системы визуализации включает ее подготовку с учетом свойств материала и посылает графическому процессору команду отобразить данный меш в его текущей позиции, с текущими поворотом и масштабом. Материал определяет шейдер, управляющий программируемыми частями графического конвейера (управляющими отображением вершин и фрагментов).

Примечательно, что если в параметре Rendering Path (Режим отображения) настройки плеера выбран вариант Forward (Упреждающий), включение и выключение направленного источника света в сцене не повлияет на количество пакетов, их будет по- прежнему девять. Первый направленный источник света в режиме Forward не вносит никаких дополнительных затрат, по крайней мере с точки зрения вызовов системы визуализации. При добавлении других источников света в сцену – направленных, точечных, локальных или пространственных – все объекты обрабатываются с дополнительным «прогоном» шейдеров для каждого источника, в зависимости от значения параметра Pixel Light Count (Количество пиксельных источников света) в настройках Quality (Качество); приоритет при обработке будут иметь источники света с более высокой яркостью.

Как упоминалось выше, теоретически можно минимизировать количество вызовов системы визуализации, уменьшив частоту смены состояний визуализации. То есть один из путей достижения цели – уменьшение количества используемых материалов. Но если для всех объектов использовать один и тот же материал, это ничего не даст, и по-прежнему останется девять пакетов:

Причина в том, что без группировки информации о мешах число изменений состояния визуализации останется тем же. Без пакетной обработки какого-либо вида система визуализации не сможет понять, что выполняется перезапись одинаковых состояний визуализации при выводе нескольких мешей, и будет перезаписывать их снова и снова. Это дает возможность выводить в процессе визуализации все меши вместе как один объект и избежать ненужных вызовов визуализации.

Динамическая пакетная обработка

Цель динамической пакетной обработки – объединить простые меши в большие группы и передать их системе визуализации в виде одного меша. Ее можно применять только к мешам, видимым в настоящий момент для камеры, а это значит, что основные вычисления должны производиться во время выполнения и не могут быть выполнены заранее. Соответственно, набор объединяемых объектов будет изменяться от кадра к кадру. Этим и объясняется применение термина «динамическая» к такой пакетной обработке.

Если вернуться на страницу Player Settings (Настройки плеера) и включить параметр Dynamic Batching (Динамическая пакетная обработка), количество пакетов должно уменьшиться с девяти до шести. Механизм динамической пакетной обработки автоматически определил, что объекты имеют одинаковый материал и схожие параметры мешей, и их можно объединить в один пакет. Это сэкономит ресурсы центрального процессора и высвободит больше времени для решения других задач, например искусственного интеллекта и моделирования законов физики.

Полный список требований, необходимых для успешной динамической пакетной обработки мешей, можно найти в электронной документации Unity по адресу: http://docs.unity3d.com/Manual/DrawCallBatching.html:

– все экземпляры мешей должны использовать одну и ту же ссылку на материал;

– динамическая пакетная обработка применяется только к системам частиц и мешам. Компоненты SkinnedMesh и другие не могут объединяться;

– общее количество атрибутов вершинного шейдера не должно превышать 900;

– все меши должны использовать либо однородное, либо неодно- родное масштабирование, но не их сочетание;

– все экземпляры мешей должны ссылаться на один и тот же файл с картой освещенности;

– шейдеры материалов не должны зависеть от количества проходов;

– экземпляры мешей не должны принимать тени в реальном времени.

Имеется также несколько недокументированных требований, выявленных в ходе конференций Unite:

– пакет вмещает не более 300 мешей;

– пакет не должен содержать более 32 000 индексов мешей.

Статическая пакетная обработка

Еще один механизм пакетной обработки в Unity реализует статическую пакетную обработку. Его назначение – дать возможность объединять в пакеты меши любого размера с той же целью и теми же способами, что и при динамической пакетной обработке, но при другом наборе условий. Существенное отличие между двумя методами пакетной обработки заключается в том, что статическая пакетная обработка происходит во время инициализации приложения, а динамическая – во время выполнения. Поэтому разработчик имеет намного больше возможностей для управления статической пакетной обработкой.

Система статической пакетной обработки имеет свой набор требований:

– как следует из названия, меш должен иметь флаг Static (Статический);

– для каждого меша, участвующего в статистической пакетной обработке, должна выделяться дополнительная память;

– экземпляры мешей могут поступать из любых источников, но должны использовать один и тот же материал.

Особенности использования статической пакетной обработки

Система статической пакетной обработки не лишена недостатков. Так как сборка пакетов осуществляется путем объединения мешей в один большой меш, система статической пакетной обработки обладает несколькими особенностями, о которых следует знать. В зависимости от сцены эти особенности вызывают проблемы разной серьезности, от незначительных неудобств до крупных недостатков:

– экономия на вызовах системы визуализации не сразу становит ся заметной в окне Stats;

– статические объекты не должны вводиться в сцену во время выполнения;

– нельзя изменять параметры преобразования мешей из статиче ских пакетов;

– если любой из мешей в статическом пакете является видимым, отображается вся группа.

Отладка статической пакетной обработки в режиме редактирования

Определение общего эффекта от применения статической пакетной обработки в сцене несколько затруднено, поскольку статическая пакетная обработка никак не отражена в режиме редактирования. Все волшебство происходит в момент инициализации сцены, что не дает возможности оценить реальные преимущества статической пакетной обработки. При этом особенно важно не оставлять реализацию этой функции на завершающий этап работы над проектом, когда потребуется потратить много времени на запуск, настройку и повторный запуск сцены, чтобы убедиться в ожидаемом уменьшении количества вызовов визуализации. Следовательно, применение статической пакетной обработки лучше начинать одновременно с созданием новой сцены.

Статические меши не должны создаваться во время выполнения

Вновь созданные статические объекты в сцене не присоединяются к существующему пакету автоматически, в противном случае это привело бы к огромным накладным расходам, связанным с перерасчетом мешей и синхронизацией с системой визуализации. Поэтому движок Unity даже не пытается сделать это.

Точно так же не следует перемещать меши из статических пакетов после их объединения, поскольку это вызовет огромные нагрузки на процессор. Система статической пакетной обработки объединяет принятые исходные данные в новый объект меша. Перемещение оригинального меша невозможно ни при установленном, ни при сброшенном флаге Static, и изменение параметров преобразования объекта не распознается системой Unity.

Привнесение искусства. Аудио. Файлы текстур. Файлы мешей и анимации.

Снизить потребление памяти, уменьшить размер исполняемого файла, достичь максимальной скорости загрузки или обеспечить согласованность частоты кадров можно разными путями. Одни методы всегда приводят к нужным результатам, другие требуют внимательности и предусмотрительности, поскольку могут привести к снижению качества или увеличить риск появления узких мест в других компонентах.

Аудио

В зависимости от назначения проекта его аудиофайлы могут иметь любые размеры, от занимающих солидную часть диска до очень небольших. Unity может использоваться как для создания небольших приложений, требующих минимума звуковых эффектов и единственного трека с фоновой мелодией, так и для разработки ролевых игр, нуждающихся в речевых диалогах с общим объемом в миллионы строк, музыкальных треках и звуках окружения.

Многие разработчики с удивлением обнаруживают, что процесс обработки аудио может стать значимым потребителем памяти и ресурсов процессора. Звуком часто пренебрегают обе заинтересованные стороны игровой индустрии; разработчики, как правило, не подключают многих ресурсов до самого последнего момента, а пользователи редко обращают внимание на звук.

Обработка аудио может стать узким местом в силу нескольких причин. Чрезмерное сжатие, большое количество операций с аудио, масса активных аудиоклипов, использование неэффективных методов хранения в памяти и низкая скорость доступа – все это способствует увеличению потребления памяти и ресурсов центрального процессора. Но немного усилий – и разумный подход к выбору параметров настройки позволит избежать негативного восприятия игры пользователем.

Загрузка аудиофайлов

Первый параметр импортированного аудиоклипа в редакторе Unity определяет тип загрузки файла (Load Type (Тип загрузки)). В этом параметре можно выбрать одно из трех значений:

– Decompress On Load (Распаковывать при загрузке);

– Compressed In Memory (Хранить в памяти в сжатом виде);

– Streaming (Потоковая передача).

Фактически от этого параметра зависят доступные значения и порядок использования остальных параметров, которые мы рассмотрим ниже. Он определяет метод загрузки выбранного аудиофайла.

Если выбрать вариант Decompress On Load (Распаковывать при загрузке), аудиофайл, хранящийся на диске в сжатом виде для экономии места, будет распакован при загрузке в память.

Если выбрать вариант Compressed In Memory (Хранить в памяти в сжатом виде), сжатый файл загрузится в память без обработки и будет распаковываться непосредственно при воспроизведении. Здесь за

счет увеличения потребления центрального процессора при воспроизведении увеличивается скорость загрузки клипа и уменьшается потребление памяти во время выполнения.

И наконец, если выбрать вариант Streaming (Потоковая передача), файлы будут загружаться, декодироваться и воспроизводиться «на лету» с использованием небольшого буфера. Этот метод имеет минимальное потребление памяти для отдельных аудиоклипов и больше всего нагружает центральный процессор. К сожалению, он не позволяет сослаться на тот же буфер повторно. Попытка запустить несколько потоков воспроизведения одного и того же аудиоклипа приведет к выделению в памяти нового буфера для каждого потока, что при бездумном применении этого метода приведет к значительному потреблению оперативной памяти и ресурсов центрального процессора.

Профилирование аудио

Чтобы подтвердить большую часть из описанного выше, воспроизведем несколько экземпляров произвольного аудиоклипа с помощью нескольких источников звука в сцене и проведем сравнительный анализ с помощью аудиопрофилировщика, отображающего потребление памяти и ресурсов центрального процессора при использовании разных значений в параметре Load Type (Тип загрузки).

После загрузки и перехода в режим воспроизведения редактор распакует аудиофайлы, затратив некоторый объем памяти и ресурсов центрального процессора. Затраты памяти на этот процесс можно увидеть в области Audio Area профилировщика. Однако после перезапуска сцены затраты памяти на распаковку аудиофайлов внезапно снижаются практически до 0 Кбайт, поскольку файлы уже были распакованы и редактор избавился от данных, которые ему больше не нужны. Это не соответствует реальной ситуации, так как приложению потребуется выполнить эту распаковку.

То есть, чтобы получить более точные результаты, следует производить профилирование автономной или удаленной версии приложения на предполагаемой платформе/устройстве.

Форматы кодирования и уровни качества

Unity поддерживает три распространенных формата кодирования аудиофайлов, а также несколько специальных для конкретных

платформ (например, HEVAG для PS Vita и XMA для XBox One). Рассмотрим три основных формата кодирования:

– Compressed;

– PCM;

– ADPCM.

Алгоритм сжатия, применяемый при выборе формата Compressed, зависит от целевой платформы. В автономных приложениях, WebGL и других немобильных платформах применяется сжатие Ogg-Vorbis, а мобильные платформы используют сжатие MPEG-3 (MP3).

Движок Unity поддерживает импорт аудиофайлов многих популярных аудиоформатов, но в исполняемый файл встраивается поддержка только одного из перечисленных. Статистические данные, получаемые для формата, который определяется параметром Compression Format (Формат сжатия), отражают, сколько места на диске сохраняется при сжатии и сколько памяти экономится при воспроизведении во время выполнения.

Выбор формата сжатия/кодирования коренным образом влияет на качество, размер файла и потребление памяти при воспроизведении во время выполнения, и только при выборе варианта Compressed имеется возможность изменять качество независимо от характеристик файла. Форматы PCM и ADPCM не предоставляют такой удобной возможности, здесь качество жестко определяется размером файла, поэтому, принимая решение пожертвовать качеством ради уменьшения размера файла, нужно сменить характеристики файла.

Формат PCM не предусматривает сжатия, лишен искажений и обеспечивает качество, близкое к качеству аналогового аудио. Он характеризуется большим размером файлов и наивысшим качеством звука. Его рекомендуется использовать для очень коротких звуковых эффектов, требующих высокой четкости, недостижимой при применении сжатия.

С другой стороны, формат ADPCM более эффективно использует память и ресурсы центрального процессора, чем PCM, но сжатие привносит изрядное количество шумов. Эти шумы незаметны в коротких звуковых эффектах хаоса, подобных звукам взрывов, столкновений и ударов, где посторонние шумы не являются критичными.

Наконец, формат Compressed позволяет уменьшить размеры файлов и получить качество звучания хуже, чем при использовании формата PCM, но лучше, чем при использовании формата ADPCM, за

счет дополнительного использования ресурсов центрального процессора. Этот формат рекомендован к применению в большинстве случаев. При выборе этого варианта появляется возможность настроить уровень качества алгоритма сжатия и достичь нужного баланса между качеством и размером файла. Работая ползунком Quality (Качество), можно найти минимальный уровень качества, не вносящий искажений, заметных пользователю.

Уменьшение числа активных источников звука

Так как любой активный источник звука потребляет ресурсы центрального процессора, само собой разумеется, что можно снизить нагрузку на центральный процессор, избавившись от избыточных источников звука в сцене. Одно из решений состоит в том, чтобы жестко ограничить количество экземпляров аудиоклипов, воспроизводимых одновременно, поскольку это приводит к избыточности звуковых эффектов. Добиться этого можно, создав объект, управляющий источниками звука. Он должен принимать запросы и воспроизводить звуки через любой незанятый источник звука.

Объект управления также должен ограничивать количество одновременно воспроизводимых экземпляров одного и того же звукового эффекта и общее количество звуковых эффектов, воспроизводимых одномоментно. При этом предпочтение должно отдаваться звуковым 2D-эффектам или отдельным экземплярам звуковых 3D-эффектов.

Обратите внимание, что окружающие 3D-звуки все еще придется помещать в определенные места в сцене, чтобы использовать эффект логарифмирования громкости, обеспечивающий псевдо 3D-эффект, поэтому система управления не является идеальным решением. Ограничение воспроизведения аудиоэффектов проще реализовать путем сокращения общего числа источников. С этой целью лучше всего удалять некоторых из них или объединять в один громкий источник звука. Естественно, такой подход влияет на качество восприятия пользователем, поскольку звук будет исходить из одного источника, а не из нескольких, соответственно, данный способ следует использовать с осторожностью.

Выбор формата кодирования

Если приложение не сильно ограничено объемом памяти и дискового пространства, сократить потребление ресурсов центрального процессора можно с использованием формата WAV, требующего

меньше затрат на декодирование аудиоданных при воспроизведении. С другой стороны, при невысоких требованиях к вычислительным ресурсам можно сэкономить пространство, использовав кодирование со сжатием.

Файлы аудиомодулей для фоновой музыки

Файлы аудиомодулей, иначе называемые трекерными модулями, – прекрасное средство экономии памяти без заметной потери качества. В Unity поддерживаются файлы с расширениями .it, .s3m, .xm и .mod. В отличие от обычных аудиофайлов в формате PCM, которые читаются как потоки битов данных, декодируемые во время выполнения, трекерные модули содержат множество небольших высококачественных образцов звука в формате PCM и определение полной звуковой дорожки в виде, схожем с нотной записью музыки, описывающем, когда, где, как громко, с каким тактом и с применением каких специальных эффектов должен воспроизводиться каждый из образцов. Этот подход значительно уменьшает размер файла без потери качества.

Файлы текстур

Термины «текстура» и «спрайт» часто подменяются в разработке игр, поэтому считаю важным уточнить, что в Unity 3D под термином текстура подразумевается простой файл изображения, то есть список данных о цветах, указывающий интерпретирующей программе, какой цвет придать каждому пикселю изображения. Спрайт, напротив, – это 2-мерный эквивалент меша, то есть отдельный прямоугольник на плоскости перед текущей камерой. Имеются также листы спрайтов – коллекции отдельных изображений, хранящихся в файле текстуры. Выделить отдельные текстуры с кадрами анимации из этих файлов можно при помощи инструмента пакетной обработки спрайтов Unity.

Форматы сжатия

По аналогии с аудиофайлами, Unity поддерживает различные методы сжатия файлов текстур для увеличения эффективности их хранения. Импортируя файл текстуры, можно определить несколько параметров. Первый параметр: Texture Type (Тип текстуры). Он влияет не на сам файл изображения, а на порядок его интерпретации, обработки и сжатия во время сборки исполняемого файла.

Для большинства типов текстур в Unity поддерживаются только три типа сжатия: Compressed (Сжатый), 16-bit (16-битная палитра) и True Color (24-битный цвет). И только при выборе варианта Advanced (Расширенный) в параметре Texture Type (Тип текстуры) появляется возможность более тонкой настройки.

Выбор форматов сжатия для типа Advanced (Расширенный) шире и разнообразней, причем он одинаков для версий 4 и 5 Unity. Некоторые форматы поддерживают альфа-канал, другие – нет.

Кроме того, разные форматы обусловливают разные уровни производительности на этапе инициализации сцены, когда выполняются распаковка и передача в графический процессор. Причем производительность зависит также от целевой платформы и выбранного формата. Тип Advanced (Расширенный) поддерживает вариант Automatic Compressed (Автоматический выбор сжатия), когда Unity пытается автоматически подобрать лучший вариант для конкретной платформы и устройства. Этот вариант можно использовать, когда нет полной уверенности в выборе оптимального формата для игры или для тестирования различных форматов, чтобы определить лучший для целевого устройства.

Улучшение производительности обработки текстур

Рассмотрим, как с помощью параметров настройки можно увеличить производительность обработки файлов текстур в зависимости от ситуации и содержимого импортируемых файлов. В каждом конкретном случае будет описываться выбор значений параметров и проанализирован эффект их применения: влияние на потребление памяти и ресурсов центрального процессора, увеличение или уменьшение качества текстур и при каких условиях их можно применять.

Уменьшение размеров файлов текстур

Чем больше размер файла текстуры, тем большей пропускной способностью должна обладать память графического процессора для ее вывода. Если общий объем данных, передаваемых в единицу времени, превысит пропускную способность памяти графической карты, образуется узкое место, поскольку графический процессор требует загрузки всех текстур до перехода к следующему этапу отображения. Чем меньше текстура, тем проще пропустить ее через графический конвейер, поэтому следует найти золотую середину между качеством и производительностью.

Использование mip-текстур

Не имеет смысла отображать мелкие отдаленные объекты, такие как скалы или деревья, с помощью излишне детализированных текстур, если игрок не сможет рассмотреть детали или если потеря производительности не оправдывает воспроизведения подробностей. Mip-текстуры, призванные для решения таких проблем , – это предварительно подготовленные наборы текстур с разным разрешением. Во время выполнения графический процессор выбирает вариант mip-текстуры, основываясь на размере поверхности, отображаемой в перспективной проекции, а затем масштабирует mip- текстуру.

При установке флага Generate Mip Maps (Генерировать mip- текстуры), Unity автоматически генерирует текстуры с низким разрешением. Альтернативные текстуры создаются с использованием методов высококачественной интерполяции и фильтрации в редакторе, а не во время выполнения.

Недостатко mmip-текстур является увеличение размера файла и времени их загрузки. В конечном итоге включение mip-текстурирования увеличивает размер файла текстуры приблизительно на 33 процента. Существуют ситуации, когда применение mip-текстурирования ничего не дает.

Применение mip-текстурирования оправдано, только когда дело касается текстур, которые должны отображаться на различных расстояниях от камеры. Для текстур, которые всегда отображаются на одном и том же расстоянии от основной камеры, альтернативные mip-текстуры никогда не используются и просто зря занимают место. В этом случае следует отключить функцию mip-текстурирования, сбросив флаг Generate Mip Maps (Генерировать mip-текстуры).

Если используется лишь одна из альтернативных mip-текстур, следует отключить mip-текстурирование и уменьшить разрешение исходного файла текстуры.

Кроме того, кандидатами на отключение функции mip-текстурирования являются:

– практически все файлы текстур в 2D-играх;

– текстуры пользовательского интерфейса (GUI);

– текстуры для мешей, спрайтов и эффектов частиц, которые всегда отображаются рядом с камерой, их примером является сам персонаж игрока, все носимые им объекты и все эффекты частиц, которые всегда окружают игрока.

Текстурные атласы

Текстурные атласы – это технология объединения множества небольших изолированных текстур в одну большую текстуру для минимизации количества материалов и, как следствие, обращений к системе визуализации при применении динамической пакетной обработки.

Каждый уникальный материал требует дополнительного обращения к системе визуализации, но каждый материал поддерживает только одну первичную текстуру. Объединив все текстуры в одну огромную текстуру, можно уменьшить количество обращений к системе визуализации для отображения объектов с общей текстурой.

Выгоды при этом очевидны: сокращение числа обращений к системе визуализации приведет к уменьшению нагрузки на центральный процессор и повышению частоты кадров, если приложение ограничено возможностями центрального процессора. Это произойдет без потери качества и существенного увеличения потребления памяти. Все, что следует сделать, – это применить систему динамической пакетной обработки Unity для уменьшения количества обращений к системе визуализации. Обратите внимание, что использование текстурных атласов не влияет на пропускную способность памяти, поскольку количество передаваемых данных не изменяется. Они просто будут собраны вместе в один большой файл текстуры.

Текстурные атласы часто применяются для отображения элементов пользовательского интерфейса и в играх с большим количеством 2D-графики и практически необходимы при разработке мобильных игр в Unity, поскольку вызовы визуализации обычно являются узким местом на этих платформах.

Текстурные атласы следует применять не только к 2D-графике и элементам пользовательского интерфейса. Этот метод можно использовать и с 3D-мешами, если имеется множество текстур с низким разрешением. 3D-игры с низким разрешением текстур или в художественном стиле плоскостной затушевки с малым количеством поли- гонов являются идеальными кандидатами для применения атласов.

Однако, так как динамическая пакетная обработка применяется только к мешам без анимации, нет никаких оснований объединять в атлас текстуры анимированных персонажей. Поскольку персонажи анимированы, графическому процессору необходимо обработать положение всех костей объекта при текущем состоянии анимации. Это означает, что для каждого персонажа необходимо произвести отдель-

ные расчеты, что повлечет дополнительные обращения к системе визуализации независимо от общности применяемых материалов.

Следовательно, полезность объединения текстур для анимированных персонажей заключается только в удобстве и экономии пространства. Главным недостатком атласов является увеличение времени разработки и расходов на нее. Применение текстурных атласов требует много усилий, и еще больше, чтобы оценить, окупится ли затраченный труд. Кроме того, следует остерегаться файлов атласов, слишком больших для целевой платформы.

Некоторые устройства (особенно мобильные) имеют ограниченный размер низкоуровневой кэш-памяти графического процессора. Если атлас слишком велик, его нужно разделить на меньшие текстуры, размеры которых соответствуют объему памяти целевого устройства. Если при отображении в каждом вызове потребуется выводить текстуры из различных частей атласа, это не только приведет к промахам кэша, но и уменьшит пропускную способность памяти, поскольку текстуры постоянно будут браться из видеопамяти и низкоуровневого кэша.

Итак, текстурные атласы не являются идеальным решением. Если их применение не приведет к значительному увеличению производительности, не следует тратить слишком много времени на их реализацию.

Вообще, текстурные атласы следует применять к высококачественным мобильным играм среднего масштаба с самого начала проекта, следя за тем, чтобы размер текстур не превышал ограничений целевой платформы и устройства. С другой стороны, простейшие мобильные игры обычно вообще не нуждаются в использовании атласов.

Для настольных игр с высококачественными текстурами атласы следует применять, только если количество обращений к системе визуализации превышает все разумные возможности аппаратного обеспечения из-за обработки множества текстур с высоким разрешением для обеспечения максимального качества. В настольных играх с низкокачественными текстурами можно вообще отказаться от атласов, так как вызовы системы визуализации не являются в них узким местом.

Файлы мешей и анимаций

Наконец, рассмотрим файлы мешей и анимаций. По сути, эти файлы являются большими массивами данных о вершинах и костях. Существует целый ряд методов, помогающих уменьшить их размеры, сохраняя сходство, если не идентичность, внешнего вида объектов. Также существует несколько способов снижения затрат на отображение больших групп объектов при помощи пакетной обработки.

Уменьшение количества полигонов

Это наиболее очевидный способ улучшения производительности, но не стоит им пренебрегать. В самом деле, поскольку пакетную обработку нельзя применить к компонентам SkinnedMeshRenderer, это единственный доступный способ снижения затрат вычислительных ресурсов центрального и графического процессоров для анимированных объектов.

Сокращение количества полигонов – простой и очевидный способ, обеспечивающий экономию ресурсов центрального процессора и памяти за счет затрат времени художников на чистку мешей.

Настройка сжатия мешей

Настройка сжатия значения для параметра Mesh Compression (Сжатие меша), управляющего сжатием импортируемых мешей: Off (Отключено), Low (Низкое), Medium (Среднее) и High (Высокое). С увеличением степени сжатия Unity отбрасывает все больше и больше вершин из меша, которые сочтет ненужными, чтобы сократить общий размер файла.

Но автоматизированная оптимизация меша – очень сложная задача для решения математическими методами. Даже лучшие алгоритмы, как правило, генерируют большое количество артефактов и нарушений. Изменение значения параметра Mesh Compression можно рассматривать только как быстрое решение задачи сокращения количества полигонов, но результат никогда не будет сопоставим с тем, чего может добиться художник.

Встраиваемые анимации

В некоторых случаях встраиваемые (запекаемые) анимации помогают уменьшить размер файла и занимаемый объем памяти, чем смешанные или наружные анимации. Это зависит от используемого инструмента 3D-моделирования и анимации и общего количества

вершин в меше. Встраиваемые анимации сохраняют позиции каждой вершины в каждом кадре, и если количество полигонов достаточно мало, можно наблюдать значительное сохранение ресурсов при внесении достаточно простых изменений. Кроме того, встраиваемый образец обычно можно настроить в экспортирующем приложении.

Оптимизация мешей движком Unity

Установка флага Optimize Meshes (Оптимизация мешей) в настройках импорта меша обеспечит реорганизацию вершин для ускорения их чтения и, в некоторых случаях, регенерацию низкоуровневого отображения для оптимизации скорости отображения. Сбрасывать его стоит только в том случае, если при профилировании точно выяснится, что его установка каким-то образом приводит к снижению производительности.

Если меш генерируется процедурно, для запуска этого процесса движком Unity можно воспользоваться методом Optimize() компонента фильтра меша. Этот процесс займет некоторое время, поэтому его следует запускать во время инициализации или в другие удобные для приостановки моменты.

Объединение мешей

Объединение мешей позволяет уменьшить количество обращений к системе визуализации в случаях, если эти меши слишком велики для динамической пакетной обработки и не участвуют в статической пакетной обработке. Такое объединение, по существу, эквивалентно статической пакетной обработке, но выполняется вручную, поэтому вряд ли стоит тратить время на то, что может сделать статическая пакетная обработка автоматически.

Данный подход имеет еще один недостаток – он создает совершенно новый файл ресурсов мешей, который требуется разместить в сцене. То есть любые изменения в оригинальных мешах не отразятся на объединенном. Это требует выполнять утомительные действия каждый раз после внесения изменений, поэтому лучше отдать предпочтение статической пакетной обработке.

4.3. Разгон физического движка

Внутреннее устройство физического движка

Технически в Unity имеются два физических движка: NVIDIA PhysX для поддержки 3D-физики и открытый движок Box2D для поддержки 2D-физики. Однако их реализация абстрагирована, и с точки зрения прикладного программного интерфейса Unity оба движка работают практически идентично.

Физические движки и время

Физические движки, как правило, работают в предположении, что итерации происходят в фиксированные моменты времени, и оба физических движка Unity действуют именно так. Выполняя вычисления, они основываются на конкретных значениях времени, независимо от того, сколько времени потребовалось для отображения предыдущего кадра. Такой подход называется обновлением с фиксированныминтервалом времени. По умолчанию считается, что интервал имеет длительность 20 миллисекунд, или 50 обновлений в секунду.

Если между отображением последовательных кадров возникает слишком большая пауза (низкая частота кадров), физическая система обновляется несколько раз перед началом нового отображения.

Цикл фиксированного обновления

Как видите, метод FixedUpdate() вызывается непосредственно перед внутренними операциями физической системы, и эти два действия неразрывно связаны. В начале процесса определяется, прошло ли достаточно времени для начала следующего фиксированного обновления. Результат зависит от времени, прошедшего с момента последнего обновления.

Если прошло достаточно времени, вызываются методы FixedUpdate() всех компонентов, а сразу после этого запускаются все сопрограммы, привязанные к фиксированным обновлениям, затем следуют вызовы обработчиков физических обновлений и триггеров/коллайдеров.

Если с момента последнего фиксированного обновления прошло менее 20 мс, текущее обновление пропускается. В этом случае обработка ввода, игровой логики и отображения должны завершиться до момента, когда Unity проверит необходимость следующего фиксиро-

ванного обновления, и т. д. Этот процесс постоянно повторяется во время выполнения. При таком подходе фиксированные обновления и физическая система получают более высокий приоритет, чем отображение, что ускоряет физическое моделирование при фиксированной частоте кадров.

Максимально допустимая длительность

Важно отметить, что если прошло много времени с момента последнего фиксированного обновления, физическое и фиксированное обновления будут рассчитываться до момента, пока не «догонят» текущее время. Например, если отображение предыдущего кадра заняло 100 мс, физической системе потребуется выполнить пять циклов обновления состояния. При этом перед вызовом метода Update() пять раз будет вызван метод FixedUpdate(), так как интервал фиксированных обновлений по умолчанию составляет 20 мс. Если обработка этих пяти обновлений займет 20 мс и более, необходимо будет вызвать обновление шестой раз!

Следовательно, во время тяжелых вычислений физический движок может не успеть завершить обновление за 20 мс, что добавит еще на 20 мс работы, и так до бесконечности, из-за чего не станет невозможно вывести очередной кадр. Чтобы предотвратить блокировку игры физическим движком, определяется максимальное время, которое ему разрешено выполнять обработку между отображениями. Этот порог называют максимальной допустимой длительностью, и если длительность текущего фиксированного обновления оказывается больше указанного значения, оно будет просто остановлено и отложено до завершения следующего отображения.

Физические обновления и изменения во время выполнения

Когда физическая система приступает к обработке следующего временного шага, она должна рассчитать перемещение всех активных объектов Rigidbody, обнаружить все столкновения и вызвать соответствующие обработчики столкновений. Именно по этой причине документация Unity четко рекомендует вносить изменения в объекты Rigidbody только внутри методов FixedUpdate() и других методов-обработчиков, вызываемых физической системой. Эти методы тесно связаны с частотой обновления физического движка, в отличие от других методов игрового цикла, например метода Update().

Это означает, что обратные вызовы, такие как OnTriggerEnter(), являются безопасным местом для изменения объектов Rigidbody, а такие методы, как Update() и сопрограммы, основанные на времени, таким местом не являются. Пренебрежение этой рекомендацией вызывает неадекватное физическое поведение, так как позволяет несколько раз внести изменения в один и тот же объект до того, как физическая система получит шанс обнаружить их и обработать, что является причиной некоторых особенно запутанных странностей в игровом процессе.

Однако в некоторых играх физическому движку приходится выполнять много вычислений при каждом обновлении, что отрицательно сказывается на частоте кадров отображения, вызывая ее падение из-за повышенной нагрузки на физическую систему. В действительности система отображения продолжает действовать в обычном режиме, но всякий раз, когда подходит время фиксированного обновления, время создания текущего отображения ограничивается, что вызывает внезапные простановки и появление дополнительных визуальных эффектов, вызванных прерыванием работы физической системы из-за превышения максимального выделяемого ей времени. Все вместе это приводит к ухудшению восприятия игры пользователями.

Следовательно, чтобы добиться гладкой и согласованной смены кадров, необходимо высвободить как можно больше времени для отображения, сведя к минимуму время, затрачиваемое физической системой на обработку.

Оптимизация производительности физической системы

Теперь, опираясь на понимание наиболее значительных возможностей физического движка Unity, можно перейти к рассмотрению методов оптимизации производительности физической системы.

Настройка сцены

Во-первых, существует ряд методов, применение которых к сцене позволит улучшить согласованность моделирования физических процессов. Обратите внимание, что эти методы не всегда направлены на уменьшение потребления центрального процессора или памяти, но их применение ведет к снижению вероятности нестабильной работы физического движка.

Масштабирование

Масштаб всех физических объектов игрового мира следует удерживать как можно ближе к соотношению 1:1:1. Это означает, что для ускорения свободного падения по умолчанию -9.81 единица масштаба игрового мира должна соответствовать 1 метру, так как сила тяжести на поверхности земли составляет 9,81 м/с2. Размеры объекта должны отражать подразумеваемый масштаб игрового мира, поскольку слишком большой масштаб приведет к замедлению падений объектов, если это происходит в реальном мире. Обратное также верно: при слишком мелком масштабе объекты будут падать слишком быстро и выглядеть нереалистично.

Настроить подразумеваемый масштаб игрового мира можно, изменив ускорение свободного падения в Edit -> Project Settings ->Physics (или Physics2D). Но имейте в виду, что результаты любых арифметических вычислений с плавающей запятой наиболее точны при значениях, близких к 0, поэтому если значения масштаба некоторых объектов значительно превышают величину (1,1,1), даже если они соответствуют подразумеваемым значениям мирового масштаба, все еще будет велика вероятность неустойчивого поведения физической системы. Итак, в начале работы над проектом следует импортировать и масштабировать наиболее распространенные физические объекты с помощью значений масштаба, близких к (1,1,1), и затем настроить соответствующее значение ускорения свободного падения. Это послужит ориентиром при работе со вновь вводимыми новыми объектами.

Позиционирование

Аналогично расположение всех объектов вблизи точки (0,0,0) приведет к увеличению точности операций с плавающей запятой и улучшению согласованности при моделировании. Космические или спортивные тренажеры пытаются имитировать невероятно большие пространства, но они обычно используют технологию, позволяющую скрытно телепортировать персонаж в игровом мире. Таким образом, при моделировании путешествия все перемещается или делится на отсеки так, чтобы участвующие в физических расчетах значения всегда были близки к нулю. Это гарантирует близость всех объектов к позиции (0,0,0) для повышения точности результатов операций с плавающей запятой при перемещении игрока на большие расстояния.

Другие виды игр также следует освободить от рисков, вносимых неточностью вычислений с плавающей запятой. Даже если работа над

проектом уже ведется, нужно постараться добиться того, чтобы все физические объекты располагались как можно ближе к позиции (0,0,0). Плюсом этого подхода является убыстрение процесса добавления и позиционирования объектов в игровом мире проекта.

Уменьшение отбрасывания лучей и ограничение проверяемого объема

Все методы отбрасывания лучей очень полезны, но отрицательно сказываются на производительности, поэтому используйте их как можно реже. Избегайте регулярного их вызова в методах-обработчиках и сопрограммах и используйте только в коде сценариев для обслуживания ключевых событий.

Если абсолютно необходимы непрерывные линии, лучи или эффекты столкновения областей в сцене, их лучше моделировать с помощью простого триггерного коллайдера вместо непрерывного отбрасывания лучей или проверок перекрытия.

Если такая подмена невозможна и непрерывное отбрасывание лучей действительно необходимо, объем вычислений можно минимизировать с помощью объектов LayerMask.

Например, «ленивое» отбрасывание лучей можно реализовать так:

```
[SerializeField] float _maxRaycastDistance;
void PerformRaycast() {
RaycastHit hitInfo = new RaycastHit();
if (Physics.Raycast(new Ray(transform.position, transform.forward), out hit, _maxRaycastDistance)) {
// обработка результатов отбрасывания лучей
}
}
```

Эта перегруженная версия метода Physics.Raycast() позволит лучу столкнуться только с первым объектом из любого слоя, встретившимся на его пути. Метод Physics.Raycast имеет несколько перегруженных версий, принимающих в качестве аргумента объект LayerMask. Этим можно воспользоваться для отбора объектов, которые нужно проверить при отбрасывании луча, что значительно снизит нагрузку на физический движок:

```
[SerializeField] float _maxRaycastDistance;
[SerializeField] LayerMask _layerMask;
void PerformRaycast() {
```

```
RaycastHit hitInfo = new RaycastHit();
if (Physics.Raycast(new Ray(transform.position, transform. forward),
out hit, _maxRaycastDistance, _layerMask)) {
// обработка результатов отбрасывания лучей
}
}
```

Избегайте сложных меш-коллайдеров

Вот как разные виды коллайдеров располагаются в порядке убывания эффективности: сферы, капсулы, цилиндры, короба, выпуклые меш-коллайдеры и, наконец, вогнутые меш-коллайдеры. Причем четыре основных примитива на порядок эффективнее любых меш-коллайдеров, поскольку математические расчеты для обнаружения и разрешения конфликтов между ними довольно кратки и опти- мизированы. Определение столкновений между меш-коллайдером выпуклой формы и другими коллайдерами обходится недешево, но определение столкновений между меш-коллайдером вогнутой формы и прочими обходится еще дороже.

Главным парадоксом физической и графической систем 3D-приложений является разница в сложности обработки объектов в форме сферы и короба. Для создания идеального сферического меша требуется бесконечное количество полигонов, но в физическом движке задача определения точек контакта и столкновений сферы решается тривиально просто. Напротив, для отображения обычного короба требуется незначительное количество полигонов и затрат, но для определения точек контакта и столкновений требуется намного больше математических расчетов и затрат на обработку. Это привело к тому, что большинство графических и физических систем заполнило игровой мир графическими объектами с малым количеством полигонов в форме сферы. Однако ненормально выглядел бы угловатый объект, катящийся подобно шару.

При работе с физическим движком важно не забывать, что физическое представление объекта не обязательно должно соответствовать его графическому представлению. Удобство заключается в том, что графический меш часто можно представить в виде тела более простой формы с очень похожим физическим поведением. Это избавляет от необходимости использовать чрезмерно сложные меш-коллайдеры.

Такое разделение графического и физического представлений позволяет оптимизировать производительность одной системы без (обязательного) негативного влияния на другую. Если это не оказывает заметного влияния на игровой процесс, можно свободно представлять сложные графические объекты с помощью невидимых простейших физических форм. Если игрок никогда этого не заметит, в этом нет никакого вреда!

Эту задачу можно решить одним из двух способов: либо путем аппроксимации физического поведения тела сложной формы посредством одного (или более) стандартных примитивов, либо с помощью значительно упрощенного меш-коллайдера.

Когда следует использовать физическую систему

Как всегда, лучший способ улучшить производительность компонента – отказаться от излишнего его использования. При анализе особенностей любого движущегося объекта следует воспользоваться моментом, чтобы спросить себя: «А так ли необходимо подключать его к физическому движку?» Если нет, можно поискать возможности замены физической системы чем-то более простым и менее затратным.

Допустим, физическая система используется для выявления попадания игрока в зону поражения, и в игре это определяется достаточно просто: зона поражения находится на определенной высоте. В этом случае можно вообще не использовать физический коллайдер, ограничившись проверкой координаты у игрока.

В качестве другого примера рассмотрим моделирование метеоритного дождя, на первый взгляд кажется, что для реализации потребуется создать множество падающих предметов с физическими компонентами Rigidbody, определять моменты столкновений с землей при помощи коллайдеров и генерировать взрывы в точках ударов. Но поверхность земли может быть совершенно плоской, или у нас имеется доступ к карте высот ландшафта, что обеспечит элементарное выявление столкновений. В этом случае перемещение объекта можно упростить с помощью tween-анимации свойства transform.position без применения любых физических компонентов. В обоих случаях затраты на работу физической системы уменьшатся за счет упрощения и замены ее выполнением кода сценария.

Обратное также возможно. Иногда действия, требующие множества вычислений в сценариях, можно относительно просто реализовать с помощью физической системы. В качестве примера рассмот-

рим систему подсчета собранных объектов. При нажатии игроком клавиши Pick up (Подобрать) следует сравнить положение собираемых объектов с позицией игрока и определить ближайший такой объект. Весь код сценария можно заменить одним вызовом метода Physics.OverlapSphere(), чтобы получить все близлежащие объекты, а затем выбрать самый ближний. При этом значительно уменьшится общее количество объектов, позиции которых нужно сравнивать при каждом нажатии клавиши.

Ваши возможности ограничены только пределами вашей изобретательности. Способность выявлять и устранять избыточные операции физической системы и/или определять дорогостоящие операции, реализованные с помощью сценариев, и заменять их средствами физического моделирования является важным навыком, который пригодится вам для поддержания высокой производительности текущих и будущих проектов игр.

4.4. Динамическая графика

Профилирование проблем отображения

Низкая производительность механизма отображения проявляется по-разному, в зависимости от того, что является ее причиной – центральный или графический процессор. В последнем случае корень проблем может располагаться в нескольких местах графического конвейера. Это затрудняет поиск, но, отыскав и решив проблему, можно надеяться на значительное улучшение, и даже небольшие усовершенствования в подсистеме отображения зачастую приводят к впечатляющим результатам.

Центральный процессор посылает инструкции отображения через графический программный интерфейс, затем через аппаратный драйвер они поступают в графический процессор и накапливаются в его буфере команд. Эти команды поочередно обрабатываются системой параллельной обработки графического процессора, пока буфер не опустеет. Но в данной последовательности гораздо больше нюансов.

Существует несколько подходов к поиску источника проблем отображения графики:

– профилирование графического процессора;

– анализ отдельных кадров с помощью отладчика кадров;

– поиск методом перебора.

Профилирование графического процессора

Поскольку в отображении графики задействованы оба процессора –центральный и графический, для выявления перегруженного компонента следует использовать области CPU Usage и GPU Usage профилировщика.

Профилирование приложений, где основная нагрузка ложится на графический процессор, осуществляется несколько сложнее. Представьте приложение, создающее небольшое количество объектов с большим количеством полигонов, использующее десятки точечных источников света, чрезмерно сложный шейдер, обрабатывающий текстуру, текстуры нормалей, карту высот, карту эмиссий, карту окклюзий и т. д.

Отладка кадров

В Unity 5 появилась новая функция отладки кадров – инструмент, позволяющий определить порядок сборки и отображения сцены отдельными обращениями к системе визуализации. Щелкая по элементам списка, можно наблюдать, как выглядит сцена в тот или иной момент времени. Кроме того, отладка кадров позволяет получить много полезных сведений о выбранном вызове, таких как текущая цель отображения, для чего предназначен вызов и какие параметры использованы.

Если снижение производительности связано с обращениями к системе визуализации, этот инструмент поможет определить, какие вызовы производились, и выяснить, выполнялись ли лишние вызовы, не оказывающие влияния на сцену. Это поможет уменьшить количество вызовов, например, путем удаления ненужных объектов или их пакетной обработки.

Узкие места на этапе предварительной обработки

Нередко используются меши, содержащие массу ненужных данных UV-векторов и векторов нормалей, поэтому их следует тщательно проверить на наличие подобных излишеств. Нужно позволить Unity оптимизировать структуру, что приведет к уменьшению количества промахов кэша при чтении вершин на этапе предварительной обработки.

Уровень детализации

Под термином Уровень детализации (LOD) понимаются широкие возможности динамической замены в зависимости от удаленности относительно камеры. Наиболее часто используется механизм

уровней детализации, основанный на мешах: динамическая замена меша на его все менее и менее детализированные версии по мере удаления от камеры. Еще одним примером может служить замена находящихся вдали анимированных персонажей их версиями с меньшим количеством костей для сокращения затрат на анимацию.

При использовании механизма уровней детализации в сцене размещается несколько объектов, присоединяемых к игровому объекту в качестве дочерних через компонент LODGroup. Этот компонент формирует прямоугольники, ограничивающие объекты, и выбирает, какой объект выводить, опираясь на размер ограничивающего прямоугольника в поле зрения камеры. Если ограничивающий прямоугольник объекта занимает большую часть видимой области, выбираются меши из нижних групп уровней детализации, а если ограничивающий прямоугольник очень мал, выбираются меши из верхних групп. Если меш находится слишком далеко, можно настроить скрытие всех дочерних объектов.

Использование этой функции может значительно увеличить время разработки, поскольку художникам придется сгенерировать версии с малым количеством полигонов, а дизайнерам уровней – создать группы детализации, настроить и проверить их, чтобы убедиться, что переходы не вызывают дрожания при удалении и приближении камеры. Кроме того, при таком подходе увеличивается потребление памяти и ресурсов центрального процессора, так как альтернативные меши хранятся в памяти, а компонент LODGroup должен регулярно проверять, не переместилась ли камера в новое положение, чтобы гарантировать соответствующее изменение уровней детализации.

В настоящее время возможности графических карт достигли такого уровня, что обработка вершин является не самой важной проблемой. Вместе с дополнительными затратами, сопровождающими применение механизма детализации уровней, разработчикам следует избегать предварительной оптимизации и положиться на механизм детализации уровней. Злоупотребление этим механизмом приведет к потерям производительности в других частях приложения и бессмысленной потере времени при разработке. Если не доказано, что именно в этом состоит проблема, это не является проблемой!

Для сцен, отображающих обширные пространства игрового мира при частых перемещениях камеры, следует изначально предусмотреть применение этого метода, поскольку при увеличении расстояния и значительном росте количества видимых объектов произойдет рез-

кое увеличение числа вершин. В сценах, ограниченных помещением, и в сценах, где камера направлена вниз на игровой мир, следует воздержаться от использования уровней детализации. В играх, занимающих промежуточное положение, следует избегать применения этой функции, пока в ней не возникнет острая необходимость. Все зависит от того, сколько вершин будет видно в любой момент времени и как часто изменяется расстояние до камеры.

Отключение скининга графическим процессором

Как уже говорилось, Unity поддерживает выполнение скининга графическим процессором для снижения нагрузки на центральный процессор за счет увеличения нагрузки на графический процессор. Так как скининг является одним из «впечатляюще параллельных» процессов, которые хорошо сочетаются с параллельной архитектурой графического процессора, передача этой задачи ему часто дает хороший выигрыш. Но на выполнение этой задачи будет тратиться драгоценное время, необходимое на определение вершин фрагментов, поэтому отключение обработки скининга графическим процессором является еще одним направлением для исследования, если узкое место обнаружится в этой области. Отключить скининг графическим процессором можно с помощью параметра Edit -> Project Settings ->Player Settings -> Other Settings -> GPU Skinning.

Узкие места на этапе окончательной обработки

Этап окончательной обработки является наиболее интересной частью конвейера графического процессора, так как здесь создается подавляющее большинство графических эффектов. Следовательно, на этом этапе значительно чаще возникают узкие места.

Здесь можно использовать два пути:

- уменьшить разрешение;
- уменьшить качество текстур.

Эти изменения ослабят рабочую нагрузку на двух важных стадиях заключительной части конвейера, увеличив скорость заполнения и пропускную способность памяти соответственно. В настоящее время скорость заполнения является наиболее распространенной причиной падения производительности при отображении графики, поэтому начнем с нее.

Скорость заполнения

Благодаря уменьшению разрешения экрана системе растеризации придется создавать значительно меньше фрагментов и транспонировать их на меньший по размерам холст пикселей. Это позволит увеличить скорость заполнения и ослабит нагрузку на ключевую часть конвейера визуализации. Следовательно, если при уменьшении разрешения экрана резко увеличится производительность, значит, дело в скорости заполнения.

Скорость заполнения охватывает весьма широкий круг понятий, относящихся к скорости, с которой графический процессор способен отображать фрагменты. Но это касается только фрагментов, которые благополучно прошли все проверки на протяжении выполнения заданного шейдера. Фрагмент – это «потенциальный пиксель», и если любая из его проверок окончится неудачей, он немедленно отбрасывается. Это значительно увеличивает производительность, так как конвейер пропускает затратную операцию отображения и переходит к следующему фрагменту.

Производители графических карт обычно указывают в рекламных материалах конкретную скорость заполнения, обычно в гигапикселях в секунду, но это несколько неправильно, так как точнее будет сказать, что это скорость в гигафрагментах в секунду. Однако это чисто академический аргумент. В любом случае, большие значения скорости заполнения говорят о том, что устройство потенциально может пропустить через конвейер больше фрагментов.

Оптимизация шейдеров

Шейдеры способны значительно снизить скорость заполнения в зависимости от их сложности, количества текстур, числа используемых математических функций и т. д. Шейдеры влияют на скорость заполнения не напрямую, а косвенно, из-за того, что при выполнении шейдеров графическому процессору приходится производить вычисления или получать данные из памяти. Параллельная природа графического процессора означает, что любое узкое место в потоке будет ограничивать количество фрагментов, передаваемых последующим потокам, но параллельная обработка обеспечивает ускорение относительно последовательной обработки.

Программирование и оптимизация шейдеров занимают особое место в области разработки игр. Их абстрактный и узкоспециальный характер требуют мышления определенного рода, отличающегося от

того, который требуется для написания кода игрового процесса и для взаимодействия с движком. В шейдерах часто используются необычные математические приемы и обходные пути для обработки данных, например предварительная компиляция значений из файлов текстур. Из-за этого и из-за важности их оптимизации шейдеры, как правило, очень трудно читать и модернизировать.

Поэтому многие разработчики пользуются предустановленными шейдерами или инструментами визуальной разработки шейдеров из Asset Store, такими как Shader Forge или Shader Sandwich. Они упрощают создание начального кода шейдера, который, впрочем, может оказаться не самым эффективным. При использовании предварительно подготовленных шейдеров или созданных с помощью инструментов стоит попытаться оптимизировать их с помощью нескольких проверенных и надежных методов.

Шейдеры для мобильных платформ

Мобильные шейдеры, встроенные в Unity, не имеют каких-либо конкретных ограничений, обязывающих их использовать только на мобильных устройствах. Они просто оптимизированы в расчете на минимальные ресурс).

Настольные приложения могут использовать эти шейдеры, но их применение, как правило, влечет потерю качества графики. Дело здесь заключается только в приемлемости снижения качества графики. Поэтому попробуйте поэкспериментировать с мобильными эквивалентами обычных шейдеров, чтобы определить, насколько хорошо они соответствуют вашей игре.

Шейдеры поверхностей

При использовании шейдеров поверхностей, которые упрощают программирование шейдеров, движок Unity берет на себя преобразование кода шейдера поверхностей и избавляет от необходимости задумываться об оптимизациях, которые были рассмотрены выше. Однако данный подход дает возможность изменять некоторые параметры для снижения точности в целях упрощения математических расчетов. Поверхностные шейдеры в общем случае достаточно эффективны, но при индивидуальном подходе можно добиться большей их оптимизации.

Атрибут approxview оказывает влияние на точность аппроксимации направления взгляда, позволяя сэкономить на этой очень за-

тратной операции. Атрибут halfasview регулирует точность представления векторов, но следует остерегаться его влияния на математические операции с данными, имеющими разные типы точности. Атрибут nofor- wardadd уменьшает количество источников направленного света до одного, сокращая количество вызовов системы визуализации, поскольку шейдер будет выполнять отображение за один проход, но уменьшая при этом сложность освещения. И наконец, атрибут noambient отключает в учет рассеянного освещения, исключая дополнительные математические операции, которые в данном случае не нужны.

Реализуйте уровни детализации на основе шейдеров

Имеется возможность заставить Unity использовать для отображения удаленных объектов более простые шейдеры, что эффективно сохранит требуемую скорость заполнения, особенно при развертывании игры на различных платформах или поддержке широкого спектра аппаратных возможностей. В шейдере можно использовать ключевое слово LOD, чтобы задать экранный размер, поддерживаемый шейдером. Если текущий уровень детализации не соответствует этому значению, будет выполнен переход к следующему шейдеру и т. д., пока не будет найден шейдер, поддерживающий заданный размер. Значение LOD для шейдера можно также задать во время выполнения, используя его свойство maximumLOD.

Пропускная способность памяти

Пропускная способность памяти – еще один важный компонент эта- па окончательной обработки и потенциальный источник проблем. Пропускная способность памяти расходуется всякий раз, когда текстура извлекается из основной видеопамяти графического процессора. Графический процессор содержит несколько ядер, имеющих доступ к той же области видеопамяти, но каждый из них имеет намного меньший по размеру локальный текстурный кэш, хранящий текущую текстуру или последнюю, с которой работал графический процессор.

Всякий раз, когда фрагментный шейдер производит выборку из текстуры, которая уже находится в локальном текстурном кэше ядра, она выполняется молниеносно и практически не заметно. Но если выполняется выборка из текстуры, отсутствующей в текстурном кэше, перед ее выполнением текстура должна быть извлечена из видеопамяти. Сама передача вызывает расходование значительной доли пропу-

скной способности памяти, точнее, эта доля равна полному размеру файла текстуры, хранящемуся в видеопамяти. Именно по этой причине, если узким местом производительности является пропускная способность памяти, снижение качества текстур при тестировании методом перебора приведет к резкому повышению производительности. Уменьшение размеров текстур увеличивает пропускную способность памяти графического процессора, позволяя ему извлекать необходимые текстуры намного быстрее. Глобально уменьшить качество текстур можно с помощью параметра Edit ->Project Settings -> Quality -> Texture Quality, присвоив ему значение Half Res (Одна вторая), Quarter Res (Одна четвертая) или Eighth Res (Одна восьмая).

Когда узким местом является пропускная способность памяти, графический процессор будет извлекать необходимые текстуры снова и снова, что приведет к задержке всего процесса, поскольку кэш текстур будет ждать появления необходимого фрагмента перед его обработкой. Графический процессор не сможет записать данные в буфер кадров к моменту его отображения на экране, заблокирует весь процесс и приведет к уменьшению частоты кадров.

Существует несколько решений проблемы недостаточной пропускной способности памяти.

Уменьшайте размеры текстур

Этот подход прост, надежен и применим практически всегда. Обычно уменьшение разрешения текстур или использование типов данных с меньшей точностью приводит к ухудшению качества изображения, но иногда переход на использование 16-битных текстур не вызывает заметного ухудшения.

Механизм mip-текстурирования – еще один отличный способ уменьшить размеры текстур, которые взад/вперед передаются между видеопамятью и текстурным кэшем. Обратите внимание, что представление сцены имеет режим заливки Mipmaps, в котором текстуры, присутствующие в сцене, окрашиваются в синий или красный цвет в зависимости от соответствия текущего масштаба текстуры положению и ориентации камеры. Это поможет определить текстуры, являющиеся кандидатами для дальнейшей оптимизации.

Организуйте ресурсы для уменьшения перекачки текстур

Этот подход основывается на пакетной обработке и атласах. Существует ли возможность пакетной обработки наиболее объемных

файлов текстур? Если это так, можно избавить графический процессор от необходимости перегонять одни и те же файлы текстуры снова и снова в течение одного кадра. Как крайнее средство можно поискать способ удалить некоторые текстуры из проекта и повторно использовать похожие файлы. Например, если имеются проблемы со скоростью заполнения, можно с помощью шейдеров фрагментов собрать в кучу файлы текстур, применяемых в игре с различными вариациями цветов.

Предварительная загрузка текстуры

Несмотря на то что это не относится напрямую к производительности, стоит отметить, что пустая текстура, используемая во время асинхронной загрузки, может резко снизить качество игры. Поэтому желательно иметь возможность управлять загрузкой текстур с диска и принудительно загружать текстуры в основную память, а затем и в видеопамять, прежде чем они действительно понадобятся.

Для этого часто создается скрытый игровой объект с нужной текстурой и помещается где-то в сцене на пути игрока к месту, где она потребуется. Как только текстурированный объект становится кандидатом на отображение, начнется процесс копирования его данных в видеопамять. Это несколько громоздкое, но в то же время простое в реализации решение, достаточно эффективное в большинстве случаев.

Можно также управлять этим процессом из кода сценария, изменяя текстуру скрытого материала:

```
GetComponent<Renderer>().material.texture = textureToPreload;
```

Перезагрузка текстур

В редких случаях, когда загружается слишком много текстурных данных и необходимые текстуры отсутствуют в видеопамяти в данный момент, графическому процессору придется извлечь их из основной памяти и затереть прежде загруженные текстуры, чтобы освободить место. Этот процесс усугубляется с течением времени из-за фрагментации памяти, и возникает риск, что затертые текстуры придется извлекать снова и снова на протяжении одного и того же кадра.

Эта проблема не стоит так остро в современных консолях PS4, Xbox One и WiiU, так как у них общее пространство памяти для центрального и графического процессоров. Их конструкция оптимизирована на аппаратном уровне и учитывает тот факт, что на устройстве

всегда запускается одно приложение и почти всегда отображается 3D-графика. Однако все прочие платформы должны разделять время и память между несколькими приложениями и иметь возможность работать без графического процессора. Поэтому на них имеется отдельная память для центрального и для графического процессоров, и необходимо гарантировать, что общий объем используемых текстур ни в какой момент времени не превысит объема доступной видеопамяти на целевом оборудовании.

Освещение и затенение

Unity предлагает два стиля динамического освещения, а также внедрение световых эффектов с помощью карт освещения. Кроме того, предоставляется несколько способов создания теней с различными уровнями сложности и затратами на обработку во время выполнения. Эти два стиля не только имеют массу вариаций, но и множество коварных ловушек, в которые легко попасть, если не проявлять должную осторожность.

Начнем с двух главных режимов освещения. Их настройки находятся в разделе Edit -> Project Settings -> Player -> Other Settings ->Rendering и зависят от платформы.

Непосредственное отображение

Непосредственное отображение является классической формой отображения источников света в сцене. Каждый объект обычно отображается за несколько проходов одного и того же шейдера. Число проходов определяется количеством, расположением и яркостью источников света. Unity пытается определить приоритетный источник направленного света для объекта и получить его первичное отображение в «базовом проходе». Затем выделяется до четырех самых мощных ближайших точечных источников света, и тот же объект отображается повторно за нескольких прогонов одного и того же фрагментного шейдера.

Отложенное затенение

Отложенное затенение, или отложенное отображение, как его иногда называют, доступно только на графических процессорах, поддерживающих, по крайней мере, Shader Model 3.0. Этот метод был весьма популярен некоторое время, но это не привело к полному отказу от метода непосредственной визуализации из-за применения с оговор-

ками и ограниченной поддержки мобильными устройствами. Сглаживание, прозрачность и анимированные персонажи, получающие тени, не поддерживаются при использовании одного только метода отложенного затенения, и его следует использовать вместе с методом непосредственного отображения, в качестве запасного варианта.

Обработка освещения в вершинном шейдере (устаревший способ)

С технической точки зрения существует более двух методов реализации освещения. Unity позволяет также использовать и пару устаревших систем, но только одна из них имеет актуальное значение, а именно обработка освещения в вершинном шейдере. Это значительно упрощенный способ расчета освещения, поскольку выполняется только для вершин, а не для пикселей. Другими словами, в цвет падающего освещения окрашиваются целые поверхности, а не отдельные пиксели.

Эта устаревшая технология почти вышла из употребления, так как отсутствие надлежащего освещения и теней затрудняет визуализацию глубины. Она может применяться для 2D-игр, в которых не планируется использовать тени, карты нормалей и различные другие особенности освещения, но эта технология все еще доступна.

Обработка теней в реальном времени

Мягкие тени весьма затратны, жесткие тени менее затратны, но нет теней, свободных от затрат. Все настройки Shadow Resolution, Shadow Projection, Shadow Distance и Shadow Cascades, определяющие поведение и сложность теней, можно найти в разделе Edit -> Project Settings -> Quality -> Shadows.

Оптимизация освещения

Кратко рассмотрев все технологии освещения, перейдем к описанию методов снижения затрат, связанных с освещением.

Использование соответствующего режима заливки

Проведите тестирование двух основных методов отображения, чтобы определить, какой лучше подходит для вашего проекта. Отложенное затенение часто используется как резервный метод, применяемый, когда непосредственное отображение пагубно влияет на про-

изводительность, но вообще многое зависит от результатов выявления узких мест, так как иногда трудно выяснить разницу между ними.

Использование масок выбраковки

Свойство Culling Mask компонента источника освещения – это маска слоев, используемая для определения объектов, которые будут затронуты данным источником света. Оно помогает эффективно сократить затраты на освещение, так как взаимодействие со слоями имеет здесь тот же смысл, что при использовании слоев для оптимизации физической системы. Объекты могут входить только в один слой, и сокращение затрат на физическое моделирование и на освещение может стать плюсом, но в случае конфликтов такой подход перестает быть идеальным.

Использование внедренных карт освещения

Внедрение освещения и затенения в сцену требует гораздо меньше затрат, чем создание их во время выполнения. Оборотной стороной этого приема является увеличение размера приложения, потребления и падения пропускной способности памяти. В конечном счете, если световые эффекты реализуются только с помощью устаревшего способа расчета освещения в вершинном шейдере или имеется единственный направленный источник освещения, внедрение карты освещения приведет к значительному снижению затрат.

Оптимизация теней

На обработку теней приходятся больше всего обращений к системе визуализации и наибольшая доля скорости заполнения, но объем передаваемых данных вершин и выбор параметра Shadow Projection будет оказывать влияние на способность генерировать источники и приемники теней в начальной части конвейера. Первым делом следует попытаться сократить количество вершин для обработки в начальной части конвейера, и эффект от внесения нужных для этого изменений будет многократно усилен.

Обращения к системе визуализации затрачиваются на затенение при отображении видимых объектов в отдельный буфер – источника или приемника тени либо обоих. Каждый объект, отображаемый в эту карту, требует дополнительных обращений к системе визуализации, что делает тени причиной многократно возрастающих затрат производительности, поэтому пользователю часто предлагаются настройки

для понижения качества, позволяющие уменьшить этот эффект или даже исключить его полностью при использовании слабого оборудования.

Параметр Shadow Distance является глобальным множителем, влияющим на отображение теней во время выполнения. Чем меньше теней, тем выше производительность всего процесса отображения.

Большие значения параметров Shadow Resolution и Shadow Cascades увеличивают расходование пропускной способности памяти и скорости заполнения. Эти параметры могут помочь исключить появление артефактов при отображении теней, но за счет увеличения размера карты теней, которую нужно пересылать и требующей холст соответствующего размера.

Стоит отметить, что мягкие тени потребляют памяти и ресурсов центрального процессора не больше, чем жесткие, поскольку единственное их различие заключается в применении более сложных шейдеров. Это означает, что в приложениях с достаточно высокой скоростью заполнения можно позволить себе улучшенную графическую четкость мягких теней.

Оптимизация графики для мобильных устройств

Способность движка Unity развертываться на мобильных устройствах обусловила рост его популярности среди любителей и команд разработчиков малого и среднего размера.

Обратите внимание, что любой из подходов, описанных ниже, может устареть в ближайшее время, если какие-то из них уже не устарели. Рынок мобильных устройств развивается невероятно быстро, и приведенные ниже методы просто отражают накопленный за последние пять лет опыт. Иногда следует проверять предложенные подходы, чтобы узнать, действуют ли все еще описанные ограничения для вновь появляющихся мобильных устройств.

Минимизация обращений к системе визуализации

Чаще всего узким местом мобильных приложений являются обращения к системе визуализации, а не скорость заполнения. Но не следует игнорировать и проблемы, связанные со скоростью заполнения. В любом мобильном приложении приемлемого качества необходимо с самого начала реализовать объединение мешей, пакетную обработку и атласы. Желательно также применить отложенное отображение, поскольку оно хорошо сочетается с другими особенностями

мобильных приложений, например отказом от использования прозрачности и большого числа анимированных персонажей.

Минимизация количества материалов

Эта проблема тесно связана с идеями пакетной обработки и атласами. Чем меньше материалов, тем меньше обращений к системе визуализации. Этот подход помогает также решать проблемы низкой пропускной способности памяти, характерные для мобильных устройств.

Уменьшение размеров текстур и количества материалов

Большинство мобильных устройств имеют намного меньший текстурный кэш, чем более мощные настольные устройства. Например, iPhone 3G поддерживает текстуры размером не более 1024 x 1024, что обусловлено ограничениями библиотеки OpenGLES1.1 и применением простой технологии отображения вершин. Устройства iPhone 3GS, iPhone 4 и iPad используют библиотеку OpenGLES 2.0, поддерживающую текстуры размеров до 2048 x 2048. Более поздние поколения поддерживают текстуры размером до 4096 x 4096. Тщательно проверьте характеристики аппаратных устройств, чтобы удостовериться, что они поддерживают размеры текстур, которые предполагается использовать. Имейте в виду, что новейшие устройства никогда не являются самым распространенными. Чтобы игра могла завоевать более широкую аудиторию, она должна поддерживать и более слабые устройства.

Литература

1. C# 5.0 и платформа .NET 4.5 для профессионалов / К. Нейгел [и др.] ; пер. с англ. Ю. Н. Артеменко.– М. : Диалектика, 2014. – 1435 с.
2. Батфилд-Эддисон, П. Unity для разработчика. Мобильные мультиплатформенные игры / П. Батфилд-Эддисон, Дж. Мэннинг. – СПб. : Питер, 2019. – 352 с.
3. Бонд, Д. Unity и C#. Геймдев от идеи до реализации / Д. Бонд ; предисл. Р. Лемарчанда. – 2-е изд.. – СПБ. : Питер, 2019. – 925 с.
4. Вольф, Д. OpenGL 4. Язык шейдеров. Книга рецептов / Д. Вольф. – М. : ДМК-Пресс, 2015. – 368 с.
5. Гейг, М. Разработка игр на Unity 2018 за 24 часа / М. Гейг. – М. : Эксмо, 2018. – 278 с.
6. Дикинсон, К. Оптимизация игр в Unity 5. Советы и методы оптимизации игровых приложений / К. Дикинсон. – М. : ДМК-Пресс, 2017. – 306 с.
7. Паласиос, Х. Unity 5.x. Программирование искусственного интеллекта в играх / Х. Паласиос. – М. : ДМК-Пресс, 2016. – 272 с.
8. Торн, А. Основы анимации в Unity / А. Торн. – М. : ДМК-Пресс, 2016. – 176 с.
9. Хокинг, Д. Unity в действии : мультиплатформенная разработка на C# / Д. Хокинг. – 2-е межд. изд. – СПб. : Питер, 2019. – 351 с.

СОДЕРЖАНИЕ

Введение 3
Введение в игровые платформы 4
1. Создание 2D игры 10
1.1. Создание простой аркадной игры 10
2.1. Создание игры, учитывающей законы физики 28
2.3. Создание игры-шутера 50
1.4. Создание карточной игры 72
1.5. Создание простой игры в слова 98
1.6. Создание приключенческой игры 121
2. Создание 3D игры 161
2.1. Создание 3D-ролика 161
2.2. Добавляем в игру врагов и снаряды 167
2.3. Работа с графикой 178
2.4. Двумерный GUI для трехмерной игры 182
2.5. Игра от третьего лица: перемещения и анимация игрока 189
2.6. Интерактивные устройства и элементы 197
2.7. Звуковые эффекты и музыка 202
2.8. Объединение фрагментов в готовую игру 209
3. Мобильные и мультиплатформенные игры 216
3.1. Создание двумерной игры 216
3.2. Создание трехмерной игры 234
3.3. Дополнительные возможности создания игр 250
4. Оптимизация игр в Unity 263
4.1. Приемы разработки сценариев 263
4.2. Пакетная обработка. Аудио и видео 277
4.3. Разгон физического движка 294
4.4. Динамическая графика 301
Литература 315

ИГРОВЫЕ ПЛАТФОРМЫ

Учебно-методическое пособие для студентов специальности 1-40 05 01-12 «Информационные системы и технологии (в игровой индустрии)» дневной формы обучения

Составитель **Комракова** Евгения Владимировна

Подписано к размещению в электронную библиотеку
ГГТУ им. П. О. Сухого в качестве электронного
учебно-методического документа 08.04.22.

Рег. № 10Е.
http://www.gstu.by