

Федеральное государственное бюджетное образовательное учреждение высшего образования

«МОСКОВСКИЙ АВТОМОБИЛЬНО-ДОРОЖНЫЙ

ГОСУДАРСТВЕННЫЙ ТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ (МАДИ)»

Кафедра «Высшая математика»

КУРСОВОЙ ПРОЕКТ

по дисциплине «Компьютерная графика»

На тему: «Разработка 3D объектов в Unity»

**Выполнил:**

Учебная группа: 3бПМ

ФИО: Моргацкая А.А.

Подпись \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

**Руководитель курсового проекта:**

Должность: ассистент

ФИО: Кутейников И.А.

Курсовой проект защищен

с оценкой «\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_»

Подпись \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

«\_\_\_\_\_» \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ 2020 г.

Москва 2020 г.

**Содержание**

[Введение 3](#_Toc43390181)

[1.1. Подготовка к разработке объектов на Unity3D 5](#_Toc43390182)

[1.2. Процесс разработки объектов на платформе Unity3D 5](#_Toc43390183)

[2. Практическая часть 5](#_Toc43390184)

[3. Разработка игровой зоны 19](#_Toc43390185)

[Заключение 22](#_Toc43390186)

[Список литературы 23](#_Toc43390187)

[Код программы 24](#_Toc43390188)

# **Введение**

Unity — межплатформенная среда разработки компьютерных игр, разработанная американской компанией Unity Technologies. Unity позволяет создавать приложения, работающие на более чем 25 различных платформах, включающих персональные компьютеры, игровые консоли, мобильные устройства, интернет-приложения и другие.

Основными преимуществами Unity являются наличие визуальной среды разработки, межплатформенной поддержки и модульной системы компонентов.

Редактор Unity имеет простой Drag&Drop интерфейс, который легко настраивать, состоящий из различных окон, благодаря чему можно производить отладку игры прямо в редакторе. Движок использует для написания скриптов C#.

Проект в Unity делится на сцены (уровни) — отдельные файлы, содержащие свои игровые миры со своим набором объектов, сценариев, и настроек. Сцены могут содержать в себе как, собственно, объекты (модели), так и пустые игровые объекты — объекты, которые не имеют модели («пустышки»). Объекты, в свою очередь содержат наборы компонентов, с которыми и взаимодействуют скрипты. Также у объектов есть название (в Unity допускается наличие двух и более объектов с одинаковыми названиями), может быть тег (метка) и слой, на котором он должен отображаться. Так, у любого объекта на сцене обязательно присутствует компонент Transform — он хранит в себе координаты местоположения, поворота и размеров объекта по всем трём осям. У объектов с видимой геометрией также по умолчанию присутствует компонент Mesh Renderer, делающий модель объекта видимой.

Также Unity поддерживает физику твёрдых тел и ткани. В редакторе имеется система наследования объектов; дочерние объекты будут повторять все изменения позиции, поворота и масштаба родительского объекта. Скрипты в редакторе прикрепляются к объектам в виде отдельных компонентов.

При импорте текстуры в Unity можно сгенерировать alpha-канал, mip-уровни, normal-map, light-map, карту отражений, однако непосредственно на модель текстуру прикрепить нельзя — будет создан материал, которому будет назначен шейдер, и затем материал прикрепится к модели. Редактор Unity поддерживает написание и редактирование шейдеров. Редактор Unity имеет компонент для создания анимации, но также анимацию можно создать предварительно в 3D-редакторе и импортировать вместе с моделью, а затем разбить на файлы.

Целью работы является разработка 3D объектов на Unity.

Тема является актуальной в связи с высоким ростом игровой индустрии в наше время. На рынке компьютерных технологий существуют как крупные компании (EA, Valve, Ubisoft), так и отдельные инди-разработчики. В большинстве случаев крупные компании используют собственные движки для разработки проектов, а Инди-разработчики же предпочитают готовые, среди которых есть и бесплатные (Unity, Unreal Engine и т.д.).

В связи с поставленной целью в работе решаются следующие задачи:

* Изучить интерфейс Unity 3D;
* Создать физический объект и задать ему движение ;
* Добавить графику, воспользовавшись Unity Assets Store .
  1. **Подготовка к разработке объектов на Unity3D**

Для того чтобы начать работать над проектом в Unity, не нужно быть профессиональным разработчиком видеоигр, но необходимо тщательно продумать и обговорить все нюансы предстоящей работы. Даже имея под рукой такой гибкий инструмент как Unity, все равно нужно представлять, что должно получиться в итоге.

* 1. **Процесс разработки объектов на платформе Unity3D**

Перед началом работы необходимо подготовить рабочее поле программы, выставить необходимые настройки и изучить ее интерфейс. Подготовка рабочего поля. Выставляем настройки рабочей области, для этого нужно создать новый проект, выбрать шаблон, место хранения, добавить дополнительные Asset в проект, если это необходимо. Далее, создав проект, открывается пустая сцена, в которой будет обязательно камера, сохраняем результат. После этого начинаем с ней работать: импортируем объекты сцены, добавляем им свойства, элементы сцены. Теперь мы можем приступить к созданию физического движущегося объекта.

Описанных выше свойств недостаточно для создания полноценной видеоигры. Необходимо описать логику поведения объектов сцены и их взаимодействие между собой. Для этого в Unity используется два языка программирования JavaScript и C#. В этой работе будет использован язык C#. В качестве редактора кода будет выступать Visual Studio Code.

1. **Практическая часть**

*Сфера*

Начнем с нового 3D проекта по умолчанию. Используем линейное цветовое пространство, которое можно настроить в настройках проекта через Edit / Project Settings / Player / Other Settings.

Сцена *SampleScene* по умолчанию имеет основную камеру и направленный свет, который мы сохраним. Создадим плоскость, представляющую землю.

Пока что мы ограничиваемся 2D перемещением по земле, поэтому расположим камеру над плоскостью, глядя сверху-вниз, чтобы получить хороший обзор игровой зоны в игровом окне. Также установим его режим проекции (*Projectionmode)* на Орфографический. Это избавляет от перспективы, позволяя нам видеть движение 2D без искажений (*Orthographic*).

Создаем 3D объект – сферу (Spher) – и задаём ему движение при помощи C# скрипта «Moving». Мы начнем с простого, перемещая сферу вокруг маленького плоского прямоугольника. Как только мы получим хороший контроль над этим процессом, мы сможем усложнить его в будущем. Дальше мы будем работать с физикой «сферы» и созданием платформы (рис. 1).

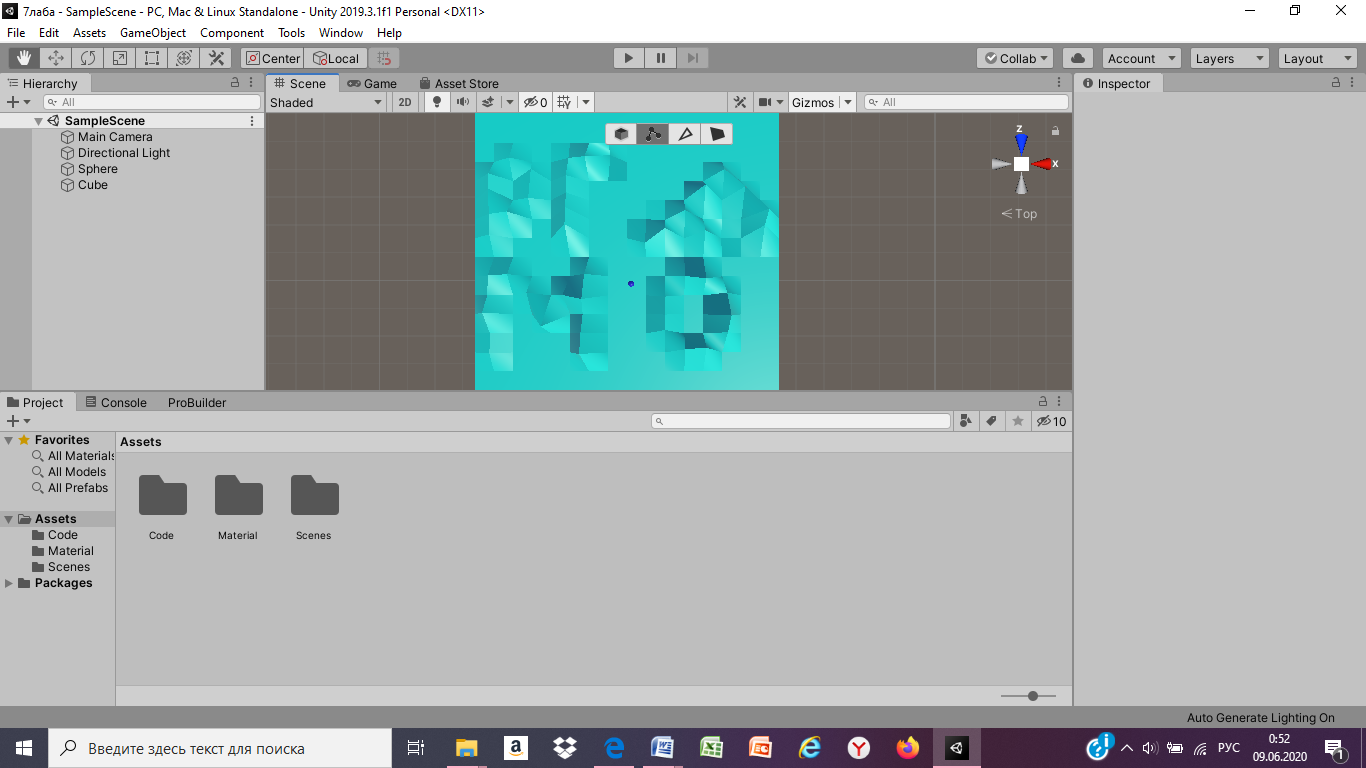


Рис. 1.

*Считывание команд игрока*

Чтобы переместить сферу, мы должны прочитать команды ввода игрока. Мы делаем это в методе обновления MovingSphere. Ввод игрока производится в 2D, поэтому мы можем сохранить его в переменной Vector2. Сначала мы установим оба компонента X и Y на ноль, затем используем их для позиционирования сферы в плоскости XZ. Таким образом, компонент Y ввода становится компонентом позиции по Z. Положение Y остается нулевым.

Самый простой способ получить направленный ввод от игрока, вызвав  
Input.GetAxis с именем оси. Unity имеет как горизонтальную (Horizontal) , так и вертикальную (Vertical) ось ввода данных, определенную по умолчанию, которую вы можете проверить в разделе «Ввод» (Input) в настройках проекта. Мы будем использовать горизонтальное значение для X и вертикальное значение для Y.

Настройки по умолчанию связывают эти оси со стрелками и клавишами WASD. Входные значения также подправлены, поэтому клавиши ведут себя как контроллер. Вы можете настроить эти параметры по своему усмотрению.

Обе оси также имеют второе определение, которое связывает их с вводом джойстика или левым стиком. Это метод обеспечивает более плавный ввод, но мы будем использовать клавиши для всех анимаций.

*Изменение скорости*

Наш максимальный входной вектор имеет величину 1, которая представляет скорость одного метра в секунду, которая равна 3,6 километра в час, то есть около 2,24 мили в час. Это не очень быстро.

Мы можем увеличить максимальную скорость, масштабируя входной вектор. Коэффициент масштабирования представляет максимальную скорость, которая является скоростью без направления. Добавим для него поле maxSpeed с атрибутом SerializeField со значением по умолчанию 10 и присвойте ему атрибут Rangeв интервале 1–100.

Умножим входной вектор и максимальную скорость, чтобы найти желаемую скорость.

*Желаемая скорость*

Управление ускорением вместо скорости приводит к намного более плавному движению, но также ослабляет наш контроль над сферой.

Мы можем контролируя скорость сферы и применяя ускорение к фактической скорости, пока она не будет соответствовать желаемой. Затем мы можем настроить отзывчивость сферы, регулируя ее максимальное ускорение.

В *Update* мы теперь используем входной вектор для определения желаемой скорости и больше не регулируем скорость по-старому.

Теперь мы сначала находим максимальное изменение скорости, умножая максимальное ускорение на t. Таким образом мы можем изменить скорость.

Эти изменения могут вызвать выброс, который мы можем предотвратить, взяв минимум увеличенного и желаемого значения. Есть метод Mathf.*Min*, который мы можем использовать здесь.

В качестве альтернативы, скорость может быть больше, чем хотелось бы. В этом случае мы вычитаем максимальное изменение и берем максимум этого и желаемого значения через Mathf.*Max*.

Мы также можем сделать все это с помощью удобного метода Mathf.*MoveTowards*, передав ему текущие и желаемые значения плюс максимально допустимое изменение.

Теперь мы можем настроить максимальное ускорение для достижения желаемого баланса между плавным движением и отзывчивостью.

*Упругость*

Нам не нужно сохранять всю скорость при движении задним ходом. Некоторые предметы обладают большей упругостью, чем другие. Итак, сделаем упругость настраиваемой, добавив поле восстановления, по умолчанию равное 0,5 с диапазоном 0–1. Это позволит нам сделать нашу сферу совершенно максимально упругой или вообще не отскакивать, или что-то среднее.

*Твердое тело*

Мы должны поддерживать взаимодействие сферы с произвольной геометрией. Вместо того, чтобы реализовывать это самостоятельно, мы будем использовать существующий физический движок Unity, которым является NVIDIA PhysX.

Существует два основных способа управления персонажем в сочетании с физическим движком. Во-первых, это подход с твердым телом (rigidbody), который заключается в том, чтобы персонаж вел себя как обычный физический объект, применяя силы или изменяя его скорость. Вторым является кинематический подход, который заключается в прямом управлении, когда физический движок запрашивается только для выполнения обнаружения столкновений.

*Компонент твердого тела*

Мы будем использовать первый подход для управления сферой, что означает, что мы должны добавить к ней компонент Rigidbody. Мы можем использовать конфигурацию по умолчанию для твердого тела.

Добавление компонента достаточно, чтобы превратить нашу сферу в физический объект, при условии, что у него все еще есть свой компонент SphereCollider. Отныне мы переходим к физическому движку для столкновений.

С нашими собственными ограничениями сфера снова может свободно перемещаться за края плоскости, после чего она падает из-за силы тяжести. Это происходит потому, что мы никогда не переопределяем положение сферы по оси Y.

Нам больше не нужны параметры конфигурации для разрешенной области. В нашем параметре упругости больше нет необходимости.

Если мы все еще хотим ограничить сферу, чтобы она оставалась на плоскости, мы можем сделать это, добавив другие объекты, чтобы заблокировать ее путь.

При попытке переместиться в угол сфера начинает дрожать, поскольку физический движок и наш собственный код борются за позиционирование сферы. Мы перемещаем его в стену, затем PhysX разрешает столкновение, выталкивая его обратно. Если мы перестанем вталкивать его в стену, то PhysX будет поддерживать движение сферы за счет импульса.

*Контролирование скорости твердого тела*

Если мы хотим использовать физический движок, мы должны позволить ему контролировать положение нашей сферы. Непосредственная корректировка положения была бы телепортацией, а это не то, чего мы хотим. Вместо этого мы должны косвенно контролировать сферу, применяя силы к ней или регулируя ее скорость.

У нас уже есть этот косвенный контроль над положением, поскольку мы меняем скорость. Все, что нам нужно сделать, это изменить наш код, чтобы он переопределял скорость компонента Rigidbody вместо того, чтобы самому регулировать положение. Для этого нам необходим доступ к компоненту, поэтому следите за ним через поле *body*, которое инициализируется методом *Awake*.

Но физические столкновения и тому подобное также влияют на скорость, поэтому извлеките ее из тела, прежде чем настраивать ее в соответствии с желаемой скоростью.

*Движение без трения*

Теперь мы отрегулируем скорость сферы, которую PhysX использует для ее перемещения. Когда разрешаются столкновения, они могут регулировать скорость, которую мы затем настраиваем снова и так далее. Результирующее движение похоже на то, что было раньше, хотя сфера более медленная и не достигает максимальной скорости. Это потому, что PhysX применяет трение. Хотя так более реалистично, это усложняет настройку нашей сферы, поэтому давайте устраним трения, а также упругость. Это делается путем создания нового физического материала через *Asset / Create / Physic Material*  и установки всех значений на ноль и режимов объединения (*Combine*) на минимум (*Minimum*).

Теперь сфера больше не подвержена никаким трениям и отскокам.

Может показаться, что сфера все еще слегка подпрыгивает при столкновении со стеной. Это происходит потому, что PhysX не предотвращает столкновения, вместо этого он обнаруживает их после того, как они произошли, а затем перемещает твердые тела, чтобы они больше не пересекались. В случае быстрого движения это может занять более одного шага физического моделирования, поэтому мы можем видеть, как эта зависимость происходит.

Если движение действительно быстрое, то сфера может в конечном итоге пройти сквозь стену или оказаться зависимой от другой стороны, что более вероятно при наличии тонкой стены.

Кроме того, сфера теперь скользит, а не катится, поэтому мы могли бы также остановить ее вращение во всех измерениях, что мы можем сделать с помощью флажков Ограничения (*Constrains*) компонента Rigidbody.

*Корректировка метода Update*

Физический движок использует фиксированный временной шаг, независимо от частоты кадров. Хотя мы уже дали контроль над сферой PhysX, мы все же влияем на ее скорость. Для достижения наилучших результатов мы должны отрегулировать скорость с фиксированным шагом по времени. Мы делаем это, разбивая наш метод обновления (*Update*) на две части. Часть, в которой мы проверяем ввод и устанавливаем желаемую скорость, может остаться в *Update*, а корректировка скорости должна перейти к новому методу *FixedUpdate*. Чтобы сделать это, мы должны хранить желаемую скорость в поле.

Метод *FixedUpdate* вызывается в начале каждого шага моделирования физики. Как часто это происходит, зависит от временного шага, который по умолчанию составляет 0,02 (пятьдесят раз в секунду)

В зависимости от частоты кадров *FixedUpdate* может вызываться ноль, один или несколько раз за вызов *Update*. Каждый кадр представляет собой последовательность вызовов *FixedUpdate*, затем вызывается *Update*, а затем отображается кадр. Это может сделать дискретную природу физического моделирования очевидной, когда временной шаг физики слишком велик по сравнению со временем кадра.

Мы можем решить это, уменьшив фиксированный временной шаг или включив режим интерполяции (*Interpolate*) для Rigidbody. Установка значения *Interpolate* создает линейную интерполяцию между последней и текущей позицией, поэтому сфера будет немного отставать от своего фактического положения в соответствии с PhysX. Другой вариант - использовать экстраполяцию (*Extrapolate*), которая интерполирует предполагаемое положение в соответствии со скоростью сферы, что действительно приемлемо только для объектов, которые в основном имеют постоянную скорость.

*Прыжки*

Поскольку наша сфера теперь может перемещаться в мире трехмерной физики, дадим ей возможность прыгать.

*Прыжки по команде*

Мы можем использовать *Input.GetButtonDown (“Jump”)*, чтобы определить, нажал ли игрок кнопку прыжка для этого кадра, которая по умолчанию является клавишей пробела. Мы делаем это в *Update*, но так же, как и при настройке скорости, мы откладываем фактический прыжок до следующего вызова *FixedUpdate*. Так что мы будем следить за тем, желателен ли переход, с помощью логического поля *requiredJump*.

Но мы можем оказаться в ситуации, когда *FixedUpdate* не будет вызываться в следующем кадре, и в этом случае значение *selectedJump* возвратится к значению false, а желаемый прыжок будет забыт. Мы можем предотвратить это, комбинируя проверку с ее предыдущим значением с помощью логической операции ИЛИ или присвоения ИЛИ. Таким образом, он остается true, пока мы явно не установим его обратно в false.

*Высота прыжка*

Настроим высоту прыжка нашей сферы. Мы могли бы сделать это, непосредственно управляя скоростью прыжка, но это не интуитивно понятно, поскольку взаимосвязь между начальной скоростью прыжка и высотой прыжка не тривиальна. Удобнее напрямую контролировать высоту прыжка, так что давайте сделаем это.

Прыжки требуют преодоления силы тяжести, поэтому от этого зависит вертикальная скорость.   
В частности, vy = √− 2gh, где g - сила тяжести, а h - желаемая высота. Знак минус находится там, потому что предполагается, что g отрицательно. Мы можем получить его через *Physics.gravity.y*, который также можно настроить через настройки проекта *Physics*. Мы используем вектор гравитации по умолчанию, который равен 9,81, что соответствует средней гравитации Земли.

Обратите внимание, что мы, скорее всего, немного отходим от желаемой высоты из-за дискретного характера физического моделирования. Максимум будет достигнут где-то между временными шагами.

*Прыжки на земле*

В настоящий момент мы можем прыгнуть в любое время, даже находясь в воздухе, что позволяет оставаться в воздухе навсегда. Правильный прыжок может быть начат только тогда, когда сфера находится на земле. Мы не можем напрямую спросить Rigidbody о том, касается ли оно в настоящее время земли, но мы можем получить уведомление, когда оно столкнулось с чем-то, поэтому мы будем использовать это.

Если у MovingSphere есть метод *OnCollisionEnter*, он будет вызван после того, как PhysX обнаружит новое столкновение. Столкновение будет фиксироваться, пока объекты находятся в контакте с другим. После этого вызывается метод *OnCollisionExit*, если он существует. Добавим оба метода в MovingSphere, установив для нового логического поля *onGround* значение true в первом и значение false во втором.

Теперь мы можем прыгать только тогда, когда мы на земле, что на данный момент подразумевает тот случай, когда мы касаемся чего-то. Если мы ни с чем не контактируем, то желаемый прыжок следует игнорировать.

Прыжки должны работать, когда сфера касается только плоскости земли, но если она также ненадолго касается стены, то прыжки станут невозможными. Это происходит потому, что *OnCollisionExit*был вызван для стены, в то время как мы все еще в контакте с землей. Решение состоит в том, чтобы не полагаться на *OnCollisionExit*, а вместо этого добавить метод *OnCollisionStay*, который вызывается на каждом физическом шаге, пока столкновение остается активным.

Каждый физический шаг начинается с вызова всех методов *FixedUpdate*, после чего PhysX делает свое дело, и в конце вызываются методы столкновения. Поэтому, когда *FixedUpdate* будет запущен, *onGround*будет иметь значение true во время последнего шага, если были какие-либо активные столкновения.

*Прыжки в воздухе*

В данный момент мы можем прыгать только на земле, но игры часто допускают двойные или даже тройные прыжки в воздухе. Давайте поддержим это и сделаем настраиваемым, сколько воздушных прыжков разрешено.

Теперь нам нужно отслеживать фазу прыжка, чтобы мы знали, разрешен ли другой прыжок. Мы можем сделать это через целочисленное поле, которое мы устанавливаем в ноль в начале *FixedUpdate*, если мы на земле. Но давайте переместим этот код вместе с поиском скорости в отдельный метод *UpdateState*, чтобы *FixedUpdate* был компактным.

С этого момента мы увеличиваем фазу прыжка при каждом прыжке. И нам разрешено прыгать либо на земле, либо пока мы еще не достигли максимально допустимого числа воздушных прыжков.

*Ограничение скорости вверх*

Быстрые воздушные прыжки позволяют достичь скорости, намного превышающей скорость одиночного прыжка. Мы собираемся изменить это, чтобы мы не могли превысить скорость прыжка, необходимую для достижения желаемой высоты одним прыжком. Первым шагом является выделение рассчитанной скорости прыжка в *Jump*.

Если у нас уже есть скорость движения вверх, вычтите ее из скорости прыжка, прежде чем добавить ее к Y-компоненту скорости. Таким образом, мы никогда не превысим скорость прыжка.

Но если мы уже движемся быстрее скорости прыжка, мы не хотим, чтобы прыжок замедлял нас. Мы можем предотвратить это, гарантируя, что измененная скорость прыжка никогда не станет отрицательной. Это сделано, беря максимум измененной скорости прыжка и ноль.

*Движение в воздухе*

В настоящее время нас не волнует, находится ли сфера на земле или в воздухе при управлении ею, но может иметь смысл, что воздушную сферу труднее контролировать. Уровень контроля может варьироваться от нулевого до полного. Это зависит от игры. Итак, давайте сделаем его настраиваемым, добавив отдельное максимальное воздушное ускорение, по умолчанию равное 1. Это резко снижает контроль в воздухе, но не снимает его полностью.

Какое ускорение мы используем при расчете изменения максимальной скорости в *FixedUpdate*, теперь зависит от того, находимся ли мы на земле или нет.

*Склоны*

Мы используем физику, чтобы перемещать нашу сферу по маленькой плоской поверхности, сталкиваться со стенами и прыгать вокруг. Все это прекрасно работает, поэтому пришло время рассмотреть более сложные среды.

*Установка тестовой сцены*

Мы импортируем пакет *ProBuilder* и используем его для создания некоторых уклонов. Пакет *ProGrids*также удобен для привязки к сетке, хотя в Unity 2019.3 он не нужен. *ProBuilder* довольно прост в использовании, но может занять некоторое время, для освоения. Речь идет в основном о гранях, причем ребра и вершины являются вторичными.

Создадим уклон, начав с куба *ProBuilder*, растянув его до 10 × 5 × 3, выдавливая его еще на 10 единиц в измерении X и сворачивая грани X до их нижних краев. Это создает треугольную двойную рампу с уклонами с обеих сторон длиной десять единиц и высотой 5 единиц.

Разместим десять из них рядом друг с другом на плоскости, варьируя их высоту от одного до десяти единиц, используя склоны с углами примерно 0,0 °, 5,7 °, 11,3 °, 16,7 °, 21,8 °, 26,6 °, 31,0 °, 35,0 °, 38,7 °, 42,0 ° и 45,0 °.

После этого разместим еще десять склонов, на этот раз начиная с 45 ° и вытягивая вершину влево на одну единицу на склон, пока не получим вертикальную стену. Это даси нам углы, которые примерно равны 48,0 °, 51,3 °, 55,0 °, 59,0 °, 63,4 °, 68,2 °, 73,3 °, 78,7 °, 84,3 ° и 90,0 °.

Закончим тестовую сцену, превратив нашу сферу в префаб и добавив 21 экземпляр, по одному на наклон, от полностью горизонтального до полностью вертикального.

*Угол земли*

В настоящее время мы используем 0,9 в качестве порога для классификации чего-либо как земли, но это значение произвольно. Мы могли бы использовать любой порог в диапазоне 0–1. Попытки обеих крайностей дают очень разные результаты.

Сделаем порог настраиваемым, контролируя максимальный угол наклона, так как он более интуитивен, чем Y-компонент вектора нормали склона. Давайте использовать 25 ° по умолчанию.

Когда поверхность горизонтальна, компонент Y ее вектора нормали равен 1. Для идеально вертикальной стены компонент Y равен нулю. Компонент Y изменяется между этими крайностями в зависимости от угла наклона: это косинус угла. Здесь мы имеем дело с единичной окружностью, где Y - вертикальная ось, а горизонтальная ось лежит где-то в плоскости XZ. Еще один способ сказать, что мы смотрим на скалярное произведение вектора вверх и нормали плоскости.

Настроенный угол определяет минимальный результат, который будет считаеться землей. Сохраним порог в поле и вычислим его с помощью *Mathf.Cos* в методе *OnValidate*. Таким образом, он остается синхронизированным с углом, когда мы меняем его через инспектора в режиме воспроизведения. Также вызывайте его в *Awake*, чтобы он вычислялся при построении проекта.

Мы указываем угол в градусах, но *Mathf.Cos* ожидает, что он будет выражен в радианах.

Мы можем преобразовать его, умножив на *Mathf.Deg2Rad*.

Теперь мы можем отрегулировать максимальный угол земли и посмотреть, как он повлияет на движение сфер. Установим угол на 40 °.

*Прыжки на склоне*

Наши сферы всегда прыгают прямо вверх, независимо от того, под каким углом они находятся в данный момент.

Другой подход заключается в том, чтобы спрыгнуть с поверхности земли в направлении ее вектора нормали. Это приведет к различным прыжкам на каждом склоне, так что давайте сделаем это.

Нам нужно отслеживать текущий контакт в нормальном состоянии в поле и сохранять его всякий раз, когда мы сталкиваемся с землей в *EvaluateCollision*.

Но мы можем не коснуться земли. В этом случае мы будем использовать вектор вверх для нормального контакта, поэтому воздушные скачки все еще идут прямо вверх. Установите этот вектор в *UpdateState*, если это необходимо.

Теперь мы должны добавить нормаль контакта прыжка, масштабированную по скорости прыжка, к скорости при прыжке, вместо того чтобы всегда только увеличивать компонент Y. Это означает, что высота прыжка является показателем того, насколько высоко мы прыгаем, когда на ровной поверхности или только в воздухе. Прыжки на склоне не будут такими высокими, но будут влиять на горизонтальную скорость.

Но это означает, что проверка положительной вертикальной скорости также больше не верна. Этот процесс должен стать проверкой скорости, выровненной с нормалью точки контакта. Мы можем найти эту скорость, проецируя скорость на нормаль контакта, вычисляя их точечное произведение через *Vector3.Dot*.

Теперь, когда прыжки выровнены по наклонам, каждая сфера в нашей тестовой сцене получает уникальную траекторию прыжка. Сферы на более крутых склонах больше не прыгают прямо на свои склоны, а замедляются, поскольку прыжок толкает их в направлении, противоположном движению.

*Следование за сферой*

Фиксированная точка обзора работает только тогда, когда сфера ограничена полностью видимой областью. Но обычно персонажи в играх могут бродить по большим территориям. Типичные способы сделать это возможным - использовать вид от первого лица или заставить камеру следовать за аватаром игрока в режиме просмотра от третьего лица. Существуют и другие подходы, например, переключение между несколькими камерами в зависимости от положения аватара.

*Движущаяся камера*

Мы создадим простую вращающуюся камеру, которая будет следовать за нашей сферой в режиме от третьего лица. Определим для нее тип компонента OrbitCamera, присвоив ему атрибут RequireComponent для принудительного подключения к игровому объекту, который также имеет регулярный компонент Camera.

Настроим основную камеру сцены с одной сферой, чтобы она имела этот компонент. Для этого создадим новую сцену с большой плоской плоскостью, расположив камеру так, чтобы она смотрела вниз под углом 45 °, а сфера находилась в центре ее обзора, на расстоянии примерно пяти единиц.

*Сохранение относительной позиции*

Чтобы камера была сфокусирована на сфере, нам нужно указать ей, на чем фокусироваться. Это может быть что угодно, поэтому добавим настраиваемое поле Transform для фокуса. Также добавим параметр для расстояния до точки орбиты, по умолчанию оно установлено на пять единиц.

Каждое обновление мы должны корректировать положение камеры, чтобы она оставалась на нужном расстоянии. Мы сделаем это в **LateUpdate** на случай, если что-нибудь переместит фокус в **Update**. Положение камеры определяется путем перемещения ее от положения фокусировки в направлении, противоположном направлению, на которое она смотрит на величину, равную заданному расстоянию. Мы будем использовать свойство **position** фокуса вместо **localPosition**, чтобы мы могли правильно фокусироваться на дочерних объектах внутри иерархии.

Камера не всегда будет оставаться на одном и том же расстоянии, и ориентации, но поскольку PhysX регулирует положение сферы с фиксированным шагом по времени, то же самое делает и наша камера. Если этот процесс не будет соответствовать частоте кадров, это приведет к дрожанию камеры.

Самый простой и надежный способ исправить это - установить для свойства Rigidbody сферы параметр интерполяции ее положения. Это избавит от дрожащего движения сферы и камеры. Обычно это требуется только для объектов, на которых фокусируется камера.

*Радиус фокуса*

Постоянное удержание сферы в точном фокусе может показаться слишком жестким. Даже самое маленькое движение сферы будет скопировано камерой, которая воздействует на весь вид. Мы можем ослабить это ограничение, заставляя камеру двигаться только тогда, когда ее точка фокусировки слишком сильно отличается от идеальной фокусировки. Мы сделаем это поле настраиваемым, добавив радиус фокуса, установленный на одну единицу по умолчанию.

Расслабленная фокусировка требует, чтобы мы отслеживали текущую точку фокусировки, так как она может больше не совпадать с положением фокуса. Инициализируем его в позиции объекта фокуса в **Awake** и перенесем обновление в отдельный метод **UpdateFocusPoint**.

*Создание орбиты для сферы*

Следующим шагом является возможность отрегулировать ориентацию камеры, чтобы она могла описать орбиту вокруг точки фокусировки. Мы сделаем возможным как ручное управление орбитой, так и автоматический поворот камеры в соответствии с ее фокусом.

*Углы орбиты*

Ориентация камеры может быть описана двумя углами орбиты. Угол X определяет его вертикальную ориентацию: 0 ° направлено прямо к горизонту, а 90 ° направлено вниз. Угол Y определяет горизонтальную ориентацию, при этом 0 ° смотрит вдоль мировой оси Z.

В **LateUpdate** теперь нам нужно построить кватернион, определяющий поворот обзора камеры с помощью метода **Quaternion.Euler**, передавая ей углы орбиты. Для этого потребуется Vector3, в который наш вектор неявно преобразуется, с поворотом Z, установленным в ноль.

Направление взгляда затем можно найти, заменив **transform.forward** на кватернион, умноженный на прямой вектор. И вместо того, чтобы только устанавливать положение камеры, мы теперь вызовем **transform.SetPositionAndRotation**с позицией взгляда и вращением за один раз.

*Контролирование орбиты*

Чтобы вручную управлять орбитой, добавьте параметр конфигурации скорости вращения, выраженный в градусах в секунду. 90° в секунду - разумное значение по умолчанию.

Добавим метод **ManualRotation**, который извлекает входной вектор. Для этого определим входные оси вертикальной камеры и горизонтальной камеры, привязанные к третьей и четвертой осям, клавишам ijkl и qe, а чувствительность мыши увеличим до 0,5. Хорошая идея - сделать чувствительность настраиваемой в вашей игре и позволить переключать направления осей.

Если есть входное значение, превышающее небольшое значение эпсилона, например, 0,001, добавьте входное значение к углам орбиты, масштабируемым по скорости вращения и разности времени. Опять же, мы делаем этот процесс независимым от игрового времени.

Обратим внимание, что сфера все еще контролируется в мировом пространстве, независимо от ориентации камеры. Так что, если вы повернете камеру на 180° по горизонтали, управление сферой будет выглядеть перевернутым. Это позволяет легко сохранять один и тот же ракурс независимо от вида камеры, но может быть дезориентирующим.

*Ограничение углов*

В то время как камера может описывать полные горизонтальные орбиты, вертикальное вращение перевернет мир с ног на голову, когда оно выйдет за пределы 90° в любом направлении. Даже до этого момента становится трудно увидеть, куда вы идете, когда смотрите в основном вверх или вниз. Итак, давайте добавим параметры конфигурации, чтобы ограничить минимальный и максимальный вертикальный угол, причем крайние значения ограничены максимумом 89° в любом направлении. Давайте использовать -30° и 60° в качестве значений по умолчанию.

Максимальное значение никогда не должно опускаться ниже минимального, поэтому используйте это ограничение в методе **OnValidate**.

Добавим метод **ConstrainAngles**, который ограничивает вертикальный угол орбиты в настроенном диапазоне. Горизонтальная орбита не имеет границ, но убедимся, что угол остается в диапазоне 0–360.

Нам нужно ограничивать углы только тогда, когда они изменились. Поэтому сделаем так, чтобы **ManualRotation** вернул, внесены ли изменение, и вызовим **ConstrainAngles** на основе этих данных в**LateUpdate**. Нам также нужно только пересчитать вращение, если было изменение, в противном случае мы можем получить существующее.

Мы также должны убедиться, что начальный поворот соответствует углам орбиты в **Awake**.

*Автоматическое выравнивание*

Общей особенностью камер орбиты является то, что они выстраиваются, чтобы оставаться позади аватара игрока. Мы сделаем это, автоматически регулируя горизонтальный угол орбиты. Но важно, чтобы игрок всегда мог переопределить это автоматическое поведение, и чтобы автоматическое вращение не включалось немедленно. Поэтому мы добавим настраиваемую задержку выравнивания, установленную по умолчанию на пять секунд. Эта задержка не имеет верхней границы.

Затем добавим метод **AutomaticRotation**, который также возвращает информацию об изменении орбиты. Его выполнение прекращается, если текущее время минус время последнего ручного вращения меньше, чем задержка выравнивания.

В **LateUpdate** мы теперь ограничиваем углы и вычисляем вращение, когда происходит ручное или автоматическое вращение, в таком порядке.

*Направление фокуса*

Критерии, используемые для выравнивания камер, различны. В нашем случае мы будем основывать его исключительно на движении точки фокусировки с предыдущего кадра. Идея состоит в том, что наиболее разумно смотреть в том направлении, в котором фокус был последним. Чтобы сделать это возможным, нам нужно знать как текущую, так и предыдущую точку фокусировки, поэтому создадим **UpdateFocusPoint** для обоих полей.

Затем автоматическое вращение рассчитывает вектор движения для текущего кадра. Поскольку мы вращаемся только горизонтально, нам нужно только 2D-движение в плоскости XZ. Если величина квадрата этого вектора движения меньше небольшого порога, например, 0,0001, тогда движения не было много, и мы не будем беспокоиться о вращении.

В противном случае мы должны выяснить горизонтальный угол, соответствующий текущему направлению. Создайте статический метод **GetAngle** для преобразования 2D-направления в угол. Компонент Y направления - это косинус угла, который нам нужен, поэтому поместим его через **Mathf.Acos** и затем пересчитаем из радианов в градусы.

Но этот угол может представлять вращение по часовой стрелке или против часовой стрелки относительно мировой оси Z. Мы можем посмотреть на компонент X направления, чтобы узнать, какое это вращение. Если X отрицательно, то оно против часовой стрелки, и мы должны вычесть угол из 360°.

Вернувшись в **AutomaticRotation**, мы можем использовать **GetAngle**, чтобы получить угол курса вращения, передав ему нормализованный вектор движения. Поскольку у нас уже есть его квадратичная величина, более эффективно сделать нормализацию самостоятельно. Результатом становится новый горизонтальный угол орбиты.

*Плавное выравнивание*

Автоматическое выравнивание работает, но скачек слишком. Давайте замедлим его, используя также настроенную скорость вращения для автоматизации, она с имитирует ручное вращение. Для этого мы можем использовать **Mathf.MoveTowardsAngle**, который работает как **Mathf.MoveTowards**, за исключением того, что он может работать с диапазоном углов 0–360.

Так лучше, но у нас всегда используется максимальная скорость вращения, даже для небольших изменений. Более естественным было бы сделать шкалу скорости вращения с разницей между текущим и желаемым углом. Мы сделаем линейное масштабирование до некоторого угла, под которым мы будем вращаться на полной скорости. Настроим этот угол, добавив опцию конфигурации выравнивания с плавным диапазоном, с диапазоном 0–90 и значением по умолчанию 45 °.

Чтобы сделать это, нам нужно знать дельту угла в **AutomaticRotation**, которую мы можем найти, передав текущий и желаемый угол в **Mathf.DeltaAngle** и взяв его модуль. Если эта дельта попадает в масштаб плавного диапазона, настройте вращение соответственно.

Мы разобрались со случаем, когда фокус перемещается от камеры, но мы также можем разобрать ситуацию, когда фокус перемещается к камере. Это предотвращает вращение камеры на полной скорости, меняя направление каждый раз, когда курс пересекает границу 180°. Он работает так же, за исключением того, что мы используем 180° минус абсолютная дельта.

Наконец, мы можем еще немного ослабить вращение крошечных углов, масштабируя скорость вращения на минимум дельты времени и квадрата движения.

Обратим внимание, что при таком подходе можно перемещать сферу прямо к камере, не поворачивая ее. Крошечные отклонения в направлении также будут затухать. Автоматическое вращение вступит в силу плавно после значительного изменения курса.

*Движение относительно камеры*

На данный момент у нас есть простая камера орбиты. Теперь сделаем движение игрока относительно точки обзора камеры.

*Получение данных*

Входные данные могут быть определены в любом пространстве, а не только в мировом пространстве или камере орбиты. Это может быть любое пространство, определенное компонентом **Transform**. Для этого добавим в **MovingSphere** поле конфигурации пространства ввода игрока.

Присвоим камеру орбиты этому полю. Это специфичная для сцены конфигурация, поэтому она не является частью префаба сферы, хотя она может быть настроена на себя, что будет порождать движение относительно ее собственной ориентации.

Если пространство ввода не установлено, то мы сохраняем ввод данных в мировом пространстве. В противном случае мы должны перейти из предоставленного пространства в мировое пространство. Мы можем сделать это, вызвав **Transform.TransformDirection** в **Update**, если задано пространство для ввода игрока.

*Нормализованное направление*

Хотя преобразование в мировое пространство заставляет сферу двигаться в правильном направлении, ее скорость движения зависит от угла вертикальной орбиты. Чем дальше он отклоняется от горизонтали, тем медленнее движется сфера. Это происходит потому, что мы ожидаем, что желаемая скорость будет лежать в плоскости XZ. Мы можем решить эту проблему, извлекая прямой и правый векторы из пространства ввода игрока, отбрасывая их компоненты Y и нормализуя их. Тогда желаемая скорость становится суммой этих векторов, масштабируемых входом игрока.

*Столкновения камеры*

В настоящее время наша камера заботится только о своем положении и ориентации относительно ее фокуса. Он ничего не знает об остальной части сцены. Таким образом, она идет прямо через другую геометрию, что вызывает несколько проблем. Во-первых, это некрасиво. Во-вторых, это может привести к тому, что геометрия будет препятствовать нашему взгляду на сферу, что затруднит навигацию. В-третьих, отсечение геометрии может выявить области, которые не должны быть видны. Мы начнем с рассмотрения только случая, когда расстояние фокусировки камеры установлено на ноль.

*Уменьшение дистанции обзора*

Существуют различные стратегии, которые могут использоваться для обеспечения правильного обзора камеры. Мы применим самое простое, а именно: потянуть камеру вперед по направлению ее взгляда, если что-то окажется между камерой и точкой фокусировки.

Самый очевидный способ обнаружить проблему - направить луч из точки фокусировки туда, где мы хотим разместить камеру. Сделаем это в **OrbitCamera.LateUpdate**, как только у нас будет направление взгляда. Если мы ударяемся обо что-то, мы используем расстояние до точки удара вместо настроенного расстояния.

Притянув камеру ближе к точке фокусировки, вы можете приблизить ее к сфере. Когда сфера пересекает ближнюю плоскость камеры, она может частично или даже полностью обрезаться. Вы можете установить минимальное расстояние, чтобы избежать этого, но это будет означать, что камера остается внутри другой геометрии. Идеального решения для этого не существует, но его можно уменьшить, ограничив вертикальные углы орбиты, не делая геометрию уровня слишком узкой и уменьшив расстояние до плоскости отсечения камеры.

*Поддержание чистоты ближней плоскости*

Использование одного луча недостаточно для полного решения проблемы. Это связано с тем, что прямоугольник ближней плоскости камеры все еще может частично прорезать геометрию, даже если между положением камеры и точкой фокусировки существует четкая линия. Решение состоит в том, чтобы вместо этого выполнить приведение к ячейке, сопоставив прямоугольник ближней плоскости камеры в мировом пространстве, который представляет собой ближайшую вещь, которую может видеть камера. Это аналог датчика камеры.

Во-первых, **OrbitCamera** нужна ссылка на свой компонент **Camera**.

Во-вторых, для проецирования блока требуется трехмерный вектор, который содержит половину протяженности блока, что означает половину его ширины, высоты и глубины.

Половину высоты можно найти, взяв тангенс половины угла поля зрения камеры в радианах, масштабированный по расстоянию, близкому к плоскости. Половина ширины масштабируется по соотношению сторон камеры. Глубина блока равна нулю.

Теперь заменим **Physics.Raycast** на **Physics.BoxCast** в **LateUpdate**.

Ближняя плоскость находится перед положением камеры, поэтому мы должны приводить только к этому расстоянию, которое является настроенным расстоянием минус расстояние ближней плоскости камеры. Если мы в результате чего-то попадаем в другую геометрию, то конечное расстояние - это расстояние удара плюс расстояние в ближней плоскости.

Обратим внимание, что это означает, что положение камеры все еще может оказаться внутри геометрии, но ее прямоугольник в ближней плоскости всегда будет оставаться снаружи. Конечно, это может не сработать, если проецирование блока уже начинается внутри геометрии. Если объект фокусировки уже пересекает геометрию, вероятно, камера это сделает.

*Радиус фокусировки*

Наш текущий подход работает, но только если радиус фокусировки равен нулю. Когда фокус ослаблен, мы можем получить точку фокусировки внутри геометрии, даже если идеальная точка фокусировки действительна. Таким образом, мы не можем ожидать, что точка фокусировки является действительным началом приведения блока, поэтому вместо этого нам придется использовать идеальную точку фокусировки. Мы приведем оттуда к положению блока в ближней плоскости, которое мы найдем, переместившись из положения камеры в положение фокусировки, пока не достигнем ближней плоскости.

Если мы что-то ударили камерой, мы позиционируем блок как можно дальше, затем мы собираемся найти соответствующую позицию камеры.

*Маскировка препятствий*

Сделаем так, чтобы камера могла пересекать некоторую геометрию, игнорируя ее при выполнении приведения блока. Это позволит игнорировать мелкую детальную геометрию, либо по соображениям производительности, либо по соображениям стабильности камеры. При желании эти объекты все еще могут быть обнаружены, но исчезают, вместо того, чтобы воздействовать на положение камеры. Прозрачную геометрию также можно игнорировать. Самое главное, мы должны игнорировать саму сферу. При проецировании из сферы это поведение всегда будет игнорироваться, но менее отзывчивая камера может закончить проецирование вне сферы. Если оно попадет в сферу, камера прыгнет на противоположную сторону сферы.

Мы контролируем это поведение через поле конфигурации маски слоя, точно так же, как те, которые использует сфера.

# **Разработка игровой зоны**

После написания кода для объекта, который будет передвигаться по нашей игровой зоне, надо сделать небольшую карту, на которой наша сфера будет передвигаться посредством ввода игроком команд для управления.

**Добавление графические объекты из Unity Aseets Store**

Для того, чтобы использовать готовые ассеты, мы должны быть зарегистрированы на официальном сайте Unity (Рис. 2).

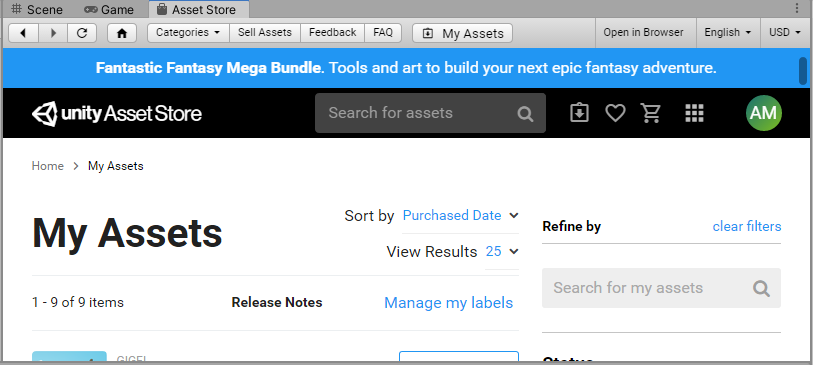


Рис. 2.

Выбираем в предложенном меню то, что нам наиболее подходит. Я импортировала несколько ассетов – Low Poly Environment Pack, RPG Poly Pack - Lite.

Нам нужно просто скачать данные ассеты, а потом импортировать их в наш проект (Рис. 3 – уже импортированный ассет).

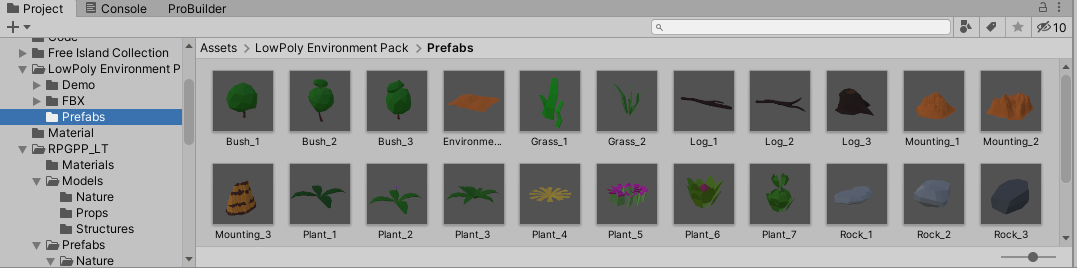


Рис. 3.

Теперь добавляем на нашу «Сцену» графические объекты (Рис. 4-5). Добавляем деревья, камни, растения, дом, вазы и тд.



Рис. 4.



Рис. 5.

Конечный результат (Рис. 6-8):

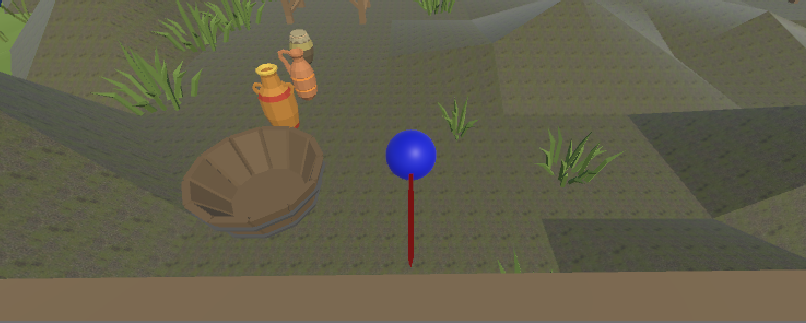


Рис. 6.



Рис. 7.



Рис. 8.

**Заключение**

При выполнении данного курсового проекта мы изучили разработку 3D объектов на платформе Unity.

Мы добились поставленных задач, а именно изучили саму платформу Unity, создали физический объект и задали ему движение, добавили 3d объекты из готового Assets.

Данный проект не является законченным продуктом, поэтому в будущем его можно доработать и улучшить. Также можно использовать любой из элементов данной работы в любом другом проекте Unity. Были рассмотрены основные аспекты разработки игр в целом и на Unity в частности.

# **Список литературы**

[1] Хокинг, Джозеф. Unity — в действии. Мультиплатформенная разработка на C#. СПб : Питер, 2016. — 336 с.

[2] Торн, Алан. Искусство создания сценариев в Unity. СПб : ДМК, 2016. - 362 с.

[3] Unity3d. Начало работы, практические советы. Рецензия // Хабрахабр. [Электронный ресурс] URL: https://habr.com/ru/post/161463/ (дата обращения 05.12.2012).

[4] Как создать простую 2D-игру на Unity. [Электронный ресурс] URL: https://skillbox.ru/media/code/kak\_sozdat\_prostuyu\_2d\_igru\_na\_unity/ (дата обращения 10.07.2019).

[5] Создание 2D игры на Unity за час! Краткий гайд. [Электронный ресурс] URL: https://itproger.com/news/187 (дата обращения 29.04.2019).

# **Код программы**

OrbitCamera.cs:

using UnityEngine;

[RequireComponent(typeof(Camera))]

public class OrbitCamera : MonoBehaviour

{

[SerializeField]

Transform focus = default;

[SerializeField, Range(1f, 20f)]

float distance = 5f;

[SerializeField, Min(0f)]

float focusRadius = 5f;

[SerializeField, Range(0f, 1f)]

float focusCentering = 0.5f;

[SerializeField, Range(1f, 360f)]

float rotationSpeed = 90f;

[SerializeField, Range(-89f, 89f)]

float minVerticalAngle = -45f, maxVerticalAngle = 45f;

[SerializeField, Min(0f)]

float alignDelay = 5f;

[SerializeField, Range(0f, 90f)]

float alignSmoothRange = 45f;

[SerializeField]

LayerMask obstructionMask = -1;

Camera regularCamera;

Vector3 focusPoint, previousFocusPoint;

Vector2 orbitAngles = new Vector2(45f, 0f);

float lastManualRotationTime;

Vector3 CameraHalfExtends

{

get

{

Vector3 halfExtends;

halfExtends.y =

regularCamera.nearClipPlane \*

Mathf.Tan(0.5f \* Mathf.Deg2Rad \* regularCamera.fieldOfView);

halfExtends.x = halfExtends.y \* regularCamera.aspect;

halfExtends.z = 0f;

return halfExtends;

}

}

void OnValidate()

{

if (maxVerticalAngle < minVerticalAngle)

{

maxVerticalAngle = minVerticalAngle;

}

}

void Awake()

{

regularCamera = GetComponent<Camera>();

focusPoint = focus.position;

transform.localRotation = Quaternion.Euler(orbitAngles);

}

void LateUpdate()

{

UpdateFocusPoint();

Quaternion lookRotation;

if (ManualRotation() || AutomaticRotation())

{

ConstrainAngles();

lookRotation = Quaternion.Euler(orbitAngles);

}

else

{

lookRotation = transform.localRotation;

}

Vector3 lookDirection = lookRotation \* Vector3.forward;

Vector3 lookPosition = focusPoint - lookDirection \* distance;

Vector3 rectOffset = lookDirection \* regularCamera.nearClipPlane;

Vector3 rectPosition = lookPosition + rectOffset;

Vector3 castFrom = focus.position;

Vector3 castLine = rectPosition - castFrom;

float castDistance = castLine.magnitude;

Vector3 castDirection = castLine / castDistance;

if (Physics.BoxCast(

castFrom, CameraHalfExtends, castDirection, out RaycastHit hit,

lookRotation, castDistance, obstructionMask

))

{

rectPosition = castFrom + castDirection \* hit.distance;

lookPosition = rectPosition - rectOffset;

}

transform.SetPositionAndRotation(lookPosition, lookRotation);

}

void UpdateFocusPoint()

{

previousFocusPoint = focusPoint;

Vector3 targetPoint = focus.position;

if (focusRadius > 0f)

{

float distance = Vector3.Distance(targetPoint, focusPoint);

if (distance > focusRadius)

{

focusPoint = Vector3.Lerp(

targetPoint, focusPoint, focusRadius / distance

);

}

if (distance > 0.01f && focusCentering > 0f)

{

focusPoint = Vector3.Lerp(

targetPoint, focusPoint,

Mathf.Pow(1f - focusCentering, Time.unscaledDeltaTime)

);

}

}

else

{

focusPoint = targetPoint;

}

}

bool ManualRotation()

{

Vector2 input = new Vector2(

Input.GetAxis("Vertical Camera"),

Input.GetAxis("Horizontal Camera")

);

const float e = 0.001f;

if (input.x < -e || input.x > e || input.y < -e || input.y > e)

{

orbitAngles += rotationSpeed \* Time.unscaledDeltaTime \* input;

lastManualRotationTime = Time.unscaledTime;

return true;

}

return false;

}

bool AutomaticRotation()

{

if (Time.unscaledTime - lastManualRotationTime < alignDelay)

{

return false;

}

Vector2 movement = new Vector2(

focusPoint.x - previousFocusPoint.x,

focusPoint.z - previousFocusPoint.z

);

float movementDeltaSqr = movement.sqrMagnitude;

if (movementDeltaSqr < 0.0001f)

{

return false;

}

float headingAngle = GetAngle(movement / Mathf.Sqrt(movementDeltaSqr));

float deltaAbs = Mathf.Abs(Mathf.DeltaAngle(orbitAngles.y, headingAngle));

float rotationChange =

rotationSpeed \* Mathf.Min(Time.unscaledDeltaTime, movementDeltaSqr);

if (deltaAbs < alignSmoothRange)

{

rotationChange \*= deltaAbs / alignSmoothRange;

}

else if (180f - deltaAbs < alignSmoothRange)

{

rotationChange \*= (180f - deltaAbs) / alignSmoothRange;

}

orbitAngles.y =

Mathf.MoveTowardsAngle(orbitAngles.y, headingAngle, rotationChange);

return true;

}

void ConstrainAngles()

{

orbitAngles.x =

Mathf.Clamp(orbitAngles.x, minVerticalAngle, maxVerticalAngle);

if (orbitAngles.y < 0f)

{

orbitAngles.y += 360f;

}

else if (orbitAngles.y >= 360f)

{

orbitAngles.y -= 360f;

}

}

static float GetAngle(Vector2 direction)

{

float angle = Mathf.Acos(direction.y) \* Mathf.Rad2Deg;

return direction.x < 0f ? 360f - angle : angle;

}

}

Moving.cs:

using UnityEngine;

public class Moving : MonoBehaviour

{

[SerializeField]

Transform playerInputSpace = default;

[SerializeField, Range(0f, 100f)]

float maxSpeed = 10f;

[SerializeField, Range(0f, 100f)]

float maxAcceleration = 10f, maxAirAcceleration = 1f;

[SerializeField, Range(0f, 10f)]

float jumpHeight = 2f;

[SerializeField, Range(0, 5)]

int maxAirJumps = 0;

[SerializeField, Range(0, 90)]

float maxGroundAngle = 25f, maxStairsAngle = 50f;

[SerializeField, Range(0f, 100f)]

float maxSnapSpeed = 100f;

[SerializeField, Min(0f)]

float probeDistance = 1f;

[SerializeField]

LayerMask probeMask = -1, stairsMask = -1;

Rigidbody body;

Vector3 velocity, desiredVelocity;

bool desiredJump;

Vector3 contactNormal, steepNormal;

int groundContactCount, steepContactCount;

bool OnGround => groundContactCount > 0;

bool OnSteep => steepContactCount > 0;

int jumpPhase;

float minGroundDotProduct, minStairsDotProduct;

int stepsSinceLastGrounded, stepsSinceLastJump;

void OnValidate()

{

minGroundDotProduct = Mathf.Cos(maxGroundAngle \* Mathf.Deg2Rad);

minStairsDotProduct = Mathf.Cos(maxStairsAngle \* Mathf.Deg2Rad);

}

void Awake()

{

body = GetComponent<Rigidbody>();

OnValidate();

}

void Update()

{

Vector2 playerInput;

playerInput.x = Input.GetAxis("Horizontal");

playerInput.y = Input.GetAxis("Vertical");

playerInput = Vector2.ClampMagnitude(playerInput, 1f);

if (playerInputSpace)

{

Vector3 forward = playerInputSpace.forward;

forward.y = 0f;

forward.Normalize();

Vector3 right = playerInputSpace.right;

right.y = 0f;

right.Normalize();

desiredVelocity =

(forward \* playerInput.y + right \* playerInput.x) \* maxSpeed;

}

else

{

desiredVelocity =

new Vector3(playerInput.x, 0f, playerInput.y) \* maxSpeed;

}

desiredJump |= Input.GetButtonDown("Jump");

}

void FixedUpdate()

{

UpdateState();

AdjustVelocity();

if (desiredJump)

{

desiredJump = false;

Jump();

}

body.velocity = velocity;

ClearState();

}

void ClearState()

{

groundContactCount = steepContactCount = 0;

contactNormal = steepNormal = Vector3.zero;

}

void UpdateState()

{

stepsSinceLastGrounded += 1;

stepsSinceLastJump += 1;

velocity = body.velocity;

if (OnGround || SnapToGround() || CheckSteepContacts())

{

stepsSinceLastGrounded = 0;

if (stepsSinceLastJump > 1)

{

jumpPhase = 0;

}

if (groundContactCount > 1)

{

contactNormal.Normalize();

}

}

else

{

contactNormal = Vector3.up;

}

}

bool SnapToGround()

{

if (stepsSinceLastGrounded > 1 || stepsSinceLastJump <= 2)

{

return false;

}

float speed = velocity.magnitude;

if (speed > maxSnapSpeed)

{

return false;

}

if (!Physics.Raycast(

body.position, Vector3.down, out RaycastHit hit,

probeDistance, probeMask

))

{

return false;

}

if (hit.normal.y < GetMinDot(hit.collider.gameObject.layer))

{

return false;

}

groundContactCount = 1;

contactNormal = hit.normal;

float dot = Vector3.Dot(velocity, hit.normal);

if (dot > 0f)

{

velocity = (velocity - hit.normal \* dot).normalized \* speed;

}

return true;

}

bool CheckSteepContacts()

{

if (steepContactCount > 1)

{

steepNormal.Normalize();

if (steepNormal.y >= minGroundDotProduct)

{

steepContactCount = 0;

groundContactCount = 1;

contactNormal = steepNormal;

return true;

}

}

return false;

}

void AdjustVelocity()

{

Vector3 xAxis = ProjectOnContactPlane(Vector3.right).normalized;

Vector3 zAxis = ProjectOnContactPlane(Vector3.forward).normalized;

float currentX = Vector3.Dot(velocity, xAxis);

float currentZ = Vector3.Dot(velocity, zAxis);

float acceleration = OnGround ? maxAcceleration : maxAirAcceleration;

float maxSpeedChange = acceleration \* Time.deltaTime;

float newX =

Mathf.MoveTowards(currentX, desiredVelocity.x, maxSpeedChange);

float newZ =

Mathf.MoveTowards(currentZ, desiredVelocity.z, maxSpeedChange);

velocity += xAxis \* (newX - currentX) + zAxis \* (newZ - currentZ);

}

void Jump()

{

Vector3 jumpDirection;

if (OnGround)

{

jumpDirection = contactNormal;

}

else if (OnSteep)

{

jumpDirection = steepNormal;

jumpPhase = 0;

}

else if (maxAirJumps > 0 && jumpPhase <= maxAirJumps)

{

if (jumpPhase == 0)

{

jumpPhase = 1;

}

jumpDirection = contactNormal;

}

else

{

return;

}

stepsSinceLastJump = 0;

jumpPhase += 1;

float jumpSpeed = Mathf.Sqrt(-2f \* Physics.gravity.y \* jumpHeight);

jumpDirection = (jumpDirection + Vector3.up).normalized;

float alignedSpeed = Vector3.Dot(velocity, jumpDirection);

if (alignedSpeed > 0f)

{

jumpSpeed = Mathf.Max(jumpSpeed - alignedSpeed, 0f);

}

velocity += jumpDirection \* jumpSpeed;

}

void OnCollisionEnter(Collision collision)

{

EvaluateCollision(collision);

}

void OnCollisionStay(Collision collision)

{

EvaluateCollision(collision);

}

void EvaluateCollision(Collision collision)

{

float minDot = GetMinDot(collision.gameObject.layer);

for (int i = 0; i < collision.contactCount; i++)

{

Vector3 normal = collision.GetContact(i).normal;

if (normal.y >= minDot)

{

groundContactCount += 1;

contactNormal += normal;

}

else if (normal.y > -0.01f)

{

steepContactCount += 1;

steepNormal += normal;

}

}

}

Vector3 ProjectOnContactPlane(Vector3 vector)

{

return vector - contactNormal \* Vector3.Dot(vector, contactNormal);

}

float GetMinDot(int layer)

{

return (stairsMask & (1 << layer)) == 0 ?

minGroundDotProduct : minStairsDotProduct;

}

}