МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ РОССИЙСКОЙ ФЕДЕРАЦИИ

ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ БЮДЖЕТНОЕ ОБРАЗОВАТЕЛЬНОЕ

УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ

НОВОСИБИРСКИЙ ГОСУДАРСТВЕННЫЙ ТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ

Кафедра вычислительной техники

**Расчетно-графическая работа**

**по дисциплине «Основы теории управления»**

«Создание 2D игры в Unity»

Факультет: АВТ

Группа: АВТ-718 Преподаватель: Воевода А.А.

Студенты: Герасименко А.О.

Саменов Т.Б.

Новосибирск

2019 г.

Оглавление

[Задача 3](#_Toc27625462)

[Цели работы 3](#_Toc27625463)

[Теоретические сведения 4](#_Toc27625464)

[Физика 4](#_Toc27625465)

[Rigidbodies (твёрдые тела) 4](#_Toc27625466)

[Засыпание 5](#_Toc27625467)

[Коллайдеры (Colliders) 5](#_Toc27625468)

[Сочленения (Joints) 10](#_Toc27625469)

[Анимация в Unity 10](#_Toc27625470)

[Свойства 11](#_Toc27625471)

[Создание и Использование Скриптов 14](#_Toc27625472)

[Создание скриптов 14](#_Toc27625473)

[Проектирование, выбор тематики 17](#_Toc27625474)

[Разработка 18](#_Toc27625475)

[Приложение 27](#_Toc27625476)

# Задача

Спроектировать и создать прототип 2D приложения на платформе PC с поддержкой физики, с использованием возможностей Unity2D.

# Цели работы

* Изучить возможности физики, предоставляемые Unity2D
* Спроектировать приложение, выбрать тематику
* Создать прототип приложения
* В перспективе доработать созданный прототип до полноценного приложения

# Теоретические сведения

## Физика

Чтобы физическое поведение было правдоподобной, объект в игре нужно правильно ускорить и задействовать столкновения, гравитацию и другие силы. Встроенный в Unity физические движки обеспечивают пользователя компонентами для обработки симуляции физики. С помощью настройки всего нескольких параметров, можно создать объекты, которые ведут себя пассивно реалистично (т.е., они будут перемещены в результате столкновений и падений, но не начнут двигаться сами по себе). Управляя физикой из скриптов, возможно придать объекту динамику автомобиля, машины или даже подвижного куска ткани.

В Unity различают 2 типа физических движков: для 2d и для 3d. Так как наша игра основывается на 2d, в дальнейшем будет затрагиваться физика именно для 2d графики.

### Rigidbodies (твёрдые тела)

Rigidbody - это основной компонент, подключающий физическое поведение для объекта. С прикреплённым Rigidbody, объект немедленно начнёт реагировать на гравитацию. Если добавлен один или несколько компонентов Collider, то при коллизиях (столкновениях) объект будет передвигаться.

Так как компонент Rigidbody управляет перемещением объекта, к которому он прикреплён, не следует пытаться воздействовать на объект из кода с помощью изменения таких свойств [Transform](https://docs.unity3d.com/ru/current/Manual/class-Transform.html), как position и rotation. Вместо этого следует применять силы для того, чтобы толкать объект и позволить физическому движку рассчитать результаты.

Существуют случаи, при которых возможно, чтобы у объектов был Rigidbody, но физический движок не управлял его движением. Например, возможно пожелать управлять своим персонажем напрямую из кода, но при этом хотите, чтобы он взаимодействовал с триггерами. Этот тип не физического движения, производимый с помощью кода, называется кинематическим движением. У компонента Rigidbody есть свойство Is Kinematic, которое может исключить объект из-под контроля физического движка, и позволить перемещать его кинематически из скрипта. Значение Is Kinematic можно менять из кода, чтобы включать или отключать физику для объекта, но эта возможность требует дополнительных ресурсов и должна использоваться как можно реже.

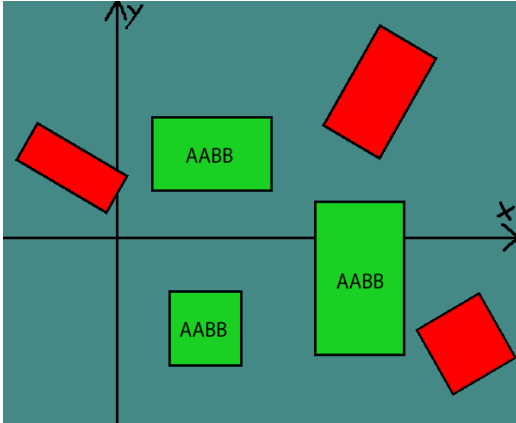


Рисунок 1 - Визуализация AABB (Axis Aligned Bounding Box)

### Засыпание

Когда твёрдое тело перемещается со скоростью, меньшей определённого минимального порога, физический движок предполагает, что оно остановилось и находится в покое. При этом, объект не будет вновь двигаться до тех пор, пока с ним не произойдёт столкновение или пока к нему не применят силу, так что он уходит в “спящий” режим. Эта оптимизация означает, что на объект не будут расходоваться ресурсы CPU, пока его вновь не “разбудят” (т.е. не вновь не приведут в движение). По многим причинам засыпание и пробуждение твёрдых тел происходит прозрачно. Однако, иногда объект не удаётся разбудить, если в него или от него переместится статичный коллайдер (тот, что без твёрдого тела) изменяя положение трансформации. Это может привести, скажем, к висящему в воздухе твёрдому телу, когда пол под ним сдвинулся вниз. В таких случаях объект можно разбудить принудительно, с помощью функции WakeUp.

### Коллайдеры (Colliders)

Компоненты **коллайдера** определяют форму объекта во время физических столкновений. Коллайдер, который невидим, не обязательно должен иметь ту же форму, что и сетка объекта, и фактически грубое приближение часто более эффективно и неразличимо в игровом процессе.

Простейшими (и наименее ресурсоемкими) коллайдерами являются так называемые примитивные типы коллайдеров. В 2D это [Box Collider 2D](https://docs.unity3d.com/ru/current/Manual/class-BoxCollider2D.html) и [Circle Collider 2D](https://docs.unity3d.com/ru/current/Manual/class-CircleCollider2D.html) . Любое их количество может быть добавлено к одному объекту для создания составных коллайдеров .

При аккуратном позиционировании и определении размеров составные коллайдеры часто могут достаточно хорошо приближаться к форме объекта, сохраняя при этом низкую нагрузку на процессор. Дополнительную гибкость можно получить, если иметь дополнительные коллайдеры на дочерних объектах (например, ящики можно поворачивать относительно локальных осей родительского объекта). При создании составного коллайдера, подобного этому, должен быть только один компонент Rigidbody, помещенный в корневой объект в иерархии.

Стоит обратить внимание, что примитивные коллайдеры не будут корректно работать с преобразованиями сдвига - это означает, что, если использовать комбинацию поворотов и неоднородных масштабов в иерархии преобразования, так что результирующая фигура больше не будет соответствовать примитивной форме, примитивный коллайдер не будет быть в состоянии представить это правильно.

Однако в некоторых случаях даже составные коллайдеры недостаточно точны.  В 2D [Polygon Collider 2D](https://docs.unity3d.com/ru/current/Manual/class-PolygonCollider2D.html) , как правило, не будет идеально соответствовать форме спрайтовой графики, но можно уточнить форму до любого уровня детализации. Однако эти коллайдеры намного интенсивнее процессорных, чем примитивные типы, поэтому использовать их следует экономно, чтобы поддерживать хорошую производительность. Кроме того, коллайдер сетки обычно не может столкнуться с другим коллайдером сетки (то есть, когда они вступают в контакт, ничего не произойдет). В некоторых случаях возможно обойти это, пометив сетчатый коллайдер как **выпуклый** в инспекторе. Это создаст форму коллайдера в виде «выпуклой оболочки», которая похожа на исходную сетку. Преимущество этого состоит в том, что коллайдер с выпуклой сеткой может сталкиваться с другими коллайдерами сетки, так что пользователь сможет использовать эту функцию, когда есть движущийся персонаж с подходящей формой. Тем не менее, хорошее общее правило - использовать сетчатые коллайдеры для геометрии сцены и приближать форму движущихся объектов, используя составные примитивные коллайдеры.

Коллайдеры могут быть добавлены к объекту без компонента Rigidbody для создания полов, стен и других неподвижных элементов сцены. Они называются **статическими** коллайдерами. В общем, не стоит менять положение статических коллайдеров, изменяя положение Transform, так как это сильно повлияет на производительность физического движка. Коллайдеры на объекте , который действительно имеет Rigidbody известны как динамические коллайдеры. Статические коллайдеры могут взаимодействовать с динамическими коллайдерами, но поскольку у них нет твердого тела, они не будут двигаться в ответ на столкновения.

#### Физические материалы (Physics Materials)

Когда коллайдеры взаимодействуют, их поверхности должны имитировать свойства материала, который они должны представлять. Например, слой льда будет скользким, в то время как резиновый шарик будет сильно растягиваться. Хотя форма коллайдеров не деформируется во время столкновений, их трение и отскок можно настроить с помощью **физических материалов.** Правильное определение параметров может потребовать небольшого количества проб и ошибок, но, например, ледяной материал будет иметь нулевое (или очень низкое) трение, а резиновый материал - с высоким коэффициентом трения и почти идеально упругим.

#### Триггеры

Система сценариев может обнаруживать столкновения и инициировать действия с помощью OnCollisionEnter функции. Однако также можно использовать физический движок просто для определения, когда один коллайдер входит в пространство другого, не создавая столкновения. Коллайдер, настроенный как **триггер** (с использованием свойства **Is Trigger**), не ведет себя как сплошной объект и просто пропускает другие коллайдеры. Когда коллайдер входит в свое пространство, триггер вызывает OnTriggerEnterфункцию в сценариях объекта триггера.

#### Функция обратного вызова при коллизии

В случае коллизий, физические движущиеся объекты с особыми именами в скриптах, которые присоединяются к вовлеченным в коллизию объектам. Пользователь может установить любой код для реакции на событие.

При первом обновлении физики, где обнаруживается столкновение, вызывается OnCollisionEnter функция. Во время обновлений, где поддерживается контакт, OnCollisionStay вызывается и, наконец, OnCollisionExit указывает, что контакт был разорван. Trigger коллайдеры называют аналогичные OnTriggerEnter, OnTriggerStayи OnTriggerExitфункцию.

У обычных не триггерных коллизий есть еще одна дополнительная деталь: как минимум один из вовлеченных в коллизию объектов должен обладать не кинематическим твердым телом (т.е. IsKinematic должен быть выключен). Если оба объекта является кинематическим, то тогда не будет вызываться функции, вроде OnCollisionEnterи т.д. Это жесткое тело будет незамедлительно вызывать OnTriggerEnterпересечение триггерного коллайдера.

#### Взаимодействия коллайдеров

Коллайдеры взаимодействуют друг с другом по разному, в зависимости от того, как настроены их компоненты [Rigidbody](https://docs.unity3d.com/ru/current/Manual/RigidbodiesOverview.html). Тремя важными конфигурациями являются статичный коллайдер (Static Collider) (т.е. компонент Rigidbody отсутствует вообще), Rigidbody коллайдер (Rigidbody Collider), и кинематический Rigidbody коллайдер (Kinematic Rigidbody Collider).

#### Статичный коллайдер (Static Collider)

Это игровой объект, у которого есть коллайдер, но нет Rigidbody. Статичные коллайдеры используются для геометрии уровней, которая всегда стоит на месте и совсем не двигается. Встречные Rigidbody объекты будут врезаться в статичный коллайдер, но его не сдвинут.

В физический движок заложено предположение, что статичные коллайдеры никогда не двигаются или меняются, и, на основе этого предположения, движок делает полезные оптимизации. Следовательно, статичные коллайдеры нельзя включать/выключать, двигать или масштабировать во время игрового процесса. Если вы измените статичный коллайдер, то в результате физическим движком будет вызван дополнительный внутренний перерасчёт, который будет сопровождаться большим падением производительности. Хуже того, изменения иногда могут оставить коллайдер в неопределённом состоянии, в результате чего будут производиться ошибочные физические расчёты. Например, рейкаст к изменённому статичному коллайдеру может не обнаружить коллайдера или обнаружить его в случайном месте в пространстве. Кроме того, Rigidbody объекты, в которых врежется статичный коллайдер, не обязательно будут “разбужены”, и статичный коллайдер не применит никакого трения. По этим причинам, следует изменять только коллайдеры с Rigidbody. Если вы хотите, чтобы на коллайдер объекта не влияли встречные Rigidbody, но, чтобы его можно было двигать при помощи скрипта, то вам следует прикрепить *кинематический* Rigidbody компонент к нему, нежели вообще не добавлять Rigidbody.

#### Rigidbody коллайдер (Rigidbody Collider)

Это игровой объект, к которому прикреплён коллайдер и нормальный не кинематический Rigidbody. Rigidbody коллайдеры полностью симулируются физическим движком и могут реагировать на коллизии и силы, приложенные из скрипта. Они могут сталкиваться с другими объектами (включая статичные коллайдеры) и являются самой распространённой конфигурацией коллайдера в играх, которые используют физику.

#### Кинематические Rigidbody коллайдеры (Kinematic Rigidbody Collider)

Это игровой объект, который прикреплен к коллайдеру и кинематическому жесткому телу (т.е. свойство *IsKinematic*компонент Жесткое тело включено). Изменяя компонент Transform, вы можете перемещать объект с кинематическим ригидным телом, но он не будет реагировать на коллизии и приложенные силы. Кинематические твердые тела должны использоваться для коллайдеров, которые могут двигаться или периодически отключаться / включаться, иначе они будут вести себя как статичные коллайдеры. Обычно это является скользящей дверью, которая может быть открыта. В отличие от статического коллайдера, движущийся кинематографический Rigidbody будет «будить» других Rigidbody.

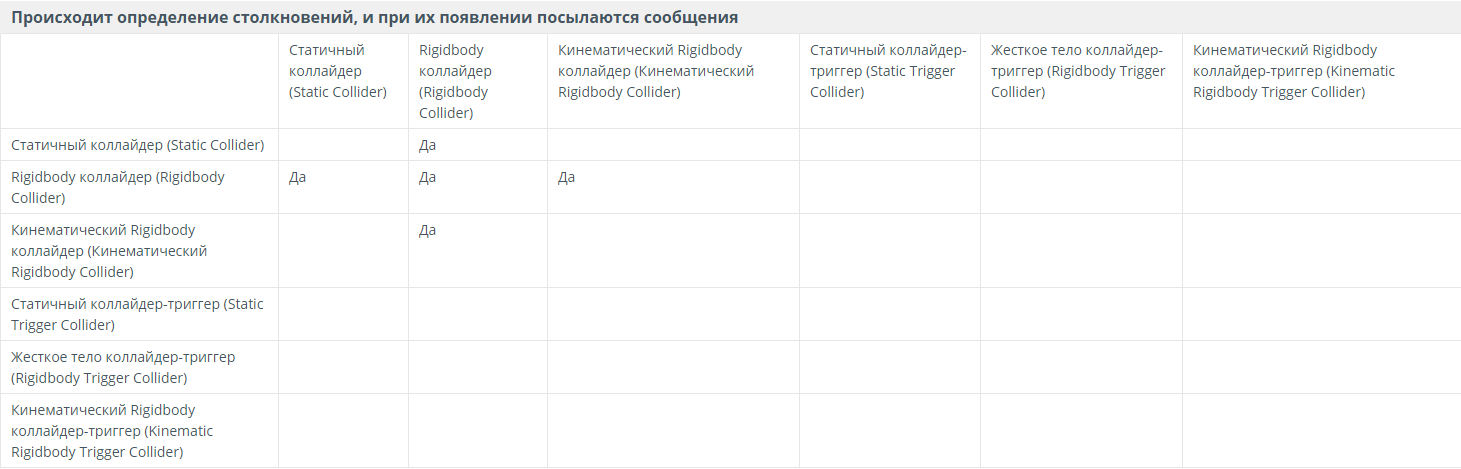
Даже когда они неподвижны, кинематические твердые тела коллайдеры ведут себя иначе, в отличие от статичных коллайдеров. Если вы хотите, чтобы в вашем скрипте принимали события триггера. Если вы не хотите, чтобы это происходило, вы можете включить свойство *IsKinematic* .

Компонент Rigidbody можно переключать между нормальным и кинематическим поведением в любое время с *помощью* свойства *IsKinematic* .

Типичным примером этого является эффект «тряпичной куклы», когда персонаж обычно движется под анимацией, но его физически выбрасывает взрыв или сильное столкновение. Каждой персонажа может быть предоставлен собственный компонент *Rigidbody с* включенным *IsKinematic* по умолчанию. Конечности будут нормально перемещаться с помощью анимации, пока *IsKinematic не* будет отключен для всех из них, и они сразу же будут вести себя как физические объекты.

#### Матрица действий коллизии

Когда сталкиваются 2 объекта, количество различных событий в скрипте зависит от конфигурационных компонентов. Ниже перечислены детали, которые должны быть вызваны объектами. Правило физики не имеет отношения к объекту, к которому нет присоединенного твердого тела.





### Сочленения (Joints)

С помощью компонента Joint, вы можете добавить один rigidbody объект к другому или к фиксированной точке в пространстве. Обычно для того, чтобы у сочленения была хоть какая-то свобода движения, Unity предоставляет различные Joint компоненты, реализующие разные ограничения. Например, [Hinge Joint](https://docs.unity3d.com/ru/current/Manual/class-HingeJoint.html) позволяет осуществлять вращение вокруг заданной точки и оси, в то время как [Spring Joint](https://docs.unity3d.com/ru/current/Manual/class-SpringJoint.html) держит объекты раздельно, но расстояние между ними может немного пружинить.

У сочленений также есть другие опции, которые можно включить для различных эффектов. Например, вы можете настроить сочленение так, чтобы оно разрывалось при применении к нему силы больше заданного порога. Некоторые сочленения также позволяют возникать движущей силе (drive force) между соединёнными объектами для автоматического приведения их в движение.

## Анимация в Unity

В Unity создавать анимацию можно с помощью:

* скриптов;
* кадров;
* скелетов.

Например, **в скрипте** можно прописать изменение параметров объекта при определенных условиях: обновлении масштабов, добавлении какого-либо цвета и так далее.

Что-то более продвинутое делается с помощью **покадровой анимации** — когда для разных состояний объекта (стоит, идет, в прыжке, атакует) создается несколько спрайтов (двумерное изображение), которые сменяют друг друга с определенным интервалом.

И, наконец, самое продвинутое, но чуть более сложное, — **скелеты**. Они позволяют использовать один спрайт, а потом приводить его в движение, изменяя положения отдельных частей.

Мы рассмотрим только скриптовую и покадровую анимацию в Unity, потому что для скелетной требуются специальные ассеты, а также они могут быть сложнее для понимания.

### Свойства

| **Свойство:** | **Функция:** |
| --- | --- |
| Animation (анимация) | Анимация, которая будет проигрываться по умолчанию, если отмечен флажок Play Automatically. |
| Animation (анимация) | Список доступных из кода анимаций. |
| **Play Automatically** | Должна ли анимация автоматически проигрываться при запуске игры? |
| **Animate Physics** | Должен ли анимируемый объект обрабатываться физическим движком? |
| **Culling Type** | Определяет, когда не должна проигрываться анимация. |
| **Always Animate** | Всегда проигрывать. |
| **Based on Renderers** | Culling на основании позы по умолчанию. |
| **Based on Clip Bounds** | Culling на основании границ клипа (рассчитываются при импорте анимации), то есть анимация не будет проигрываться, если границы клипа не попадают в поле зрения. |
| **Based on User Bounds** | Culling на основании границ, выбранных пользователем, то есть, анимация не будет проигрываться, если выбранные пользователем границы не попадают в поле зрения |

#### Скриптинг анимации (Legacy)

Система анимации в Unity позволяет создавать великолепно анимированных персонажей. Она поддерживает блендинг, микширование, сложение анимаций, синхронизацию цикла ходьбы, анимационные слои, контроль всех аспектов проигрывания (время, скорость, веса блендинга), скиннинг мешей с 1, 2 или 4 костями на вершину, а также основанные на физике rag-dolls (тряпичные куклы) и процедурную анимацию. Для получения лучших результатов, рекомендуется почитать о практических подходах и принципах создания персонажей с оптимальной производительностью в Unity на странице Моделирование оптимизированных персонажей.

Создание анимированного персонажа включает в себя две вещи - перемещение в пространстве сцены и соответствующая анимация. Чтобы узнать больше о перемещении персонажа, прочитайте [Character Controller page](https://docs.unity3d.com/ru/current/Manual/class-CharacterController.html). На этой странице разбираются вопросы анимации. Фактически, вся анимация может быть сделана во встроенном редакторе Unity.

Вы можете попробовать [примеры](http://unity3d.com/support/resources/example-projects/) показывающие готовую анимацию персонажей. Чуть позже, когда вы освоите базовые вещи, можете взглянуть на [описание класса Animation](https://docs.unity3d.com/ru/current/ScriptReference/Animation.html).

#### Смешивание анимации

В современных играх, смешивание анимации является необходимой функцией для обеспечения персонажа плавными анимациями. Аниматоры создают отдельные анимации, например, цикл ходьбы, цикл бега, анимация в состоянии покоя или стрельбы. В любой момент во время игры, должна быть возможность перехода с анимации спокойствия в цикл ходьбы и обратно. Естественно, вы хотите чтобы этот переход был плавным и без внезапных рывков в движении.

В этом случае смешивание анимации становится полезным. В Unity можно иметь любое количество анимаций, которые проигрываются на одном и том же персонаже. Все анимации смешиваются или складываются вместе для создания окончательной анимации.

Сначала сделаем плавный переход между анимациями спокойствия и ходьбы. Чтобы облегчить работу написания скрипта, сперва нужно изменить Wrap Mode анимации на Loop. Затем нужно отключить Play Automatically для уверенности, что никто, кроме нашего скрипта, не проигрывает анимацию.

Наш первый скрипт для анимации персонажа довольно простой; нам нужен способ для определения скорости движения персонажа, а после этого делать переход между анимациями ходьбы и покоя. Для этого простого теста мы будем использовать стандартные оси ввода:-

Чтобы использовать этот скрипт в вашем проекте:-

1. Создайте Javascript файл используя Assets->Create Other->Javascript.
2. Скопируйте и вставьте код в него.
3. Перетащите скрипт на персонажа (он должен быть привязан к GameObject, который имеет анимацию).

После нажатия на кнопку Play, персонаж начнет шагать на месте, пока вы будете удерживать нажатой кнопку со стрелкой вверх, и вернется в позу ожидания, если вы отпустите ее.

#### Слои анимации

Слои это невероятно полезная концепция, позволяющая классифицировать анимацию и приоритезировать веса.

Система анимации Unity может смешивать несколько анимаций, в соответствие с вашими пожеланиями. Вы можете назначить веса вручную, или просто использовать animation.CrossFade(), который будет анимировать вес автоматически.

#### Веса смешивания всегда нормализуются перед применением

Предположим, что у вас есть цикл ходьбы и цикл бега, имеющие веса, равные 1 (100%). Когда Unity генерирует окончательную анимацию, веса нормализуются, то есть вклад цикла ходьбы составит 50% анимации, и вклад цикла бега также составит 50%.

Типично, вам захочется указать, какая анимация получает больший вес, когда проигрывается две анимации. Конечно, можно вручную убедиться, что веса суммируются в 100%, но легче просто использовать для этого слои.

#### Аддитивные (additive) анимации

Аддитивные анимации и технология смешивания анимаций позволяют уменьшить общее количество анимаций, которые вам надо создать для вашей игры, и эти техники также важны для создания лицевой анимации.

Предположим, вам захотелось создать персонажа, который наклоняется в стороны во время поворотов, когда он ходит или бегает. Это приводит к 4 комбинациям (идти-наклоняться-влево, идти-наклоняться-вправо, бежать-наклоняться-влево, бежать-наклоняться-вправо), для каждой из которых нужна анимация. Создание отдельной анимации на каждую комбинацию, очевидно, ведет к множеству дополнительной работы, даже в таком простом случае. Но количество комбинаций увеличивается с каждым добавляемым действием. К счастью, аддитивные анимации и смешивание позволяет избежать необходимости создания отдельных анимаций для комбинаций простых движений.

## Создание и Использование Скриптов

Поведение игровых объектов контролируется с помощью компонентов (Components), которые присоединяются к ним. Несмотря на то, что встроенные компоненты Unity могут быть очень разносторонними, вскоре вы обнаружите, что вам нужно выйти за пределы их возможностей, чтобы реализовать ваши собственные особенности геймплея. Unity позволяет вам создавать свои компоненты, используя скрипты. Они позволяют активировать игровые события, изменять параметры компонентов, и отвечать на ввод пользователя каким вам угодно способом.

Unity изначально поддерживает три языка программирования:

* **C#** (произносится как Си-шарп), стандартный в отрасли язык подобный Java или C++;
* **UnityScript**, язык, разработанный специально для использования в Unity по образцу JavaScript;

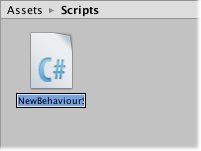
В дополнение к этим, с Unity могут быть использованы многие другие языки семейства .NET, если они могут компилировать совместимые DLL - см. [эту страницу](https://docs.unity3d.com/ru/current/Manual/UsingDLL.html) для получения подробностей.

Изучение искусства программирования и использования этих языкам выходит за рамки данного введения. Однако есть множество книг, обучающих материалов и ресурсов для изучения программирования в среде Unity. Посетите [Обучающий раздел](http://unity3d.com/learn) на нашем сайте для получения подробной информации.

### Создание скриптов

В отличии от других ассетов, скрипты обычно создаются непосредственно в Unity. Вы можете создать скрипт используя меню Create в левом верхнем углу панели Project или выбрав Assets > Create > C# Script (или JavaScript/Boo скрипт) в главном меню.

Новый скрипт будет создан в папке, которую вы выбрали в панели Project. Имя нового скрипта будет выделено, предлагая вам ввести новое имя.



Лучше ввести новое имя скрипта сразу после создания чем изменять его потом. Имя, которое вы введете будет использовано, чтобы создать начальный текст в скрипте, как описано ниже.

#### Структура файла скрипта

После двойного щелчка на скрипте в Unity, он будет открыт в текстовом редакторе. По умолчанию Unity будет использовать MonoDevelop, но вы можете выбрать любой редактор из панели External Tools в настройках Unity.

Содержимое файла будет выглядеть примерно так:

using UnityEngine;

using System.Collections;

public class MainPlayer : MonoBehaviour {

// Use this for initialization

void Start () {

}

// Update is called once per frame

void Update () {

}

}

Скрипт взаимодействует с внутренними механизмами Unity за счет создания класса, наследованного от встроенного класса, называемого MonoBehaviour. Вы можете думать о классе как о своего рода плане для создания нового типа компонента, который может быть прикреплен к игровому объекту. Каждый раз, когда вы присоединяете скриптовый компонент к игровому объекту, создается новый экземпляр объекта, определенный планом. Имя класса берется из имени, которое вы указали при создании файла. Имя класса и имя файла должны быть одинаковыми, для того, чтобы скриптовый компонент мог быть присоединен к игровому объекту.

Основные вещи, достойные внимания, это две функции, определенные внутри класса. Функция **Update** - это место для размещения кода, который будет обрабатывать обновление кадра для игрового объекта. Это может быть движение, срабатывание действий и ответная реакция на ввод пользователя, в основном всё, что должно быть обработано с течением времени во игровом процессе. Чтобы позволить функции Update выполнять свою работу, часто бывает полезно инициализировать переменные, считать свойства и осуществить связь с другими игровыми объектами до того, как будут совершены какие-либо действия. Функция **Start** будет вызвана Unity до начала игрового процесса (т.е. до первого вызова функции Update), и это идеальное место для выполнения инициализации.

Заметка для опытных программистов: вы можете быть удивлены, что инициализация объекта выполняется не в функции-конструкторе. Это потому, что создание объектов обрабатывается редактором и происходит не в начале игрового процесса, как вы могли бы ожидать. Если вы попытаетесь определить конструктор для скриптового компонента, он будет мешать нормальной работе Unity и может вызвать серьезные проблемы с проектом.

A UnityScript script works a bit differently to C# script:

#pragma strict

function Start () {

}

function Update () {

}

Здесь функции Start и Update имеют такое же значение, но класс не объявлен явно. Предполагается, что скрипт сам по себе определяет класс; он будет неявно производным от MonoBehaviour и получит своё имя от имени файла скриптового ассета.

#### Управление игровым объектом

Как было сказано ранее, скрипт определяет только план компонента и, таким образом, никакой его код не будет активирован до тех пор, пока экземпляр скрипта не будет присоединен к игровому объекту. Вы можете прикрепить скрипт перетаскиванием ассета скрипта на игровой объект в панели Hierarchy или через окно Inspector выбранного игрового объекта.

Имеется также подменю Scripts в меню Component, которое содержит все скрипты, доступные в проекте, включая те, которые вы создали сами. Экземпляр скрипта выглядит так же, как и другие компоненты в окне Inspector:

https://docs.unity3d.com/ru/current/uploads/Main/ScriptInInspector.png

После присоединения скрипт начнет работать, когда вы нажмете Play и запустите игру. Вы можете проверить это добавив следующий код в функцию Start:

// Use this for initialization

void Start () {

Debug.Log("I am alive!");

}

**Debug.Log** это команда, которая просто выводит сообщение на консольный вывод Unity. Если вы нажмете сейчас Play, вы увидите сообщение внизу основного окна редактора Unity в окне Console (меню: Window > Console).

## Проектирование, выбор тематики

Начнем с того, что выберем тематику нашего будущего приложения. В данной работе мы не будем вдаваться в подробности сюжета. Наше приложение - это 2D игра - хоррор, с мрачной историей и глубоким сюжетом. Как и в любой игре, у нас есть главный персонаж, который должен уметь двигаться, прыгать, взаимодействовать с окружающим миром, и при всем при этом не проваливаться сквозь текстуры, не застревать в них и не врезаться в невидимые стены. В свою очередь, окружающий мир также способен воздействовать на главного героя, будь то поваленный шкаф, лежащий на пути, либо противник.

С тематикой и задумкой разобрались. Следующий шаг - примерный набросок приложения. Игра должна иметь стартовое меню, набор настроек, разные кнопки и возможность сохранения.

## Разработка

Исходный код скриптов, написанных и задействованных во время разработки будет предоставлен в приложении.



Рисунок 2 - Главное меню

В главном меню реализована возможность перехода в меню «Настройки» и выхода из игры. Так же будет добавлена кнопка «Загрузить игру».



Рисунок 3 - Меню настроек

В меню настроек присутствуют стандартные настройки для большинства PC игр : регулировка громкости, разделяющаяся на музыку и внутриигровые звуки, яркость, возможность смены разрешения на 800x600, 1280x720, 1366x768 и 1920x1080, а так же полноэкранный или оконный режим. Изначально все настройки выставлены на средний уровень, а разрешение выбрано 1280x720.



Рисунок 4 - Первая локация с интерактивными объектами

Первая локация – то место, где впервые окажется главный герой после загрузочного экрана. Предполагается, что большинство объектов будут интерактивными, что позволит игроку «развернуться» в данной локации и исследовать каждый уголок. Таким образом, игрок сможет узнать больше о главном герое, как он живет, чем занимается, с кем общается и многое другое.



Рисунок 5 - Меню паузы

Возможность паузы в игре – важная особенность. Нередко требуется отойти от компьютера или поговорить по телефону. В меню паузы можно попасть, начав игру и нажав клавишу «Esc». В данном меню реализованы возможности изменения настроек, так же, как и в главном меню, переход в главное меню, выход из игры и, конечно же, продолжение игры. По ходу разработки будет добавлены кнопки «Сохранить» и «Загрузить».



Рисунок 6 - Главный герой в начальной локации\

С помощью анимации был добавлен главный герой, а также анимированы его действия: бег, прыжок, бездействие, смерть. Анимация является неотъемлемой частью большинства игр. Без нее игры кажутся простыми и убогими. Анимация придает главному герою некий шарм.



Рисунок 7 - Способность персонажа к прыжками и бегу

Прыжок был добавлен в игру для того, чтобы персонаж не только бегал из стороны в сторону, но также был способен преодолевать наземные препятствия.

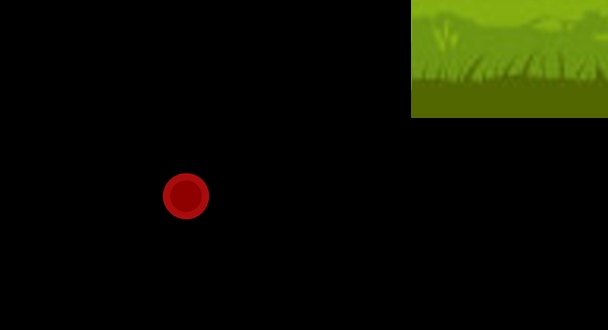
  


Рисунок 8 - Смерть героя

Смерть вносит большой вклад в игры. Благодаря ей геймер способен больше погрузиться в игровой процесс и насладиться им. Данный элемент игры доказывает, что не все так просто, игру можно пройти приложив некоторые усилия.

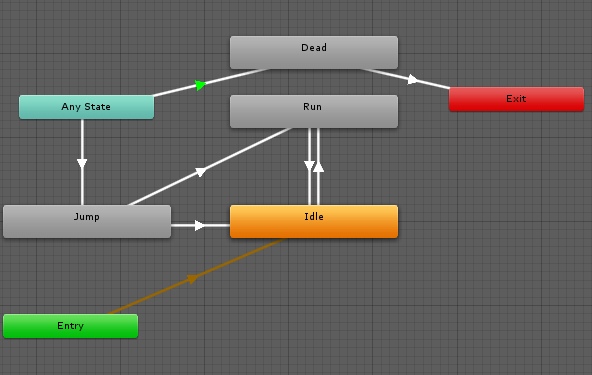


Рисунок 9 - Блок-схема анимации персонажа

Данная блок – схема визуализирует переход из одного состояния анимации в другой. Во время выполнения некоторого условия происходит переход между блоками.



Рисунок 10 - Переход на вторую локацию

Переход на следующую анимацию был реализован посредством взаимодействия с дверью. Данная дверь ведет наружу.



Рисунок 11 - Вторая локация

Вторая локация была введена для разнообразия и насыщения игрового процесса. Благодаря дополнительным локациям игра преображает свой вид.

**Музыкальное сопровождение**

Музыкальное сопровождение данного проекта было выполнено в секвенсоре FL Studio 12. В ходе проекта была написана главная тема с использованием внутренних (FL Keys) и сторонних (Nexus, Sylenth1) плагинов секвенсора.

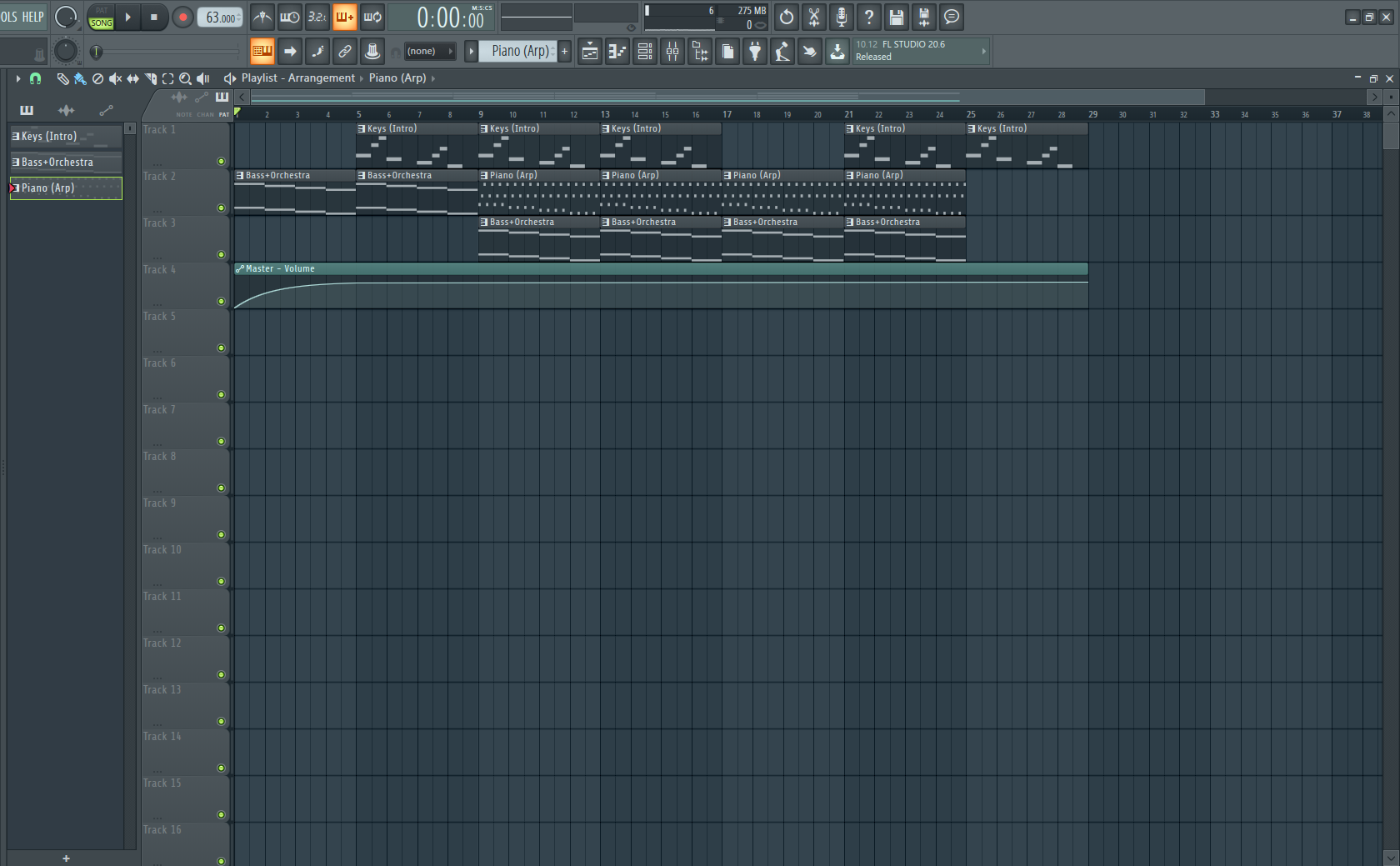


Рисунок 12 - Структура проекта в FL Studio 12

На мастер-канал были наложены следующие плагины:

* The Glue
* FabFilter-Pro Q2



Рисунок - Мастер-канал

# Приложение

**DverVGarag.cs**

using System.Collections;

using System.Collections.Generic;

using UnityEngine;

using UnityEngine.SceneManagement;

public class DverVGarag : MonoBehaviour {

public int numberScene;

void OnTriggerStay2D(Collider2D other)

{

other.attachedRigidbody.AddForce(-0.1F \* other.attachedRigidbody.velocity);

Debug.Log("Perehod");

if (other.tag == "Player")

{

if (Input.GetKeyDown(KeyCode.E))

{

SceneManager.LoadScene(numberScene);

}

}

}

}

**LoadScene.cs**

using System.Collections;

using UnityEngine;

using UnityEngine.UI;

using UnityEngine.SceneManagement;

public class LoadScene : MonoBehaviour

{

public int sceneID;

public Image Loading;

public Text progressText;

// Start is called before the first frame update

void Start()

{

StartCoroutine(Asyncload());

}

IEnumerator Asyncload()

{

AsyncOperation operation = SceneManager.LoadSceneAsync(sceneID);

while (!operation.isDone)

{

float progress = operation.progress / 0.9f;

Loading.fillAmount = progress;

progressText.text = string.Format("{0:0}%", progress \* 100);

yield return null;

}

}

}

**MainMenu.cs**

using System.Collections;

using System.Collections.Generic;

using UnityEngine;

using UnityEngine.SceneManagement;

using UnityEngine.UI;

public class MainMenu : MonoBehaviour

{

public void Start()

{

}

public void PlayGame()

{

SceneManager.LoadScene(SceneManager.GetActiveScene().buildIndex + 1);

}

public void QuitGame()

{

Debug.Log("Quit");

Application.Quit();

}

}

**Newcam.cs**

using System.Collections;

using System.Collections.Generic;

using UnityEngine;

public class newcam : MonoBehaviour

{

public float dampTime = 0.15f;

private Vector3 velocity = Vector3.zero;

public Transform target;

void Update()

{

if (target)

{

Vector3 point = GetComponent<Camera>().WorldToViewportPoint(

new Vector3(target.position.x, target.position.y + 0.75f, target.position.z));

Vector3 delta = new Vector3(target.position.x, target.position.y + 0.75f, target.position.z) - GetComponent<Camera>().ViewportToWorldPoint(new Vector3(0.5f, 0.5f, point.z));

Vector3 destination = transform.position + delta;

transform.position = Vector3.SmoothDamp(transform.position, destination, ref velocity, dampTime);

}

}

}

**PauseMenu.cs**

using System.Collections;

using System.Collections.Generic;

using UnityEngine;

using UnityEngine.SceneManagement;

public class Pause\_menu : MonoBehaviour

{

public static bool GameIsPaused = false;

public GameObject pauseMenuUI;

// Update is called once per frame

void Update()

{

if (Input.GetKeyDown(KeyCode.Escape))

{

if (GameIsPaused)

{

Resume();

}

else

{

Pause();

}

}

}

public void Resume()

{

pauseMenuUI.SetActive(false);

Time.timeScale = 1f;

GameIsPaused = false;

}

void Pause()

{

pauseMenuUI.SetActive(true);

Time.timeScale = 0f;

GameIsPaused = true;

}

public void Load\_menu()

{

Debug.Log("Load");

Time.timeScale = 0f;

SceneManager.LoadScene("menu");

}

public void Quit()

{

Debug.Log("Quit");

Application.Quit();

}

}

**PlayerController.cs**

using UnityEngine;

using System.Collections;

using System.Collections.Generic;

public class PlayerController : MonoBehaviour

{

public float maxSpeed = 6f;

private bool isFacingRight = true;

private Animator anim;

public bool isGrounded = false;

public Transform groundCheck;

private float groundRadius = 0.2f;

public LayerMask whatIsGround;

public bool isDead = false;

public Transform deadCheck;

public LayerMask whatIsDead;

public void Start()

{

anim = GetComponent<Animator>();

}

public bool CheckDeath()

{

if (isDead) return true;

isDead = Physics2D.OverlapCircle(deadCheck.position,groundRadius, whatIsDead);

anim.SetBool("isDead", isDead);

if (isDead)

{

isGrounded = true;

anim.SetBool("Ground", isGrounded);

return true;

}

return false;

}

public void FixedUpdate()

{

if (CheckDeath())

{

return;

}

isGrounded = Physics2D.OverlapCircle(groundCheck.position, groundRadius, whatIsGround);

anim.SetBool("Ground", isGrounded);

anim.SetFloat("vSpeed", GetComponent<Rigidbody2D>().velocity.y);

if (!isGrounded)

return;

float move = Input.GetAxis("Horizontal");

anim.SetFloat("Speed", Mathf.Abs(move));

GetComponent<Rigidbody2D>().velocity = new Vector2(move \* maxSpeed, GetComponent<Rigidbody2D>().velocity.y);

if (move > 0 && !isFacingRight)

Flip();

else if (move < 0 && isFacingRight)

Flip();

}

private void Update()

{

if (isDead)

{

return;

}

if (isGrounded && Input.GetKeyDown(KeyCode.Space))

{

anim.SetBool("Ground", false);

GetComponent<Rigidbody2D>().AddForce(new Vector2(0, 350));

}

}

private void Flip()

{

isFacingRight = !isFacingRight;

Vector3 theScale = transform.localScale;

theScale.x \*= -1;

transform.localScale = theScale;

}

}

**Settings.cs**

using System.Collections;

using System.Collections.Generic;

using UnityEngine;

using UnityEngine.UI;

using System;

using System.IO;

public class Settings : MonoBehaviour

{

public Toggle FullScreenToggle;

public bool isFull;

List<string> resolutions;

public int CurrentResolution = 1;

public Dropdown DropdownResolution;

public void Start()

{

isFull = true;

CurRes();

}

public void Update()

{

CurRes();

}

public void ActiveToggle()

{

if (!FullScreenToggle.isOn)

{

Screen.fullScreen = false;

isFull = false;

Debug.Log("Full disabled");

}

else

{

Screen.fullScreen = true;

isFull = true;

Debug.Log("Full enabled");

}

}

public void Awake()

{

resolutions = new List<string>

{

"800x600",

"1280x720",

"1366x768",

"1920x1080"

};

DropdownResolution.ClearOptions();

DropdownResolution.AddOptions(resolutions);

}

public void ResolutionChange(int r)

{

switch (r)

{

case 0:

Screen.SetResolution(800, 600, isFull);

break;

case 1:

Screen.SetResolution(1280, 720, isFull);

break;

case 2:

Screen.SetResolution(1366, 768, isFull);

break;

case 3:

Screen.SetResolution(1920, 1080, isFull);

break;

default:

break;

}

CurrentResolution = r;

}

public void CurRes()

{

DropdownResolution.captionText.text = resolutions[CurrentResolution];

}

}

**CameraController.cs**

using UnityEngine;

using System.Collections;

public class CameraContoller : MonoBehaviour

{

public float dumping = 1.5f;

public Vector2 offset = new Vector2(2f, 1f);

public bool isLeft;

private Transform player;

private int lastX;

void Start()

{

offset = new Vector2(Mathf.Abs(offset.x), offset.y);

FindPlayer(isLeft);

}

public void FindPlayer(bool playerIsLeft)

{

player = GameObject.FindGameObjectWithTag("Player").transform;

lastX = Mathf.RoundToInt(player.position.x);

if (playerIsLeft)

{

transform.position = new Vector3(player.position.x - offset.x, player.position.y - offset.y, transform.position.z);

}

else

{

transform.position = new Vector3(player.position.x + offset.x, player.position.y + offset.y, transform.position.z);

}

}

void Update()

{

if (player)

{

int currentX = Mathf.RoundToInt(player.position.x);

if (currentX > lastX) isLeft = false; else if (currentX < lastX) isLeft = true;

lastX = Mathf.RoundToInt(player.position.x);

Vector3 target;

if (isLeft)

{

target = new Vector3(player.position.x - offset.x, player.position.y + offset.y, transform.position.z);

}

else

{

target = new Vector3(player.position.x + offset.x, player.position.y + offset.y, transform.position.z);

}

Vector3 currentPosition = Vector3.Lerp(transform.position, target, dumping \* Time.deltaTime);

transform.position = currentPosition;

}

}

}