**Содержание**

Ход работы 2

V0.0.005 2

V0.0.006 4

V0.0.007 5

V0.0.008 6

V0.0.009 7

V0.0.010 8

Дальнейшие действия в реализации проекте 10

Ход работы

На втором этапе создания проекта было решено воспользоваться системой контроля версий Unity Collaborate. При загрузке текущего проекта игра получила версию v0.0.004.

V0.0.005

Следующей целью стало добавление экранных джойстиков и проверка их работы на целевом устройстве.

Изначально было решено использовать уже добавленные в новую систему ввода Unity джойстики. Для их использования необходимо лишь нарисовать джойстики, и задать какой тип ввода они будут моделировать.

Был нарисован джойстик и расположен слева внизу. Он моделирует левый стик геймпада.

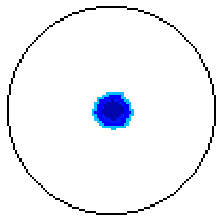


Рисунок 1. Экранный джойстик.

Новая система ввода позволяет достаточно легко переключаться между разными устройствами ввода, поэтому реализация перемещения персонажа с помощью геймпада не заняла много времени. При тестировании первого экранного джойстика было обнаружено множество недостатков:

* Стик джойстика не “магнитится” к месту касания, из-за чего персонажем управлять не удобно. Необходимо немного изменить внутреннюю реализацию стандартного джойстика.
* Джойстик вызывает метод перемещения только когда происходят изменения, т.е. если сместить стик и удерживать на одном месте, то события не будут вызываться. На перемещение персонажа это не повлияло т.к. и ранее в методе Move задавалось лишь направление, а перемещение происходило в независимом методе Update, но нужно помнить об этой особенности в будущем.
* При повторном касании стика (например, другим пальцем) он возвращается в начальное положение (на центр джойстика) и управляется от обоих касаний. Джойстик не должен обрабатывать касания, если хотя бы одно уже есть. Этот недостаток будет устранен позже.
* После добавления экранного джойстика касания стали сильно нагружать систему. Это заметно по нагреву устройства и снижению FPS (Frame per second – кадры в секунду). Главной причиной такого поведения является то, что для обработки нажатий новая система ввода Unity загружает множество различных компонентов и просчитывает множество различных значений. Мы пользуемся лишь малой их частью, поэтому это лишняя нагрузка на устройство. На данном этапе мы не будем устранять эту проблему т.к. для её устранения необходимо написать собственный код для джойстика, что требует большого количества времени.
* В некоторых ситуациях при множественных касаниях стик не возвращается на место после отпускания пальца, из-за чего персонаж продолжает бежать. Причина пока не известна.

Чтобы стик стал магнитящемся в исходный скрипт Unity было внесено несколько изменений:

* Ранее касание считывалось только со стика, теперь считывается со всего джойстика.
* При смещении касания из координаты стика больше не вычитается его начальная координата.

Так как при запуске приложения на мобильном устройстве нет возможности выводить какие-либо данные в консоль, появилась новая задача – сделать консоль внутри приложения. Чтобы не указывать ссылку на консоль в каждом скрипте, для которого необходимо выводить информацию было принято решение создать отдельный скрипт для управления консолью и предоставить доступ к этому классу в любом месте кода. Один из способов сделать это – создать в классе управления консолью статическое поле, хранящее ссылку на первый созданный объект консоли. Тогда ко всем методам можно получить доступ через ИмяКласса.ИмяСтатическогоЧлена. Так как в будущем нам еще понадобится подобное поведение для различных глобальных систем воспользуемся шаблоном проектирования “Одиночка”.

**Одиночка (англ. Singleton)** — порождающий шаблон проектирования, гарантирующий, что в однопоточном приложении будет единственный экземпляр некоторого класса, и предоставляющий глобальную точку доступа к этому экземпляру.

Для реализации такого шаблона необходимо создать обобщенный класс, где параметром T будет объект этого же класса. Подобная реализация позволит избежать ошибок с типами данных в будущем. Также важно унаследовать этот класс от класса MonoBehaviour, чтобы классы, наследуемые от класса Singleton, не теряли возможности корректно взаимодействовать с объектами Unity. В этом классе описывается вся логика, такая как: не может быть более одного объекта одиночки, статическое поле хранит ссылку на первый созданный объект. Теперь при наследовании этого класса любой класс получит эту логику.

Теперь, после реализации в классе консоли метода для вывода текста, в любом месте кода можно отправить сообщение в консоль.

Так как важнейшим показателем оптимизации является FPS, был добавлен и счетчик FPS. Также для определения версии игры, загруженной на устройство, было решено добавить надпись с текущей версией игры.

Добавление встроенной в игру консоли позволило проводить тестирование на мобильном устройстве. В результате получилось настроить управление экранными стиками с помощью касаний пальцев, а не с помощью клавиатуры и мыши. При этом пришлось отказаться от управления клавиатурой и мышью, из-за чего тестирование приложения теперь возможно лишь на мобильном устройстве. Это значительно влияет на скорость тестирования нововведений.

V0.0.006

Теперь, когда удалось наладить приемлемую работу с джойстиком, следующей задачей становится добавление стрельбы, управляемой вторым джойстиком.

Во-первых, для этого необходимо добавить новые характеристики для игрока в классе PlayerStats\_SO, такие как: урон, время перезарядки, скорость пуль, текущее оружие. Во-вторых, для добавления стрельбы необходимо добавить следующие классы:

* Weapon. В игре планируется ввод различного оружия, из-за чего возникает необходимость в создании данного класса. В нем будет описываться параметры конкретного оружия. Для начала: спрайт, одна или несколько точек спавна пуль и их направление.
* Bullet. Объекты этого класса будут хранить значение о каждой пуле на сцене. При инициализации объекта ему задаются: урон, скорость полета, направление, стартовая позиция, время жизни, спрайт. После вызова метода Fire начинается движение согласно вложенной в объект логике.
* FireManager. Класс-singleton. Этот класс необходим т.к. возможность стрелять будет иметь не только персонаж, но и враги или ловушки. Этот класс получает необходимые для инициализации пули данные, создает её и запускает, вызывая её метод Fire. Пока нет врагов, для простоты реализации, метод Fire класса FireManager принимает 2 аргумента: точка спавна пули и её направление. Остальные данные берутся из класса PlayerStats\_SO.
* WeaponManager. Класс-singleton. Так как оружие будет только у игрока, этот класс реализует всю логику, связанную с изменением оружия, его перемещением, изменением его характеристик. Этот класс имеет методы для установки текущего оружия, а также для стрельбы из него. При вызове метода Fire этого класса будет выполнен вызов метода Fire класса FireManager столько раз, сколько точек спавна пуль есть у оружия. В качестве параметров будут переданы точки спавна пуль и их направления, после чего метод Fire класса WeaponManager нельзя будет вызвать, пока не пройдет перезарядка.

После реализации этих классов добавим метод Fire в класс PlayerController, который будет вызываться при событии, вызванном движением правого джойстика. Он будет поворачивать оружие в том же направлении, что и направление смещения стика. Пока производится касание правого джойстика вызываем метод Fire класса WeaponManager. Он соберет данные о текущем положении оружия и вызовет метод Fire класса FireManager. Класс FireManager соберет основные характеристики для пули (урон, скорость полета и т.д.), инициализирует новую пулю с этими характеристиками и вызовет для пули метод Fire, после чего пуля вылетит.



Рисунок 2. Схема запуска пули.

V0.0.007

После создания системы стрельбы, было решено добавить новый вид оружия, после чего сохранить различные виды оружия в префабы. Это необходимо, чтобы при смене оружия можно было удалить со сцены текущее и создать новое.

Создание нового оружия было начато с его спрайта и спрайта пуль. Т.к. новое оружие имеет 3 “дула”, то у него и 3 дочерних элемента, определяющих позицию спавна пуль и их направление полета. Для каждого оружия был выбран свой индекс (0, 1). Индекс текущего оружия записан в PlayerStats\_SO и после перезапуска будет взято последнее используемое оружие. В класс WeaponManager добавлен метод для смены оружия. Этот метод вызывается кнопкой на экране.

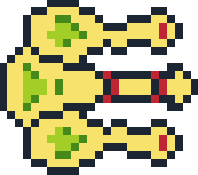
   

Рисунок 3. Спрайты оружий и пуль.

Теперь классы FireManager и WeaponManager получают индекс оружия из класса PlayerStats\_SO и на его основе устанавливают нужный спарайт пули и префаб оружия.

V0.0.008

В этой версии было решено создать общий пул для пуль, добавить способ проще создавать ландшафт на карте, достроить карту и добавить скрипт для автоматической установки значения Sorting Order для объектов.

В зависимости от усилений, собранных пользователем, и типов врагов, на карте может создаваться очень много выстрелов. Создание и удаление объектов – очень ресурсозатратный процесс, поэтому вместо создания и уничтожения множества одинаковых объектов используют пулы объектов.

**Пул объектов (Object Pool)** – некоторое хранилище схожих объектов. Основная идея заключается в том, что вместо создания и удаления объектов они включаются/отключаются.

Таким образом если в данный момент все пули активны, то создается новая пуля, если же есть не активные пули, они активируются. Когда пуля должна разрушиться она не уничтожается, а деактивируется. Вместе с этим в класс Bullet добавлен метод для изменения значений, а не полной инициализации. Так как за создание пуль отвечает класс FireManager, пул объектов реализован внутри этого класса.

Для более удобного создания поверхностей был написан скрипт, который устанавливает указанный префаб, пока он не закроет собой площадь X\*Y, в указанной точке, после чего генерирует вокруг себя коллайдер того же размера. Теперь можно создавать большие прямоугольные площадки с минимальными затратами времени. В результате достроить карту не составило проблем.



Рисунок 4. Увеличенная первая локация.

До сих пор после установки каждого элемента окружения приходилось отдельно устанавливать параметр SortingOrder. Если объектов окружения будет больше, то это подобная установка значений займет много времени. Для решения этой проблемы был написан скрипт, устанавливающий значение SortingOrder в зависимости от Y координаты в методе Start. Теперь чтобы очередь отрисовки корректно установилась для объекта необходимо лишь прикрепить к нему этот скрипт. Но несмотря на то, что данный способ сработал, изменял он очередь отрисовки только после запуска игры. Поэтому во время проектирования уровня объекты перекрывают друг-друга случайным образом, что мешает правильному восприятию локации.

Также в этой версии были добавлены коллайдеры для пуль. Ранее они пролетали сквозь другие объекты. Теперь же при столкновении с ними пули отправляются обратно в пул. Для того, чтобы для пуль обрабатывались только соприкосновения с другими объектами необходимо указать, что их коллайдеры являются триггерами. Иначе бы пули при попадании по другим коллайдерам смещали их. Также добавлена проверка по тегу на то, что объект, который задела пуля, не является игроком.

V0.0.009

Для правильного отображения объектов поверх друг-друга было решено использовать плиточную карту (tilemap).

Unity обладает хорошим набором инструментов для построения плиточных карт. Для этого необходимо преобразовать спрайты в плитки и создать сетку, указав размер ячеек и приоритетное направление отрисовки. После этого можно наносить плитки на сетку. В свойствах плитки можно указать, нужно ли создавать для неё коллайдер. В целях оптимизации к плиточной карте был добавлен Composite Collider 2D. Он объединяет множество коллайдеров в один. Для плиточных карт уровень отрисовки и очередь указываются не для каждой плитки, а для всей карты сразу, поэтому, если разместить на плиточной карте элементы окружения, они будут отображаться корректно относительно друг-друга, но персонаж будет либо всегда перед объектами окружения, либо всегда позади. Поэтому элементы окружения останутся отдельными объектами.

Для того, чтобы персонажа было видно, когда он заходит за элементы окружения был написан скрипт HideOnTrigger. При добавлении на любой объект этого скрипта и коллайдера с пометкой isTrigger, объект будет становиться полупрозрачным, когда его коллайдер пересечется с игроком. После добавления этого скрипта на деревья и добавления Edge Collider 2D (этот коллайдер является линией) они будут скрываться, когда персонаж за ними.



Рисунок 5. Полупрозрачное дерево.

V0.0.010

Теперь, когда персонаж может ходить и стрелять, а построение локаций стало достаточно простым, настало время для добавления врагов.

Первым врагом будет слизень. Для начала он должен хаотично передвигаться, получать урон от игрока и в конечном итоге умирать.



Рисунок 6. Спрайты слизня.

Для создания первого врага потребуется создать несколько скриптов:

* Enemy. В нем будут храниться основные характеристики врагов. Уровень, здоровье, урон, скорость, трение.
* EnemyMove. Этот класс будет иметь несколько различных методов для передвижения объекта. Иными словами, благодаря этому скрипту враги смогут перемещаться.
* TakeDamage. Этот класс будет улавливать попадания вредоносных объектов и обеспечивать корректную обработку получения урона.
* EnemyHealthbar. Этот скрипт отвечает за отображения уровня здоровья врагов.

Первое реализованное передвижение – передвижение с помощью случайных воздействий сил (импульсных или постоянных). Для этого задаются минимальное и максимальное время движения или стояния.

В классе TakeDamage создан метод OnTriggerEnter2D, который срабатывает, когда коллайдер, помеченный как триггер касается объекта с коллайдером без пометки триггера и компонентом Rigidbody 2D. Этот метод на вход получает коллайдер объекта, с которым произошло столкновение. Объект коллайдера может быть неявно преобразован к объекту класса GameObject (любой объект сцены). Поэтому к нему можно вызвать метод TryGetComponent(Bullet), который вернет истину, если к объекту прикреплен компонент (скрипт) Bullet. Если такой компонент существует, значит произошло столкновение с пулей, и мы можем получить её данные, например, урон. После этого вызовется метод Damage, в котором урон вычтется из здоровья врага и обработается любая другая логика, например, смерть врага или отталкивание. Также в этом методе мы вызовем собственное событие OnDamageRecieved, передав в него полученный урон. Слушателем этого события будет объект класса EnemyHealthbar. На основе полученного полоска здоровья врага уменьшится.

Чтобы враги появлялись в указанных местах были созданы скрипты EnemySpawnManager и SpawnEnemy. Первый скрипт является синглтоном. Он хранит префабы всех врагов, сортирует их по уровню и имеет метод для выдачи случайного врага определенного уровня. Скрипт SpawnEnemy имеет 2 открытых поля: минимальный и максимальный уровень врага. Этот скрипт обращается к скрипту EnemySpawnManager и спавнит врагов на месте каждого дочернего объекта с тегом Spawnpoint. Таким образом можно создать различные формы для спавна монстров.

Также была создана новая сцена с большой площадью. На ней будут спавниться монстры, и можно будет тестировать первые сражения.

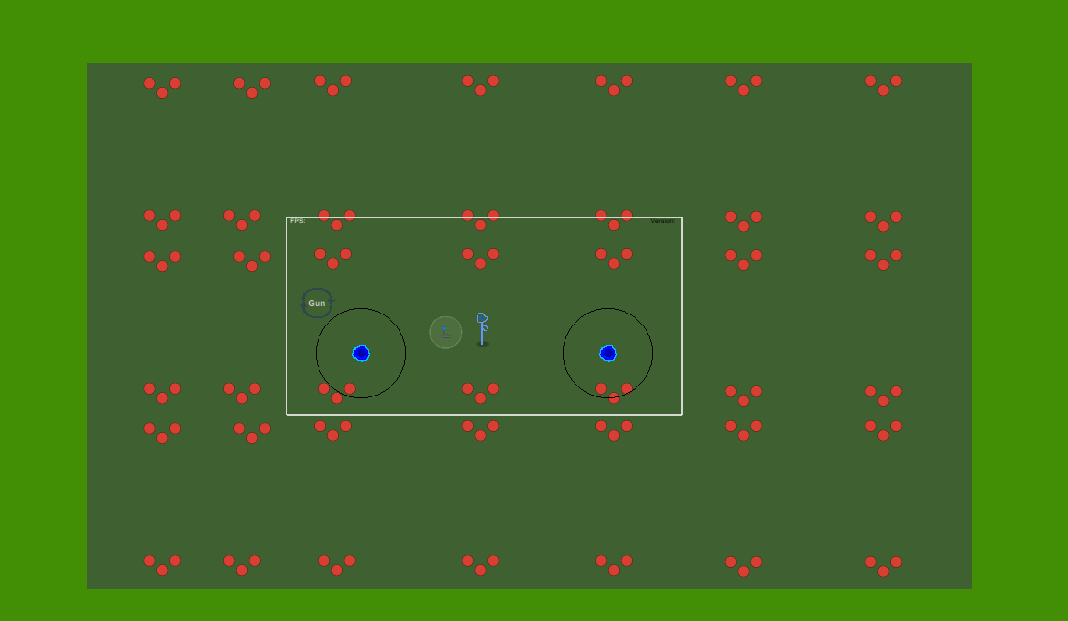


Рисунок 7. Локация с точками спавна монстров.

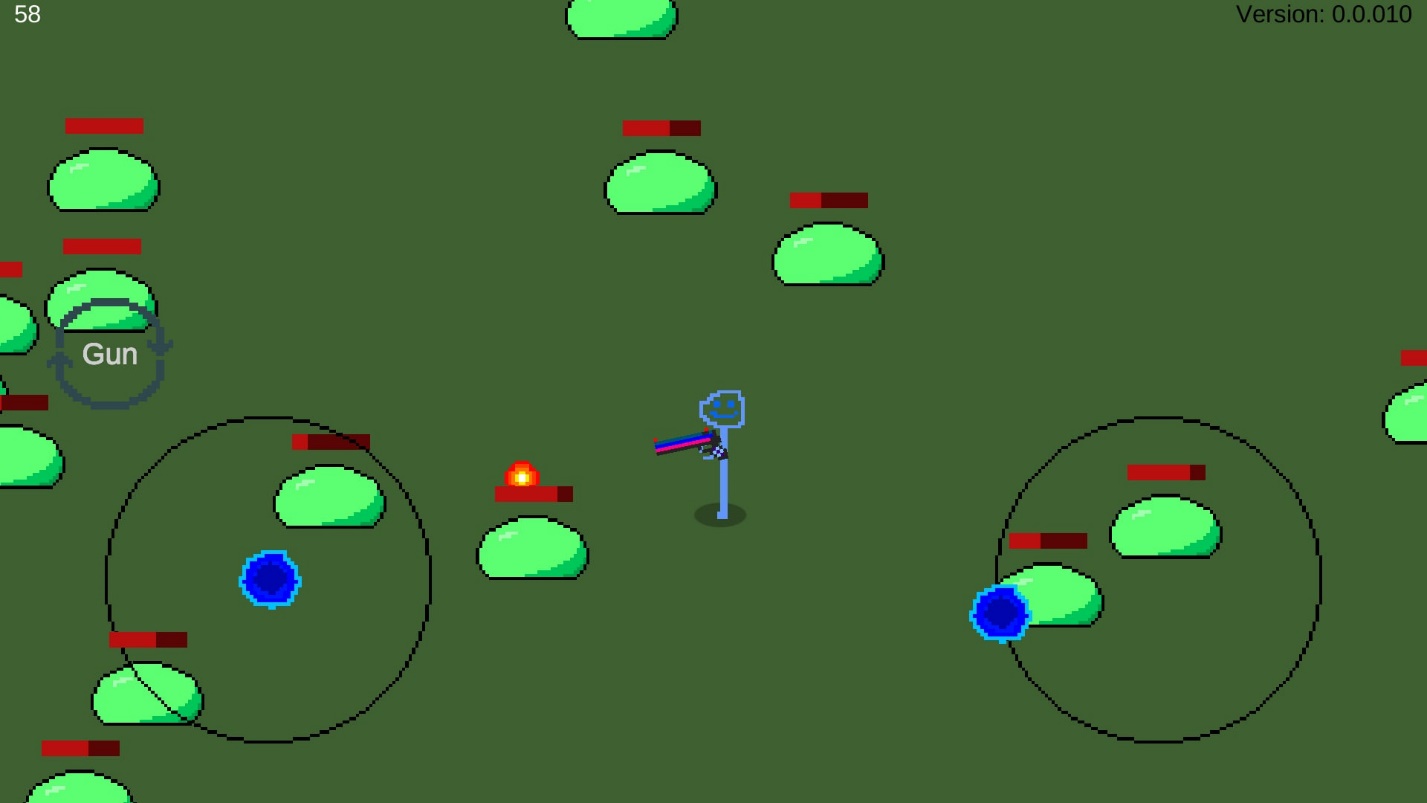


Рисунок 8. Монстры после запуска игры.

Дальнейшие действия в реализации проекте

Далее будут добавлены начальное меню, меню паузы, Game Manager (контролирует состояние игры и осуществляет переход между сценами), новый враг – скелет, первые предметы (подбор, сохранение в инвентарь), поддержка нескольких языков и система баффов-дебаффов с шейдерами для отображения эффектов.