**Содержание**

Описание проекта 2

Ход работы 2

Описание проекта

Проект – “Разработка мобильной игры в жанре 2D рогалик”. Целью проекта является получение навыков создания мобильных игр, а также навыков по работе с проектами, их поддержке и развитию. Не маловажно и получение навыков по работе с платформой Unity.

Дальнейшее описание не является полностью утвержденным проектом, некоторые функции и детали могут меняться, но данное описание позволит получить общее представление о проекте.

В результате работы должна получиться мобильная игра жанра рогалик. Основная цель игры – пройти множество уровней и победить финального босса. Игрок начинает игру в стартовом городе с начальным оружием, а после прохождения каждой локации получает предмет, влияющий на наго. Уровни представляют из себя некоторую ограниченную локацию с врагами на ней. После уничтожения всех врагов будет открыт проход на следующую локацию. Время от времени на локациях будет лишь один сильный враг – босс локации.

Игрок и враги имеют множество характеристик, таких как: уровень жизни, сила атаки, скорость бега, дальность атаки, сила регенерации здоровья, шанс и множитель критического урона, скорость перезарядки, скорость полета снарядов и многое другое. Все эти параметры будут изменяться в зависимости от врага или от усилений, собранных игроком.

Ход работы

В самом начале было просмотрено множество различных игр с целью определения приблизительного внешнего вида игры. Выбор пал на игру Blazing Beaks из-за её простоты.



Рисунок 1. Скриншот из игры Blazing Beaks.

Для подбора спрайтов нижнего и верхнего уровня земли было нарисовано множество различных изображений.

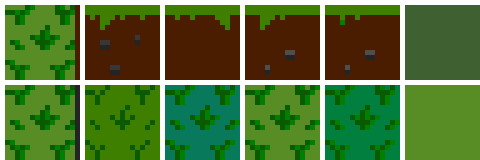


Рисунок 2. Первый набор полов и боковых стен.

Также было нарисовано дерево. Чтобы при наслоении друг на друга деревья оставались различимыми было решено обвести крону более темным цветом.

C:\Users\Dixas\Downloads\Tree2.pngC:\Users\Dixas\Downloads\derevo (1).png

Рисунок 3. Первые деревья.

При импорте изображений (спрайтов) в Unity, для того, чтобы изображения не теряли четкости при масштабировании, необходимо отключить фильтрацию и сглаживание, а для того, чтобы все пиксели на всех спрайтах были одного размера, было задано соответствие – 1 единица измерения мира равна 16 пикселям. Этот параметр также указывается при импорте изображений.

Чтобы проще подбирать сочетания нижних и верхних участков земли было решено сохранять все одинаковые объекты на сцене в префабы.

**Префаб (Prefab)** - это один из типов объектов, предназначенный для многократного использования, который можно вставлять несколько раз в одну и ту же сцену. При изменении параметра префаба изменяются и все объекты на сцене.

После создания префабов нижней земли (Ground), верхней земли (Highground) и дерева (Tree) их осталось лишь выставить на сцену. Для того, чтобы объекты корректно перекрывали друг друга при наложении (объекты, расположенные выше-левее, должны находиться “дальше”), необходимо создать несколько уровней рендеринга (Sorting Layer) и задать спрайтам свою очередь отрисовки (Order Layer). Для земли был создан уровень “Ground”. Для нижней земли установлена очередь 0, для верхней – 1, для теней – 2. Деревья вынесены в отдельный уровень – “Environment”. Этот уровень имеет больший приоритет, чем Ground и его объекты будут отображаться всегда поверх уровня Ground. В очередь отрисовки необходимо установить значение, обратное координате Y и домноженное на 10. Домножение необходимо для большей точности. Очередь отрисовки не может быть дробным числом.



Рисунок 4. Различные сочетания первой локации.

Было решено оставить пол монотонным, а также добавить под деревьями полупрозрачные тёмные круги, чтобы создать эффект тени.

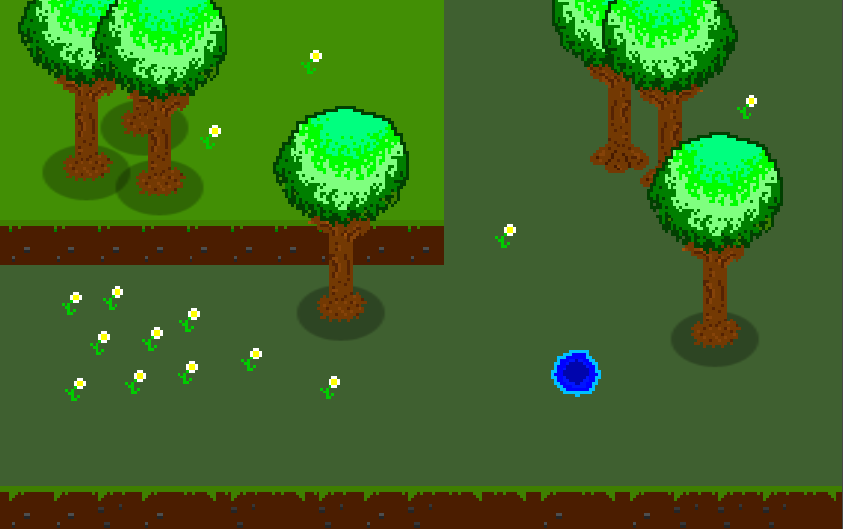


Рисунок 5. Итоговый вид локации.

После построения первого уровня следующей задачей стало добавление персонажа, а также добавление возможности управлять им и взаимодействовать с окружающими объектами, в первую очередь – сталкиваться.

Был нарисован простейший персонаж, а также анимация для него из четырех спрайтов, и первое оружие.

C:\Users\Dixas\Downloads\chel0040.png

Рисунок 6. Спрайты анимации ходьбы для первого персонажа.

Если выделить и перенести несколько спрайтов на сцену, то создастся анимация. Анимацию можно редактировать в окне Animation в Unity.

Для добавления персонажа на сцене был создан пустой объект Player. К этому объекту будут прикрепляться другие объекты, которые должны перемещаться или масштабироваться вместе с игроком. Также на этот объект будут прикрепляться компоненты/скрипты, связанные непосредственно с персонажем. Первыми прикрепленными объектами являются объекты:

* Main Camera. Этот объект является “глазами” пользователя. При перемещении игрока камера будет следовать за ним. Таким образом персонаж всегда будет по центру экрана.
* Sprite. На этот объект прикрепим компоненты Sprite Renderer (этот компонент отрисовывает спрайт на сцене) и Animator (этот компонент имеет контроллер анимаций, который определяет текущую анимацию объекта).
* Shadow. На этом объекте прикреплен компонент Sprite Renderer, отрисовывающий спрайт тени (такой же объект прикреплен к префабу дерева)
* WeaponSlot. Этот объект будет родительским для оружия. Он задает положение оружия относительно персонажа и задает точку поворота.



Рисунок 7. Иерархия персонажа

Теперь необходимо написать код для управления персонажем.

Все скрипты, прикрепляемые к объектам на сцене, наследуются от класса MonoBehaviour. Благодаря этому скрипты могут изменять свойства компонентов на объекте.

При написании кода мы будем часто использовать 3 метода:

* Update – вызывается каждый кадр, пока объект активен.
* Start – вызывается до метода Update, один раз, при первом появлении объекта на сцене.
* Awake – вызывается до метода Start, при пробуждении объекта.

Зачастую метод Awake используется для установки начальных значений текущего объекта, которые могут быть установлены без участия других объектов. В методе Start устанавливаются значения, зависящие от значений других объектов.

Методы вызываются для всех активных объектов на сцене в случайном порядке. Чтобы установить определенный порядок вызова необходимо изменить очередь выполнения необходимого скрипта в настройках приложения.

Перед написанием скрипта управления персонажем необходимо выбрать одну из двух систем пользовательского ввода. Существует старая и новая системы. Основное отличие в том, что в старой системе для определения пользовательского ввода необходимо в каком-либо месте в коде обратиться к классу Input, чтобы узнать о каком-либо действии пользователя. Например, если необходимо выполнить действие, как только пользователь нажмет клавишу “W”, то потребуется каждый кадр в коде вызывать if (Input.GetKeyDown(KeyKode.W)) {…}. Если определять действия пользователя подобным образом, то множество подобных вызовов могут привести к затормаживанию приложения. Также подобный код плохо читаем и не удобен в использовании. Значительным плюсом старой системы является наличие богатой документации, а также множество статей с решением различных проблем, возникающих при работе с системой.

Новая система ввода следует событийно-ориентированной парадигме программирования. В этом случае при нажатии на кнопку вызывается событие, а слушатели этого события могут отреагировать на него каким-либо образом. Код становится более читаемым и гибким. Для этой системы ввода есть официальная документация на английском. Так как система новая, то статей по ней очень мало, на русском языке их почти нет.

Так как целью проектной работы является получение новых навыков по разработке игр, было решено выбрать новую систему ввода.

Для простоты тестирования игры было решено сначала реализовать управление с помощью клавиатуры и мыши. Чтобы для игрока обрабатывалась физика (по осям x и y) добавим к нему компонент Rigidbody 2D, при этом необходимо отключить обработку поворота персонажа по оси Z. Создадим свой скрипт PlayerController. Этот скрипт будет отвечать за передвижения игрока. Здесь создадим метод, который будет вызываться событием перемещения от системы ввода. В этом методе мы получаем 1 параметр – объект, хранящий информацию о произошедшем событии. Из этого объекта получим значение типа Vector2, содержащий x и y значения – направление перемещения. Умножая эти значения на какие-либо константы (скорость персонажа) получим силу передвижения по осям x и y. Прикладывая эту силу с помощью компонента Rigidbody 2D к объекту, заставим персонажа двигаться. Также необходимо менять очередь отрисовки для персонажа при передвижении по оси Y.

Создадим метод, вызываемый событием передвижения мыши. В нем добавим поворот персонажу. Если мышь в правой части экрана, персонаж повернут в право, если в левой – в лево.

Для управления анимациями создадим скрипт PlayerAnimationController. Перехватывая событие перемещения запускаем анимацию Move, если значения, полученные от пользователя не нулевые, иначе запускаем анимацию Idle (анимация Idle пока состоит из одного спрайта).

Для сохранения параметров персонажа создадим скрипт PlayerStats\_SO, наследуемый от класса ScriptableObject вместо MonoBehaviour. Такой скрипт нельзя прикреплять к объектам, но можно создавать пакеты ресурсов, хранящие описанные в нем данные. Их значения можно использовать и изменять, обращаясь к ним из других скриптов. Также измененные значения сохранятся после перезапуска игры. В этом скрипте создадим два поля: speed и drag, отвечающие за скорость персонажа и за трение соответственно.

Добавим к объекту Player скрипт PlayerController, а к его дочернему объекту Sprite скрипт PlayerAnimationController. Создадим первый ресурс с PlayerStats\_SO и пропишем в нем скорость и трение для персонажа и настроим взаимодействие скрипта PlayerController с этим ресурсом.

Теперь есть локация и персонаж, который может по ней перемещаться. Осталось добавить столкновения. Т.к. для определения столкновения сложных моделей нужна большая вычислительная мощность, для определения столкновений пользуются примитивными фигурами, такими как эллипс (круг) и прямоугольник (квадрат). В Unity роль физических границ тела выполняют коллайдеры.

**Коллайдеры** — это компоненты, которые обеспечивают обнаружение столкновений с использованием их различных «ограничивающих рамок».

Т.к. персонаж должен сталкиваться с окружающими объектами только небольшой областью вокруг ног, добавим к объекту Player на сцене компонент Capsule Collider 2D и расположим его на уровне ног. Для персонажа используется капсула, а не прямоугольник для того, чтобы при прохождении вокруг угловых объектов на сцене персонаж не цеплялся за углы, а плавно обходил. Так как мы перемещаем персонажа с помощью действия на него силы, то после добавления компонентов Box Collider 2D к префабам верхнего пола, боковых стен и деревьев, персонаж начнет сталкиваться с ними. Для улучшения производительности все клетки пола и боковых стен были прикреплены к одному родительскому объекту, который имеет один коллайдер. Таким образом физика не будет обрабатываться для каждой клетки отдельно, что ускорит расчет каждого кадра.

Рисунок 8. Коллайдеры различных объектов.

В результате первого этапа практической работы была создана пробная локация и персонаж, способный перемещаться по ней и сталкиваться с препятствиями.

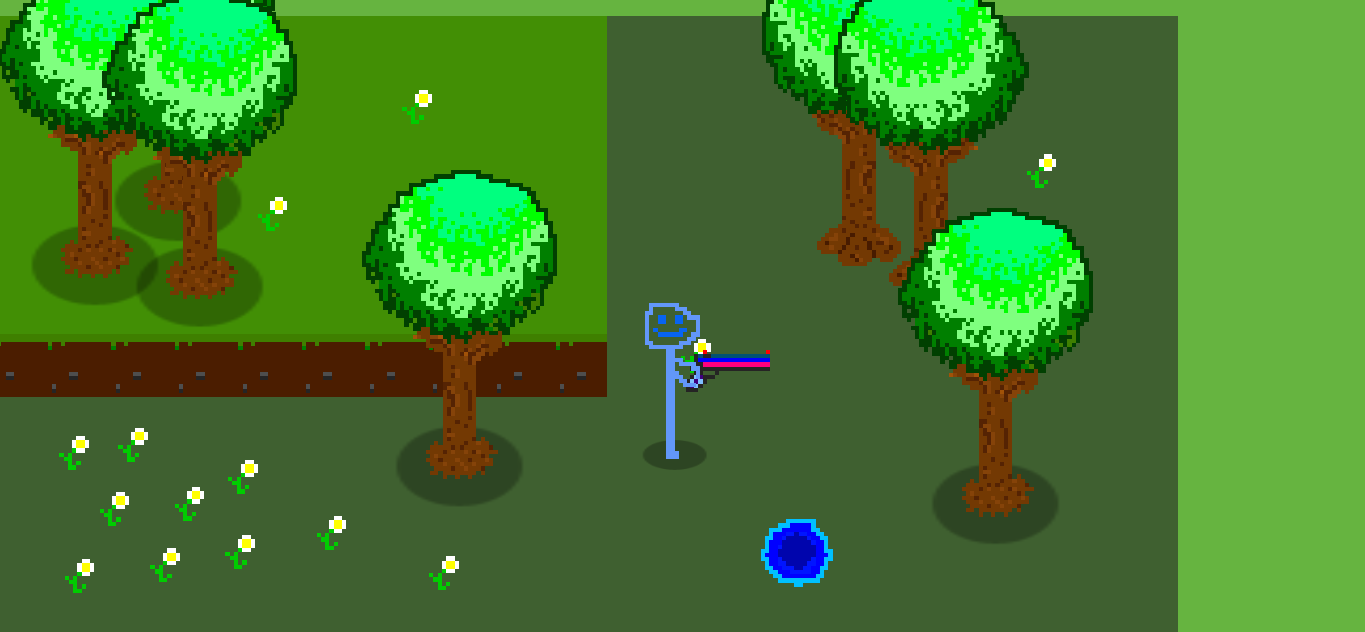


Рисунок 9. Итог выполнения первого этапа.