Министерство науки и высшего образования Российской Федерации

Федеральное государственное автономное образовательное учреждение

высшего образования

«КАЗАНСКИЙ (ПРИВОЛЖСКИЙ) ФЕДЕРАЛЬНЫЙ УНИВЕРСИТЕТ»

ВЫСШАЯ ШКОЛА ИНФОРМАЦИОННЫХ ТЕХНОЛОГИЙ

И ИНТЕЛЛЕКТУАЛЬНЫХ СИСТЕМ

Направление подготовки: 09.03.04 – Программная инженерия

Профиль: Технологии разработки информационных систем

КУРСОВАЯ РАБОТА

Разработка компьютерной ролевой игры

Студент 2 курса

Группы 11-708

«\_\_\_\_»\_\_\_\_\_\_\_ 201\_ г. подпись (ФИО)

Научный руководитель

\_\_\_\_\_\_\_\_\_\_\_, \_\_\_\_\_\_\_\_\_\_\_\_\_

уч. степень уч. звание

«\_\_\_\_» \_\_\_\_\_\_\_201\_ г. подпись (ФИО)

Казань - 2019 г

**Содержание**

Введение 2

Общая тематика 3

Механики 3

Механика передвижения 3

Взаимодействие объектов 5

Механика атаки и стрельбы 9

Перезарядка и отображение патронов 12

Управление здоровьем и его отображение 14

Управление UI 17

Информация о предметах UI 18

Инвентарь 20

Основной функционал игрока 22

Основной функционал враждебного нпс 23

Характеристики персонажа и врагов 24

Анимации 25

Камера 26

Design 27

Источники 28

**Введение**

Как можно заметить в двадцать первом веке в сфере развлечений очень-очень сильно и быстро развивается направление видеоигр. Смотря на спрос и предложения, можно заметить, что конкуренция огромна, а конкуренция порождает большое развитие технологий, идей, событий. Но из-за того, что индустрия велика, также создается большое количество некачественного контента

Индустрия растет и влечет за собой невообразимый поток не только программистов, но и людей других специальностей: звукорежиссеров, 3D-моделлеров, сценаристов, бухгалтеров, менеджеров и т.д., так что игры в современном мире делаются за счет большого количества часов непрерывной работы десятков, а порой и сотен людей! Из этого можно сделать вывод, что создания даже небольшой игры требует много знаний, времени и терпения.

В этой курсовой работе рассматривается создание функционала для игры time-killer`а на одного игрока, разбор логики работы и дополнения функций проекта разными примочками.

Видеоигра была создана на основе платформы Unity 2018.3.2f1 и языка программирования C#. Также были задействованы другие программы: Adobe Photoshop CS6, Substance Painter, Autodesk Maya 2019

**Общая тематика**

Основные типы для общей тематики игры:

1. Single player
2. Time-killer
3. 3D
4. Isometric camera projection
5. Non target shooter

**Механики**

**Механика передвижения**

Большое количество времени в игре занимает передвижения персонажа по локации для разных целей: поиск предметов, избежания врага или нападение на него.

Этого можно достичь благодаря функциональности компонента NavMesh. Было создано несколько соответствующих методов для настройки и управления данной механики.

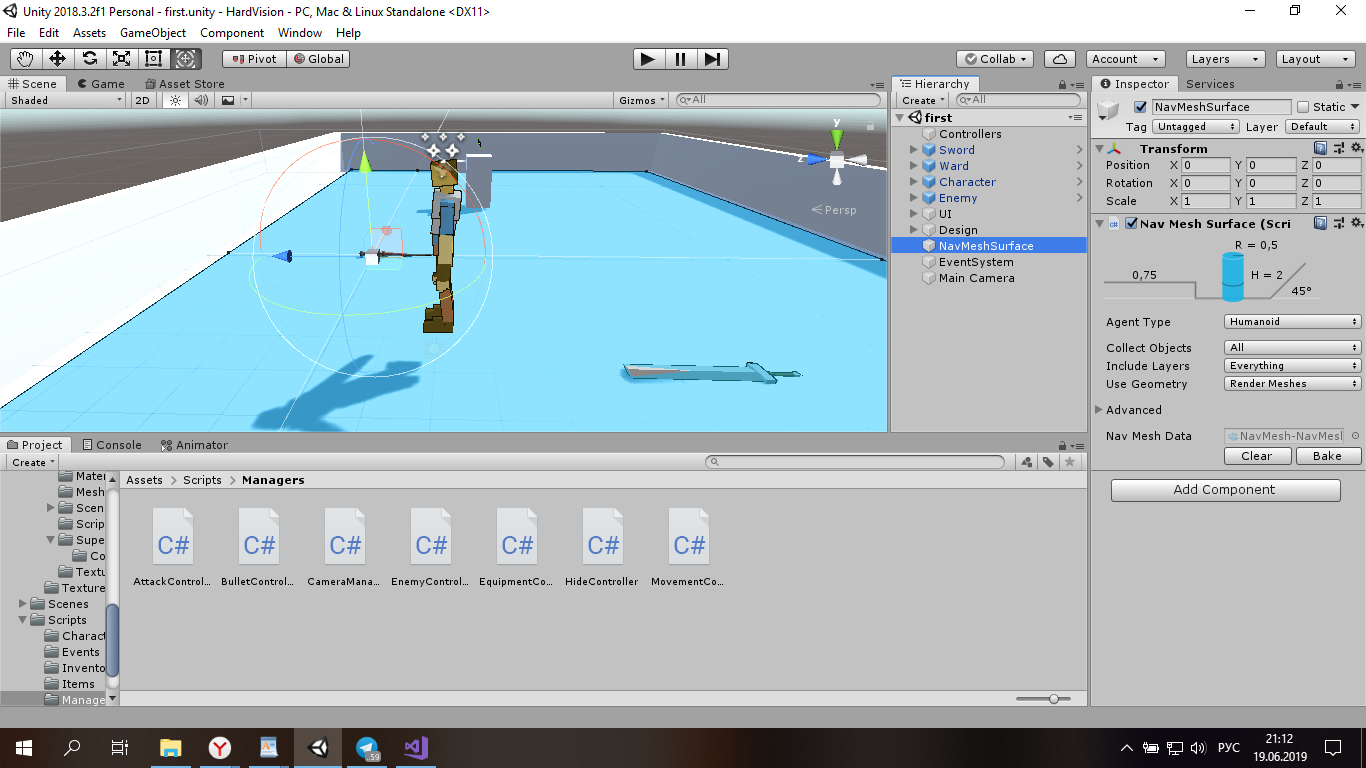
Так для персонажа было создан скрипт Movement Controller с методом Agent Move By Click. Данный метод вызывается, когда нужно передать какую-то точку на карте и переместиться (Рисунок 1)

Рисунок 1. Синее пространство для перемещения игрока

Сам скрипт выглядит непосредственно так:

public Vector3 AgentMoveByClick(Transform moveObj, Vector3 targetPos)

{

RaycastHit hit;

Ray ray = Camera.allCameras[0].ScreenPointToRay(Input.mousePosition);

if (Physics.Raycast(ray, out hit))

{

targetPos.x = hit.point.x;

targetPos.y = moveObj.position.y;

targetPos.z = hit.point.z;

}

return targetPos; }

То есть по нажатию мыши можно узнать расположение точки куда нужно двигаться и после возвратить направление и направить туда персонажа.

Учитывая, что это non target shooter, то для враждебных объектов не используется сложная функциональность. Враг либо стоит и стреляет в игрока, либо убегает от него. Но об этом виде передвижения подробнее в разделе взаимодействия объектов на сцене.

**Взаимодействия объектов**

Смысловая и логическая нагрузка в реализации сосредоточена на взаимодействии объектов на сцене между собой. К примеру игрок может взаимодействовать с оружием на сцене, может взаимодействовать с враждебными нпс.

Базовый скрипт Interactable несет следующую идею:

1. Написан для всех объектов, с которыми взаимодействует игрок
2. Имеет методы, определяющие находится ли объект в фокусе или нет
3. Содержит виртуальный метод переопределения Interact

void Update()

{

if (isFocus && !hasInteracted)

{

var distanceToPlayer = Vector3.Distance(player.position, interactionTransform.position);

if (distanceToPlayer <= radius)

{

Interact();

hasInteracted = true;

}

}

}

public virtual void Interact()

{

Debug.Log("Interact: " + transform.name);

}

public void OnDefocused()

{

isFocus = false;

player = null;

hasInteracted = false;

}

public void OnFocused(Transform playerTransform)

{

isFocus = true;

player = playerTransform;

hasInteracted = false;

}

Смотря на реализацию в проекте можно заметить, что скрипт Interactable наследует скрипт PickupItem, с помощью которого игрок может поднимать предметы

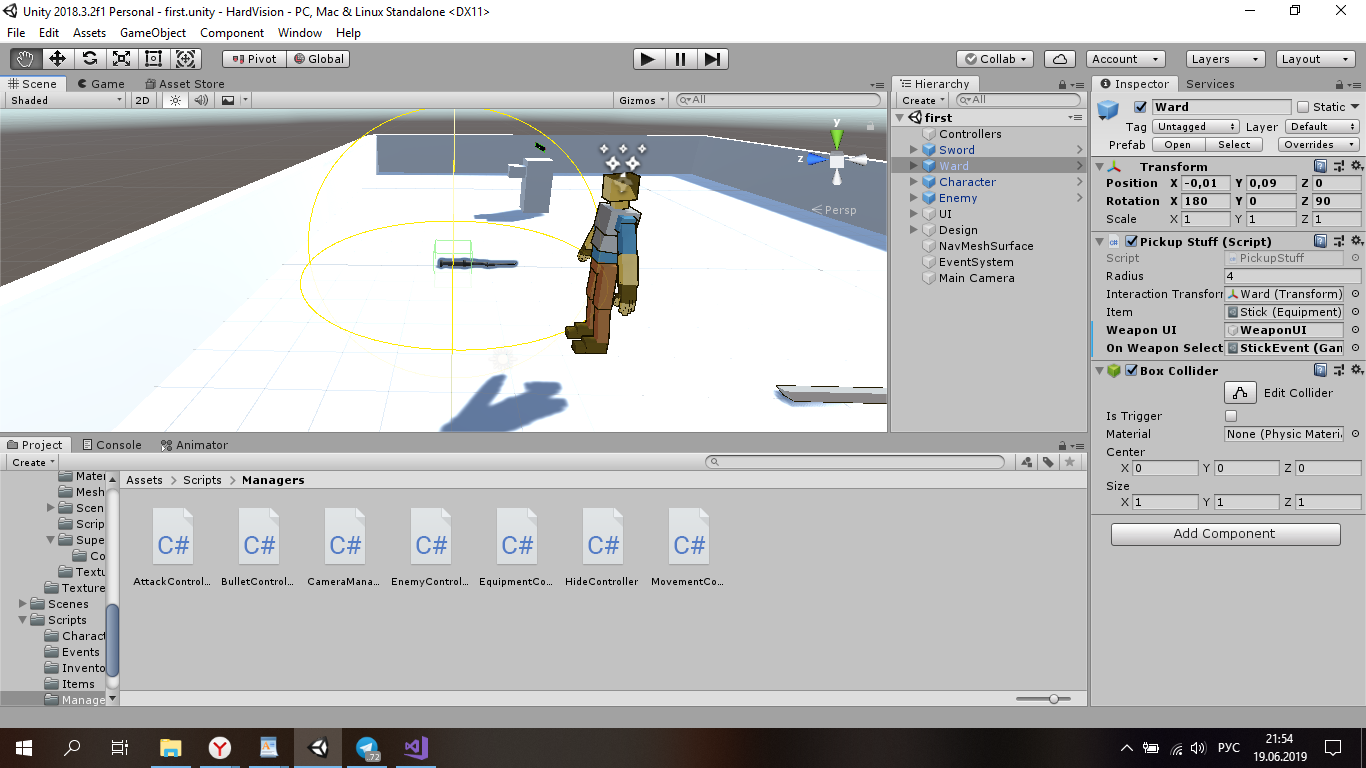


Рисунок 2. Желтый радиус, позволяющий поднять оружие.

Так выглядит типичный объект для поднятия. В скрипт передаются:

1. Радиус, на котором предмет можно поднять(Рисунок 2)
2. Координаты расположения объекта
3. Скриптабл объект, хранящий информацию об объекте
4. Интерфейс, отображающий информацию.
5. Скриптабл объект для отображения с помощью ивента

Непосредственно код, реализующий отображение информации об объекте и его поднятие с добавлением в инвентарь.

public override void Interact()

{

base.Interact();

PickUp();

}

private void OnMouseDown()

{

OnWeaponSelected.Raise();

WeaponUI.SetActive(true);

}

private void PickUp()

{

var wasPickedUp = InventoryController.instance.Add(item);

if (wasPickedUp)

Destroy(gameObject); }

Пример отображения информации о предмете после нажатия левой кнопки мыши по нему(Рисунок 3):

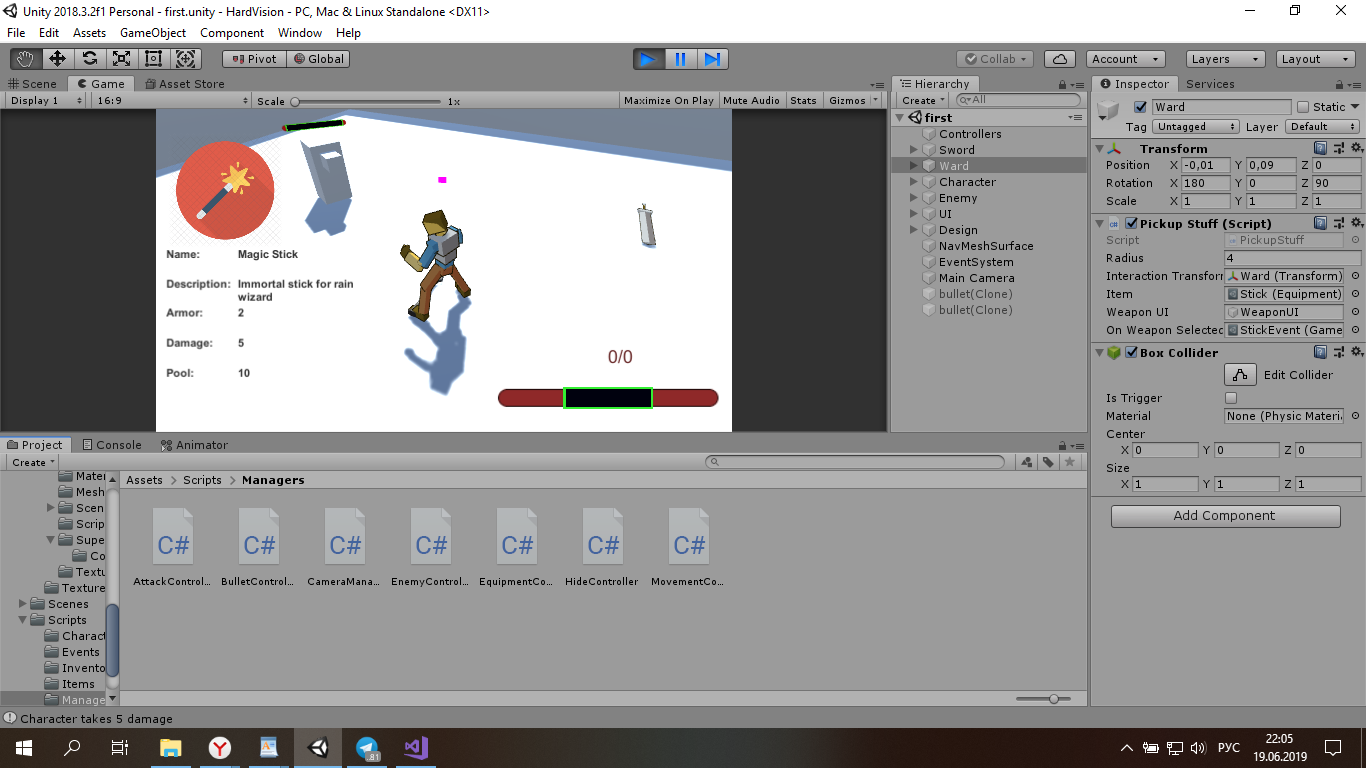


Рисунок 3. Отображение информации о предмете

Возвращаясь к функционалу враждебного нпс, следует обратить внимание на скрипт передвижения:

Данный скрипт позволяет убегать от игрока, если тот находится в радиусе действия. Учитывая, что действие происходит в методе Update, то можно понять, что алгоритм просчитывает путь отступления каждый кадр:

void Update()

{

if (isFollowed)

{

RunAway();

}

distanceToPlayer = Vector3.Distance(transform.position, Player.position);

if (distanceToPlayer < lookRadius)

{

isFollowed = true;

}

}

void RunAway()

{

isFollowed = false;

var playerDirection = transform.position - Player.position;

var newDestination = transform.position + playerDirection;

Agent.SetDestination(newDestination);

}

**Механика атаки и стрельбы**

Еще одной из возможностей игры является стрельба. Всё просто: персонаж стреляет во врагов, а они в свою очередь в него. Чтобы решить данную задачу потребовалось создать скрипт AttackController, который может вызываться из тела игрока и противника.

Игрок может пользоваться несколькими видами атаки:

1. Стандартный пинок, для которого не нужно наличия оружия
2. Атака дальнего боя. Данную атаку игрок может использовать при наличии соответствующего оружия с типом дальнего боя
3. Атака ближнего боя. Данную атаку можно использоваться, если имеется оружие с типом ближнего боя.

Тело метода, которое осуществляет выстрел с помощью оружия дальнего боя:

public void Shot(GameObject bullet, Transform spawnPos, CharacterStats stats, float speed = Speed)

{

var currentBullet = PoolController.SharedInstance.GetPooledObject("Bullet");

if (currentBullet != null)

{

currentBullet.transform.position = spawnPos.transform.position;

currentBullet.transform.rotation = spawnPos.transform.rotation;

currentBullet.GetComponent<BulletController>().myStats = stats;

currentBullet.SetActive(true);

currentBullet.GetComponent<Rigidbody>().velocity = currentBullet.transform.forward \* speed;

}

}

С помощью Pool контроллера нужно принять объект пули, задать соответствующие свойства и отправить ее в полет к нужному объекту, чтобы тот в свою очередь уже совершал какие-то действия из-за действия Collision объекта пули с другим объектом(игрок, враг)

Стандартный пинок и атака ближнего боя осуществляются с помощью скрипта MeleeAttacker:

private void Update()

{

if (Input.GetKey(KeyCode.G))

{

isAttack = true;

anim.SetBool("LegAttack", true);

}

if (Input.GetKeyUp(KeyCode.G))

{

isAttack = false;

anim.SetBool("LegAttack", false);

}

if (Input.GetKey(KeyCode.T))

{

isAttack = true;

anim.SetBool("MeleeAttack", true);

}

if (Input.GetKeyUp(KeyCode.T))

{

isAttack = false;

anim.SetBool("MeleeAttack", false);

}

}

void OnCollisionEnter(Collision collision)

{

if (isAttack)

if (collision.gameObject.tag == "Enemy")

{

collision.gameObject.GetComponent<HealthController>().TakeDamage(playerStats.MeleeDamage.GetValue());

collision.gameObject.GetComponent<EnemyController>().onHitTaken.Invoke();

gameObject.SetActive(false);

}

}

Всё просто, нажать на клавишу, активировать булеву переменную, активировать анимацию и ждать реакции метода OnCollisionEnter.

После взаимодействия данных механик с объектами осуществляется другая функциональность под названием менеджер по управлению здоровьем. Данная функциональность содержит несколько механик: отслеживание здоровья у текущего объекта и отнятие здоровья у объекта с помощью коллизии с другим объектом.

**Перезарядка и отображение патронов**

При создании более реалистичной физики для оружия был придуман соответствующий магазин для оружия с перезарядкой.

При использовании оружия магазин имеет следующие параметры:

1. Ограниченное количество патронов
2. Перезарядку с задержкой на некоторое время. Т.е персонаж не может стрелять в этот момент из-за булевого состояния.

Так в методе Update скрипта PlayerHubConroller проверяется условие нажатия кнопки, наличия патронов и булевого состояния перезарядки

if (Input.GetKeyDown(KeyCode.F) && EquipmentController.instance.currentEquipment[5] != null && canShoot)

{

attackController.Shot(bullet, bulletPos, PlayerStats.instance);

EquipmentController.instance.bullets++;

UIController.instance.Refresh();

if (EquipmentController.instance.bullets == EquipmentController.instance.currentEquipment[5].Magazine)

{

canShoot = false;

UIController.instance.SwitchReload();

StartCoroutine(Reload());

}

}

Отображение состояний происходит с помощью UI на сцене и скрипта UIController. Были реализованы следующие методы:

public void UpdateUI()

{

if (isReloading)

ReloadText.text = "Reloading";

else

ReloadText.text = $"{currentBullets}/{maxBullets}";

}

public void Refresh()

{

var rangedWeapon = EquipmentController.instance.currentEquipment[5];

if (rangedWeapon != null)

maxBullets = rangedWeapon.Magazine;

currentBullets = maxBullets - EquipmentController.instance.bullets;

UpdateUI();

}

public void SetDefaultState()

{

maxBullets = 0;

currentBullets = maxBullets;

UpdateUI();

}

public void SwitchReload()

{

isReloading = !isReloading;

UpdateUI();

}

Так в зависимости от состояния IsReloading в методе UpdateUI может отображаться перезарядка или количество патронов в магазине.

Метод Refresh обновляет количество патронов в магазине.

SetDefaultState устанавливает стандартное состояние

SwitchReload меняет состояние перезарядки

**Управление здоровьем**

Одно из условий завершения игры является полная потеря здоровья. Так с целью отслеживания здоровья в процессе игры было написано несколько скриптов для подсчета и отображения.

Скрипт HealthController отвечает за подсчет здоровья объекта, к которому привязан.

void Awake()

{

healthWidth = healthBar.sizeDelta.x / stats.maxHealth;

}

void Update()

{

if (stats.CurrentHealth < stats.maxHealth)

Regenerate();

}

public void TakeDamage(float damageAmount)

{

stats.TakeDamage(damageAmount);

healthBar.sizeDelta = new Vector2(healthBar.sizeDelta.x - healthWidth \* damageAmount, healthBar.sizeDelta.y);

if (healthBar.sizeDelta.x <= 0)

gameObject.SetActive(false);

}

public void Regenerate()

{

stats.Regenerate();

healthBar.sizeDelta = new Vector2(healthBar.sizeDelta.x + stats.regeneration.GetValue(), healthBar.sizeDelta.y);

}

Так метод Awake отображает в соотношении количество текущего здоровья к максимальному.

Update вызывает метод регенерации, если здоровье ниже максимального

TakeDamage вызывается при коллизии с другими объектами, которые могут отнимать здоровье при совершении коллизии.

Скрипт BulletController является одним из источников, с помощью которого наносится урон при коллизии:

public void OnCollisionEnter(Collision col)

{

if (col.gameObject.tag == "Character")

{

col.gameObject.GetComponent<HealthController>().TakeDamage(myStats.damage.GetValue());

col.gameObject.GetComponentInChildren<PlayerHubController>().onHitTaken.Invoke();

gameObject.SetActive(false);

}

if (col.gameObject.tag == "Enemy")

{

col.gameObject.GetComponent<HealthController>().TakeDamage(myStats.damage.GetValue());

col.gameObject.GetComponent<EnemyController>().onHitTaken.Invoke();

gameObject.SetActive(false);

}

gameObject.SetActive(false);

}

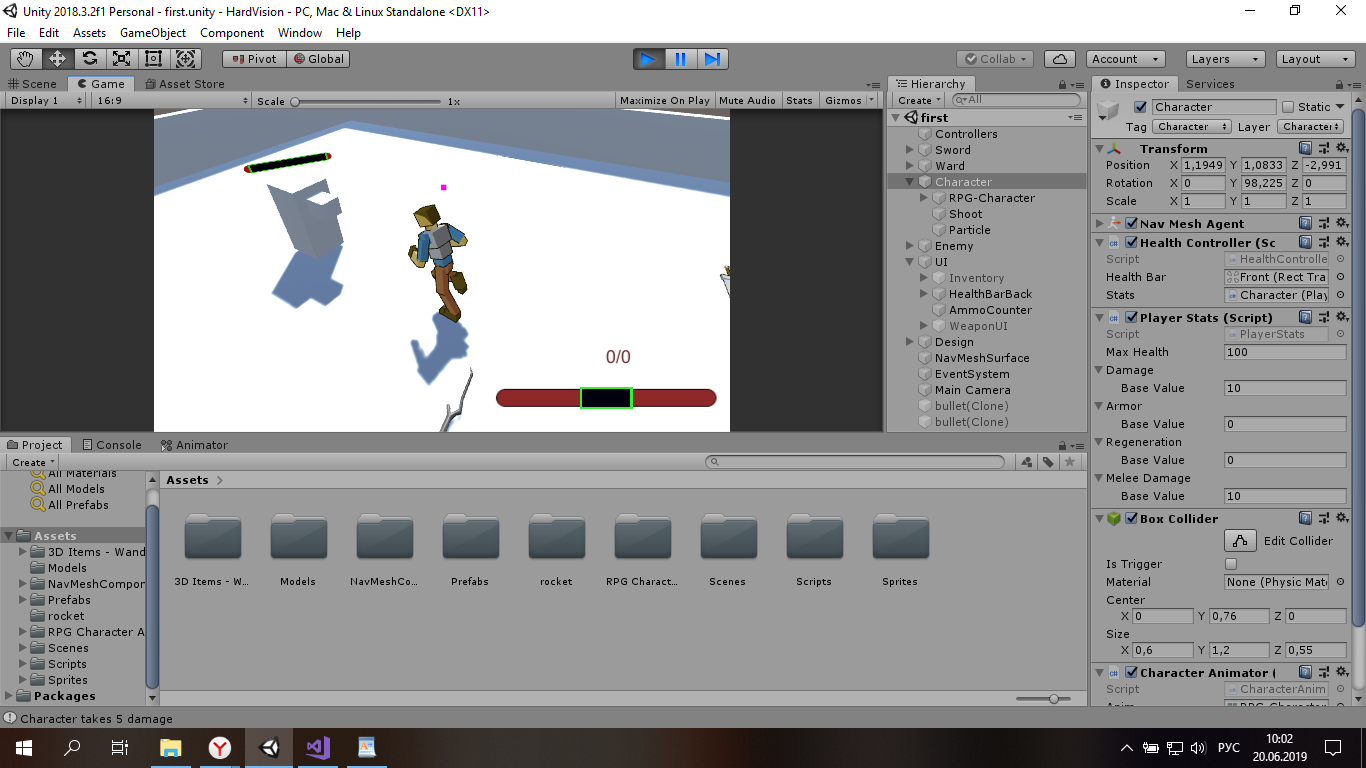
Отображение здоровья происходит с помощью соответствующих UI компонентов. Так у игрока здоровье отображается в правом нижнем углу экрана, а у враждебных нпс над телом объекта на сцене.(Рисунок 4)

Рисунок 4. Отображение здоровья у персонажа и врага

В данном случае видно, как выглядит неполное здоровье персонажа и полное здоровье врага.(Рисунок 4)

**Управление UI**

Управление UI строится из множества разных компонентов игры. Так например с помощью левой кнопки мыши можно вывести информацию о предмете, если кликнуть на него.

Также существует отображение здоровья с помощью UI, как было показано выше.

Присутствует отображение количества патронов в магазине.

Последний из видов отображения является отображение инвентаря и объектов в нем.

Все из представленных UI отображаются с помощью разного функционала, который может скрывать, показывать и изменять отображаемое.

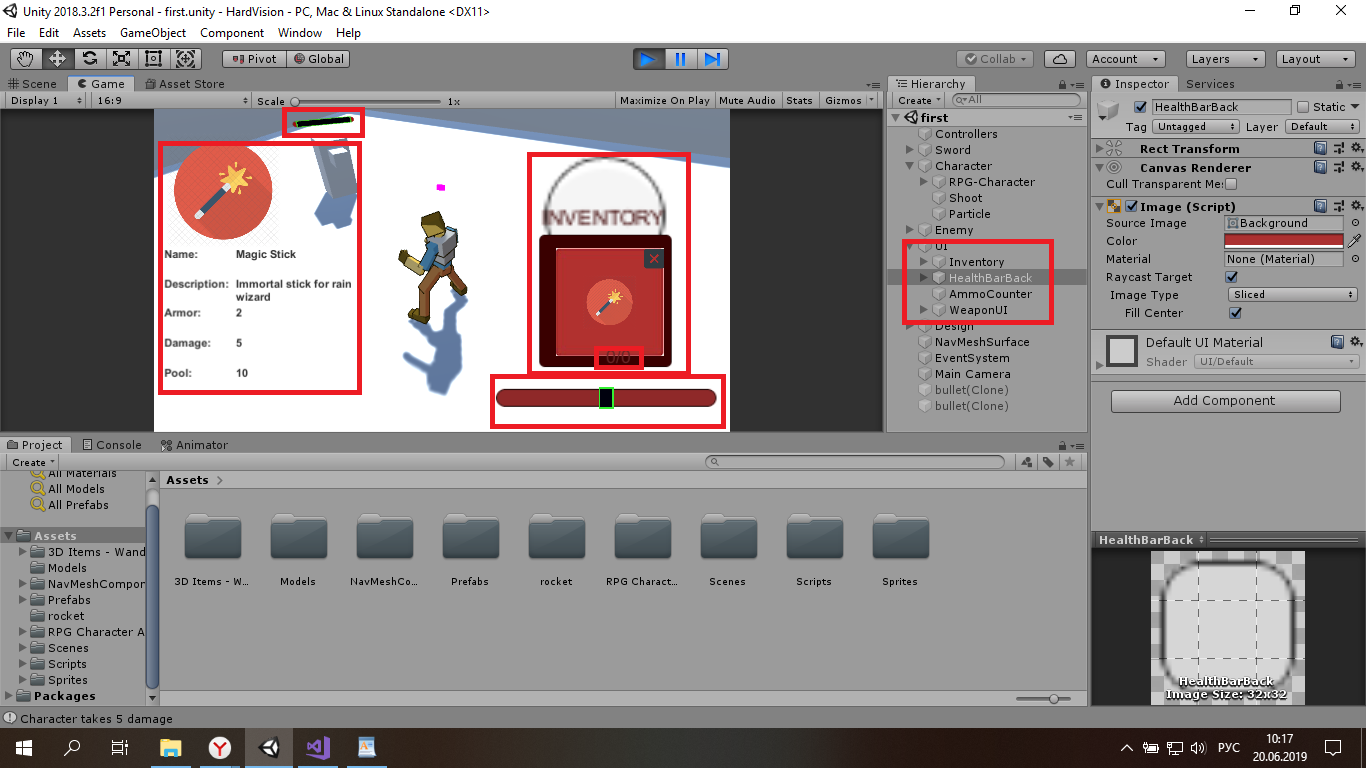
****

Рисунок 5. Отображение UI

Так на данной картинке представлены все виды UI присутствующие в игре.

**Информация о предметах UI**

Учитывая, что многие виды UI уже были рассмотрены, предоставим вниманию оставшиеся. Одним из них является информация о предметах.

Для чего это сделано? Каждый игрок жаждет заполучить оружие получше и при этом отсеять оружие похуже. С этой целью было создан функционал просмотра характеристик каждого оружия. Каждое оружие хранит скриптабл объект, с помощью которого можно вытащить информацию в UI с помощью скрипта.

Так данный объект хранит несколько свойств:

1. Имя объекта
2. Описание
3. Урон
4. Броня
5. Количество заряда в магазине

Так всю информацию об объекте можно вытащить с помощью трёх скриптов:

1. Item Merchant
2. Game Event
3. Game Event Listener

Первый скрипт реализует присвоение информации из объекта в UI:

public void UpdateDisplayUI(Equipment swordData)

{

itemName.text = swordData.name;

description.text = swordData.Description;

icon.sprite = swordData.icon;

armorModifier.text = swordData.ArmorModifier.ToString();

damageModifier.text = swordData.DamageModifier.ToString();

magazine.text = swordData.Magazine.ToString();

}

Второй в свою очередь управляет Game Event Listener. Добавляет или удаляет объект из списка. Также можно запустить все ивенты из списка.

Рассмотрим код изнутри:

public void Raise()

{

for (int i = listeners.Count - 1; i >= 0; i--)

{

listeners[i].OnEventRaised();

}

}

public void RegisterListener(GameEventListener listener)

{

listeners.Add(listener);

}

public void UnregisterListener(GameEventListener listener)

{

listeners.Remove(listener);

}

GameEventListener осуществляет похожий функционал, но уже на уровне GameEvent.

private void OnEnable()

{

gameEvent.RegisterListener(this);

}

private void OnDisable()

{

gameEvent.UnregisterListener(this);

}

public void OnEventRaised()

{

response.Invoke();

}

**Инвентарь**

Одна из главных механик данного проекта является хранение предметов в инвентаре и их извлечение при необходимости. Предметы могут добавляться в инвентарь, как из внешних ресурсов, таких как найти оружие на сцене, так и при снятии экипировки. Отображение инвентаря также реализуется с помощью открытия и скрытия UI.

Для этого был написан скрипт Equipment в котором хранится информация о:

1. Тип экиперуемого предмема
2. Его объект
3. Позиция и угол поворота
4. Описание
5. Количество урона
6. Количество защиты
7. Запас магазина

При использовании в инвентаре вызывался данный метод:

public override void Use()

{

base.Use();

EquipmentController.instance.Equip(this);

RemoveFromInventory();

}

Таким образом объект был экипирован на тело персонажа и удалялся из инвентаря.

Тем не менее, устроен сам инвентарь немного иначе. Реализация состоит из трех скриптов:

1. InventoryController
2. InventorySlot
3. InventoryUI

Контроллер отвечает за количество место в инвентаре, добавление и удаление.

public bool Add(Item item)

{

if (!item.isDefaultItem)

{

if (items.Count >= Size)

{

Debug.Log("Not enough space");

return false;

}

items.Add(item);

OnItemChangedCallBack?.Invoke();

}

return true;

}

public void Remove(Item item)

{

items.Remove(item);

OnItemChangedCallBack?.Invoke();

}

Слот отвечает за взаимодействие с объектом в инвентаре: взять предмет или же удалить его из инвентаря.

И скрипт InventotyUI следит за отображением всего происходящего.

Так с помощью этих функций игрок может забрать предмет в инвентарь, экипировать его, снять, удалить из инвентаря. Очень удобный функционал для смены экипировки под разные игровые ситуации.

**Основной функционал персонажа**

Скрипт, отвечающий за все основные активности персонажа, находится в префабе RPG-Character. В нем сосредоточены следующие механики:

1. Управление камерой
2. Управление передвижением
3. Управление атакой

На самом деле этот скрипт(Рисунок 6) не содержит реализации, он просто использует функционал и логику других скриптов для управления персонажем.

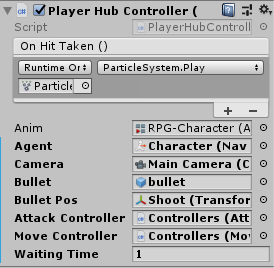
****

Рисунок 6. PlayerHubController

Как можно заметить, данный скрипт принимает огромное количество разных скриптов, объектов, компонентов.

Так, например, благодаря MoveController и NavMeshAgent, персонаж может передвигаться и запускать некоторые анимации через Animator. Используя Bullet, Bullet position и AttackController позволяют стрелять, перезаряжаться. С помощью метода OnHitTaken при получении урона запускается Particle System

**Основной функционал вражеского нпс**

Помимо игрока на сцене также должны присутствовать другие активные сущности. К данной сущности относится сущность Enemy с реализованным скриптом Enemy Controller(Рисунок 7). В этом классе представлено использование механики атаки вражеского нпс на персонажа.

Так AttackController непосредственно атакует игрока.

Bullet указывает чем и Shoot показывает через какую точку происходит атака.Character указывает на кого направлено действие.

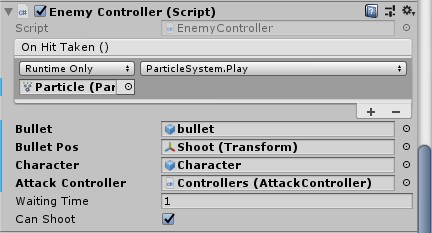
****

Рисунок 7. EnemyController

Enemy также умеет убегать, если ему добавить соответствующий скрипт Running Enemy. Данный скрипт, если видит в поле зрения персонажа, то сообщает Enemy и пытается отступить от игрока:

void RunAway()

{

isFollowed = false;

var playerDirection = transform.position - Player.position;

var newDestination = transform.position + playerDirection;

Agent.SetDestination(newDestination);

}

**Характеристики персонажа и врагов**

В этой системе присутствует механика отслеживания здоровья у объектов, но где непосредственно находятся все данные о максимальном, текущем и минимальном здоровье, уроне и защите? Такая функциональность обязательно должна присутствовать. Так в чем же она заключается?

В скрипт передаются данные о максимальном здоровье, уроне, защите и регенерации. Относительно этих базовых характеристик вычисляются дальнейшие действие.

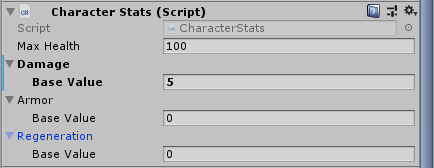


Рисунок 8. Character stats

Например так вычисляется урон, нанесенный сущности:

public void TakeDamage(float damage)

{

damage -= armor.GetValue();

damage = Mathf.Clamp(damage, 0, int.MaxValue);

CurrentHealth -= damage;

Debug.Log(transform.name + " takes " + damage + " damage");

}

Или же метод регенерации здоровья:

public void Regenerate()

{

CurrentHealth += regeneration.GetValue();

}

**Анимации**

Для более интересной и динамичной игры была добавлена анимация персонажа. Для данной работы использовался готовый ассет с готовой моделью и анимациями. Логика взаимодействий разрабатывалась в аниматоре под условия игры.

По дефолту всегда включается Idle-анимация. В зависимости от того бежит ли персонаж или использует один из видов атаки, то и соответственно переключается на определенный тип анимации.

Сам граф логики анимаций выглядит так(Рисунок 9):

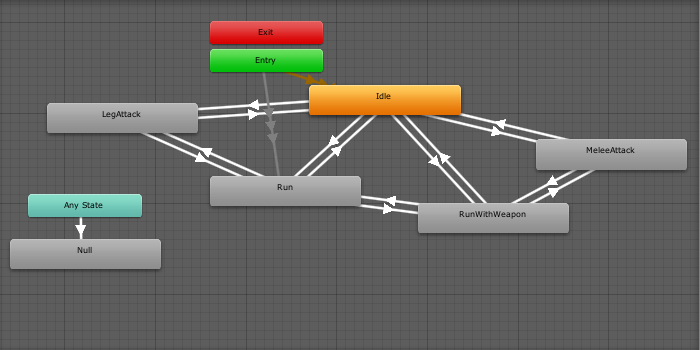


Рисунок 9. Реализация графа анимаций в аниматоре

Например, если подобрать оружие и попытаться бежать после его использования, то анимация Run будет заменена на RunWithWeapon из-за соответствующих проверок и переключений булевых переменных.

LegAttack не зависит от наличия экипируемого оружия. MeleeAttack в свою очередь наоборот зависит от наличия оружия ближнего боя.

Реализуется активация просто с помощью вызова метода в Update с проверкой нажатия клавиши и булевого состояния.

**Design**

В данном проекте было использованы некоторые ассеты и некоторые свои наработки. Визуальная составляющая была направлена на минимализм, чтобы игрок был сосредоточен на конечной цели игры time-killer non target shooter`а.

Рассмотрим все возможные добавленные элементы дизайна в проекте:

1. Были использованы собственные наработки 3D объектов:

1.1. Здание для наполнения предметами и врагами.

1.2. Ракета для визуальной наполненности

1.3. Колодец также для визуальной составляющей, как декорация

2. Использование ассетов:

2.1. Персонаж

2.2. Меч

2.3. Магическая палка

3. Также использование примитивов для создания пространства:

3.1. Кубы для стен

3.2. Plane для пола

4. Использование UI для интерфейса:

4.1. Использование картинок для визуального отображения оружия, инвентаря

4.2. Использование текста для описания

5. Использование Particle System при получении урона

При создании собственных 3D моделей был использован такой программный софт, как Autodesk Maya 2019 и Substance Painter. Были изучены темы создания малополигональных моделей и импортировании их в Unity.

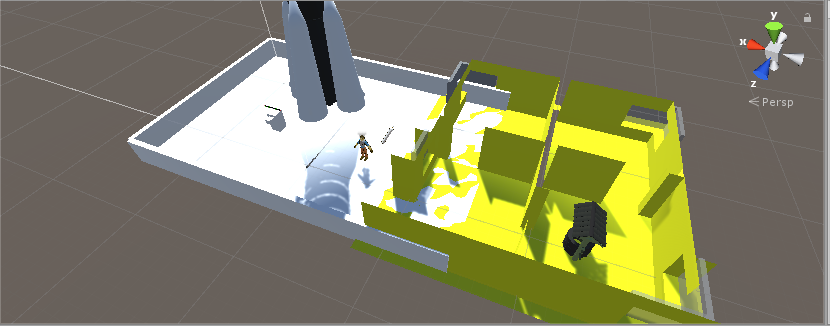
Так выглядит сцена со всеми присутствующими на ней дизайн-объектах:****

Рисунок 10. Вид сцены сверху

**Управление камерой**

Так как в игре персонаж управляется в изометрии, то было необходимо создать соответствующий скрипт с управлением камеры. При создании скрипта учитывались: угол наклона, зум и поворот камеры относительно игрока.

Так был создан скрипт CameraManager:

void Update()

{

currentZoom -= Input.GetAxis("Mouse ScrollWheel") \* zoomSpeed;

currentZoom = Mathf.Clamp(currentZoom, minZoom, maxZoom);

transform.LookAt(target.position);

offset = Quaternion.AngleAxis(Input.GetAxis("RotateCamera") \* rotationSpeed \* Time.deltaTime, Vector3.up) \* offset;

transform.position = target.position + offset \* currentZoom;

}

**Заключение**

В данном проекте использовалось большое количество разных механик. Многие механики интегрировались в другие или как-то их дополняли. Тем не менее в конечном итоге механики отлично друг друга реализовали. Игра позволила узнать много новых технологий, логик, реализаций. Вышла довольно хорошая игра, позволяющая убивать время, как и написано в жанре игры. Благодаря проекту возрос интерес к изучению нового контента в сфере игр.

**Источники**

1. Статьи https://unity3dschool.ru/stati/
2. Статьи https://habr.com/ru/
3. Cтатьи https://vc.ru/tag/unity
4. Статьи https://ru.stackoverflow.com/
5. Статьи https://tproger.ru/
6. Unity Physics https://docs.unity3d.com/ScriptReference/Physics.html
7. Видеоуроки по Unity от GameGrind https://www.youtube.com/user/AwfulMedia
8. Обучающие видеоуроки по Autodesk Maya https://www.youtube.com/user/Artemmilan1
9. Статьи http://gamesmaker.ru/3d-game-engines/unity3d/
10. Документация Clip Paint Studio https://www.clipstudio.net/en/manuals/
11. Статьи https://devtribe.ru/p/unity/resources/articles
12. Документация Unity https://docs.unity3d.com/Manual/index.html
13. Unity NavMesh https://docs.unity3d.com/ScriptReference/AI.NavMesh.html
14. Официальный базовый учебный видеокурс Substance Painter http://3d-vladimir.blogspot.com/2016/04/substance-painter.html
15. Видеоуроки по Unity от Brackeys https://www.youtube.com/user/Brackeys
16. Документация Autodesk Maya https://knowledge.autodesk.com/ru/support/maya/getting-started/caas/simplecontent/content/maya-documentation.html
17. Книга Unity for Absolute Beginners. Sue Blackman.