МИНИСТЕРСТВО ОБРАЗОВАНИЯ И НАУКИ РОССИЙСКОЙ ФЕДЕРАЦИИ

Федеральное государственное автономное образовательное учреждение высшего образования
«Санкт–Петербургский государственный университет
аэрокосмического приборостроения» Кафедра вычислительных систем и сетей

ОТЧЁТ ПО ПРАКТИКЕ

ЗАЩИЩЁН С ОЦЕНКОЙ

РУКОВОДИТЕЛЬ

доц., канд. техн. наук., доц.		А.В.Никитин
должность, уч. степень, звание	подпись, дата	инициалы, фамилия

ОТЧЁТ ПО ПРАКТИКЕ

вид практики производственная

тип практики научно-исследовательская

на тему индивидуального задания Исследование возможностей разработки учебно-игрового приложения для ПК и шлема виртуальной реальности HTC Vive.

выполнен Валяевым Дмитрием Валерьевичем

фамилия, имя, отчество обучающегося в творительном падеже

по направлению подготовки 09.04.01 (код) Информатика и вычислительная техника (наименование направления)

направленности 12 (код) Системы мультимедиа и компьютерная графика (наименование направленности)

Обучающийся группы № 4640М (номер) Д.В. Валяев (подпись, дата; инициалы, фамилия)

Санкт–Петербург 2017

ТЕХНИЧЕСКОЕ ЗАДАНИЕ
на разработку учебно-игрового многопользовательского приложения для симуляторов XD-Motion, Fly-Motion, 5D-Motion и шлемов виртуальной реальности

1. Исходные требования

	Объект разработки	Валяев
1	Сценарий игры + карта + вид и размеры всех объектов	**Карта**

2	3D модель фрагмента виртуального города, включая:
	• Фон – горизонт, небо и т.п.
	• Ландшафт – равнина или холмистая местность, реки и озера, растительность (трава, кусты, деревья), сеть дорог, тоннелей и мостов и др.
	• Внешний вид зданий.
	• Автомобиль + самолет (или вертолет, квадрокоптер) + тележка.
	• Персонажи
	  • Пользовательские.
	  • Автономные.
	• Освещение, соответствующее времени года и суток, включая тени.
	• Звуковое сопровождение
	  • Фоновое.
	  • Соответствующее различным действиям.
		Лягушки

3	Анимации объектов и персонажей в соответствии с игровым сценарием.	**лягушки: прыгает и кусает**
4	Автономные персонажи с искусственным интеллектом (навигация, реакция)	**лягушки: навигация, реакция**
5	Управление моделями персонажей и транспортных средств исходя из возможностей симуляторов и шлемов виртуальной реальности в соответствии с игровым сценарием.	**Десктоп, вайв**
6	• Оптимизации рендера (уровень детализации, окклюзия)	**оптимизация**
7	Единый интерфейс пользователя (информационные и управляющие элементы) с возможностями: • Просмотра сцены от 1-го или 3-го лица. • Отображения местоположения при многопользовательском режиме взаимодействия (мини-карта).	**интерфейс здоровья и патронов**
8	Режимы работы: • Автономный. • Многопользовательский.	**Автономно: десктоп и вайв**
9	Доставка пользователю: • Локальная.	
10	Средства взаимодействия с приложением: • Симуляторы XD-Motion, Fly-Motion, 5D-Motion. • Шлемы виртуальной реальности с контроллерами. • Десктоп	**Вайв, десктоп**

2. Календарный план

№	Работа	Сроки	Результат
1	Уточнение сценария и объектов разработки, подготовка исходных данных	14.09 - 28.09.2017	Сценарий, исходные данные
2	Уточнение общего и индивидуальных ТЗ, этапов и сроков разработки	28.09.2017	Развернутое общее и индивидуальные ТЗ, уточненный КП
3	Разработка основы модели виртуальной местности	28.09 – 31.10.2017	Задел под модель виртуальной местности
4	Разработка алгоритма поведения и трёхмерных моделей неигровых персонажей	14.09 – 31.11.2017	Злые лягушки, хуманоид
5	Реализация автономного управления моделями посредством XD-Motion, Fly-Motion, 5D-Motion	01.10 – 31.11.2017	Возможность использовать всё это для взаимодействия с моделью
6	Реализация автономного управления моделями посредством шлемов виртуальной реальности и десктопа	31.10 – 28.12.2017	Возможность использовать это для взаимодействия с моделью
7	Разработка полной модели виртуального города	31.10 – 28.12.2017	Модель города
8	Реализация многопользовательского режима. Апробация и демонстрация результатов. Подготовка отчета.	01.10 – 28.12.2017	Отчет. Инструкции. Акт апробации. Акт сдачи-приемки. ПО на носителе.

Содержание

Перечень сокращений, символов и специальных терминов с их определениями.

Пивот – точка вращения.

Плейсхолдер – пустой объект, который обозначает место, в котором может появиться что-либо другое.

Префаб – заготовка объекта.

Кастомный – созданный разработчиком самостоятельно, специально для данной ситуации.

Спавн – появление объекта в виртуальном пространстве.

Спавнить – вызывать спавн.

Рантайм – время выполнения.

Десктоп – обычный настольный компьютер.

Паттерн - эффективный способ решения характерных задач проектирования.

Синглтон – паттерн, предполагающий наличие в памяти одного единственного уникального экземпляра данного объекта.

VR – виртуальная реальность.

Кейс – конкретный случай применения объекта

Маппинг – создание виртуальной карты/модели чего-либо

Введение.

Шлем виртуальной реальности (англ. Head-mounted display) — устройство, позволяющее частично погрузиться в мир виртуальной реальности, создающее зрительный и акустический эффект присутствия в заданном управляющим устройством (компьютером) пространстве. Представляет собой конструкцию, надеваемую на голову, снабженное видеоэкраном и акустической системой. Название «шлем» достаточно условное: современные модели гораздо больше похожи на очки, чем на шлем.

Шлем создаёт объёмную картинку, демонстрируя два изображения — по одному для каждого глаза. Кроме того, шлем может содержать гироскопический или инфракрасный датчик положения головы.

В настоящее время шлемы виртуальной реальности остаются довольно дорогими для широкого потребителя, что ограничивает потенциальную аудиторию созданных для VR приложений. Одно из возможных решений данной проблемы – создание приложений, способных работать как с VR шлемами, так и с обычными десктопами.

Особенностью созданного приложения является архитектура, позволяющая использовать одни и те же игровые предметы при запуске приложения с обеих платформ.

Игровые предметы позволяют игроку различным способом взаимодействовать с окружением. Например, предметом «рука» игрок может поднимать физические объекты, а различные виды оружия позволяют атаковать неигровых персонажей.

Дополнительным результатом выполнения научно-исследовательской работы является создание неигрового персонажа с искусственным интеллектом на основе алгоритмического паттерна “State Machine”.

1. Описание созданных скриптов.

1.1 Структура классов игровых сущностей.

Любая игровая сущность, имеющая такие свойства, как запас здоровья и принадлежность к команде, является наследником класса BaseEntity. Полная структура классов игровых сущностей приведена на рисунке.

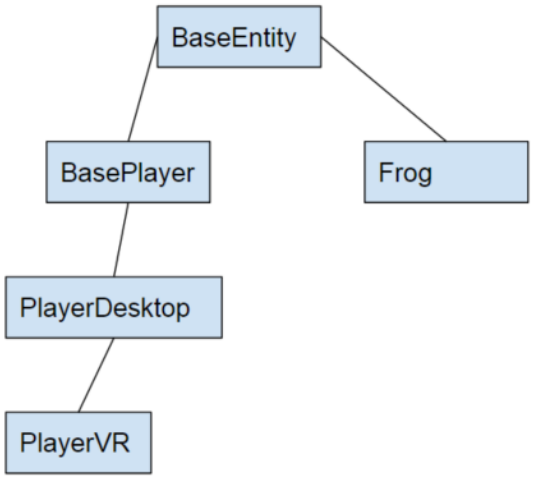

Рисунок 1: Структура классов игровых сущностей.

BaseEntity – базовый класс для любой уничтожаемой сущности. Содержит в себе поля и методы для подсчёта оставшегося здоровья, базовую логику уничтожения сущностей с нулевым здоровьем, а так же информацию об отношению сущности к какой-либо команде. По-умолчанию есть три команды:
-Friendly – дружеская сущность.
-Hostile – враждебная сущность.
-Neutral – нейтральная сущность.
Так же в данном классе реализовано опознание другой сущности, которая нанесла урон данной сущности. В случае получения информации о нанесении другой сущностью урона, выполняется поиск на объекте компонента BaseAI, отвечающего за искусственный интеллект, и, в случае успеха поиска, ему сообщается об атаковавшем объекте.

```
Public class BaseEntity : MonoBehaviour
{
    [SerializeField] private TeamList team;
    public TeamList Team
    {
        get { return team; }
        set { team = value; }
    }
    [SerializeField] protected float maxHealth;
    protected float currentHealth;
    private bool destroyInLateUpdate;

    public EventHub Ehub { get; private set; }

    void Awake()
    {
        currentHealth = maxHealth;
        Ehub = FindObjectOfType<EventHub>();
    }


    public void TakeDamage(float value)
    {
        currentHealth -= value;
        OnDamageTaken(value);

        Debug.Log(string.Format("{0} took {1} damage. Remaining health = {2}", gameObject, 6im
es, currentHealth));

        if (currentHealth <= 0)
            OnDeath();
    }

    public void TakeDamage(float value, BaseEntity attacker)
    {
        TakeDamage(value);
        BaseAI brain = this.GetComponent<BaseAI>();
        if (brain != null)
        {
            //Debug.Log("got brain");
            brain.AttackedBy(attacker);
        }
    }

    protected virtual void OnDamageTaken(float value) { }

    protected virtual void OnDeath()
    {
        Ehub.SignalEntityDeath(this);
        destroyInLateUpdate = true;
    }
```

```
    void LateUpdate()
    {
        if (destroyInLateUpdate)
            Destroy(this.gameObject);
    }
}
```

BasePlayer – наследник класса *BaseEntity*, добавляющий функционал, присущий сущностям, управляемым живым человеком. Уникальным для таких сущностей является наличие индикатора здоровья, а так же другой алгоритм смерти – игроки не исчезают – они возвращаются в точку изначального появления, иначе в программе возникли бы ошибки.

```
Public class BasePlayer : BaseEntity {

    [SerializeField] private Slider healthBar;
    protected Vector3 startingPosition;

    public override void Start()
    {
        base.Start();
        startingPosition = transform.position;
    }

    protected override void Update()
    {
        base.Update();
        //Debug.Log("changing healthbar");
        healthBar.value = currentHealth / maxHealth;
    }

    protected override void OnDeath()
    {
        Ehub.SignalEntityDeath(this);
        currentHealth = maxHealth;
        transform.position = startingPosition;
    }

}
```

PlayerDesktop – наследник класса *BasePlayer*, добавляющий функционал, присущий игроку, управляемому с обычного настольного компьютера. В него входит возможность выбрать оружие и использовать его.

```
Public class PlayerDesktop : BasePlayer
{
    [SerializeField] private List<BaseWeapon> weapons;
    [SerializeField] private Transform weaponSlot;
    //private DesktopUIManager uiManager;


    private BaseWeapon currentWeapon;
    private int currentWeaponIndex = 0;

    public override void Start()
    {
        base.Start();
        //uiManager = this.GetComponent<DesktopUIManager>();
        currentWeapon = Instantiate(weapons[currentWeaponIndex], weaponSlot.position, weaponSl
ot.rotation);
        currentWeapon.transform.parent = weaponSlot;
        currentWeapon.Holder = this;
        currentWeapon.Team = Team;
        Debug.Log("spawning guns");
```

```
    }

    public void FireGun()
    {
        if (currentWeapon != null)
            currentWeapon.Fire();
    }

    protected override void OnDamageTaken(float value)
    {
        //uiManager.SetShownHealth(currentHealth);
    }

    public void SwitchGun()
    {
        if (currentWeapon != null)
            Destroy(currentWeapon.gameObject);

        currentWeaponIndex += 1;
        currentWeaponIndex = currentWeaponIndex % weapons.Count;

        currentWeapon = Instantiate(weapons[currentWeaponIndex], weaponSlot.position, weaponSl
ot.rotation);
        currentWeapon.transform.parent = weaponSlot;
        currentWeapon.Holder = this;
        currentWeapon.Team = Team;
    }


}
```

PlayerVR – наследник класса *PlayerDesktop*, изменяющий функционал последнего для корректности работы со шлемом виртуальной реальности. В частности, немного изменен алгоритм телепортации.

```
Public class PlayerVR : PlayerDesktop {

    [SerializeField] Transform cameraRig;

    public override void Start()
    {
        base.Start();
        startingPosition = cameraRig.transform.position;
    }

    protected override void OnDeath()
    {
        base.OnDeath();
        cameraRig.transform.position = startingPosition;
    }


}
```

1.2 Структура основных классов оружия.

Любое оружие или другой предмет, который игрок может использовать, должен быть наследником класса BaseWeapon. Полная структура классов оружия приведена на рисунке.

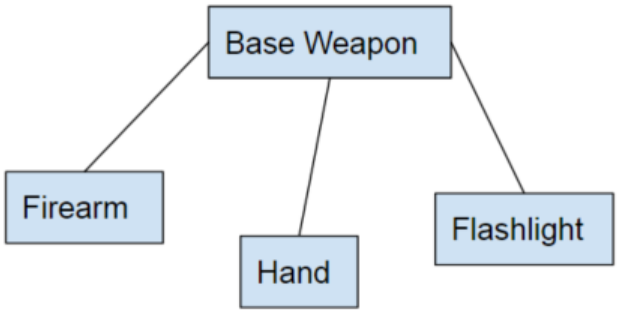

Рисунок 2: Основные классы оружия.

BaseWeapon – базовый класс для всех используемых предметов. Содержит информацию об оставшихся боеприпасах и о том, кто держит оружие в данный момент, контролирует время между выстрелами. При выстреле вызывает общедоступный UnityEvent OnFire.

```
Public class BaseWeapon : MonoBehaviour
{
    [SerializeField] protected int ammo = 99;
    [SerializeField] protected float coolDown = 0.5f;
    private float nextShotTime;
    public BaseEntity Holder { get; set; }
    public TeamList Team { get; set; }
    public UnityEvent OnFire;
    public virtual void Start()
    {
        nextShotTime = Time.time;
    }

    public void Fire()
    {
        if (Time.time > nextShotTime && ammo > 0)
        {
            nextShotTime = Time.time + coolDown;
            ammo--;
            OnPrimaryFire();
        }
    }

    protected virtual void OnPrimaryFire()
    {
    }
}
```

Firearm – наследник класса *BaseWeapon*, отвечает за работу огнестрельного оружия. Содержит следующую информацию:

- Ссылка на префаб пули.
- Ссылка на место появления пули.
- Ссылка на систему частиц, включающаяся при выстреле.
- Ссылка на экран с отображением количества патронов.

```
Public class Firearm : BaseWeapon
{
    [SerializeField] Transform bulletSpawnPoint;
    [SerializeField] GameObject bulletPrefab;
    [SerializeField] ParticleSystem muzzleFlashParticle;
    [SerializeField] Text ammoCounter;


    public override void Start()
    {
```

```
        base.Start();
        Team = this.transform.root.GetComponentInChildren<BaseEntity>().Team;
        if (ammoCounter != null)
            ammoCounter.text = string.Format("{0}", ammo);
    }

    protected override void OnPrimaryFire()
    {
        GameObject bullet = Instantiate(bulletPrefab, bulletSpawnPoint.position, bulletSpawnPo
int.rotation);
        bullet.GetComponent<BaseBullet>().Team = Team;
        bullet.GetComponent<BaseBullet>().Holder = Holder;
        bullet.GetComponent<Rigidbody>().velocity += this.GetComponent<Rigidbody>().velocity;
        foreach (Collider col in GetAllColliders())
        {
            Physics.IgnoreCollision(col, bullet.GetComponent<Collider>());
        }

        muzzleFlashParticle.Play();
        if (ammoCounter != null)
            ammoCounter.text = string.Format("{0}", ammo);
        OnFire.Invoke();
    }

    private Collider[] GetAllColliders()
    {
        return transform.root.GetComponentsInChildren<Collider>();
    }
}
```

Hand – наследник класса *BaseWeapon*, содержит логику работы руки, которая при «выстреле» может схватить физический объект, с которым она соприкасается.

```
Public class Hand : BaseWeapon {

    bool grabbed = false;
    GameObject touchedObject;
    [SerializeField] FixedJoint joint;
    [SerializeField] Transform fingers;
    [SerializeField] Transform originalFingerPlaceholder;
    [SerializeField] Transform closedFingerPlaceholder;
    bool fistClosed = false;
    Vector3 lastPos;
    Vector3 currentVelocity;

    void OnCollisionEnter(Collision col)
    {
        GameObject other = col.gameObject;
        if (other.GetComponent<Rigidbody>() != null)
        {
            touchedObject = other;
        }
    }

    void OnCollisionExit(Collision col)
    {
        GameObject other = col.gameObject;
        if (other == touchedObject)
        {
            touchedObject = null;
        }
    }

    protected override void OnPrimaryFire()
    {
        ammo++;
        if (grabbed && joint.connectedBody!=null)
        {
```

```
                Rigidbody other = joint.connectedBody;
                joint.connectedBody = null;
                other.velocity = currentVelocity;
            Debug.Log(string.Format("Other object is {0}, its velocity is {1}, controller velo
city is {2}", other.gameObject, other.velocity, currentVelocity));
                grabbed = false;
        }
        else if (touchedObject != null && !grabbed)
        {
            joint.connectedBody = touchedObject.GetComponent<Rigidbody>();
            grabbed = true;
        }
        fistClosed = !fistClosed;
        if (grabbed)
            fistClosed = true;
        MoveFingers();

    }

    void MoveFingers()
    {
        if (fistClosed)
        {
            fingers.rotation = closedFingerPlaceholder.rotation;
        }
        else
        {
            fingers.rotation = originalFingerPlaceholder.rotation;
        }
    }

    void Update()
    {
        currentVelocity = (this.transform.position - lastPos) / Time.deltaTime;
        lastPos = this.transform.position;
    }

    public override void Start()
    {
        Collider[] playerColliders = GetComponentsInParent<Collider>();
        foreach (var current in playerColliders)
        {
            Physics.IgnoreCollision(this.GetComponent<Collider>(), current);
        }
    }

}
```

Flashlight – наследник класса *BaseWeapon*, содержит логику работы фонарика, который включается/выключается при «выстреле».

```
Public class Flashlight : BaseWeapon {

    [SerializeField] Light lightSource;
    private bool switchedOn = true;

    protected override void OnPrimaryFire()
    {
        ammo = 99;
        switchedOn = !switchedOn;
        lightSource.enabled = switchedOn;
    }
}
```

1.3 Структура основных классов пуль.

Все пули являются наследниками класса BaseBullet.

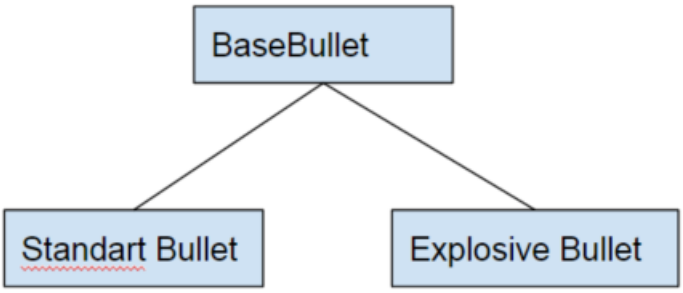

Рисунок 3: Основные классы пуль.

BaseBullet – основной класс для всех пуль. Содержит информацию о скорости вылета пули и о том, кто её выпустил. Содержит логику реакции на столкновения с другими объектами.

```
Public class BaseBullet : MonoBehaviour
{
    private Rigidbody rb;
    [SerializeField] private float launchForce;

    public TeamList Team { get; set; }

    public BaseEntity Holder { get; set; }

    public UnityEvent OnSetOff;

    [SerializeField] private GameObject soundHolderToSpawn;

    void Start()
    {
        rb = this.GetComponent<Rigidbody>();
        rb.AddForce(transform.forward * launchForce);
    }

    void OnCollisionEnter(Collision col)
    {
        SetOff(col);
        OnSetOff.Invoke();
        if (soundHolderToSpawn != null)
        {
            Instantiate(soundHolderToSpawn, transform.position, transform.rotation);
        }
        Destroy(this.gameObject);
    }

    protected virtual void SetOff(Collision col)
    {
    }
}
```

StandartBullet – обычная пуля, наследник класса BaseBullet. Содержит в себе информацию об уроне, наносимом при попадании, содержит логику нанесения урона враждебным сущностям. Содержит информацию том, какую систему частиц воспроизвести при попадании.

```
Public class StandartBullet : BaseBullet
{
    [SerializeField] private float damage;
    [SerializeField] private ParticleSystem hitParticle;

    protected override void SetOff(Collision col)
    {
```

```
        //Debug.Log(col.collider.gameObject);
        BaseEntity entityHit = col.collider.gameObject.GetComponent<BaseEntity>();
        if (entityHit != null)
        {
            if (entityHit.Team != Team)
            {
                entityHit.TakeDamage(damage);
            }
        }
        Instantiate(hitParticle, col.contacts[0].point, Quaternion.LookRotation(col.contacts[0
].normal));

        BaseEntity target = col.collider.gameObject.GetComponentInParent<BaseEntity>();
        if (target != null)
            target.TakeDamage(damage, Holder);
    }
}
```

ExplosiveBullet – взрывающаяся пуля, наследник класса BaseBullet. Содержит логику нанесения урона в радиусе взрыва, а так же отбрасывания физических объектов от эпицентра. Содержит информацию том, какую систему частиц воспроизвести при попадании.

```
Public class ExplosiveBullet : BaseBullet {

    [SerializeField]
    protected float radius = 5.0F;

    [SerializeField]
    protected float power = 1000.0F;

    [SerializeField]
    protected float baseDamage = 20.0F;

    [SerializeField]
    protected ParticleSystem ps;

    protected override void SetOff(Collision col)
    {
        Vector3 explosionPos = transform.position;
        Collider[] colliders = Physics.OverlapSphere(explosionPos, radius);
        foreach (Collider hit in colliders)
        {
            Rigidbody rb = hit.GetComponent<Rigidbody>();

            if (rb != null && rb.tag != "Projectile")
                rb.AddExplosionForce(power, transform.position, radius, 3.0F);

            BaseEntity entityHit = hit.GetComponent<BaseEntity>();

            if (entityHit != null)
            {
                if (entityHit.Team != Team)
                {
                    Vector3 direction = rb.transform.position - transform.position;
                    float amountOfDamage = baseDamage -
 baseDamage * (direction.magnitude / radius);
                    entityHit.TakeDamage(amountOfDamage, Holder);
                    Debug.Log(string.Format("Entity {0} took {1} damage.", rb.gameObject, amou
ntOfDamage));
                }
            }

        }
        ParticleSystem particle = Instantiate (ps, explosionPos, Quaternion.Euler(Vector3.up))
        particle.transform.Rotate (new Vector3 (-90,0,0));
        particle.Play ();
```

```
    }
}
```

1.4 Синглтоны.

На каждой сцене проекта должно быть два синглтона:

SceneManager – объект этого класса отвечает за появление игрока на сцене. Он определяет, какой именно префаб игрока будет использован. Это зависит от того устройство, с которого игрок подключается к сцене. Также этот класс отвечает за события паузы и возобновления игры.

```
Public class SceneManager : MonoBehaviour
{
    private static SceneManager instance = null;
    private bool IamUseless = false;


    [SerializeField]
    PlayerTypes playerType;

    [SerializeField]
    Transform spawnPoint;

    [SerializeField]
    private Canvas desktopUI;


    [Header("Player prefabs")]
    [SerializeField]
    GameObject desktopPrefab;

    [SerializeField] GameObject vrPrefab;
    [SerializeField] GameObject xdPrefab;
    [SerializeField] GameObject flyPrefab;
    [SerializeField] GameObject fiveDPrefab;

    private bool paused = false;
    // Use this for initialization
    void Awake()
    {
        if (instance == null)
            instance = this;
        else
        {
            Debug.LogWarning("Multiple instances of SceneManager (singleton) on the scene! Ext
erminate!!!1");
            IamUseless = true;
            Destroy(this);
        }

        Cursor.visible = false;

        //playerType = ModeSelector.playerType;
        switch (playerType)
        {
            case PlayerTypes.Desktop:
                Instantiate(desktopPrefab, spawnPoint.position, spawnPoint.rotation);
                VRSettings.enabled = false;
                break;

            case PlayerTypes.VR:
                Instantiate(vrPrefab, spawnPoint.position, spawnPoint.rotation);
                VRSettings.enabled = true;
                break;
        }
    }
```

```
    public void OnDestroy()
    {
        if (IamUseless)
            instance = null;
    }

    private void Update()
    {
        if (playerType == PlayerTypes.Desktop && Input.GetKeyDown(KeyCode.Escape))
        {
            if (paused)
            {
                Unpause();
            }
            else
            {
                Pause();
            }
        }
    }

    public void Unpause()
    {
        desktopUI.gameObject.SetActive(false);
        Time.timeScale = 1.0f;
        paused = false;
        Cursor.visible = false;
    }

    public void Pause()
    {
        desktopUI.gameObject.SetActive(true);
        Time.timeScale = 0.0f;
        paused = true;
        Cursor.visible = true;
    }

    public void Quit()
    {
        Application.Quit();
    }
}
```

EventHub – синглтон, реализующий паттерн “Publisher/Subscriber”. Он предназначен для централизованного оповещения всех заинтересованных сущностей о каких-либо событиях, например, о смерти каких-либо других сущностей.

```
Public delegate void EntityDeathHandler(BaseEntity killedUnit);

public class EventHub : MonoBehaviour
{
    private static EventHub instance = null;
    private bool IamUseless = false;

    public event EntityDeathHandler EntityDeathEvent;

    public void SignalEntityDeath(BaseEntity killedEntity)
    {
        Debug.Log(“EventHub: Signaling units death”);
        if (EntityDeathEvent != null)
        {
            EntityDeathEvent(killedEntity);
        }
    }

    // Use this for initialization
    void Awake()
```

```
    {
        if (instance == null)
            instance = this;
        else
        {
            Debug.LogWarning("Multiple instances of EventHub (singleton) on the scene! Extermi
nate!!!1");
            IamUseless = true;
            Destroy(this);
        }
    }

    // Update is called once per frame
    void Update()
    {
    }

    public void OnDestroy()
    {
        if (IamUseless)
            instance = null;
    }
}
```

1.5 Скрипты для HTC Vive.

VRInputIncapsulator – класс, предназначенный для инкапсуляции сигналов с органов управления в более удобную для программиста форму.

```
Public class VRInputIncapsulator : MonoBehaviour
{
    [SerializeField] Transform leftController;
    public Transform LeftController
    {
        get { return leftController; }
        set
        {
            leftController = value;
            Debug.Log(leftController.position);
        }
    }

    public Transform RightController { get; set; }

    public Transform Head { get; set; }

    bool leftTrigger;
    public bool LeftTrigger
    {
        get { return leftTrigger; }
        set
        {
            leftTrigger = value;
            if (leftTrigger)
                OnLeftTriggerPressed();
            else
                OnLeftTriggerReleased();
        }
    }

    bool rightTrigger;
    public bool RightTrigger
    {
        get { return rightTrigger; }
        set
```

```
    {
        rightTrigger = value;
        if (rightTrigger)
            OnRightTriggerPressed();
        else
            OnRightTriggerReleased();
    }
}

bool leftGrip;
public bool LeftGrip
{
    get { return leftGrip; }
    set
    {
        leftGrip = value;
        if (leftGrip)
            OnLeftGripPressed();
        else
            OnLeftGripReleased();
    }
}

bool rightGrip;
public bool RightGrip
{
    get { return rightGrip; }
    set
    {
        rightGrip = value;
        if (rightGrip)
            OnRightGripPressed();
        else
            OnRightGripReleased();
    }
}

bool leftTouch;
public bool LeftTouch
{
    get { return leftTouch; }
    set
    {
        leftTouch = value;
        if (leftTouch)
            OnLeftTouchPressed();
        else
            OnLeftTouchReleased();
    }
}

bool rightTouch;
public bool RightTouch
{
    get { return rightTouch; }
    set
    {
        rightTouch = value;
        if (RightTouch)
            OnRightTouchPressed();
        else
            OnRightTouchReleased();
    }
}

public virtual void OnRightTouchPressed() { }

public virtual void OnLeftTouchPressed() { }
```

```
    public virtual void OnRightGripPressed() { }

    public virtual void OnRightGripDown() { }

    public virtual void OnLeftGripPressed() { }

    public virtual void OnLeftGripDown() { }

    public virtual void OnRightTriggerPressed() { }

    public virtual void OnLeftTriggerPressed() { }

    public virtual void OnRightTouchReleased() { }

    public virtual void OnLeftTouchReleased() { }

    public virtual void OnRightGripReleased() { }

    public virtual void OnLeftGripReleased() { }

    public virtual void OnRightTriggerReleased() { }

    public virtual void OnLeftTriggerReleased() { }
}
```

VRController – обрабатывает сигналы с контроллера в исходном виде, передаёт их классу *VRInputIncapsulator*.

```
Public enum ControllerSides
{ Left, Right };

public class VRController : MonoBehaviour
{
    [SerializeField] ControllerSides controllerSide;
    [SerializeField] VRInputIncapsulator inputManager;

    private SteamVR_TrackedObject trackedObj;
    private SteamVR_Controller.Device Controller
    {
        get { return SteamVR_Controller.Input((int)trackedObj.index); }
    }

    void Start()
    {


        trackedObj = GetComponent<SteamVR_TrackedObject>();
        // if (controllerSide == ControllerSides.Right)
        //    inputManager.RightController = this.transform;
        // else
        //     inputManager.LeftController = this.transform;
    }

    void Update()
    {
        if (controllerSide == ControllerSides.Right)
        {
            if (Controller.GetPress(SteamVR_Controller.ButtonMask.Touchpad))
                inputManager.RightTouch = true;
            else
                inputManager.RightTouch = false;

            if (Controller.GetPress(SteamVR_Controller.ButtonMask.Trigger))
                inputManager.RightTrigger = true;
            else
                inputManager.RightTrigger = false;
```

```
            if (Controller.GetPress(SteamVR_Controller.ButtonMask.Grip))
                inputManager.RightGrip = true;
            else
                inputManager.RightGrip = false;

            if (Controller.GetPressDown(SteamVR_Controller.ButtonMask.Grip))
                inputManager.OnRightGripDown();
        }
        else
        {
            if (Controller.GetPress(SteamVR_Controller.ButtonMask.Touchpad))
                inputManager.LeftTouch = true;
            else
                inputManager.LeftTouch = false;

            if (Controller.GetPress(SteamVR_Controller.ButtonMask.Trigger))
            {
                //Debug.Log("left controller trigger pressed");
                inputManager.LeftTrigger = true;
            }
            else
                inputManager.LeftTrigger = false;

            if (Controller.GetPress(SteamVR_Controller.ButtonMask.Grip))
                inputManager.LeftGrip = true;
            else
                inputManager.LeftGrip = false;
            if (Controller.GetPressDown(SteamVR_Controller.ButtonMask.Grip))
                inputManager.OnLeftGripDown();
        }
    }
}
```

VRManager – класс-наследник *VRInputIncapsulator*, используется для связи входящих сигналов с префабом игрока.

```
Public class VRManager : VRInputIncapsulator
{
    private VRTeleport teleport;
    [SerializeField] private PlayerVR player;

    void Start()
    {
        teleport = GetComponent<VRTeleport>();
        //player = GetComponentInChildren<PlayerDesktop>();
    }

    public override void OnLeftTouchPressed()
    {
        //Debug.Log("Left trigger pressed");
        teleport.AimTeleportLaser(LeftController);
    }

    public override void OnLeftTouchReleased()
    {
        teleport.AttemptTeleportation();
        //Debug.Log("Left trigger released");
    }

    public override void OnRightTriggerPressed()
    {
        player.FireGun();
    }

    public override void OnRightGripDown()
    {
```

```
        player.SwitchGun();
    }
}
```

VRTeleport – класс, реализующий работу телепортатора. Реализует такие функции, как собственно телепортация, отсчёт времени перезарядки телепортатора и отрисовка луча телепортатора

```
public class VRTeleport : MonoBehaviour
{
    [SerializeField] Transform cameraRig;
    [SerializeField] Transform head;
    [SerializeField] Vector3 teleportReticleOffset;
    [SerializeField] LayerMask teleportMask;
    [SerializeField] GameObject laserPrefab;
    GameObject laser;
    [SerializeField] GameObject teleportReticlePrefab;
    [SerializeField] float maxTeleportDistance = 30;
    [SerializeField] float teleportCooldown = 2.0f;
    Slider coolDownSlider;
    [SerializeField] GameObject teleporterPrefab;
    [SerializeField] Transform teleporterPlaceholder;
    float nextTeleport;
    GameObject reticle;
    Vector3 hitPoint;
    bool shouldTeleport;
    bool inRange;

    void Start()
    {
        //trackedObj = GetComponent <SteamVR_TrackedObject> ();
        laser = Instantiate(laserPrefab);
        reticle = Instantiate(teleportReticlePrefab);
        nextTeleport = Time.time;
        GameObject instantiatedTeleporter = Instantiate(teleporterPrefab, teleporterPlaceholde
r.position, teleporterPlaceholder.rotation);
        instantiatedTeleporter.transform.parent = teleporterPlaceholder;
        coolDownSlider = instantiatedTeleporter.GetComponentInChildren<Slider>();
        coolDownSlider.value = 1;
    }

    public void AimTeleportLaser(Transform source)
    {
        RaycastHit hit;
        if (Physics.Raycast(source.transform.position, source.transform.forward, out hit, 300,
 teleportMask))
        {
            if (Vector3.Distance(hit.point, source.position) < maxTeleportDistance)
            {
                inRange = true;
            }
            else
            {
                inRange = false;
            }
            hitPoint = hit.point;
            ShowLaser(source.position, hit);
            reticle.SetActive(true);
            reticle.transform.position = hitPoint + teleportReticleOffset;
            shouldTeleport = true;
        }
    }

    public void AttemptTeleportation()
    {
        laser.SetActive(false);
        reticle.SetActive(false);
```

```
        if (shouldTeleport && CooldownComplete() && inRange)
        {
            Teleport();
        }
    }

    private void ShowLaser(Vector3 sourcePosition, RaycastHit hit)
    {
        laser.SetActive(true);
        laser.transform.position = Vector3.Lerp(sourcePosition, hitPoint, .5f);
        laser.transform.LookAt(hitPoint);
        laser.transform.localScale = new Vector3(laser.transform.localScale.x, laser.transform
.localScale.y, hit.distance);

        if (CooldownComplete() && inRange)
        {
            laser.GetComponent<Renderer>().material.color = Color.green;
            reticle.GetComponent<Renderer>().material.color = Color.green;
        }
        else
        {
            laser.GetComponent<Renderer>().material.color = Color.red;
            reticle.GetComponent<Renderer>().material.color = Color.red;
        }
    }

    private void Teleport()
    {
        nextTeleport = Time.time + teleportCooldown;
        shouldTeleport = false;
        reticle.SetActive(false);
        Vector3 difference = cameraRig.transform.position - head.position;
        difference.y = 0;
        cameraRig.position = hitPoint + difference;
    }

    bool CooldownComplete()
    {
        if (Time.time > nextTeleport)
            return true;
        else
            return false;
    }

    void Update()
    {
        if (Time.time > nextTeleport)
        {
            coolDownSlider.value = 1;
        }
        else
        {
            coolDownSlider.value = 1 - (nextTeleport - Time.time) / teleportCooldown;
        }
    }
}
```

1.6 Скрипты анимации аватара для HTC Vive.

AvatarBodyFollow – скрипт, заставляющий тело аватара перемещаться вслед за головой.

```
Public class AvatarBodyFollow : MonoBehaviour {

    [SerializeField] Transform objectToFollow;

    // Update is called once per frame
    void Update () {
        transform.position = objectToFollow.position;
        Vector3 newRotation = objectToFollow.rotation.eulerAngles;
        newRotation.x = 0;
        newRotation.z = 0;
        transform.localRotation = Quaternion.Euler(newRotation);
    }
}
```

AvatarHandsNeck – скрипт, управляющий движением рук аватара.

```
Public class AvatarHandsNeck : MonoBehaviour {

    [SerializeField] Transform leftShoulder;
    [SerializeField] Transform leftHandEnd;
    [SerializeField] Transform rightShoulder;
    [SerializeField] Transform rightHandEnd;
    [SerializeField] GameObject limbPrefab;
    GameObject leftHand;
    GameObject rightHand;
    GameObject neck;

    void Start()
    {
        leftHand = Instantiate(limbPrefab);
        rightHand = Instantiate(limbPrefab);
    }

    void Update()
    {
        DrawLimb(leftHand, leftShoulder, leftHandEnd);
        DrawLimb(rightHand, rightShoulder, rightHandEnd);
    }

    void DrawLimb(GameObject limb, Transform start, Transform end)
    {
        if (start == null || end == null)
        {
            Destroy(limb);
            return;
        }
        if (limb == null)
        {
            limb = Instantiate(limbPrefab);
        }
        limb.transform.position = Vector3.Lerp(start.position, end.position, 0.5f);
        limb.transform.LookAt(end.position);
        limb.transform.localScale = new Vector3(limb.transform.localScale.x, limb.transform.lo
calScale.y, Vector3.Distance(start.position, end.position));
    }
}
```

AvatarHeadFollow – скрипт, заставляющий голову аватара двигаться вслед за головой.

```
Public class AvatarHeadFollow : MonoBehaviour {

    [SerializeField] Transform objectToFollow;

    // Update is called once per frame
```

```
    void Update () {
        transform.position = objectToFollow.position;
        transform.rotation = objectToFollow.rotation;
    }
}
```

2. Описание и структура основных игровых префабов.

2.1 Десктоп игрок

Префаб PlayerDesktop является аватаром игрока в случае, если приложение запускается с обычного компьютера.

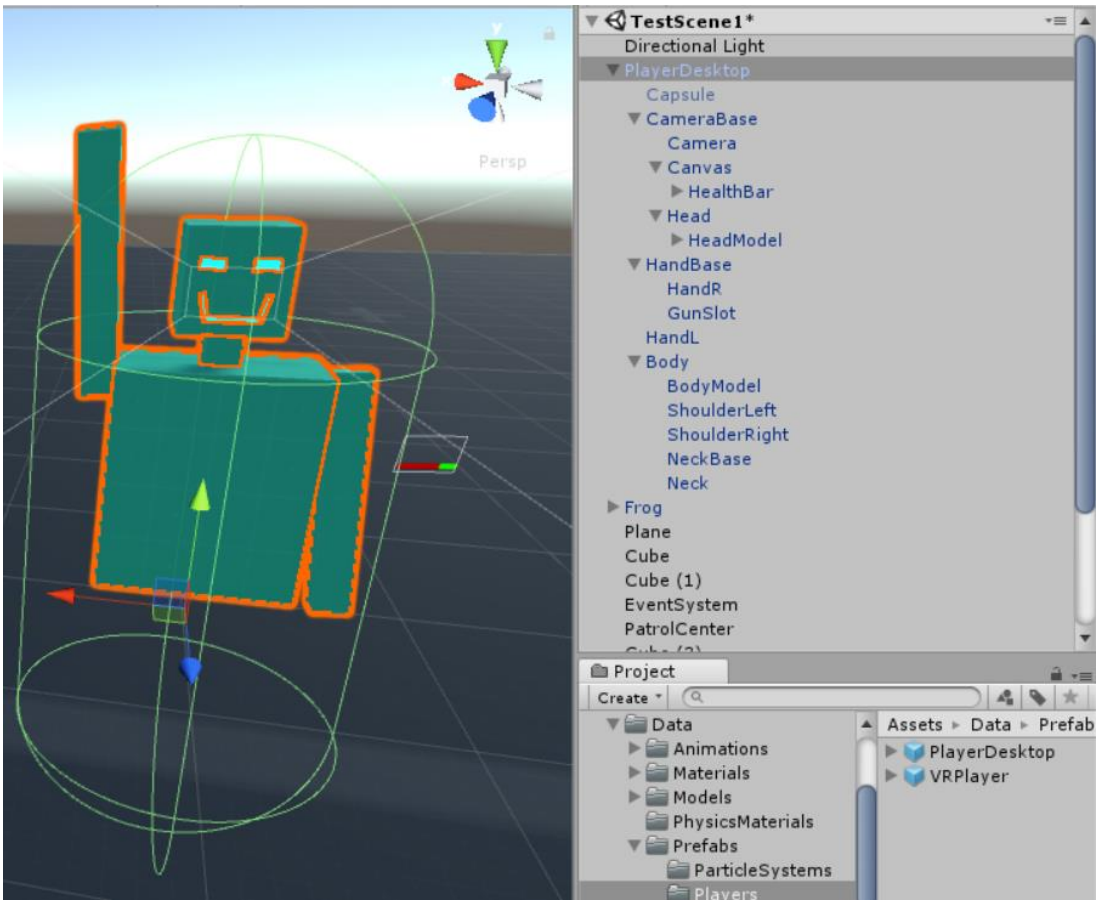

Рисунок 4: Внешний вид и иерархия префаба PlayerDesktop.

Основные части:

PlayerDesktop – базовый объект. Содержит все основные скрипты и коллайдер игрока.

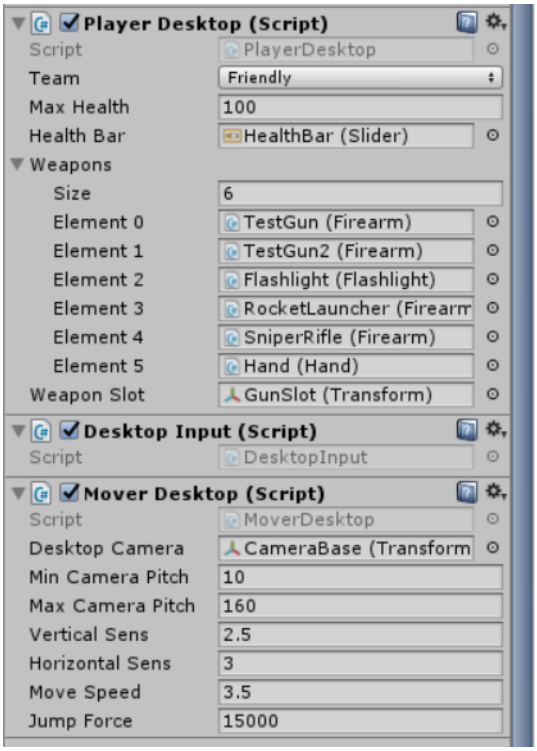

Рисунок 5: Основные настройки десктоп-игрока.

CameraBase – пивот для головы и камеры.
Camera – камера.
Canvas – worldspace canvas
HealthBar – расположенный в канвасе индикатор здоровья игрока
HandBase – пивот для рук персонажа. Содержит компонент MatchRotation, который заставляет пивот повторять вращение CameraBase.
HandR – правая рука персонажа
GunSlot – плейсхолдер для оружия, которое может быть помещено в руку
HandL – левая рука персонажа
Body – вершина иерархии статичной модели тела персонажа

2.2 VR игрок
Префаб VRPlayer является автаром игрока в случае, если он заходит в приложение используя HTCVive.

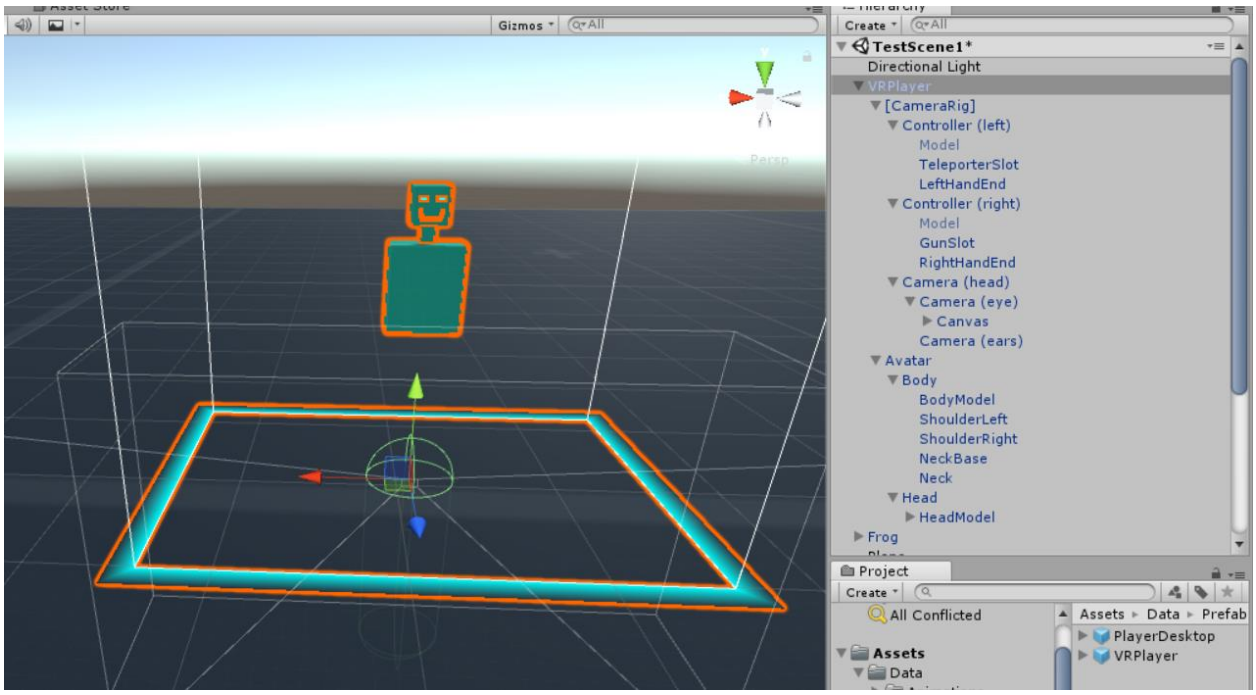

Рисунок 6: Внешний вид и иерархия префаба VRPlayer.

Основные части:
VRPlayer – основа префаба. Содержит в себе ряд настроек:

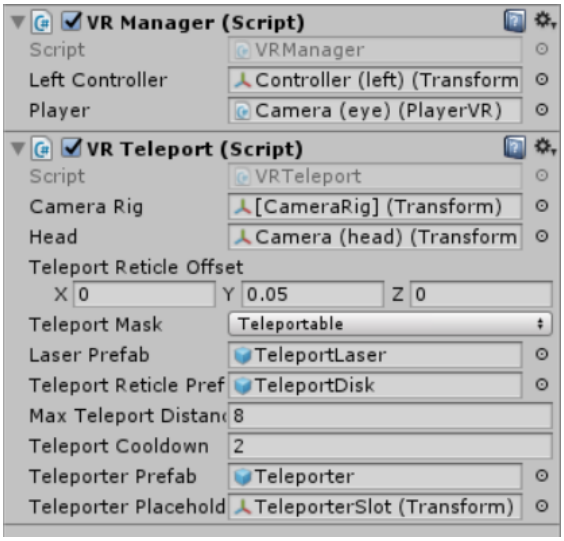

Рисунок 7: Настройки VRPlayer.

[CameraRig] (и всё, что под ним в иерархии) – переделанный префаб игрока, содержащийся в плагине SteamVR Plugin.
Controller (left) – левый контроллер. Содержит кастомный скрипт VRController.
TeleporterSlot – плейсхолдер для спавна префаба телепортера.
LeftHandEnd – плейсхолдер для окончания левой руки аватара
Controller (right) – правый контроллер
GunSlot – плейсхолдер для спавна оружия
RightHandEnd - плейсхолдер для окончания правой руки аватара
Camera - камера. Содержит на себе так же коллайдер игрока и некоторые настройки:

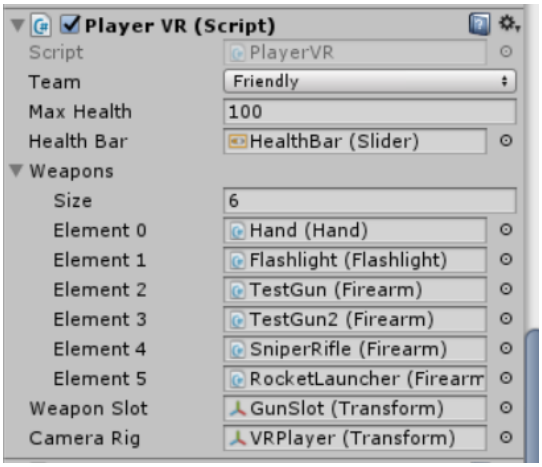

Рисунок 8: Настройки Camera.

Avatar (и всё что под ним в иерархии) – содержит двигающийся аватар игрока, управляемый скриптами AvatarHandsNeck, AvatarBodyFollow, AvatarHeadFollow.

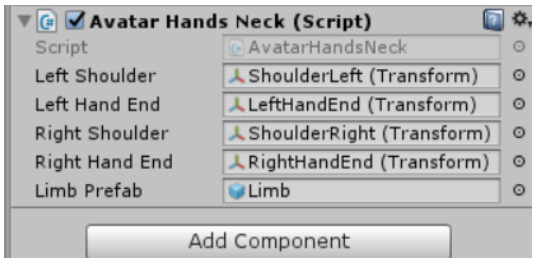

Рисунок 9: Настройки анимированного аватара.

Телепортер позволяет игроку перемещаться между различными точками карты. Чтобы точка стала доступна для телепортации, объект должен находиться в слое Teleportable. Телепортация ограничена по расстоянию, а время между телепортациями не может быть меньше определенной величины. Это было сделано, чтобы ограничить среднюю скорость перемещения для сохранения баланса.

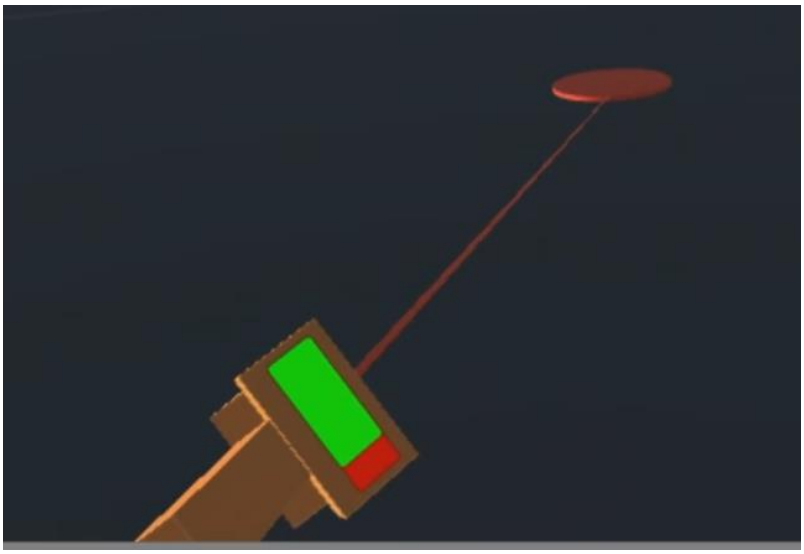

Рисунок 10: Внешний вид телепортера.

2.3 Оружие

Многие образцы оружия (TestGun, TestGun2, RocketLauncher) имеют примерно одинаковое строение. Рассмотрим его на примере TestGun2.

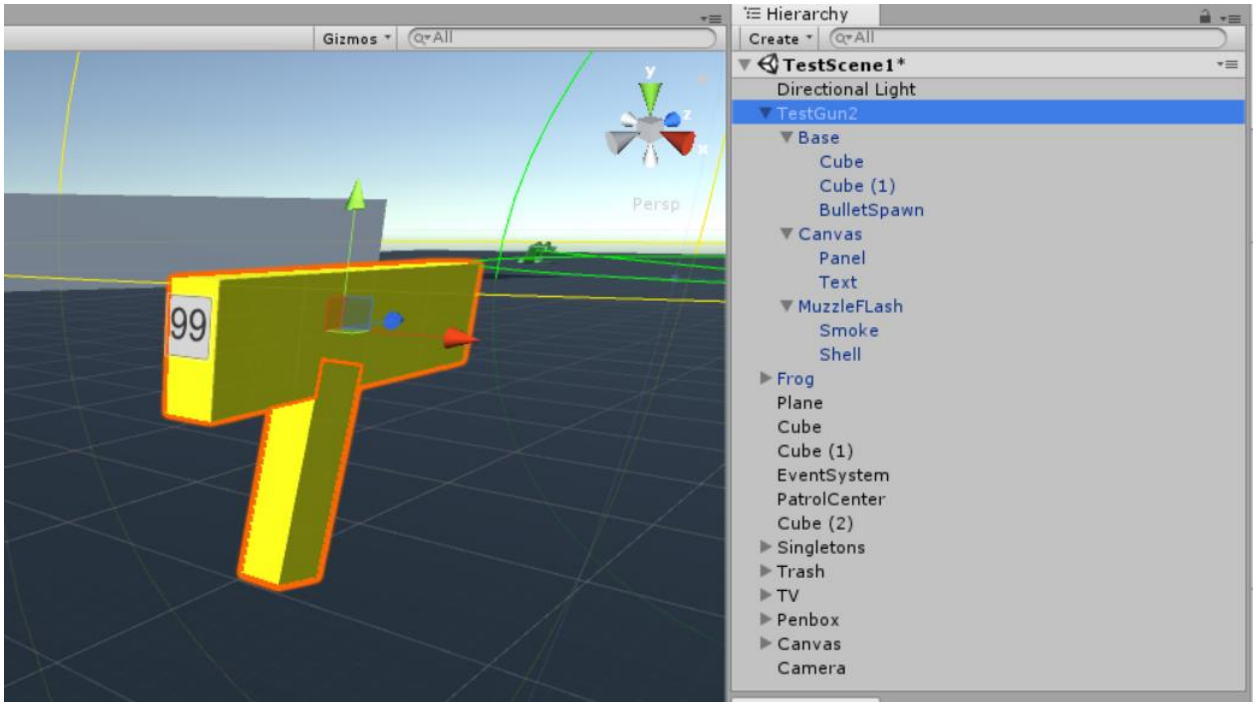

Рисунок 11: Внешний вид и иерархия префаба TestGun.

BaseGun – базовый объект. Несёт в себе основные настройки .

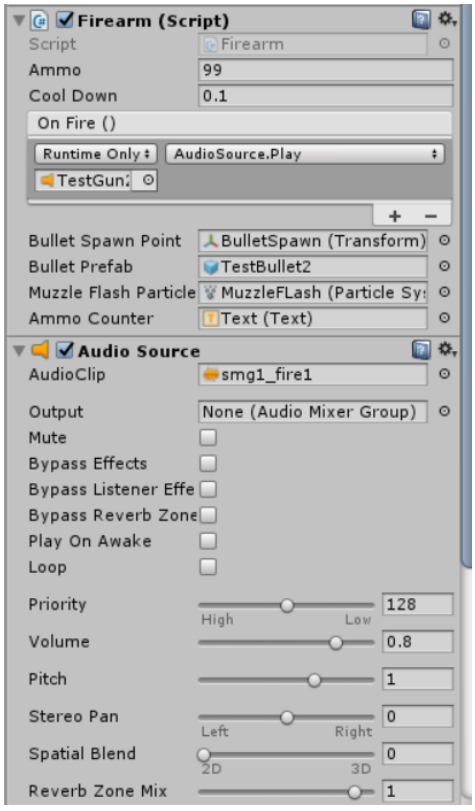

Рисунок 12: Настройки TestGun2.

Base – вершина иерархии для визуальной модели оружия.
BulletSpawn – плейсхолдер, точка появления пули
Canvas – worldpsace ui, содержит индикатор количества оставшихся патронов
MuzzleFlash – Система частиц, включающаяся при выстреле

SniperRifle
Данный префаб представляет из себя снайперскую винтовку. Похож на другие префабы оружия с одним отличием – его дополняют объекты Camera и плоскость Screen, у которой в качестве материала выступает Render Texture

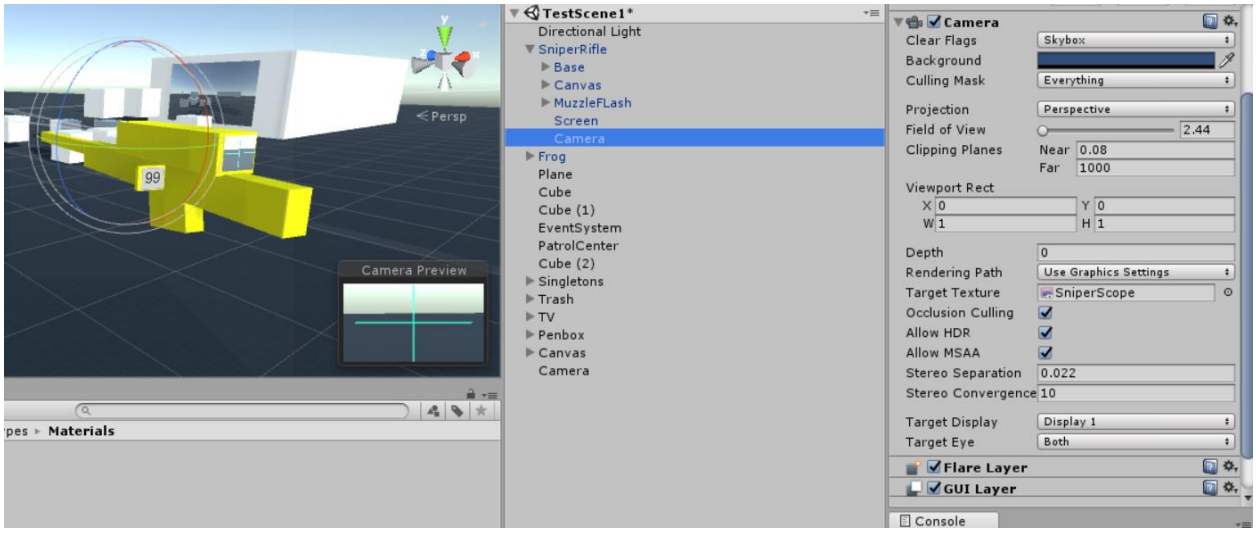

Рисунок 13: SniperRifle

2.4 Пули и ракеты.
Пули и ракеты имеют примерно одинаковое строение. Как правило, это просто сфера с наложенным компонентом StandartBullet или любым другим, унаследованным от него.

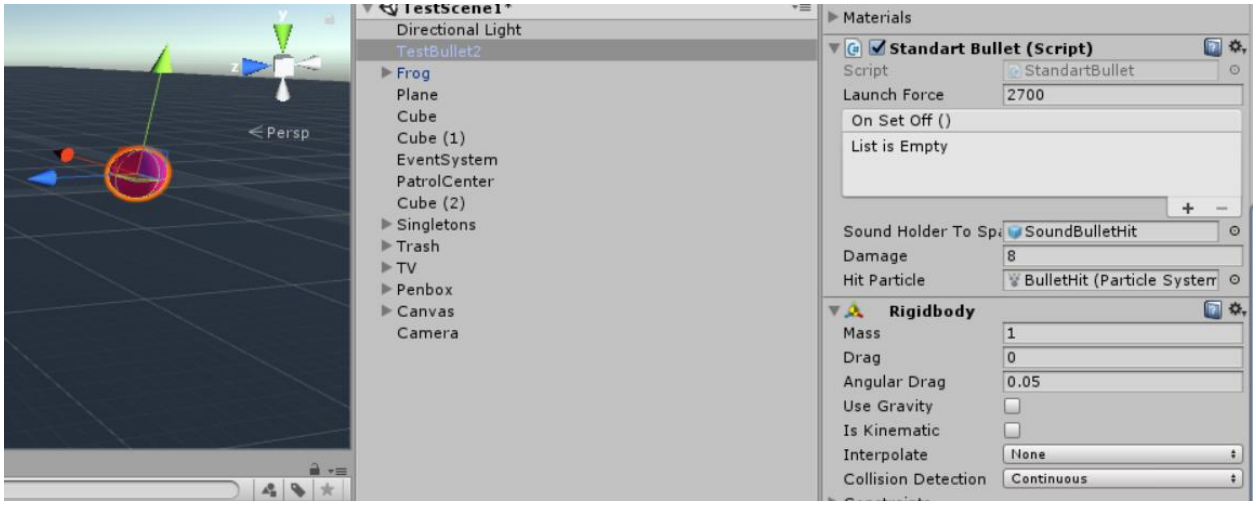

Рисунок 14: Настройки пули

В качестве метода обнаружения столкновения используется Continious. Это не очень хорошо для производительности, но позволяет избежать пролетания пули через стены.

2.5 Рука.

Префаб Hand позволяет игроку взаимодействовать с физическими объектами. Если, касаясь какого-то объекта, игрок нажмет на «выстрелить», физический объект «приклеится» к руке (сделано посредством создания в рантайме нового компонента Fixed Joint).

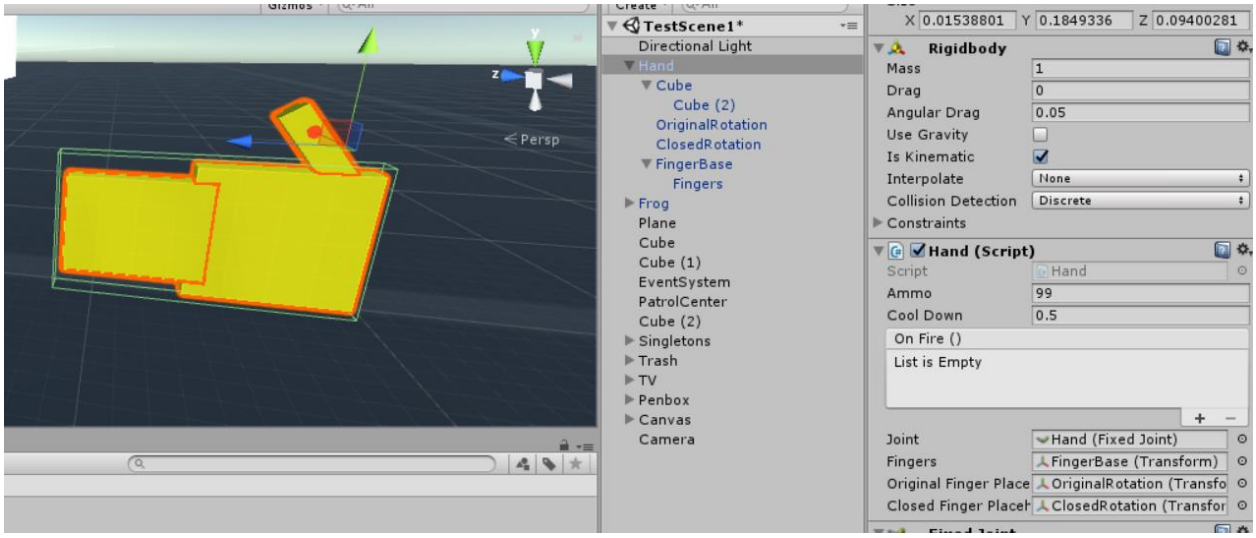

Рисунок 15: Настройки руки

Hand – база руки, содержит настройки и коллайдер.
Cube, Cube2 – ладошка и большой палец
FingerBase – пивот для пальцев
Fingers – пальцы
OriginalRotation – плейсхолдер стандартного вращения пальцев
ClosedRotation – плейсхолдер вращения повёрнутых пальцев.

2.6 Фонарик

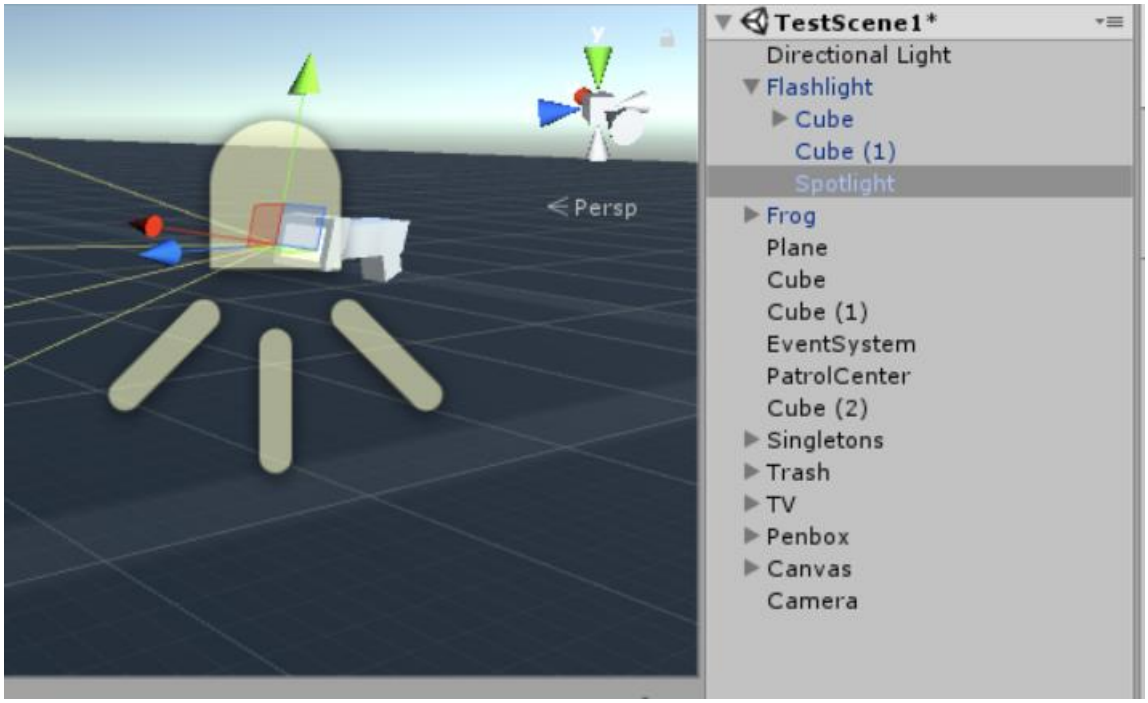

Рисунок 16: Внешний вид и иерархия фонарика

Фонарик – самый простой из префабов оружия. Включает в себя основу (Flashlight), визуальный корпус и источник света Spotlight, который можно включать/выключать, нажав на «выстрелить».

3 Создание лягушки

3.1 Создание и настройка модели лягушки.

Модель лягушки была создана в программе 3dmax. Она была экспортирована вместе с анимациями в двух отельных частях – голова и тело. Это было сделано для того, чтобы голова была отдельным объектом и могла смотреть на игрока.

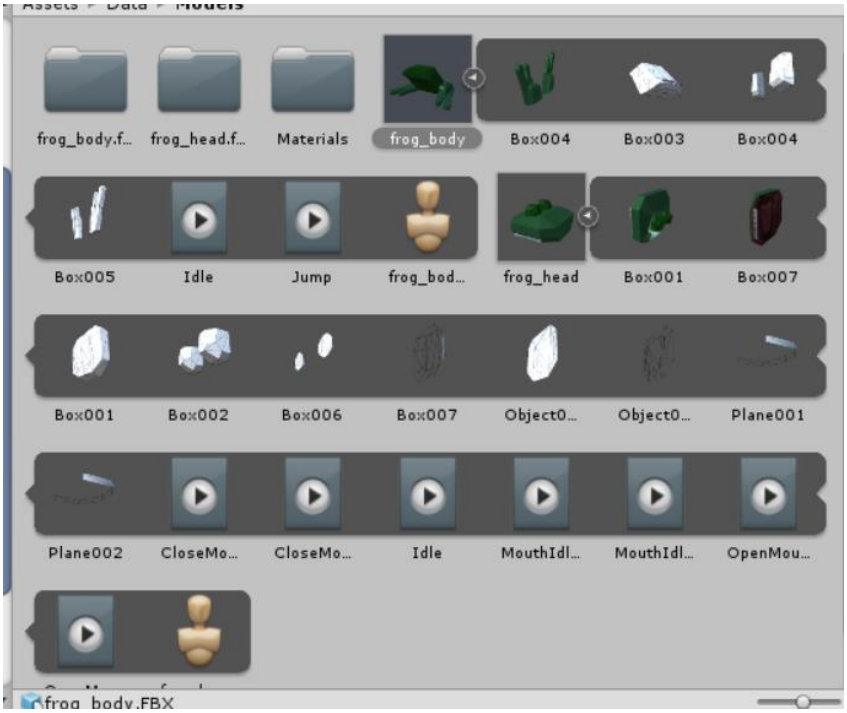

Рисунок 17: Импортированная модель с анимациями.

Для головы и для тела были созданы два отдельных AnimationControllera. Анимации тела: статичная, прыжок.

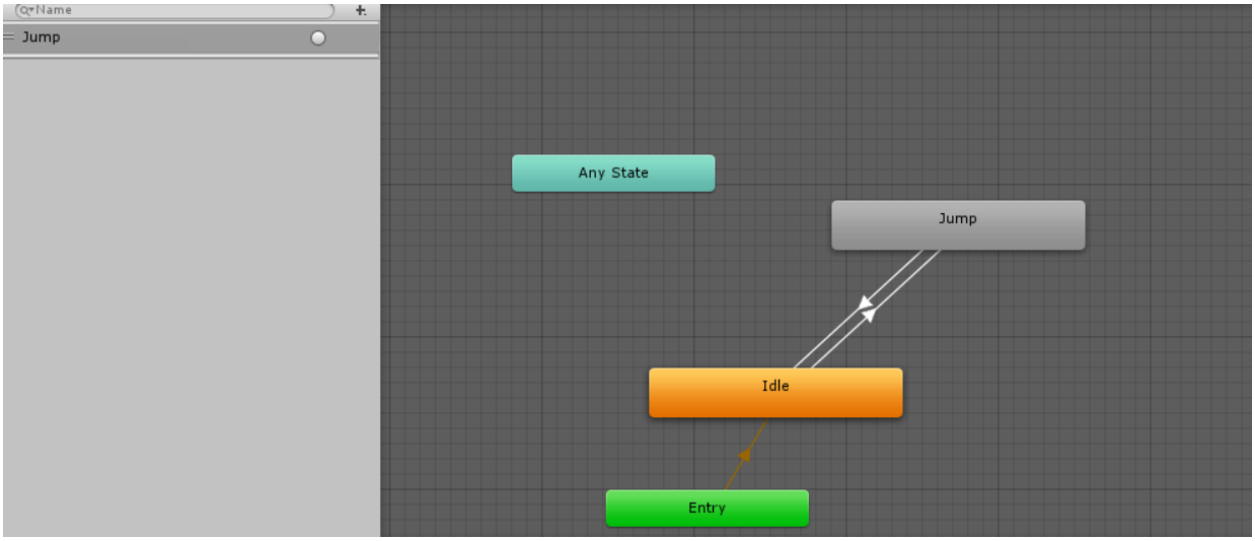

Рисунок 18: Контроллер анимаций тела.

Анимации головы: статичная, открывание рта (2шт), статичная с открытым ртом (2шт), закрывание рта (2шт).

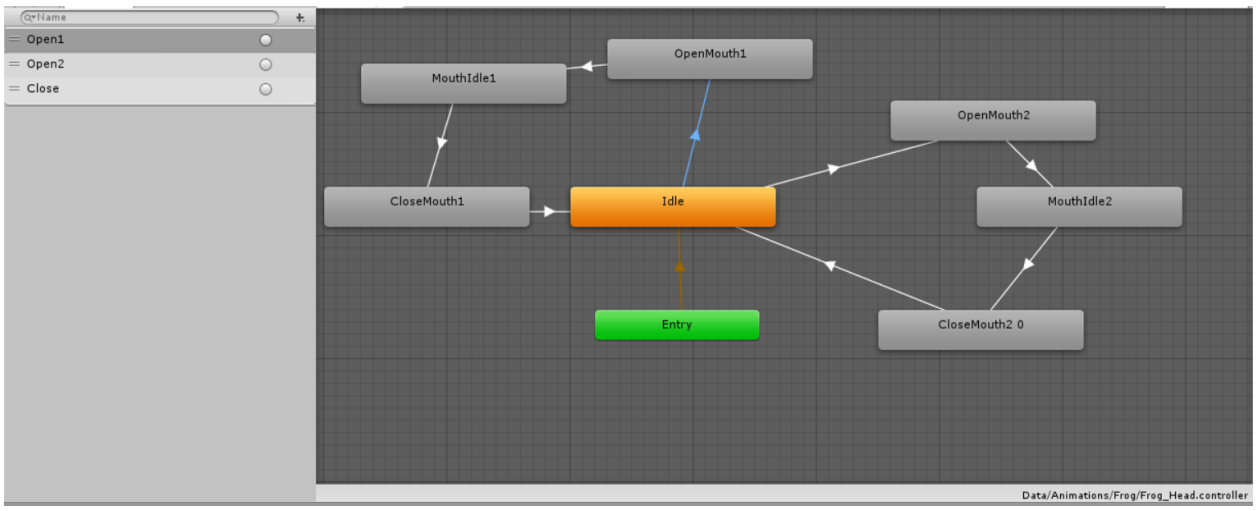

Рисунок 19: Контроллер анимаций головы.

3.2 Разработка искусственного интеллекта.

Для создания искусственного интеллекта было решено использовать паттерн State Machine (граф состояний). Были реализованы следующие состояния:

1) Patrolling – лягушка патрулирует, двигаясь между случайными точками в заданной области.
2) Attack - лягушка движется к цели и атакует её.
3) Seeking – лягушка ищет скрывшуюся цель в точке, в которой цель видели в последний раз.

Вид графа приведен на рисунке

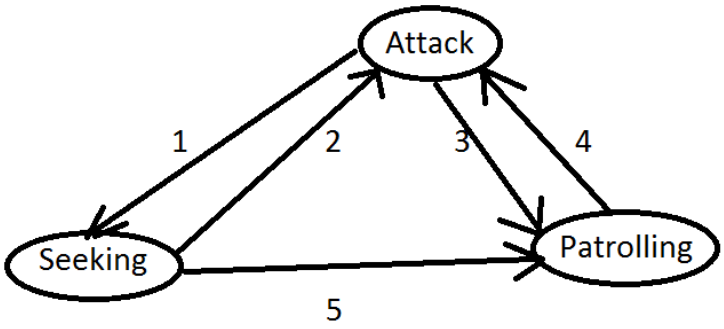

Рисунок 20: Вид графа поведения лягушки.

На рисунке обозначены следующие переходы:

1) Цель скрылась из виду.
2) Цель вновь обнаружена.
3) Цель уничтожена или цель покинула радиус видимости.
4) Цель обнаружена в пределах триггера или лягушка атакована кем-либо.
5) Поиск цели не дал результатов.

3.3 Разработка префаба лягушки.

Префаб лягушки содержит следующие части:

1) Frog – базовый компонент. Содержит в себе коллайдер тела и всю основную логику. Является объектом-родителем для всех остальных частей.

2) frog_body – визуальная модель тела лягушки.
3) FrogHead – базовый объект для головы лягушки. Содержит коллайдер. Его наследником является объект frog_head, являющийся визуальной моделью головы лягушки.
4) Trigger – объект содержащий триггер, при пересечении которого лягушка начинает атаковать игрока.
5) StateText – текст над головой лягушки, показывающий текущее состояние ИИ.

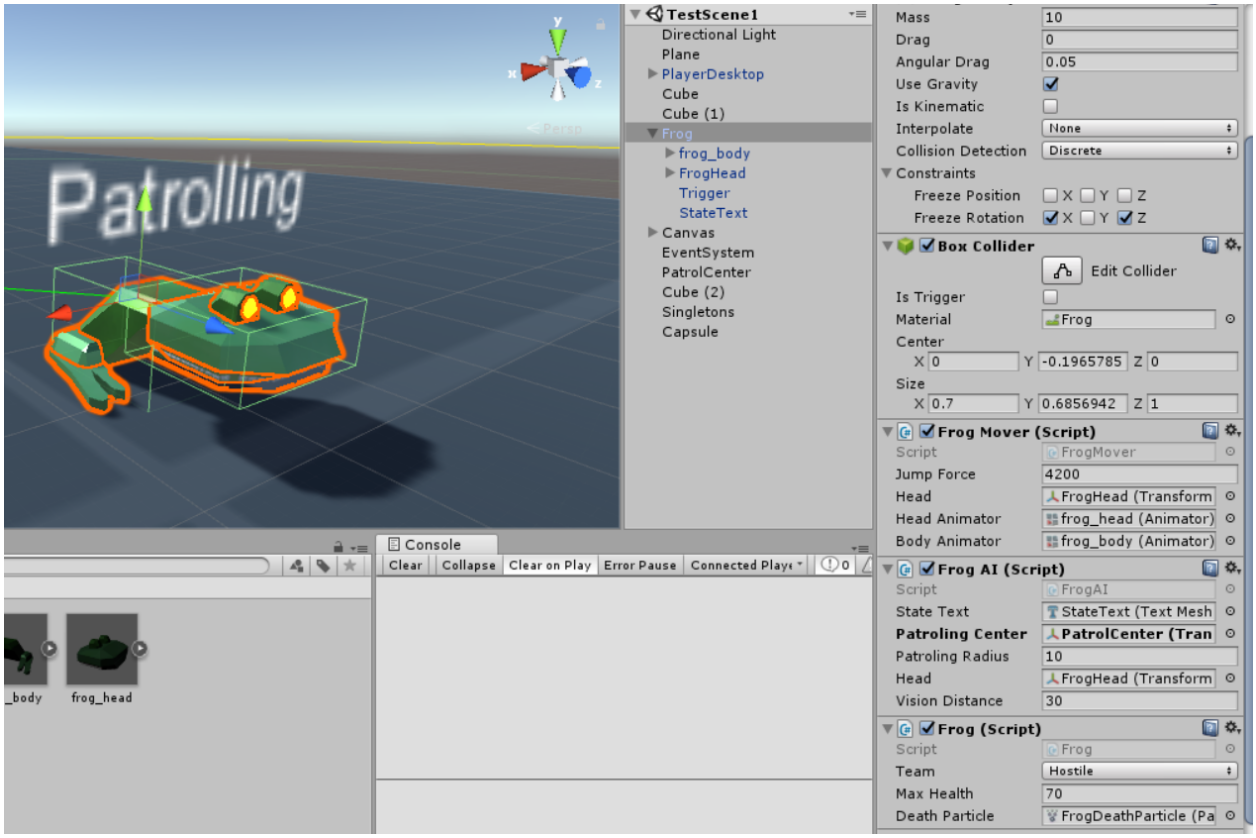

Рисунок 21: Вид префаба лягушки.

Объект Frog содержит следующие компоненты:
1) FrogMover – отвечает за движение лягушки.
2) FrogAI – содержит AI лягушки.
3) Frog – наследник базового класса BaseEntity, содержащий информацию о здоровье и команде объекта.

3.4 Исходный код компонентов.

Frog.cs
```
public class Frog : BaseEntity {

    [SerializeField] ParticleSystem deathParticle;

    protected override void OnDeath()
    {
```

```
        Instantiate(deathParticle, transform.position,
Quaternion.LookRotation(Vector3.up));
        Destroy(this.gameObject);
        base.OnDeath();
    }
}
```

BaseMover.cs

```
public class BaseMover : MonoBehaviour {

    protected Rigidbody objectToMove;

    public virtual void Start()
    {
        objectToMove = GetComponent<Rigidbody>();
    }

}
```

FrogMover.cs

```
public class FrogMover : BaseMover {

    private Rigidbody rb;
    [SerializeField] float jumpForce;
    [SerializeField] Transform head;
    private Transform currentTarget;
       public Transform CurrentTarget
       {
              get {return currentTarget;}
              set {currentTarget = value;}
       }
    float moveTime = 4;
    float moveStarted;
    float nextMove;
    float jumpCooldown = 3f;
    float nextJump;

    [SerializeField] Animator headAnimator;
    [SerializeField] Animator bodyAnimator;

    bool aggressive = true;
    public bool Aggressive
    {
        get {return aggressive;}
        set {aggressive = value;}
    }

    bool canBite = false;

       // Use this for initialization
       public override void Start ()
    {
        base.Start();
        rb = this.GetComponent<Rigidbody>();
        nextMove = Time.time;
        nextJump = Time.time;
        Physics.IgnoreCollision(this.GetComponent<Collider>(),
this.GetComponentsInChildren<Collider>()[1]);
```

```
        }

        // Update is called once per frame
        void FixedUpdate () {

        }

    public void JumpTowards()
    {
        if (Time.time > nextMove)
        {
            nextMove = Time.time + moveTime;
            moveStarted = Time.time;
        }
        if (Time.time > moveStarted + 1 && Time.time < moveStarted + 2)
        {
            BodyMatchHead();
        }
        if (Time.time > moveStarted + 2) //&& LooksAtTarget(currentTarget.position))
        {
            Jump();
        }

    }

    public void Jump()
    {
        if (Time.time > nextJump)
        {
            nextJump = Time.time + jumpCooldown;
            Vector3 forward = head.forward;
            forward.y = 0;
            forward = forward.normalized;
            rb.AddForce((forward + Vector3.up * 0.9f).normalized * jumpForce);
            bodyAnimator.SetTrigger("Jump");
            if (aggressive)
                Bite();
        }
    }

        public bool LookAtTarget ()
    {
        return LookAtTarget(currentTarget.position);
    }


    public bool LookAtTarget (Transform lookTargetTransform)
    {
        return LookAtTarget(lookTargetTransform.position);
    }

    public bool LookAtTarget (Vector3 lookTarget)
    {
        float rotSpeed = 360f;
        Vector3 D = lookTarget - head.transform.position;
        Quaternion rot = Quaternion.Slerp(transform.rotation, Quaternion.LookRotation(D),
rotSpeed * Time.deltaTime);
        head.rotation = rot;
```

```
        float minRotation = -80;
        float maxRotation = 80;
        Vector3 currentRotation = head.localRotation.eulerAngles;
        if (currentRotation.y < 180)
        {
            currentRotation.y = Mathf.Clamp(currentRotation.y, minRotation, maxRotation);
        }
        else
        {
            currentRotation.y = Mathf.Clamp(currentRotation.y, 360+minRotation, 360);
        }
        head.localRotation = Quaternion.Euler (currentRotation);

        return LooksAtTarget(lookTarget);
    }

    public void BodyMatchHead()
    {
        float rotSpeed = 4f;
        Quaternion rot = Quaternion.Slerp(transform.rotation, head.rotation, rotSpeed *
Time.deltaTime);
        //rot.eulerAngles = new Vector3(0, rot.eulerAngles.y, 0);
        transform.eulerAngles = new Vector3(0, rot.eulerAngles.y, 0);
    }

    public bool LooksAtTarget (Vector3 lookTarget)
    {
        Quaternion targetRotation = Quaternion.LookRotation (lookTarget - head.position);
        Quaternion currentRotation = head.rotation;
        float rotatingError = Mathf.Abs (targetRotation.eulerAngles.y -
currentRotation.eulerAngles.y);
        if (rotatingError < 5)
            return true;
        else
            return false;
    }

    void OnCollisionEnter (Collision col)
    {
        //Debug.Log(string.Format("Frog collided with {0}", col.collider.gameObject));
        headAnimator.SetTrigger("Close");
        if (canBite)
        {
            BaseEntity enemy = col.collider.gameObject.GetComponent<BaseEntity>();
            //Debug.Log("pam");
            if (enemy != null && enemy.Team != this.GetComponent<BaseEntity>().Team)
            {
                enemy.TakeDamage(15);
                //Debug.Log("poom");
            }
        }
        canBite = false;
    }

    void Bite ()
    {
        canBite = true;
        if (Physics.Raycast(head.position, head.forward, 8.0f))
        {
```

```
            int c = Random.Range(0, 2);
            switch (c)
            {
                case 0:
                    headAnimator.SetTrigger("Open1");
                    break;
                case 1:
                    headAnimator.SetTrigger("Open2");
                    break;
            }
        }
    }
}
```

FrogAI.cs

```
public class FrogAI : BaseAI {

       public enum FrogAiStates
       {
              Patroling, AttackNear, AttackFar, Seeking
       }

       FrogAiStates currentState = FrogAiStates.Patroling;
    private FrogAiStates CurrentState
    {
        get { return currentState; }
        set
        {
            currentState = value;
            stateText.text = string.Format("{0}", currentState);
        }
    }

    [SerializeField] TextMesh stateText;

       [SerializeField] private Transform patrolingCenter;
       private Vector3 currentTarget;

       [SerializeField] private float patrolingRadius;
    [SerializeField] private Transform head;
    [SerializeField] private float visionDistance = 30.0f;

       private BaseEntity currentTargetEntity;
       private FrogMover mover;
    private Vector3 lastKnownLocation;

       void Start()
       {
              mover = this.GetComponent<FrogMover>();
        currentTarget = patrolingCenter.position;
        FindObjectOfType<EventHub>().EntityDeathEvent += new
EntityDeathHandler(EntityDeathDetected);
       }


       void Update ()
       {
        //Debug.Log(currentState);
              switch (currentState)
```

```
        {
                    case (FrogAiStates.Patroling):
                           PatrolingState();
                           break;
                    case (FrogAiStates.AttackNear):
                           AttackNearState();
                           break;
                    case (FrogAiStates.AttackFar):
                           AttackFarState();
                           break;
                    case (FrogAiStates.Seeking):
                           SeekingState();
                           break;
        }

        if (currentTargetEntity != null &&
Vector3.Distance(currentTargetEntity.transform.position, transform.position) > visionDistance)
        {
            CancelInvoke();
            CurrentState = FrogAiStates.Patroling;
            currentTargetEntity = null;
        }
       }

    void EntityDeathDetected (BaseEntity died)
    {
        if (currentTargetEntity == died)
        {
            CurrentState = FrogAiStates.Patroling;
            currentTargetEntity = null;
        }
    }


       void PatrolingState()
       {
              Vector3 frogPosition = new Vector3 (transform.position.x, 0,
transform.position.z);
              Vector3 targetPosition = new Vector3 (currentTarget.x, 0, currentTarget.z);
              float distanceToTarget = Vector3.Distance(frogPosition, targetPosition);

              if (distanceToTarget > 3)
              {
            mover.Aggressive = false;
            mover.LookAtTarget(currentTarget);
                    mover.JumpTowards();
              }
              else
              {
                    currentTarget = GetPointInCircle(patrolingCenter.position,
patrolingRadius);
              }
       }

       void AttackNearState()
       {
              mover.Aggressive = true;
              mover.LookAtTarget(currentTargetEntity.transform.position);
              mover.JumpTowards();
```

```
    if (!IsInvoking())
        InvokeRepeating("CheckTargetVisible", 0.0f, 0.5f);
   }

   void AttackFarState()
   {

   }

   void SeekingState()
{
    mover.Aggressive = false;
    mover.LookAtTarget(currentTarget);
    mover.JumpTowards();
    float distanceToTarget = Vector3.Distance(transform.position, currentTarget);
    if (distanceToTarget < 3)
    {
        CurrentState = FrogAiStates.Patroling;
        //currentTargetEntity = null;
    }
   }

   void OnTriggerEnter (Collider col)
{
    CheckIfEnemy(col);
   }

void OnTriggerStay (Collider col)
{

}

void CheckIfEnemy(Collider col)
{
    BaseEntity other = col.gameObject.GetComponent<BaseEntity>();
    if (other == null)
        return;
    if (other == this.GetComponent<BaseEntity>())
        return;
    else
    {
        if (other.Team != this.GetComponent<BaseEntity>().Team)
        {
            CurrentState = FrogAiStates.AttackNear;
            mover.CurrentTarget = other.transform;
            currentTargetEntity = other;
        }

    }
}

   void CheckTargetVisible()
   {
    if (currentTargetEntity == null)
    {
        CancelInvoke();
        CurrentState = FrogAiStates.Patroling;
        return;
    }
```

```
            RaycastHit hit;
        if (Physics.Linecast(head.position, currentTargetEntity.transform.position, out hit))
            {
            //Debug.Log(string.Format("Linecast hit {0}", hit.collider.gameObject));
            BaseEntity other = hit.collider.gameObject.GetComponent<BaseEntity>();
            if (other != currentTargetEntity && other != this.GetComponent<BaseEntity>())
            {
                if (CurrentState == FrogAiStates.AttackNear)
                {
                    CurrentState = FrogAiStates.Seeking;
                    currentTarget = hit.point;
                }
            }
            else if (other == currentTargetEntity)
            {
                CurrentState = FrogAiStates.AttackNear;
            }
            }
        }

        public override void AttackedBy(BaseEntity attacker)
    {
        //Debug.Log(string.Format("attacked by {0}",attacker));
        if (attacker != null)
        {
            currentTargetEntity = attacker;
            CurrentState = FrogAiStates.AttackNear;
        }
    }

        public Vector3 GetPointInCircle(Vector3 center, float radius)
    {
        float x, z;
        do
        {
            x = Random.Range(center.x - radius, center.x + radius);
            z = Random.Range(center.z - radius, center.z + radius);
        }
        while ((x-center.x)*(x-center.x) + (z-center.z)*(z-center.z) >= radius*radius);
        //Debug.Log(string.Format("{0} units need a circle with radius {1}. Giving coordinates
{2}.", numberOfUnits, circleRadius, new Vector3(x, center.y, z)));
        return new Vector3(x, center.y, z);
    }

    void OnDrawGizmos()
    {
        Gizmos.color = Color.green;
        Gizmos.DrawWireSphere(patrolingCenter.position, patrolingRadius);
        Gizmos.DrawLine(transform.position, currentTarget);
        Gizmos.color = Color.yellow;
        Gizmos.DrawWireSphere(transform.position, visionDistance);

    }
}
```

4 Демонстрация работы программы.

При запуске программы игрок попадает в сцену и может использовать весь свой арсенал оружия и инструментов. В сцене так же прыгает лягушка.

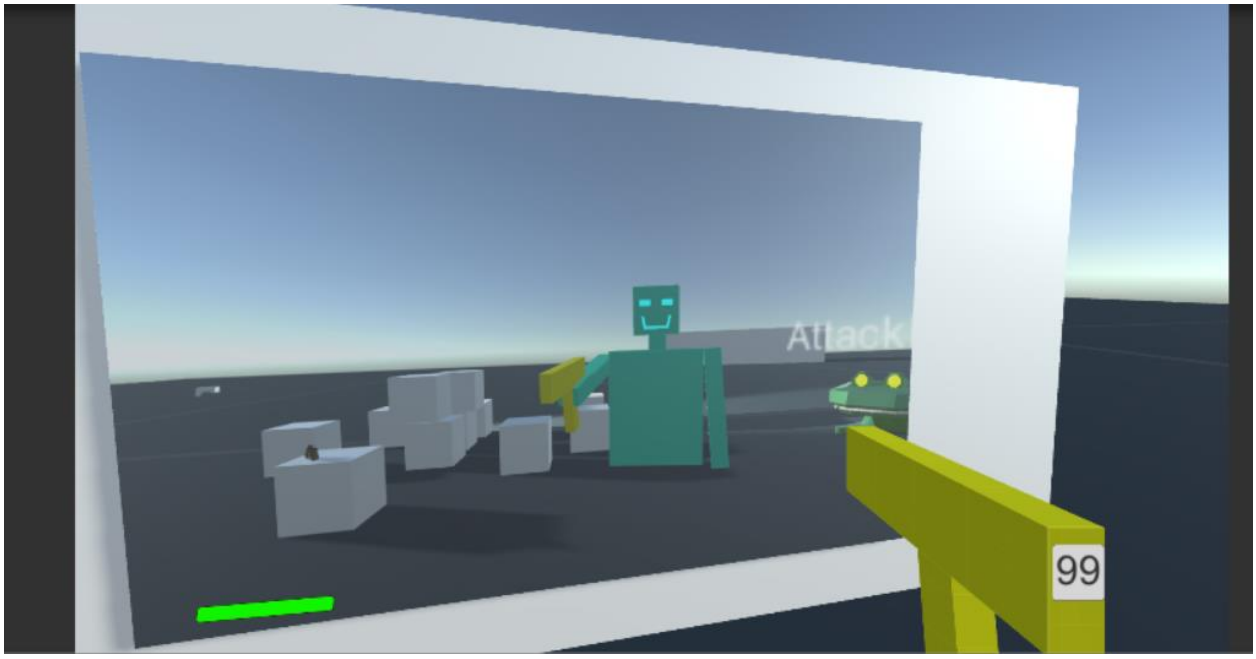

Рисунок 22: Игрок зашел с десктопа

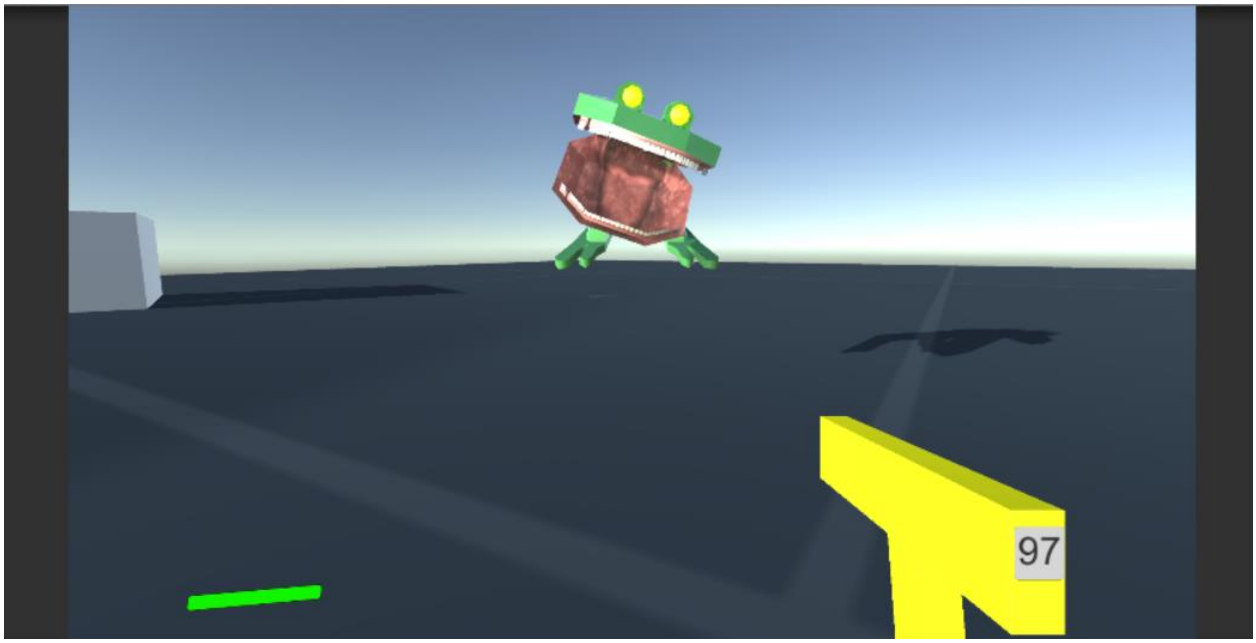

Рисунок 23: Лягушка атакует игрока

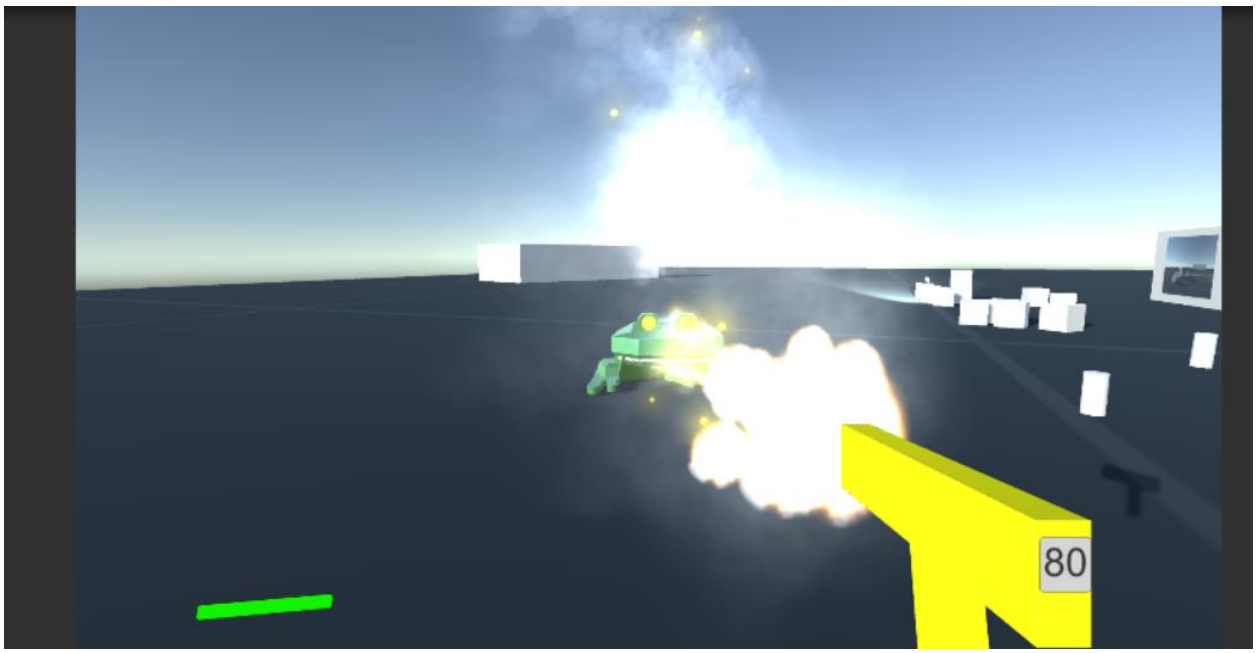

Рисунок 24: Игрок атакует лягушку

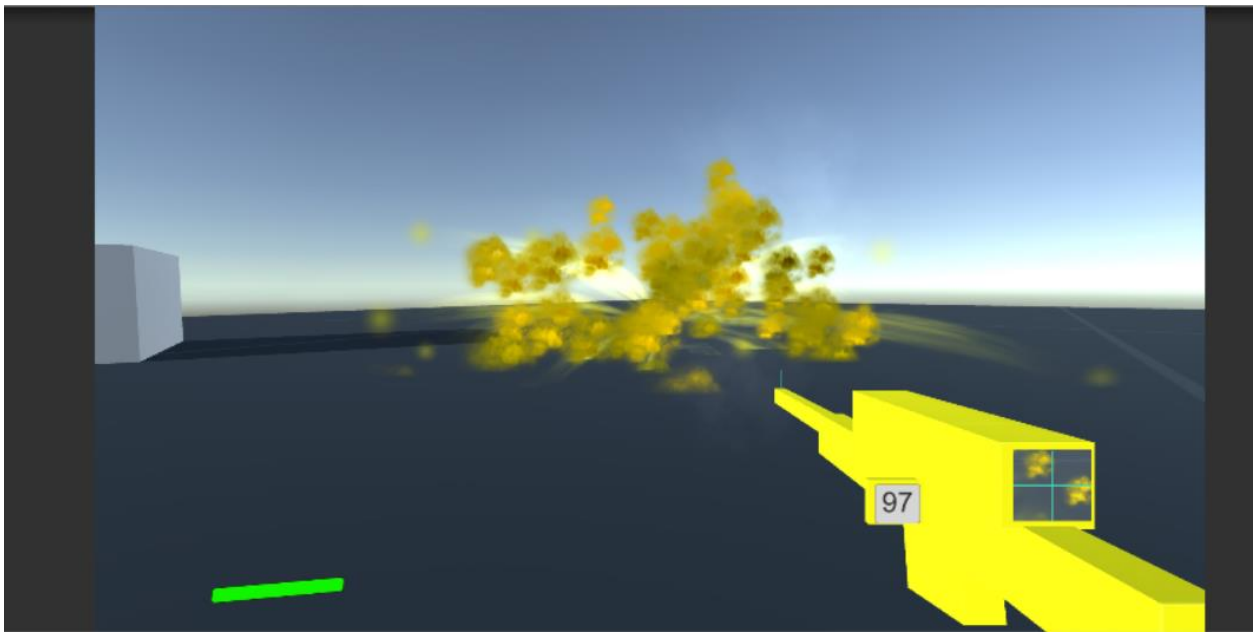

Рисунок 25: Игрок убил лягушку

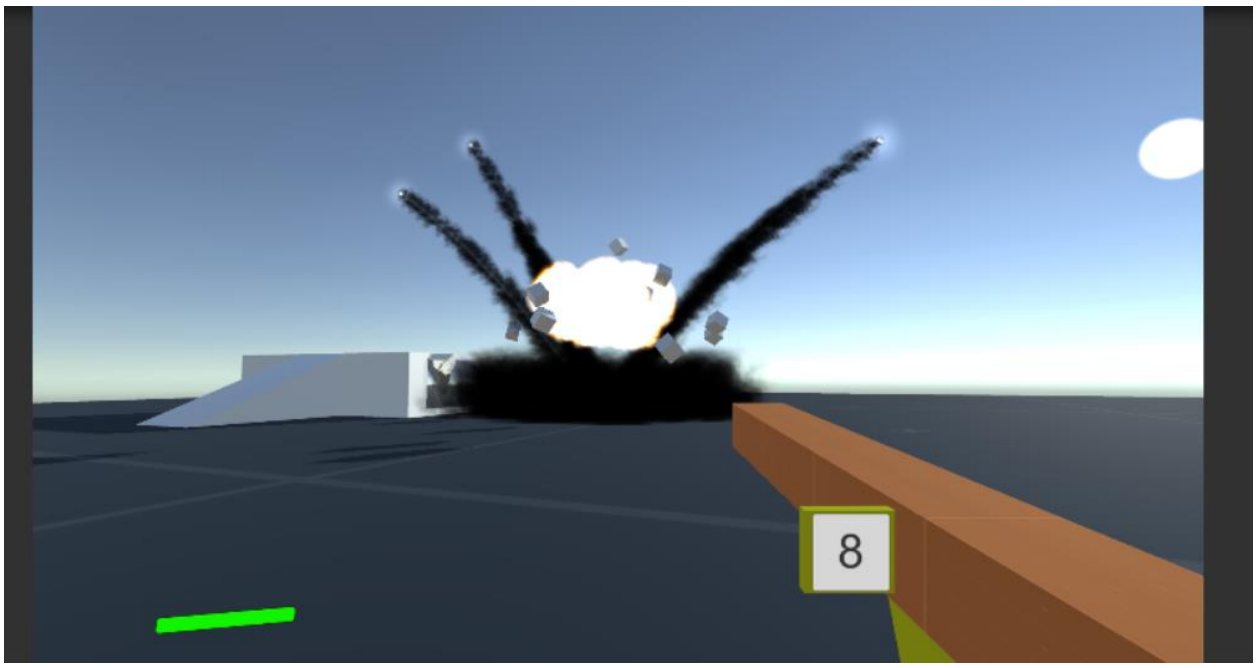

Рисунок 26: Взрыв разбрасывает ящики

При нажатии на Escape появляется меню паузу. Игрок может продолжить игру или выйти из игры.

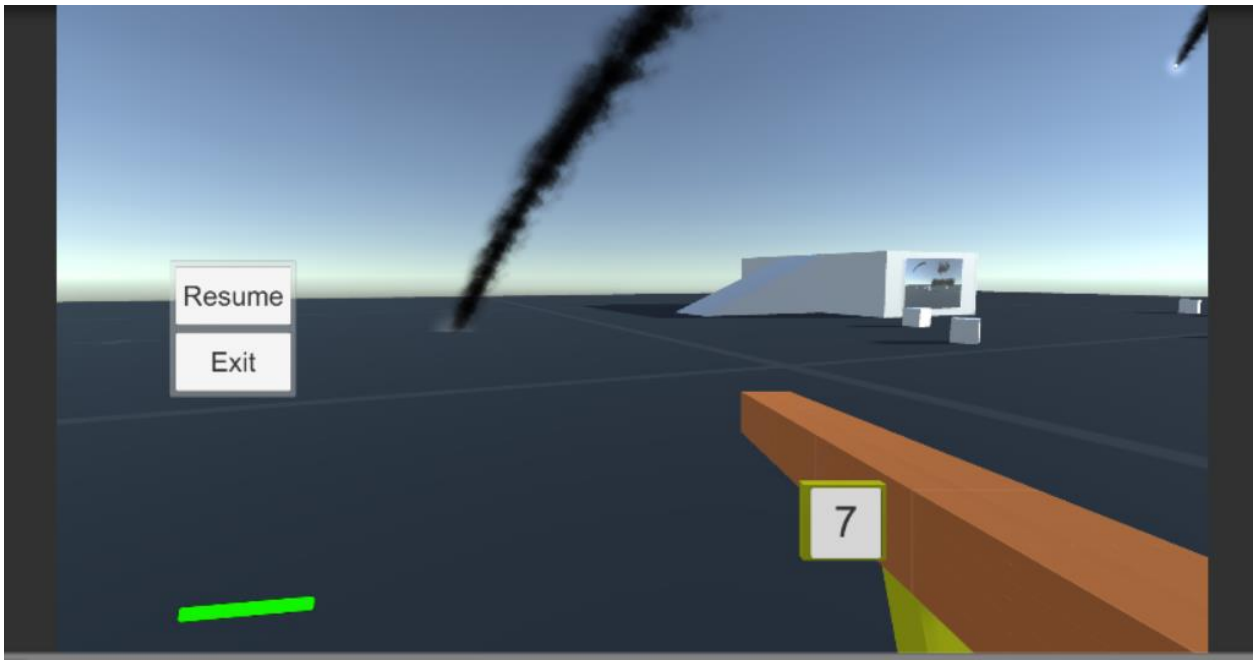

Рисунок 27: Меню паузы

5. ВКРМ

5.1 Анализ предметной области задания.

Дополненная реальность (Augmented Reality) расширяет (дополняет) различные ощущения реального мира виртуальными (компьютерно-генерируемыми) объектами, которые сосуществуют и взаимодействуют в одном и том же пространстве, как в реальном мире.

Данная технология дополняет, например, поле зрения пользователя посредством компьютерных устройств соответствующей информацией, что позволяет ему получать со

своей точки зрения и в соответствии с индивидуальными настройками необходимые для работы сведения о наблюдаемых объектах и помощь в решении поставленных задач.
В настоящее время большинство систем дополненной реальности реализуется с использованием «живого» цифрового видео, которое обрабатывается и дополняется компьютерно-генерируемыми изображениями.
Системы дополненной реальности имеют следующие свойства:

- комбинируют реальные и виртуальные объекты в реальной среде;
- работают в интерактивном режиме в реальном времени;
- реалистично совмещают (регистрируют) реальные и виртуальные объекты друг с другом.

Существуют различные подходы к созданию AR приложений. Самой распространённой является ***маркерная технология***. Использует статические метки/маркеры для запуска вывода данных. Подразумевает использование определённого маркера в виде ключа для активации 3D-объекта. Программная среда распознаёт через камеру маркер либо объект, который перед ней находится, после чего выводит поверх него 3D-модель или изображение.
У разных программ свои методы распознавания изображений, что накладывает определённые требования к подготовке и использованию. Маркерами могут выступать: QR-коды, сгенерированные точки, сохранённые изображения, логотипы брендов и так далее. В данный момент технологии позволяют распознавать как 2D, так и 3D-маркеры.

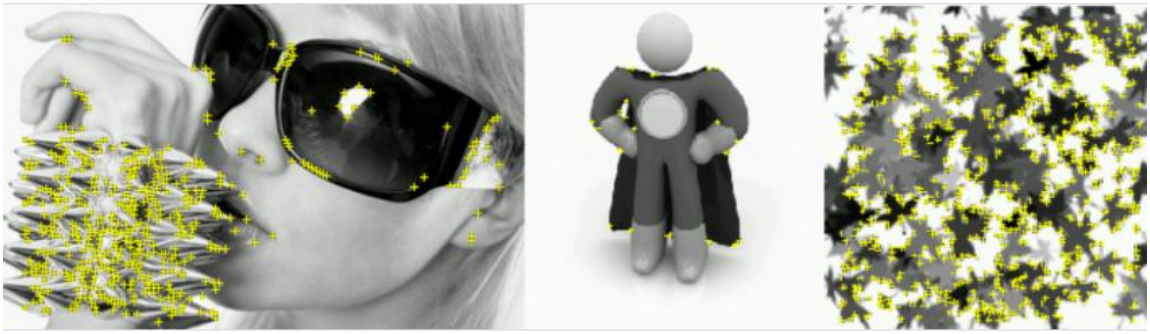

Рисунок 28: Ключевые точки маркера в Vuforia.

Жёлтые крестики — это точки, которые будет искать программа во время сканирования картинки для поиска в своей базе соответствующего узора и затем выводить поверх него изображение. Как видно на втором примере, при создании хороших меток возникают сложности (чем больше жёлтых крестиков — тем быстрее происходит распознавание и качественнее вывод информации).
Данная технология позволяет добиться высокого уровня распознавания при условии наличия маркера в кадре.

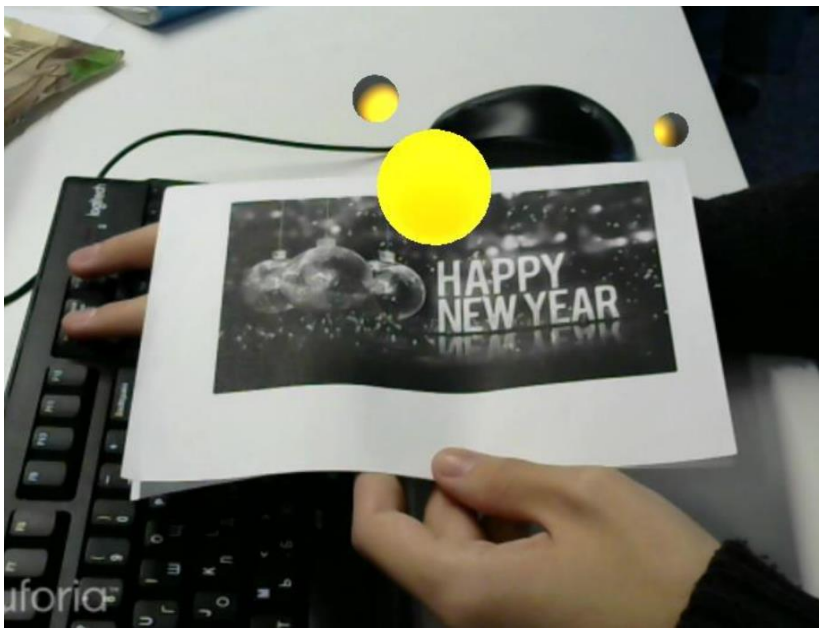

Рисунок 29: Пример изображения, полученного с помощью Vuforia.

Пространственный маппинг (Spatial Mapping) – процесс создания модели объектов реального мира в виртуальном мире. Данный метод отслеживания положения устройства в пространстве доступен для шлема дополненной реальности Microsoft Hololens. Благодаря наличию различных датчиков, данный шлем позволяет в реальном времени строить трёхмерную модель окружающего пространства и определять своё положение в нём.

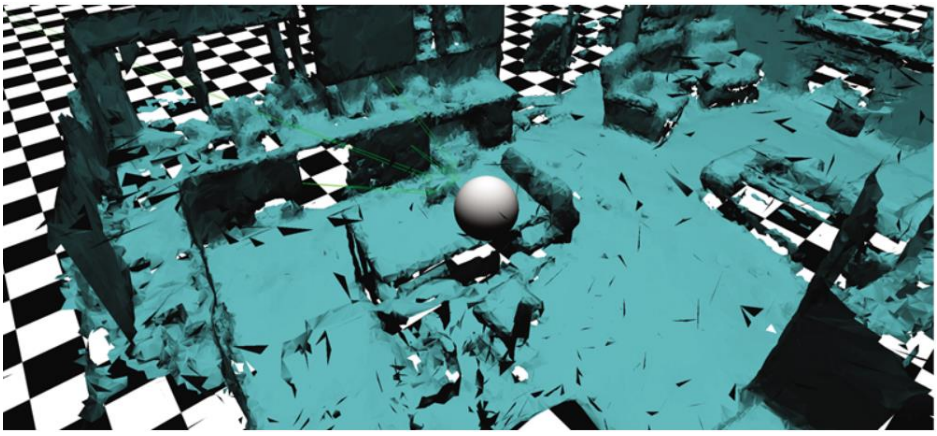
Рисунок 30: Трёхмерная модель комнаты, распознанная Microsoft Hololens.

5.2 Постановка задачи.

Такие методы создания дополненной реальности, как маркерная дополненная реальность и пространственный маппинг, имеют свои преимущества и недостатки, приведенные в таблице:

	Маркерная ДР	Пространственный мэппинг
Высокая точность определения местоположения	+	+
Возможность смотреть в сторону от объекта	-	+

Возможность осмотреть объект со всех сторон	-	+
Возможность распознать конкретный объект в пространстве	+	-
Возможность различать разные объекты одинаковой формы	+	-

Таблица 1: Сравнение маркерной ДР и ДР на основе пространственного мэппинга

Как видно из таблицы, каждый метод имеет как серьезные преимущества, так и серьезные недостатки. Целью данной научно исследовательской работы будет объединение данных технологий в одном приложении, с целью получения системы, имеющей преимущества обеих технологий.

5.3 Выбор применяемых технологий и инструментов.

Рисунок 31: Логотип Vuforia.

Vuforia — это платформа дополненной реальности и инструментарий разработчика программного обеспечения дополненной реальности (Software Development Kit — SDK) для мобильных устройств, разработанные компании Qualcomm. Vuforia использует технологии компьютерного зрения, а также отслеживания плоских изображений и простых объёмных реальных объектов (к примеру, кубических) в реальном времени. С версии 2.5 Vuforia распознаёт текст, а с 2.6 — имеет возможность распознавать цилиндрические маркеры

Возможность регистрации изображений позволяет разработчикам располагать и ориентировать виртуальные объекты, такие, как 3D-модели и медиаконтент, в связке с реальными образами при просмотре через камеры мобильных устройств. Виртуальный объект ориентируется на реальном образе так, чтобы точка зрения наблюдателя относилась к ним одинаковым образом для достижения главного эффекта — ощущения, что виртуальный объект является частью реального мира.

Vuforia поддерживает различные 2D- и 3D-типы мишеней, включая безмаркерные Image Target, трёхмерные мишени Multi-Target, а также реперные маркеры, выделяющие в сцене объекты для их распознавания. Дополнительные функции включают обнаружение преград с использованием так называемых «Виртуальных кнопок» («Virtual Buttons»), детектирование целей и возможность программно создавать и реконфигурировать цели в рамках самомодифицирующегося кода.

Vuforia предоставляет интерфейсы программирования приложений на языках C++, Java, Objective-C, и .Net через интеграцию с игровым движком Unity. Таким образом SDK поддерживает разработку нативных AR-приложений для iOS и Android, в то же время предполагая разработку в Unity, результаты которой могут быть легко перенесены на обе платформы. Приложения дополненной реальности, созданные на платформе Vuforia, совместимы с широким спектром устройств, включая iPhone, iPad, смартфоны и планшеты на Android с версии 2.2 и процессором, начиная с архитектур ARMv6 или 7 с возможностью проведения вычислений с плавающей запятой.

Рисунок 32: Внешний вид Microsoft Hololens.

Microsoft Hololens - очки дополненной реальности, разработанные и производимые компанией Microsoft. Причина популярности Hololens в том,, что он является одним из первых компьютеров, работающих на "Платформе Дополненной Реальности Windows" (Windows Mixed Reality) под операционной системой Windows 10.
Hololens имеет высокотехнологичную техническую начинку, в которую входят:
- Модуль определения положения и ориентации (inertial measurement unit), включающий акселерометр, гироскоп и магнитометр.
- Четыре сенсора распознавания окружения ("environment understanding sensors"), по два с каждой стороны.
- Энергоэффективная камера измерения глубины с углом обзора 120 на 120 градусов для восприятия жестов пользователя.
- Камера для съёмки фото и видео с разрешением 2.4 мегапикселя.
- Массив из четырёх микрофонов.
- Два датчика уровня освещенности.
- Центральный процессор (CPU) и видеокарта (GPU) объединены в чипе типа "система на кристалле" и используют 1 гигабайт оперативной памяти формата LPDDR3. Данный модуль используется для работы операционной системы Windows 10.
- Специально разработанный для Hololens сопроцессор "Модуль Обработки Голограмм" (Microsoft Holographic Processing Unit (HPU)) для обработки данных с датчиков, имеющий 1 гигабайт собственной оперативной памяти. Этот модуль также используется для таких

задач, как построение модели окружающего пространства, распознавание жестов, голоса и речи.
- Модули Wi-Fi и Bluetooth для беспроводной связи.

Рисунок 33: Логотип Unity.

Unity — это инструмент для разработки двух- и трёхмерных приложений и игр, работающий под операционными системами Windows, Linux и OS X. Редактор Unity имеет простой Drag&Drop интерфейс, который легко настраивать. Базовая версия редактора Unity имеет обширный инструментарий, ускоряющий разработку благодаря режиму игры, доступному прямо в интерфейсе Unity. Этот редактор доступен для Windows и Mac, имеет инструменты как для художников, так и для программистов. Редактор Unity поддерживает установку расширений, создаваемых в соответствии с потребностями разработчика или загружаемых из Asset Store, где можно найти многие ресурсы, инструменты и расширения, ускоряющие разработку проекта.

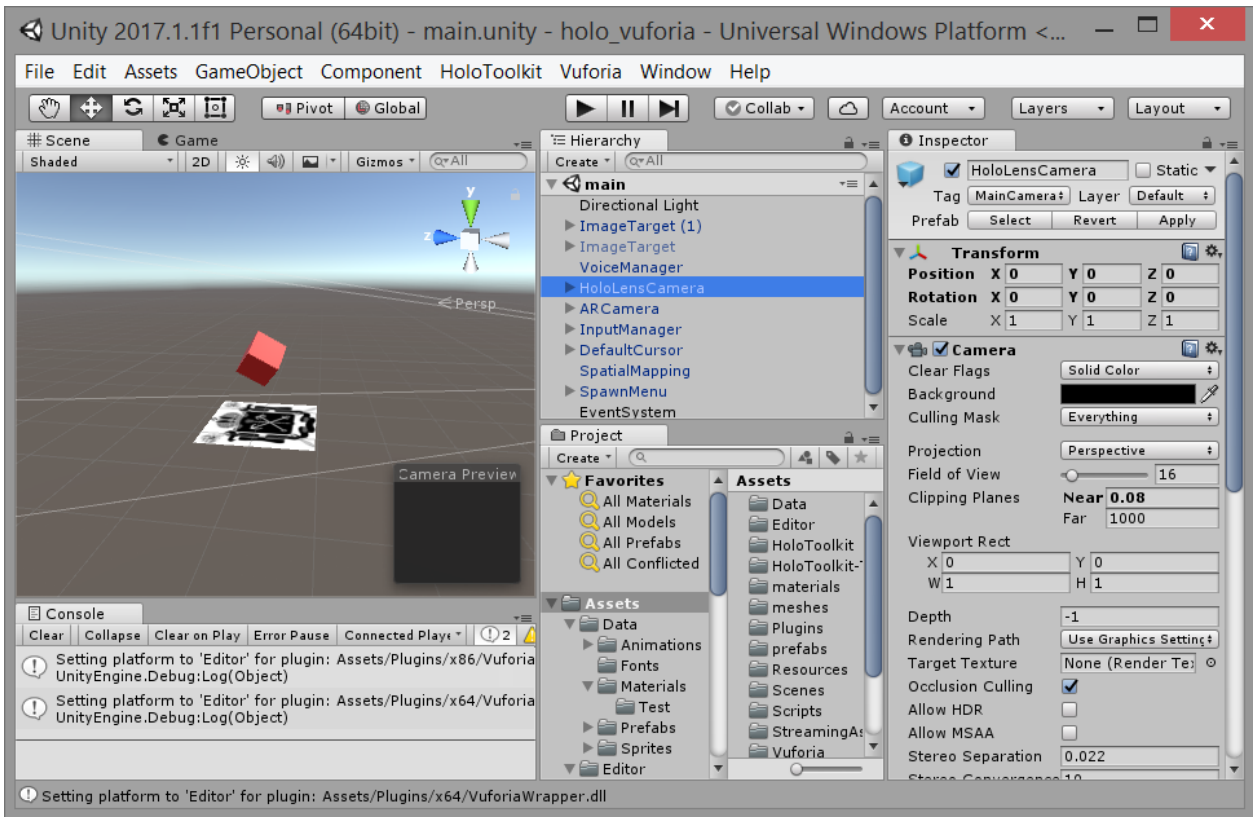

Рисунок 34: Вид окна редактора Unity.

Unity был выбран в качестве основного движка так как он содержит плагины для работы как с Vuforia, так и с Microsoft Hololens, а гибкость движка позволяет комбинировать и связывать различные плагины в рамках одного проекта. Был выбран движок версии 2017.1.1. так как он обеспечил наиболее стабильную работу.

Рисунок 35: Логотип MixedRealityToolkit.

MixedRealityToolkit-Unity – бесплатный набор скриптов и компонентов, представляющий удобный интерфейс для создания приложений для устройств виртуальной и/или дополненной реальности. Включает инструменты для работы с вводом, звуком, пространственным маппингом и т.д.

Рисунок 36: Логотип 3ds Max.

Autodesk 3ds Max - (ранее 3D Studio MAX) — полнофункциональная профессиональная программная система для создания и редактирования трёхмерной графики и анимации, доразработанная компанией Autodesk. Содержит самые современные средства для художников и специалистов в области мультимедиа. Работает в семействе операционных систем Windows (как 32-, так и 64-битных).

5.4 Создание маркера для Vuforia.

В качестве первого маркера было выбрано перекрашенное изображение герба Санкт-Петербурга:

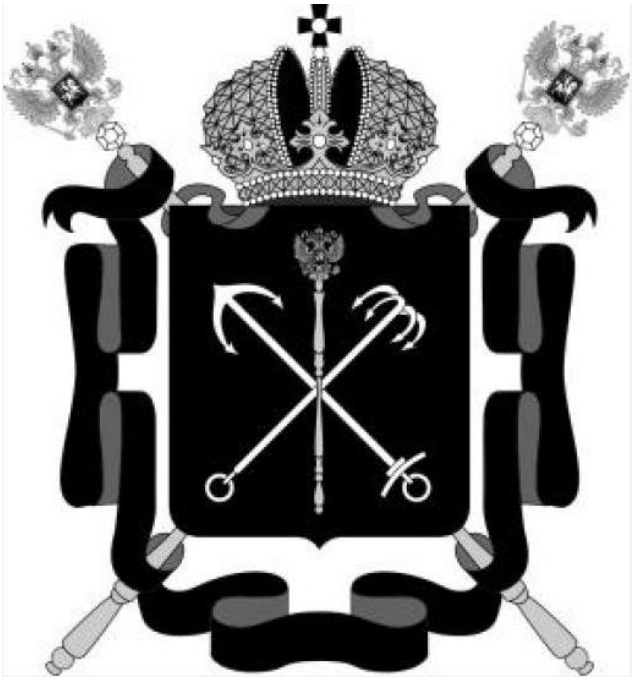

Рисунок 37: Первый маркер.

Данный маркер получил оценку своего качества - максимальные 5 звезд.

Рисунок 38: Ключевые точки, обнаруженные на маркере.

5.5 Создание сцены для Hololens и Vuforia.

При создании сцены были использованы следующие объекты из префабов HoloToolkit:

HoloLensCamera – аналог камеры, дополненный возможностью эмуляции Hololens в окне редактора Unity.

DefaultCursor – стандартный AR-курсор для Hololens

SpatialMapping – префаб, позволяющий загрузить в сцену меш окружающего пространства, построенный сенсорами Hololens.
InputManager – префаб, позволяющий использовать сигналы ввода с Hololens, а также эмулировать их в редакторе Unity.

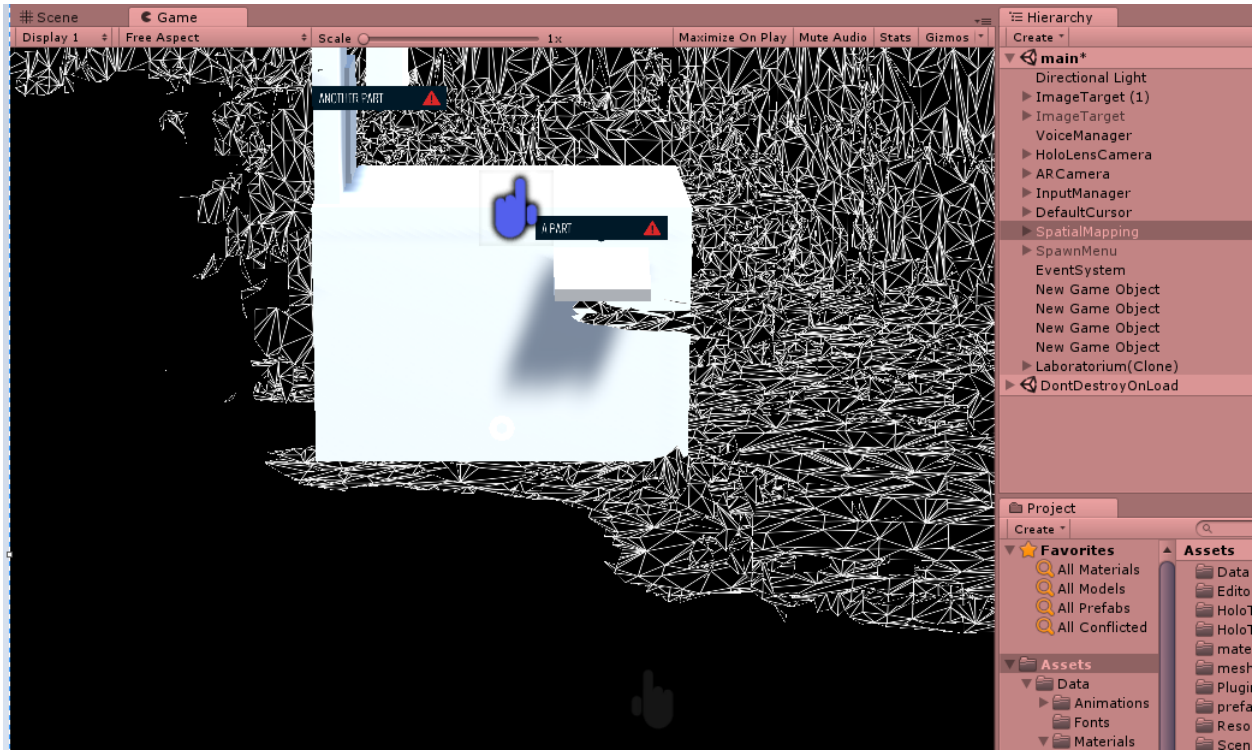

Рисунок 39: Эмуляция Hololens в редакторе Unity.

Для подключения Vuforia были использованы следующие префабы:
ARCamera – указывает Vuforia, какую камеру использовать для съёмки AR.
ImageTarget – представление AR маркера в виртуальном пространстве.

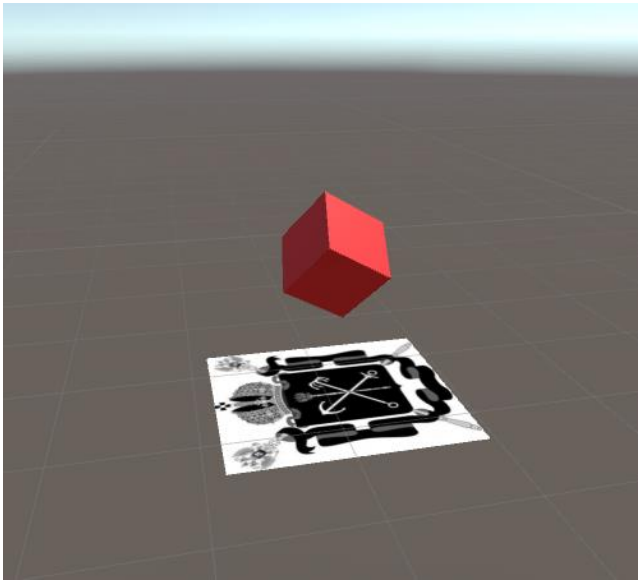
Рисунок 40: Тестовый кубик над маркером.

5.6 Создание прототипа трёхмерной модели внутренней части устройства.

Для демонстрации возможности смотреть сквозь стенки устройства с помощью Hololens потребовалось выбрать объект, который будет служить макетом корпуса устройства. Была выбрана коробка размером 70x22x53 см. Для того, чтобы заполнить макет виртуальным содержимым, была создана трёхмерная модель механизмов, которые могли бы находиться внутри. Для создания модели использовалась программа Autodesk 3DS Max.

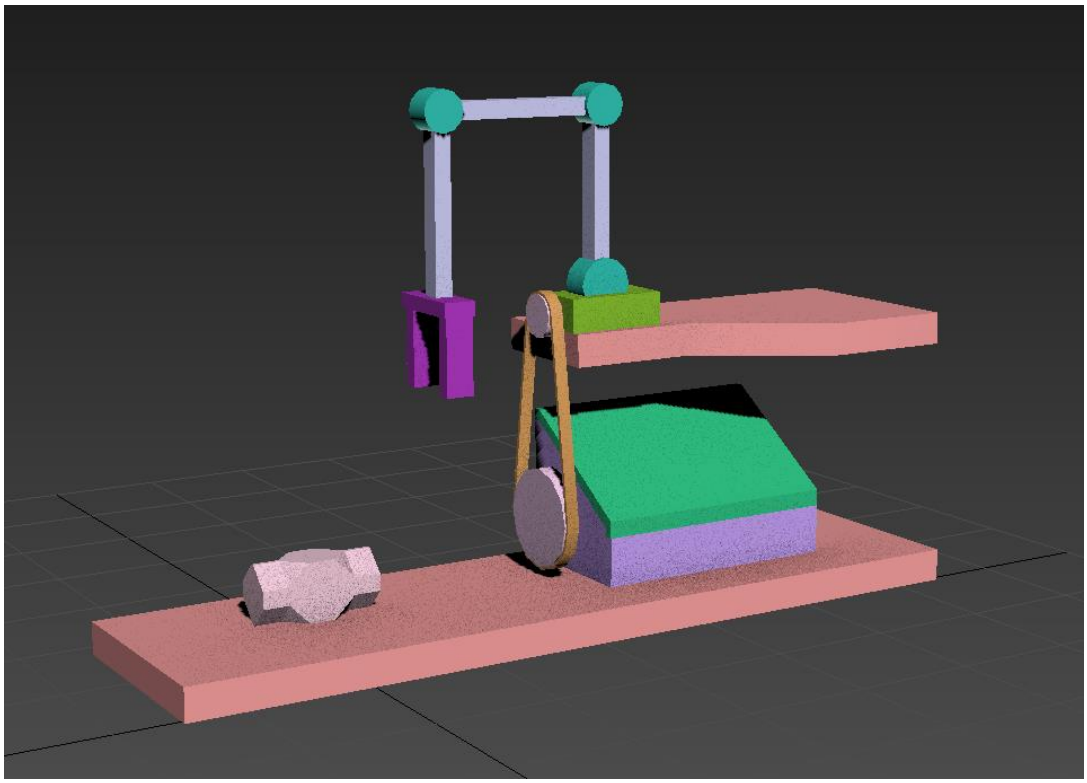

Рисунок 41: Ранний прототип модели без текстур.

Для того, чтобы модель не была статичной, потребовалось добавить несколько анимаций. Было решено реализовать два кейса:

1) Пользователь хочет увидеть текущие события, происходящие внутри макета. Для данного случая была создана анимация движения руки, перемещающей объект.
2) Пользователь хочет заменить сломанную деталь. Для такого случая была создана анимация замены детали, находящейся на нижнем уровне.

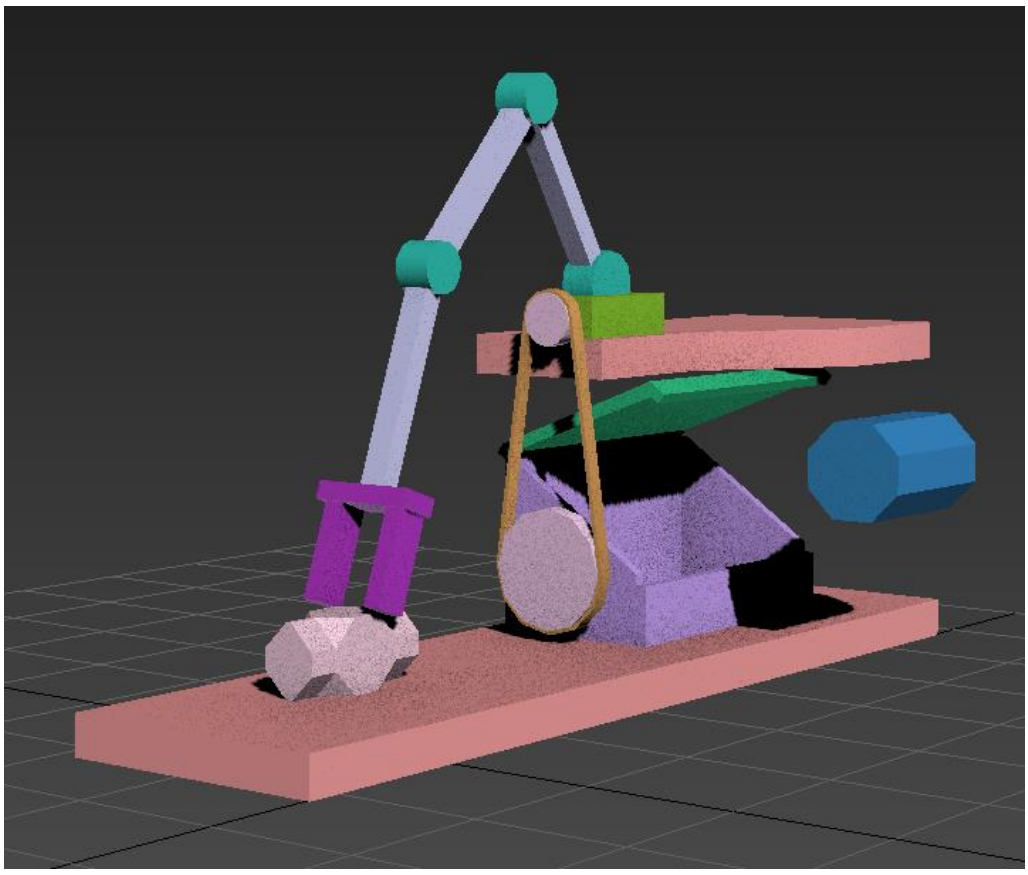

Рисунок 42: Анимации в действии.

Заключение

В результате выполнения научно-исследовательской работы было создано приложение для HTC Vive и десктопа, позволяющее использовать одни и те же игровые предметы при игре с обеих платформ. Таким образом, в созданную игру можно играть и при отсутствии у пользователя шлема виртуальной реальности, что является хорошим заделом для полномасштабной игры, объединяющей пользователей шлемов виртуальной реальности и десктопов.

Игровые предметы позволяют игроку различным способом взаимодействовать с окружением. Например, предметом «рука» игрок может поднимать физические объекты, а различные виды оружия позволяют атаковать неигровых персонажей.

Дополнительным результатом выполнения НИР является создание неигрового персонажа с искусственным интеллектом на основе алгоритмического паттерна “State Machine”.

Также была изучена и определена задача, которая будет решена в процессе создания ВКРМ, а именно компенсация недостатков маркерной дополненной реальности и дополненной реальности на основе пространственного мэппинга за счёт объединения этих двух технологий.