МИНОБРНАУКИ РОССИИ

Федеральное государственное бюджетное образовательное учреждение высшего образования

**«МИРЭА – Российский технологический университет»**

**РТУ МИРЭА**

**Институт кибербезопасности и цифровых технологий**

Кафедра КБ-9 «Предметно-ориентированные информационные системы»

# ОТЧЕТ ПО ПРАКТИЧЕСКОЙ РАБОТЕ №1

по дисциплине: «Алгоритмические основы разработки симуляторов»

на тему: «Расчёт баллистики с учётом сопротивления воздуха»

Выполнил: студент группы БСБО-08-23

Гончаров А.С.

Проверил: ассистент кафедры КБ-9

Малько Е. И.

**Москва, 2025**

## Цели и задачи

Целью данной практической работы является реализация баллистической системы в Unity с аэродинамическим сопротивлением. Создание пушки с физически точным полетом снарядов и визуализацией траектории. Моделирование квадратичного сопротивления воздуха через силу. Интеграция уравнений движения методом Эйлера. Визуализация траектории LineRenderer с обрезкой SphereCast.

Разработка системы снарядов с компонентом QuadraticDrag, применяющим сопротивление в FixedUpdate. Снаряды наследуют параметры массы, радиуса и начальной скорости из системы визуализации. Управление пушкой с относительным перемещением и поворотом по осям.

Создание мишеней со случайными параметрами в заданных диапазонах. Движение мишеней по круговым траекториям. Система спавна с контролем количества и интервалов. Регистрация попаданий через коллизии. Подсчет очков и статистики стрельбы. Синхронизация превью траектории с реальным полетом снарядов.

## Реализация

Разработка начата с создания базовой сцены Unity. Добавлен Plane земли и создана иерархия объектов пушки: CannonRoot (корневой объект), Cannon (основание), Small cannon (визуальная модель дула) и LaunchPoint (точка выстрела). На CannonRoot добавлен LineRenderer для визуализации траектории. Также в качестве визуального оформления на локации были расставлены модели домов (см. рис. 1).

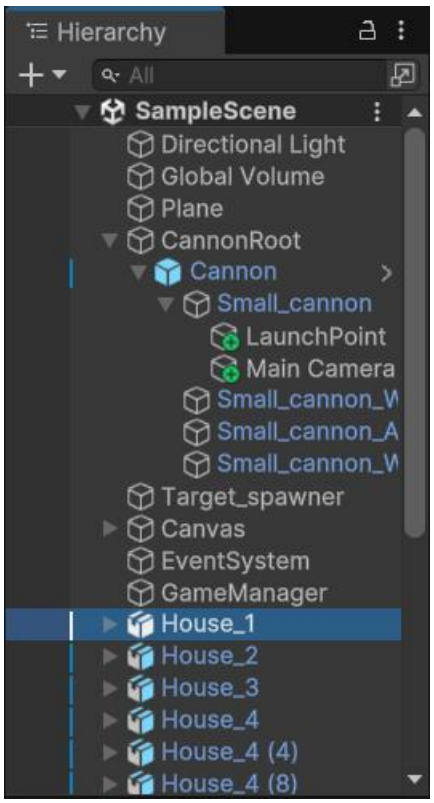


Рис. 1. Иерархия сцены

Реализован скрипт TrajectoryRenderer, рассчитывающий баллистическую траекторию. Для вакуумного случая используется аналитическая формула: position = startPosition + velocity * time + 0.5 * gravity * time². Для моделирования сопротивления воздуха применен численный метод Эйлера с силой Fd = -0.5 * density * dragCoefficient * area * velocity². LineRenderer отображает рассчитанные точки траектории.

TrajectoryRenderer содержит метод Awake (см. рис. 2), который выполняет первичную инициализацию компонента: получает ссылку на LineRenderer, настраивает его параметры включая использование мировых координат и ширину линии, создает материал по умолчанию если он отсутствует, а также рассчитывает площадь поперечного сечения снаряда на основе радиуса.

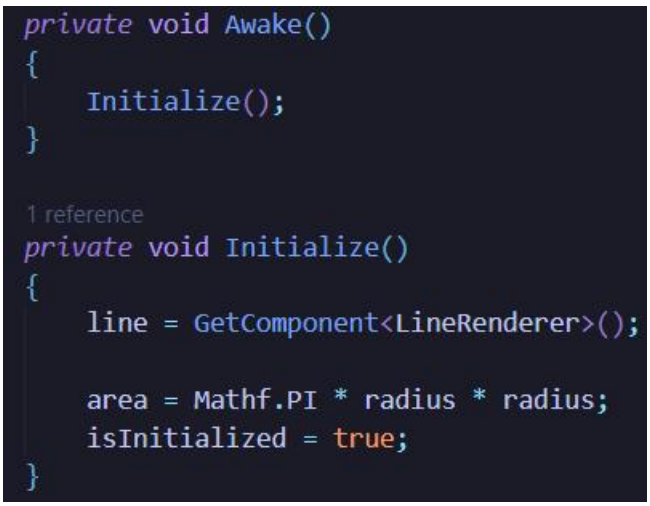



Рис. 2. Методы Awake и Initialize в скрипте TrajectoryRenderer

Метод UpdateArea (см. рис. 3) пересчитывает площадь поперечного сечения при изменении радиуса снаряда, что необходимо для корректного расчета аэродинамического сопротивления.

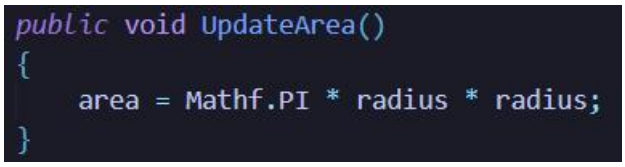



Рис. 3. Метод UpdateArea в скрипте TrajectoryRenderer

Основной метод DrawWithAirEuler (см. рис. 4, 5) реализует численное интегрирование уравнений движения методом Эйлера: инициализирует начальные позицию и скорость, затем для каждой точки траектории вычисляет относительную скорость с учетом ветра, рассчитывает силу сопротивления по формуле квадратичного сопротивления, определяет результирующее ускорение от гравитации и сопротивления воздуха, обновляет скорость и позицию снаряда, а также проверяет столкновения через SphereCast для обрезки траектории.

```
public void DrawWithAirEuler(Vector3 startPosition, Vector3 startVelocity)
{
    if (line == null) return;

    Vector3 p = startPosition;
    Vector3 v = startVelocity;

    Vector3[] points = new Vector3[pointsCount];
    points[0] = p;
    int actualPointsCount = pointsCount;

    for (int i = 1; i < pointsCount; i++)
    {
        Vector3 vRel = v - wind;
        float speed = vRel.magnitude;
        Vector3 drag = speed > 1e-6f ?
            (-0.5f * airDensity * dragCoefficient * area * speed) * vRel :
            Vector3.zero;
```

Рис. 4. Метод DrawWithAirEuler в скрипте TrajectoryRenderer, часть 1

```
        Vector3 a = Physics.gravity + drag / mass;
        v += a * timeStep;
        Vector3 newP = p + v * timeStep;

        if (useSphereCast && CheckCollision(p, newP, radius, out RaycastHit hit))
        {
            points[i] = hit.point;
            actualPointsCount = i + 1;
            break;
        }

        p = newP;
        points[i] = p;
    }

    line.positionCount = actualPointsCount;
    for (int i = 0; i < actualPointsCount; i++)
    {
        line.SetPosition(i, points[i]);
    }
}
```

Рис. 5. Метод DrawWithAirEuler в скрипте TrajectoryRenderer, часть 2

Метод CheckCollision (см. рис. 6) выполняет проверку столкновения между двумя точками траектории с использованием SphereCast, возвращая информацию о столкновении если оно произошло.

```
    private bool CheckCollision(Vector3 from, Vector3 to, float sphereRadius, out RaycastHit hit)
    {
        Vector3 direction = to - from;
        float distance = direction.magnitude;

        if (distance > 0)
        {
            direction.Normalize();

            if (Physics.SphereCast(from, sphereRadius, direction, out hit, distance, collisionMask))
            {
                return true;
            }
        }

        hit = new RaycastHit();
        return false;
    }
}
```

Рис. 6. Метод CheckCollision в скрипте TrajectoryRenderer

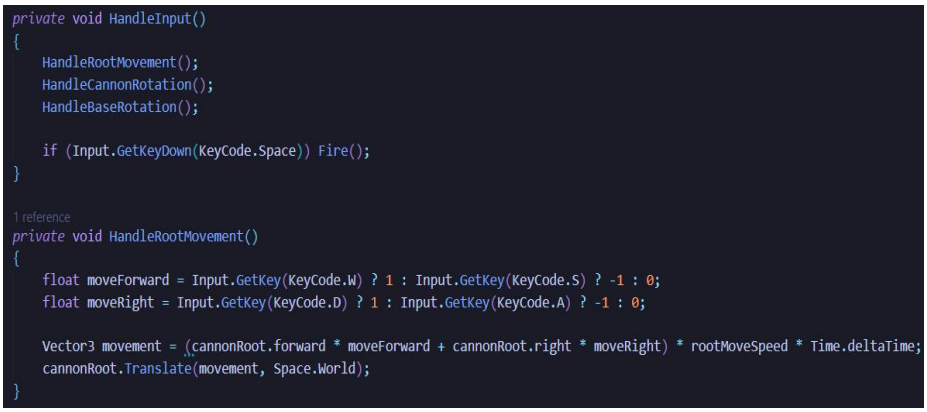
Скрипт CannonController реализует управление артиллерийской установкой и механику стрельбы. В методе Start (см. рис. 7) выполняется первичная инициализация: получается ссылка на компонент TrajectoryRenderer, устанавливается начальный угол запуска 45 градусов, генерируются стартовые параметры снаряда и обновляется визуализация траектории. Метод Update обеспечивает непрерывный опрос ввода и актуализацию отображения траектории (см. рис. 7).

```
private void Start()
{
    trajectoryRenderer = GetComponent<TrajectoryRenderer>();
    currentLaunchAngle = 45f;
    GenerateNewPreviewParams();
    UpdateTrajectory();
}

0 references
private void Update()
{
    HandleInput();

    if (trajectoryRenderer != null)
    {
        UpdateTrajectory();
    }
}
```

Рис. 7. Методы Start и Update в скрипте CannonController

Обработка пользовательского ввода централизована в методе HandleInput (см. рис. 8), который последовательно вызывает методы

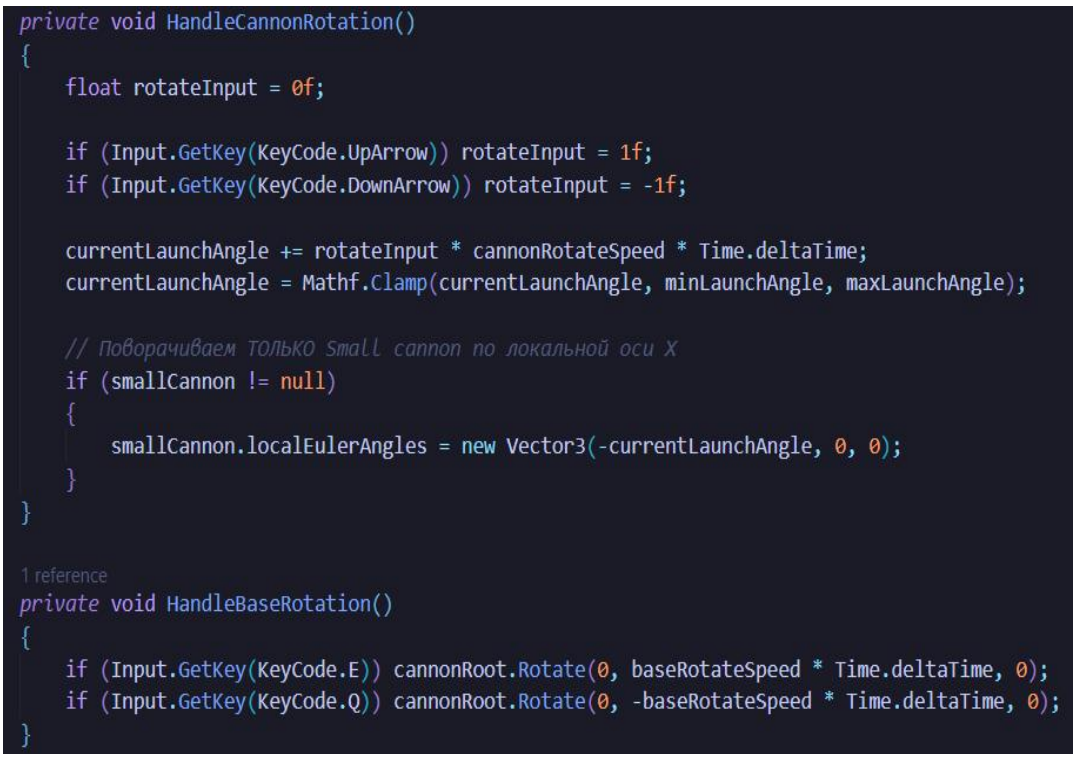
управления перемещением, вращением и осуществляет проверку нажатия клавиши выстрела. HandleRootMovement (см. рис. 8) обрабатывает перемещение всей артиллерийской установки в мировом пространстве относительно её собственной системы координат: клавиши W/S обеспечивают движение вперед/назад по направлению взгляда пушки, A/D - движение влево/вправо относительно текущей ориентации, с применением коррекции на rootMoveSpeed и Time.deltaTime для плавности перемещения.



```
private void HandleInput()
{
    HandleRootMovement();
    HandleCannonRotation();
    HandleBaseRotation();

    if (Input.GetKeyDown(KeyCode.Space)) Fire();
}

1 reference
private void HandleRootMovement()
{
    float moveForward = Input.GetKey(KeyCode.W) ? 1 : Input.GetKey(KeyCode.S) ? -1 : 0;
    float moveRight = Input.GetKey(KeyCode.D) ? 1 : Input.GetKey(KeyCode.A) ? -1 : 0;

    Vector3 movement = (cannonRoot.forward * moveForward + cannonRoot.right * moveRight) * rootMoveSpeed * Time.deltaTime;
    cannonRoot.Translate(movement, Space.World);
}
```

Рис. 8. Методы HandleInput и HandleRootMovement в скрипте CannonController

Метод HandleCannonRotation (см. рис. 9) управляет углом возвышения ствола через клавиши стрелок вверх/вниз: увеличивает или уменьшает currentLaunchAngle с учетом скорости cannonRotateSpeed и ограничивает полученное значение в диапазоне от minLaunchAngle до maxLaunchAngle, после чего применяет вычисленный угол к локальному вращению smallCannon по оси X с отрицательным знаком для корректного визуального представления. HandleBaseRotation (см. рис. 9) обеспечивает поворот всей платформы вокруг вертикальной оси с помощью клавиш Q/E, изменяя вращение cannonRoot с заданной скоростью baseRotateSpeed.



```
private void HandleCannonRotation()
{
    float rotateInput = 0f;

    if (Input.GetKey(KeyCode.UpArrow)) rotateInput = 1f;
    if (Input.GetKey(KeyCode.DownArrow)) rotateInput = -1f;

    currentLaunchAngle += rotateInput * cannonRotateSpeed * Time.deltaTime;
    currentLaunchAngle = Mathf.Clamp(currentLaunchAngle, minLaunchAngle, maxLaunchAngle);

    // Поворачиваем ТОЛЬКО Small cannon по локальной оси X
    if (smallCannon != null)
    {
        smallCannon.localEulerAngles = new Vector3(-currentLaunchAngle, 0, 0);
    }
}

1 reference
private void HandleBaseRotation()
{
    if (Input.GetKey(KeyCode.E)) cannonRoot.Rotate(0, baseRotateSpeed * Time.deltaTime, 0);
    if (Input.GetKey(KeyCode.Q)) cannonRoot.Rotate(0, -baseRotateSpeed * Time.deltaTime, 0);
}
```

Рис. 9. Методы HandleCannonRotation и HandleBaseRotation в скрипте CannonController

Метод GenerateNewPreviewParams (см. рис. 10) генерирует случайные значения массы и радиуса снаряда в установленных пределах minMass/maxMass и minRadius/maxRadius для последующего использования в превью траектории и реальных выстрелах. UpdateTrajectoria (см. рис. 10) обновляет параметры визуализации: передает текущие массу и радиус в Метод TrajectoryRenderer, вычисляет вектор начальной скорости как произведение направления launchPoint.forward на initialSpeed и инициирует перерисовку траектории через DrawWithAirEuler.

```
private void GenerateNewPreviewParams()
{
    currentPreviewMass = Random.Range(minMass, maxMass);
    currentPreviewRadius = Random.Range(minRadius, maxRadius);
}

2 references
private void UpdateTrajectory()
{
    if (trajectoryRenderer == null || launchPoint == null) return;

    trajectoryRenderer.mass = currentPreviewMass;
    trajectoryRenderer.radius = currentPreviewRadius;
    trajectoryRenderer.UpdateArea();

    Vector3 direction = launchPoint.forward;
    currentVelocity = direction * initialSpeed;

    trajectoryRenderer.DrawWithAirEuler(launchPoint.position, currentVelocity);
}
```

Рис. 10. Методы GenerateNewPreviewParams и UpdateTrajectoria в скрипте CannonController

Метод Fire (см. рис. 11) реализует механику выстрела: создает экземпляр снаряда projectilePrefab в позиции и с вращением launchPoint, получает компонент QuadraticDrag и настраивает физические параметры снаряда в соответствии с текущими значениями массы, радиуса и рассчитанной скорости, регистрирует выстрел в GameManager через RegisterShot, выводит в консоль параметры выстрела и инициирует генерацию новых параметров для следующего превью. OnGUI отображает текущие параметры системы в углу экрана: угол наклона ствола, массу и радиус снаряда для информирования игрока.

```
    private void Fire()
{
    if (projectilePrefab == null || trajectoryRenderer == null || launchPoint == null) return;

    GameObject projectile = Instantiate(projectilePrefab, launchPoint.position, launchPoint.rotation);

    QuadraticDrag quadraticDrag = projectile.GetComponent<QuadraticDrag>();
    if (quadraticDrag != null)
    {
        quadraticDrag.SetPhysicalParams(
            currentPreviewMass,
            currentPreviewRadius,
            trajectoryRenderer.dragCoefficient,
            trajectoryRenderer.airDensity,
            trajectoryRenderer.wind,
            currentVelocity
        );
    }

    if (GameManager.Instance != null)
    {
        GameManager.Instance.RegisterShot();
    }

    Debug.Log($"Выстрел: масса={currentPreviewMass:F2}, радиус={currentPreviewRadius:F2}, угол={currentLaunchAngle:F1}°");
    GenerateNewPreviewParams();
}
```

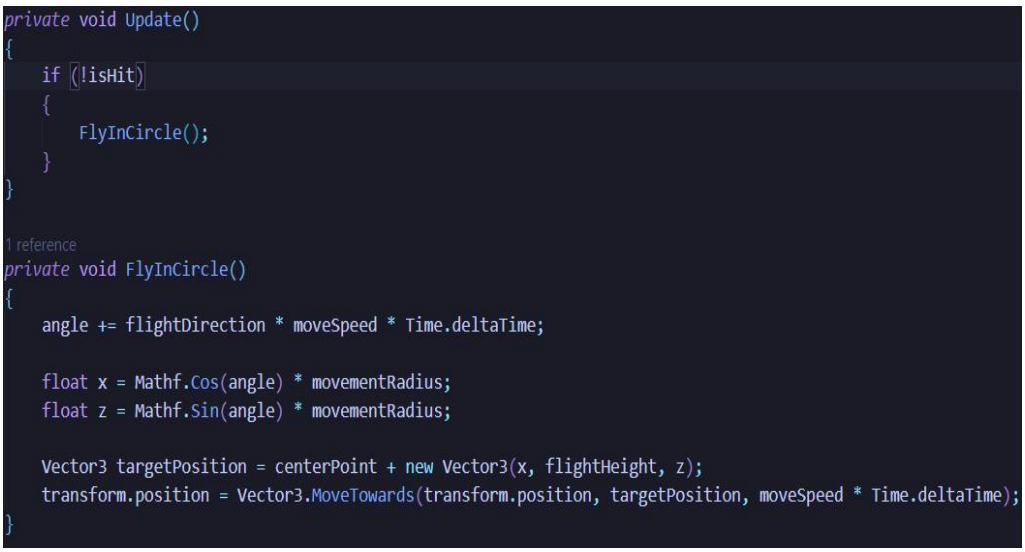
Рис. 11. Метод Fire в скрипте CannonController

Скрипт Target реализует поведение мишени с физическими параметрами и механизмом поражения. В методе Start (см. рис. 12) выполняется первичная инициализация: получается ссылка на компонент Rigidbody, фиксируется начальная позиция как центр окружности движения и вызывается InitializeTarget (см. рис. 12) для настройки параметров. InitializeTarget генерирует случайные значения массы и радиуса в заданных диапазонах, применяет их к физическим свойствам Rigidbody и визуальному масштабу объекта, устанавливает случайные начальный угол и направление движения по окружности, отключает гравитацию и включает кинематический режим для обеспечения плавного перемещения, а также фиксирует высоту полета через модификацию позиции по оси Y.

```
private void Start()
{
    rb = GetComponent<Rigidbody>();
    centerPoint = transform.position;
    InitializeTarget();
}

1 reference
private void InitializeTarget()
{
    currentMass = Random.Range(minMass, maxMass);
    currentRadius = Random.Range(minRadius, maxRadius);

    rb.mass = currentMass;
    transform.localScale = Vector3.one * (currentRadius * 2f);

    angle = Random.Range(0f, 360f);
    flightDirection = Random.value > 0.5f ? 1f : -1f;

    rb.useGravity = false;
    rb.isKinematic = true;

    Vector3 pos = transform.position;
    pos.y = flightHeight;
    transform.position = pos;
}
```

Рис. 12. Методы Start и InitializeTarget в скрипте Target

Метод Update (см. рис. 13) непрерывно проверяет состояние мишени и при отсутствии попадания вызывает FlyInCircle для обеспечения движения. FlyInCircle (см. рис. 13) реализует циклическое перемещение по окружности: увеличивает текущий угол с учетом направления движения и скорости, вычисляет новые координаты X и Z через тригонометрические функции с учетом радиуса движения, формирует целевую позицию с сохранением высоты полета и плавно перемещает мишень к вычисленной точке с использованием MoveTowards для обеспечения равномерного движения.



```
private void Update()
{
    if (!isHit)
    {
        FlyInCircle();
    }
}

1 reference
private void FlyInCircle()
{
    angle += flightDirection * moveSpeed * Time.deltaTime;

    float x = Mathf.Cos(angle) * movementRadius;
    float z = Mathf.Sin(angle) * movementRadius;

    Vector3 targetPosition = centerPoint + new Vector3(x, flightHeight, z);
    transform.position = Vector3.MoveTowards(transform.position, targetPosition, moveSpeed * Time.deltaTime);
}
```

Рис. 13. Методы Update и FlyInCyrcle в скрипте Target

Метод OnCollisionEnter (см. рис. 14, 15) обрабатывает столкновения с другими объектами: проверяет отсутствие предыдущего попадания и соответствие тега столкнувшегося объекта "Projectile", после чего устанавливает флаг isHit и вызывает OnTargetHit для обработки поражения. Метод OnTargetHit (см. рис. 14, 15) активирует физическое поведение при попадании: отключает кинематический режим и включает гравитацию для начала падения, добавляет случайное угловое вращение по всем осям для визуального эффекта, применяет импульсную силу со случайными компонентами для создания реалистичной траектории падения, регистрирует начисление очков через GameManager с передачей параметров мишени и выводит отладочную информацию о попадании.

```
private void OnCollisionEnter(Collision collision)
{
    if (!isHit && collision.gameObject.CompareTag("Projectile"))
    {
        Debug.Log("ПОПАДАНИЕ ЗАРЕГИСТРИРОВАНО!");
        isHit = true;
        OnTargetHit();
    }
}

1 reference
private void OnTargetHit()
{

    rb.isKinematic = false;
    rb.useGravity = true;

    rb.angularVelocity = new Vector3(
        Random.Range(-5f, 5f),
        Random.Range(-5f, 5f),
        Random.Range(-5f, 5f)
    );
```

Рис. 14. Методы On CollisionEnter и OnTargetHit в скрипте Target

```
    rb.AddForce(new Vector3(
        Random.Range(-2f, 2f),
        Random.Range(1f, 3f),
        Random.Range(-2f, 2f)
    ), ForceMode.Impulse);

    if (GameManager.Instance != null)
    {
        GameManager.Instance.AddScore(scoreValue, currentMass, currentRadius);
    }

    Debug.Log($"Попадание! Мишень падает: масса={currentMass:F2}, радиус={currentRadius:F2}");

    StartCoroutine(DestroyAfterFall());
}
```

Рис. 15. Метод OnTargetHit в скрипте Target, продолжение

Метод DestroyAfterFall (см. рис. 16)реализует задержку перед уничтожением объекта через корутину: ожидает 3 секунды после попадания для завершения физической симуляции падения, после чего уничтожает игровой объект. Метод OnDrawGizmosSelected (см. рис. 16) обеспечивает визуализацию в редакторе Unity: отображает сферу желтого цвета, представляющую траекторию движения мишени вокруг зафиксированного центра, что упрощает настройку и отладку параметров движения.

```
    private IEnumerator DestroyAfterFall()
    {
        yield return new WaitForSeconds(3f);
        Destroy(gameObject);
    }

    0 references
    private void OnDrawGizmosSelected()
    {
        Gizmos.color = Color.yellow;
        Gizmos.DrawWireSphere(centerPoint, movementRadius);
    }
}
```

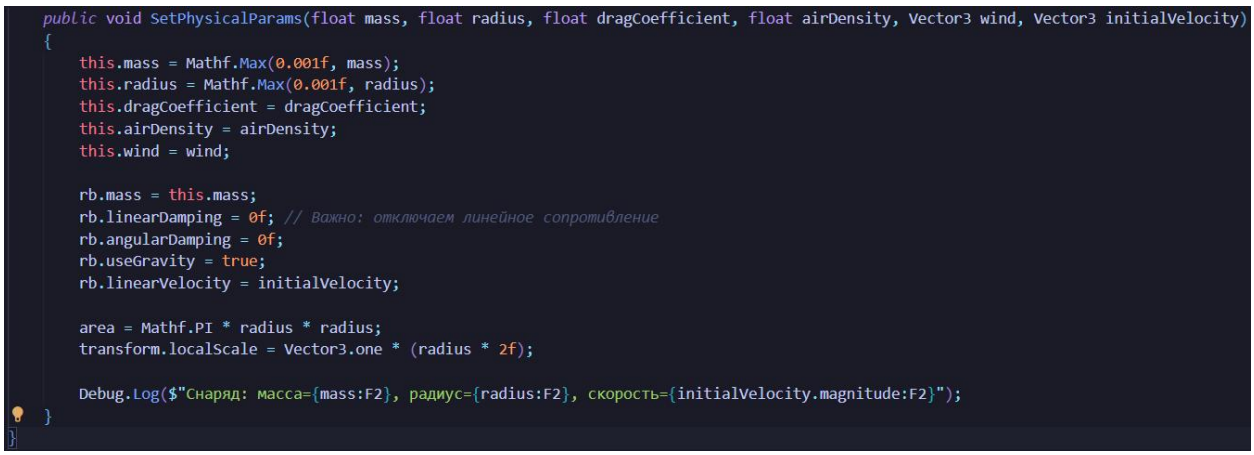
Рис. 16. Методы DestroyAfterFall и OnDrawGizmosSelected в скрипте Target

Скрипт QuadraticDrag реализует физику аэродинамического сопротивления для снарядов на основе квадратичной модели. В методе Awake (см. рис. 17) выполняется получение ссылки на компонент Rigidbody Основные вычисления происходят в FixedUpdate (см. рис. 17), который вызывается на каждом фиксированном временном интервале физического движка: сначала вычисляется относительная скорость снаряда относительно ветра через разность линейной скорости Rigidbody и вектора ветра, определяется модуль полученной скорости, и при его превышении порогового значения 1e-6 вычисляется вектор силы сопротивления по формуле квадратичного трения Fd = -0.5 * ρ * Cd * A * |v| * v, где ρ - плотность воздуха, Cd - коэффициент сопротивления, A - площадь поперечного сечения, |v| - модуль относительной скорости, v - вектор относительной скорости. Рассчитанная сила применяется к Rigidbody через AddForce с режимом Force для непрерывного воздействия.

```
private void Awake()
{
    rb = GetComponent<Rigidbody>();
}

0 references
private void FixedUpdate()
{
    Vector3 vRel = rb.linearVelocity - wind;
    float speed = vRel.magnitude;
    if (speed < 1e-6f) return;

    Vector3 drag = -0.5f * airDensity * dragCoefficient * area * speed * vRel;
    rb.AddForce(drag, ForceMode.Force);
}
```

Рис. 17. Методы Awake и FixedUpdate в скрипте QuadraticDrug

Метод SetPhysicalParams (см. рис. 18) обеспечивает полную настройку физических параметров снаряда: принимает массу, радиус, коэффициент сопротивления, плотность воздуха, вектор ветра и начальную скорость, при этом масса и радиус ограничиваются снизу значением 0.001 для избежания некорректных вычислений. Полученные параметры применяются к компоненту Rigidbody: устанавливается масса, отключается линейное и угловое демпфирование для исключения дублирования эффектов сопротивления, включается гравитация, задается начальная линейная скорость. Дополнительно вычисляется площадь поперечного сечения как $\pi r^2$ для использования в расчетах сопротивления, масштабируется визуальное представление снаряда через transform.localScale в соответствии с физическим радиусом, и выводятся отладочные о параметрах созданного снаряда. Данная реализация обеспечивает физически достоверное поведение снарядов с учетом аэродинамических эффектов при различных комбинациях параметров.



```
public void SetPhysicalParams(float mass, float radius, float dragCoefficient, float airDensity, Vector3 wind, Vector3 initialVelocity)
{
    this.mass = Mathf.Max(0.001f, mass);
    this.radius = Mathf.Max(0.001f, radius);
    this.dragCoefficient = dragCoefficient;
    this.airDensity = airDensity;
    this.wind = wind;

    rb.mass = this.mass;
    rb.linearDamping = 0f; // Важно: отключаем линейное сопротивление
    rb.angularDamping = 0f;
    rb.useGravity = true;
    rb.linearVelocity = initialVelocity;

    area = Mathf.PI * radius * radius;
    transform.localScale = Vector3.one * (radius * 2f);

    Debug.Log($"Снаряд: масса={mass:F2}, радиус={radius:F2}, скорость={initialVelocity.magnitude:F2}");
}
}
```

Рис. 18. Метод SetPhysicalParams в скрипте QuadraticDrug

Скрипт TargetSpawner реализует систему создания и управления мишенями на сцене. В методе Update (см. рис. 19) осуществляется основной цикл спавна: уменьшается значение таймера spawnTimer на время, прошедшее с последнего кадра, и при достижении таймером нулевого значения совместно с проверкой условия на максимальное количество мишеней через CountTargets происходит вызов SpawnTarget и сброс таймера на значение spawnInterval. Метод SpawnTarget (см. рис. 19) создает экземпляр мишени в случайной позиции в пределах заданной области: координаты X и Z генерируются случайным образом в диапазоне от -spawnArea.x до spawnArea.x и от -spawnArea.z до spawnArea.z соответственно, а координата Y фиксируется на уровне flightHeight для обеспечения полета на постоянной высоте. После создания экземпляра targetPrefab через Instantiate получается компонент Target и производится настройка его параметров: устанавливаются диапазоны массы и радиуса из соответствующих полей spawner'a, а также высота полета для обеспечения согласованности поведения.

```
private void Update()
{
    spawnTimer -= Time.deltaTime;

    if (spawnTimer <= 0 && CountTargets() < maxTargets)
    {
        SpawnTarget();
        spawnTimer = spawnInterval;
    }
}

1 reference
private void SpawnTarget()
{
    if (targetPrefab == null) return;

    Vector3 spawnPosition = new Vector3(
        Random.Range(-spawnArea.x, spawnArea.x),
        flightHeight,
        Random.Range(-spawnArea.z, spawnArea.z)
    );

    GameObject target = Instantiate(targetPrefab, spawnPosition, Quaternion.identity);
    Target targetScript = target.GetComponent<Target>();

    if (targetScript != null)
    {
        targetScript.minMass = targetMinMass;
        targetScript.maxMass = targetMaxMass;
        targetScript.minRadius = targetMinRadius;
        targetScript.maxRadius = targetMaxRadius;
        targetScript.flightHeight = flightHeight;
```

Рис. 19. Методы Update и SpawnTarget в скрипте TargetSpawner

Метод (см. рис. 20) CountTargets определяет текущее количество активных мишеней на сцене через поиск всех объектов с компонентом Target и возвращает их количество, что обеспечивает контроль за соблюдением лимита maxTargets. Метод OnDrawGizmosSelected (см. рис. 20) визуализирует зону спавна в редакторе Unity: отображает зеленый проволочный куб с центром, смещенным на flightHeight по вертикали относительно позиции spawner'a, и размерами, соответствующими удвоенным значениям spawnArea по осям X и Z при минимальной толщине по оси Y, что предоставляет наглядное представление о пространственных границах появления мишеней во время редактирования сцены. Данная система гарантирует постоянное поддержание заданного количества

мишеней с регулируемой частотой появления и контролируемым распределением в пространстве.

```
private int CountTargets()
{
    return GameObject.FindObjectsOfType<Target>().Length;
}

0 references
private void OnDrawGizmosSelected()
{
    Gizmos.color = Color.green;
    Gizmos.DrawWireCube(transform.position + Vector3.up * flightHeight, new Vector3(spawnArea.x * 2, 0.1f, spawnArea.z * 2));
}
```

Рис. 20. Методы CountTargets и OnDrawGizmosSelected в скрипте TargetSpawner

Скрипт GameManager реализует систему управления игровой статистикой и отображения результатов. Класс использует паттерн Singleton через статическое свойство Instance, что обеспечивает глобальный доступ к менеджеру из любых компонентов игры. В методе Awake (см. рис. 21) проверяется существование экземпляра GameManager - если Instance равен null, текущий объект назначается единственным экземпляром, в противном случае объект уничтожается для предотвращения дублирования.

Метод Update (см. рис. 21) непрерывно обновляет пользовательский интерфейс через вызов UpdateUI. Метод AddScore (см. рис. 21) обрабатывает начисление очков за попадание: увеличивает общий счет на указанное количество points, инкрементирует счетчик пораженных мишеней targetsHit и выводит в консоль информацию о начисленных очках и параметрах пораженной мишени. RegisterShot фиксирует каждый произведенный выстрел через увеличение счетчика totalShots.

```
private void Awake()
{
    if (Instance == null)
    {
        Instance = this;
    }
    else
    {
        Destroy(gameObject);
    }
}

0 references
private void Update()
{
    UpdateUI();
}

1 reference
public void AddScore(int points, float mass, float radius)
{
    score += points;
    targetsHit++;

    Debug.Log($"+{points} очков! Мишень: масса={mass:F2}, радиус={radius:F2}");
}
```

Рис. 21. Методы Awake, Update и AddScore в скрипте GameManager

Метод RegisterShot (см. рис. 22) фиксирует каждый произведенный выстрел через увеличение счетчика totalShots, обеспечивая учет общей активности стрельбы.

Метод UpdateUI (см. рис. 22) обновляет текстовые элементы интерфейса с проверкой наличия ссылок на компоненты Text: в scoreText отображается текущее количество накопленных очков, в statsText выводится упрощенная статистика попаданий в формате абсолютного значения пораженных мишеней. Система обеспечивает минималистичный но эффективный учет ключевых игровых показателей с фокусом на отображение основных метрик производительности игрока.

```
public void RegisterShot()
{
    totalShots++;
}

1 reference
private void UpdateUI()
{
    if (scoreText != null)
        scoreText.text = $"Очки: {score}";

    if (statsText != null)
        statsText.text = $"Попадания: {targetsHit}";
```

Рис. 22. Методы Register в скрипте GameManager

Описанные скрипты реализуют поставленные задачи по реализации артелирийской системы (см. рис. 23).

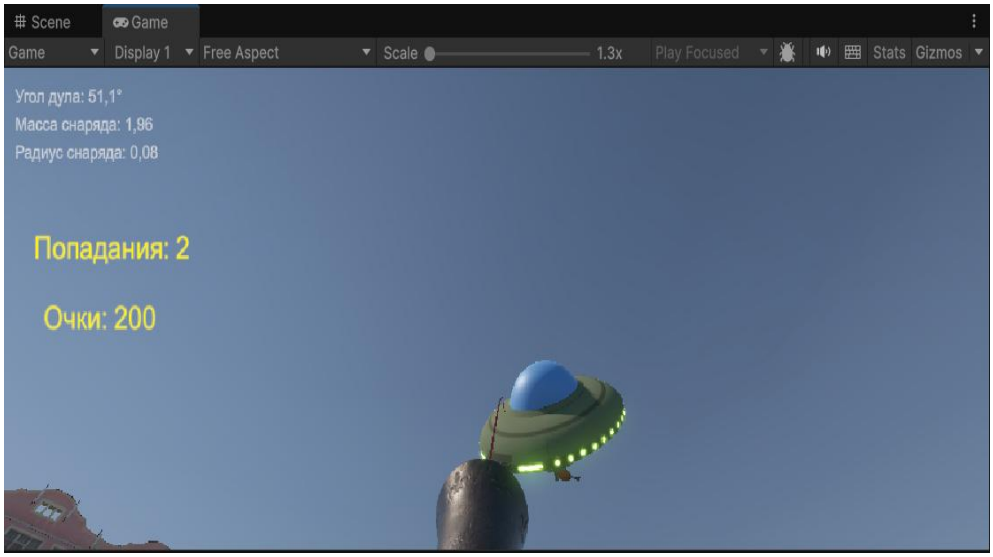



Рис. 23. Рабочая система баллистики

Увеличение массы снаряда положительно сказывается на дальности полета - тяжелые снаряды лучше сохраняют кинетическую энергию, менее подвержены торможению воздухом и демонстрируют более стабильную траекторию. Однако тяжелые снаряды требуют большего времени для разгона и имеют более крутую начальную траекторию, что осложняет прицеливание на коротких дистанциях.

Рост радиуса снаряда приводит к значительному увеличению аэродинамического сопротивления из-за квадратичной зависимости силы сопротивления от площади поперечного сечения. Крупнокалиберные снаряды быстро теряют скорость, особенно на начальном участке траектории,

что сокращает максимальную дальность стрельбы. При этом крупные снаряды обладают лучшей заметностью и упрощают визуальное прицеливание, но требуют более точного расчета упреждения из-за сильного баллистического снижения.

Оптимальное соотношение масса-радиус достигается при максимальной массе при минимальном радиусе, что обеспечивает высокую плотность и баллистический коэффициент. На практике снаряды среднего калибра с повышенной плотностью демонстрируют наилучшие показатели точности и дальности, сохраняя устойчивость к воздушным потокам без чрезмерного падения скорости. Мелкие легкие снаряды подвержены сильному сносу ветром, крупные тяжелые - требуют точных начальных расчетов из-за выраженного баллистического искривления.

**Вывод**

Реализована комплексная баллистическая система в Unity, корректно моделирующая физику полёта снарядов с учётом квадратичного сопротивления воздуха. Достигнута полная синхронизация между визуализацией траектории через LineRenderer и реальным движением снарядов с физическими параметрами. Создана интерактивная учебная среда с системой случайных мишеней и подсчётом статистики стрельбы.

Проведённый анализ подтвердил теоретические зависимости: увеличение массы снаряда повышает дальность полёта за счёт сохранения кинетической энергии, а рост радиуса сокращает дистанцию из-за возрастания аэродинамического сопротивления. Оптимальная точность достигается при балансе параметров - снаряды средней массы и малого радиуса демонстрируют наилучшие баллистические характеристики.