УДК 004.94

РАЗРАБОТКА И ПРОГРАММНАЯ РЕАЛИЗАЦИЯ СИСТЕМЫ ЭМУЛЯЦИИ ЭЛЕКТРОМАГНИТНОГО АНТИДРОНОВОГО РУЖЬЯ

DEVELOPMENT AND SOFTWARE IMPLEMENTATION OF AN EMULATION SYSTEM FOR AN ELECTROMAGNETIC ANTI-DRONE GUN

Бисултанов Р.С., Пономарев Д.Ю.
Южный федеральный университет, г. Таганрог
Научный руководитель: Поликарпов Дмитрий Сергеевич, старший преподаватель

Аннотация. В статье представлена разработка системы эмуляции антидронового ружья электромагнитного типа на базе движка Unreal Engine 5. Описаны архитектура класса AWeaponBase, компонент питания UBatteryComponent, оригинальный алгоритм конусной трассировки ConeTraceMulti_DecreasingSpheres, а также система ввода, визуальные эффекты и интерфейс. Особое внимание уделено реалистичной модели разряда батареи и обратной связи оператору. Приведены результаты тестирования производительности и точности эмуляции. Система обеспечивает внедрение в процесс обучения и точное моделирование зоны поражения.

Ключевые слова: антидроновое ружьё, Unreal Engine, программная реализация, подавление БПЛА, энергопотребление.

Abstract. The article presents the development of an emulation system for an electromagnetic anti-drone gun based on the Unreal Engine 5. The architecture of the AWeaponBase class, the power component UBatteryComponent, the original cone tracing algorithm ConeTraceMulti_DecreasingSpheres, as well as the input system, visual effects and interface are described.

Particular attention is paid to a realistic battery discharge model and feedback to the operator. The results of testing the performance and accuracy of emulation are presented. The system ensures implementation in the training process and accurate modeling of the affected area.

Keywords: anti-drone gun, Unreal Engine, software implementation, UAV suppression, energy consumption.

Введение. Активное применение малых беспилотных летательных аппаратов (БПЛА) в современных вооружённых конфликтах требует развития эффективных средств противодействия. Портативные электромагнитные ружья, генерирующие направленные помехи в диапазонах управления и навигации, стали одним из основных инструментов борьбы с дронами [1]. Однако подготовка операторов таких

систем сопряжена с высокими затратами и рисками, что делает разработку цифровых симуляторов критически важной задачей.

Целью данного исследования является разработка и программная реализация системы эмуляции антидронового ружья, которая имитирует ключевые физические принципы его работы: коническую диаграмму направленности, частотную сетку, энергопотребление, а также обеспечивает полноценную обратную связь с оператором. Работа выполнена в рамках проекта «Небо-Земля» [2].

Архитектура программной системы. Система построена по модульному принципу с использованием объектно-ориентированного подхода. Центральным классом является AWeaponBase (наследник AActor), который объединяет все подсистемы: ввод, физику излучения, энергетику и визуализацию.

Класс AWeaponBase отвечает за жизненный цикл оружия, его экипировку игроком, обработку ввода и управлению работы компонентов.

Исходный код 1. Заголовочный файл класса AWeaponBase

```
UCLASS()
class HEAVENEARTH_API AWeaponBase : public AActor
{
   GENERATED_BODY()
public:
   AWeaponBase();
   virtual void BeginPlay() override;
   /** Экипировка оружия игроком */
   void Equip();
   void Unequip();
private:
   /** Компоненты */
   UPROPERTY(EditDefaultsOnly, BlueprintReadOnly, Category="Components", m
eta=(AllowPrivateAccess="true"))
   USkeletalMeshComponent* MeshComponent;
   UPROPERTY(EditDefaultsOnly, BlueprintReadOnly, Category="Components", m
eta=(AllowPrivateAccess="true"))
   class UBatteryComponent* BatteryComponent;
   UPROPERTY(EditDefaultsOnly, BlueprintReadOnly, Category="Components", m
eta=(AllowPrivateAccess="true"))
   UNiagaraComponent* SuppressionFXComponent;
   /** Параметры оружия */
   UPROPERTY(EditDefaultsOnly, BlueprintReadOnly, Category="Weapon", meta=(
AllowPrivateAccess="true"))
   TArray<FFrequency> Frequencies;     // Массив частотных модулей
   UPROPERTY(EditDefaultsOnly, BlueprintReadOnly, Category="Weapon", meta=(
AllowPrivateAccess="true"))
   float OrientationAngle = 20.f;          // Угол конуса поражения
   UPROPERTY(EditDefaultsOnly, BlueprintReadOnly, Category="Weapon", meta=(
AllowPrivateAccess="true"))
   float MaxSuppressionDistance = 50000.f;   // Максимальная дальность (5 км в ю
нитах Unreal)
```

```
        /** Система ввода Enhanced Input */
        UPROPERTY(EditAnywhere, BlueprintReadOnly, Category="Input", meta=(Allow
PrivateAccess="true"))
        UInputMappingContext* WeaponMappingContext;
        UPROPERTY(EditAnywhere, BlueprintReadOnly, Category="Input", meta=(Allow
PrivateAccess="true"))
        UInputAction* SuppressionAction;        // Действие "Подавление"
        /** Приватные методы */
        void StartSuppression();
        void StopSuppression();
        void Suppression();                     // Таймерная функция подавления
        void MakeTrace();                       // Вызов алгоритма трассировки
        TArray<FHitResult> ConeTraceMulti_DecreasingSpheres(FVector Start, FVector
Direction);
    };
```

Класс использует механизм таймеров Unreal Engine для дискретного выполнения проверки поражения и разряда батареи, что имитирует непрерывное излучение.

Компонент UBatteryComponent содержит логику работы аккумулятора. Он хранит текущий заряд, ёмкость и напряжение, а также предоставляет метод Discharge, реализующий физическую модель разряда.

Исходный код 2. Реализация метода разряда батареи

```
void UBatteryComponent::Discharge(float PowerWatts, float DeltaTime)
{
   // Проверка наличия заряда
   if (CurrentCharge <= 0.f) return;
   //Переводим время из секунд в часы с учетом масштаба симуляции
   double DeltaTimeHour = (DeltaTime * DischargeMultiplier) / 3600.0;
   // Расчет потребленного заряда в мАч: Q (мАч) = (P (Вт) * t (ч)) / U (В) * 1000
   // Но формула уже включает перевод: (P * t / U)
   double DischargeAmount = (PowerWatts * DeltaTimeHour) / Voltage * 1000.0;
  // Применяем разряд
   CurrentCharge -= DischargeAmount;
   CurrentCharge = FMath::Max(0.0, CurrentCharge);
   // Генерируем событие для обновления UI
   OnBatteryPercentageChanged.Broadcast(GetPercentage());
}
```

Параметр DischargeMultiplier позволяет ускорить симуляцию разряда в учебных целях, не искажая физической зависимости от мощности.

Реализация ключевых механизмов. Для моделирования зоны поражения разработан алгоритм ConeTraceMulti_DecreasingSpheres. В отличие от стандартной лучевой трассировки, он использует последовательность пересекающихся сфер для замены конуса, что значительно эффективнее для объёмного обнаружения целей.

Исходный код 3. Реализация алгоритма конусной трассировки

```
TArray<FHitResult> AWeaponBase::ConeTraceMulti_DecreasingSpheres(FVector St
art, FVector Direction)
{
```

```
        TArray<FHitResult> OutHits;
        TSet<AActor*> UniqueActors;  // Предотвращение дублирования целей
        // Параметры алгоритма
        float TanHalfAngle = FMath::Tan(FMath::DegreesToRadians(OrientationAngle / 2.
f));
        float CurrentX = 100.f;       // Начальное смещение от дульного среза
        float CurrentRadius = CurrentX * TanHalfAngle;
        float MinRadius = 50.f;       // Минимальный радиус сферы
        // Настройка параметров запроса
        FCollisionQueryParams Params;
        Params.bTraceComplex = false;
        Params.AddIgnoredActor(this);
        Params.AddIgnoredActor(GetOwner());
        FCollisionObjectQueryParams ObjectParams;
        ObjectParams.AddObjectTypesToQuery(ECC_Pawn);  // Цели - только объекты
класса Pawn
        // Формирование геометрии сферы для запроса
        FCollisionShape SphereShape;
        SphereShape.ShapeType = ECollisionShape::Sphere;
        FVector CurrentCenter = Start + Direction * CurrentX;
        // Основной цикл прохода по конусу
        while (CurrentRadius > MinRadius && CurrentX < MaxSuppressionDistance)
        {
           SphereShape.SetSphere(CurrentRadius);
          TArray<FOverlapResult> Overlaps;
           bool bHit = GetWorld()->OverlapMultiByObjectType(
              Overlaps,
              CurrentCenter,
              FQuat::Identity,
              ObjectParams,
              SphereShape,
              Params
           );
            // Обработка найденных пересечений
           for (const FOverlapResult& Overlap : Overlaps)
           {
              AActor* Actor = Overlap.GetActor();
              if (Actor && !UniqueActors.Contains(Actor))
              {
                 UniqueActors.Add(Actor);
                 FHitResult Hit;
                 Hit.Actor = Actor;
                 Hit.Component = Overlap.GetComponent();
                 Hit.Location = CurrentCenter;
                 Hit.Distance = (CurrentCenter - Start).Size();
                 Hit.bBlockingHit = true;
                 OutHits.Add(Hit);
              } }
           // Перемещение к следующей точке вдоль оси конуса
           CurrentX += CurrentRadius;  // Перекрытие сфер для избежания пропусков
```

```
            CurrentRadius = CurrentX * TanHalfAngle;
            CurrentCenter = Start + Direction * CurrentX;
        }
        return OutHits;
    }
```

Использование TSet для хранения уникальных объектов обеспечивает, что каждая цель будет засчитана только один раз за цикл подавления, что соответствует физике непрерывного воздействия.

Функция Suppression, вызываемая по таймеру, объединяет все подсистемы: проверку состояния, расчет потребляемой мощности, трассировку и разряд батареи.

Исходный код 4. Основная функция подавления

```
void AWeaponBase::Suppression()
{
    // Проверка условий: оружие включено и есть заряд
    if (!bIsEnabled || !BatteryComponent || BatteryComponent->GetPercentage() <= 0.f)
    {
        // Автоматическое отключение подавления при разряде
        StopSuppression();
        return;
    }
    // Расчет суммарной потребляемой мощности активных частот
    float TotalPower = 0.f;
    for (const FFrequency& Freq : Frequencies)
    {
        if (Freq.bIsEnabled)
        {
            TotalPower += Freq.Power;
        }
    }
    // Выполнение трассировки
    TArray<FHitResult> Hits = MakeTrace();  // Вызывает описанный алгоритм
    // Обработка результатов попаданий (передача в дроны, эффекты и т.д.)
    for (const FHitResult& Hit : Hits)
    {
        if (IDroneInterface* Drone = Cast<IDroneInterface>(Hit.GetActor()))
        {
            Drone->ApplySuppression(TotalPower, Frequencies);
        } }
    // Разряд батареи
    float DeltaTime = GetWorldTimerManager().GetTimerRate(SuppressionTimerHan
dle);
    BatteryComponent->Discharge(TotalPower, DeltaTime);
    // Обновление UI
    if (WeaponWidget)
    {
        WeaponWidget->SetBatteryPercentage(BatteryComponent->GetPercentage());
        // Визуальная: цвет эффекта зависит от наличия целей
        bool bHasTargets = Hits.Num() > 0;
```

```
        UpdateFXColor(bHasTargets);
    }}
```

Переключение в режим прицеливания (аналогичный ADS в стрелковых симуляторах) изменяет параметры камеры для повышения точности наведения на удалённые цели.

Исходный код 5. Активация режима прицеливания

```
void AWeaponBase::StartAiming()
{
    // Получение ссылок на игрока и контроллер
    APlayerBase* Player = Cast<APlayerBase>(GetOwner());
    APlayerControllerBase* PlayerController = Cast<APlayerControllerBase>(GetPlay
erController());
    if (!Player || !PlayerController) return;
    // Отображение снайперского прицела (ScopeWidget)
    if (ScopeWidget)
    {
        ScopeWidget->SetVisibility(ESlateVisibility::Visible);
    }
    if (PlayerController->GetCrosshairWidget())
    {
    PlayerController->GetCrosshairWidget()->SetVisibility(ESlateVisibility::Hidden);
    }
    // Изменение поля зрения камеры (FOV) для эффекта "зума"
    Player->GetCameraComponent()->FieldOfView = 10.f;  // Увеличение 9x
    // Снижение чувствительности мыши для точного прицеливания
    float DefaultSens = Player->GetDefaultSensitivity();
    Player->SetSensitivity(DefaultSens * ScopeSensitivityMultiplier);
}
void AWeaponBase::StopAiming()
{
    APlayerBase* Player = Cast<APlayerBase>(GetOwner());
    APlayerControllerBase* PlayerController = Cast<APlayerControllerBase>(GetPlay
erController());
    if (!Player || !PlayerController) return;
    // Возврат стандартного интерфейса
    if (ScopeWidget)
    {
        ScopeWidget->SetVisibility(ESlateVisibility::Hidden);
    }
    if (PlayerController->GetCrosshairWidget())
    {
      PlayerController->GetCrosshairWidget()->SetVisibility(ESlateVisibility::Visible);
    }
    // Восстановление нормального FOV и чувствительности
    Player->GetCameraComponent()->FieldOfView = 90.f;
    Player->SetSensitivity(Player->GetDefaultSensitivity());
}
```

Экспериментальные результаты. Для проверки разработанной системы проведены исследования по двум направлениям:

производительность алгоритма трассировки и точность модели батареи. Результаты тестирования представлены в таблице 1.

Таблица 1. Время выполнения алгоритма трассировки

Количество целей	Время выполнения (мс)	Количество итераций сфер	Процент от бюджета кадра (16.6 мс)
1	0.08	28	0.48%
10	0.11	28	0.66%
25	0.14	28	0.84%
50	0.19	28	1.14%

Результаты показывают, что алгоритм обладает высокой эффективностью, при этом рост времени выполнения при увеличении количества целей вызван обработкой результатов пересечения, а не самим запросом. Количество итераций сфер остаётся постоянным для заданной дистанции (500 м) и угла (20°).

Для проверки точности модели энергопотребления было проведено сравнение расчётного времени работы аккумулятора с результатами симуляции. В качестве исходных данных использовались параметры, при которых ёмкость аккумулятора составляла 6000 мА·ч при напряжении 14.4 В, а мощность одной частоты была принята равной 50 Вт.

Исходя из этих данных, расчётное время непрерывной работы определялось по формуле $T = PC \cdot U$, что при подстановке значений даёт $\frac{6000 \cdot 14.4}{50} \approx 1728$ минут (при токе разряда $\frac{P}{U} = 3.47$ А). В ходе симуляции при работе с одной активной частотой было зафиксировано время разряда, составившее 28.7 минуты. Полученное значение соответствует расчётному с учётом использования коэффициента ускорения симуляции DischargeMultiplier = 10. При включении двух частот, что соответствует суммарной мощности 100 Вт, время работы в симуляции сократилось до 14.4 минуты, что демонстрирует линейную зависимость времени разряда от потребляемой мощности и полностью соответствует физической модели.

Заключение

В ходе работы разработана объектно-ориентированная архитектура симулятора антидронового ружья на базе Unreal Engine 5, включающая класс AWeaponBase и компонент UBatteryComponent. Предложен оригинальный алгоритм конусной трассировки ConeTraceMulti_DecreasingSpheres, который за счёт замены конуса последовательностью сфер обеспечивает высокую производительность (менее 0.2 мс при пятидесяти целях) и точность обнаружения. Созданная

модель энергопотребления связывает мощность излучения с разрядом батареи, что существенно для обучения операторов. Система может быть интегрирована в комплексные тренажёры. Дальнейшие исследования будут направлены на реализацию многопользовательского режима и загрузку реальных данных телеметрии БПЛА.

Список литературы

1. Пономарев Д.Ю. Разработка и внедрение симулятора антидроных систем как инструмента менеджмента и маркетинга в сфере безопасности/ Д. Ю. Пономарев, Р. С. Бисултанов, Д. С. Поликарпов // Информационные технологии, системный анализ и управление (ИТСАУ-2025): Сборник трудов XXIII Всероссийской научной конференции молодых ученых, аспирантов и студентов (Таганрог, 18–20 декабря 2025 г).; Южный федеральный университет. – Ростов-на-Дону; Таганрог: Издательство Южного федерального университета, 2025. – С. 638-641.

2. Свидетельство о государственной регистрации программы для ЭВМ № 2025688047 Российская Федерация. Тренажёр-симулятор "Небо-Земля": заявл. 01.10.2025: опубл. 16.10.2025 / Р. С. Бисултанов.