МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ
РОССИЙСКОЙ ФЕДЕРАЦИИ
Федеральное государственное автономное
образовательное учреждение высшего
образования
«СЕВЕРО-КАВКАЗСКИЙ ФЕДЕРАЛЬНЫЙ УНИВЕРСИТЕТ»

МЕТОДИЧЕСКИЕ УКАЗАНИЯ ПО ВЫПОЛНЕНИЮ ПРАКТИЧЕСКИХ РАБОТ ПО ДИСЦИПЛИНЕ «Разработка игровых приложений»

для студентов направления подготовки 09.03.04 Программная инженерия
«Разработка и сопровождение программного обеспечения»

Ставрополь, 2026

Содержание

Введение

Современная игровая индустрия требует от разработчиков не только творческого подхода, но и глубоких инженерных знаний: архитектуры игровых приложений, паттернов проектирования, физического моделирования, искусственного интеллекта, оптимизации производительности и работы с внешними сервисами. Дисциплина «Разработка игровых приложений» ориентирована на формирование практических навыков создания игр с использованием открытых и доступных инструментов — прежде всего Godot Engine (лицензия MIT), а также Unity Personal для образовательных целей.

Лабораторный практикум состоит из 7 работ, построенных по принципу нарастающей сложности: от знакомства с движком и компонентной моделью до реализации законченного игрового проекта с публикацией билда и технической документации. Каждая работа включает теоретическое обоснование, пример выполнения и вариативные задания трёх уровней (базовый, средний, продвинутый), что позволяет адаптировать курс под разный уровень подготовки студентов. Итоговая аттестация — защита комплексного проекта (ЛР №7) с демонстрацией работающей игры и презентации.

Лабораторная работа №1. Знакомство с игровым движком. Game Loop и компоненты

1. Цель работы

Установить Godot Engine (или Unity), создать минимальный проект, реализовать игровой цикл и компонентную структуру.

2. Формируемые компетенции и индикаторы

- **ПК-3** (разработка): базовое умение создавать сцены и скрипты.
- **ПК-9** (документирование): оформление отчёта о структуре проекта.

3. Теоретическое обоснование

3.1. Игровой движок и архитектура

- Игровой движок предоставляет готовые подсистемы: рендеринг, физику, ввод, аудио.
- **Godot Engine** использует концепцию **сцены** (Scene) и **узлов** (Node). Сцена — это дерево узлов. Каждый узел выполняет определённую функцию (спрайт, коллизия, скрипт).
- **Game Loop** — основной цикл игры, который обрабатывает ввод, обновляет состояние и рендерит кадр. В Godot основными методами являются _ready() (вызывается при входе в сцену) и _process(delta) (вызывается каждый кадр с аргументом delta — время между кадрами).

3.2. Компонентный подход

- Вместо глубокого наследования игровые объекты строятся из компонентов (узлов). Например, персонаж может иметь узлы Sprite2D, CollisionShape2D, MovementComponent (пользовательский скрипт).
- Скрипты пишутся на GDScript (язык, похожий на Python) или C#.

4. Задачи работы

1. Установить Godot Engine (через официальный сайт или Steam для образования).
2. Создать новый проект, настроить главную сцену (корневой узел Node2D).
3. Добавить дочерние узлы: Sprite2D (текстура персонажа) и CollisionShape2D (прямоугольник коллизии).
4. Написать скрипт для управления персонажем с помощью клавиш-стрелок с использованием _process(delta).
5. Реализовать простейшую систему компонентов: отдельные скрипты MovementComponent и HealthComponent, прикреплённые к персонажу.

6. Оформить отчёт в Markdown с описанием дерева узлов и ключевых фрагментов кода.

5. Пример практического выполнения (базовый уровень)

Дерево узлов:

```
Player (CharacterBody2D)
├── Sprite2D (текстура)
├── CollisionShape2D (коллизия)
└── MovementComponent (скрипт)
```

Код скрипта узла MovementComponent **(GDScript):**

```
extends Node
@export var speed := 200

func move(character: CharacterBody2D, delta: float):
    var input = Vector2(
        Input.get_axis("move_left", "move_right"),
        Input.get_axis("move_up", "move_down")
    )
    character.velocity = input * speed
    character.move_and_slide()
```

Основной скрипт персонажа:

```
extends CharacterBody2D

@onready var movement = $MovementComponent

func _process(delta):
    movement.move(self, delta)
```

6. Варианты индивидуальных заданий

Базовый уровень (1–5)

1. Реализовать перемещение квадрата по стрелкам, отображать текущие координаты в консоли.
2. Добавить спрайт вместо квадрата, настроить зону видимости камеры.
3. Создать два независимых объекта, управляемых разными клавишами (WASD и стрелки).
4. Реализовать поворот спрайта в сторону движения.
5. Добавить компонент HealthComponent с переменной здоровья и методом take_damage().

Средний уровень (6–10)

6. Реализовать LifecycleComponent, который логирует вызовы _ready(), _process(), _exit_tree().
7. Создать систему тегов (групп) и проверку столкновений через get_tree().get_nodes_in_group().
8. Настроить импорт анимации из спрайтового листа через AnimatedSprite2D.
9. Реализовать плавное движение с ускорением/замедлением (аналог lerp).
10. Добавить возможность смены управления (с клавиатуры на мышь) через сигналы.

Продвинутый уровень (11–15)

11. Создать систему компонентов, загружаемых из JSON-конфигурации (файл определяет, какие компоненты прикрепить).
12. Реализовать редактор компонентов в инспекторе с помощью @export_group.
13. Написать скрипт, который выводит в реальном времени граф вызовов _process всех объектов.
14. Интегрировать базовую систему событий (шина событий) для обмена сообщениями между компонентами.
15. Создать минимальную демонстрацию пула объектов на примере частиц (без профилирования — это будет в ЛР6).

7. Методические указания

- При установке Godot на ОС Astra Linux / Ред ОС рекомендуется использовать версию из официального репозитория или Flatpak.
- Для проверки работы _process(delta) выводить значение delta в консоль.
- Типичная ошибка: забыть добавить коллизию — персонаж будет проходить сквозь стены. Убедиться, что корневой узел имеет тип CharacterBody2D.

8. Контрольные вопросы

1. Что такое игровой цикл? Какие этапы он включает?
2. Для чего нужен параметр delta в _process(delta)?
3. В чём отличие _ready() от _process()?
4. Что такое узел (Node) и сцена (Scene) в Godot?
5. Как организовать компонентный подход в Godot?
6. Какие способы взаимодействия между компонентами вы знаете?
7. Как добавить спрайт к узлу?
8. Что делает метод move_and_slide()?

9. Как настроить обработку ввода через Input.get_axis()?
10. Какие ограничения накладывает использование _process() для физики?

9. Требования к отчёту

Отчёт в формате PDF или Markdown, содержащий:

- Скриншот дерева узлов главной сцены.
- Листинг кода скрипта движения и компонента здоровья.
- Скриншот работающего проекта.
- Выводы о проделанной работе.

10. Критерии оценивания

Критерий	Базовый (макс. 5)	Средний (макс. 8)	Продвинутый (макс. 10)
Установка и создание проекта	1	1	1
Реализация движения	2	2	2
Компонентная архитектура	1	2	2
Дополнительные требования (теги, анимация, конфигурация)	0	2	3
Качество отчёта	1	1	1
Ответы на вопросы	0	0	1

Лабораторная работа №2. Конечный автомат для управления анимацией и AI

1. Цель работы

Реализовать конечный автомат (State Machine) для персонажа, управляющего состояниями «Idle», «Walk», «Jump».

2. Компетенции

- **ПК-2** (архитектура), **ПК-3** (разработка).

3. Теоретическое обоснование

3.1. Паттерн «Состояние» (State)

- Позволяет объекту менять своё поведение при изменении внутреннего состояния. Каждое состояние инкапсулируется в отдельный класс.
- В игровых движках FSM (Finite State Machine) используется для управления анимацией, AI, UI.

3.2. Реализация FSM в Godot

- Базовый класс State с методами enter(), update(delta), exit().
- Машина состояний содержит ссылку на текущее состояние и обеспечивает переходы.
- Сигналы или прямое условие (например, is_on_floor) инициируют смену состояния.

4. Задачи работы

1. Создать базовый класс State с виртуальными методами.
2. Реализовать состояния IdleState, WalkState, JumpState для персонажа.
3. Настроить переходы: при нажатии клавиш движения → Walk, при прыжке → Jump, при приземлении → Idle.
4. Добавить анимации для каждого состояния (через AnimationPlayer).
5. Создать простого врага (NPC) с FSM для патрулирования между двумя точками.

5. Пример выполнения (базовый)

Базовый класс State:

```gdscript
class_name State
extends Node

func enter():
    pass

func update(delta):
    pass

func exit():
    pass
```

Состояние Idle:

```gdscript
extends State

func enter():
    parent.animation_player.play("idle")

func update(delta):
    if Input.get_axis("move_left", "move_right") != 0:
        transition.emit("Walk")
```

FSM (на узле персонажа):

```gdscript
var states = {}
var current_state = null

func _ready():
    for child in get_children():
        if child is State:
            states[child.name] = child
            child.parent = self
            child.transition = _on_transition
    current_state = states["Idle"]
    current_state.enter()

func _on_transition(state_name):
    current_state.exit()
    current_state = states[state_name]
    current_state.enter()
```

6. Варианты индивидуальных заданий

Базовый (1–5)

1. Только Idle и Walk (без прыжка).
2. Добавить состояние Attack по нажатию пробела.
3. Реализовать FSM для NPC: патрулирование влево-вправо.
4. Использовать AnimationTree вместо прямого вызова анимаций.

5. Сделать отладочный вывод текущего состояния на экран.

Средний (6–10)

6. Добавить состояние Crouch (приседание) с изменением коллизии.
7. Реализовать иерархическую FSM (суперсостояния: Ground, Air).
8. Создать врага с тремя состояниями: Patrol, Chase, Attack.
9. Интегрировать таймеры для задержки переходов (например, после атаки).
10. Использовать @export для настройки скоростей и границ патрулирования.

Продвинутый (11–15)

11. Реализовать FSM через StateChart (визуальный конечный автомат) с использованием StateChart addon.
12. Написать систему стековых состояний (Pushdown Automaton) для меню и пауз.
13. Добавить состояния на основе данных из JSON.
14. Создать редактор FSM в отдельной сцене для настройки переходов.
15. Реализовать машину состояний, управляемую поведенческими деревьями (BT) — связь с ЛР4.

7. Методические указания

- Для анимации использовать AnimationPlayer и метод play().
- Переходы удобно реализовывать через сигнал transition, эмитируемый из состояний.
- Для NPC патрулирования использовать move_toward() и проверку достижения границ.

8. Контрольные вопросы

1. Какие преимущества даёт паттерн «Состояние» по сравнению с условными операторами?
2. Как в Godot организовать доступ из состояния к родительскому объекту?
3. Чем отличается _process(delta) в состоянии от _physics_process(delta)?
4. Как избежать дублирования кода в состояниях?
5. Что такое иерархический конечный автомат?
6. Как синхронизировать анимацию с переходом состояний?
7. Как реализовать запрет на смену состояния во время анимации?
8. Можно ли использовать FSM для UI? Приведите пример.

9. Какие альтернативы FSM существуют (поведенческие деревья, Utility AI)?
10. Как отладить FSM в Godot (визуализация текущего состояния)?

9. Отчёт

- Диаграмма состояний с переходами.
- Листинги классов состояний и машины.
- Скриншоты анимации и NPC.
- Вывод о применимости FSM.

10. Критерии оценивания (макс. 10 баллов)

Критерий	Баллы
Реализация базовых состояний (Idle, Walk, Jump)	3
Корректные переходы между состояниями	2
Анимация для каждого состояния	1
FSM для NPC с патрулированием	2
Отчёт и ответы на вопросы	2

Лабораторная работа №3. Физика, коллизии и обработка ввода

1. Цель работы

Создать сцену с платформером, использующую физический движок и абстракцию ввода, настроить коллизии и сбор предметов.

2. Формируемые компетенции и индикаторы

- **ПК-3 (разработка): умение настраивать физические тела, слои коллизий и обрабатывать ввод.**
- **ПК-6 (данные и интеграция): понимание обработки событий ввода и абстракции управления.**

3. Теоретическое обоснование

3.1. Физические тела в Godot

- **CharacterBody2D — управляемое физическое тело, предназначенное для персонажей. Предоставляет методы move_and_slide() и move_and_collide(), не реагирует на гравитацию автоматически (её нужно реализовать вручную).**
- **StaticBody2D — статическое тело (земля, платформы), не двигается, используется для коллизий.**
- **RigidBody2D — полностью физическое тело, подчиняется законам физики (сила тяжести, импульсы). Используется для динамических объектов (ящики, бочки).**
- **Area2D — область, не участвующая в физическом отталкивании, но обнаруживающая пересечения (триггеры). Используется для монет, зон поражения, триггеров камеры.**

3.2. Коллизии: слои и маски

- **Каждый физический объект имеет слой (битовая маска, 32 бита) и маску (с какими слоями он сталкивается).**
- **Настройка через инспектор: например, игрок — слой 1, враг — слой 2, предмет — слой 3. Маска игрока может включать слой 2 и 3, чтобы сталкиваться с врагами и собирать предметы.**
- **Area2D использует сигналы body_entered, body_exited и area_entered для обнаружения пересечений.**

3.3. Обработка ввода и InputMap

- **InputMap — встроенная абстракция. Действиям (например, move_left, jump) сопоставляются клавиши (клавиатура, геймпад, мышь).**
- **В коде используется Input.is_action_just_pressed("jump") вместо проверки конкретной клавиши.**

- **Ремаппинг (переназначение) можно сделать через проект → Input Map или программно через InputMap.action_add_event().**

3.4. Гравитация и прыжки

- **Для CharacterBody2D гравитацию реализуют добавлением velocity.y += gravity * delta.**
- **Прыжок: установка velocity.y = jump_velocity, если персонаж на полу (is_on_floor()).**

4. Задачи работы

1. **Создать статическую платформу (StaticBody2D) с прямоугольной коллизией.**
2. **Создать персонажа (CharacterBody2D) со спрайтом, коллизией и скриптом.**
3. **Реализовать движение влево-вправо, гравитацию и прыжок.**
4. **Настроить слои коллизий: игрок (слой 1), платформы (слой 1), монеты (слой 2).**
5. **Создать монету (Area2D) с сигналом body_entered, который увеличивает счёт и удаляет монету.**
6. **Настроить InputMap для действий: move_left, move_right, jump. Добавить возможность переназначения клавиш через файл конфигурации.**
7. **Отображать текущий счёт на экране (Label).**

5. Пример практического выполнения (базовый уровень)

Структура сцены:

text

Main (Node2D)

├── Player (CharacterBody2D)

│ ├── Sprite2D

│ ├── CollisionShape2D (RectangleShape2D)

│ └── Player.gd

├── Ground (StaticBody2D)

│ └── CollisionShape2D (RectangleShape2D)

├── Coin (Area2D)

│ ├── Sprite2D

│ ├── CollisionShape2D (RectangleShape2D)

| └── Coin.gd

└── UI (CanvasLayer)

└── ScoreLabel (Label)

Код игрока (Player.gd):

gdscript

```
extends CharacterBody2D

@export var speed = 300
@export var jump_velocity = -400
@export var gravity = 1200

func _physics_process(delta):
    # Гравитация
    if not is_on_floor():
        velocity.y += gravity * delta

    # Горизонтальное движение
    var direction = Input.get_axis("move_left", "move_right")
    velocity.x = direction * speed

    # Прыжок
    if Input.is_action_just_pressed("jump") and is_on_floor():
        velocity.y = jump_velocity

    move_and_slide()
```

Код монеты (Coin.gd):

gdscript

```
extends Area2D
```

signal collected

```
func _on_body_entered(body):
    if body.name == "Player":
        collected.emit()
        queue_free()
```

Главный скрипт сцены (сбор счёта):

gdscript

```
extends Node2D

var score = 0

func _ready():
    $UI/ScoreLabel.text = "Score: 0"
    for coin in get_tree().get_nodes_in_group("coins"):
        coin.collected.connect(_on_coin_collected)

func _on_coin_collected():
    score += 1
    $UI/ScoreLabel.text = "Score: " + str(score)
```

InputMap (проект → Input Map):

- **move_left: клавиша A, стрелка влево**
- **move_right: клавиша D, стрелка вправо**
- **jump: пробел**

6. Задания для индивидуального выполнения

Базовый уровень (варианты 1–5)

1. **Реализовать платформер с одной платформой, прыжком и сбором 3 монет. Отображать количество собранных монет.**
2. **Добавить движущуюся платформу (AnimationPlayer или _process с перемещением), по которой игрок может ездить.**

3. **Настроить различные типы поверхностей: трава (обычное трение), лёд (уменьшенное трение). Использовать move_and_slide() с параметром floor_max_angle.**
4. **Реализовать двойной прыжок (дополнительный прыжок в воздухе).**
5. **Добавить врага, который движется влево-вправо и наносит урон при касании (уменьшение здоровья игрока).**

Средний уровень (варианты 6–10)

6. **Реализовать систему ускорения: игрок может разбегаться, удерживая клавишу Shift (увеличение speed постепенно).**
7. **Создать зону «шипов» (Area2D), которая убивает игрока при касании и перезапускает уровень.**
8. **Интегрировать Camera2D с сглаживанием (drag margins), чтобы камера следовала за игроком, но не выходила за границы уровня.**
9. **Реализовать поднимаемые предметы (аптечки), которые восстанавливают здоровье (здоровье отображать на UI).**
10. **Настроить переназначение клавиш через UI: кнопка «Управление» → выбор действия → нажатие клавиши → сохранение в user://input_map.cfg.**

Продвинутый уровень (варианты 11–15)

11. **Реализовать систему контроллера (геймпада) с поддержкой вибрации при получении урона. Использовать Input.get_joy_axis().**
12. **Создать «физические» ловушки: падающие камни (RigidBody2D) с разрушением платформ.**
13. **Внедрить систему динамического изменения гравитации (например, переворот уровня на 180°).**
14. **Реализовать запись и воспроизведение ввода (replay system): логировать действия игрока и проигрывать их для демонстрации.**
15. **Сделать многопользовательскую локальную игру (два игрока на одной клавиатуре) с разными наборами клавиш.**

7. Методические указания по выполнению

- **Всегда проверяйте, что для CharacterBody2D включена опция Snap to Floor, чтобы избежать залипания на наклонных поверхностях.**
- **Для движущихся платформ используйте AnimatableBody2D или обычный StaticBody2D с перемещением через код, но не забывайте информировать физический движок через move_and_slide().**

- **Для сборки сцены с множеством монет используйте группу coins и подключайте сигнал в цикле.**
- **При настройке слоёв коллизий: игрок (слой 1), враги (слой 2), монеты (слой 3), платформы (слой 1). Маска игрока должна включать слои 1 и 2, но не 3 (чтобы не сталкиваться с монетами физически, а срабатывал сигнал Area2D). Монета — Area2D с маской 1.**

8. Контрольные вопросы

1. **В чём разница между _process(delta) и _physics_process(delta)? Почему физику обрабатывают во втором?**
2. **Что делает метод move_and_slide() и какие параметры он принимает?**
3. **Как настроить слои коллизий, чтобы монета срабатывала только при касании игрока, но не врага?**
4. **Как реализовать движущуюся платформу, чтобы игрок не соскальзывал с неё?**
5. **Что такое InputMap и как программно добавить новое действие?**
6. **Как сохранить назначенные клавиши между запусками игры?**
7. **Какие сигналы есть у Area2D?**
8. **Как реализовать сбор предметов без использования сигналов (например, через get_overlapping_bodies())?**
9. **Почему после прыжка иногда происходит двойной прыжок? Как это исправить?**
10. **Как сделать так, чтобы камера не отображала область за пределами уровня?**

9. Требования к отчёту

Отчёт в формате PDF/Markdown, содержащий:

- **Схему сцены (дерево узлов) и описание слоёв коллизий.**
- **Листинги скриптов игрока, монеты и главной сцены.**
- **Скриншоты, демонстрирующие движение, прыжок, сбор монеты и изменение счёта.**
- **Описание процесса переназначения клавиш (если реализовано).**
- **Выводы о работе физики и обработки ввода.**

10. Критерии оценивания (максимум 10 баллов)

Критерий	Базовый	Средний	Продвинутый
Движение и прыжки персонажа	2	2	2
Корректная работа коллизий (платформы, стены)	2	2	2
Сбор монет и отображение счёта	2	2	2
Дополнительные механики (движ. платформы, лёд, ускорение, контроллер)	0	2	2
Переназначение клавиш / поддержка геймпада	0	1	1
Качество отчёта и ответы на вопросы	1	1	1
Максимальный балл	7	10	10

Примечание: для получения оценки «отлично» необходимо выполнить все критерии среднего или продвинутого уровня.

Лабораторная работа №4. Игровой AI: поиск пути по навигационной сетке (NavMesh)

1. Цель работы

Реализовать поведение врага, который преследует игрока, используя алгоритм A* на навигационной сетке (NavMesh). Добавить систему выбора действия (Utility AI) на основе состояния.

2. Формируемые компетенции

- **ПК-2 (архитектура): проектирование системы AI с использованием FSM, Utility AI и навигации.**
- **ПК-3 (разработка): реализация навигационных агентов, настройка NavMesh.**
- **ПК-6 (данные и интеграция): анализ сложности алгоритма A* применительно к игровым мирам.**

3. Теоретическое обоснование

3.1. Навигационная сетка (NavMesh)

- **NavMesh — это полигональное представление проходимой области. В Godot создаётся с помощью узла NavigationRegion2D, внутри которого рисуются полигоны (или импортируются из TileMap).**
- **Алгоритм поиска пути (A*) выполняется на графе полигонов. Это значительно быстрее попиксельного поиска.**
- **Узел NavigationAgent2D прикрепляется к движущемуся объекту (врагу) и предоставляет методы: set_target_position(), get_next_path_position().**

3.2. Алгоритм A*

- **Оценивает стоимость пути: f(n) = g(n) + h(n), где g — реальная стоимость от старта до n, h — эвристика (евклидово расстояние или Манхэттен).**
- **В NavMesh эвристика — расстояние до цели.**
- **Godot скрывает детали A*, но студент должен понимать принцип.**

3.3. Utility AI (система полезности)

- **Альтернатива FSM для сложного поведения. Каждое действие оценивается по «полезности» (score) на основе текущего состояния мира.**
- **Выбирается действие с наибольшей полезностью. Например, враг вычисляет: score_chase = 1.0 / distance, score_attack = (health > 30 ? 0.8 : 0.2), score_flee = (health < 20 ? 0.9 : 0.0).**
- **В данной работе предлагается простая Utility AI, работающая параллельно с FSM или внутри неё.**

4. Задачи работы

1. **Построить навигационную сетку в Godot: добавить NavigationRegion2D, нарисовать полигоны, охватывающие пол.**
2. **Создать врага (CharacterBody2D) с NavigationAgent2D, спрайтом и коллизией.**
3. **Реализовать в скрипте врага: получение цели (позиция игрока), обновление target_position у агента, перемещение по get_next_path_position().**
4. **Добавить препятствия (стены) и убедиться, что враг обходит их.**
5. **Реализовать простую Utility AI: враг выбирает между действиями «преследовать», «атаковать», «отступить» на основе здоровья и расстояния до игрока.**
6. **Визуализировать отладочный путь (например, рисовать линию до следующей точки пути).**

5. Пример практического выполнения (базовый уровень)

Структура сцены:

text

Main (Node2D)

├── NavigationRegion2D

│ └── NavigationPolygon (полигоны пола)

├── Player (CharacterBody2D)

├── Enemy (CharacterBody2D)

│ ├── Sprite2D

│ ├── CollisionShape2D

│ ├── NavigationAgent2D

│ └── Enemy.gd

├── Obstacles (StaticBody2D) – стены

Код врага (Enemy.gd):

gdscript

```
extends CharacterBody2D


@export var speed = 150
```

```
@export var attack_range = 50
@export var flee_health_threshold = 30

@onready var agent = $NavigationAgent2D
var target_player = null
var health = 100

func _ready():
    target_player = get_tree().get_first_node_in_group("player")

func _physics_process(delta):
    if not target_player:
        return

    # Utility AI: выбор действия
    var distance = global_position.distance_to(target_player.global_position)
    var action = choose_action(distance, health)

    match action:
        "chase":
            chase(delta)
        "attack":
            attack()
        "flee":
            flee(delta)

func choose_action(distance, health):
    var score_chase = 1.0 / max(distance, 1.0)
    var score_attack = 0.8 if distance < attack_range else 0.0
```

```
    var score_flee = 0.9 if health < flee_health_threshold else 0.0

    var best = "chase"
    var best_score = score_chase
    if score_attack > best_score:
        best = "attack"
        best_score = score_attack
    if score_flee > best_score:
        best = "flee"
    return best

func chase(delta):
    agent.target_position = target_player.global_position
    var next_pos = agent.get_next_path_position()
    var direction = (next_pos - global_position).normalized()
    velocity = direction * speed
    move_and_slide()
    # Поворот спрайта в сторону движения
    if direction.x != 0:
        $Sprite2D.flip_h = direction.x < 0

func attack():
    # Условно: наносим урон, если в диапазоне
    if global_position.distance_to(target_player.global_position) < attack_range:
        target_player.take_damage(10)
        # Таймер задержки атаки (можно добавить cooldown)

func flee(delta):
    var direction = (global_position - target_player.global_position).normalized()
```

```
    velocity = direction * speed * 1.2  # быстрее убегает
    move_and_slide()
```

Визуализация отладочного пути (добавить в Enemy.gd):

```
gdscript
func _draw():
    if agent.get_current_navigation_path().size() > 1:
        var path = agent.get_current_navigation_path()
        for i in range(path.size() - 1):
            draw_line(path[i], path[i+1], Color.RED, 2)
```

6. Задания для индивидуального выполнения

Базовый уровень (варианты 1–5)

1. **Реализовать врага, который просто преследует игрока по NavMesh (без выбора действия). Добавить стену-препятствие.**
2. **Добавить двух врагов с различными параметрами скорости. Настроить избегание друг друга через NavigationAgent2D с параметром avoidance_enabled.**
3. **Создать патрулирующего врага, который ходит между двумя точками, а при попадании игрока в поле зрения переключается в режим преследования.**
4. **Реализовать отображение здорового врага (зелёный) и раненого (красный) в зависимости от здоровья.**
5. **Настроить отладочную визуализацию NavMesh (в редакторе включить Visible Navigation).**

Средний уровень (варианты 6–10)

6. **Реализовать динамическое обновление NavMesh при уничтожении препятствия (например, игрок сбивает стену).**
   Использовать NavigationRegion2D.bake_navigation_polygon() в рантайме.
7. **Добавить группу врагов с координацией: если один враг обнаружил игрока, остальные получают уведомление (сигнал) и тоже начинают преследование.**
8. **Внедрить поведенческое дерево (Behavior Tree) с помощью плагина Beehave или самостоятельной реализации для врага (состояния: Patrol, Chase, Attack).**
9. **Реализовать врага-лучника: он останавливается на дистанции и стреляет снарядами (пул объектов из ЛР6).**
10. **Создать Utility AI с несколькими факторами: расстояние до игрока, уровень здоровья врага, количество союзников поблизости.**

Продвинутый уровень (варианты 11–15)

11. **Реализовать систему тактического укрытия: враг ищет ближайший узел с тегом cover, когда здоровье низкое, и движется к нему.**
12. **Интегрировать навигацию для летающих врагов (в 3D с NavigationRegion3D).**
13. **Написать кастомный скрипт для динамического построения NavMesh на основе карты высот (для процедурно генерируемых уровней).**
14. **Реализовать алгоритм «группового поведения» (flocking) с использованием NavMesh для толпы врагов.**
15. **Создать систему AI, обучаемую с помощью подкрепления (очень упрощённо: враг запоминает позиции, где игрок часто появляется, и патрулирует их).**

7. Методические указания

- **Для построения NavMesh в Godot: создайте NavigationRegion2D, в инспекторе откройте NavigationPolygon, нажмите New NavigationPolygon, затем нажмите Add Polygon и нарисуйте полигоны по полу. Можно использовать кнопку Bake NavMesh из TileMap (если уровень на тайлах).**
- **Убедитесь, что у игрока и врагов есть узел CollisionShape2D. Навигация не работает без коллизий.**
- **При движении по get_next_path_position() враг может останавливаться, если дистанция до следующей точки меньше определённого порога. Это нормально.**
- **Для визуализации пути включите Debug → Visible Collision Shapes или используйте _draw().**
- **Типичная ошибка: враг не обходит стены, потому что полигоны NavMesh перекрывают стены. Следите, чтобы навигационная сетка не заходила внутрь препятствий.**

8. Контрольные вопросы

1. **Что такое навигационная сетка и почему она эффективнее попиксельного поиска пути?**
2. **Как работает алгоритм A*? Какая эвристика используется в 2D-играх?**
3. **Что делает NavigationAgent2D? Как получить следующую точку пути?**
4. **Как обновить NavMesh динамически, если игрок построил стену?**
5. **В чём отличие Utility AI от конечного автомата? Приведите сценарии, где Utility AI лучше.**
6. **Как реализовать избегание препятствий несколькими врагами (local avoidance)?**
7. **Как визуализировать текущий путь в Godot для отладки?**

8. **Какие параметры влияют на качество и скорость поиска пути в NavMesh?**
9. **Можно ли использовать NavMesh для 3D-игр? Какие узлы для этого нужны?**
10. **Как реализовать систему «приоритетов» действий без Utility AI (например, через взвешенные случайные выборы)?**

9. Требования к отчёту

Отчёт должен включать:

- **Скриншоты редактора с нарисованной навигационной сеткой (полигоны).**
- **Листинг кода врага с функциями выбора действия и перемещения.**
- **Скриншоты/видео, демонстрирующие преследование и обход препятствий.**
- **График или таблица, показывающая время поиска пути для разных размеров карты (необязательно, но поощряется).**
- **Анализ выбранного подхода AI: почему именно Utility AI/FSM.**

10. Критерии оценивания (максимум 10 баллов)

Критерий	**Базовый**	**Средний**	**Продвинутый**
Построение NavMesh и перемещение врага к цели	**3**	**3**	**3**
Обход статических препятствий	**2**	**2**	**2**
Реализация Utility AI (или дополнительных состояний)	**2**	**2**	**2**
Динамическое обновление NavMesh / координация группы	**0**	**2**	**1**
Поведенческое дерево / укрытия / flocking	**0**	**0**	**1**
Отчёт и ответы на вопросы	**1**	**1**	**1**
Максимальный балл	**8**	**10**	**10**

Лабораторная работа №5. Сохранение состояния игры и интеграция с REST API

1. Цель работы

Реализовать систему сохранения/загрузки игрового прогресса (позиция игрока, очки, инвентарь) в JSON-файл. Настроить отправку игровых событий на внешний REST API и отображение таблицы лидеров, полученной через HTTP-запросы.

2. Формируемые компетенции и индикаторы

- **ПК-3 (разработка): умение сериализовать данные, работать с файловой системой, выполнять HTTP-запросы.**
- **ПК-6 (данные и интеграция): интеграция с облачными сервисами, понимание клиент-серверного взаимодействия.**
- **ПК-9 (документирование): описание формата сохранений и API-эндпоинтов.**

3. Теоретическое обоснование

3.1. Сохранение и загрузка состояния

- **В Godot данные можно сохранять в текстовом (JSON, ini) или бинарном формате.**
- **FileAccess — основной класс для работы с файлами. Путь user:// указывает на пользовательскую директорию (на каждой платформе свою, но универсальную).**
- **Сериализация: преобразование словаря или пользовательского объекта в строку JSON через JSON.stringify().**
- **Десериализация: JSON.parse_string().**

3.2. Интеграция с REST API

- **Узел HTTPRequest в Godot выполняет асинхронные HTTP-запросы (GET, POST, PUT, DELETE).**
- **Событие request_completed сигнализирует о завершении запроса.**
- **Типичные эндпоинты:**
  - **POST /scores — отправить результат игрока.**
  - **GET /leaderboard — получить список лучших результатов.**
- **Можно использовать бесплатные тестовые сервисы: mockapi.io, jsonplaceholder.typicode.com, или эмулировать локально через простой HTTP-сервер (Python http.server или FastAPI).**

3.3. Формат данных сохранения
Пример JSON:

json

{

"player_name": "Hero",

"score": 1200,

"position": [150, 300],

"inventory": ["sword", "potion"],

"level": 2

}

4. Задачи работы

1. **Разработать структуру данных для сохранения: очки, позиция игрока, список собранных предметов.**
2. **Реализовать функцию save_game(), записывающую текущее состояние в user://savegame.json.**
3. **Реализовать функцию load_game(), восстанавливающую состояние при запуске игры.**
4. **Создать эндпоинт на тестовом REST API (или локальной эмуляции) для приёма рекордов.**
5. **Настроить отправку POST-запроса с результатом игрока при окончании уровня или смерти.**
6. **Получить таблицу лидеров через GET-запрос и отобразить в UI (например, в VBoxContainer с динамически создаваемыми метками).**
7. **Обеспечить корректную обработку ошибок сети (тайм-аут, недоступность сервера).**

5. Пример практического выполнения (базовый уровень)

Структура данных сохранения и методы:

gdscript

extends Node

var save_path = "user://savegame.json"

var player_data = {

"score": 0,

"position": Vector2.ZERO,

```
    "inventory": []
}

func save_game():
    var data_to_save = {
        "score": player_data.score,
        "position": [player_data.position.x, player_data.position.y],
        "inventory": player_data.inventory
    }
    var file = FileAccess.open(save_path, FileAccess.WRITE)
    file.store_string(JSON.stringify(data_to_save))

func load_game():
    if FileAccess.file_exists(save_path):
        var file = FileAccess.open(save_path, FileAccess.READ)
        var content = file.get_as_text()
        var json = JSON.parse_string(content)
        if json:
            player_data.score = json.get("score", 0)
            player_data.position = Vector2(json["position"][0], json["position"][1])
            player_data.inventory = json.get("inventory", [])
```

Отправка рекорда на сервер (POST):

gdscript

```
func submit_score(player_name, final_score):
    var http = HTTPRequest.new()
    add_child(http)
    http.request_completed.connect(_on_submit_completed)
    var body = JSON.stringify({"name": player_name, "score": final_score})
    var headers = ["Content-Type: application/json"]
```

```
    http.request("https://mockapi.io/scores", headers, HTTPClient.METHOD_POST, body)


func _on_submit_completed(result, response_code, headers, body):
    if response_code == 200:
        print("Score submitted")
    else:
        print("Error: ", response_code)
```

Получение таблицы лидеров (GET):

```
gdscript
func fetch_leaderboard():
    var http = HTTPRequest.new()
    add_child(http)
    http.request_completed.connect(_on_leaderboard_received)
    http.request("https://mockapi.io/leaderboard")


func _on_leaderboard_received(result, response_code, headers, body):
    if response_code == 200:
        var json = JSON.parse_string(body.get_string_from_utf8())
        display_leaderboard(json)
```

6. Задания для индивидуального выполнения

Базовый уровень (варианты 1–5)

1. **Реализовать сохранение/загрузку только очков и позиции игрока. При загрузке восстанавливать позицию.**
2. **Добавить в сохранение имя игрока (через диалог ввода при первом запуске).**
3. **Отправлять рекорд на https://postman-echo.com/post (тестовый эхо-сервер) и выводить ответ в консоль.**
4. **Отображать таблицу лидеров из локального JSON-файла (без сети) — симуляция.**
5. **Добавить автоматическое сохранение каждые 30 секунд (таймер).**

Средний уровень (варианты 6–10)

6. **Реализовать сохранение нескольких слотов (slot 1, slot 2, slot 3) с выбором через UI.**
7. **Использовать шифрование сохранений (простейшее XOR с ключом) для защиты от редактирования.**
8. **Настроить автоопределение недоступности сети: если запрос не удался, сохранить рекорд в локальную очередь и повторить при следующем запуске.**
9. **Добавить в таблицу лидеров пагинацию (загрузка следующих 5 записей по кнопке).**
10. **Реализовать интеграцию с Яндекс.Cloud Functions: написать простую функцию на Python, которая принимает POST и сохраняет в Object Storage, развернуть локально в Docker.**

Продвинутый уровень (варианты 11–15)

11. **Реализовать синхронизацию сохранений между устройствами: при загрузке игры проверять, есть ли на сервере более новая версия (по timestamp) и предлагать загрузить её.**
12. **Внедрить OAuth-авторизацию (через сервис типа Firebase или Keycloak) для привязки рекордов к аккаунту.**
13. **Реализовать систему достижений (achievements), которые хранятся на сервере и синхронизируются.**
14. **Создать собственный минимальный бэкенд на Node.js/Express (или FastAPI) с хранением рекордов в SQLite, развернуть его в локальном Docker-контейнере, связать с игрой.**
15. **Интегрировать WebSocket-уведомления (через Godot WebSocketClient) для обновления таблицы лидеров в реальном времени при появлении нового рекорда.**

7. Методические указания

- **В Godot 4.x FileAccess заменил старый File. Используйте FileAccess.open().**
- **Для отладки HTTP-запросов включите в редакторе Проект → Настройки → Отладка → Удалённая отладка → Режим или используйте print() в обработчике сигнала.**
- **При работе с HTTPRequest не забывайте удалять узел после завершения, чтобы избежать утечек: http.queue_free() или remove_child(http).**
- **Mock-сервер для тестирования можно поднять командой Python: python -m http.server 8000 и поместить leaderboard.json в ту же директорию, но для POST**

потребуется более сложный обработчик. Используйте mockapi.io (бесплатно до 1000 запросов).

- **В JSON координаты храните в виде массива [x, y], поскольку Vector2 не сериализуется напрямую.**

8. Контрольные вопросы

1. **Какие способы хранения данных в Godot существуют? В чём преимущества JSON перед бинарным форматом?**
2. **Как правильно работать с FileAccess? Что означает путь user:// и res://?**
3. **Как выполнить асинхронный HTTP-запрос в Godot? Как обработать ответ?**
4. **Что такое CORS и почему при запросе к стороннему API может возникнуть ошибка?**
5. **Как сериализовать пользовательский класс (Resource) в JSON?**
6. **Какие данные не следует сохранять в открытом виде? Как их зашифровать?**
7. **Как реализовать повторную отправку запроса при ошибке сети?**
8. **Как отличить ошибку сервера от ошибки сети в HTTPRequest?**
9. **Как в Godot отобразить динамический список (Leaderboard) с прокруткой?**
10. **Какие меры безопасности нужно принять при отправке рекордов, чтобы игрок не подделал счёт?**

9. Требования к отчёту

Отчёт должен содержать:

- **Описание формата JSON для сохранения игры.**
- **Листинги функций сохранения/загрузки и HTTP-запросов.**
- **Скриншоты успешной отправки рекорда (можно из консоли браузера для mock-сервера).**
- **Скриншот таблицы лидеров, отображаемой в игре.**
- **Анализ потенциальных проблем (задержка сети, отсутствие соединения).**

10. Критерии оценивания (максимум 10 баллов)

Критерий	**Базовый**	**Средний**	**Продвинутый**
Сохранение и загрузка состояния (JSON, файл)	**3**	**3**	**2**

Критерий	Базовый	Средний	Продвинутый
Отправка POST-запроса с рекордом	2	2	2
Отображение таблицы лидеров (GET)	2	2	2
Обработка ошибок сети / очереди	0	1	1
Шифрование / многопользовательность / облачная синхронизация	0	1	2
Отчёт и ответы на вопросы	1	1	1
Максимальный балл	8	10	10

Лабораторная работа №6. Профилирование и оптимизация. Пул объектов

1. Цель работы

Выявить узкое место производительности в игровой сцене (частое создание/удаление объектов) и устранить его с помощью паттерна «Пул объектов» (Object Pool). Провести профилирование, сравнить FPS и время кадра до и после оптимизации.

2. Формируемые компетенции

- **ПК-2 (архитектура): выбор и реализация паттерна Object Pool для управления памятью.**
- **ПК-3 (разработка): использование встроенного профилировщика Godot для анализа производительности.**

3. Теоретическое обоснование

3.1. Паттерн Object Pool

- **Цель: переиспользовать ранее созданные объекты вместо их уничтожения и повторного создания.**
- **Применим для часто используемых объектов: пули, осколки, частицы, монеты, враги.**
- **Основные операции: get() (извлечь неактивный объект из пула, активировать и вернуть) и release() (деактивировать и вернуть в пул).**

3.2. Профилирование в Godot

- **Встроенный профилировщик: отладка → мониторы (FPS, время кадра, количество объектов, память).**
- **Можно программно замерять FPS через Engine.get_frames_per_second() и время обработки кадра.**
- **Инструмент Performance (класс Performance в GDScript) предоставляет метрики: get_monitor(Performance.OBJECT_COUNT).**

3.3. Проблема без пула

- **При спавне 100 пуль в секунду и их удалении каждый кадр выделяется новая память, растёт нагрузка на сборщик мусора (GDScript управляется автоматически, но создание/удаление узлов требует затрат).**
- **Падение FPS особенно заметно на мобильных устройствах и в web-сборках.**

4. Задачи работы

1. **Создать тестовую сцену, в которой по таймеру (например, 0.05 сек) спавнится объект (пуля или частица), который перемещается по экрану и через 1 секунду удаляется (queue_free()).**

2. **Замерить FPS и время кадра (через профилировщик и Engine.get_frames_per_second()) при нагрузке 50–100 одновременных объектов.**
3. **Реализовать пул объектов: предварительно создать pool_size экземпляров, сделать их неактивными (visible = false, process_mode = DISABLED).**
4. **Переработать спавн: вместо instantiate() брать объект из пула, активировать, задать позицию и скорость.**
5. **Вместо queue_free() возвращать объект в пул (деактивировать).**
6. **Повторить замеры производительности и сравнить результаты (таблица FPS до/после).**
7. **Оформить выводы об эффективности пула.**

5. Пример практического выполнения (базовый уровень)

Сценарий спавна без пула (для теста производительности):

gdscript

```
extends Node2D

@export var bullet_scene: PackedScene
var timer: Timer

func _ready():
    timer = Timer.new()
    timer.wait_time = 0.05
    timer.timeout.connect(_spawn_bullet)
    add_child(timer)
    timer.start()

func _spawn_bullet():
    var bullet = bullet_scene.instantiate()
    add_child(bullet)
    bullet.position = Vector2(100, 100)
    # Через 1 секунду удалить
```

```
    bullet.get_node("Timer").start(1.0)

# В сцене пули:
func _on_timer_timeout():
    queue_free()
```

Реализация пула:

```
gdscript
extends Node2D

@export var bullet_scene: PackedScene
@export var pool_size = 50
var pool = []

func _ready():
    for i in range(pool_size):
        var bullet = bullet_scene.instantiate()
        bullet.active = false
        add_child(bullet)
        pool.append(bullet)
    # Таймер спавна
    var timer = Timer.new()
    timer.wait_time = 0.05
    timer.timeout.connect(_spawn_from_pool)
    add_child(timer)
    timer.start()

func _spawn_from_pool():
    var bullet = get_from_pool()
    if bullet:
```

```
        bullet.active = true
        bullet.position = Vector2(100, 100)
        bullet.get_node("LifetimeTimer").start(1.0)

func get_from_pool():
    for b in pool:
        if not b.active:
            return b
    return null  # пул исчерпан (можно расширить)

func return_to_pool(bullet):
    bullet.active = false
    bullet.velocity = Vector2.ZERO
    bullet.visible = false  # если нужно
```

В скрипте пули:

```
gdscript
var active = false
var speed = 300

func _process(delta):
    if active:
        position += Vector2(speed * delta, 0)

func _on_lifetime_timeout():
    get_parent().return_to_pool(self)  # родитель — менеджер пула
```

Измерение FPS:

```
gdscript
func _process(delta):
    if Engine.get_frames_per_second() < 30:
```

```
print("Low FPS!")
```

6. Задания для индивидуального выполнения

Базовый уровень (варианты 1–5)

1. **Реализовать пул для 30 пуль, сравнить FPS при спавне 20 пуль в секунду.**
2. **Сделать пул для врагов, которые появляются из точки спавна и движутся к игроку.**
3. **Добавить в пул возможность динамического расширения (если объекты закончились, создать новый и добавить в пул).**
4. **Вывести на экран отладочную информацию: количество активных объектов, размер пула, текущий FPS.**
5. **Измерить время выполнения _process в профилировщике до и после оптимизации.**

Средний уровень (варианты 6–10)

6. **Реализовать пул, работающий с разными типами объектов (пули, враги, частицы) через единый менеджер, использующий словарь: {type: pool_array}.**
7. **Добавить объекту обратный отсчёт времени жизни без отдельного узла Timer (использовать переменную lifetime и уменьшать в _process).**
8. **Провести профилирование с использованием встроенного профайлера и снять скриншоты графиков. Сравнить memory usage.**
9. **Реализовать пул для эффектов частиц (например, взрыв при уничтожении врага).**
10. **Визуализировать разницу в производительности в виде графика (FPS vs количество объектов) для двух подходов.**

Продвинутый уровень (варианты 11–15)

11. **Создать универсальный пул с поддержкой предварительного выделения памяти и кэшированием экземпляров сцены (используя PackedScene.instantiate()).**
12. **Интегрировать пул с системой объектно-событийного менеджера, автоматически возвращающего объекты при выходе за границы экрана.**
13. **Реализовать пул, работающий в многопоточном режиме (используя WorkerThreadPool для фоновой подготовки объектов).**
14. **Профилировать игру на мобильном устройстве (или эмуляторе) и сравнить эффективность пула в условиях ограниченной памяти.**
15. **Разработать плагин для Godot, который автоматически заменяет instantiate() и queue_free() на пул через анализ графа сцены (система аспектов).**

7. Методические указания

- **Для профилирования откройте окно Отладка → Мониторы. Добавьте FPS, Время кадра, Количество объектов.**
- **Используйте print(Performance.get_monitor(Performance.OBJECT_COUNT)) для отслеживания количества узлов.**
- **При тестировании без пула легко получить падение FPS до 15-20. В пуле FPS должен оставаться около 60.**
- **Не храните в пуле объекты с активными таймерами или обработкой _process, если они неактивны — отключайте их через set_process(false) или process_mode = PROCESS_MODE_DISABLED.**
- **Для сцены пули убедитесь, что её корневой узел — CharacterBody2D или Area2D, чтобы коллизии отключались вместе с активностью.**

8. Контрольные вопросы

1. **В чём заключается проблема частого создания и удаления объектов в играх?**
2. **Как работает паттерн Object Pool? Приведите примеры использования.**
3. **Какие метрики производительности нужно отслеживать при профилировании?**
4. **Как в Godot отключить узел так, чтобы он не тратил ресурсы на обработку?**
5. **Как измерить время выполнения отдельного фрагмента кода в GDScript?**
6. **Почему пул объектов не всегда эффективен (overhead на управление пулом)?**
7. **В каких случаях лучше использовать пул, а когда — обычное создание объектов?**
8. **Что такое сборщик мусора в GDScript и как пул помогает снизить его нагрузку?**
9. **Как протестировать производительность на целевой платформе (например, Web)?**
10. **Как автоматически возвращать объекты в пул при выходе за границы экрана?**

9. Требования к отчёту

Отчёт должен включать:

- **Схему работы пула (диаграмма последовательности).**
- **Листинги классов пула и спавна.**
- **Таблицу с замерами FPS и времени кадра (не менее 3 замеров для каждого подхода).**
- **Скриншоты профилировщика до и после оптимизации.**

- **Выводы о приросте производительности и применимости пула в данном сценарии.**

10. Критерии оценивания (максимум 10 баллов)

Критерий	**Базовый**	**Средний**	**Продвинутый**
Реализация спавна с удалением (без пула) и замер FPS	**2**	**2**	**2**
Реализация пула объектов (базового)	**3**	**3**	**2**
Переход на использование пула вместо создания/удаления	**2**	**2**	**2**
Динамическое расширение пула / поддержка разных типов	**0**	**1**	**1**
Анализ производительности (таблицы, графики, профилировщик)	**1**	**2**	**2**
Отчёт и ответы на вопросы	**1**	**1**	**1**
Максимальный балл	**9**	**11**	**10**

Лабораторная работа №7. Комплексный игровой проект. Подготовка билда и документации

1. Цель работы

Разработать законченную мини-игру (2D платформер / top-down shooter / аркада) с интеграцией всех изученных механик: управление, физика, AI врага, система сохранений, пул объектов, UI (меню, настройки, Game Over). Экспортировать билд под Windows и Web. Подготовить техническую документацию и публично защитить проект.

2. Формируемые компетенции и индикаторы

- **ПК-2 (архитектура): проектирование игровой архитектуры с использованием изученных паттернов.**
- **ПК-3 (разработка): полноценная реализация игрового проекта на Godot/Unity.**
- **ПК-6 (данные и интеграция): интеграция с REST API, работа с сохранениями.**
- **ПК-9 (документирование): подготовка Game Design Document, инструкции по сборке, архитектурной диаграммы.**

3. Теоретическое обоснование

3.1. Жизненный цикл разработки игры

- **Концепция → Прототип → Alpha (основные механики) → Beta (полировка, баланс) → Релиз.**
- **В рамках лабораторной работы реализуется минимально жизнеспособный продукт (MVP), готовый к публикации.**

3.2. Архитектура игрового проекта

- **Разделение на сцены: MainMenu, Game, Settings, GameOver, Leaderboard.**
- **Использование автозагружаемых синглтонов (autoload) для глобального состояния: GameManager, SaveSystem, AudioManager.**
- **Паттерны: State Machine (состояния игры: MENU, PLAYING, PAUSED, GAMEOVER), Object Pool, Observer (сигналы), Singleton.**

3.3. Экспорт проекта

- **Godot: Проект → Экспорт → Добавить платформу (Windows Desktop, Web).**
- **Для Web потребуется настроить шаблон HTML (можно стандартный). Поддержка WebGL.**
- **Упаковка ресурсов: оптимизация текстур, сжатие звуков.**

3.4. Техническая документация

- **Game Design Document (GDD): концепция, жанр, целевая аудитория, управление, механики, арт-стиль.**
- **Руководство по сборке: как скомпилировать проект из исходников, какие зависимости нужны.**
- **Архитектурная диаграмма: диаграмма классов/узлов, схема взаимодействия сцен и синглтонов.**

4. Задачи работы

1. **Спроектировать и реализовать мини-игру (выбрать жанр: платформер, шутер, аркада, головоломка). Минимальные требования:**
    - **Управление персонажем (клавиатура / мышь).**
    - **Физические коллизии (стены, платформы, враги).**
    - **AI хотя бы одного типа врага (патрулирование/преследование с использованием навигации или простой логики).**
    - **Система сохранения/загрузки (очки, позиция, инвентарь/прогресс уровня).**
    - **Пул объектов (снаряды, враги, эффекты).**
    - **UI: главное меню, настройки (громкость, управление), экран Game Over, отображение счёта/жизней.**
2. **Интегрировать таблицу лидеров через REST API (отправка рекорда, получение топа).**
3. **Экспортировать проект под Windows (выполняемый .exe) и Web (HTML5).**
4. **Подготовить документацию:**
    - **GDD (1–2 страницы).**
    - **Архитектурную диаграмму.**
    - **Инструкцию по запуску из исходников и из билда.**
5. **Провести публичную защиту: демонстрация игры (live или запись), презентация (10–15 слайдов), ответы на вопросы по архитектуре, паттернам и реализации.**

5. Пример практического выполнения (базовый уровень)

Концепция игры (пример): *Space Defender* — top-down shooter, игрок управляет космическим кораблём, уничтожает врагов, собирает бонусы. После смерти отображается окно Game Over с возможностью ввести имя и отправить рекорд.

Структура автозагружаемых синглтонов (autoload):

- **GameManager.gd — состояние игры (score, lives, текущая сцена), методы game_over(), load_next_level().**
- **SaveSystem.gd — сохранение/загрузка прогресса (лучший счёт, разблокированные уровни).**
- **AudioManager.gd — управление музыкой и звуками.**
- **LeaderboardManager.gd — HTTP-запросы для таблицы лидеров.**

Пример GameManager:

gdscript

```
extends Node


enum GameState { MENU, PLAYING, PAUSED, GAMEOVER }
var current_state = GameState.MENU
var score = 0
var lives = 3

signal score_updated(new_score)
signal lives_updated(new_lives)
signal game_over_signal(final_score)

func add_points(points):
    score += points
    score_updated.emit(score)

func decrease_life():
    lives -= 1
    lives_updated.emit(lives)
    if lives <= 0:
        game_over()

func game_over():
```

```
    current_state = GameState.GAMEOVER
    game_over_signal.emit(score)
    get_tree().change_scene_to_file("res://scenes/GameOver.tscn")
```

Основная игровая сцена:

- **Игрок (CharacterBody2D) с движением, стрельбой (пулы пуль).**
- **Враги (навигация NavMesh или простая логика движения).**
- **Спавнер врагов с использованием пула.**
- **Сбор монет/бонусов (Area2D).**

Пул объектов для пуль (согласно ЛР6):

gdscript

```
extends Node

var bullet_pool = []
var bullet_scene = preload("res://scenes/Bullet.tscn")
var pool_size = 30

func _ready():
    for i in range(pool_size):
        var b = bullet_scene.instantiate()
        b.active = false
        add_child(b)
        bullet_pool.append(b)

func shoot(origin, direction):
    var bullet = get_from_pool()
    if bullet:
        bullet.active = true
        bullet.global_position = origin
        bullet.direction = direction
```

Сохранение прогресса (лучший счёт):

gdscript

```
func save_best_score(score):
    var data = {"best_score": score}
    var file = FileAccess.open("user://best_score.json", FileAccess.WRITE)
    file.store_string(JSON.stringify(data))


func load_best_score():
    if FileAccess.file_exists("user://best_score.json"):
        var file = FileAccess.open("user://best_score.json", FileAccess.READ)
        var json = JSON.parse_string(file.get_as_text())
        return json.get("best_score", 0)
    return 0
```

Экспорт:

- **Windows: Проект → Экспорт → Добавить Windows Desktop → Выбрать путь → Экспорт проекта.**
- **Web: Добавить Web → Настроить Страница HTML (можно оставить по умолчанию) → Экспорт. Результат: файлы .html, .wasm, .pck.**

6. Задания для индивидуального выполнения (вариативность)

Базовый уровень (жанры 1–5)

1. **2D платформер: 1 уровень, 3 врага (патруль), 5 монет, сохранение очков и позиции при загрузке.**
2. **Top-down shooter: 2 типа врагов, пул пуль, таблица лидеров (лучшие 5 результатов).**
3. **Аркада (Pong-подобная): управление ракеткой, AI противника, сохранение рекорда между сессиями.**
4. **Головоломка с физикой: перемещаемые ящики, сохранение прогресса по уровням.**
5. **Кликер (Idle-игра): накопление ресурсов, сохранение в JSON, бонусы.**

Средний уровень (6–10)

6. **Платформер с несколькими уровнями: меню выбора уровня, сохранение пройденных уровней.**
7. **Shooter с прокачкой: очки опыта, улучшения (скорострельность, урон), сохранение характеристик.**
8. **Игра с процедурной генерацией уровня: карта генерируется случайно, сохранение сида генерации.**
9. **Многопользовательская локальная игра (2 игрока на одной клавиатуре): управление разными клавишами, режим «vs» или «кооператив».**
10. **Игра с внутриигровым магазином и виртуальной валютой: покупка скинов, сохранение баланса в JSON, синхронизация с сервером (опционально).**

Продвинутый уровень (11–15)

11. **Полноценный rogue-lite: случайная генерация комнат, сохранение прогресса перманентных улучшений.**
12. **Игра с поддержкой геймпада (XInput): полная настройка управления, вибрация.**
13. **Десктопная игра с интеграцией Steam API (или аналогом) для достижений и облачных сохранений (через GodotSteam).**
14. **Клиент-серверная игра с авторизацией: свой бэкенд на Node.js / FastAPI, WebSocket для реального времени.**
15. **Сборка и публикация на itch.io / VK Play: настройка метаданных, трейлера, страницы.**

7. Методические указания

- **Для автозагрузки (синглтонов) в Godot: Проект → Настройки проекта → Autoload → добавить скрипт (например, GameManager.gd).**
- **Используйте SceneTree.change_scene_to_file() для перехода между сценами.**
- **Для предотвращения потери данных при закрытии игры вызывайте save_game() в NOTIFICATION_WM_CLOSE_REQUEST или в _notification().**
- **При экспорте в Web учтите, что некоторые функции (например, доступ к файловой системе user://) работают ограниченно (используется IndexedDB, сохраняется между сессиями).**
- **Для тестирования на разных платформах используйте удалённую отладку (Godot позволяет отлаживать Web-версию через браузер).**
- **В отчёт включите видео-демонстрацию (ссылка на YouTube или встроенный плеер), так как скриншоты не передают динамику.**

8. Контрольные вопросы (для защиты)

1. **Почему вы выбрали именно эту архитектуру? Какие альтернативы рассматривали?**
2. **Как в вашем проекте реализована система сохранений? Можно ли восстановить данные после сбоя?**
3. **Какие паттерны проектирования вы применили и зачем?**
4. **Как вы оптимизировали производительность (пул, отложенная загрузка, LOD)?**
5. **Какие трудности возникли при экспорте под Web? Как их решили?**
6. **Как обеспечивается безопасность рекордов (защита от подделки)?**
7. **Какие метрики качества кода вы проверяли?**
8. **Как вы организовали работу в команде (если была командная разработка)?**
9. **Как бы вы масштабировали проект, если бы нужно было добавить сетевую игру?**
10. **Какие уроки вы извлекли, завершив этот проект?**

9. Требования к отчёту (расширенный)

Отчёт должен включать следующие разделы:

1. **Game Design Document (GDD)**
    - **Название, жанр, краткое описание.**
    - **Целевая платформа (Windows, Web).**
    - **Основные механики и управление.**
    - **Арт-стиль, звуковое сопровождение.**
2. **Архитектура проекта**
    - **Диаграмма сцен и автозагружаемых синглтонов.**
    - **Описание ключевых классов/скриптов (что за что отвечает).**
    - **Схема взаимодействия сигналов.**
3. **Реализация ключевых механик**
    - **Физика и управление (листинг кода).**
    - **AI врага (листинг, объяснение алгоритма).**
    - **Система сохранений (формат JSON, листинг).**
    - **Пул объектов (листинг, анализ эффективности).**
    - **Интеграция с REST API (листинг, скриншоты запросов).**
4. **UI и пользовательский опыт**

- **Скриншоты главного меню, настроек, экрана Game Over.**
- **Обоснование расположения элементов.**

5. **Тестирование и отладка**
   - **Какие тесты проводили (ручное, модульное).**
   - **Найденные ошибки и их исправление.**
6. **Инструкция по сборке и запуску**
   - **Как клонировать репозиторий.**
   - **Как открыть в Godot и запустить из редактора.**
   - **Как экспортировать билды под Windows и Web.**
7. **Результаты защиты**
   - **Ссылка на видео-демонстрацию.**
   - **Слайды презентации (встроить в отчёт или приложить отдельно).**
   - **Выводы и самооценка.**

10. Критерии оценивания (максимум 30 баллов)

Критерий	**Базовый (1–5)**	**Средний (6–10)**	**Продвинутый (11–15)**
Работоспособность билдов	**Windows (3 балла)**	**Windows + Web (5 баллов)**	**Windows + Web + Linux (6 баллов)**
Полнота механик	**Минимум 3 обязательные механики (6 баллов)**	**4–5 механик (8 баллов)**	**Все 6 обязательных механик (10 баллов)**
Качество кода и архитектуры	**Базовое соблюдение стиля (3 балла)**	**Паттерны, автозагрузка, комментирование (5 баллов)**	**Промышленный стиль, документация функций (6 баллов)**

Критерий	Базовый (1–5)	Средний (6–10)	Продвинутый (11–15)
Интеграция с REST API	Отправка рекорда (2 балла)	Отправка + получение топа (3 балла)	+ обработка ошибок, офлайн-режим (4 балла)
Документация (GDD, архитектура, инструкция)	3 балла	4 балла	5 баллов
Презентация и защита	2 балла	3 балла	4 балла
Бонус: публикация на платформе (itch.io / VK Play)	+1	+2	+3
Максимальный балл	20	28	38 (с бонусами)