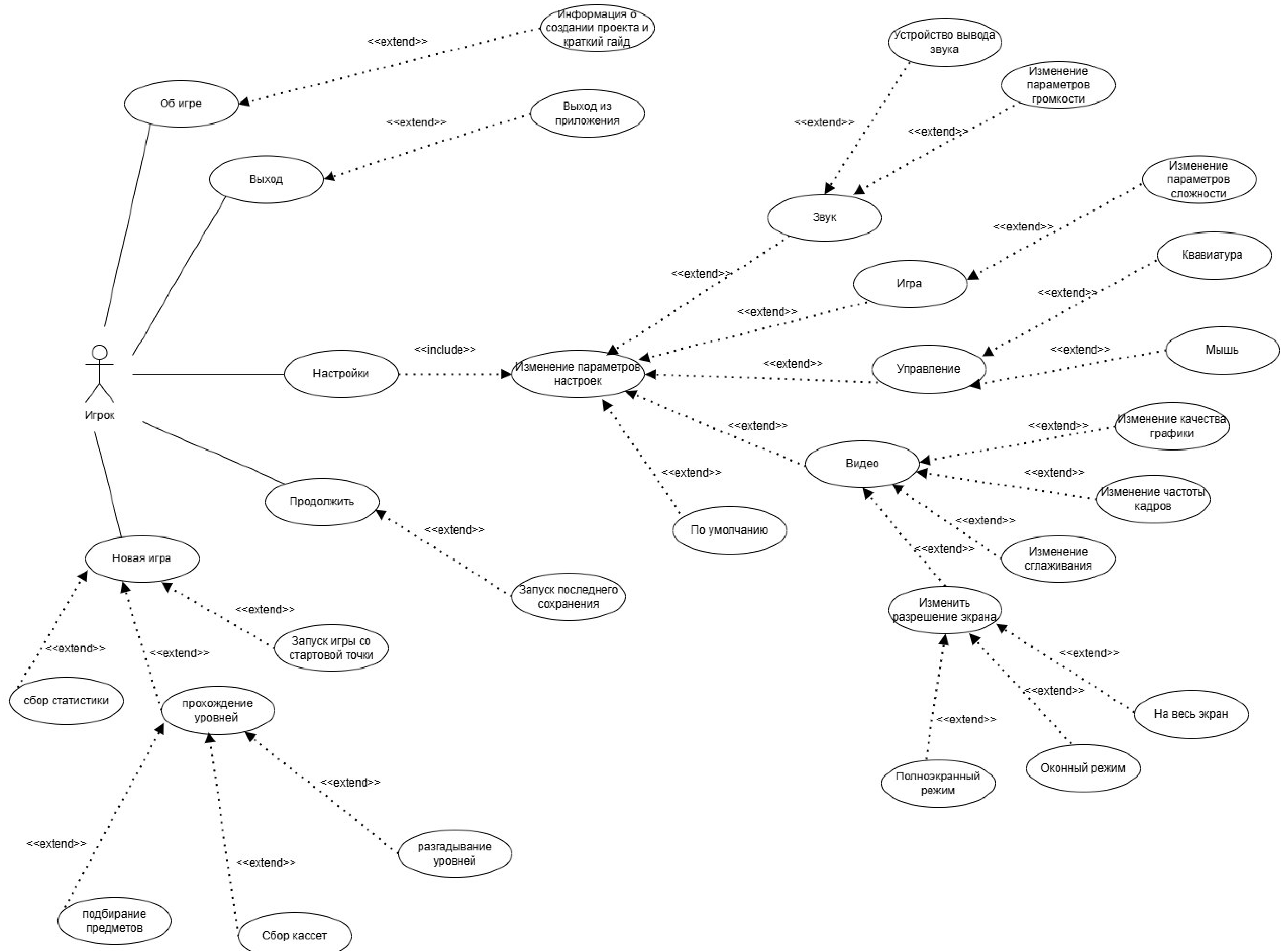
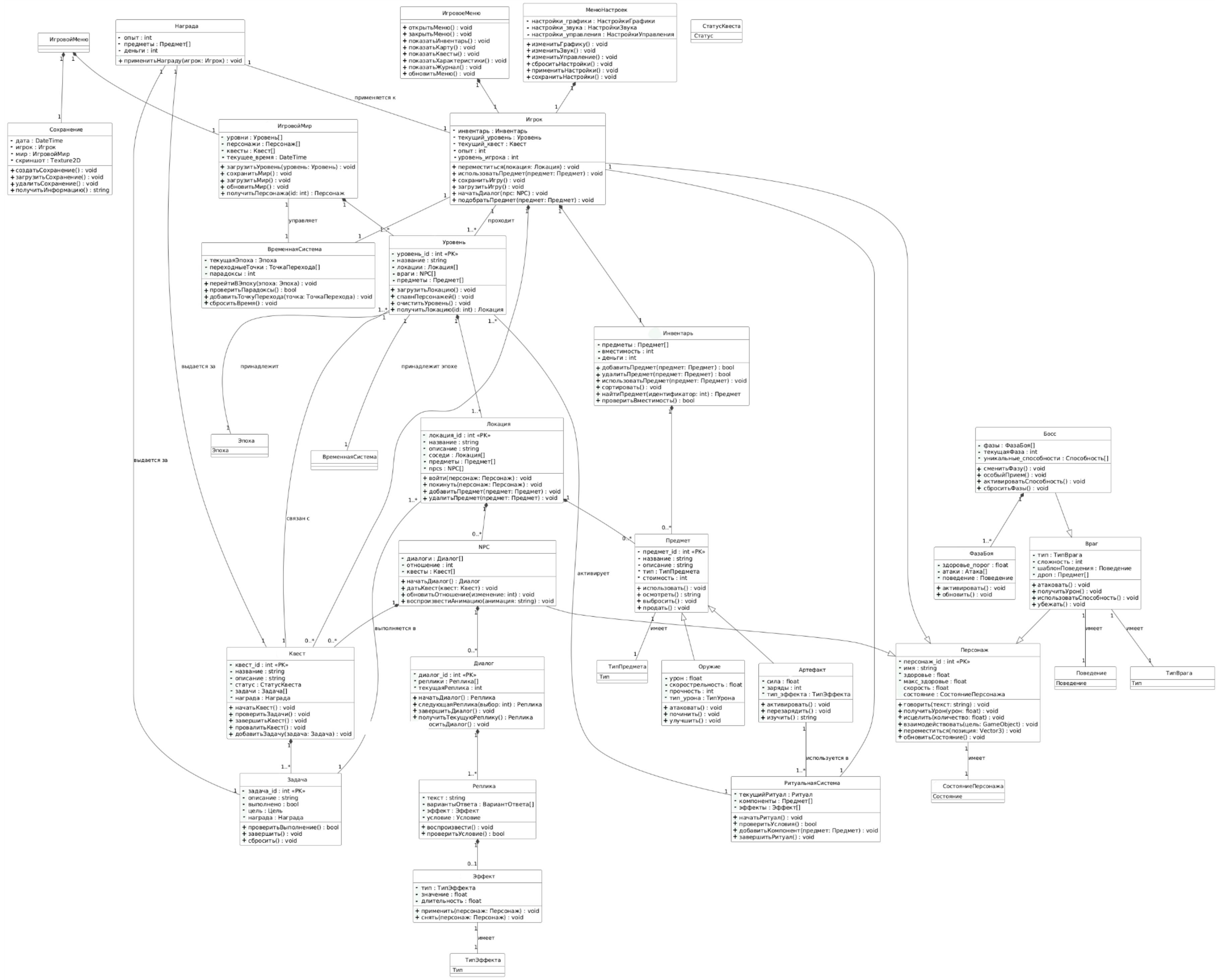
Приложение А
Диаграмма вариантов использования

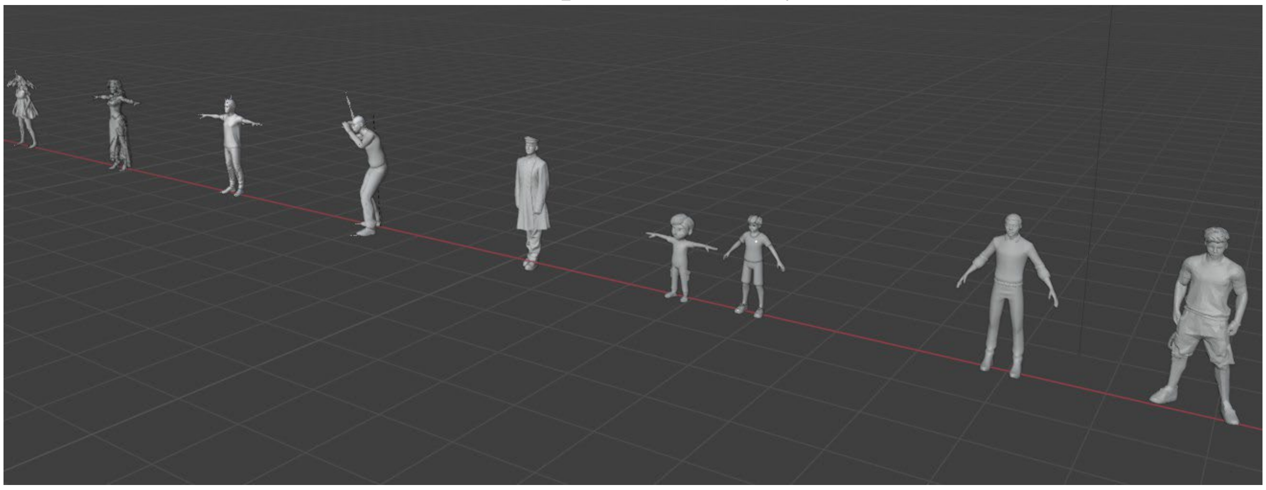

Информация о создании проекта и краткий гайд
<<extend>>
Об игре
Выход из приложения
<<extend>>
Выход
Устройство вывода звука
Изменение параметров громкости
<<extend>>
<<extend>>
Звук
Изменение параметров сложности
<<extend>>
Игра
Квавиатура
<<extend>>
<<extend>>
<<extend>>
Игрок
Настройки
<<include>>
Изменение параметров настроек
<<extend>>
Управление
<<extend>>
Мышь
<<extend>>
Изменение качества графики
<<extend>>
Видео
<<extend>>
Изменение частоты кадров
<<extend>>
По умолчанию
Продолжить
<<extend>>
<<extend>>
Изменение сглаживания
<<extend>>
Новая игра
Запуск последнего сохранения
Изменить разрешение экрана
<<extend>>
Запуск игры со стартовой точки
<<extend>>
<<extend>>
<<extend>>
<<extend>>
сбор статистики
прохождение уровней
На весь экран
<<extend>>
Полноэкранный режим
Оконный режим
<<extend>>
разгадывание уровней
<<extend>>
<<extend>>
подбирание предметов
Сбор кассет

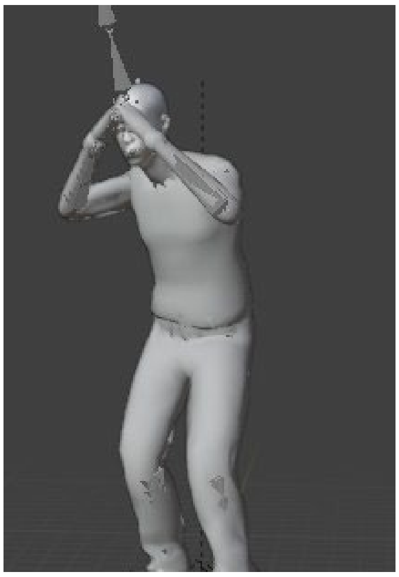
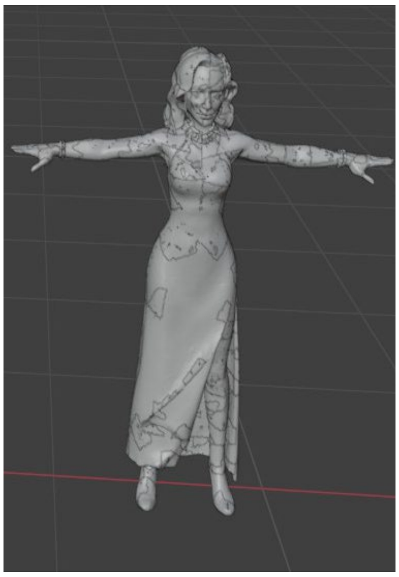
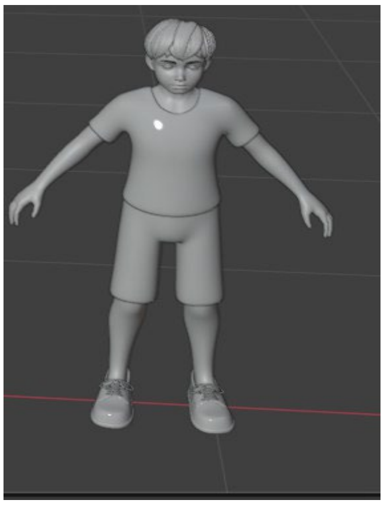
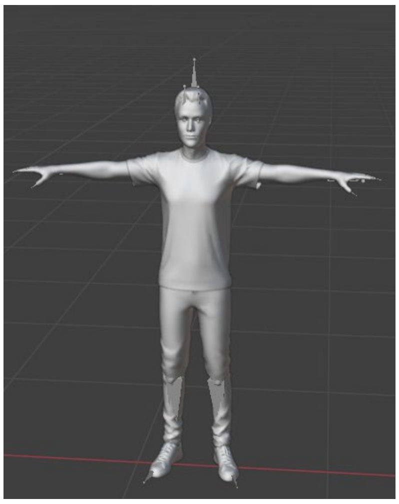
Приложение Б
Диаграмма классов

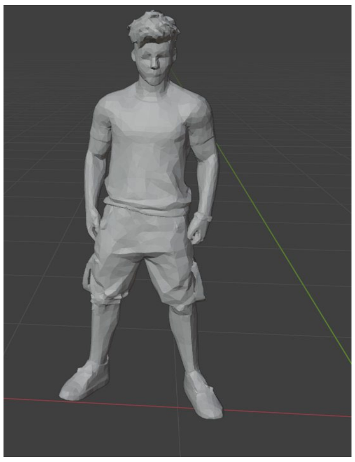
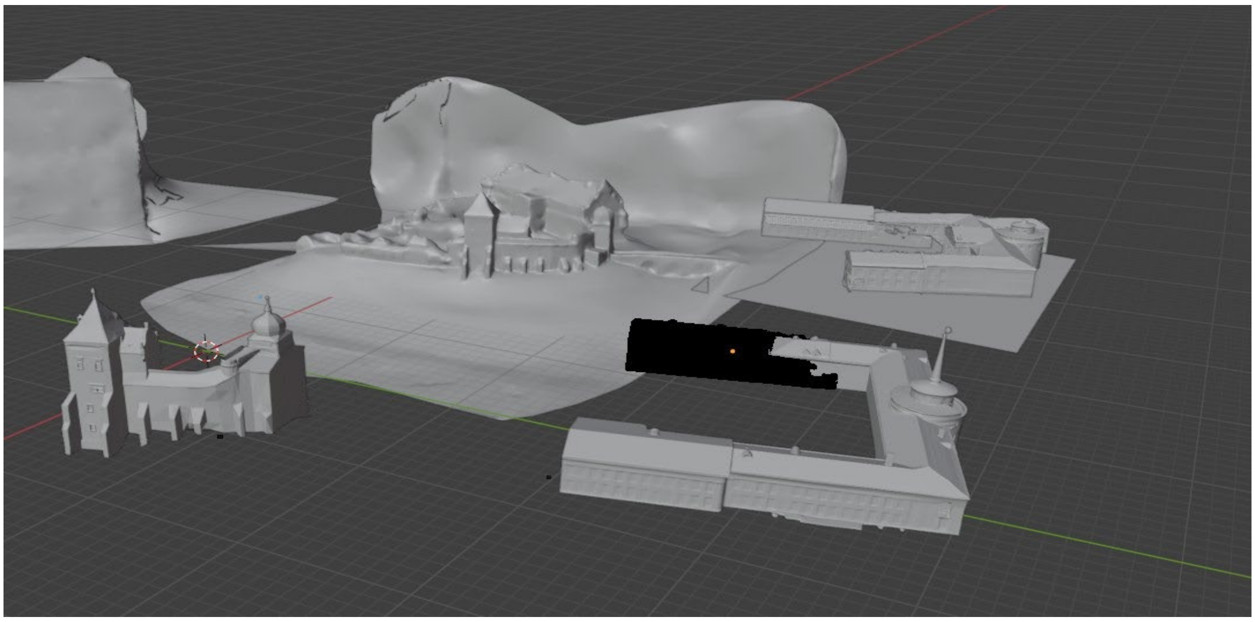

ИгровойМеню
Награда
- опыт : int
- предметы : Предмет[]
- деньги : int
+ применитьНаграду(игрок: Игрок) : void
ИгровоеМеню
+ открытьМеню() : void
+ закрытьМеню() : void
+ показатьИнвентарь() : void
+ показатьКарту() : void
+ показатьКвесты() : void
+ показатьХарактеристики() : void
+ показатьЖурнал() : void
+ обновитьМеню() : void
МенюНастроек
- настройки_графики : НастройкиГрафики
- настройки_звука : НастройкиЗвука
- настройки_управления : НастройкиУправления
+ изменитьГрафику() : void
+ изменитьЗвук() : void
+ изменитьУправление() : void
+ сброситьНастройки() : void
+ применитьНастройки() : void
+ сохранитьНастройки() : void
СтатусКвеста
Статус
Сохранение
- дата : DateTime
- игрок : Игрок
- мир : ИгровойМир
- скриншот : Texture2D
+ создатьСохранение() : void
+ загрузитьСохранение() : void
+ удалитьСохранение() : void
+ получитьИнформацию() : string
ИгровойМир
- уровни : Уровень[]
- персонажи : Персонаж[]
- квесты : Квест[]
- текущее_время : DateTime
+ загрузитьУровень(уровень: Уровень) : void
+ сохранитьМир() : void
+ загрузитьМир() : void
+ обновитьМир() : void
+ получитьПерсонажа(id: int) : Персонаж
Игрок
- инвентарь : Инвентарь
- текущий_уровень : Уровень
- текущий_квест : Квест
- опыт : int
- уровень_игрока : int
+ переместиться(локация: Локация) : void
+ использоватьПредмет(предмет: Предмет) : void
+ сохранитьИгру() : void
+ загрузитьИгру() : void
+ начатьДиалог(npc: NPC) : void
+ подобратьПредмет(предмет: Предмет) : void
применяется к
управляет
проходит
ВременнаяСистема
- текущаяЭпоха : Эпоха
- переходныеТочки : ТочкаПерехода[]
- парадоксы : int
+ перейтиВЭпоху(эпоха: Эпоха) : void
+ проверитьПарадоксы() : bool
+ добавитьТочкуПерехода(точка: ТочкаПерехода) : void
+ сброситьВремя() : void
Уровень
- уровень_id : int «PK»
- название : string
- локации : Локация[]
- враги : NPC[]
- предметы : Предмет[]
+ загрузитьЛокацию() : void
+ спавнПерсонажей() : void
+ очиститьУровень() : void
+ получитьЛокацию(id: int) : Локация
Инвентарь
- предметы : Предмет[]
- вместимость : int
- деньги : int
+ добавитьПредмет(предмет: Предмет) : bool
+ удалитьПредмет(предмет: Предмет) : bool
+ использоватьПредмет(предмет: Предмет) : void
+ сортировать() : void
+ найтиПредмет(идентификатор: int) : Предмет
+ проверитьВместимость() : bool
выдается за
принадлежит
принадлежит эпохе
Эпоха
Эпоха
ВременнаяСистема
Локация
- локация_id : int «PK»
- название : string
- описание : string
- соседи : Локация[]
- предметы : Предмет[]
- npcs : NPC[]
+ войти(персонаж: Персонаж) : void
+ покинуть(персонаж: Персонаж) : void
+ добавитьПредмет(предмет: Предмет) : void
+ удалитьПредмет(предмет: Предмет) : void
выдается за
связан с
активирует
Босс
- фазы : ФазаБоя[]
- текущаяФаза : int
- уникальные_способности : Способность[]
+ сменитьФазу() : void
+ особыйПрием() : void
+ активироватьСпособность() : void
+ сброситьФазы() : void
NPC
- диалоги : Диалог[]
- отношение : int
- квесты : Квест[]
+ начатьДиалог() : Диалог
+ датьКвест(квест: Квест) : void
+ обновитьОтношение(изменение: int) : void
+ воспроизвестиАнимацию(анимация: string) : void
Предмет
- предмет_id : int «PK»
- название : string
- описание : string
- тип : ТипПредмета
- стоимость : int
+ использовать() : void
+ осмотреть() : string
+ выбросить() : void
+ продать() : void
ФазаБоя
- здоровье_порог : float
- атаки : Атака[]
- поведение : Поведение
+ активировать() : void
+ обновить() : void
Враг
- тип : ТипВрага
- сложность : int
- шаблонПоведения : Поведение
- дроп : Предмет[]
+ атаковать() : void
+ получитьУрон() : void
+ использоватьСпособность() : void
+ убежать() : void
выполняется в
имеет
Квест
- квест_id : int «PK»
- название : string
- описание : string
- статус : СтатусКвеста
- задачи : Задача[]
- награда : Награда
+ начатьКвест() : void
+ проверитьЗадачи() : void
+ завершитьКвест() : void
+ провалитьКвест() : void
+ добавитьЗадачу(задача: Задача) : void
Диалог
- диалог_id : int «PK»
- реплики : Реплика[]
- текущаяРеплика : int
+ начатьДиалог() : Реплика
+ следующаяРеплика(выбор: int) : Реплика
+ завершитьДиалог() : void
+ получитьТекущуюРеплику() : Реплика
оситьДиалог() : void
ТипПредмета
Тип
Оружие
- урон : float
- скорострельность : float
- прочность : int
- тип_урона : ТипУрона
+ атаковать() : void
+ починить() : void
+ улучшить() : void
Артефакт
- сила : float
- заряды : int
- тип_эффекта : ТипЭффекта
+ активировать() : void
+ перезарядить() : void
+ изучить() : string
Персонаж
- персонаж_id : int «PK»
- имя : string
- здоровье : float
- макс_здоровье : float
скорость : float
состояние : СостояниеПерсонажа
+ говорить(текст: string) : void
+ получитьУрон(урон: float) : void
+ исцелить(количество: float) : void
+ взаимодействовать(цель: GameObject) : void
+ переместиться(позиция: Vector3) : void
+ обновитьСостояние() : void
Поведение
Поведение
ТипВрага
Тип
используется в
Задача
- задача_id : int «PK»
- описание : string
- выполнено : bool
- цель : Цель
- награда : Награда
+ проверитьВыполнение() : bool
+ завершить() : void
+ сбросить() : void
Реплика
- текст : string
- вариантыОтвета : ВариантОтвета[]
- эффект : Эффект
- условие : Условие
+ воспроизвести() : void
+ проверитьУсловие() : bool
РитуальнаяСистема
- текущийРитуал : Ритуал
- компоненты : Предмет[]
- эффекты : Эффект[]
+ начатьРитуал() : void
+ проверитьУсловия() : bool
+ добавитьКомпонент(предмет: Предмет) : void
+ завершитьРитуал() : void
СостояниеПерсонажа
Состояние
Эффект
- тип : ТипЭффекта
- значение : float
- длительность : float
+ применить(персонаж: Персонаж) : void
+ снять(персонаж: Персонаж) : void
имеет
ТипЭффекта
Тип

Приложение В
Модели персонажей, карты и меню

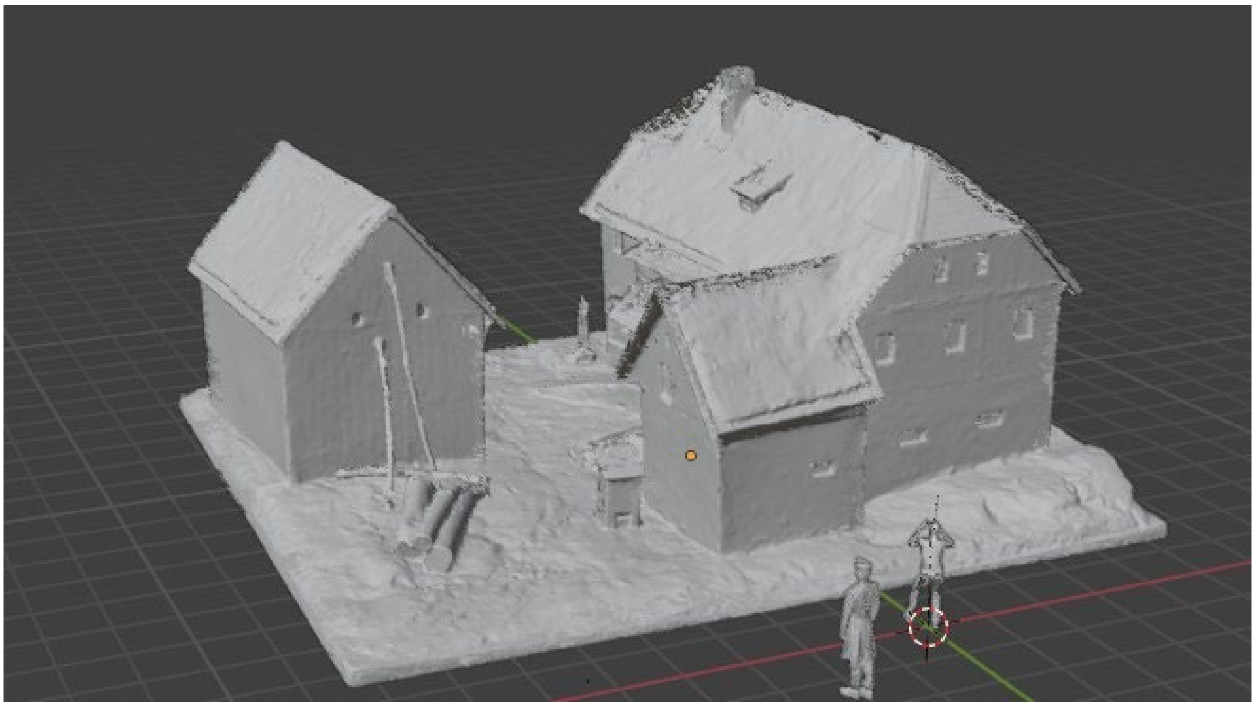
Все модели персонажей(Рисунок 1-11)



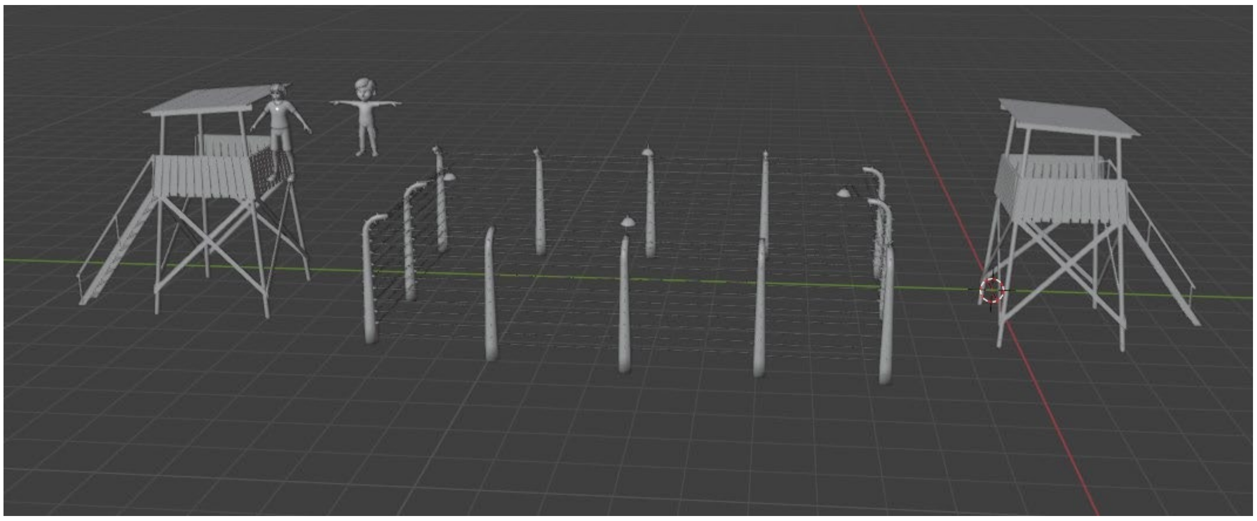





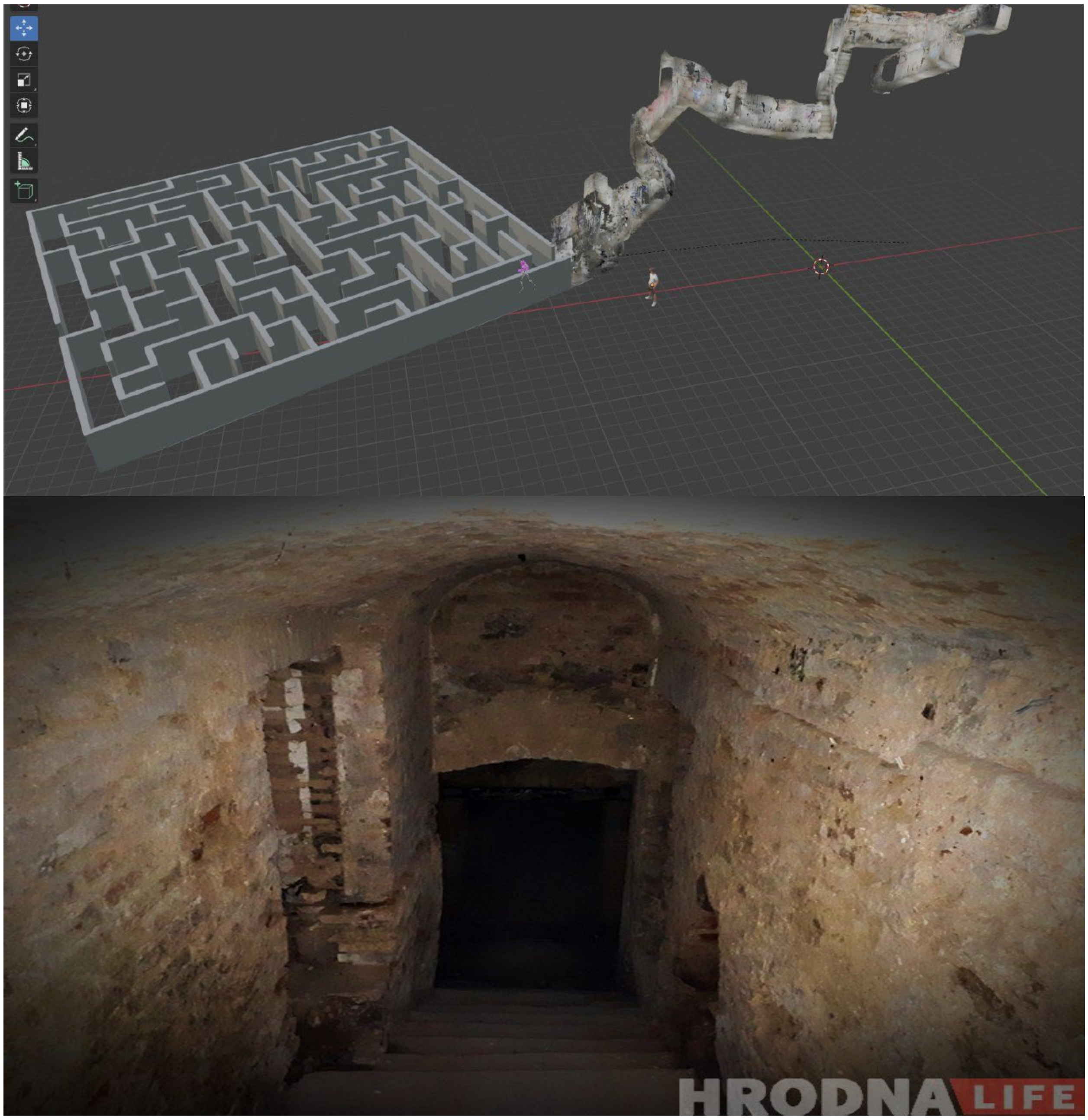
4 уровень – Старый и новый замки.(Рисунок 12)



3 уровень - Усадьба.(Рисунок 13)



2 уровень – Лагерь(Рисунок 14)



1 уровень – лабиринт(Рисунок 15)

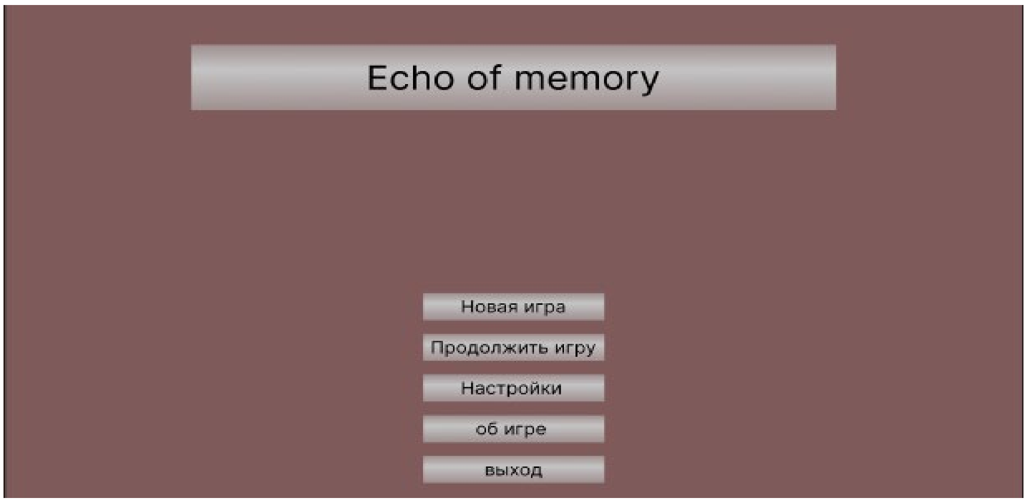


Рисунок 16 – Главное меню

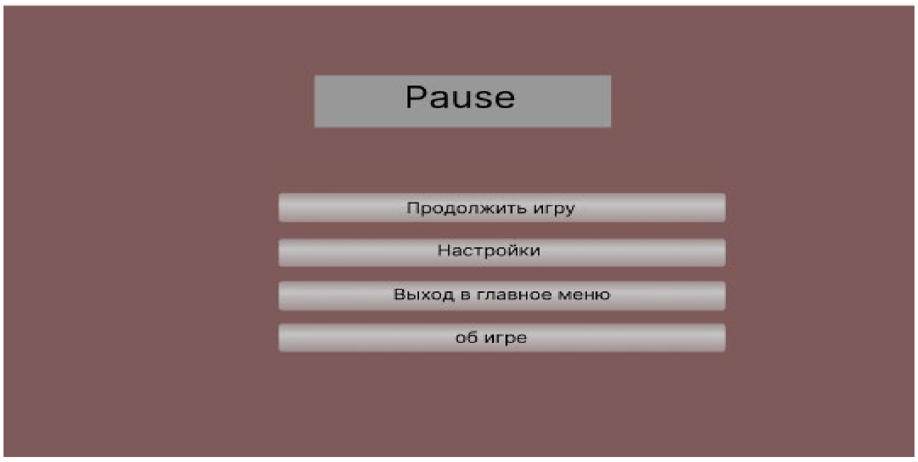


Рисунок 17 – Меню паузы

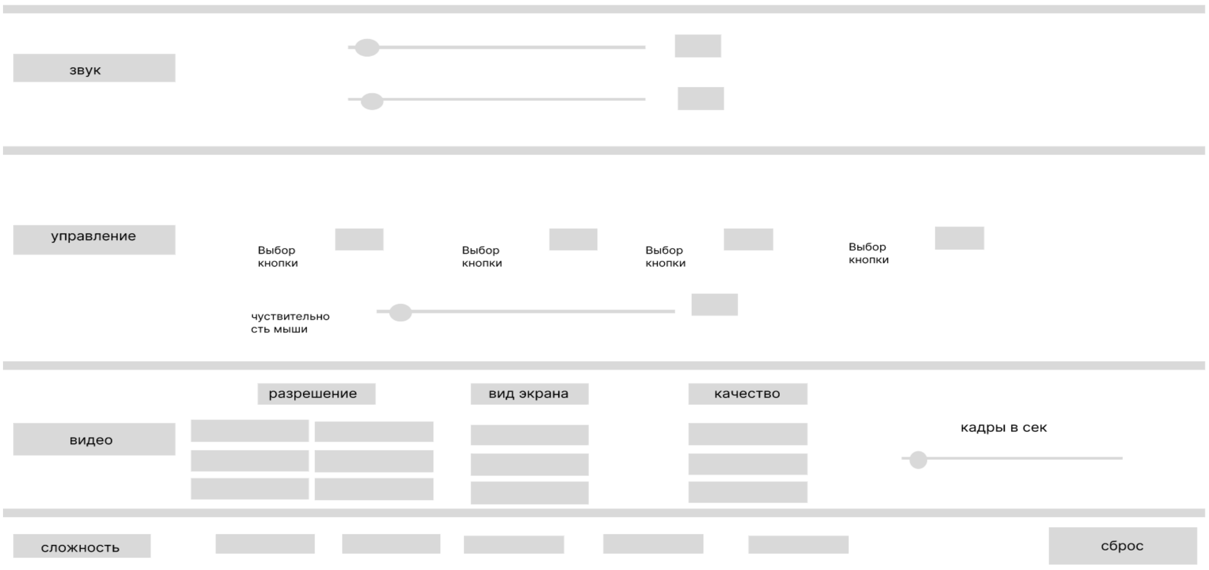


Рисунок 18 – Интерфейс игровой сцены

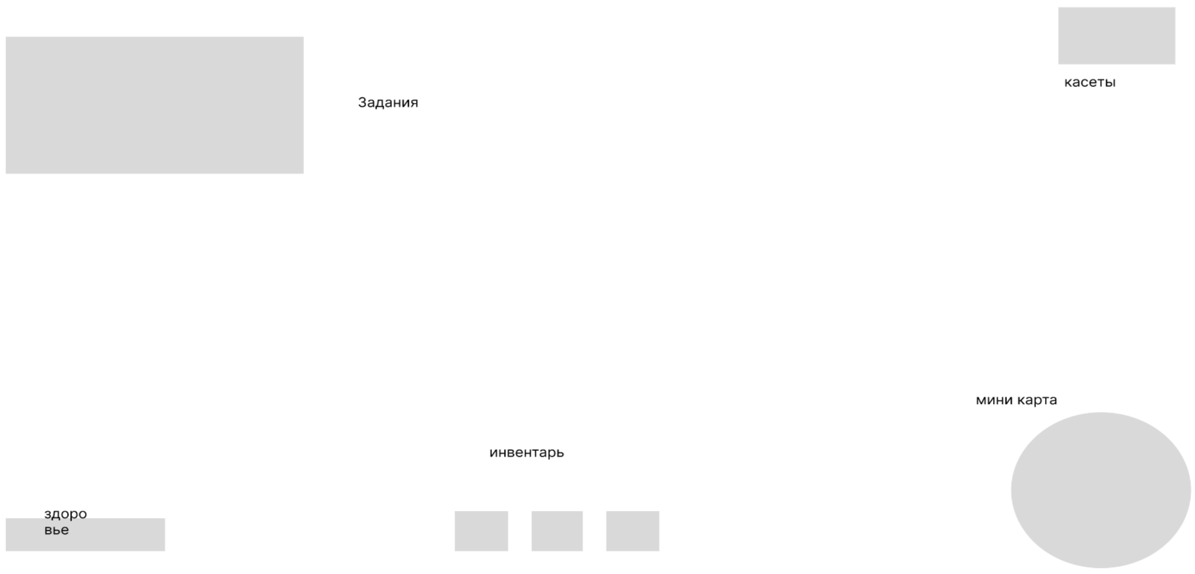


Рисунок 19 – Меню настроек

Ссылка на фигму: *Untitled*

Приложение Г
Листинг

```
using UnityEngine;
using System.Collections.Generic;

public class PlayerController :
MonoBehaviour
{
[Header("Настройки движения")]
public float walkSpeed = 5f;
public float runSpeed = 8f;
public float jumpForce = 1.5f;
public float gravity = -15f;
public float crouchSpeed = 2.5f;
public float crouchHeight = 1f;
private float originalHeight;
private bool isCrouching = false;
private Vector3
originalCameraPosition;
private Vector3 originalCenter;

[Header("Настройки камеры")]
public float mouseSensitivity = 2f;

[Header("Настройки поднятия
предметов")]
public float pickupRange = 5f;
public float throwForce = 10f;
public float rotateSensitivity = 1.5f;
public Vector3 holdOffset = new
Vector3(0, -0.1f, 1f);

[Header("Настройки подсветки")]
public Color outlineColor =
Color.yellow;
public float outlineWidth = 0.05f;

[Header("Здоровье и стамина")]
public float maxHealth = 100f;
public float maxStamina = 100f;

// Компоненты
private CharacterController controller;
private Camera playerCamera;
private Transform cameraTransform;
private Transform holdPosition;

// Переменные камеры
private float cameraPitch = 0f;
private float cameraYaw = 0f;
private bool isCameraLocked = false;

// Переменные движения
private Vector3 velocity;
private bool isGrounded;
private bool isMoving = false;

// Поднятие предметов
private GameObject heldObject;
private Rigidbody heldObjectRb;
private bool isHolding = false;
private float objectRotationX = 0f;
private float objectRotationY = 0f;

// Подсветка предметов
private GameObject highlightedObject;
private List<Material>
originalMaterials = new List<Material>();
private bool isOutlineApplied = false;

// Статистика
private float health;
private float stamina;

// Для отображения UI
private bool showUI = true;
private string interactMessage = "";

void Start()
{
Debug.Log("===
ИНИЦИАЛИЗАЦИЯ ИГРОКА ===");

// CharacterController
controller =
GetComponent<CharacterController>();
if (controller == null)
{
controller =
gameObject.AddComponent<CharacterController
>();
controller.height = 2f;
controller.radius = 0.3f;
controller.center = new Vector3(0,
1f, 0);
}

originalHeight = controller.height;
originalCenter = controller.center;

// Устанавливаем позицию
персонажа как в координатах
transform.position = new
Vector3(0.1711886f, 0.46f, 0.00840497f);
Debug.Log($"Позиция персонажа
установлена: {transform.position}");

// Камера
playerCamera =
GetComponentInChildren<Camera>();
if (playerCamera == null)
{
playerCamera = Camera.main;
if (playerCamera == null)
{
GameObject camObj = new
GameObject("PlayerCamera");

camObj.transform.SetParent(transform);
playerCamera =
camObj.AddComponent<Camera>();

camObj.AddComponent<AudioListener>();
}
else
{

playerCamera.transform.SetParent(transform);
}
}

// Позиционируем камеру

playerCamera.transform.localPosition = new
Vector3(0, 0.5f, 0);
originalCameraPosition =
playerCamera.transform.localPosition;

playerCamera.transform.localRotation =
Quaternion.identity;
cameraTransform =
playerCamera.transform;

// Создаём позицию для
удержания предметов
GameObject holdPos = new
GameObject("HoldPosition");

holdPos.transform.SetParent(cameraTransform);
holdPos.transform.localPosition =
holdOffset;
holdPos.transform.localRotation =
Quaternion.identity;
holdPosition = holdPos.transform;

// Инициализируем значения
health = maxHealth;
stamina = maxStamina;

// Настраиваем курсор
Cursor.lockState =
CursorLockMode.Locked;
Cursor.visible = false;

Debug.Log("Готово!
Управление:");
Debug.Log("WASD - движение");
Debug.Log("Shift - БЕЖАТЬ
(тратит стамину)");
Debug.Log("Ctrl - присесть");
Debug.Log("Space - прыжок
(маленький)");
Debug.Log("E - взять/бросить
предмет");
Debug.Log("ПКМ - вращать
предмет");
Debug.Log("H - тестово нанести
10 урона");
Debug.Log("F1 - скрыть/показать
здоровье и стамину");
```

```
}

void Update()
{
HandleMovement();
HandleCrouch();
HandleStamina();
HandlePickupSystem();
CheckFalling();

if (!isCameraLocked)
{
HandleCamera();
}
else
{
if (Input.GetMouseButton(1))
{
HandleObjectRotation();
}
}

if
(Input.GetKeyDown(KeyCode.H))
{
TakeDamage(10f);
Debug.Log("Нанесён тестовый
урон 10 HP! Здоровье: " + health);
}

if
(Input.GetKeyDown(KeyCode.F1))
{
showUI = !showUI;
Debug.Log("Здоровье и стамина
" + (showUI ? "показаны" : "скрыты"));
}
}

void OnGUI()
{
if (!showUI) return;

GUIStyle healthStyle = new
GUIStyle(GUI.skin.label);
healthStyle.fontSize = 16;
healthStyle.normal.textColor =
Color.red;
healthStyle.fontStyle =
FontStyle.Bold;

GUIStyle staminaStyle = new
GUIStyle(GUI.skin.label);
staminaStyle.fontSize = 16;
staminaStyle.normal.textColor =
Color.green;
staminaStyle.fontStyle =
FontStyle.Bold;

GUIStyle interactStyle = new
GUIStyle(GUI.skin.label);
interactStyle.fontSize = 14;
interactStyle.normal.textColor =
Color.yellow;
interactStyle.alignment =
TextAnchor.MiddleCenter;

GUIStyle crouchStyle = new
GUIStyle(GUI.skin.label);
crouchStyle.fontSize = 12;
crouchStyle.normal.textColor =
Color.cyan;
crouchStyle.fontStyle =
FontStyle.Bold;

GUI.Label(new Rect(10, 10, 300,
25), "❤️ Здоровье: " + Mathf.RoundToInt(health)
+ "/" + maxHealth, healthStyle);

Color staminaColor = stamina < 30f
? Color.red : (stamina < 60f ? Color.yellow :
Color.green);
staminaStyle.normal.textColor =
staminaColor;
GUI.Label(new Rect(10, 35, 300,
25), "⚡ Стамина: " +
Mathf.RoundToInt(stamina) + "/" + maxStamina,
staminaStyle);
```

```
if
(!string.IsNullOrEmpty(interactMessage))
{
GUI.Label(new
Rect(Screen.width / 2 - 200, Screen.height - 80,
400, 30), interactMessage, interactStyle);
}

if (isCrouching)
{
GUI.Label(new Rect(10, 60, 400,
20), "Приседание (отпустите Ctrl чтобы
встать)", crouchStyle);
}
}

void HandleCamera()
{
float mouseX =
Input.GetAxis("Mouse X") * mouseSensitivity;
float mouseY =
Input.GetAxis("Mouse Y") * mouseSensitivity;

cameraYaw += mouseX;
transform.rotation =
Quaternion.Euler(0, cameraYaw, 0);

cameraPitch -= mouseY;
cameraPitch =
Mathf.Clamp(cameraPitch, -90f, 90f);

cameraTransform.localEulerAngles
= new Vector3(cameraPitch, 0f, 0f);
}

void HandleMovement()
{
isGrounded = controller.isGrounded;

if (isGrounded && velocity.y < 0)
{
velocity.y = -2f;
}

// Прыжок
if
(Input.GetKeyDown(KeyCode.Space) &&
isGrounded && stamina > 5f && !isCrouching)
{
velocity.y =
Mathf.Sqrt(jumpForce * -2f * gravity);
stamina -= 5f;
}

// Гравитация
velocity.y += gravity *
Time.deltaTime;

// Проверяем движение
Vector3 move = Vector3.zero;
isMoving = false;

if (Input.GetKey(KeyCode.W)) {
move += transform.forward; isMoving = true; }
if (Input.GetKey(KeyCode.S)) {
move -= transform.forward; isMoving = true; }
if (Input.GetKey(KeyCode.A)) {
move -= transform.right; isMoving = true; }
if (Input.GetKey(KeyCode.D)) {
move += transform.right; isMoving = true; }

if (move.magnitude > 0.1f)
{
move.Normalize();

float speed;
if (isCrouching)
{
speed = crouchSpeed;
}
else
{
// МОЖНО БЕЖАТЬ
ТОЛЬКО ЕСЛИ ЕСТЬ СТАМИНА
bool canRun =
Input.GetKey(KeyCode.LeftShift) && stamina >
0;
speed = canRun ? runSpeed :
walkSpeed;
// Если пытаемся бежать без
стамины - пишем в лог
if
(Input.GetKey(KeyCode.LeftShift) && stamina
<= 0)
{
Debug.Log("Не могу
бежать! Стамина на нуле");
}
}

controller.Move(move * speed *
Time.deltaTime);
}
else
{
isMoving = false;
}

controller.Move(velocity *
Time.deltaTime);
}

void HandleCrouch()
{
bool shouldCrouch =
Input.GetKey(KeyCode.LeftControl);

if (shouldCrouch && !isCrouching)
{
isCrouching = true;

Vector3 currentPosition =
transform.position;

controller.height = crouchHeight;
controller.center = new Vector3(0,
crouchHeight / 2f, 0);

controller.enabled = false;
transform.position =
currentPosition;
controller.enabled = true;

cameraTransform.localPosition =
Vector3.Lerp(
cameraTransform.localPosition,
new Vector3(0, 0.2f, 0),
Time.deltaTime * 10f
);
}
else if (!shouldCrouch &&
isCrouching)
{
if (!CheckCeiling())
{
isCrouching = false;

Vector3 currentPosition =
transform.position;

controller.height =
originalHeight;
controller.center =
originalCenter;

controller.enabled = false;
transform.position =
currentPosition;
controller.enabled = true;

cameraTransform.localPosition
= Vector3.Lerp(

cameraTransform.localPosition,
originalCameraPosition,
Time.deltaTime * 10f
);
}
}

if (isCrouching)
{
cameraTransform.localPosition =
Vector3.Lerp(
cameraTransform.localPosition,
new Vector3(0, 0.2f, 0),
Time.deltaTime * 10f
);
```

```
}
else if (!isCrouching &&
!CheckCeiling())
{
cameraTransform.localPosition =
Vector3.Lerp(
cameraTransform.localPosition,
originalCameraPosition,
Time.deltaTime * 10f
);
}
}

bool CheckCeiling()
{
RaycastHit hit;
float checkDistance = 0.5f;
Vector3 rayStart =
transform.position + Vector3.up *
(controller.height / 2f);

bool hasCeiling =
Physics.Raycast(rayStart, Vector3.up, out hit,
checkDistance);
Debug.DrawRay(rayStart,
Vector3.up * checkDistance, hasCeiling ?
Color.red : Color.green);

return hasCeiling;
}

void HandleStamina()
{
// МЕДЛЕННОЕ восстановление
стамины
if
(!Input.GetKey(KeyCode.LeftShift) && stamina
< maxStamina)
{
float recoverySpeed = 4f; //
Медленно восстанавливаем
stamina += recoverySpeed *
Time.deltaTime;
stamina = Mathf.Min(stamina,
maxStamina);
}

// МЕДЛЕННАЯ трата стамины
при беге
if
(Input.GetKey(KeyCode.LeftShift) && isMoving
&& stamina > 0)
{
float drainSpeed = 15f; //тратим
при беге
stamina -= drainSpeed *
Time.deltaTime;
stamina = Mathf.Max(stamina, 0);
}
}

void HandlePickupSystem()
{
if (!isHolding)
{

FindAndHighlightPickupableObject();

if
(Input.GetKeyDown(KeyCode.E))
{
TryPickup();
}
}
else
{
if (heldObject != null)
{
heldObject.transform.position =
holdPosition.position;

if
(Input.GetMouseButtonDown(1))
{
StartObjectRotation();
}

if
(Input.GetMouseButtonUp(1))
{
```

```
StopObjectRotation();
}

if
(Input.GetKeyDown(KeyCode.E))
{
ThrowObject();
}
}
}
}

void
FindAndHighlightPickupableObject()
{
Ray ray = new
Ray(cameraTransform.position,
cameraTransform.forward);
RaycastHit hit;

if (Physics.Raycast(ray, out hit,
pickupRange))
{
GameObject obj =
hit.collider.gameObject;
Rigidbody rb =
obj.GetComponent<Rigidbody>();

if (rb != null && !rb.isKinematic)
{
if (obj != highlightedObject)
{
ClearHighlight();
highlightedObject = obj;
ApplyOutline(obj);
}

interactMessage = "E - Взять "
+ obj.name + " (расстояние: " +
hit.distance.ToString("F1") + "м)";
Debug.DrawRay(ray.origin,
ray.direction * hit.distance, Color.green);
return;
}
}

ClearHighlight();
interactMessage = "";
Debug.DrawRay(ray.origin,
ray.direction * pickupRange, Color.red);
}

void ApplyOutline(GameObject obj)
{
Renderer[] renderers =
obj.GetComponentsInChildren<Renderer>();
originalMaterials.Clear();

foreach (Renderer renderer in
renderers)
{
List<Material> rendererMaterials
= new List<Material>();
foreach (Material mat in
renderer.materials)
{
rendererMaterials.Add(mat);
}

originalMaterials.AddRange(rendererMaterials);

Material[] outlineMaterials = new
Material[renderer.materials.Length];
for (int i = 0; i <
renderer.materials.Length; i++)
{
Material outlineMat = new
Material(Shader.Find("Standard"));

outlineMat.CopyPropertiesFromMaterial(renderer.
materials[i]);

outlineMat.EnableKeyword("_EMISSION");

outlineMat.SetColor("_EmissionColor",
outlineColor);
outlineMat.SetFloat("_Mode",
3);
```

```
outlineMat.SetInt("_SrcBlend",
(int)UnityEngine.Rendering.BlendMode.SrcAlpha
);
outlineMat.SetInt("_DstBlend",
(int)UnityEngine.Rendering.BlendMode.OneMinu
sSrcAlpha);
outlineMat.SetInt("_ZWrite",
0);

outlineMat.DisableKeyword("_ALPHATEST_ON
");

outlineMat.EnableKeyword("_ALPHABLEND_O
N");

outlineMat.DisableKeyword("_ALPHAPREMUL
TIPLY_ON");
outlineMat.renderQueue =
3000;
outlineMaterials[i] =
outlineMat;
}

renderer.materials =
outlineMaterials;
}

isOutlineApplied = true;
}

void ClearHighlight()
{
if (highlightedObject != null &&
isOutlineApplied)
{
Renderer[] renderers =
highlightedObject.GetComponentsInChildren<Re
nderer>();

int materialIndex = 0;
foreach (Renderer renderer in
renderers)
{
Material[]
originalRendererMaterials = new
Material[renderer.materials.Length];
for (int i = 0; i <
renderer.materials.Length; i++)
{
if (materialIndex <
originalMaterials.Count)
{

originalRendererMaterials[i] =
originalMaterials[materialIndex];
materialIndex++;
}
}
renderer.materials =
originalRendererMaterials;
}

originalMaterials.Clear();
highlightedObject = null;
isOutlineApplied = false;
}
}

void StartObjectRotation()
{
if (isHolding && heldObject !=
null)
{
isCameraLocked = true;
Cursor.lockState =
CursorLockMode.None;
Cursor.visible = true;

objectRotationX = 0f;
objectRotationY = 0f;
interactMessage = "Двигайте
мышью для вращения предмета | E - Бросить";
}
}

void StopObjectRotation()
{
if (isCameraLocked)
```

```
{
isCameraLocked = false;
Cursor.lockState =
CursorLockMode.Locked;
Cursor.visible = false;

if (isHolding)
{
interactMessage = "E -
Бросить | ПКМ - Вращать предмет";
}
}
}

void HandleObjectRotation()
{
if (!isHolding || heldObject == null)
return;

float mouseX =
Input.GetAxis("Mouse X") * rotateSensitivity;
float mouseY =
Input.GetAxis("Mouse Y") * rotateSensitivity;

objectRotationX += mouseX;
objectRotationY += mouseY;

objectRotationY =
Mathf.Clamp(objectRotationY, -90f, 90f);

Quaternion targetRotation =
Quaternion.Euler(objectRotationY, -
objectRotationX, 0f);
heldObject.transform.rotation =
Quaternion.Lerp(heldObject.transform.rotation,
targetRotation, Time.deltaTime * 5f);
}

void TryPickup()
{
Ray ray = new
Ray(cameraTransform.position,
cameraTransform.forward);
RaycastHit hit;

if (Physics.Raycast(ray, out hit,
pickupRange))
{
GameObject obj =
hit.collider.gameObject;
Rigidbody rb =
obj.GetComponent<Rigidbody>();

if (rb == null || rb.isKinematic)
{
Debug.Log("Нельзя поднять
этот предмет");
return;
}

ClearHighlight();

heldObject = obj;
heldObjectRb = rb;
isHolding = true;

heldObjectRb.isKinematic = true;
heldObjectRb.useGravity = false;
heldObjectRb.linearVelocity =
Vector3.zero;
heldObjectRb.angularVelocity =
Vector3.zero;

Collider col =
heldObject.GetComponent<Collider>();
if (col != null) col.enabled = false;

heldObject.layer = 2;


heldObject.transform.SetParent(holdPosition);

heldObject.transform.localPosition =
Vector3.zero;

heldObject.transform.localRotation =
Quaternion.identity;

objectRotationX = 0f;
objectRotationY = 0f;
Debug.Log("Предмет поднят с
расстояния: " + hit.distance.ToString("F1") + "
метров");

interactMessage = "E - Бросить |
ПКМ - Вращать предмет";
}
else
{
Debug.Log("Предмет слишком
далеко! Максимальное расстояние: " +
pickupRange + "м");
}
}

void ThrowObject()
{
if (heldObject == null) return;

if (isCameraLocked)
{
StopObjectRotation();
}

Collider col =
heldObject.GetComponent<Collider>();
if (col != null) col.enabled = true;

heldObject.layer = 0;

heldObjectRb.isKinematic = false;
heldObjectRb.useGravity = true;


heldObject.transform.SetParent(null);

Vector3 throwDirection =
cameraTransform.forward;

heldObjectRb.AddForce(throwDirection *
throwForce, ForceMode.Impulse);

heldObjectRb.angularVelocity =
new Vector3(
Mathf.Deg2Rad *
objectRotationY * 0.2f,
Mathf.Deg2Rad * -
objectRotationX * 0.2f,
0
);

heldObject = null;
heldObjectRb = null;
isHolding = false;
interactMessage = "";
}

void CheckFalling()
{
if (transform.position.y < -20f)
{
controller.enabled = false;
transform.position = new
Vector3(0.1711886f, 0.46f, 0.00840497f);
velocity = Vector3.zero;
controller.enabled = true;
TakeDamage(20f);
}
}

public void TakeDamage(float
damage)
{
health -= damage;
health = Mathf.Max(health, 0f);

if (health <= 0)
{
Respawn();
}
}

void Respawn()
{
controller.enabled = false;
transform.position = new
Vector3(0.1711886f, 0.46f, 0.00840497f);
velocity = Vector3.zero;
controller.enabled = true;
```

```
health = maxHealth;
stamina = maxStamina;
Debug.Log("Игрок возрождён!
Здоровье и стамина восстановлены.");

if (isCrouching)
{
isCrouching = false;
controller.height = originalHeight;
controller.center = originalCenter;
cameraTransform.localPosition =
originalCameraPosition;
}
}

void OnDrawGizmosSelected()
{
if (cameraTransform != null)
{
Gizmos.color = Color.blue;

Gizmos.DrawRay(cameraTransform.position,
cameraTransform.forward * pickupRange);

Gizmos.color = new Color(0, 0, 1,
0.1f);

Gizmos.DrawWireSphere(cameraTransform.positi
on + cameraTransform.forward * (pickupRange /
2f), pickupRange / 2f);

if (holdPosition != null)
{
Gizmos.color = Color.green;

Gizmos.DrawWireSphere(holdPosition.position,
0.1f);
}
}
}
}
```