Horrorek: dziennik budowy

Stan prac i kolejne odcinki budowy gry: co powstało, jak to złożyć w edytorze i jak to sprawdzić.

- Status prac
- Odcinek: Pierwszy pokój i kamera, która wie, gdzie patrzeć
- Odcinek: Klucz, drzwi i pamięć gry
- Odcinek: Drzwi, które tylko udawały, że się otwierają
- Odcinek: Ognisko, sen i druga wersja tego samego pokoju
- Odcinek: Skok i sen, który czasem jest koszmarem
- Odcinek: Trzy pokoje i sen, który idzie za graczem
- Odcinek: Menu, które widzi tylko twórca
- Odcinek: Sen, który za każdym razem jest inny
- Odcinek: Ognisko, które leczy, uczy i pamięta
- Odcinek: Pierwsza krew
- Odcinek: To, co zostaje
- Odcinek: Zahaczenie

Status prac

Horrorek

Psychologiczny horror RPG ze stylizowanym low-poly i kamerą trzecioosobową sterowaną myszą i padem. Silnik: Godot. Projekt łączy naukę, wydanie gry i bazę pod sequel.

Status prac

Zadanie	Zakres	Stan	Gałąź lub PR	Testy	Data
001	Pokój testowy, gracz (ruch względem kamery), kamera za plecami i strefa stałego ujęcia (Phantom Camera; strefy usunięte w 003-D), akcje wejścia	Scalone	PR #3	`test_plugins`, `test_room`	2026-10-27
001b	Brakująca akcja wejścia `dialogic_default_action`	Scalone	PR #5	`test_plugins`	2026-10-27
002	`GameState` (przedmioty, flagi, zapis), interakcje (`Interactable`, `InteractionSensor`, `Pickup`, `Door`), podpowiedź, tłumaczenia PL/EN, `tools/run-tests.ps1`	Scalone	PR #6	`test_state`, `test_interaction`, `test_i18n`, `test_room`	2026-10-27

Zadanie	Zakres	Stan	Gałąź lub PR	Testy	Data
003B, część 1	Stan świata (czas, dzień, rzeczywistość), `RealmManager`, `RealmAware`, `RealmLook`, wersja snu pokoju testowego	Scalone	PR #7	`test_state`, `test_realm`, `test_room`	2026-10-27
003A	Poprawka kolizji drzwi, sterowanie kamerą myszą i padem, status prac, strony poprawek i mechanik, manifest BookStack	Scalone	PR #9	`test_door_passage`, `test_camera_control` i wszystkie poprzednie	2026-10-27
003B, część 2	Punkt medytacji, menu (odpoczynek, sen, przebudzenie), punkty w pokoju testowym	Scalone	PR #8, poprawka scalenia PR #10	`test_meditation`, `test_room`	2026-10-27
003-D	Usunięcie stref kamer, menu pauzy z wyjściem z gry, mysz w menu medytacji (`MouseMode`), strony mechanik (medytacja i sen, menu), `projekt.md`	Scalone	PR #12	`test_pause_menu`, `test_room`, `test_camera_control` i wszystkie poprzednie	2026-10-27
004	Skok gracza (akcja `jump`), koszmar jako losowa odmiana snu (`NightmareRules`, `NightmareDirector`, `Realm.NIGHTMARE`, `nightmare_look`), akt i zakres questu w `GameState` (zapis v3), koszmar w pokoju testowym, `projekt.md` 3.1–3.2	Scalone	PR #13	`test_player_jump`, `test_nightmare` i wszystkie poprzednie	2026-10-28

Zadanie	Zakres	Stan	Gałąź lub PR	Testy	Data
005A	Wiele pokoi: scena główna (`content/main.tscn`) z trwałym graczem, kamerą i interfejsem, `LocationManager` z rejestrem pokoi, spawny, portale, pokój w zapisie (v4), trwałość `Pickup` (`pickup_id`), pokoje robocze A, B, C z questem klucz–drzwi–piwnica	Scalone	PR #14	`test_location`, `test_quest_rooms`, `test_door_passage` (pokój B) i wszystkie poprzednie	2026-10-28
005B	Rzeczywistość i ogniska między pokojami (ognisko pamięta pokój, powrót ze snu ładuje pokój ogniska, zapis v5), wersja snu i koszmaru w pokoju A, zasoby `basic_*`, usunięty `room_test.tscn` (testy przeniesione do `test_main_scene`), odcinek „pokoje”, nagłówek README	Scalone	PR #16	`test_main_scene`, `test_realm_rooms` i wszystkie poprzednie	2026-10-28

Zadanie	Zakres	Stan	Gałąź lub PR	Testy	Data
005C	Menu deweloperskie (F9, tylko w buildach debug): rzeczywistość, wymuszony koszmar i reset zużycia, akt i zakres questu, przejście do pokoju, czas, flagi, przedmioty, nowy stan gry; rozszerzalne przez `add_section` (miejsce na „Sen proceduralny” z 006); wykluczanie z pauzą i medytacją	Scalone	PR #18	`test_dev_menu`, `test_pause_menu` i wszystkie poprzednie	2026-10-28
006	Sny proceduralne: ziarno snu w zapisie (v6), `DreamGenerator` (sloty, palety snu i koszmaru, wykluczenia drogi, punktów stałych i `story_critical`, odnogi z obozami i `DreamSpawnPoint`), pokój „droga” (`room_d`), sekcja „Sen proceduralny” w menu deweloperskim, testy manualne w README, `projekt.md` 3.3–3.5	Scalone	PR #20	`test_dream_generator`, `test_dev_menu` i wszystkie poprzednie	2026-10-28

Zadanie	Zakres	Stan	Gałąź lub PR	Testy	Data
007	Poprawki po 006 (automatyczna ochrona elementów pokoju, ostrzeżenie o zduplikowanym `slot_id`); statystyki (zdrowie, poczytalność, XP, poziom) w zapisie v7; odpoczynek leczy; awans jak w soulslike (`LevelUpOption`, „Awans” przy ognisku, „Postać” w pauzie); `SaveManager` (3 sloty, zapis atomowy, pozycja gracza), „Zapisz” przy ognisku, „Wczytaj” w pauzie; sekcje „Postać” i „Zapis” w menu deweloperskim	Scalone	PR #21	`test_stats`, `test_leveling`, `test_save` i wszystkie poprzednie	2026-10-28

Zadanie	Zakres	Stan	Gałąź lub PR	Testy	Data
008A	Walka: `Hitbox`, `Hurtbox`, `Combatant` (warstwy bez ognia sojuszniczego, okna nietykalności), maszyna stanów gracza (atak lekki i silny, unik, ogłuszenie, śmierć), wytrzymałość z regeneracją, namierzanie (`TargetLock`, kamera namierzania, znacznik, chód bokiem), przeciwnik testowy z AI i nagrodą XP, HUD z paskami, sekcja „Walka” w menu deweloperskim, nowe akcje wejścia; `projekt.md` 3.6–3.7	Scalone	PR #22	`test_combat`, `test_enemy`, `test_target_lock` i wszystkie poprzednie	2026-10-28
007B-poprawka	Esc na ekranie awansu i zapisu nie zamyka już menu pod spodem (zdarzenie oznaczane przed `close()`, bez błędu `null` viewportu); tekst poziomu i XP w HUD (PL/EN); testy manualne 007 w README	Scalone	PR #23	`test_screen_input`, `test_hud` i wszystkie poprzednie	2026-10-28

Zadanie	Zakres	Stan	Gałąź lub PR	Testy	Data
008B	Śmierć gracza (`DeathDirector`, ekran śmierci, powrót do ostatniego ogniska z jawy, snu i koszmaru, `DeathRules`), pozostałość niewydanego XP przypisana do pokoju i rzeczywistości (`Remnant`, `RemnantKeeper`, zapis v8, odzyskanie obniża poczytalność), poczytalność i koszmar na jawie (`SanityRules`, `sanity_broken_changed`, nakładka wyglądu), „Zabij gracza” w menu deweloperskim; poprawka: śmierć w trakcie przejścia (portal, zaśnięcie, obudzenie) czeka na jego koniec, zdrowie dopiero po powrocie, wymuszony powrót po 5 s przy zawieszonym przejściu	Scalone	PR #24	`test_death`, `test_remnant`, `test_sanity` i wszystkie poprzednie	2026-10-28

Zadanie	Zakres	Stan	Gałąź lub PR	Testy	Data
008C	Zahaczenie: wynik trafienia (`HitResult`), lista oczekujących w `Hitbox` (słabsze trafienie po końcu nietykalności w oknie ataku, bez ogłuszenia), spowolnienie i pchnięcie (`MovementModifiers`) bez wpływu na unik i wypad; informacja tekstowa nad celem (`CombatText` , „Zahaczenie!”, liczby obrażeń w buildzie debug, `CombatTextSettings`), przełączniki w menu deweloperskim	W toku	`agent/claude/faza1-zahaczenie`	`test_graze` , `test_combat_text` i wszystkie poprzednie	2026-10-28

Testy manualne

Obszar	Co sprawdzono ręcznie	Wynik	Zadanie
Kamera	sterowanie kamerą myszą	działa	003, 005B
Mysz i menu	mysz w menu pauzy i medytacji, Esc zamyka menu medytacji i nie otwiera pauzy, wyjście z gry	działa	003-D, 005B
Ruch	przejście przez drzwi, skok	działa	003, 004, 005B
Quest	klucz, drzwi, przejście pokojami A, B, C	działa	005A, 005B
Sen	zaśnięcie przy ognisku	działa	003, 005B
Koszmar	wygląd koszmaru (wymuszony menu deweloperskim)	obejrzany	005C
Sen proceduralny	generowanie warstwy losowej w pokoju "droga"	działa	006

Obszar	Co sprawdzono ręcznie	Wynik	Zadanie
Statystyki i awans	odpoczynek przy ognisku leczy, awans przy ognisku, opcje wymagające medytacji wyszarzone poza ogniskiem (ekran "Postać"), opcja dostępna zawsze działa	działa	007
Zapis	zapis przy ognisku, wczytanie z menu pauzy (pokój i pozycja), zapis i wczytanie w śnie w pokoju "droga" (ten sam układ), uszkodzony plik zapisu	działa	007
Testy automatyczne	tools/run-tests.ps1 po 005C	16 z 16 zielonych	005C

Nie sprawdzano ręcznie: pada, menu deweloperskiego poza wymuszeniem koszmaru, Dialogic oraz walki (zadanie 008A: ataki, unik, namierzanie i kamera namierzania, przeciwnik, HUD) oraz śmierci, pozostałości i koszmaru na jawie (008B), zahaczenia i napisów nad celem (008C).

Znane problemy

- Poza pozycjami z tabeli „Testy manualne” gra nie była oglądana w oknie, tylko sprawdzana testami bezgłowymi (np. sen proceduralny w pokoju „droga”). W trybie bezgłowym Godot nie zmienia trybu myszy, więc testy sprawdzają `MouseMode.current`.
- Phantom Camera wypisuje przy starcie ostrzeżenie „Using both Look At and Follow Mode on the same PCam3D has not been fully tested yet” (kamera namierzania). To ostrzeżenie wtyczki, nie błąd.
- Przeciwnicy po śmierci gracza nie odradzają się i nie wracają na miejsca (brak trwałości przeciwników do zadania 009).
- Lewy przycisk myszy to jednocześnie `attack_light` i `dialogic_default_action`; do sprawdzenia przy pierwszym dialogu.
- Elementy snu proceduralnego (`content/dream/props/`) to makiety bez kolizji: gracz przez nie przechodzi.
- Windows: komunikaty WASAPI (GetBufferSize, output_device invalidated) przy zmianie urządzenia audio; bez wpływu, gra nie ma jeszcze dźwięku.
- `dialogic_default_action`: przycisk pada ma `device: 0` (jak w kodzie wtyczki), więc może działać tylko z pierwszym kontrolerem. Do sprawdzenia na padzie.
- Renderer `gl_compatibility` jest wartością tymczasową; decyzja czeka na test na iGPU (zob. `docs/krok-zero-raport.md`).
- `RealmAware` nie obsługuje zagnieżdżania (węzeł `RealmAware` pod innym `RealmAware`).
- Spacja jest jednocześnie skokiem (`jump`) i przewijaniem dialogów (`dialogic_default_action`). Dialogic nie wstrzymuje gry, więc przy rozmowach trzeba będzie blokować skok.

- Szanse koszmaru w `content/realm/nightmare_rules.tres` (0.05, 0.10, 0.20) są zastępcze; `quest_scope` nie jest jeszcze ustawiany przez Questify (wszystko w jednym, pustym zakresie).
- Pokoje A, B i C używają wspólnych zasobów wyglądu `content/realm/basic_*.tres`; docelowo każda lokacja dostanie własne.
- Nie ma pokoju deweloperskiego (poligonu): `room_test.tscn` został usunięty w 005B, „developer room” wróci przy rozbudowie projektu.

Następne kroki

1. Przegląd i scalenie 007.
2. Ręczny test w grze (ułatwia go menu deweloperskie F9): odpoczynek i leczenie, ekran awansu przy ognisku i „Postać” w pauzie, zapis i wczytanie (także po ponownym uruchomieniu gry), pad, pozostałe sekcje menu deweloperskiego.
3. Kolejne zadania: walka (008, korzysta z `Stats.damage/heal`), przeciwnicy w obozach snu (009), pozycje menu pauzy (język, czułość myszy).
4. Test renderowania na iGPU i decyzja o renderer.

Struktura

Folder	Zawartość	Kto zmienia
`core/`	Komponenty wielokrotnego użytku (kamera, interakcje, medytacja i sen, zapis), bez treści fabularnej	Programiści i agenci, przez PR
`content/`	Questy, dialogi, przedmioty, pokoje	Autorzy treści
`addons/`	Wtyczki Godota z przypiętymi wersjami (pojawi się razem z projektem Godota)	Tylko człowiek
`docs/`	Dokumentacja robocza i rejestr zasobów AI	Wszyscy

Zasady pracy

- Gałąź `main` jest chroniona: zmiany wyłącznie przez Pull Request z przeglądem człowieka.
- Agenci AI pracują na osobnych kontach, na gałęziach `agent/*`, bez dostępu do sekretów. PR-y z udziałem AI oznaczamy etykietą `ai`.
- Zmiany w `.gitea/`, `.teamcity/` i `addons/` zawsze wymagają człowieka (chronione wzorce plików).
- Każdy zasób wygenerowany przez AI ma wpis w `docs/ai-ledger.md` przed użyciem w grze.

- Sekretów nigdy nie commitujemy: kluczy Steam, certyfikatów, haseł, tokenów.

Opublikowano z repozytorium Horrorek (commit 87717ad) skryptem publikuj-bookstack.ps1.

Odcinek: Pierwszy pokój i kamera, która wie, gdzie patrzeć

Odcinek: Pierwszy pokój i kamera, która wie, gdzie patrzeć

> Szkic do BookStacka. Faza 1. Poziomy: **[A]** klikane w edytorze, **[B]** gotowy skrypt do wklejenia, **[C]** skrypt z komentarzami dydaktycznymi. Pliki w repozytorium: `core/player/`, `core/camera/`, `content/rooms/room_test.tscn`, `tests/test_room.gd`.
>
> **Zmiana (zadanie 005B):** pokój testowy `content/rooms/room_test.tscn` i test `tests/test_room.gd` zastąpiły pokoje A, B, C (`content/rooms/room_a.tscn`, `room_b.tscn`, `room_c.tscn`, scena główna `content/main.tscn`) i testy `test_main_scene.gd`, `test_realm_rooms.gd`, `test_quest_rooms.gd`. Odcinek opisuje stan z chwili powstania.

> **Zmiana decyzji (27.10.2026, zadanie 003-D).** Zrezygnowaliśmy ze stref kamer stałych. Kamera za plecami zostaje i jest teraz sterowana myszą i prawą gałką (`core/camera/camera_control.gd`, zob. odcinek „Drzwi, które tylko udawały, że się otwierają” i strona „Kamera: sterowanie”). Skrypt `camera_zone.gd` i strefa w rogu pokoju testowego zostały usunięte, więc klocki 1 i 2 poniżej opisują **archiwalny** stan. Pojęcie priorytetu kamery zostaje jako wiedza ogólna o

> Phantom Camera, ale **obecnie nie jest używane w grze**.

Opowieść

Szara kapsuła stoi pośrodku pustego pokoju. Ma jeszcze cztery ściany, podłogę i jedno światło, ale to już jest nasz bohater. Kamera wisi kilka kroków za nim, tak jak w Ocarina of Time. Kiedy kapsuła rusza, kamera idzie za nią.

Teraz podejdź w lewy, daleki róg. Obraz nagle przeskakuje: patrzysz z góry, zza ramienia kogoś, kto stoi w kącie. Nikogo tam nie ma. W horrorze to ujęcie mówi graczowi: *jesteś obserwowany*. Wystarczy zrobić krok w tył, a kamera wraca za plecy, jakby nic się nie stało.

Za tym efektem stoją dwie kamery i jedna liczba: **priorytet**.

Pojęcie: kamera i priorytet

> *Wiedza ogólna, obecnie nieużywana w grze: w scenie jest jedna kamera wirtualna, więc priorytety niczego nie przełączają.*
>
> **W scenie może być wiele „kamer wirtualnych”, ale tylko jedno „oko”.**
>
> - **Oko** to prawdziwa `Camera3D`. Pod nią siedzi `PhantomCameraHost`, który decyduje, czyim okiem ona patrzy.
> - **Kamery wirtualne** (`PhantomCamera3D`) to propozycje ujęć: „za plecami”, „z rogu”, „z sufitu”. Same niczego nie wyświetlają.
> - Każda kamera wirtualna ma **priorytet** (liczba całkowita). Host zawsze wybiera tę z najwyższym priorytetem i płynnie przenosi do niej oko.
>
> W pierwszej wersji pokoju (do zadania 003-D) kamera za plecami miała priorytet **10**. Kamera z rogu miała **0**, dopóki gracz był poza strefą, a po wejściu do strefy **20**. 20 > 10, więc host przełączał ujęcie. Po wyjściu wartość wracała do 0, a 10 znów wygrywało.
>
> Nie trzeba pisać „przełącz kamerę na X”. Wystarczy zmienić liczbę, resztą zajmuje się host.

Klocek 1 [A]: strefa kamery w edytorze (archiwalny)

Strefy kamer zostały usunięte z projektu (zadanie 003-D). Instrukcja zostaje jako przykład pracy z priorytetami.

1. Otwórz `content/rooms/room_test.tscn`.
2. Dodaj węzeł **PhantomCamera3D** (Dodaj węzeł → wyszukaj „PhantomCamera3D”). Nazwij go np. `DoorCamera`. Ustaw go tam, skąd ma patrzeć.
3. W Inspektorze kamery: **Priority = 0**, **Look At Mode = Simple**, **Look At Target** = `Player/CameraTarget`.
4. Dodaj węzeł **Area3D** i nazwij go np. `DoorZone`. Przeciągnij na niego skrypt `res://core/camera/camera_zone.gd`.
5. Do `DoorZone` dodaj dziecko **CollisionShape3D** z kształtem **BoxShape3D**. Rozciągnij pudełko na obszar, w którym ujęcie ma działać.
6. W Inspektorze `DoorZone` w polu **Zone Camera** wskaż `DoorCamera`.
7. Uruchom scenę (F6) i wejdź w strefę.

Pułapka: jeśli ujęcie się nie zmienia, sprawdź, czy **Active Priority** strefy (domyślnie 20) jest większe niż priorytet kamery za plecami (10).

Klocek 2 [B]: skrypt strefy (archiwalny)

Plik `res://core/camera/camera_zone.gd` został usunięty w zadaniu 003-D; kod poniżej to zapis historyczny. Wystarczyło go przypiąć do `Area3D` (klocek 1). W skrócie:

```
class_name CameraZone
extends Area3D

@export var zone_camera: PhantomCamera3D
@export var active_priority: int = 20
@export var inactive_priority: int = 0
@export var player_group: StringName = &"player"
```

```
func _ready() -> void:
	body_entered.connect(_on_body_entered)
	body_exited.connect(_on_body_exited)
	deactivate()


func activate() -> void:
	if zone_camera == null:
		push_warning("CameraZone '%s': brak przypisanej kamery." % name)
		return
	zone_camera.priority = active_priority


func deactivate() -> void:
	if zone_camera == null:
		push_warning("CameraZone '%s': brak przypisanej kamery." % name)
		return
	zone_camera.priority = inactive_priority


func _on_body_entered(body: Node3D) -> void:
	if body.is_in_group(player_group):
		activate()


func _on_body_exited(body: Node3D) -> void:
	if body.is_in_group(player_group):
		deactivate()
```

Klocek 3 [C]: jak gracz wie, gdzie jest „przód”

Wersja z komentarzami dydaktycznymi. Pełny kod jest w `res://core/player/player.gd`.

```
func _get_move_direction() -> Vector3:
	# 1. Pytamy o wejście: x to lewo/prawo, y to przód/tył.
	#    Input.get_vector() działa tak samo dla gałki i dla WASD.
```

```
    #    Gałka wychylona w przód daje y = -1 (taka jest konwencja Godota).
    var input: Vector2 = Input.get_vector("move_left", "move_right", "move_forward", "move_back")
    if input == Vector2.ZERO:
        return Vector3.ZERO

    # 2. Bierzemy kamerę, przez którą gracz TERAZ patrzy.
    #    Nie pytamy, która to kamera wirtualna. To jest zawsze jedno
    #    „oko” (patrz ramka o priorytecie), więc działa też w strefach.
    var view_basis: Basis = global_basis
    var camera: Camera3D = get_viewport().get_camera_3d()
    if camera != null:
        view_basis = camera.global_basis

    # 3. W Godocie kamera patrzy wzdłuż -Z. Zerujemy składową Y, bo kamera
    #    patrzy lekko w dół, a gracz ma chodzić po podłodze, nie w nią.
    var forward: Vector3 = Vector3(-view_basis.z.x, 0.0, -view_basis.z.z).normalized()
    var right: Vector3 = Vector3(view_basis.x.x, 0.0, view_basis.x.z).normalized()

    # 4. Składamy kierunek: prawo razy x, przód razy -y (przypomnienie: przód to y = -1).
    var direction: Vector3 = right * input.x - forward * input.y
    return direction.limit_length(1.0)
```

Do przemyślenia: co się dzieje, gdy gracz trzyma „w przód”, a kamera przeskoczy do ujęcia z rogu? Kierunek zmienia się razem z kamerą. Gry w stylu OoT często „zamrażają” stary kierunek, dopóki gracz nie puści gałki. To temat na kolejny odcinek.

Sprawdź, że działa

Z katalogu głównego projektu:

```
tools\godot\Godot_v4.7.2-stable_win64_console.exe --headless --path . -s
res://tests/test_room.gd
```

Oczekiwane: same linie `OK`, na końcu `test_room: PASS`.

Ciekawostka z kulis

Phantom Camera w trybie Third Person po starcie gry **przenosi kamerę wirtualną pod nowy węzeł** `SpringArm3D`. Dlatego w działającej grze nie znajdziesz jej pod ścieżką `PlayerCamera`. Nasz test zapamiętuje referencje, zanim to nastąpi. Ramię sprężynowe omija kolizje tylko z obiektem z pola `Follow Target`, więc kamera za plecami śledzi samo ciało gracza (z przesunięciem `Follow Offset` na wysokość głowy), a nie marker `CameraTarget`.

Opublikowano z repozytorium Horrorek (commit 9fa3148) skryptem publikuj-bookstack.ps1.

Odcinek: Klucz, drzwi i pamięć gry

Odcinek: Klucz, drzwi i pamięć gry

> Szkic do BookStacka. Faza 1. Poziomy: **[A]** klikane w edytorze, **[B]** gotowy skrypt do wklejenia, **[C]** skrypt z komentarzami dydaktycznymi. Pliki w repozytorium: `core/state/`, `core/items/`, `core/interaction/`, `core/ui/`, `content/items/`, `content/i18n/ui.csv`, `content/rooms/room_test.tscn`, testy `tests/test_state.gd`, `tests/test_interaction.gd`, `tests/test_i18n.gd`.
>
> **Zmiana (zadanie 005B):** pokój testowy `content/rooms/room_test.tscn` i test `tests/test_room.gd` zastąpiły pokoje A, B, C (`content/rooms/room_a.tscn`, `room_b.tscn`, `room_c.tscn`, scena główna `content/main.tscn`) i testy `test_main_scene.gd`, `test_realm_rooms.gd`, `test_quest_rooms.gd`. Odcinek opisuje stan z chwili powstania.

Opowieść

W pokoju testowym coś się zmieniło. We wschodniej ścianie pojawiły się drzwi, a pod południową ścianą leży mały, złoty przedmiot.

Kapsuła podchodzi do drzwi. Na dole ekranu pojawia się napis **„Zamknięte na klucz”**. Naciśnięcie przycisku nic nie daje: klamka nie ustępuje.

Kapsuła wraca, zbliża się do złotego przedmiotu i widzi **„Podnieś”**. Jedno naciśnięcie i przedmiot znika. Gra go jednak nie zapomina: zapisała w swojej pamięci, że gracz ma teraz `key_cellar`.

Przy drzwiach napis brzmi już **„Otwórz”**. Skrzydło obraca się powoli na zawiasie, a gra zapisuje kolejną rzecz: `door_cellar_open = true`. Jeśli kiedyś wczytamy ten zapis, drzwi będą otwarte od

pierwszej klatki.

W tej scenie działają trzy rzeczy: **pamięć gry** (autoload `GameState`), **sygnały**, którymi elementy dają sobie znać o zdarzeniach, i **klucze tłumaczeń**, dzięki którym ten sam napis może być po polsku albo po angielsku.

Pojęcie: sygnał i autoload

> **Autoload to węzeł, który istnieje przez całą grę i jest dostępny z każdego miejsca po nazwie.**
>
> - Zwykłe węzły żyją w scenie. Gdy zmieniasz pokój, scena znika razem z nimi.
> - Autoload Godot tworzy raz, przy starcie gry, i trzyma go nad wszystkimi scenami. Rejestruje się go w `project.godot` (sekcja `[autoload]`) albo w *Project Settings → Globals → Autoload*.
> - U nas autoloadem jest `GameState`: lista przedmiotów i flag. Drzwi w piwnicy i drzwi na strychu pytają tę samą pamięć: `GameState.has_item(&"key_cellar")`.
>
> **Sygnał to ogłoszenie „coś się stało", bez wiedzy, kto słucha.**
>
> - Węzeł *emituje* sygnał (`opened.emit()`), a inne węzły mogą się do niego *podłączyć* (`door.opened.connect(...)`) albo zrobić to w edytorze w zakładce **Węzeł → Sygnały**.
> - Drzwi nie wiedzą, że istnieje dźwięk skrzypienia ani napis na ekranie. Ogłaszają tylko `denied` albo `opened`, a reszta sama decyduje, co z tym zrobić.
> - Dzięki temu klocki są małe i wymienne: podpowiedź na ekranie słucha sygnału `focus_changed` czujnika gracza i nic więcej o grze nie wie.
>
> Zasada kciuka: **autoload pamięta, sygnał ogłasza.**

Klocek 1 [A]: drzwi zamknięte na klucz

1. Otwórz pokój, np. `content/rooms/room_test.tscn`.

2. Przeciągnij do sceny `res://core/interaction/pickup.tscn`. W Inspektorze w polu **Item** wskaż przedmiot z `content/items/` (np. `item_key_cellar.tres`).
3. Przeciągnij do sceny `res://core/interaction/door.tscn` i ustaw drzwi w przejściu.
4. W Inspektorze drzwi:
   - **Required Item**: ten sam przedmiot co w kroku 2,
   - **Open Flag**: nazwa flagi, np. `door_cellar_open` (bez polskich znaków),
   - opcjonalnie **Open Angle Degrees** i **Open Duration**.
5. Jeśli w scenie nie ma jeszcze podpowiedzi, przeciągnij `res://core/ui/interaction_prompt.tscn`.
6. Chcesz dźwięku przy próbie otwarcia zamkniętych drzwi? Zaznacz drzwi, otwórz zakładkę **Węzeł → Sygnały**, kliknij dwukrotnie `denied` i podłącz go do `AudioStreamPlayer3D` → `play`.
7. Uruchom scenę (F6).

Pułapki:

- Brakuje napisu przy drzwiach? Sprawdź, czy gracz ma w scenie węzeł `InteractionSensor` (jest w `core/player/player.tscn`).
- Widzisz klucz zamiast tekstu, np. `PROMPT_OPEN`? Klucza brakuje w `content/i18n/ui.csv` (patrz klocek 2).

Klocek 2 [A]: nowy napis w dwóch językach

1. Otwórz `content/i18n/ui.csv` (arkusz kalkulacyjny lub edytor tekstu, kodowanie UTF-8).
2. Dodaj wiersz: klucz WIELKIMI LITERAMI, tekst polski, tekst angielski, np. `PROMPT_READ,Przeczytaj,Read`
3. Zapisz plik. Godot przy następnym otwarciu (albo `--import`) sam przebuduje pliki `.translation`.
4. W Inspektorze obiektu wpisz klucz w polu **Prompt Key**, nie sam tekst.
5. Uruchom `tools\run-tests.ps1`: test `test_i18n` sprawdzi, czy nowy klucz ma oba tłumaczenia.

Klocek 3 [B]: przedmiot do podniesienia

Gotowy plik: `res://core/interaction/pickup.gd`. Tak wygląda cała podklasa `Interactable`, bo resztę (warstwy fizyki, sygnał `interacted`, podpowiedź) dziedziczy:

```
class_name Pickup
extends Interactable

@export var item: ItemData
@export_range(1, 99) var amount: int = 1


func can_interact(actor: Node3D) -> bool:
	return super(actor) and item != null


func _perform(_actor: Node3D) -> void:
	GameState.add_item(item.id, amount)
	enabled = false
	queue_free()
```

Według tego wzoru zrobisz własny obiekt, np. dźwignię: `extends Interactable`, w `_perform()` ustaw flagę przez `GameState.set_flag(&"lever_cellar_pulled")`.

Klocek 4 [C]: jak czujnik wybiera, z czym rozmawiać

Wersja z komentarzami dydaktycznymi. Pełny kod jest w `res://core/interaction/interaction_sensor.gd`.

```
func refresh_focus() -> void:
	# 1. Szukamy najlepszego kandydata. Na start: nikt i nieskończenie daleko.
	var best: Interactable = null
	var best_distance: float = INF

	# 2. _candidates to obiekty, które fizyka zgłosiła jako "w zasięgu"
	#    (sygnały area_entered / area_exited naszego Area3D).
	for candidate: Interactable in _candidates:
		# Obiekt mógł właśnie zniknąć (podniesiony klucz robi queue_free()).
		if not is_instance_valid(candidate) or candidate.is_queued_for_deletion():
			continue
		# Pytamy obiekt, czy w ogóle chce rozmawiać (np. otwarte drzwi już nie).
```

```
		if not candidate.can_interact(actor):
			continue
		# Porównujemy KWADRAT odległości: wynik jest ten sam, a nie trzeba liczyć pierwiastka.
		var distance: float = global_position.distance_squared_to(candidate.global_position)
		if distance < best_distance:
			best = candidate
			best_distance = distance

	# 3. Sprzątamy listę z obiektów, których już nie ma.
	_candidates = _candidates.filter(func(c: Interactable) -> bool: return is_instance_valid(c))

	# 4. Ogłaszamy zmianę TYLKO wtedy, gdy coś się zmieniło: inny cel albo inna
	#    podpowiedź tego samego celu (drzwi: "Zamknięte na klucz" -> "Otwórz").
	#    Bez tego warunku sygnał leciałby 60 razy na sekundę.
	var prompt: StringName = best.get_prompt_key(actor) if best != null else &""
	if best != current or prompt != _current_prompt:
		current = best
		_current_prompt = prompt
		focus_changed.emit(current)
```

Do przemyślenia: najbliższy nie zawsze znaczy „ten, na który patrzę”. W Ocarina of Time wybór uwzględnia też kierunek, w którym stoi Link. Jak dodać do porównania kąt między przodem gracza a celem?

Sprawdź, że działa

Z katalogu głównego projektu, w PowerShellu:

```
.\tools\run-tests.ps1
```

Oczekiwane: tabela, w której każdy wiersz ma `Wynik = OK`, a na końcu napis `Wszystkie testy zaliczone`. Logi trafiają do `docs\logs\tests\`.

Ciekawostka z kulis

Testy uruchamiamy poleceniem `godot -s plik.gd`. W tym trybie Godot kompiluje plik testu, **zanim** zarejestruje nazwy autoloadów. Gdyby test miał w kodzie typ `Door`, Godot musiałby od razu skompilować `door.gd`, a tam stoi `GameState`, którego kompilator jeszcze nie zna. Dlatego testy

ładują sceny drzwi i przedmiotów przez `load()` już w trakcie działania. W prawdziwej grze ten problem nie występuje, bo sceny ładują się po autoloadach.

Drugi szczegół: przerwana funkcja testu nie dochodzi do swoich sprawdzeń, więc test mógłby wypisać „PASS”, choć część kroków się nie wykonała. Dlatego `test_interaction.gd` liczy sprawdzenia i porównuje je z oczekiwaną liczbą, a `run-tests.ps1` dodatkowo szuka w logach linii `SCRIPT ERROR`.

Opublikowano z repozytorium Horrorek (commit 87717ad) skryptem publikuj-bookstack.ps1.

Odcinek: Drzwi, które tylko udawały, że się otwierają

Odcinek: Drzwi, które tylko udawały, że się otwierają

> Szkic do BookStacka. Faza 1. Poziomy: **[A]** klikane w edytorze, **[B]** gotowy skrypt do wklejenia, **[C]** skrypt z komentarzami dydaktycznymi. Pliki w repozytorium: `core/interaction/door.tscn`, `core/interaction/door.gd`, `core/camera/camera_control.gd`, testy `tests/test_door_passage.gd`, `tests/test_camera_control.gd`.

Opowieść

Klucz do piwnicy jest w kieszeni. Kapsuła podchodzi do drzwi, na ekranie pojawia się **„Otwórz”**, skrzydło obraca się płynnie na zawiasie i staje otworem. Kapsuła rusza naprzód i… zatrzymuje się w progu, jakby stała przed szybą.

Drzwi były otwarte tylko dla oka. Dla fizyki wciąż stały zamknięte: niewidzialna płyta w dokładnie tym miejscu, gdzie przed chwilą było skrzydło. Testy z poprzedniego odcinka tego nie wyłapały, bo sprawdzały kąt obrotu, a nie to, czy da się przejść.

Tym razem zaczęliśmy od testu, który robi to co gracz: prowadzi kapsułę przez otwór. Na starych drzwiach test od razu pokazał błąd: prześwit 0 m, przejście zablokowane na każdym torze. Po poprawce ta sama kapsuła przechodzi na drugą stronę ze środka i z obu krawędzi otworu.

Przy okazji gracz dostał to, czego brakowało od pierwszego pokoju: **może się rozejrzeć**. Ruch myszy albo prawej gałki obraca kamerę wokół kapsuły. Gdy jednak gracz wejdzie w róg pokoju i obraz przejmie kamera stałego ujęcia, obracanie się wyłącza, bo tamto ujęcie ma zostać takie, jakie zaplanował autor.

Pojęcie: kolizja a obrót rodzica

> **W Godocie mesh i kolizja to dwa osobne światy: obraz i fizyka.**
>
> - Zwykłe węzły 3D (Node3D, MeshInstance3D) poruszają się razem z rodzicem. Obracasz rodzica, a obraz dzieci obraca się z nim.
> - Ciało fizyczne (StaticBody3D, AnimatableBody3D, CharacterBody3D) żyje dodatkowo w silniku fizyki. Silnik dowiaduje się o jego ruchu wtedy, gdy porusza się **samo ciało**.
> - `AnimatableBody3D` z `sync_to_physics = true` przenosi do fizyki własny ruch, np. z tweena czy animacji. Ruchu węzła nad nim nie przenosi.
>
> Wniosek: **obracaj ciało, nie jego rodzica.** Jeśli coś ma się obracać wokół zawiasu, niech zawiasem będzie samo ciało, a kształt kolizji i mesh przesuń w jego dzieciach.
>
> I druga lekcja: **test ma robić to, co gracz.** Kąt obrotu był poprawny, a przejście i tak zablokowane.

Klocek 1 [A]: drzwi z prawidłowym zawiasem

1. Otwórz `core/interaction/door.tscn`.
2. Zawiasem jest węzeł **Hinge** typu `AnimatableBody3D`. W Inspektorze ma zaznaczone **Sync To Physics**. Nie odznaczaj tego pola.
3. Pod zawiasem są **CollisionShape3D** (pudełko 1.2 × 2.2 × 0.1) i **Mesh**. Oba są przesunięte o 0.6 m w osi X, czyli o pół szerokości skrzydła.
4. Chcesz drzwi innej szerokości? Zmień rozmiar pudełka i mesha, a przesunięcie dzieci ustaw na połowę nowej szerokości. Węzeł **Hinge** ustaw na krawędzi otworu.
5. Otwór w ścianie musi być co najmniej tak szeroki jak skrzydło. W pokoju testowym ma 1.2 × 2.2 m, a kapsuła gracza ma 0.7 × 1.7 m.

Klocek 2 [B]: kamera sterowana myszą i padem

Gotowy plik: `res://core/camera/camera_control.gd`. Wystarczy go podpiąć w scenie:

1. Dodaj do pokoju węzeł **Node** i nazwij go `CameraControl`.
2. Przypnij skrypt `res://core/camera/camera_control.gd`.
3. W Inspektorze wskaż **Player Camera** (kamera za plecami) i **Host** (`MainCamera3D/PhantomCameraHost`).
4. Dostosuj czułość (**Mouse Sensitivity**, **Stick Sensitivity**), granice pochylenia (**Pitch Min**, **Pitch Max**) i ewentualnie **Invert Y**.

Tak wygląda serce komponentu:

```
func apply_look(delta_degrees: Vector2) -> bool:
	if not is_input_enabled():
		return false
	var pitch_delta: float = -delta_degrees.y if invert_y else delta_degrees.y
	yaw = wrapf(yaw + delta_degrees.x, -180.0, 180.0)
	pitch = clampf(pitch + pitch_delta, pitch_min, pitch_max)
	_apply_to_camera()
	return true
```

W grze: Esc zwalnia mysz, kliknięcie w okno przechwytuje ją z powrotem.

Klocek 3 [C]: test, który chodzi jak gracz

Wersja z komentarzami dydaktycznymi. Pełny kod jest w `res://tests/test_door_passage.gd`.

```
## Prowadzi kapsułę z START_X wzdłuż +X przez WALK_FRAMES klatek fizyki. Zwraca końcowe x.
func _walk(lane: float) -> float:
	# 1. Stawiamy kapsułę przed drzwiami, na wybranym torze (środek albo krawędź otworu).
	_walker.global_position = Vector3(START_X, 0.05, lane)
	_walker.velocity = Vector3.ZERO
	await _physics_frames(1)
	for i: int in WALK_FRAMES:
		# 2. Czekamy na klatkę fizyki: kolizje liczą się w jej rytmie, nie w rytmie obrazu.
		await physics_frame
		var delta: float = 1.0 / Engine.physics_ticks_per_second
		# 3. Idziemy prosto przed siebie, z grawitacją, jak gracz trzymający gałkę w przód.
		_walker.velocity.x = SPEED
```

```
□□_walker.velocity.z = 0.0
□□_walker.velocity.y = 0.0 if _walker.is_on_floor() else _walker.velocity.y - 9.8 * delta
□□# 4. move_and_slide to ten sam ruch co u gracza: zatrzyma się na ścianie albo skrzydle.
□□_walker.move_and_slide()
□□if _walker.global_position.x > PASSED_X + 0.5:
□□□break
□# 5. Wynik to miejsce, w którym kapsuła skończyła: przed drzwiami albo za nimi.
□return _walker.global_position.x
```

Do przemyślenia: test mierzy też prześwit promieniami puszczanymi wzdłuż przejścia. Dlaczego promienie puszczone **w poprzek** otworu nie wykryłyby zamkniętego skrzydła? Podpowiedź: skąd startowałby taki promień?

Sprawdź, że działa

Z katalogu głównego projektu, w PowerShellu:

```
.\tools\run-tests.ps1
```

Oczekiwane: w tabeli każdy wiersz ma `Wynik = OK`, także `test_door_passage` i `test_camera_control`.

Ciekawostka z kulis

Na pierwszej wersji testu kamery wszystkie obroty „nie działały”. Okazało się, że host Phantom Camera wybiera aktywną kamerę dopiero po pierwszej klatce **procesu**, a test czekał tylko na klatki **fizyki**. Do tego czasu `CameraControl` uznawał, że aktywna jest „jakaś inna kamera”, i grzecznie ignorował wejście. Pomogło czekanie na oba rodzaje klatek. Ta pułapka jest opisana na stronie referencyjnej kamery.

Opublikowano z repozytorium Horrorek (commit 87717ad) skryptem publikuj-bookstack.ps1.

Odcinek: Ognisko, sen i druga wersja tego samego pokoju

Odcinek: Ognisko, sen i druga wersja tego samego pokoju

> Szkic do BookStacka. Faza 1. Poziomy: **[A]** klikane w edytorze, **[B]** gotowy skrypt do wklejenia, **[C]** skrypt z komentarzami dydaktycznymi. Pliki w repozytorium: `core/state/game_state.gd`, `core/realm/`, `core/meditation/`, `core/ui/fade_overlay.tscn`, `content/realm/`, `content/rooms/room_test.tscn`, testy `tests/test_state.gd`, `tests/test_realm.gd`, `tests/test_meditation.gd`.
>
> **Zmiana (zadanie 005B):** pokój testowy `content/rooms/room_test.tscn` i test `tests/test_room.gd` zastąpiły pokoje A, B, C (`content/rooms/room_a.tscn`, `room_b.tscn`, `room_c.tscn`, scena główna `content/main.tscn`) i testy `test_main_scene.gd`, `test_realm_rooms.gd`, `test_quest_rooms.gd`. Odcinek opisuje stan z chwili powstania.

Opowieść

W pokoju testowym płoną dwa małe ogniska. Kapsuła podchodzi do pierwszego i na ekranie pojawia się **„Medytuj”**.

Gra zamiera, a na środku ekranu wyskakuje krótka lista: **„Odpocznij < 1 godz. >”**, **„Zaśnij”** i **„Wróć”**. Strzałką w prawo gracz ustawia siedem godzin i wybiera odpoczynek. Zegar w pamięci gry przeskakuje z 20:00 na 3:00, a dzień zmienia się z pierwszego na drugi.

Przy ognisku gracz wybiera **„Zaśnij”**. Ekran gaśnie, a po chwili się rozjaśnia. To ten sam pokój, ale ściany mają kolor zaschniętego wina, w powietrzu wisi mgła, stół zniknął, a na środku stoi czarny monolit z czerwoną poświatą. Gracz nie przeniósł się do innej sceny: zmieniła się **rzeczywistość**.

Drugie ognisko stoi przy południowej ścianie. Tam „Zaśnij” jest wyszarzone i brzmi **„Nie możesz teraz zasnąć”**. Odblokuje się dopiero wtedy, gdy gracz otworzy drzwi do piwnicy kluczem z poprzedniego odcinka.

We śnie menu ogniska ma tylko jedną drogę: **„Obudź się”**. Ekran znów gaśnie i gracz stoi przy ognisku, przy którym zasnął, twarzą do płomienia.

Pojęcie: stan świata i dwie rzeczywistości

> **Stan świata to wszystko, co gra musi pamiętać, żeby po wczytaniu zapisu wyglądać tak samo.**
>
> - U nas mieszka on w jednym miejscu: w autoloadzie `GameState`. Poza przedmiotami i flagami są tam teraz **godzina**, **dzień**, **rzeczywistość** (`waking` albo `dream`) i **ostatni punkt medytacji**.
> - Wszystko inne tylko ten stan **odzwierciedla**. `RealmManager` nie decyduje, czy jest sen. Zapisuje to w `GameState` i ogłasza zmianę sygnałem `realm_changed`.
> - Dzięki temu zapis gry to jeden słownik z `GameState.serialize()`. Po wczytaniu (`deserialize()`) świat sam przełącza się na właściwą rzeczywistość, bez animacji.
>
> **Dwie rzeczywistości to dwie wersje tej samej lokacji, nie dwie sceny.**
>
> - Węzły **bez** komponentu `RealmAware` są wspólne: podłoga, drzwi, ogniska.
> - Węzły **z** `RealmAware` istnieją tylko w wybranych rzeczywistościach. Poza nimi są niewidoczne, nie działają i nie kolidują.
> - Wygląd lokacji (tło, światło, mgła) opisują dwa zasoby `RealmLook`, które `RealmEnvironment` podmienia przy każdej zmianie rzeczywistości.

> Zasada kciuka: **GameState pamięta, RealmManager przełącza, RealmAware i RealmEnvironment się dostosowują.**

Klocek 1 [A]: ognisko w nowej lokacji

1. Przeciągnij do sceny `res://core/meditation/meditation_point.tscn`.
2. W Inspektorze ustaw:
   - **Point Id**: stały identyfikator, np. `meditation_akt1_las` (bez polskich znaków),
   - **Dream Id**: identyfikator snu tej lokacji, np. `akt1_las`,
   - **Dream Available**: odznacz, jeśli przy tym ognisku nie da się zasnąć,
   - **Required Flag**: flaga fabularna, np. `door_cellar_open` (puste = brak warunku).
3. Rozwiń węzeł i przesuń **ReturnMarker** tam, gdzie gracz ma stanąć po przebudzeniu. Niebieska oś markera (−Z) to kierunek, w którym gracz będzie patrzył.
4. Uruchom scenę (F6), podejdź do ogniska i naciśnij przycisk interakcji.

Klocek 2 [A]: wersja snu lokacji

1. Utwórz dwa zasoby **RealmLook** w `content/realm/` (prawy klik → *Nowy zasób* → `RealmLook`), np. `akt1_las_waking.tres` i `akt1_las_dream.tres`. W wersji snu zwykle zmniejsza się **Light Energy** i **Ambient Energy** oraz włącza **Fog Enabled**.
2. Dodaj do sceny węzeł **Node**, przypnij mu skrypt `res://core/realm/realm_environment.gd`, wskaż oba zasoby oraz ścieżki do **WorldEnvironment** i **DirectionalLight3D** lokacji.
3. Obiekt tylko ze snu: dodaj **Node3D**, przypnij skrypt `res://core/realm/realm_aware.gd`, w polu **Realms** zostaw `dream`. Wszystko, co włożysz pod ten węzeł (meshe, kolizje, CSG, obszary), pojawi się tylko we śnie.
4. Obiekt tylko z jawy: to samo, ale w **Realms** wpisz `waking`.
5. Inny wygląd ścian we śnie: zrób dwa zestawy ścian, każdy pod własnym `RealmAware` (`waking` i `dream`), tak jak `WallsWaking` i `WallsDream` w `room_test.tscn`.

Pułapka: nie wkładaj `RealmAware` pod inny `RealmAware`. Zagnieżdżony węzeł jest pomijany przy wyłączaniu kolizji rodzica.

Klocek 3 [B]: własna pozycja w menu ogniska

Menu buduje się z listy pozycji, więc nowy system dopisuje się do niego bez zmian w `meditation_menu.gd`. Przykład: skrypt sceny, który do każdego ogniska dodaje pozycję „Wróć” w innym miejscu listy (w kolejnych zadaniach zastąpią ją leczenie, zapis i awans):

```
extends Node
## Dopisuje pozycję do menu wszystkich ognisk w scenie.

func _ready() -> void:
	for point: Node in get_tree().get_nodes_in_group(&"meditation_points"):
		point.menu_opened.connect(_on_menu_opened)



func _on_menu_opened(menu: MeditationMenu) -> void:
	menu.add_option(
		&"example",                      # id pozycji
		&"MEDITATE_BACK",                # klucz tłumaczenia etykiety
		func() -> bool: return true,     # czy pozycja jest teraz dostępna
		menu.close,                      # co zrobić po wyborze
		{ "realms": [&"waking"] as Array[StringName], "order": 50 }
	)
```

Dodatkowe pola w ostatnim słowniku: `realms` (gdzie pozycja jest widoczna), `locked_label_key` (napis, gdy pozycja jest niedostępna), `on_adjust` (strzałki lewo/prawo), `label_suffix` (tekst doklejany do etykiety) i `order` (kolejność, „Wróć” ma 1000).

Klocek 4 [C]: jak RealmAware „wyłącza” obiekt

Wersja z komentarzami dydaktycznymi. Pełny kod jest w `res://core/realm/realm_aware.gd`.

```
func _disable() -> void:
	_active = false
```

```
	# 1. Ukrywamy węzeł. Dzieci dziedziczą widoczność, więc znikają razem z nim.
	visible = false
	# 2. Wyłączamy przetwarzanie: _process i _physics_process przestają się wywoływać.
	#    Dzieci z process_mode = Inherit też stają.
	process_mode = Node.PROCESS_MODE_DISABLED
	# 3. Pułapka: wyłączone przetwarzanie NIE wyłącza kolizji. Niewidzialna
	#    ściana dalej by blokowała gracza. Dlatego przechodzimy po wszystkich
	#    potomkach z kolizjami i zerujemy je ręcznie.
	for node: Node in _collect_physics_nodes(self):
		if node is CollisionObject3D:
			var body: CollisionObject3D = node
			# Zapamiętujemy oryginały, żeby po powrocie przywrócić je co do bitu.
			var values: Dictionary = {
				"collision_layer": body.collision_layer,
				"collision_mask": body.collision_mask,
			}
			# Warstwa 0 = nikt mnie nie widzi; maska 0 = ja nikogo nie widzę.
			body.collision_layer = 0
			body.collision_mask = 0
			if node is Area3D:
				var area: Area3D = node
				values["monitoring"] = area.monitoring
				values["monitorable"] = area.monitorable
				# set_deferred: Godot nie pozwala zmienić monitoringu w trakcie
				# obsługi sygnału fizyki, więc zmiana czeka do końca klatki.
				area.set_deferred(&"monitoring", false)
				area.set_deferred(&"monitorable", false)
			_saved[node] = values
		elif node is CSGShape3D:
			# CSG ma własny przełącznik kolizji.
			var csg: CSGShape3D = node
			_saved[node] = { "use_collision": csg.use_collision }
			csg.use_collision = false
```

Do przemyślenia: przeciwnik ze snu to po prostu scena przeciwnika pod węzłem `RealmAware` z `realms = [dream]`. Co powinno się stać z przeciwnikiem, który goni gracza w chwili przebudzenia?

Haki na kolejne zadania

- **Pozycje menu (zadania 004–006).** Leczenie, zapis gry i awans dopisują się przez `MeditationPoint.menu_opened(menu)` i `menu.add_option(...)`, jak w klocku 3. Menu ich nie zna i nie trzeba go przerabiać. Pozycja „Wróć” ma `order = 1000`, więc nowe pozycje lądują nad nią.
- **Zapis przy ognisku.** `GameState.serialize()` obejmuje już godzinę, dzień, rzeczywistość i ostatni punkt (`SAVE_VERSION = 2`). Zapisy w wersji 1 wczytują się z wartościami domyślnymi.
- **Przeciwnicy we śnie.** Przeciwnik pod `RealmAware` z `realms = [dream]` na jawie jest niewidoczny, nie przetwarza AI i nie koliduje. Po powrocie do snu wszystko wraca do oryginalnych wartości. Reakcje na samo przejście (np. reset pościgu) można podpiąć do `RealmManager.realm_changing`.
- **Pora dnia.** `GameState.hour_changed` i `day_changed` czekają na systemy zależne od czasu, np. światło lokacji albo zdarzenia o północy.
- **Odwiedzone sny.** `RealmManager.enter_dream` ustawia flagę `dream_<id>_visited`, której questy mogą używać jako warunku.

Sprawdź, że działa

Z katalogu głównego projektu, w PowerShellu:

```
.\tools\run-tests.ps1
```

Oczekiwane: tabela, w której każdy wiersz ma `Wynik = OK` (także `test_realm` i `test_meditation`), a na końcu napis `Wszystkie testy zaliczone`.

Ciekawostka z kulis

Menu medytacji **pauzuje grę** (`get_tree().paused = true`), a mimo to działa, bo ma `process_mode = Always`. To samo dotyczy zaciemnienia i `RealmManager`. Dlatego „Zaśnij” najpierw zamyka menu (wznowienie gry), a dopiero potem zaczyna przejście w sen. Gdyby kolejność była odwrotna, sen zaczynałby się w zamrożonym świecie, a gracz po przebudzeniu stałby w pauzie.

Drugi szczegół: w zapisie godzina jest liczbą z ułamkiem (`float`). Flagi w GameState zamieniają „okrągłe” liczby z JSON na całkowite, ale godzina celowo tę konwersję omija. Bez tego 4:00 wróciłoby z zapisu jako liczba całkowita 4, a 4:30 jako ułamek, czyli raz jednym typem, raz drugim.

Opublikowano z repozytorium Horrorek (commit 87717ad) skryptem publikuj-bookstack.ps1.

Odcinek: Skok i sen, który czasem jest koszmarem

Odcinek: Skok i sen, który czasem jest koszmarem

> Szkic do BookStacka. Faza 1. Poziomy: **[A]** klikane w edytorze, **[B]** gotowy skrypt do wklejenia, **[C]** skrypt z komentarzami dydaktycznymi. Pliki w repozytorium: `core/player/player.gd`, `core/realm/nightmare_rules.gd`, `core/realm/nightmare_director.gd`, `core/realm/realm_manager.gd`, `content/realm/nightmare_rules.tres`, `content/realm/room_test_nightmare.tres`, testy `tests/test_player_jump.gd`, `tests/test_nightmare.gd`.
>
> **Zmiana (zadanie 005B):** pokój testowy `content/rooms/room_test.tscn` i test `tests/test_room.gd` zastąpiły pokoje A, B, C (`content/rooms/room_a.tscn`, `room_b.tscn`, `room_c.tscn`, scena główna `content/main.tscn`) i testy `test_main_scene.gd`, `test_realm_rooms.gd`, `test_quest_rooms.gd`. Odcinek opisuje stan z chwili powstania.

Opowieść

Kapsuła nauczyła się skakać. Spacja albo X na padzie i podrywa się na niecały metr, ląduje, może skoczyć znowu. W powietrzu drugie naciśnięcie nic nie daje, bo tu nikt nie lata.

Potem wraca do ogniska i wybiera **„Zaśnij”**. Ten sam napis co zawsze. Ekran gaśnie, rozjaśnia się… i coś jest nie tak. Mgła jest gęstsza i czerwona. Ściany mają kolor snu, ale światło prawie zgasło. Monolitu, który zwykle stoi we śnie na środku pokoju, nie ma. Zamiast niego pod sufitem wisi coś, co wygląda jak oko.

To był **koszmar**, odmiana snu. Gracz nie wiedział, że go dostanie: menu niczego nie zapowiadało. Ognisko tylko rzuciło kością. W pierwszym akcie szansa jest mała, w późniejszych rośnie. W tym

samym queście koszmar już się nie powtórzy. Następne zaśnięcie da zwykły sen, aż fabuła ruszy dalej.

„Obudź się” działa tak samo jak ze zwykłego snu: gracz staje przy ognisku, przy którym zasnął.

Pojęcie: losowanie z regułą

> **Losowość w grze to zawsze liczba losowa plus reguła, która mówi, kiedy wolno losować.**
>
> - **Szansa** to liczba od 0 do 1. Losujemy liczbę z tego samego zakresu: jeśli wypadła mniejsza niż szansa, zdarzenie zachodzi. Szansa 0 to „nigdy”, szansa 1 to „zawsze”.
> - **Reguła** mówi, czy w ogóle losujemy. U nas: koszmar najwyżej raz na quest. Jeśli w bieżącym zakresie questu już był, nie losujemy, tylko od razu wybieramy zwykły sen.
> - **Pamięć** to miejsce, w którym reguła zapisuje „już było”. U nas `GameState`, więc informacja trafia do zapisu gry i nie znika po wczytaniu.
> - **Ziarno** (seed) sprawia, że generator liczb losowych daje zawsze ten sam ciąg. Gra używa losowego ziarna, a testy stałego, żeby wynik był powtarzalny.
>
> Zasada kciuka: **szansę ustawia autor, regułę i pamięć pilnuje kod, ziarno ustawiają testy.**

Klocek 1 [A]: szansa koszmaru i wygląd koszmaru

1. Otwórz `content/realm/nightmare_rules.tres`. W polu **Chance Per Act** jest lista szans dla kolejnych aktów (pierwsza pozycja to akt 1). Obecne wartości 0.05, 0.10, 0.20 to zastępstwo do ustalenia przez autora. Akt spoza listy używa ostatniej wartości.
2. Zaznacz ognisko (`MeditationPoint`) i w polu **Nightmare Rules** wskaż ten plik. Puste pole oznacza, że przy tym ognisku nigdy nie wypadnie koszmar.
3. Utwórz zasób **RealmLook** dla koszmaru lokacji, np. `content/realm/akt1_las_nightmare.tres` (zwykle mniej światła, czerwieńsza i gęstsza mgła), i wskaż go w polu **Nightmare Look** węzła `RealmEnvironment`. Puste pole oznacza wygląd zwykłego snu.

4. Obiekt tylko z koszmaru: węzeł z `realm_aware.gd` i **Realms** = `nightmare`. Obiekt ze snu, który ma być też w koszmarze: **Realms** = `dream`, `nightmare`. Sam `dream` w koszmarze się nie pokaże.

Klocek 2 [B]: losowanie koszmaru

Gotowy plik: `res://core/realm/nightmare_director.gd`. Cała logika mieści się w jednej funkcji:

```
func roll(rules: NightmareRules) -> bool:
	if rules == null:
		return false
	var scope: StringName = GameState.get_quest_scope()
	if GameState.was_nightmare_used(scope):
		return false
	var chance: float = rules.chance_for_act(GameState.get_act())
	if chance <= 0.0:
		return false
	var hit: bool = rng.randf() < chance
	if hit:
		GameState.mark_nightmare_used(scope)
	return hit
```

Menu medytacji woła ją przy „Zaśnij”, a potem `RealmManager.enter_dream(dream_id, Realm.NIGHTMARE albo Realm.DREAM)`.

Klocek 3 [C]: skok tylko z ziemi

Wersja z komentarzami dydaktycznymi. Pełny kod jest w `res://core/player/player.gd`.

```
func _physics_process(delta: float) -> void:
	# 1. W powietrzu działa grawitacja: prędkość pionowa co klatkę maleje o g · delta.
	if not is_on_floor():
		velocity.y -= _gravity * delta
	# 2. Skok tylko na ziemi. Ponieważ to "elif", w powietrzu ta gałąź się
	#    nie wykona, więc podwójny skok jest niemożliwy bez dodatkowego licznika.
	#    is_action_just_pressed = tylko w klatce naciśnięcia, nie przy trzymaniu.
	elif Input.is_action_just_pressed(&"jump"):
		# 3. Nadajemy prędkość w górę. Wysokość wyniesie ok. v² / (2g):
```

```
□□#    4.5² / (2 · 9.8) ≈ 1.03 m.
□□velocity.y = jump_velocity
□# ... dalej ruch w poziomie i move_and_slide() ...
```

Do przemyślenia: gdy menu wstrzymuje grę, `_physics_process` gracza w ogóle się nie wykonuje, więc Spacja w menu nie wywoła skoku. Co się stanie z Dialogic, który też używa Spacji, a nie wstrzymuje gry?

Sprawdź, że działa

Z katalogu głównego projektu, w PowerShellu:

```
.\tools\run-tests.ps1
```

Oczekiwane: w tabeli każdy wiersz ma `Wynik = OK`, także `test_player_jump` i `test_nightmare`.

Ciekawostka z kulis

Zmierzony skok w teście ma 1.07 m, a wzór daje 1.03 m. To nie błąd: Godot liczy ruch krokami co 1/60 sekundy, a nie w sposób ciągły, więc szczyt wypada trochę wyżej. Dlatego test porównuje wynik ze wzorem z tolerancją ±8 cm.

Druga ciekawostka: w teście pokoju „Zaśnij” musi dawać zwykły sen, ale ogniska mają już reguły koszmaru z szansą 5%. Raz na dwadzieścia uruchomień test by padał. Zamiast wyłączać losowanie, test zaznacza koszmar jako już użyty w bieżącym queście, czyli korzysta z tej samej reguły, którą testuje `test_nightmare`.

Opublikowano z repozytorium Horrorek (commit 87717ad) skryptem publikuj-bookstack.ps1.

Odcinek: Trzy pokoje i sen, który idzie za graczem

Odcinek: Trzy pokoje i sen, który idzie za graczem

> Szkic do BookStacka. Faza 1. Poziomy: **[A]** klikane w edytorze, **[B]** gotowy skrypt do wklejenia, **[C]** skrypt z komentarzami dydaktycznymi. Pliki w repozytorium: `content/main.tscn`, `core/game/game_root.gd`, `core/world/`, `content/world/rooms.tres`, `content/rooms/room_a.tscn`, `room_b.tscn`, `room_c.tscn`, `core/realm/realm_manager.gd`, testy `tests/test_location.gd`, `tests/test_quest_rooms.gd`, `tests/test_main_scene.gd`, `tests/test_realm_rooms.gd`.

Opowieść

Pokój testowy, w którym kapsuła spędziła pierwsze odcinki, zniknął. Na jego miejscu są trzy pokoje robocze.

W pokoju A leży klucz, płonie ognisko, a w północnej ścianie ciemnieje przejście. Kapsuła podnosi klucz i przechodzi: ekran gaśnie na chwilę, a ona stoi już w pokoju B, twarzą do wnętrza. Kamera przyszła razem z nią, bo kamera, gracz i menu nie należą do żadnego pokoju. Mieszkają w scenie głównej, a pokoje się pod nimi wymieniają.

W pokoju B są drzwi do piwnicy. Klucz je otwiera, za nimi jest przedsionek i kolejne przejście, do piwnicy C. Tam stoi drugie ognisko. Kapsuła zasypia przy nim, a gdy ekran się rozjaśnia, piwnica jest ciemniejsza i pełna mgły.

Kapsuła nie czeka. We śnie wraca do pokoju B: on też jest we śnie, bo sen idzie za graczem przez każde przejście. Pokój A we śnie ma inne ściany i czarny monolit w miejscu, gdzie na jawie stał stół.

Gdy przychodzi pora się obudzić, gra pamięta, gdzie gracz zasnął. Choć kapsuła stoi w pokoju A, obudzi się w piwnicy C, przy ognisku, przy którym zasnęła.

Pojęcie: scena główna i pokoje

> **Gra to jedna scena, która trzyma rzeczy trwałe, i wymieniane pod nią pokoje.**
>
> - **Scena główna** (`content/main.tscn`) ma gracza, kamerę, podpowiedź, menu pauzy i zaciemnienie. Te rzeczy nie znikają przy zmianie pokoju.
> - **Pokój** to osobna scena z podłogą, ścianami, światłem, obiektami i spawnami, bez gracza i kamery. `LocationManager` wstawia go pod węzeł `RoomHost`, a stary usuwa.
> - **Stan** (przedmioty, flagi, bieżący pokój, rzeczywistość, ostatnie ognisko) mieszka w `GameState`, więc pokój można zniszczyć i zbudować od nowa, a gra pamięta, że klucz jest zebrany, a drzwi otwarte.
> - **Rzeczywistość** też jest stanem, nie cechą pokoju. Nowy pokój przy starcie pyta `RealmManager`: „jawa, sen czy koszmar?” i ustawia swoje obiekty i wygląd.
>
> Zasada kciuka: **pokój jest wymienny, stan jest trwały.**

Klocek 1 [A]: nowy pokój i przejście do niego

1. Utwórz scenę `content/rooms/room_d.tscn` z korzeniem **Node3D**. Dodaj podłogę i ściany (np. **CSGCombiner3D** z **Use Collision** i **CSGBox3D** w środku), **WorldEnvironment**, **DirectionalLight3D** i węzeł `RealmEnvironment` z zasobami wyglądu (np. `content/realm/basic_*.tres`). Nie dodawaj gracza ani kamery.
2. Dodaj **Marker3D** ze skryptem `res://core/world/spawn_point.gd` i wpisz **Spawn Id**, np. `from_b`. Niebieska oś markera (−Z) to kierunek, w którym gracz będzie patrzył po wejściu.
3. Otwórz `content/world/rooms.tres` i dodaj do **Rooms** nowy **RoomInfo**: **Id** `room_d`, **Scene** nowy pokój, **Name Key** np. `ROOM_D_NAME` (dopisz go do `content/i18n/ui.csv`).
4. W pokoju, z którego ma prowadzić przejście, przeciągnij `res://core/world/portal.tscn` i ustaw **Target Room** `room_d` i **Target Spawn** `from_b`. Opcjonalnie **Required Flag**, jeśli przejście ma być zablokowane do czasu spełnienia warunku.

5. Przedmioty, które mają zostać zebrane na zawsze, dostają **Pickup Id** (np. `pickup_room_d_lamp`). Bez niego odradzają się przy każdym wejściu.

Klocek 2 [B]: powrót ze snu do pokoju ogniska

Gotowy kod w `res://core/realm/realm_manager.gd`. Ta funkcja woła się wewnątrz `return_to_waking()`, gdy ekran jest już zaciemniony:

```
func _load_last_point_room() -> void:
	var room_id: StringName = GameState.get_last_point_room()
	if room_id == &"" or not LocationManager.has_world():
		return
	if room_id != LocationManager.get_current_room():
		LocationManager.load_room_now(room_id)
```

Pokój ogniska zapisuje samo ognisko przy interakcji: `GameState.set_last_point(point_id, LocationManager.get_current_room())`.

Klocek 3 [C]: wymiana pokoju krok po kroku

Wersja z komentarzami dydaktycznymi. Pełny kod jest w `res://core/world/location_manager.gd`.

```
func load_room_now(room_id: StringName, spawn_id: StringName = &"") -> bool:
	# 1. Czy taki pokój jest w rejestrze i czy scena główna się podłączyła?
	if not _can_load(room_id):
		return false
	var info: RoomInfo = _registry.find(room_id)
	var from: StringName = GameState.get_room()
	# 2. Ogłaszamy zmianę, zanim cokolwiek zniknie (np. dla dźwięku albo zapisu).
	room_changing.emit(from, room_id)
	if is_instance_valid(_room):
		# 3. Stary pokój wyjmujemy z drzewa OD RAZU. Samo queue_free() usunęłoby go
		#    dopiero na końcu klatki, a do tego czasu jego spawny i ogniska byłyby
```

```
        #    nadal w grupach i myliłyby się z nowymi.
        _room_host.remove_child(_room)
        _room.queue_free()
    # 4. Nowy pokój: instancja sceny pod RoomHost. Jego _ready() zrobi resztę,
    #    np. RealmAware ustawi się według bieżącej rzeczywistości, a Pickup
    #    z zapisanym zebraniem sam zniknie.
    _room = info.scene.instantiate()
    _room.name = String(room_id)
    _room_host.add_child(_room)
    # 5. Stan: bieżący pokój trafia do GameState, więc też do zapisu gry.
    GameState.set_room(room_id)
    # 6. Gracz staje na spawnie (pozycja i obrót markera, zerowa prędkość).
    _place_player(spawn_id)
    room_changed.emit(room_id)
    return true
```

Do przemyślenia: zapis gry pamięta pokój, ale nie pozycję gracza w pokoju, więc po wczytaniu gracz stoi na pierwszym spawnie. Kiedy to wystarcza, a kiedy trzeba zapisać dokładną pozycję?

Sprawdź, że działa

Z katalogu głównego projektu, w PowerShellu:

```
.\tools\run-tests.ps1
```

Oczekiwane: w tabeli każdy wiersz ma `Wynik = OK`, w tym `test_location`, `test_quest_rooms`, `test_main_scene` i `test_realm_rooms`. W edytorze: F5 uruchamia scenę główną i pokój A.

Ciekawostka z kulis

Usuwając pokój testowy, nie można było po prostu skasować jego testu. Każde sprawdzenie z `test_room.gd` musiało znaleźć nowy dom: okablowanie kamery i pauzy trafiło do `test_main_scene.gd`, klucz i drzwi z podpowiedziami do `test_quest_rooms.gd`, a wersja snu pokoju razem ze ścianami, stołem i monolitem przeszła do pokoju A i jest sprawdzana tak samo jak wcześniej.

Jeden z nowych testów na początku „padał”: po obudzeniu z koszmaru gracz lądował w piwnicy, a test spodziewał się pokoju A. Kod miał rację, bo ostatnim ogniskiem było to w piwnicy. Pomylił się test. To dobry przykład, że czerwony test najpierw się czyta, a dopiero potem poprawia kod.

Opublikowano z repozytorium Horrorek (commit 87717ad) skryptem publikuj-bookstack.ps1.

Odcinek: Menu, które widzi tylko twórca

Odcinek: Menu, które widzi tylko twórca

> Szkic do BookStacka. Faza 1. Poziomy: **[A]** klikane w edytorze, **[B]** gotowy skrypt do wklejenia, **[C]** skrypt z komentarzami dydaktycznymi. Pliki w repozytorium: `core/dev/dev_menu.tscn`, `core/dev/dev_menu.gd`, `core/dev/dev_actions.gd`, `content/main.tscn`, test `tests/test_dev_menu.gd`.

Opowieść

Żeby obejrzeć koszmar w piwnicy, trzeba było dotąd: podnieść klucz w pokoju A, przejść do B, otworzyć drzwi, zejść do piwnicy, zasnąć przy ognisku i liczyć na 5% szansy. Za każdym razem.

Teraz wystarczy nacisnąć **F9**. Gra staje, pojawia się kursor i panel z przyciskami. „Koszmar” i ekran gaśnie, a po chwili jesteśmy w koszmarze w tym pokoju, w którym staliśmy. „Pokój: room_c” i jesteśmy w piwnicy. „door_cellar_open”: przełącznik i drzwi są otwarte, choć nikt ich nie otwierał. „Następne zaśnięcie = koszmar”: ognisko przy najbliższym „Zaśnij” na pewno da koszmar, a gra nie odnotuje, że koszmar w tym queście już był.

Gracz tego menu nigdy nie zobaczy. W wersji wydanej węzeł menu usuwa się sam w pierwszej klatce, jakby go nigdy nie było.

Pojęcie: narzędzie deweloperskie a stan gry

> **Dobre narzędzie deweloperskie nie ma własnych sztuczek. Zmienia stan gry tymi samymi drogami, którymi zmienia go gra.**
>
> - Przycisk „Sen” nie przestawia ręcznie widoczności ścian. Woła `RealmManager.enter_dream()`, czyli to samo, co ognisko. Dlatego to, co widzimy, jest prawdziwym snem, a nie podróbką.
> - Narzędzie może **pominąć warunki** (losowanie koszmaru, flagę fabularną przy ognisku), ale nie **mechanikę** (zaciemnienie, sygnały, zapis w `GameState`).
> - Treść (które flagi, które przedmioty) nie jest wpisana w kod menu. Ustawia ją autor w scenie głównej, w polach `known_flags` i `give_items`.
> - Wydanie jest czyste: `OS.is_debug_build()` mówi, czy to wersja deweloperska. Jeśli nie, menu znika.
>
> Zasada kciuka: **skrót przez warunki tak, skrót przez mechanikę nie.**

Klocek 1 [A]: własna flaga i przedmiot w menu

1. Otwórz `content/main.tscn` i zaznacz węzeł **DevMenu**.
2. W polu **Known Flags** dopisz nazwę flagi, np. `lever_cellar_pulled`. W menu pojawi się przełącznik.
3. W polu **Give Items** dodaj zasób przedmiotu z `content/items/`. W menu pojawi się przycisk „Daj: …”.
4. Uruchom grę (F5) i naciśnij F9.

Klocek 2 [B]: nowa sekcja menu

Dowolny skrypt sceny może dopisać sekcję. Sekcja „Sen proceduralny” ma już zarezerwowane miejsce w kolejności (zadanie 006):

```
func _ready() -> void:
	var dev_menu: Node = get_tree().current_scene.get_node_or_null("DevMenu")
	if dev_menu == null:
		return  # wydanie: menu się usunęło
	dev_menu.add_section(&"procedural_dream", "Sen proceduralny", _build_dream_section)


func _build_dream_section(box: VBoxContainer) -> void:
	var button: Button = Button.new()
	button.text = "Nowe ziarno"
	button.pressed.connect(func() -> void: print("przeładowanie snu"))
	box.add_child(button)
```

Klocek 3 [C]: jak menu znika z wydania

Wersja z komentarzami dydaktycznymi. Pełny kod jest w `res://core/dev/dev_menu.gd`.

```
## Sprawdzenie, czy to build debug. Statyczne pole, żeby testy mogły je podmienić.
static var debug_build_check: Callable = Callable(OS, &"is_debug_build")

## Włącza menu także w wydaniu (tylko testy).
@export var force_enabled: bool = false


func _ready() -> void:
	# 1. Wersja wydana (release): is_debug_build() zwraca false.
	#    Wtedy węzeł usuwa się sam i nie zostawia w grze żadnego śladu:
	#    ani klawisza F9, ani panelu, ani przycisków.
	if not (force_enabled or bool(debug_build_check.call())):
		queue_free()
		return
	# 2. Menu ma działać także wtedy, gdy samo wstrzymuje grę (pauza).
	process_mode = Node.PROCESS_MODE_ALWAYS
	# 3. Panel jest schowany do pierwszego F9.
	_root.visible = false
	_title.text = TEXTS[&"title"]
```

```
_add_default_sections()
```

Do przemyślenia: dlaczego sprawdzenie builda jest w polu `debug_build_check`, a nie bezpośrednio jako `OS.is_debug_build()`? Jak bez tego sprawdzić w teście, że w wydaniu menu znika?

Sprawdź, że działa

Z katalogu głównego projektu, w PowerShellu:

```
.\tools\run-tests.ps1
```

Oczekiwane: w tabeli każdy wiersz ma `Wynik = OK`, w tym `test_dev_menu`. W edytorze: F5, potem F9.

Ciekawostka z kulis

Przełączenie „Sen → Koszmar” wymagało nowej funkcji w `RealmManager`. Bez niej menu musiałoby najpierw obudzić gracza, a obudzenie przenosi go do ostatniego ogniska, czasem do innego pokoju. Narzędzie, które przy okazji teleportuje, myliłoby więcej, niż pomagało. Stąd `switch_dream_kind()`: zmienia odmianę snu z zaciemnieniem, nie ruszając gracza.

Opublikowano z repozytorium Horrorek (commit 87717ad) skryptem publikuj-bookstack.ps1.

Odcinek: Sen, który za każdym razem jest inny

Odcinek: Sen, który za każdym razem jest inny

> Szkic do BookStacka. Faza 1. Poziomy: **[A]** klikane w edytorze, **[B]** gotowy skrypt do wklejenia, **[C]** skrypt z komentarzami dydaktycznymi. Pliki w repozytorium: `core/dream/`, `content/dream/`, `content/rooms/room_d.tscn`, `core/realm/realm_manager.gd`, test `tests/test_dream_generator.gd`.

Opowieść

Z pokoju A prowadzi nowe przejście, na drogę. Na jawie to zwykła, pusta polana: wydeptana droga ze wschodu na zachód, płot, namiot, ognisko i niebieskawy kamień, który ktoś tu postawił nie bez powodu.

Kapsuła zasypia przy ognisku. Kiedy ekran się rozjaśnia, polana jest pełna drzew. Między nimi leży sadzawka, a od drogi odchodzi wąska ścieżka, która skręca raz w lewo, raz w prawo i kończy się małym obozem: namiot, żarzące się palenisko. Ktoś tu był. Albo będzie.

Kapsuła się budzi i zasypia jeszcze raz. Teraz drzewa stoją gdzie indziej, ścieżka skręca w inną stronę, a sadzawek jest dwie. Za to droga, namiot, ognisko i niebieski kamień są dokładnie tam, gdzie zawsze. Sen zmienia to, co do niego należy, i nie rusza tego, co należy do świata.

A jeśli kapsuła we śnie wyjdzie na chwilę do pokoju A i wróci? Drzewa stoją tak samo jak przed wyjściem. To wciąż ten sam sen.

Pojęcie: powtarzalna losowość (ziarno)

> **Komputer nie losuje naprawdę. Liczy kolejne „losowe” liczby ze wzoru, a punktem startowym jest ziarno.**
>
> - To samo ziarno daje zawsze ten sam ciąg liczb, a więc ten sam sen. Inne ziarno daje inny ciąg i inny sen.
> - Przy każdym zaśnięciu gra losuje **nowe ziarno**, więc sen jest niespodzianką. Ziarno trafia do zapisu gry, więc po wczytaniu sen jest ten sam.
> - Każdy slot ma **własny generator**, zasiany z trzech rzeczy: ziarna snu, nazwy pokoju i nazwy slotu. Dzięki temu dodanie nowego slotu nie przesuwa wszystkich drzew w pozostałych.
> - Generator **nigdy** nie używa „globalnej” losowości (`randf()`). Jedna taka linijka i sen po wczytaniu wyglądałby inaczej.
>
> Zasada kciuka: **losowo dla gracza, powtarzalnie dla komputera.**

Klocek 1 [A]: nowy slot snu

1. Otwórz pokój i dodaj **Marker3D** ze skryptem `res://core/dream/dream_slot.gd` (najlepiej pod węzłem `Slots`).
2. Ustaw **Slot Id** (np. `vegetation_west`, bez polskich znaków) i **Slot Type** (Vegetation, Water, Camp albo Branch).
3. Strefa: dodaj dziecko **Area3D** z **CollisionShape3D** (pudełko albo kula) i rozciągnij ją na obszar slotu. Bez strefy liczy się **Radius**.
4. Wskaż **Dream Palette** (np. `content/dream/palette_dream_vegetation.tres`) i opcjonalnie **Nightmare Palette**.
5. Ustaw **Min Items**, **Max Items** i **Min Distance**.
6. Rzeczy, których sen nie może zasłonić: pod namiotem albo ogniskiem dodaj **Marker3D** ze skryptem `res://core/dream/dream_anchor.gd` i ustaw **Radius**; obiekt fabularny dodaj do grupy `story_critical`.
7. Pokój musi mieć węzeł **Node** ze skryptem `res://core/dream/dream_generator.gd`, a w polu **Road Path** wskazaną drogę (`Path3D`).
8. Uruchom grę, zaśnij i naciśnij F9. W sekcji „Sen proceduralny” włącz „Pokaż strefy slotów” i przeładuj sen nowym ziarnem.

Klocek 2 [B]: ziarno slotu

Gotowy kod w `res://core/dream/dream_generator.gd`:

```
static func slot_seed(dream_seed: int, room: StringName, slot_id: StringName) -> int:
	return ("%d|%s|%s" % [dream_seed, room, slot_id]).hash()
```

i jego użycie przy każdym slocie:

```
var rng: RandomNumberGenerator = RandomNumberGenerator.new()
rng.seed = slot_seed(dream_seed, room_key, slot.slot_id)
```

Klocek 3 [C]: czy tu wolno postawić drzewo?

Wersja z komentarzami dydaktycznymi. Pełny kod jest w `res://core/dream/dream_generator.gd`.

```
func _is_free(point: Vector3, element_radius: float, same_slot: Variant, min_distance: float,
		ignore_group: String = "") -> bool:
	# 1. Droga główna: liczymy odległość od jej osi (Path3D). Pas road_clearance
	#    plus promień elementu musi zostać wolny, bo droga należy do warstwy stałej.
	if _road_distance(point) < road_clearance + element_radius:
		return false
	# 2. Punkty stałe (namiot, ognisko): każdy ma własny promień ochrony.
	for anchor: Node in get_tree().get_nodes_in_group(DreamAnchor.GROUP):
		if anchor is DreamAnchor and get_parent().is_ancestor_of(anchor):
			if _flat_distance(point, (anchor as DreamAnchor).global_position) < (anchor as
DreamAnchor).radius + element_radius:
				return false
	# 3. Obiekty fabularne: sen nie może zasłonić niczego, co jest potrzebne w queście.
	for story: Node in get_tree().get_nodes_in_group(STORY_CRITICAL_GROUP):
		if story is Node3D and get_parent().is_ancestor_of(story):
			if _flat_distance(point, (story as Node3D).global_position) < story_clearance +
element_radius:
				return false
	# 4. Inne elementy snu: nie nachodzą na siebie (suma promieni), a w tym samym
```

```
	#    slocie trzymają jeszcze min_distance. Kawałki tej samej odnogi mogą się stykać.
	for other: Dictionary in _occupied:
		if ignore_group != "" and other["group"] == ignore_group:
			continue
		var needed: float = float(other["radius"]) + element_radius
		if other["slot"] == same_slot:
			needed = maxf(needed, min_distance)
		if _flat_distance(point, other["position"]) < needed:
			return false
	return true
```

Do przemyślenia: ta funkcja sprawdza tylko rzeczy, które się nie ruszają. Co by się stało z powtarzalnością, gdyby brała pod uwagę pozycję gracza?

Sprawdź, że działa

Z katalogu głównego projektu, w PowerShellu:

```
.\tools\run-tests.ps1
```

Oczekiwane: w tabeli każdy wiersz ma `Wynik = OK`, w tym `test_dream_generator`. W grze: pokój A, portal na drogę, zaśnięcie przy ognisku, F9 i sekcja „Sen proceduralny”.

Ciekawostka z kulis

W liczniku menu deweloperskiego typy pokazywały się w przypadkowej kolejności: „water, vegetation, branch, camp”. Powód: sortowanie identyfikatorów `StringName` w Godocie nie porównuje tekstu, tylko wewnętrzne adresy. Pomogło sortowanie po zamianie na zwykły tekst. Ta sama pułapka groziłaby generatorowi, dlatego sloty są sortowane po `String(slot_id)`, a nie po samym `StringName`.

Opublikowano z repozytorium Horrorek (commit 87717ad) skryptem publikuj-bookstack.ps1.

Odcinek: Ognisko, które leczy, uczy i pamięta

Odcinek: Ognisko, które leczy, uczy i pamięta

> Szkic do BookStacka. Faza 1. Poziomy: **[A]** klikane w edytorze, **[B]** gotowy skrypt do wklejenia, **[C]** skrypt z komentarzami dydaktycznymi. Pliki w repozytorium: `core/stats/`, `core/save/save_manager.gd`, `core/ui/level_up_screen.tscn`, `core/ui/save_slots_screen.tscn`, `content/stats/`, testy `tests/test_stats.gd`, `tests/test_leveling.gd`, `tests/test_save.gd`.

Opowieść

Kapsuła ma teraz zdrowie i poczytalność. Na razie nic jej nie rani, ale menu deweloperskie potrafi ją „podrapać”: zdrowie spada do 40, poczytalność do 60.

Przy ognisku menu ma trzy nowe pozycje. **„Odpocznij”** przesuwa zegar i przywraca zdrowie oraz poczytalność. **„Zaśnij”** zabiera do snu, ale ran nie leczy: sen to nie odpoczynek. **„Awans”** otwiera ekran z doświadczeniem i opcjami: Wytrzymałość, Wola, Opanowanie. Każdy zakup zjada trochę doświadczenia, podnosi poziom o jeden i powiększa jedną z pul. Następny zakup kosztuje więcej.

W menu pauzy, z dala od ognia, jest pozycja **„Postać”**. Ten sam ekran, ale Wytrzymałość i Wola są wyszarzone z dopiskiem „Tylko przy ognisku”. Opanowanie da się kupić wszędzie.

Ognisko daje jeszcze jedno: **„Zapisz”**. Trzy sloty, każdy z opisem: gdzie, na jakim poziomie, który dzień i która godzina. Po wczytaniu, nawet po wyjściu z gry, kapsuła stoi dokładnie tam, gdzie stała, z tymi samymi ranami i tym samym doświadczeniem.

Pojęcie: stan, który przeżywa wyłączenie gry

> **Zapis gry to zdjęcie stanu, a nie zdjęcie sceny.**
>
> - Gra nie zapisuje węzłów, drzew ani kamer. Zapisuje dane, z których scenę da się odbudować: `GameState` (przedmioty, flagi, czas, sen, statystyki) oraz pokój i pozycję gracza.
> - Dlatego od początku projektu wszystko, co ma przetrwać, mieszka w `GameState`, a statystyki w węźle `Stats` są tylko logiką nad tymi danymi.
> - **Wersja** w pliku pozwala wczytać stare zapisy po zmianach w grze: brakujące pola dostają wartości domyślne. Plik z nowszej wersji jest odrzucany z czytelnym komunikatem.
> - **Atomowość**: nowy zapis powstaje obok starego (plik `.tmp`) i podmienia go dopiero, gdy jest cały. Wyłączony prąd w połowie zapisu nie zniszczy poprzedniego.
>
> Zasada kciuka: **najpierw sprawdź, potem zmień.** Uszkodzony plik nie może zepsuć gry, w którą właśnie grasz.

Klocek 1 [A]: nowa opcja awansu

1. W `content/stats/` utwórz zasób **LevelUpOption** (prawy klik → *Nowy zasób* → `LevelUpOption`), np. `levelup_focus.tres`.
2. Ustaw **Id** (`focus`), **Name Key** (`LEVELUP_FOCUS_NAME`; dopisz go do `content/i18n/ui.csv`), **Target Stat** (np. `stat_sanity.tres`), **Amount**, **Cost Multiplier**.
3. Zaznacz **Requires Meditation**, jeśli opcja ma być dostępna tylko przy ognisku.
4. Otwórz `content/stats/level_up_table.tres` i dopisz opcję do listy **Options**.
5. Uruchom grę, naciśnij F9, w sekcji „Postać” dodaj sobie XP i obejrzyj ekran awansu przy ognisku.

Inne liczby (zdrowie startowe, przyrost na poziom, koszty kolejnych poziomów) zmienia się w `content/stats/stat_*.tres` i `level_curve.tres`, a leczenie przy konkretnym ognisku w polu **Rest Rules** (`MeditationRules`).

Klocek 2 [B]: zapis atomowy

Gotowy kod w `res://core/save/save_manager.gd`:

```
var temp_path: String = path + ".tmp"
var file: FileAccess = FileAccess.open(temp_path, FileAccess.WRITE)
if file == null:
	return _fail(slot, ERROR_WRITE, "%s: %s" % [temp_path,
error_string(FileAccess.get_open_error())])
file.store_string(JSON.stringify(data, "\t"))
file.close()
var error: Error = DirAccess.rename_absolute(temp_path, path)
```

Klocek 3 [C]: zakup awansu

Wersja z komentarzami dydaktycznymi. Pełny kod jest w `res://core/stats/stats.gd`.

```
func level_up(option: LevelUpOption, in_meditation: bool) -> bool:
	# 1. Czy wolno? Przy ognisku (requires_meditation), poniżej najwyższego
	#    poziomu i z wystarczającym doświadczeniem.
	if not can_level_up(option, in_meditation):
		return false
	# 2. Płacimy: koszt z krzywej poziomów razy mnożnik opcji.
	var cost: int = level_up_cost(option)
	set_value(XP, get_xp() - cost)
	# 3. Poziom o jeden w górę. Następny awans będzie droższy.
	GameState.set_stat(LEVEL, get_level() + 1)
	value_changed.emit(LEVEL, float(get_level()))
	# 4. Premia do maksimum statystyki trafia do GameState, więc też do zapisu gry.
	var target: StringName = option.target_stat.id
	GameState.add_stat_bonus(target, option.amount)
	# 5. Pula (zdrowie, poczytalność) rośnie razem z maksimum, żeby awans było czuć od razu.
	if option.target_stat.is_pool:
		change(target, option.amount)
	else:
		value_changed.emit(target, get_value(target))
	leveled_up.emit(get_level())
	return true
```

Do przemyślenia: dlaczego `add_xp()` nie podnosi poziomu sama? Co zmieniłoby to w rytmie gry, w której gracz wraca do ogniska, żeby „wydać” to, co zebrał?

Sprawdź, że działa

Z katalogu głównego projektu, w PowerShellu:

```
.\tools\run-tests.ps1
```

Oczekiwane: w tabeli każdy wiersz ma `Wynik = OK`, w tym `test_stats`, `test_leveling` i `test_save`. W grze: F9, „Postać”, +1000 XP, potem ognisko, „Awans”, „Zapisz”, a na koniec Esc, „Wczytaj”.

Ciekawostka z kulis

Test uszkodzonego zapisu przechodził, a i tak by „padł” w zbiorczym uruchomieniu. Funkcja `JSON.parse_string()` przy złym pliku wypisuje do logu `ERROR: Parse JSON failed`, a skrypt testów słusznie traktuje każdą linię `ERROR:` jako błąd. Rozwiązanie: `JSON.new().parse()`, które zwraca kod błędu (i numer linii do komunikatu), nie krzycząc w logu. Gracz z uszkodzonym plikiem zobaczy spokojne „Plik zapisu jest uszkodzony”, a nie czerwony błąd silnika.

Opublikowano z repozytorium Horrorek (commit 87717ad) skryptem publikuj-bookstack.ps1.

Odcinek: Pierwsza krew

Odcinek: Pierwsza krew

> Szkic do BookStacka. Faza 1. Poziomy: **[A]** klikane w edytorze, **[B]** gotowy skrypt do wklejenia, **[C]** skrypt z komentarzami dydaktycznymi. Pliki w repozytorium: `core/combat/`, `core/ui/hud.tscn`, `content/combat/`, `content/stats/stat_stamina.tres`, testy `tests/test_combat.gd`, `tests/test_enemy.gd`, `tests/test_target_lock.gd`.

Opowieść

Kapsuła pierwszy raz ma czym się bronić. W rogu ekranu pojawiają się trzy paski: czerwony to zdrowie, fioletowy to poczytalność, zielony to wytrzymałość.

F9, „Walka”, „Przywołaj: enemy_test”. Trzy metry przed kapsułą staje czerwonawa postać z mieczem z pudełka. Kiedy kapsuła podejdzie bliżej, postać rusza w jej stronę i unosi miecz. To zapowiedź: jest prawie sekunda, żeby zareagować.

Tab: nad postacią zapala się żółty znacznik, kamera płynnie przesuwa się za plecy kapsuły i patrzy na przeciwnika. WASD nie prowadzi już przed siebie, tylko krąży wokół celu. Shift: odskok w bok, miecz przecina powietrze tam, gdzie przed chwilą stała kapsuła. Lewy przycisk to szybkie cięcie, prawy to wolny, ciężki zamach, który odrzuca przeciwnika. Każde cięcie i każdy unik zjada kawałek zielonego paska; gdy pasek jest pusty, kapsuła stoi bezradnie, aż odsapnie.

Kilka cięć i postać pada. Licznik doświadczenia w menu „Postać” rośnie o 50.

Pojęcie: hitbox, hurtbox i okno czasu

>

> **W walce liczą się dwie strefy i jedno okno.**
>
> - **Hurtbox** to miejsce, w które można dostać (ciało). **Hitbox** to miejsce, które rani (miecz). Trafienie to ich spotkanie.
> - Hitbox jest włączony tylko w **oknie trafienia**: zamach go nie ma, powrót też nie. Dzięki temu gracz widzi zamach i ma czas na unik, a miecz opuszczony po ciosie nikogo nie rani.
> - **Warstwy kolizji** pilnują, kto kogo może trafić: miecz gracza widzi tylko ciała przeciwników, miecz przeciwnika tylko ciało gracza. Nie trzeba pisać żadnego „jeśli to sojusznik, to nie”.
> - **Okno nietykalności** to krótki czas, w którym trafienia się nie liczą: po otrzymanym ciosie (żeby jeden zamach nie zabrał całego zdrowia) i w środku uniku (nagroda za dobre wyczucie czasu).
>
> Zasada kciuka: **liczby walki to dane, nie kod.** Długość zamachu, obrażenia i koszty leżą w zasobach, więc autor może je stroić w Inspektorze bez programisty.

Klocek 1 [A]: nowy atak gracza

1. W `content/combat/` skopiuj `attack_light.tres` (prawy klik → *Duplikuj*), np. jako `attack_quick.tres`.
2. Ustaw **Id**, **Damage**, **Windup** (zamach), **Active** (okno trafienia), **Recovery** (powrót), **Stamina Cost** (0 = bez kosztu), **Reach**, **Width**, **Height** (rozmiar strefy trafienia) i **Lunge Speed** (krok do przodu w zamachu).
3. Otwórz `content/main.tscn`, zaznacz `Player/PlayerCombat` i przeciągnij nowy zasób do pola **Light Attack** albo **Heavy Attack**.
4. F9, „Walka”, „Podgląd kolizji walki”: czerwone pudełko pokazuje strefę ataku.

Tak samo stroi się unik (`dodge.tres`) i przeciwnika (`enemy_test.tres`: zdrowie, prędkość, promień wykrycia, atak, odpoczynek, nagroda XP).

Klocek 2 [B]: hitbox i hurtbox w nowej postaci

Dodaj do sceny postaci (np. nowego przeciwnika) trzy węzły:

```
Enemy (CharacterBody3D, skrypt res://core/combat/enemy.gd)
├── Combatant (Node, combatant.gd, Team = Enemy)
├── Hurtbox (Area3D, hurtbox.gd, Team = Enemy)
│   └── CollisionShape3D (kapsuła na ciele)
└── Hitbox (Area3D, hitbox.gd, Team = Enemy)
    └── CollisionShape3D (pudełko; rozmiar ustawi AttackData)
```

Warstwy kolizji ustawiają się same z pola **Team**. Nie zmieniaj ich ręcznie w Inspektorze.

Klocek 3 [C]: okno trafienia

Wersja z komentarzami dydaktycznymi. Pełny kod jest w `res://core/combat/player_combat.gd`.

```
func _step_attack(delta: float) -> void:
	_time += delta
	# 1. Zamach minął: otwieramy okno trafienia. Hitbox dostaje dane ataku
	#    (obrażenia, odrzut, rozmiar strefy) i zaczyna szukać hurtboxów.
	if attack_phase == AttackPhase.WINDUP and _time >= current_attack.windup:
		attack_phase = AttackPhase.ACTIVE
		hitbox.activate(current_attack)
	# 2. Okno minęło: miecz przestaje ranić, zaczyna się powrót.
	#    Celowo osobne "if", nie "elif": przy dużym delta (zacięcie klatki)
	#    obie fazy mogą minąć w jednym kroku.
	if attack_phase == AttackPhase.ACTIVE and _time >= current_attack.windup +
current_attack.active:
		attack_phase = AttackPhase.RECOVERY
		hitbox.deactivate()
	# 3. Powrót minął: gracz znów chodzi i może zaatakować albo uniknąć.
	if attack_phase == AttackPhase.RECOVERY and _time >= current_attack.total_time():
		_end_attack()
		_set_state(State.MOVE)
```

Do przemyślenia: czemu czas płynie w `step(delta)`, a nie bezpośrednio w `_physics_process`? Co to daje testom, które sprawdzają „0.1 s po kliknięciu jeszcze nie rani, 0.25 s po kliknięciu już tak”?

Sprawdź, że działa

Z katalogu głównego projektu, w PowerShellu:

```
.\tools\run-tests.ps1
```

Oczekiwane: w tabeli każdy wiersz ma `Wynik = OK`, w tym `test_combat`, `test_enemy` i `test_target_lock`. W grze: F9, „Walka”, „Przywołaj”, Tab, lewy i prawy przycisk myszy, Shift.

Ciekawostka z kulis

Pierwsze uruchomienie nowych testów pokazało przeciwnika, który już na starcie unosił miecz, choć gracz stał 20 metrów dalej. Test wyłączał mu `_physics_process`, żeby samemu sterować czasem, ale w trybie `-s` Godot woła `_ready` węzłów dodanych w `_initialize` dopiero w pierwszej klatce i przy okazji z powrotem włącza przetwarzanie. Przeciwnik zdążył „zobaczyć” gracza stojącego w tym samym punkcie i zaatakować, zanim test go ustawił. Rozwiązanie: wyłączać przetwarzanie dopiero po pierwszej klatce.

Opublikowano z repozytorium Horrorek (commit 87717ad) skryptem publikuj-bookstack.ps1.

Odcinek: To, co zostaje

Odcinek: To, co zostaje

> Szkic do BookStacka. Faza 1. Poziomy: **[A]** klikane w edytorze, **[B]** gotowy skrypt do wklejenia, **[C]** skrypt z komentarzami dydaktycznymi. Pliki w repozytorium: `core/combat/death_director.gd`, `core/combat/remnant.tscn`, `core/combat/remnant_keeper.gd`, `core/stats/sanity_rules.gd`, `core/ui/death_screen.tscn`, `content/combat/death_rules.tres`, `content/stats/sanity_rules.tres`, testy `tests/test_death.gd`, `tests/test_remnant.gd`, `tests/test_sanity.gd`.

Opowieść

Kapsuła ma przy sobie 120 punktów doświadczenia i ani jednego wydanego. Przeciwnik testowy trafia raz, drugi, trzeci. Ekran gaśnie, na czarnym tle pojawia się czerwony napis „Nie żyjesz”.

Po chwili kapsuła stoi przy ognisku, cała i zdrowa, z zerem na liczniku doświadczenia. Tam, gdzie upadła, unosi się niebieskawa kula: „Odzyskaj pozostałość”. Doświadczenie wraca, ale coś w środku pęka. Poczytalność spada o 20.

Jeszcze jedna taka wyprawa i pasek poczytalności zjeżdża poniżej połowy. Świat się nie zmienia: te same ściany, to samo ognisko. Tylko światło robi się chore, tło ciemnieje, mgła gęstnieje. Kapsuła nie śpi. To jawa wygląda jak koszmar. Odpoczynek przy ognisku przywraca poczytalność i kolory.

A gdyby cios dosięgnął kapsułę w progu, gdy już przechodziła schodami do innego pokoju albo zasypiała przy ognisku? Śmierć poczeka, aż przejście się dokończy, i dopiero wtedy zabierze ją do ogniska. Kapsuła nigdy nie wstaje tam, gdzie upadła.

Gdyby kapsuła zginęła we śnie, obudzi się na jawie przy ognisku, a jej pozostałość zostanie we śnie. Odzyska ją tylko wtedy, gdy znów zaśnie (zwykłym snem, nie koszmarem) i odnajdzie to miejsce.

Pojęcie: stan zapisywany i stan liczony

> **Nie wszystko, co gra „pamięta”, trzeba zapisywać.**
>
> - **Pozostałość** to fakt z przeszłości: gdzie, kiedy i ile. Nie da się jej odtworzyć z innych danych, więc trafia do `GameState` i do zapisu gry.
> - **Koszmar na jawie** to wniosek z bieżącej poczytalności. Gdyby go zapisać osobno, mógłby się rozjechać z paskiem poczytalności (np. po zmianie progu przez autora). Dlatego `Stats` liczy go za każdym razem od nowa.
> - **Histereza** to dwa progi zamiast jednego: inny do wejścia, inny do wyjścia. Bez niej wygląd świata migotałby, gdy poczytalność drga wokół 50%.
>
> Zasada kciuka: **zapisuj fakty, wyliczaj wnioski.**

Klocek 1 [A]: strojenie śmierci i poczytalności

1. Otwórz `content/combat/death_rules.tres`. Ustaw, ile XP zostaje w pozostałości (**Remnant Xp Fraction**), ile zdrowia i poczytalności wraca po śmierci, ile poczytalności kosztuje odzyskanie (**Recovery Sanity Cost**), jak długo widać ekran śmierci.
2. Otwórz `content/stats/sanity_rules.tres`. **Break Fraction** to próg wejścia w koszmar na jawie, **Recover Fraction** to próg wyjścia (histereza: ustaw np. 0.6).
3. Wygląd koszmaru na jawie to pole **Nightmare Look** w węźle `RealmEnvironment` pokoju; ten sam, którego używa koszmar we śnie.
4. Sprawdź w grze: F9, „Postać”, ustaw poczytalność na 30.

Klocek 2 [B]: reakcja na pęknięcie poczytalności

Gotowy skrypt do węzła, który ma reagować na koszmar na jawie (np. dźwięk tła albo przeciwnicy w zadaniu 009):

```
extends Node

func _ready() -> void:
	var stats: Stats = Stats.find(get_tree())
	if stats != null:
		stats.sanity_broken_changed.connect(_on_sanity_broken_changed)
		_on_sanity_broken_changed(stats.is_sanity_broken())


func _on_sanity_broken_changed(broken: bool) -> void:
	print("Koszmar na jawie: ", broken)
```

Klocek 3 [C]: pozostałość przy śmierci

Wersja z komentarzami dydaktycznymi. Pełny kod jest w `res://core/combat/death_director.gd`.

```
func leave_remnant() -> int:
	# 1. Każda śmierć unieważnia poprzednią pozostałość: druga szansa jest jedna.
	GameState.clear_remnant()
	var stats: Stats = Stats.find(get_tree())
	if stats == null:
		return 0
	# 2. Do pozostałości trafia ułamek niewydanego XP; reszta przepada.
	var fraction: float = rules.remnant_xp_fraction if rules != null else 1.0
	var amount: int = floori(stats.get_xp() * fraction)
	# 3. Gracz traci całe niewydane XP, także tę część, która nie trafiła do pozostałości.
	stats.set_value(Stats.XP, 0)
	# 4. Zapamiętujemy fakt: ile, w której rzeczywistości, w którym pokoju i gdzie.
	#    RemnantKeeper pokaże kulę tylko tam, gdzie zgadza się pokój i rzeczywistość.
	if amount > 0 and player != null:
		GameState.set_remnant(amount, GameState.get_realm(), LocationManager.get_current_room(),
player.global_position)
	return amount
```

Do przemyślenia: dlaczego pozostałość zapisujemy **przed** powrotem do ogniska, a nie po nim? Co by się stało z rzeczywistością i pokojem pozostałości, gdyby kolejność była odwrotna?

Sprawdź, że działa

Z katalogu głównego projektu, w PowerShellu:

```
.\tools\run-tests.ps1
```

Oczekiwane: w tabeli każdy wiersz ma `Wynik = OK`, w tym `test_death`, `test_remnant` i `test_sanity`. W grze: F9, „Postać”, +100 XP, potem „Walka”, „Zabij gracza”, wróć w miejsce śmierci i odzyskaj pozostałość.

Ciekawostka z kulis

Pierwsza wersja testów czekała na sygnał `respawned` (`await director.respawned`). Osobno test przechodził, ale w zbiorczym uruchomieniu menu deweloperskie zawiesiło się na tej linii: ekran śmierci trwał w teście 0.05 s, a zanim test doszedł do `await`, śmierć zdążyła się skończyć i sygnał już wybrzmiał. Na sygnał, który już był, czeka się w nieskończoność. Rozwiązanie: test odpytuje `DeathDirector.is_busy()` co klatkę, z limitem czasu. Zasada: **na sygnał czekaj tylko wtedy, gdy masz pewność, że jeszcze nie przyszedł.**

Druga pułapka wyszła przy przeglądzie: śmierć w trakcie przejścia portalem. `RealmManager.respawn_now` w trakcie przejścia odmawia i zwraca `false`, ale `DeathDirector` tego nie sprawdzał i od razu przywracał zdrowie. Kapsuła wstawała w miejscu, a chwilę później przejście kończyło się w pokoju docelowym portalu. Do tego śmierć i portal dzielą jedną nakładkę zaciemnienia. Nowe zaciemnienie przerywało stare, a przerwany tween nie wysyła `finished`, więc portal czekał na nie w nieskończoność. Teraz `DeathDirector` czeka na koniec przejścia, ponawia powrót do skutku i dopiero wtedy leczy, a `FadeOverlay` kończy przerwane czekanie. A gdyby przejście zawisło na dobre? Po 5 s `DeathDirector` zgłasza błąd i wymusza powrót. Czarny ekran bez wyjścia jest gorszy niż zgubione przejście. Zasada: **gdy funkcja może odmówić, sprawdź jej wynik, zanim zrobisz następny krok.** Opis: strona poprawki „Śmierć w trakcie przejścia”.

Opublikowano z repozytorium Horrorek (commit 87717ad) skryptem publikuj-bookstack.ps1.

Odcinek: Zahaczenie

Odcinek: Zahaczenie

> Szkic do BookStacka. Faza 1. Poziomy: **[A]** klikane w edytorze, **[B]** gotowy skrypt do wklejenia, **[C]** skrypt z komentarzami dydaktycznymi. Pliki w repozytorium: `core/combat/hitbox.gd`, `core/combat/combatant.gd`, `core/combat/movement_modifiers.gd`, `core/ui/combat_text.gd`, `core/ui/combat_text_settings.gd`, `content/combat/attack_*.tres`, `content/combat/combat_text_settings.tres`, testy `tests/test_graze.gd`, `tests/test_combat_text.gd`.

Opowieść

Przeciwnik testowy unosi miecz. Kapsuła odskakuje w bok o ułamek sekundy za wcześnie: ostrze spada, gdy jest jeszcze nietykalna, i przechodzi przez nią bez śladu. Ale unik się kończy, a ostrze wciąż tnie powietrze, w którym kapsuła nadal stoi.

Nad jej głową pojawia się pomarańczowy napis „Zahaczenie!”, a pod nim mała liczba: 6. Nie 15, jak przy pełnym ciosie. Kapsuła nie zatacza się i nie traci kontroli, tylko odsuwa się trochę od napastnika i przez chwilę idzie ciężej, jakby brodziła w wodzie. Po pół sekundy znów biegnie normalnie.

Tak wygląda kłucie piką podczas ucieczki: nie zabija, ale przypomina, że zbyt wczesny unik nie jest w pełni bezpieczny. Gdyby kapsuła odskoczyła dalej, poza zasięg ostrza, nic by jej nie dosięgło.

Pojęcie: nietykalność i zahaczenie

> **Nietykalność to okno czasu, nie tarcza na cały atak.**

> - **Nietykalność** (unik, chwila po trafieniu) sprawia, że trafienie w tym oknie jest **odrzucone**, a nie zignorowane. `Combatant.apply_damage` mówi, co się stało: `HIT`, `INVULNERABLE` albo `IGNORED` (martwy, brak danych).
> - **Okno ataku** (`AttackData.active`) to czas, przez który strefa trafienia jest groźna.
> - **Zahaczenie** to spotkanie obu okien. Strefa trafiła cel nietykalny, nietykalność skończyła się wcześniej niż okno, a cel nadal stoi w strefie. Wtedy dostaje słabsze trafienie, raz, bez ogłuszenia, ze spowolnieniem i małym odepchnięciem.
>
> Zasada kciuka: **unik chroni przed ciosem, który spada w jego oknie. Przed ciosem, który trwa dłużej niż unik, chroni dopiero odległość.**

Klocek 1 [A]: strojenie zahaczenia i napisów

1. Otwórz `content/combat/attack_enemy_test.tres` (albo `attack_light.tres`, `attack_heavy.tres`). W grupie **Zahaczenie**:
   - **Can Graze** wyłącza zahaczenie tego ataku;
   - **Graze Damage Multiplier** to część obrażeń (0.4);
   - **Graze Knockback Scale** to część odrzutu (0.3);
   - **Slow Multiplier** i **Slow Duration** ustawiają spowolnienie (60% prędkości przez 0.6 s).
2. Na węźle `Combatant` przeciwnika odznacz **Can Be Grazed**, jeśli ma być odporny na zahaczenie (np. ciężki golem).
3. Otwórz `content/combat/combat_text_settings.tres`: czas życia napisu, uniesienie, wysokość nad celem, limit napisów, rozmiary i kolory.
4. W grze: F9 → „Walka” → „Napisy nad celem” i „Liczby obrażeń nad celem”.

Klocek 2 [B]: własna reakcja na zahaczenie

Gotowy skrypt do węzła-dziecka gracza albo przeciwnika (np. dźwięk szurnięcia):

```
extends Node


func _ready() -> void:
	var combatant: Combatant = Combatant.find_in(get_parent())
	if combatant != null:
		combatant.grazed.connect(_on_grazed)



func _on_grazed(data: DamageData) -> void:
	print("Zahaczony za ", data.amount, " pkt, spowolniony na ", data.slow_duration, " s")
```

Klocek 3 [C]: lista oczekujących w Hitbox

Wersja z komentarzami dydaktycznymi. Pełny kod jest w `res://core/combat/hitbox.gd`.

```
func _check_pending(overlapping: Array[Area3D]) -> void:
	# Ten kod działa co klatkę fizyki, dopóki okno ataku jest otwarte.
	for target_owner: Node in _pending.keys():
		var hurtbox: Hurtbox = _pending[target_owner]
		# 1. Cel wyszedł ze strefy (albo zniknął): uciekł, zahaczenia nie będzie.
		if not is_instance_valid(hurtbox) or not overlapping.has(hurtbox):
			_pending.erase(target_owner)
			continue
		# 2. Nadal nietykalny: czekamy na następną klatkę.
		if hurtbox.is_invulnerable():
			continue
		# 3. Nietykalność minęła, cel stoi w strefie: zahaczenie, dokładnie raz.
		#    Wpis do _hit_owners blokuje kolejne trafienia w tym oknie.
		_pending.erase(target_owner)
		_hit_owners.append(target_owner)
		var data: DamageData = _make_damage(hurtbox, true)
		if hurtbox.receive_hit(data) == Combatant.HitResult.HIT:
			grazed.emit(hurtbox, data)
```

Do przemyślenia:

- Dlaczego lista oczekujących znika w `deactivate()`?
- Co by się stało, gdyby cel był oznaczany jako trafiony już przy odrzuceniu, jak przed tym zadaniem?

Sprawdź, że działa

Z katalogu głównego projektu, w PowerShellu:

```
.\tools\run-tests.ps1
```

Oczekiwane: w tabeli każdy wiersz ma `Wynik = OK`, w tym `test_graze` i `test_combat_text`.

W grze:

1. F9 → „Walka” → „Przywołaj: enemy_test”.
2. Odskocz, gdy przeciwnik zaczyna cios, ale zostań w zasięgu miecza: nad kapsułą pojawi się „Zahaczenie!”.

Ciekawostka z kulis

Pierwsza wersja wyłącznika `can_graze` miała dziurę. Atak bez zahaczenia nie wpisywał celu na listę oczekujących, ale też nie oznaczał go jako obsłużonego. `Hitbox` próbował więc trafić go w każdej klatce i po końcu uniku zadawał… pełny cios. Wyłącznik, który miał osłabić atak, wzmacniał go. Test `can_graze = false: brak zahaczenia` złapał to od razu.

Zasada: **każda gałąź wyniku (`HIT`, `INVULNERABLE`, `IGNORED`) musi zostawić cel w jasnym stanie.**

Opublikowano z repozytorium Horrorek (commit 87717ad) skryptem publikuj-bookstack.ps1.