UNIVERSITATEA ”ALEXANDRU-IOAN CUZA” DIN IAȘI

FACULTATEA DE INFORMATICĂ

LUCRARE DE LICENȚĂ

Joc video 2D de tip dungeon crawler cu generare procedurală a nivelurilor

propusă de

Rareș Pipirig

Sesiunea: iulie, 2025

Coordonator științific

Conf. Dr. Alex Mihai Moruz

UNIVERSITATEA ”ALEXANDRU-IOAN CUZA” DIN IAȘI

FACULTATEA DE INFORMATICĂ

Joc video 2D de tip dungeon crawler cu generare procedurală a nivelurilor

Rareș Pipirig

Sesiunea: iulie, 2025

Coordonator științific

Conf. Dr. Alex Mihai Moruz

Cuprins

Abstract

Această lucrare prezintă dezvoltarea unui joc video 2D de tip dungeon crawler, cu niveluri generate procedural și un gameplay inspirat de genul souls-like. Jocul propus este un rogue-like bazat pe mobilitate și reacții precise, în care jucătorul parcurge o succesiune de cinci niveluri cu dificultate progresivă, fiecare fiind generat dinamic printr-un algoritm de tip Binary Space Partitioning (BSP).

Sistemul de luptă se bazează pe gestionarea resurselor (HP și Stamina), mecanici de luptă bazate pe sincronizare și mobilitate, iar progresia este susținută printr-un sistem de upgrade-uri între niveluri.

Lucrarea descrie ideile fundamentale, personajele, bucla de joc și realizarea tehnică a algoritmului de generare a nivelurilor. De asemenea, este analizat modul în care variația și dificultatea sunt reglate la nivelul camerei, pe baza unor proprietăți precum dimensiunile camerei, amplasarea obstacolelor și a inamicilor.

Obiectivul proiectului este de a oferi un gameplay scurt, dar fascinant, concentrându-se pe rejucabilitate, densitate și accesibilitate.

Inspirație

Inspirația pentru acest proiect vine din două titluri foarte diferite, dar complementare ca stil de joc: Soul Knight, dezvoltat de studio-ul ChillyRoom, un joc de mobil cu un ritm alert, axat pe acțiune continuă, sesiuni scurte și niveluri generate procedural, și Dark Souls, creat de FromSoftware, recunoscut pentru sistemul său de luptă complex, bazat pe precizie, sincronizare și gestionarea atentă a resurselor.

Prin urmare, prin fuziunea acestor două direcții, jocul propune o experiență similară cu cea a titlurilor din seria Souls, dar într-o formă condensată, mai dinamică (cu sesiuni de joc mai scurte), mai accesibilă și axată pe mobilitate.

Motivație

Dezvoltarea jocurilor video este printre cele mai dinamice și interdisciplinare domenii ale informaticii contemporane, incluzând algoritmi, design vizual, inteligență artificială și interactivitate. Proiectul a fost axat pe dezvoltarea unui joc 2D-dungeon crawler cu generare procedurală a nivelurilor, având ca inspirație jocurile de tip roguelike, care oferă o valoare mare de rejucabilitate, cu scopul de a explora în profunzime mecanismele generării automate de conținut.

Integrarea unui sistem de joc bazat pe precizie, mobilitate și reacții rapide într-un cadru cu niveluri generate dinamic aduce provocări tehnice importante, în special legate de proiectarea mecanicilor de joc și menținerea unui echilibru între dificultate și varietate.

Introducere

Industria jocurilor video s-a dezvoltat semnificativ în ultimele decenii, evoluând prin combinarea tehnologiei informatice cu arta și dând naștere unor interacțiuni tot mai dinamice și complexe. Jocurile de tip rogue-like și dungeon crawler au văzut o creștere semnificativă în popularitate datorită capacității lor de a oferi gameplay variat, rejucabilitate și provocări constante. Generarea procedurală a nivelurilor constituie un element esențial în acest tip de jocuri, facilitând construirea dinamică a hărților și oferind o experiență de joc diversificată.

Această lucrare urmărește realizarea unui joc video care combină structura procedurală și rejucabilitatea specifică genului roguelike cu un sistem de luptă inspirat din jocurile souls-like, centrat pe precizie, sincronizare și gestionarea inteligentă a acțiunilor în condiții de presiune.

Pentru dezvoltarea jocului a fost utilizat motorul de joc Unity, un mediu de dezvoltare adoptat pe scară largă în industrie, datorită flexibilității sale și a suportului solid oferit atât proiectelor 2D, cât și celor 3D. Unity permite scrierea logicii jocului în limbajul C#, folosind o versiune personalizată a platformei .NET, prin intermediul runtime-ului Mono. Această combinație oferă acces la o parte semnificativă din funcționalitățile bibliotecilor .NET Standard, facilitând astfel organizarea codului, gestionarea componentelor și dezvoltarea rapidă a funcționalităților complexe. Unity oferă, de asemenea, un set complet de unelte pentru fizică, animații, sisteme de particule, interfață grafică (UI) și navigație, toate fiind utilizate în cadrul acestui proiect pentru a crea o experiență de joc coerentă și interactivă.

Pe lângă utilizarea platformei Unity și .NET, au fost integrate și alte tehnologii și concepte relevante în cadrul proiectului. Pentru gestionarea codului sursă a fost folosită platforma GitHub, care a contribuit la o dezvoltare mai structurată și la o evidență clară a modificărilor. În ceea ce privește generarea nivelurilor, a fost implementat un algoritm de tip Binary Space Partitioning (BSP), o tehnică frecvent utilizată în dezvol-

tarea jocurilor pentru împărțirea logică a spațiului într-un mod eficient și controlabil. Acest algoritm a stat la baza structurii procedurale a hărților din joc, permițând variație și progresie în dificultate pe parcursul sesiunilor de joc.

Scopul proiectului este de a dezvolta nu doar un prototip jucabil, ci și de a înțelege care sunt componentele cheie care alcătuiesc un astfel de sistem: de la algoritmi pentru generarea structurilor de nivel și echilibrarea dificultății, până la implementarea mecanicilor de joc și a elementelor vizuale.

Prin această lucrare, se arată cum componentele tehnice și de design funcționează împreună în producția unui joc coerent, structurat în jurul unei experiențe scurte, dar de mare intensitate.

Capitolul 1

Fundamente Teoretice și Concepte de Bază

În dezvoltarea jocului prezentat în această lucrare, o înțelegere a influențelor de design este esențială. Astfel, se iau în vedere două genuri de jocuri ce stau la baza conceptelor propuse: Roguelike și Soulslike. Aceste tipuri de jocuri sunt foarte diferite, dar complementare, atât din punctul de vedere al structurii buclei de joc, cât și din cel al interacțiunii jucătorului cu mecanicile jocului.

Un joc Roguelike tradițional este de obicei descris ca un joc pe calculator cu un accent puternic pe gameplay complex și rejucabilitate, în care jucătorul are un timp nelimitat pentru a face o mișcare, ceea ce face ca jocul să fie comparabil mai degrabă cu șahul decât cu jocurile bazate pe reflexe, precum shooterele din perspectiva întâi (First Person Shooters). De asemenea, jocul oferă conținut și provocări noi la fiecare sesiune de joc, folosind tehnici de generare procedurală a conținutului [1].

Un joc Soulslike este un subgen al jocurilor de tip action RPG, cunoscut pentru nivelul ridicat de dificultate, lumi vaste populate de inamici și accentul pus pe narațiunea transmisă prin mediul înconjurător, de obicei într-un univers de fantezie întunecată (dark fantasy). Dezvoltatorul și publisher-ul japonez FromSoftware, împreună cu regizorul Hidetaka Miyazaki, au pus bazele genului prin lansarea jocului Demon's Souls (2009), popularizându-l mai târziu prin intermediul seriei Dark Souls (2011-2016) [2].

1.1 Game Design în jocuri de tip roguelike și souls-like

Jocurile Roguelike au văzut o tot mai mare popularitate în ultimii ani, având ca exemplu titluri precum Balatro (care a și câștigat Best Indie Game of the Year Award în 2024), dezvoltat de LocalThunk și publicat de Playstack, sau Hades 2, dezvoltat și publicat de Supergiant Games. Probabil cel mai important aspect al unui Roguelike este capacitatea sa de a fi rejucat. Aceste jocuri sunt concepute tocmai cu acest scop, oferind un grad mare de variație de la o sesiune la alta și, de cele mai multe ori, bazându-se pe un sistem de progresie proiectat în jurul acestei variații. O caracteristică definitorie a genului Roguelike este permadeath-ul (moartea permanentă), aceasta referindu-se la resetarea progresului jucătorului de câte ori acesta pierde; acest aspect aduce, în mod paradoxal, senzația de progres tocmai prin procesul de învățare. Hărțile (nivelurile), obiectele sau inamicii sunt de cele mai multe ori generate procedural, oferind o imprevizibilitate care nu doar că menține interesul jucătorului, ci și contribuie la stabilirea identității jocului. Roguelike-urile încurajează învățarea prin experimentare. Fiecare moarte nu este doar o penalizare, ci și o oportunitate de a înțelege mai bine sistemele jocului. Astfel, designul susține îmbunătățirea abilităților jucătorului, nu doar progresul numeric.

Figura 1.1: Exemplu de captură de ecran în cadrul unui joc Roguelike - Soul Knight (2017). Se observă personajul controlat de jucător în centrul ecranului, resursele jucătorului în colțul din stânga-sus, mai mulți inamici în partea de sus și o porțiune din nivelul generat procedural în fundal.

Spre deosebire de jocurile de tip Roguelike, care deveniseră populare încă din anii '80, cele Soulslike au început să apară pe piață în număr considerabil în jurul anilor 2010. Pe departe cele mai semnificative jocuri pentru acest gen sunt jocurile Dark Souls (2011-2016), produse de FromSoftware și publicate de Bandai Namco Entertainment. Originalul Dark Souls (2011) este considerat de critici ca fiind nu doar unul dintre cele mai influente și apreciate jocuri din toate timpurile, ci și un reper fundamental în consolidarea genului action RPG. Dark Souls 3 (2016) este totuși recunoscut ca fiind cel mai popular din serie, doborând recorduri de vânzări la lansare și fiind considerat ca o concluzie adecvată și satisfăcătoare a trilogiei.

Soulslike-urile sunt cunoscute în principal pentru dificultatea lor, ea fiind folosită ca instrument de design. Aceste jocuri nu sunt doar dificile, ci încurajează învățarea prin intermediul eșecului și al încercărilor repetate. Dificultatea are scopul de a crește importanța fiecărei acțiuni – greșelile sunt pedepsite, iar victoria este cu atât mai satisfăcătoare; jucătorul este încurajat să ia decizii calculate. Progresia nu este doar mecanică, ci și personală: jucătorul devine mai bun în joc, nu doar personajul. Povestea este adesea transmisă indirect prin medii, descrieri de obiecte sau personaje enigmatice, ceea ce încurajează explorarea și implicarea activă a jucătorului în reconstrucția universului ficțional; la acest aspect se adaugă lipsa de indicații vizuale sau hărți aglomerate – jucătorul trebuie să se orienteze vizual și să rețină repere din mediul înconjurător.

Figura 1.2: Captură de ecran în cadrul jocului Dark Souls 3 (2016)

Proiectul propus, intitulat ”No path back”, urmărește să combine concepte și elemente de game design aparținând celor două genuri prezentate anterior. Gameplay-ul de bază este inspirat din filosofia designului de tip souls-like, punând accent pe un sistem de luptă ce recompensează precizia, sincronizarea și învățarea prin repetiție. Mecanicile centrale sunt atacurile melee și evitarea sincronizată a atacurilor inamice (timed dodges), completate de abilități axate pe mobilitate. Structura jocului este optimizată pentru sesiuni scurte de joc. Fiecare rundă începe cu un nivel generat procedural, iar trecerea la următorul este permisă doar după îndeplinirea unor condiții predefinite. La finalul fiecărui nivel, jucătorul trebuie să aleagă între mai multe opțiuni de upgrade, aceste decizii influențând progresia și strategiile disponibile în etapele următoare.

1.2 Structura buclei de joc: sesiunea de joc scurtă și intensă

No Path Back este împărțit în cinci niveluri pe care jucătorul trebuie să le parcurgă pentru a ajunge la final. Obiectivul principal pentru fiecare sesiune de joc este, așadar, acela de a progresa cât mai departe în cadrul acestora, iar progresia poate fi măsurată în mod liniar prin gradul de completare a nivelurilor, determinat de abilitățile jucătorului. Astfel, fiecare sesiune oferă oportunitatea de dezvoltare a strategiilor și de familiarizare cu mecanicile de joc, menținând totodată un nivel ridicat de dificultate, întrucât, pe măsură ce competența jucătorului crește și acesta avansează, gradul de dificultate se intensifică proporțional.

Fiecare sesiune de joc este diferită, jucătorul confruntându-se cu un set nou de camere și o configurație diferită a inamicilor de fiecare dată. Parametrii transmiși algoritmului de generare a camerelor mențin curba graduală de creștere a dificultății, oferind în același timp suficientă variație pentru a păstra un grad ridicat de variabilitate, mai ales de-a lungul tranziției de la un nivel la altul.

Toate aceste elemente contribuie la o experiență de joc scurtă și intensă: jucătorul este încurajat să reîncerce după fiecare eșec pentru a ajunge cât mai departe, evitând în același timp aderarea la o rutină fixă, precum cea întâlnită în jocurile de tip Soulslike, în care jucătorii sunt adesea încurajați să memoreze traseele de pe hartă și tiparele de luptă ale inamicilor.

Bucla principală de joc este concepută pentru a oferi o experiență intensă, dar accesibilă, în sesiuni scurte de aproximativ 10–15 minute. Jocul începe într-o cameră de start fără inamici, unde jucătorul are ocazia să se familiarizeze rapid cu mecanicile de bază (mișcare, atac, dodge). De acolo, acesta avansează printr-o succesiune de niveluri generate procedural, fiecare compus din mai multe camere interconectate.

La finalul fiecărui "run", indiferent de succes sau eșec, jucătorul este încurajat să încerce din nou, beneficiind de experiența acumulată sau, după preferință, de un mod de joc alternativ fără upgrade-uri, destinat celor care doresc o provocare suplimentară.

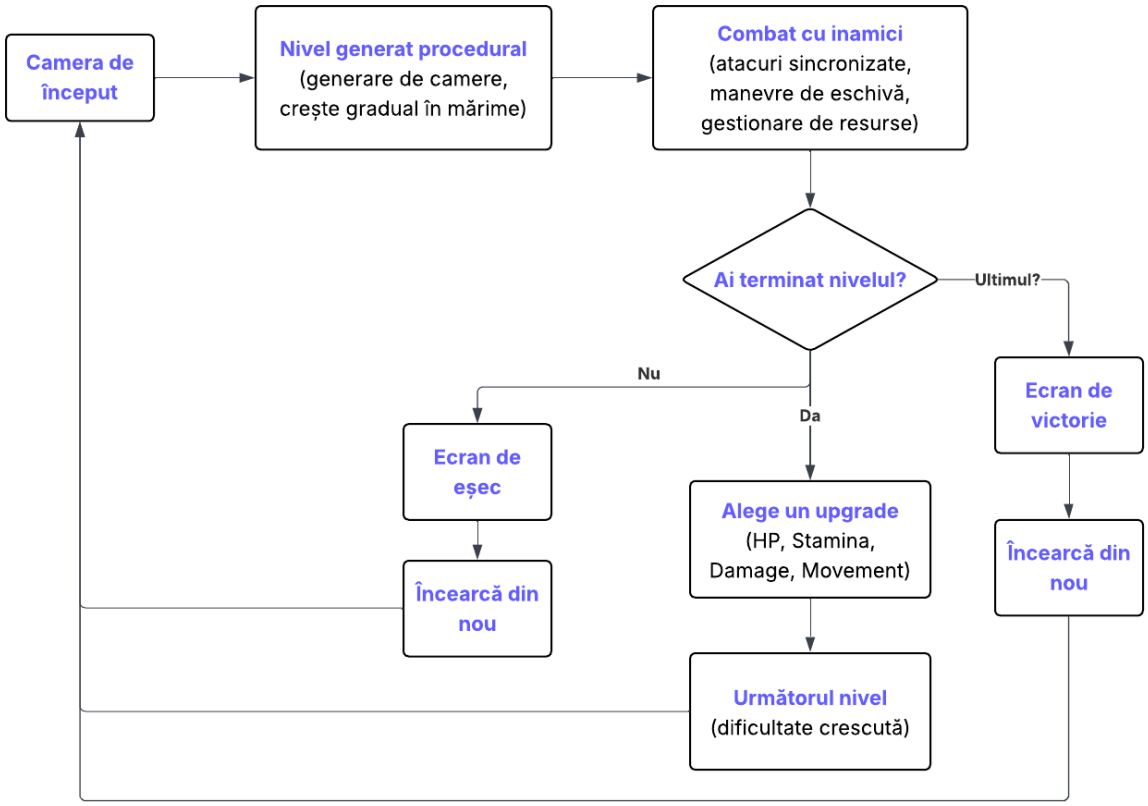

Figura 1.3: Diagrama buclei de joc a proiectului propus, intitulat "No path back".

1.3 Gestionarea resurselor jucătorului

Sistemul de luptă din joc este construit în jurul ideii de gestionare activă a resurselor, oferind jucătorului un cadru tactic în care fiecare acțiune are un cost și o consecință.

Resursele principale monitorizate pe parcursul unui „run” sunt HP-ul (viața), Stamina și timpii de cooldown asociați anumitor abilități.

HP-ul reprezintă sănătatea jucătorului și este afectat de fiecare atac primit din partea inamicilor. Stamina este o resursă regenerabilă rapid, dar care se consumă foarte ușor, fiind folosită pentru toate acțiunile principale ale jucătorului: atacuri, dodge-uri, sprint sau abilitatea specială de tip dash attack. Astfel, este introdus elementul tactic specific jocurilor Soulslike ce încurajează deciziile calculate și le pedepsește pe cele luate în mod impulsiv: jucătorul nu poate executa un număr nelimitat de manevre agresive sau evazive fără să își asume un risc, rămânerea fără Stamina îl lasă vulnerabil în fața atacurilor. Jucătorul recuperează aceste două resurse în mod pasiv, dar suficient de lent încât să fie sancționat comportamentul grăbit.

Pe lângă resursele numerice tradiționale, jucătorul gestionează și două resurse abstracte esențiale: potențialul ofensiv și mobilitatea. Potențialul ofensiv reflectă capacitatea de a provoca daune inamicilor, influențând eficiența atacurilor și impactul acestora în luptă. Mobilitatea, pe de altă parte, are un rol dublu: permite evitarea atacurilor prin manevre defensive rapide și, totodată, oferă un avantaj strategic în inițierea acțiunilor ofensive, facilitând repoziționarea sau angajarea rapidă în luptă.

1.4 Elementele de progresie și echilibrare

Progresia în cadrul jocului este concepută pentru a susține sesiuni de joc scurte, intense și rejucabile. Fiecare ”run” are o durată estimativă de 10–15 minute și este alcătuit dintr-o succesiune de cinci niveluri generate procedural. Aceste niveluri evoluează gradual din punct de vedere al dimensiunii spațiale și al complexității întâlnirilor cu inamicii, asigurând o creștere progresivă a dificultății. După finalizarea fiecărui nivel, jucătorul este recompensat cu oportunitatea de a alege un upgrade care să îi îmbunătățească resursele sau abilitățile de luptă. Alegerea acestor upgrade-uri devine o componentă strategică esențială, influențând atât stilul de joc adoptat, cât și șansele de reușită în nivelurile următoare. În acest mod, progresia se realizează printr-un echilibru între abilitățile dobândite de jucător și deciziile tactice asumate în timpul fiecărei sesiuni.

La finalul fiecărui nivel, jucătorului îi este prezentată opțiunea de a îmbunătăți una cele 4 resurse: HP-ul, Stamina (atât ca valori numerice totale, cât și rata de recuperare pasivă), potențialul ofensiv (este aplicat un multiplicator peste valoarea numerică

a daunelor fiecărui atac) sau mobilitatea (viteza sau distanța de mișcare, atât în cadrul mersului cât și al abilităților bazate pe mobilitate).

O altă formă de progresie este cea bazată pe principiul morții permanente (permadeath) ca stimulent al învățării. Ceea ce progresează este abilitatea jucătorului în sine. Acesta este un principiu des întâlnit în jocurile de tip Roguelike și stă la baza identității acestui proiect.

Capitolul 2

Personajul principal și inamicii

Acest capitol este dedicat analizei sistemului de personaje din proiectul ”No Path Back”, cu accent pe mecanica de joc și rolul lor în construirea experienței de gameplay. Personajele sunt proiectate să ofere o varietate de interacțiuni tactice, reflectând filosofia de design Soulslike combinată cu dinamica specifică genului Roguelike.

Personajul principal, denumit ”Swordsman”, este conceput pentru a oferi jucătorului un control precis asupra acțiunilor, cu un sistem de resurse care recompensează sincronizarea și pedepsește abuzul.

Inamicii sunt proiectați pentru a completa această dinamică prin comportamente distincte. Fiecare tip de inamic este creat pentru a testa anumite abilități ale jucătorului, iar combinațiile dintre ei în camerele generate procedural creează provocări unice. Designul vizual al personajelor, realizat în Clip Studio Paint, reflectă identitatea jocului prin siluete recognoscibile și elemente care facilitează citirea rapidă a acțiunilor în timpul jocului. Această abordare vizuală este esențială într-un mediu procedural unde jucătorul trebuie să ia decizii rapide bazate pe informații clare.

2.1 Jucătorul (Swordsman)

Personajul principal, denumit Swordsman, este conceput pentru a oferi o experiență de luptă rapidă și controlată, inspirată din jocurile souls-like, dar adaptată pentru un gameplay mai dinamic și accesibil. Mecanica sa se bazează pe patru acțiuni fundamentale: mers, evitare (dodge), sprint și atac.

El se folosește de două resurse principale: HP - care determină supraviețuirea jucătorului; odată epuizat, jucătorul moare și începe un nou ”run”; Stamina - folosită

pentru toate acțiunile offensive și defensive. Stamina se regenerează rapid, dar se consumă și mai repede, ceea ce forțează jucătorul să gestioneze acțiunile cu grijă. Atacurile repetitive (spamming) vor epuiza rapid stamina, lăsând jucătorul vulnerabil. Evitarea excesivă poate duce la lipsa resurselor necesare pentru a contraataca.

Sunt, de asemenea, două mecanici diferite pentru mobilitate pură: Dodge (scurt apăsat) – o mișcare rapidă în orice direcție, oferind invincibilitate temporară la început, concepută pentru a evita atacurile inamicilor; Sprint (apăsat lung) – O mișcare mai extinsă, care permite jucătorului să acopere distanțe mai mari, folositoare pentru repoziționare sau evadare. Jucătorii pot alege între o evitare rapidă (pentru contracararea unui atac iminent) sau un sprint mai lung (pentru schimbarea poziției în luptă).

Jucătorul are de asemenea la dispoziție două tipuri diferite de atacuri: Sword Slash (atacul de bază) - rază scurtă, atac "melee", cu un cost redus de Stamina, permițând mai multe lovituri succesive, dar riscând epuizarea dacă este folosit excesiv, și fără cooldown, dar limitat de Stamina, ceea ce încurajează timing precis; Dash Attack (atac în mișcare) - abilitate secundară care combină mobilitatea și crowd control-ul, jucătorul se deplasează rapid într-o direcție, lovind toți inamicii din cale și cu un cooldown moderat, ceea ce împiedică abuzul și încurajează folosirea strategică. Jucătorul poate folosi cele două atacuri în direcția de mers sau în direcția de țintire.

Designul Swordsman-ului urmărește fluiditatea - mișcările rapide și regenerarea staminei mențin ritmul alert, profunzimea tactică – jucătorul trebuie să echilibreze atacul, apărarea și poziționarea - și accesibilitatea – controale simple, dar cu un nivel înalt de control asupra acțiunilor, ele fiind gândite pentru a fi folosite pe un controller, dar cu posibilitatea de a fi accesate complet doar din tastatură.

Figura 2.1: Schiță de concept și aspectul din joc pentru Swordsman.

2.2 Inamicii

2.2.1 Depraved

Depraved reprezintă cel mai simplu tip de inamic din joc, cu un singur atac melee și o IA directă, acest monstru este proiectat să fie ușor de gestionat în număr mic, dar periculos în valuri sau când apare alături de alți inamici. Are un singur tip de atac: o lovitură rapidă la distanță scurtă, ușor de anticipat dacă Depraved este singur. Dacă lovește, provocă daune moderate, dar poate deveni problematic când mai mulți Depraved atacă simultan.

Singur poate fi eliminat cu un dodge și câteva contraatacuri, dar devine o provocare în mulțime, poate distrage atenția permițând inamicilor mai puternici să atace de la distanță.

Fiind predictibil, Depraved permite învățarea mecanicilor de bază (dodge, counter). Pe măsură ce jucătorul progreseaza, numărul de Depraved contribuie la creșterea organică a dificultății și pune presiune pe gestionarea resurselor - evitarea atacurilor multiple consumă Stamina, iar neglijența poate duce la pierderi inutile de HP.

Depraved este un inamic potrivit pentru a menține luptele dinamice – simplu în design, dar capabil să schimbe radical dificultatea atunci când apare în număr mare.

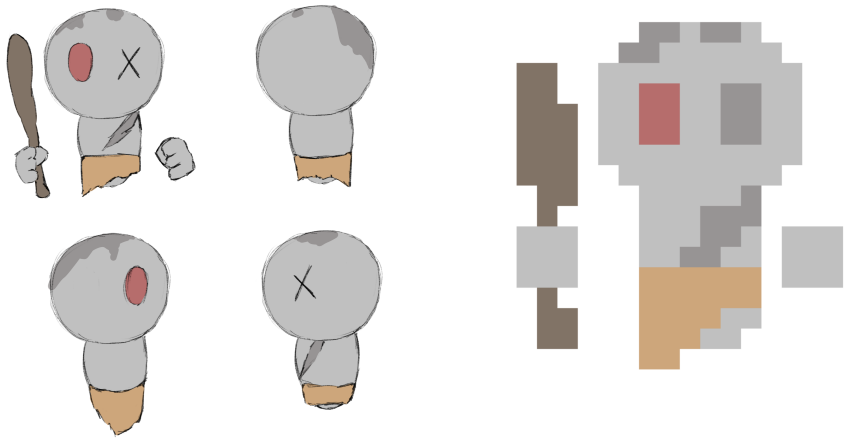

Figura 2.2: Schiță de concept și aspectul din joc pentru Depraved.

2.2.2 Mage

Mage-ul introduce o dimensiune nouă în luptă, transformând confruntările din simple dueluri melee în bătălii tactice ce necesită conștientizare spațială și gestionare a priorităților. Spre deosebire de Depraved care atacă direct, Mage-ul păstrează o distanță moderată, forțând jucătorul să schimbe constant poziția și să țină cont de multiple surse de pericol simultan.

El are, ca și Depraved, un singur atac: Fireball - un proiectil cu traiectorie lentă, dar precisă, ușor de evitat individual, dar care devine problematic când sunt lansate multiple simultan sau jucătorul este ocupat să evite Depraved, forțând jucătorul să păstreze o mobilitate constantă.

Comportamentul lui este unul tactic, mișcându-se lateral pentru a menține unghiul de atac optim și încercând mereu să rămână în limita razelor de acțiune. Devine cel mai periculos când Depraved blochează căile de retragere sau alt Mage creează suprapuneri de zone de foc, forțând jucătorul să își prioritizeze țintele.

Mage-ul schimbă dinamica luptelor, introducând conceptul de "amenințare la distanță" și transformând mediul de luptă într-un puzzle dinamic. El forțează jucatorul să gestioneze mai multe tipuri de amenințări simultan și îl învață să monitorizeze întreg câmpul de luptă și să planifice mișcări cu mai mulți pași înainte. Singur este provocator dar gestionabil, dar în combinații devine multiplicator de dificultate.

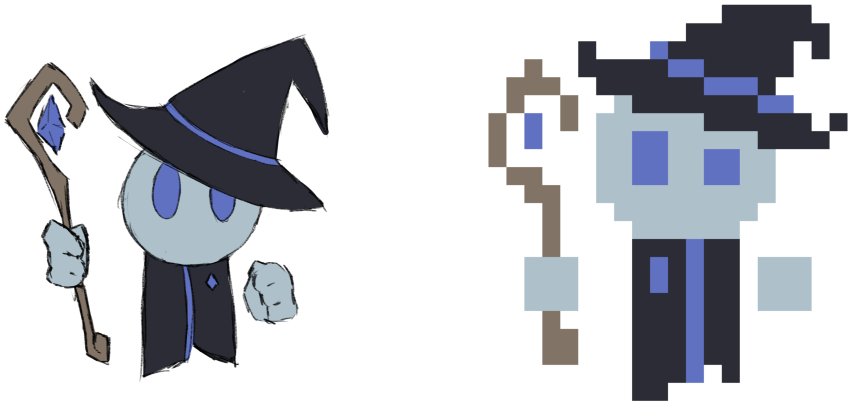

Figura 2.3: Schiță de concept și aspectul din joc pentru Mage.

Capitolul 3

Arhitectura

Proiectul propus a fost dezvoltat prin intermediul motorului de joc Unity, care se bazează pe limbajul de programare C# și pe o versiune personalizată a framework-ului .NET, utilizând runtime-ul Mono pentru a rula scripturile. Pentru scrierea script-urilor și identificarea erorilor a fost folosit mediul de dezvoltare integrat (Integrated Development Environment – IDE) Visual Studio, dezvoltat de compania Microsoft.

Gestiunea versiunilor a fost realizată prin intermediul platformei GitHub, iar pentru menținerea coerenței conceptelor de game design a fost utilizată platforma Notion.

Algoritmul de generare a nivelurilor este bazat pe metoda Binary Space Partitioning (BSP), propusă de Schumacker et al în anul 1969.

Elementele grafice ale jocului provin, pe de o parte, din setul de sprite-uri "DungeonTileset II" creat de 0x72 (https://0x72.itch.io/dungeontileset-ii), iar, pe de altă parte, au fost realizate manual folosind programul Clip Studio Paint.

3.1 Overview

Pentru dezvoltarea proiectului, a fost utilizat motorul de joc Unity, unul dintre cele mai populare și accesibile instrumente de creare a aplicațiilor interactive în timp real. Unity oferă un mediu de dezvoltare integrat (IDE) robust, capabil să gestioneze atât componente 2D, cât și 3D, și este cunoscut pentru suportul extensiv al platformelor multiple, incluzând PC, console, dispozitive mobile și web [3]. Această flexibilitate a contribuit la alegerea Unity pentru proiectul de față, oferind toate uneltele necesare pentru dezvoltarea unui joc 2D complex, cu elemente de generare procedurală și inte-

ractivitate în timp real.

Un avantaj major al Unity este utilizarea limbajului de programare C#, care oferă o sintaxă clară și puternică pentru gestionarea logicii jocului. Arhitectura component-based a motorului facilitează dezvoltarea modulară, în care fiecare GameObject poate fi extins cu comportamente personalizate prin scripturi, promovând astfel o abordare scalabilă și ușor de întreținut. În plus, Unity oferă un sistem avansat de event-driven programming, sistem de fizică integrat (atât 2D cât și 3D), precum și suport pentru sisteme de animație bazate pe stări (Animation Controller), aspecte esențiale pentru realizarea unui gameplay fluid și dinamic într-un joc orientat pe mișcare, cum este cel de tip Roguelike [4].

În contextul acestui proiect, diversele facilități regăsite în motorul Unity au fost esențiale în implementarea mecanicilor de bază ale jocului: controlul jucătorului în medii 2D utilizând noul sistem de gestionare al input-ului, coliziunile, facilitățile RigidBody2D, managementul resurselor și animațiile. De asemenea, Unity a permis integrarea și testarea ușoară a unui algoritm de generare procedurală a nivelurilor.

Motorul de joc Unity utilizează limbajul de programare C# ca principal mediu de scripting, oferind dezvoltatorilor un cadru robust, orientat pe obiect și ușor de integrat cu sistemele oferite de motor. C# permite o structurare clară a codului și favorizează reutilizarea componentelor, aspecte esențiale în dezvoltarea de jocuri modulare și scalabile. Pentru rularea codului C# în cadrul motorului, Unity se bazează pe Mono, o implementare open-source a framework-ului .NET. Acest runtime permite execuția multiplatformă a aplicațiilor Unity și oferă suport pentru funcționalități precum garbage collection, reflection și interoperabilitate cu biblioteci externe. Integrarea dintre Unity, C# și Mono facilitează un echilibru eficient între performanță și flexibilitate, făcând posibilă dezvoltarea de aplicații interactive complexe într-un mod accesibil și rapid [5].

Pentru gestiunea versiunilor a fost utilizată platforma GitHub, un serviciu de găzduire a codului sursă bazat pe sistemul de control al versiunilor Git. GitHub permite urmărirea modificărilor în cod, revenirea la versiuni anterioare, precum și lucrul colaborativ prin ramuri paralele (branches) și solicitări de îmbinare (pull requests). În contextul dezvoltării jocurilor, GitHub oferă o metodă sigură și transparentă de gestionare a codului și a resurselor asociate, permițând automatizarea proceselor de livrare și integrare continuă (CI/CD). De asemenea, GitHub oferă integrare cu platforma Unity prin mecanisme de versionare adaptate pentru fișiere binare sau meta-informații specifice proiectelor de jocuri [6].

Pentru organizarea procesului de dezvoltare și documentarea etapelor de game design, a fost utilizată platforma Notion, o aplicație digitală de tip workspace all-in-one, care permite structurarea informațiilor sub formă de pagini, baze de date, liste și calendare [7]. În cadrul dezvoltării individuale a proiectului, Notion a oferit un mediu centralizat pentru planificarea conceptelor de joc, notarea ideilor și a mecanicilor, precum și pentru urmărirea progresului în timp. Utilizarea acestui instrument a facilitat menținerea coerenței designului pe parcursul etapelor de dezvoltare și a permis documentarea eficientă a deciziilor luate, contribuind astfel la un proces de lucru clar și organizat.

Pentru generarea procedurală a nivelurilor, a fost implementat un algoritm bazat pe Binary Space Partitioning (BSP), o tehnică de divizare recursivă a spațiului propusă inițial de Schumacker et al. (1969) în contextul randării scenelor 3D [8][9]. În cazul acestui proiect, BSP a fost adaptat pentru generarea de camere și coridoare într-un mediu bidimensional, specific jocurilor de tip Roguelike. Divizarea spațiului permite construirea unei structuri arborescente care oferă control asupra distribuției camerelor și asupra conexiunilor dintre ele, păstrând în același timp un echilibru între varietatea layout-ului și coerența traseului de joc. Astfel, se obține un sistem flexibil și extensibil de generare a nivelurilor.

Pentru realizarea elementelor grafice originale ale jocului, a fost utilizat programul Clip Studio Paint, un software de editare și desen digital apreciat pentru versatilitatea și funcționalitățile sale dedicate artiștilor 2D. Clip Studio Paint oferă o gamă largă de pensule personalizabile, suport pentru straturi multiple și unelte avansate pentru crearea de sprite-uri și animații frame-by-frame, fiind astfel o alegere frecventă în industria jocurilor video și a benzii desenate digitale [10]. Utilizarea acestui program a permis dezvoltarea unor elemente vizuale detaliate și coerente stilistic, adaptate nevoilor specifice ale proiectului.

3.2 Structura proiectului

3.2.1 Assets

Structura folderului Assets predefinit în unity:

- **Animations** - Conține toate animațiile jocului (pentru personaje, obiecte, efecte speciale).

- **Misc** - Elemente diverse care nu intră în alte categorii (UI temporar, teste).
- **Objects** - Prefabs pentru toate elementele din proiect, plasabile direct în spațiul de joc.
  - Main Character: Prefabs specifice personajului principal.
  - Enemies: Resurse pentru inamici.
- **Resources** - Asset-uri încărcate dinamic (folosite pentru Resources.Load). Majoritar compus din tiles pentru nivelurile generate procedural.
- **Scenes** - Toate scenele proiectului (scena principală și scenele de test).
- **Scripts** - Logica jocului.
  - Characters - logica corespunzătoare personajelor.
    * Main character - Scripturile care controlează mișcarea, atacurile și resursele jucătorului.
    * Enemies - IA și comportamente pentru fiecare tip de inamic.
  - Level Generation - Logica corespunzătoare gestionării generării de niveluri.
- **Sprites** - Elemente vizuale 2D.
  - Main character - Sprite-urile personajului principal și ale elementelor ce țin de personajul principal (spre exemplu indicatorul pentru țintă).
  - Enemies: Sprite-uri și diverse elemente pentru toți inamicii.

Obiectivele acestei structuri:

- **Modularitate** - Schimbări la un sistem (de exemplu inamici) nu afectează alte părți ale proiectului.
- **Scalabilitate** - Adăugarea de noi caracteristici (de exemplu un inamic nou) se face într-un folder dedicat.
- **Lizibilitate** - Se poate naviga rapid pentru a găsi asset-uri sau scripturi specifice.

Exemplu de flux de lucru pentru adăugarea unui inamic nou:

- Pune sprite-urile în Assets/Sprites/Enemies.
- Configurează animațiile în Assets/Animations/Enemies.
- Adaugă scriptul pentru logica personajului în Assets/Scripts/Enemies.
- Creează un prefab în Assets/Objects/Enemies.

3.2.2 Packages

Pachetele relevante neincluse sau neinstalate implicit in proiectul de Unity:

- Aseprite Importer [11]
  - Adaugă suport pentru importarea fișierelor .ase/.aseprite.
  - Folosit pentru integrarea directă cu editorul de pixeli Aseprite.
- Pixel Perfect [12]
  - Păstrează dimensiunile întregi ale pixelilor chiar și când camera se mișcă sau când jocul rulează la rezoluții diferite.
  - Elimină artefacte de interpolare (pixeli ”murdari” sau aliniați greșit).
  - Folositor pentru jocuri 2D retro sau orice proiect care folosește sprite-uri cu pixeli definiți.
- PSD Importer [13]
  - Convertește automat PSD-uri în sprite-uri sau texturi fără a fi nevoie de export manual.
  - Păstrează transparența și grupările de layere.
  - Util pentru importarea directă în proiectul de Unity a proiectelor din Clip Studio Paint convertite în fisiere de tip .psb (Photoshop Big Document).
- Input System
  - Noul Input System din Unity.
  - Pachetul este recomandat pentru proiectele noi, înlocuind vechiul Input Manager (legacy).

Capitolul 4

Implementarea

4.1 Sistemul de control al jucătorului

Noul Input System din Unity este o revizuire majoră a sistemului vechi de gestionare a input-urilor, introdus pentru a oferi flexibilitate sporită, suport pentru dispozitive moderne și performanță îmbunătățită. Acesta permite:

- Gestionarea centralizată a tuturor dispozitivelor (gamepad, tastatură, mouse, touch, dispozitive VR).
- Scheme de input customizabile (ex: aceeași acțiune poate fi mapată la mai multe dispozitive).
- Event-based sau polling (pentru răspunsuri la acțiuni sau citire continuă).
- Suport pentru rebinding (schimbarea controalelor în timp real) [14].

4.1.1 PlayerController

Scriptul denumit `PlayerController` reprezintă componenta principală responsabilă de gestionarea tuturor comportamentelor asociate jucătorului. Acesta centralizează logica de mișcare, interacțiune, combat și management al resurselor, constituind astfel nucleul funcțional al entității controlate de utilizator.

Atribute principale:

- Referințe către instanțele scripturilor auxiliare atașate GameObject-ului jucătorului, precum `AttackController`, `MovementController` și `StaminaSystem`.

- `private PlayerInput playerInput` – instanță a sistemului de input oferit de Unity (Input System), utilizată pentru a prelua și interpreta comenzile jucătorului.
- Diverse câmpuri de configurare pentru comportamentul knockback, aplicat atunci când jucătorul este lovit de un inamic.
- `private int HP` – valoarea curentă a vieții jucătorului.
- `internal bool isInvincible` – flag utilizat pentru marcarea perioadelor de invincibilitate temporară (de exemplu, în timpul unui dodge).

Funcții principale:

- `private void OnEnable()` și `private void OnDisable()` – funcții care se ocupă de inițializarea și dezactivarea sistemului de input, stabilind legătura dintre comenzile definite și metodele apelate.
- `public async void TakeDamage(int damage, Vector2 aimDirection)` – apelată la primirea unui atac; actualizează valoarea HP-ului și aplică un efect de recul (knockback) în direcția specificată.
- `public void Upgrade(string path)` – responsabilă de aplicarea upgrade-urilor la finalul fiecărui nivel. În funcție de parametrul `path`, modifică valori numerice din alte componente precum viața, stamina sau daunele.

4.1.2 AttackController

Scriptul denumit `AttackController` reprezintă componenta responsabilă de gestionarea tuturor aspectelor care țin de atac.

Atribute principale:

- `private PlayerController playerController` – instanță a scriptului `PlayerController` asociat GameObject-ului jucătorului.
- `internal Vector2 aimInput` – valoarea input-ului legat de aim.

- `internal Vector2 aimDirection = Vector2.down` – valoarea folosită de diversele funcții de atac pentru țintă (are valoarea input-ului de aim când acesta există și valoarea direcției de mers când nu).
- Diverse câmpuri de configurare pentru comportamentul Sword Slash.
- Diverse câmpuri de configurare pentru comportamentul Dash Attack.

Funcții principale:

- `internal async void SwordSlash()` – este metoda responsabilă de executarea atacului de tip Sword Slash și calculul daunelor.
- `internal async void DashAttack()` – este metoda responsabilă de executarea atacului de tip Dash Attack, calculul daunelor și inițierea temporizatorului de cooldown care previne reutilizarea imediată a abilității.

4.1.3 MovementController

Scriptul denumit `MovementController` reprezintă componenta responsabilă de gestionarea tuturor aspectelor care țin de mobilitatea personajului principal.

Atribute principale:

- `private PlayerController playerController` – instanță a scriptului `PlayerController` asociat GameObject-ului jucătorului.
- `internal Vector2 movementInput` – valoarea input-ului legat de mișcare.
- `internal Rigidbody2D rb` – referință la obiectul de tip RigidBody2D asociat GameObjectul-ui jucătorului, și pe care se bazează tot ce ține de mobilitate.
- `internal Vector2 lastRecordedDirection = Vector2.down` – reține ultima valoare a input-ului de direcție de mișcare; important în executarea acțiunilor de mobilitate când nu există input de direcție activ.
- Diverse câmpuri de configurare pentru comportamentul de mers.
- Diverse câmpuri de configurare pentru comportamentul Dash.
- Diverse câmpuri de configurare pentru comportamentul Sprint.

Funcții principale:

- `private void Move(Vector2 targetPos)` – funcție universală de mișcare pentru jucător; se folosește de o combinație de flag-uri pentru a determina viteza potrivită de miscare în direcția indicată de parametrul `targetPos`; această funcție este folosită inclusiv pentru comportamentele de Dash și Sprint.
- `internal async void StartDash()` și `internal void StopDash()` – sunt metodele responsabile de executarea acțiunii de tip Dash, continuarea ei cât timp jucătorul ține apăsată tasta corespunzătoare și inițierea acțiunii de Sprint după caz.

4.1.4 StaminaSystem

Scriptul denumit `StaminaSystem` reprezintă componenta responsabilă de gestionarea tuturor aspectelor care țin de resursa Stamina.

Atribute principale:

- Diverse câmpuri de configurare pentru resursa de Stamina (cooldown, capacitatea maximă, rata de recuperare, etc.).
- `private float currentStamina` – valoarea curentă a resursei Stamina a jucătorului.

Funcții principale:

- `private void Update()` – funcția care se ocupă de recuperarea pasivă a resursei Stamina.
- `public bool UseStamina(float amount)` - funcția apelată de fiecare componentă ce are nevoie de Stamina pentru a aplica consumarea resursei.

4.1.5 HP

Scriptul denumit `HPSystem` reprezintă componenta responsabilă de gestionarea tuturor aspectelor care țin de resursa HP.

Atribute principale:

- Diverse câmpuri de configurare pentru resursa de HP (cooldown, capacitatea maximă, rata de recuperare, etc.).
- `public PlayerController playerController` – instanță a scriptului `PlayerController` asociat GameObject-ului jucătorului.

Funcții principale:

- `private void Update()` – funcția care se ocupă de recuperarea pasivă a resursei HP.

4.1.6 Componente auxiliare

Deși animațiile nu joacă un rol central în gameplay-ul jocului, implementarea lor a fost realizată prin intermediul a două scripturi dedicate. Script-ul `AnimationController` gestionează tranzițiile dintre stările vizuale ale personajului în funcție de direcția de mers a jucătorului, iar script-ul `AnimationContainer` ajută la încărcarea asset-urilor ce țin de animație în memorie. Având în vedere numărul redus de animații utilizate, aceste clase au un rol funcțional limitat și nu sunt integrate direct în logica de combat sau mișcare.

De asemenea, în cadrul proiectului există clasa `PlayerUI`, responsabilă de gestionarea elementelor de interfață grafică asociate jucătorului.

4.2 Inamicii

4.2.1 Depraved

Script-ul `DepravedController` gestionează toate funcționalitățile personajului Depraved.

Atribute principale:

- `private int HP` – valoarea curentă a vieții personajului.
- `internal Rigidbody2D rb` – referință la obiectul de tip RigidBody2D asociat GameObjectul-ui personajului, și pe care se bazează tot ce ține de mobilitate.

- Diverse câmpuri de configurare pentru comportamentul knockback, aplicat atunci când personajul este lovit de jucător.
- Diverse câmpuri de configurare pentru comportamentul de mers.
- Diverse câmpuri de configurare pentru comportamentul de stun.
- `Vector2 playerDirection = Vector2.zero` – direcția curentă a jucătorului relativ la personaj.
- Diverse câmpuri de configurare pentru detecția jucătorului.
- Diverse câmpuri de configurare pentru atacul melee.

Funcții principale:

- `private void Move()` – funcție universală pentru mișcare; Depraved încearcă mereu să meargă către jucător.
- `private void GetPlayerDirection()` - funcție apelată la fiecare frame pentru detecția permanentă a poziției jucătorului relativ la personaj.
- `private bool HasLineOfSight(Vector3 targetPosition)` - verifică dacă jucătorul se alfă în câmpul vizual al personajului.
- `private bool isCloseEnoughToPlayer()` - funcție care apreciază dacă personajul s-a apropiat suficient de jucător.
- `public async void TakeDamage(int damage, Vector2 aimDirection)` – apelată la primirea unui atac; actualizează valoarea HP-ului, aplică un efect de recul (knockback) în direcția specificată și aplică stun.
- `public async void HitPlayer()` - dacă jucătorul se află în raza de acțiune, se încearcă lovirea lui cu un atac.

4.2.2 Mage

Script-ul `MageController` gestionează funcționalitățile personajului Mage ce țin de mișcare, detecția jucătorului și atac.

Atribute principale:

- `private int HP` – valoarea curentă a vieții personajului.
- `internal Rigidbody2D rb` – referință la obiectul de tip RigidBody2D asociat GameObjectul-ui personajului, și pe care se bazează tot ce ține de mobilitate.
- Diverse câmpuri de configurare pentru comportamentul knockback, aplicat atunci când personajul este lovit de jucător.
- Diverse câmpuri de configurare pentru comportamentul de mers.
- Diverse câmpuri de configurare pentru comportamentul de stun.
- `Vector2 playerDirection = Vector2.zero` – direcția curentă a jucătorului relativ la personaj.
- Diverse câmpuri de configurare pentru detecția jucătorului.
- Diverse câmpuri de configurare pentru atacul Fireball.

Funcții principale:

- `private void Move()` – funcție universală pentru mișcare. Mage încearcă să păstreze o distanță moderată față de jucător.
- `private void GetPlayerDirection()` - funcție apelată la fiecare frame pentru detecția permanentă a poziției jucătorului relativ la personaj.
- `public async void TakeDamage(int damage, Vector2 aimDirection)` – apelată la primirea unui atac; actualizează valoarea HP-ului, aplică un efect de recul (knockback) în direcția specificată și aplică stun.
- `public async void HitPlayer()` - dacă jucătorul se află în raza de acțiune, se lansează un Fireball.

Script-ul `FireballController` gestionează funcționalitățile GameObject-ului de tip Fireball.

Atribute principale:

- `private Vector2 direction` – direcția proiectilului, definită la momentul inițializării lui.

- `private float speed` – viteza proiectilului.
- `Rigidbody2D rb` – referință la obiectul de tip RigidBody2D asociat GameObjectul-ui asociat instanței de Fireball, și pe care se bazează mișcarea proiectilului.
- Diverse câmpuri de configurare pentru detecția jucătorului.
- `private int damage` – dauna atacului.

Funcții principale:

- `public void Initialize(Vector2 target)` – funcție ce inițializează direcția de mers și rotația proiectilului în momentul în care este creat; funcție apelată din cadrul script-ului `MageController`.
- `private void FixedUpdate()` – deoarece proiectilul se deplasează liniar, linia de cod ce aplică velocitatea pentru mișcare atributului `rb` se află direct în interiorul funcției `FixedUpdate()` care este apelat la fiecare ciclu de fizică (Physics Cycle).
- `public void TakeDamage()` – distruge obiectul dacă este lovit de jucător.
- `private void CheckPlayerHit()` – verifică dacă proiectilul face contact cu jucătorul; dacă da, jucătorului îi este apelată funcția `TakeDamage()` iar apoi GameObject-ul asociat proiectilului este distrus.
- `private void OnCollisionEnter2D(Collision2D collision)` – distruge obiectul dacă acesta are o coliziune cu un perete.

4.3 Generarea procedurală a nivelurilor

Sistemul de generare procedurală a nivelurilor este realizat prin trei componente principale:

- `BSP` – Implementează algoritmul Binary Space Partitioning pentru divizarea spațiului în camere.
- `RoomGenerator` – Populează fiecare cameră cu obstacole, inamici și decoruri, pe baza unor reguli de dificultate.

- `LevelGenerationController` – Coordonează procesul și asigură legătura între camere.

Detalii despre logica fiecărui script și parametrii de configurare vor fi prezentați în secțiunea dedicată generării procedurale (Capitolul 5).

Capitolul 5

Generarea procedurală de niveluri

Generarea procedurală de niveluri în proiectul propus se bazează pe trei componente principale: `BSP`, responsabilă de generarea structurii (layout-ului) nivelului; `RoomGeneration`, care se ocupă de generarea și decorarea individuală a tuturor camerelor și a coridoarelor, precum și de plasarea acestora în spațiul de joc; și `LevelGenerationController`, care interconectează aceste componente și plasează inamicii în nivel conform unui algoritm de calcul al dificultății, ce primește parametri diferiți în funcție de nivelul curent.

5.1 BSP

Această componentă a algoritmului reprezintă o implementare relativ clasică a metodei BSP (Binary Space Partitioning), adaptată prin introducerea unor constrângeri suplimentare sau modificate, menite să corespundă cerințelor specifice ale proiectului.

Algoritmul debutează prin definirea dimensiunilor spațiului de joc, a numărului maxim de iterații permise și a numărului maxim de camere care ar trebui generate.

Primul pas al procesului constă în construirea unui graf binar, în care rădăcina reprezintă întreaga hartă. Ulterior, în funcție de numărul de iterații sau de împărțiri permise de parametrii algoritmului, fiecare secțiune este divizată aleatoriu în două sub-secțiuni, care devin copiii nodului părinte. Acest proces de împărțire se repetă recursiv, până la atingerea limitelor impuse.

După divizarea completă a spațiului de joc, în fiecare secțiune sunt plasate camere de dimensiuni generate aleatoriu. Rezultatul intermediar al acestui proces este transpus într-o reprezentare vizuală sub forma unei matrice de caractere.

În etapa finală, algoritmul se concentrează pe conectarea camerelor generate. Se parcurg toate nodurile interne (care nu sunt frunze) ale arborelui binar și se generează coridoare între perechile de camere corespunzătoare copiilor fiecărui nod. Pentru fiecare astfel de pereche, sunt selectate aleatoriu două camere și se caută o rută liberă între acestea. În cazul în care nu se identifică o rută validă, selecția aleatorie este reluată până la găsirea unei conexiuni posibile. Această conexiune este garantată, întrucât secțiunile care trebuie conectate sunt întotdeauna adiacente din punct de vedere spațial. Coridoarele sunt trasate prin stabilirea unor puncte de intrare în pereții camerelor, selectate tot aleatoriu. Se acordă prioritate traseelor drepte, iar coridoarele în formă de „L" (L-shaped corridors) sunt utilizate doar în situațiile în care nu este posibilă o conexiune liniară.

Atribute principale:

- Diverse câmpuri de configurare pentru parametrii algoritmului.
- `internal char[,] matrix` – matricea pe care este scrisă structura nivelului generat.
- `internal List<Leaf> leaves` – graful binar folosit în executarea algoritmului de generare.
- `public class Leaf` – definiția unui nod al grafului folosit de algoritm; conține atribute legate de poziția și dimensiunile secțiunii de hartă pe care o reprezintă, a camerei din interiorul ei și a intrărilor din acea cameră.

Funcții principale:

- `public void GenerateDungeon(int width, int height, int numberOfIterations, int maxSplits)` – gestionează rularea fiecărei etape a algoritmului în parametrii specificați.
- `void CreateSplits(int maxSplits)` – împarte spațiul de joc în secțiuni pe care le adaugă în graf.
- `public bool Split(int minRoomSize)` – împarte aleatoriu o secțiune în două secțiuni copil.
- `public void CreateRoom()` – plasează o cameră într-o secțiune.

- `void WriteToMatrix()` – scrie rezultatul parțial al algoritmului în matricea de caractere.
- `void CreateRoads()` – iterează prin toate nodurile interne ale grafului și creează conexiuni intre copiii lor.
- `void CreateCorridor(Leaf l)` – trasează un coridor între două secțiuni ale spațiului de joc.
- `bool CanConnectRooms(Rect room1, Rect room2)`, `bool CanConnectStraight(Rect room1, Rect room2)` și `bool HasClearPath(Rect room1, Rect room2)` – verifică posibilitatea de a conecta două camere.
- `void ConnectRooms(Rect room1, Rect room2)` – conectează două camere după ce a fost verificată posibilitatea formării acelei conexiuni.
- `void DrawStraightCorridor(Vector2Int start, Vector2Int end)` – trasează un drum drept dintr-un punct în altul.
- `void DrawStraightCorridorBetweenRooms(Rect room1, Rect room2)` – trasează un drum drept de la o cameră la alta.
- `void DrawLShapedCorridor(Rect leftRoom, Rect rightRoom)` – trasează un drum în formă de ”L” de la o cameră la alta.
- `Vector2Int GetBestConnectionPoint(Rect fromRoom, Rect toRoom)` – selectează aleatoriu un punct optim pentru a plasa intrarea într-o cameră.

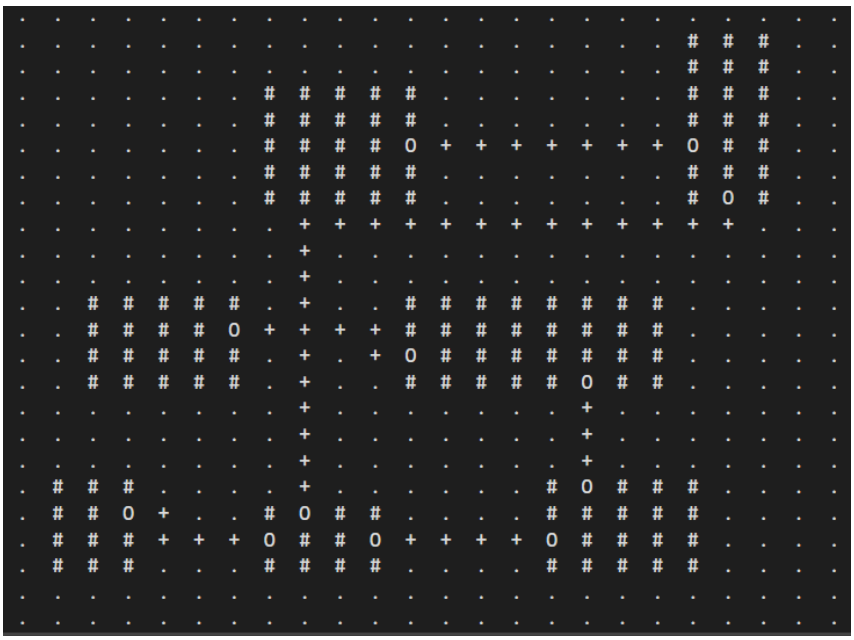

Figura 5.1: Exemplu de structură a nivelului generată de clasa `BSP`. Caracterul '.' marchează spațiul liber, '#' marchează spații aparținând unei camere, 'O' marchează intrări în camere iar '+' marchează coridoare.

5.2 RoomGenerator

Această componentă a algoritmului are ca scop generarea configurației tile-urilor pentru fiecare cameră și secțiune de drum din harta jocului.

Algoritmul debutează prin încărcarea în memorie a tuturor sprite-urilor și tile-urilor necesare procesului de generare a nivelului. În plus, sunt inițializate șabloanele corespunzătoare fiecărui tip distinct de secțiune de drum.

Spre deosebire de clasa `BSP`, care primește un set de parametri o singură dată și generează întreaga structură a nivelului, clasa `RoomGenerator` este concepută ca o colecție de funcții utilizate individual pentru fiecare cameră și secțiune de drum, la inițiativa clasei `LevelGenerationController`.

Atribute principale:

- Diverse array-uri de sprite-uri, utilizate pentru încărcarea în memorie a tileset-urilor necesare generării nivelurilor.
- `internal Dictionary<char, Sprite> tileDict` – dicționar care realizează corespondența dintre cheile stocate în matrici de configurație sau în șabloane și sprite-urile încărcate în memorie.

- `internal Dictionary<int, char[,]> roadWallsDict` și `internal Dictionary<int, char[,]> roadFloorDict` – dicționare care conțin șabloane pentru generarea pereților și podelelor din secțiunile de drum.

- `public Tilemap backgroundTilemap`, `public Tilemap wallTilemap` și `public Tilemap floorTilemap` – referințe către cele trei tilemap-uri pe care sunt plasate toate tile-urile în timpul procesului de generare. Dintre acestea, `wallTilemap` include coliziuni pentru toate tile-urile, permițând implementarea coliziunii pentru pereți într-un mod eficient și cu un consum redus de resurse.

- `internal int mapHeight` – înălțimea efectivă a hărții (considerată pătrată), necesară pentru conversia coordonatelor din formatul matricial utilizat pentru configurarea nivelului în formatul specific motorului Unity.

Funcții principale:

- `internal void InitializeTileDictionaries()` – încarcă în memorie tileset-urile necesare și inițializează dicționarul de sprite-uri, precum și dicționarele care conțin șabloanele utilizate pentru generarea nivelului.

- `public void PlaceMatrixOnTilemap(char[,] matrix, Tilemap targetTilemap, Vector2Int roomPosition)` – primește o matrice de caractere corespunzătoare unei configurații de sprite-uri și plasează tile-urile asociate pe tilemap-ul specificat, începând de la poziția indicată. Conversia coordonatelor din spațiul logic (matrice de caractere) în coordonatele motorului Unity se realizează cu ajutorul funcției auxiliare `internal Vector3 GetUnityPosition(float originalX, float originalY)`.

- `public char[,] GenerateRoomLayout(BSP.Leaf leaf, bool isFirst)` – generează structura unei camere pe baza unui nod frunză produs de algoritmul BSP. Inițial, spațiul este umplut cu tile-uri corespunzătoare unei zone libere, urmate de pereții exteriori ai camerei. Ulterior, cu excepția camerei inițiale (unde este plasat jucătorul), sunt generate procedural obstacole în interiorul camerei. În final, sunt create deschideri în pereți pentru a permite accesul în cameră.

- `bool IsAreaEmptyWithBuffer(char[,] matrix, int x, int y, int w, int h)` – verifică dacă există suficient spațiu liber într-o zonă a

camerei pentru a permite plasarea unui obstacol generat procedural, ținând cont și de un tampon de siguranță față de elementele învecinate.

- `public char[,] GenerateSpritesFloor(char[,] layout)` – primește layout-ul camerei, generat de funcția `GenerateRoomLayout()`, și produce matricea de caractere ce descrie configurația sprite-urilor pentru podea. Inițial, sunt plasate sprite-urile asociate pereților verticali, iar ulterior sunt decorate zonele libere ale podelei.

- `internal char[,] DecorateFloor(char[,] layout)` – funcție auxiliară utilizată pentru decorarea procedurală a podelei unei camere, pe baza layout-ului acesteia.

- `public char[,] GenerateSpritesWalls(char[,] layout)` – generează matricea de caractere corespunzătoare sprite-urilor pereților, pe baza configurației camerei obținută anterior. Pentru fiecare tile marcat ca perete, sunt verificați vecinii pentru a determina configurația locală, iar în funcție de aceasta, se selectează sprite-ul adecvat (de exemplu, pentru o intersecție de colț între un perete vertical și unul orizontal). Sunt aplicate verificări negative pentru a identifica situații în care pereții formează intrări sau colțuri aflate la marginea matricii.

- `private bool HasNeighbor(char[,] matrix, int x, int y, int dx, int dy, char target)` – funcție auxiliară pentru generarea pereților; verifică existența unui vecin specific în direcția indicată, fără riscul de a accesa indecși în afara limitelor matricei.

5.3 LevelGenerationController

Clasa `LevelGenerationController` reprezintă componenta centrală a algoritmului de generare procedurală a nivelului. Aceasta integrează funcționalitățile celorlalte clase pentru a construi nivelul complet, începând cu generarea structurii acestuia prin intermediul clasei `BSP`, continuând cu plasarea fundalului și a șabloanelor adecvate de drumuri în regiunile corespunzătoare, generarea și poziționarea camerelor și a inamicilor în spațiul de joc, și finalizând cu plasarea jucătorului în camera de început.

Totodată, această clasă este responsabilă de verificarea și aplicarea constrângerilor impuse algoritmului, precum și de regenerarea nivelului în situațiile în care acestea nu sunt respectate.

Atribute principale:

- Referințe către clasele `BSP` și `RoomGenerator` necesare pentru coordonarea algoritmului de generare a nivelului.
- `private Dictionary<int, int[]> level` – dicționar ce conține parametrii necesari generării tuturor celor 5 niveluri și ale nivelurilor de test.
- `public char[,] layout` – rezultatul rulării algoritmului BSP, care reprezintă întreaga structură a nivelului.
- Diverse referințe către prefab-urile corespunzătoare personajului principal și ale inamicilor pentru plasarea lor în spațiul de joc.

Funcții principale:

- `void InitializeLevelValues()` – inițializează în memorie toți parametrii necesari pentru generarea nivelurilor, asigurând consistența datelor utilizate în cadrul algoritmului.
- `void GenerateLevel(int selectedLevel)` – reprezintă funcția principală care coordonează generarea unui nivel complet. Procesul începe prin eliminarea tuturor tile-urilor existente de pe cele trei tilemap-uri utilizate, fie în contextul trecerii la un nivel nou, fie ca urmare a reluării generării din cauza unei constrângeri încălcate. Apoi, sunt efectuate secvențial următoarele etape: generarea layout-ului de bază, plasarea fundalului, determinarea și plasarea șabloanelor pentru drumuri, generarea și plasarea camerelor și a inamicilor în funcție de scorul de dificultate, precum și plasarea jucătorului în camera inițială. Întregul proces este încapsulat într-o structură `try-catch`, iar în cazul încălcării unei constrângeri, o excepție este aruncată, ceea ce determină reluarea întregului algoritm. Un mecanism de siguranță (*failsafe*) este inclus pentru a preveni apariția unei bucle infinite, posibilă în cazul unor parametri incorect configurați.
- `void GenerateLayout(int selectedLevel)` – apelează funcția `GenerateDungeon()` din cadrul clasei `BSP` și parcurge nodurile arborelui

rezultat pentru a verifica respectarea unor constrângeri suplimentare. În cazul în care acestea sunt încălcate, este aruncată o excepție care determină reluarea algoritmului.

- `bool IsCorner(Rect room, Vector2Int pos)` și `bool IsOutside(Rect room, Vector2Int pos)` – funcții auxiliare utilizate pentru verificarea respectării constrângerilor privind poziționarea intrărilor în camere.

- `void PlaceBackground()` – creează o matrice de dimensiunea spațiului de joc extinsă cu o zonă de siguranță (tampon), umplută cu tile-uri negre, și o plasează pe tilemap-ul de fundal.

- `void PlaceCorridors()` – parcurge întreaga matrice de layout și, pentru fiecare secțiune de drum, analizează vecinătățile folosind funcția `HasNeighbor()`, pentru a determina legăturile cu alte drumuri adiacente. Apoi, în funcție de topologia locală, selectează șablonul corespunzător și decorează tile-urile de podea înainte de a le plasa efectiv în joc.

- `void PlaceRooms(int targetDifficulty)` – parcurge toate nodurile frunză generate de algoritmul `BSP`, generează și decorează layout-ul fiecărei camere (pereți și podea), iar pentru camerele diferite de cea inițială, apelează funcția `SpawnEnemies()`.

- `void SpawnEnemies(Leaf l, char[,] layout, int targetDifficulty)` – determină dificultatea camerei pe baza dimensiunii sale (prin `GetSizeRating()`) și ajustează această valoare în funcție de gradul de acoperire oferit de obstacole (`ApplyCoverReduction()`). Apoi, adaugă inamici succesiv folosind `SpawnEnemy()` până la atingerea dificultății țintă.

- `List<Vector2Int> GetAllFloorTiles(char[,] layout)` – identifică toate tile-urile de podea disponibile în cameră, pentru a permite plasarea aleatorie a inamicilor.

- `float GetSizeRating(float roomSize, float minSize, float maxSize)` – calculează un scor de dificultate pentru o cameră, pornind de la dimensiunea acesteia. Mai întâi, dimensiunea camerei este normalizată într-un

interval standardizat $[0, 1]$. Această normalizare permite evaluarea relativă a dimensiunii camerei în raport cu valorile minime și maxime posibile. Ulterior, se aplică o funcție de rădăcină pătrată asupra valorii normalizate, pentru a introduce un efect de randament descrescător:

$$\text{difficulty} = 100 - 50 \cdot \sqrt{\frac{\text{roomSize} - \text{minSize}}{\text{maxSize} - \text{minSize}}}$$

- `float ApplyCoverReduction(float baseDifficulty, char[,] layout)` – aplică o funcție de ajustare bazată pe gradul de acoperire (cover) din interiorul camerei. Această ajustare penalizează camerele care oferă prea multă protecție, reducând scorul de dificultate într-un mod gradual. Procentul de acoperire este ridicat la pătrat, rezultând un impact nonlinear ce accentuează efectul zonelor excesiv de protejate. Reducerea este proporțională cu acest impact și limitată la o valoare maximă, după care se aplică o limitare inferioară asupra dificultății finale și o normalizare în intervalul $[0, 1]$:

$$\text{finalDifficulty} = \frac{\max\left(\text{baseDifficulty} - \left(\text{coverPercentage}^2 \cdot \text{maxReduction}\right),\ 30\right)}{100}$$

- `float SpawnEnemy(float currentDifficulty, float multiplier, List<Vector2Int> floorTiles, float roomX, float roomY)` – selectează aleator un tip de inamic din lista disponibilă, îl plasează într-o poziție liberă aleatorie din cameră și adaugă scorul de dificultate corespunzător (ajustat cu multiplicatorul) la totalul camerei.

- `void SpawnPlayer()` – localizează centrul camerei de început și plasează jucătorul în poziția corespunzătoare în spațiul de joc.

Figura 5.2: Exemplu de generare a nivelului procedural, corespunzător structurii din **Figura 5.1**

Concluzii

Lucrarea de față a demonstrat, atât teoretic cât și practic, modul în care pot fi îmbinate concepte fundamentale din game design și informatică pentru realizarea unui joc video coerent, interactiv și rejucabil. Proiectul propus — un joc 2D de tip dungeon crawler cu niveluri generate procedural — își propune să ofere o experiență de joc intensă, concentrată în sesiuni scurte, dar variate, punând accent pe precizie, mobilitate și luarea deciziilor în condiții de presiune.

Pe parcursul dezvoltării, s-a urmărit crearea unui sistem modular, extensibil și ușor de întreținut, prin utilizarea motorului de joc Unity și a limbajului de programare C#. De asemenea, s-au aplicat principii de design algoritmic pentru generarea nivelurilor, prin intermediul metodei Binary Space Partitioning (BSP), adaptată specificului jocului. Aceasta a permis construirea unui sistem flexibil de creare a hărților, capabil să susțină progresia în dificultate și să ofere diversitate vizuală și tactică.

Totodată, au fost implementate mecanici inspirate din genurile souls-like și roguelike, axate pe gestionarea resurselor, sincronizarea acțiunilor și evoluția abilităților jucătorului. Componenta de combat a fost proiectată să recompenseze planificarea și să penalizeze abuzul de acțiuni, încurajând astfel un stil de joc atent și reactiv.

Prin integrarea coerentă a acestor componente, proiectul atinge obiectivul propus: acela de a crea un prototip jucabil care să exploreze echilibrul dintre complexitate tehnică, diversitate a gameplay-ului și accesibilitate. Lucrarea confirmă viabilitatea utilizării generării procedurale ca mecanism central într-un joc modern și oferă o bază solidă pentru extinderea viitoare a conceptului.

Bibliografie

1. RogueBasin, https://roguebasin.com
2. Wikipedia, https://en.wikipedia.org/
3. Unity Technologies, *Unity Manual* - https://docs.unity3d.com/Manual/index.html
4. Unity Technologies, *Scripting API* - https://docs.unity3d.com/ScriptReference/
5. Unity Technologies, *Scripting in Unity* - https://docs.unity3d.com/Manual/ScriptingSection.html
6. GitHub Docs, *About GitHub* - https://docs.github.com/en/get-started/quickstart/github-overview
7. Notion, *What is Notion?* - https://www.notion.com/help/guides/what-is-notion
8. Schumacker R. A., Brand B. E., Gilliland M. C., Sharp W. R., Warnock J. E., *Study for applying the hardware required for generation and display of three-dimensional images*, 1969
9. Herbert Wolverson, *Procedural Map Generation Techniques* - https://www.youtube.com/watch?v=TlLIOgWYVpI
10. CELSYS, *Clip Studio Paint — Digital Painting and Comics Software* - https://www.clipstudio.net/en/
11. Comunitatea Aseprite, *Aseprite Importer for Unity* - https://github.com/aseprite/aseprite/tree/main/docs

12. Unity Technologies, *2D Pixel Perfect Package Documentation* - https://docs.unity3d.com/Manual/com.unity.2d.pixel-perfect.html

13. Unity Technologies, *PSD Importer Package Documentation* - https://docs.unity3d.com/Manual/com.unity.2d.psdimporter.html

14. Unity Technologies, *Input System Package Documentation* - https://docs.unity3d.com/Packages/com.unity.inputsystem@1.14/manual/index.html