MINISTERUL EDUCAȚIEI ȘI CERCETĂRII AL REPUBLICII MOLDOVA

UNIVERSITATEA DE STAT „ALECU RUSSO” DIN BĂLȚI

FACULTATEA DE ȘTIINȚE REALE, ECONOMICE ȘI ALE MEDIULUI

CATEDRA DE MATEMATICĂ ȘI INFORMATICĂ

UTILIZAREA PLATFORMELOR GODOT ȘI UNITY ÎN CREAREA JOCURILOR

TEZĂ DE MASTER

Autor:
Studentul grupei AW21M
Dumitru CAZACU-CONDRAT

_____

Conducător științific:
Corina NEGARA
dr., conf. univ.

_____

BĂLȚI, 2025

Controlată:

Data_____

Conducător științific: Corina NEGARA, dr., conf. univ.

_____

Aprobată

și recomandată pentru susținere

la ședința Catedrei de matematică și informatică

Proces-verbal nr._____din_____

Șeful Catedrei de matematică și informatică

Asistent universitar: Vitalie ȚÎCĂU

_____

Aprobat

Șeful Catedrei de matematică și informatică

Asistent univ. V. Țîcău

„___” „_____” 2024

Graficul calendaristic de executare a tezei de master

Tema tezei de master: Dezvoltarea și implementarea unui sistem de evaluare al elevilor

confirmată prin ordinul rectorului USARB nr.___ din „_____”

Termenul limită de prezentare a tezei de master la Catedra de matematică și informatică „___” „_____” 2025.

Etapele executării tezei de master:

№	Etapele	Termenul de realizare	Viza de executare
1.	Stabilirea obiectivelor și aprobarea planului inițial al tezei de master	16.09.2024	Realizat
2.	Studierea surselor bibliografice	14.10.2024	Realizat
3.	Aprobarea planului tezei	28.10.2024	Realizat
4.	Colectarea materialelor practice	30.12.2024	Realizat
5.	Elaborarea și prezentarea capitolului teoretic	27.01.2025	Realizat
6.	Elaborarea și prezentarea capitolului experimental	21.02.2025	Realizat
7.	Prezentarea variantei finale a tezei	27.03.2025	Realizat
8.	Susținerea prealabilă a tezei de master	02.05.2025	Realizat
9.	Prezentarea tezei de master la catedră	10.05.2025	Realizat

Masterand _____

(semnătura)

Conducător științific _____

(semnătura)

ADNOTARE

Cazacu-Condrat Dumitru

UTILIZAREA PLATFORMELOR GODOT ȘI UNITY ÎN CREAREA JOCURILOR

Teză de master. Bălți 2025

Structura tezei: introducere, patru capitole, concluzii generale, bibliografie din 37 referințe, 52 pagini de text, 2 tabele și 13 figuri.

Cuvintele cheie: jocuri, motor grafic, Godot, Unity, C#, GDScript, 2D, 3D, game design, gameplay.

Domeniul de studii: dezvoltarea aplicațiilor interactive și jocurilor video utilizând tehnologiile*: Godot Engine, Unity, C#, GDScript*.

Scopul lucrări: realizarea unei analize comparative a platformelor Godot și Unity în procesul de dezvoltare a jocurilor video, prin cercetare teoretică și implementare practică, pentru a oferi dezvoltatorilor un cadru decizional fundamentat.

Obiectivele lucrării:

1. Cercetarea și analiza surselor bibliografice privind evoluția și arhitectura motoarelor de joc în general și a platformelor Unity și Godot în particular.
2. Analiza comparativă a specificațiilor tehnice, funcționalităților și limitărilor motoarelor Unity și Godot.
3. Cercetarea limbajelor de programare și a instrumentelor specifice fiecărei platforme.
4. Proiectarea arhitecturii unui joc care poate fi implementat pe ambele platforme.
5. Proiectarea și implementarea unui joc demonstrativ în ambele platforme pentru evaluarea directă a procesului de dezvoltare.
6. Măsurarea și compararea performanței tehnice a jocurilor realizate pe ambele platforme.
7. Elaborarea unui cadru decizional pentru ghidarea alegerii motorului optim în funcție de specificul proiectului.

Metodologia cercetării s-a bazat pe metode teoretice: documentare științifică, analiză comparativă, sinteză, interpretare, precum și metode practice: prototipare, testare, măsurarea performanței, dezvoltare software.

Semnificația teoretică a cercetării: lucrarea contribuie la domeniul dezvoltării jocurilor prin analiza detaliată a arhitecturilor și capacităților tehnice ale motoarelor Godot și Unity, oferind un cadru teoretic pentru selecția optimă a platformei în funcție de cerințele proiectului.

Valoarea aplicativă a lucrării constă în implementarea practică a unui joc pe ambele platforme, care poate servi ca material educațional pentru dezvoltatorii începători și avansați, precum și ca referință pentru alegerea platformei optime în funcție de tipul și bugetul proiectului.

ANNOTATION

Cazacu-Condrat Dumitru

USING GODOT AND UNITY PLATFORMS IN GAME DEVELOPMENT

Master's degree. Bălți 2025

Structure of thesis: introduction, three chapters, general conclusions, bibliography of 37 titles, 52 pages of basic text, 2 tables and 13 figures.

Keywords: game development, game engine, Godot Engine, Unity, C#, GDScript, 2D, 3D, game design, gameplay.

Field of study: development of interactive applications and video games using technologies: Godot Engine, Unity, C#, GDScript.

Research goal: conducting a comparative analysis of the Godot and Unity platforms in the video game development process, through theoretical research and practical implementation, to provide developers with a substantiated decision-making framework.

Objectives of the work: (1) Research and analysis of bibliographic sources regarding the evolution and architecture of game engines in general and the Unity and Godot platforms in particular. (2) Comparative analysis of technical specifications, functionalities, and limitations of Unity and Godot engines. (3) Research of programming languages and specific tools for each platform. (4) Designing the architecture of a game that can be implemented on both platforms. (5) Design and implementation of a demonstrative game on both platforms for direct evaluation of the development process. (6) Measurement and comparison of the technical performance of games created on both platforms. (7) Development of a decision-making framework to guide the choice of the optimal engine based on project specifics.

The methodology of the research was based on theoretical methods: scientific documentation, comparative analysis, synthesis, interpretation, as well as practical methods: prototyping, testing, performance measurement, software development.

The theoretical significance of the research the work contributes to the field of game development through detailed analysis of the architectures and technical capabilities of Godot and Unity engines, providing a theoretical framework for optimal platform selection based on project requirements.

The practical value of the study application consists of the practical implementation of a game on both platforms, which can serve as educational material for beginner and advanced developers, as well as a reference for choosing the optimal platform depending on the type and budget of the project

CUPRINS

INTRODUCERE

Dezvoltarea jocurilor video reprezintă un domeniu complex în continuă evoluție, care îmbină elemente de programare, design grafic, storytelling și inginerie software. În acest context, motoarele de joc sunt instrumente esențiale care permit creatorilor să implementeze viziunea lor într-un produs final funcțional. Aceste platforme au mai multe roluri fundamentale în industria gamedev: accelerează procesul de dezvoltare, asigură optimizarea performanței, facilitează implementarea mecanicilor complexe de joc, permit portarea pe multiple dispozitive și oferă acces la un ecosistem de resurse și instrumente specializate.

În contextul actual, studierea comparativă a motoarelor de joc Godot și Unity devine tot mai relevantă și necesară, mai ales în condițiile diversificării peisajului tehnologic și a cerințelor variate ale dezvoltatorilor. Iată câteva aspecte care evidențiază actualitatea acestei teme:

a) Democratizarea dezvoltării jocurilor: Există o tendință crescândă de democratizare a procesului de creare a jocurilor, cu un număr tot mai mare de dezvoltatori independenți și studiouri mici care caută soluții accesibile dar puternice. Personalizarea și individualizarea învățământului: Sistemele de evaluare pot juca un rol important în personalizarea și individualizarea învățământului, oferind feedback specific și adaptat nevoilor fiecărui elev și identificând punctele lor tari și slabe pentru a oferi suportul necesar.
b) Evoluția modelelor de licențiere: Schimbările recente în politicile de licențiere ale unor motoare de joc consacrate au determinat mulți dezvoltatori să exploreze alternative viabile, crescând interesul pentru comparații obiective între platforme.
c) Creșterea complexității proiectelor: Pe măsură ce așteptările consumatorilor cresc, dezvoltatorii au nevoie de platforme care pot susține proiecte tot mai complexe, fiind necesară o evaluare riguroasă a capacităților tehnice ale motoarelor disponibile..
d) Diversificarea platformelor țintă: Expansiunea pieței de jocuri pe mobile, VR/AR, cloud gaming și platforme emergente necesită motoare versatile cu capacități robuste de portare multi-platform.
e) Evoluția tehnologică rapidă: Implementarea noilor tehnologii precum ray-tracing, machine learning și simulări fizice avansate ridică întrebări privind capacitatea diferitelor motoare de a ține pasul cu inovația.
f) Considerente economice și de eficiență: În contextul economic actual, studios și dezvoltatorii independenți analizează cu atenție raportul cost-beneficiu și eficiența dezvoltării pe diferite platforme.
g) Nevoia de specializare educațională: Instituțiile educaționale și programele de formare profesională caută informații obiective pentru a decide ce tehnologii să includă în curricula lor.

Reieșind din actualitatea studiului, putem evidenția și problema care constă în lipsa unor analize comparative comprehensive, bazate pe criterii științifice și experimente practice, care să ghideze dezvoltatorii în alegerea platformei optime pentru proiectele lor specifice. Majoritatea comparațiilor existente sunt fie subiective, fie limitate la aspecte particulare, fără a oferi o viziune holistică asupra avantajelor și dezavantajelor fiecărei platforme. Acest instrument propus are scopul de a uni diverse domenii de activitate, de la profesional la creativ, pentru a oferi o soluție cuprinzătoare și adaptată nevoilor complexe ale profesorilor.

Această lucrare propune o abordare metodică care îmbină analiza teoretică cu implementarea practică pentru a oferi o soluție cuprinzătoare și adaptată nevoilor complexe ale dezvoltatorilor de jocuri. Aplicația propusă nu doar va simplifica procesele, ci și va contribui la crearea unei platforme fiabile și eficiente pentru gestionarea activităților elevilor. În plus, dezvoltarea experimentală a jocurilor identice pe ambele platforme este justificată prin capacitatea sa de a evidenția diferențele concrete de performanță, productivitate și calitate a rezultatelor finale, furnizând date obiective dincolo de specificațiile teoretice.

De asemenea, se evidențiază că multe comparații existente între motoarele de joc se concentrează pe aspecte izolate (precum performanța grafică sau facilitatea de utilizare), fără a lua în considerare întregul ciclu de dezvoltare și toate aspectele relevante pentru diferite tipuri de proiecte. Limitările în metodologia de comparare, precum și lipsa unei perspective multi-criteriale pot conduce la concluzii incomplete sau inadecvate pentru anumite contexte de dezvoltare.

Prin urmare, această lucrare se distinge prin intenția de a depăși aceste deficiențe și de a oferi o analiză comparativă comprehensivă, riguroasă și echilibrată a platformelor Unity și Godot.

Scopul lucrării constă în realizarea unei analize comparative a platformelor Godot și Unity în procesul de dezvoltare a jocurilor video, prin cercetare teoretică și implementare practică, pentru a oferi dezvoltatorilor un cadru decizional fundamentat.

Pentru atingerea acestui scop au fost identificate următoarele *obiective:*

1. Cercetarea și analiza surselor bibliografice privind evoluția și arhitectura motoarelor de joc în general și a platformelor Unity și Godot în particular.
2. Analiza comparativă a specificațiilor tehnice, funcționalităților și limitărilor motoarelor Unity și Godot.
3. Cercetarea limbajelor de programare și a instrumentelor specifice fiecărei platforme.
4. Proiectarea arhitecturii unui joc care poate fi implementat pe ambele platforme.
5. Proiectarea și implementarea unui joc demonstrativ în ambele platforme pentru evaluarea directă a procesului de dezvoltare.
6. Măsurarea și compararea performanței tehnice a jocurilor realizate pe ambele platforme.

7. Elaborarea unui cadru decizional pentru ghidarea alegerii motorului optim în funcție de specificul proiectului.

Valoarea teoretică a lucrării constă în sistematizarea cunoștințelor despre arhitectura și funcționalitatea motoarelor de joc contemporane, dezvoltarea unei metodologii de evaluare comparativă a platformelor de dezvoltare și contribuția la fundamentarea teoretică a deciziilor tehnologice în domeniul dezvoltării jocurilor.

Valoarea practică a lucrării constă în furnizarea unor date concrete privind performanța, productivitatea și calitatea rezultatelor obținute pe cele două platforme, oferind dezvoltatorilor un instrument decizional validat experimental și prezentarea unor soluții concrete pentru provocările specifice întâlnite în procesul de dezvoltare pe fiecare platformă,

Structura tezei de master reflectă logica cercetării și rezultatele acesteia. Ea constă dintr-o introducere, trei capitole, o concluzie și o bibliografie.

Primul capitol prezintă fundamentele teoretice în dezvoltarea jocurilor video, analizând evoluția istorică a industriei, arhitectura și componentele principale ale unui motor de joc, tendințele contemporane, impactul economic și rolul motoarelor de joc în procesul de dezvoltare.

Al doilea capitol se concentrează pe analiza tehnică a platformei Unity, examinând arhitectura și funcționalitățile sale, programarea în C# și integrarea scripturilor, sistemul de fizică și detectarea coliziunilor, managementul resurselor și optimizarea performanței, precum și studii de caz reprezentative.

Al treilea capitol oferă o analiză tehnică a platformei Godot, cu accent pe arhitectura și specificațiile tehnice, limbajul GDScript și paradigma orientată pe noduri, sistemul de semnale, funcționalitățile avansate și optimizarea performanței, precum și studii de caz relevante. La fel la sfârșit a fot realizat un tabel care prezintă o analiză comparativă a celor două platforme.

Al patrulea capitol descrie partea practică a lucrării, structura și funcționalitatea acesteia.

Teza de master este prezentată în următorul volum, unde este descris conținutul principal pe de 41 pagini, inclusiv 11 figuri, 2 tabele și o bibliografie de referințe.

1. FUNDAMENTE TEORETICE ÎN DEZVOLTAREA JOCURILOR VIDEO

1.1. Evoluția istorică a industriei jocurilor video

Evoluția industriei jocurilor video poate fi structurată în etape distincte, relevante pentru înțelegerea contextului actual în care operează platformele moderne de dezvoltare. Începând cu experimentele timpurii din anii 1950-1960, precum "Spacewar!" dezvoltat la MIT în 1962, industria a evoluat de la simple demonstrații tehnologice la un sector economic global cu o valoare estimată la peste 200 miliarde USD în 2024 [1].

Prima generație de console de jocuri (1972-1977), marcată de apariția sistemului Magnavox Odyssey și a arcade-ului Pong creat de Atari, a stabilit fundațiile unei industrii emergente. Aceste sisteme primitive utilizau circuite dedicate, fără software modificabil, fiecare joc necesitând hardware specific. Criza jocurilor video din 1983 a restructurat fundamental industria, conducând la apariția consolelor de a treia generație, precum Nintendo Entertainment System (NES), care au introdus standarde de calitate și au consolidat modelul de publicare controlată [2].

Evoluția tehnologică a continuat accelerat, generațiile succesive de hardware aducând îmbunătățiri semnificative în capacitățile grafice și computaționale. Această progresie a creat necesitatea unor instrumente de dezvoltare din ce în ce mai sofisticate, capabile să abstractizeze complexitatea crescândă a platformelor hardware.

Apariția primelor motoare de joc comerciale în anii 1990, precum Doom Engine și Quake Engine dezvoltate de id Software, a reprezentat un moment crucial, separând pentru prima dată componentele reutilizabile ale jocului (motorul) de conținutul specific (artă, nivele, reguli). Această separare a facilitat dezvoltarea unor ecosisteme specializate și a pus bazele modelului modern de motoare de joc ca platforme independente de dezvoltare [3].

Perioada anilor 2000 a marcat tranziția către motoare de joc universale, precum Unreal Engine (lansat inițial în 1998) și Unity (2005), concepute pentru a deservi o gamă largă de proiecte și genuri. Democratizarea tehnologiei, accelerată de modelele de licențiere accesibile și de distribuția digitală, a transformat fundamental peisajul industriei [3]. Apariția mai recentă a motorului open-source Godot în 2014 reprezintă o continuare a acestei tendințe de democratizare, oferind o alternativă complet liberă platformelor comerciale existente.

Contextul actual este caracterizat de coexistența mai multor ecosisteme majore de dezvoltare, fiecare cu propriile avantaje competitive și comunități asociate. Această diversificare a instrumentelor disponibile a redus semnificativ barierele de intrare în industrie, contribuind la explozia dezvoltării independente și la diversificarea tipurilor de jocuri create [3].

1.2. Arhitectura și componentele principale ale unui motor de joc

Motoarele de joc moderne reprezintă sisteme software complexe, structurate modular pentru a facilita dezvoltarea eficientă a aplicațiilor interactive. Arhitectura tipică a unui motor de joc contemporan încorporează multiple subsisteme specializate, organizate într-o ierarhie funcțională care abstractizează diversele aspecte ale dezvoltării. La baza acestei arhitecturi se află stratul de abstractizare hardware, responsabil pentru interfațarea cu sistemele de operare și componentele fizice ale dispozitivului. Acest nivel gestionează accesul la resurse de sistem precum memoria, procesorul, dispozitivele de intrare și unitățile de procesare grafică, oferind un nivel de izolare față de specificațiile tehnice ale platformelor țintă [4]. Peste acest fundament se construiesc subsistemele specializate care implementează funcționalitățile esențiale pentru dezvoltarea jocurilor:

1. Gestiunea Sistemul de randare (Rendering System): responsabil pentru generarea output-ului vizual, acest subsistem gestionează geometria 3D, texturarea, iluminarea, efectele post-procesare și pipeline-ul grafic. Implementările moderne utilizează frecvent API-uri grafice precum OpenGL, DirectX, Vulkan sau Metal, abstractizând complexitatea acestora prin interfețe unificate [5].
2. Motorul de fizică (Physics Engine): simulează interacțiunile fizice ale obiectelor din lumea virtuală, incluzând detectarea coliziunilor, dinamica corpurilor rigide, articulații și constrângeri, fluide și sisteme de particule. Motoarele moderne integrează frecvent soluții specializate precum PhysX, Box2D sau Bullet Physics [6].
3. Sistemul audio: gestionează playback-ul, mixarea și procesarea efectelor sonore și a muzicii, incluzând frecvent funcționalități de spațializare 3D și ocluzie acustică pentru experiențe imersive.
4. Sistemul de animație: controlează mișcarea și transformarea modelelor, implementând tehnici precum animația scheletală, blend trees, state machines și animații procedurale.
5. Sistemul de inteligență artificială: oferă mecanisme pentru comportamentul entităților non-player, incluzând pathfinding, decision making, mașini cu stări finite și alte algoritmi specifici.
6. Sistemul de intrare (Input System): abstractizează procesarea input-ului de la diverse dispozitive (tastatură, mouse, controller, touchscreen), oferind o interfață uniformă pentru captarea interacțiunii utilizatorului.
7. Sistemul de management al resurselor: gestionează încărcarea, descărcarea și caching-ul asseturilor (modele 3D, texturi, sunete, etc.), optimizând utilizarea memoriei și minimizând timpii de încărcare.

Un aspect distinctiv al motoarelor moderne este suportul pentru workflow-uri vizuale prin intermediul editorului integrat, care permite manipularea directă a scenelor, entităților și componentelor, reducând necesitatea programării explicite pentru configurare și poziționare [7].

1.3. Tendințe contemporane în dezvoltarea jocurilor

Industria dezvoltării jocurilor video traversează o perioadă de transformare rapidă, marcată de multiple tendințe tehnologice și metodologice care reconfigurează practicile stabilite. Aceste evoluții influențează direct cerințele impuse motoarelor de joc și, implicit, deciziile legate de selecția platformei de dezvoltare [8].

Adopția tehnologiilor emergente reprezintă un vector important de transformare. Integrarea tehnicilor de machine learning și inteligență artificială generativă în procesul de dezvoltare facilitează automatizarea parțială a creării de conținut, de la generarea texturii la comportamente adaptative ale NPC-urilor. Motoarele de joc evoluează pentru a incorpora nativ aceste capacități, Unity implementând ML-Agents ca framework specializat, în timp ce Godot dezvoltă integrări open-source pentru biblioteci precum TensorFlow [8].

Cross-platform development (dezvoltarea multi-platformă) a devenit un standard industrial, determinat de fragmentarea crescândă a ecosistemului de dispozitive. Capacitatea de a compila și optimiza pentru multiple platforme țintă (PC, console, mobile, VR/AR, web) dintr-o singură bază de cod reprezintă un criteriu esențial în evaluarea motoarelor de joc [9]. Atât Unity cât și Godot oferă capabilități extensive de export multi-platformă, deși cu diferențe semnificative în maturitatea implementărilor pentru anumite platforme specifice.

Metodologiile agile și DevOps au transformat procesele de producție, accelerând ciclurile de dezvoltare și facilitând actualizarea continuă a produselor existente. Integrarea continuă, deployment automatizat și instrumentele de colaborare au devenit componente esențiale ale ecosistemelor de dezvoltare moderne [10]. Maturitatea instrumentelor de colaborare și integrarea cu sisteme de version control reprezintă factori decisionali importanți în selecția platformei.

Live service games (jocuri ca serviciu) au redefinit modelele de monetizare și ciclurile de viață ale produselor, necesitând arhitecturi software care facilitează actualizarea continuă a conținutului și integrarea serviciilor online. Capacitatea motorului de a gestiona eficient updates incrementale și de a interacționa cu backend services influențează semnificativ alegerea platformei pentru proiecte cu componentă multiplayer semnificativă [11].

Dezvoltarea indie și democratizarea tehnologiei continuă să transforme peisajul industrial, micii dezvoltatori independent având acces la instrumente anterior disponibile doar studiourilor AAA. Această democratizare a condus la o explozie de creativitate și experimentare, dar și la

saturarea pieței și intensificarea competiției [12]. Accesibilitatea economică și curbele de învățare ale motoarelor reprezintă factori critici pentru dezvoltatorii cu resurse limitate.

Realitatea virtuală și augmentată au evoluat din nișe experimentale în segmente de piață distincte, impunând cerințe specifice legate de performanță, interactivitate și design. Suportul nativ pentru aceste tehnologii și optimizările specifice pentru experiențe imersive au devenit diferențiatori importanți între platformele de dezvoltare [13].

1.4. Impactul economic al industriei jocurilor video

Industria jocurilor video s-a transformat dintr-un sector de nișă într-unul dintre cele mai dinamice și profitabile segmente ale economiei globale. Cu o valoare estimată la peste 200 miliarde USD în 2022 și o prognoză de creștere la aproximativ 285 miliarde USD până în 2027, această industrie depășește în prezent cinematografia și muzica combinate ca valoare de piață [14]. Acest impact economic substanțial influențează direct ecosistemul tehnologic al dezvoltării jocurilor, inclusiv evoluția și adoptarea motoarelor de joc.

Structura veniturilor în industria jocurilor a suferit transformări fundamentale în ultimul deceniu. Modelul tradițional de vânzare unică (pay-to-play) a fost completat sau chiar înlocuit de modele alternative precum free-to-play cu microtransacții, abonamente, season passes și conținut descărcabil (DLC). Această diversificare a strategiilor de monetizare impune cerințe specifice motoarelor de joc, de la suport pentru sisteme complexe de economie virtuală la integrarea perfecă cu platforme de distribuție și procesare a plăților.

Segmentarea pieței reflectă o diversificare continuă, cu secțiuni distincte pentru jocuri pe console, PC, mobile, cloud gaming și realitate virtuală/augmentată. Fiecare segment prezintă caracteristici tehnice și economice specifice, influențând direct criteriile de selecție pentru platformele de dezvoltare. Segmentul mobile, de exemplu, care reprezintă peste 50% din piața globală, necesită optimizări agresive pentru performanță și consum energetic, în timp ce segmentul AAA pe console și PC prioritizează fidelitatea vizuală și complexitatea sistemelor [15].

Distribuția geografică a dezvoltării jocurilor s-a modificat substanțial, cu apariția unor hub-uri semnificative în regiuni precum Europa de Est, America Latină și Asia de Sud-Est, pe lângă centrele tradiționale din America de Nord, Japonia și Europa Occidentală. Această globalizare a industriei a amplificat cererea pentru motoare de joc accesibile economic și cu documentație multilingvă, contribuind la popularitatea soluțiilor cu modele de licențiere flexibile și comunități internaționale active.

Modelele economice ale motoarelor de joc însele reprezintă un factor important în ecosistemul industrial. Unity Technologies a adoptat inițial un model freemium cu licențe plătite pentru venituri

peste anumite praguri, evoluând ulterior către un sistem de abonamente diferențiate pe nivele. După controversele din 2023 legate de introducerea "Runtime Fee", compania a revenit la un model mai tradițional de licențiere [16]. În contrast, Godot Engine operează sub licența MIT, complet gratuită și open-source, finanțată prin donații, sponsorizări și granturi de la entități precum Blender Foundation și Mozilla Foundation [17].

1.5. Rolul motoarelor de joc în procesul de dezvoltare

Motoarele de joc moderne au evoluat dincolo de simpla funcție tehnică de randare și simulare, devenind platforme integrate care definesc fundamental procesul de dezvoltare. Acest rol extins influențează nu doar aspectele tehnice ale creației, ci și metodologiile de lucru, structura echipelor și fluxul general al producției.

Accelerarea dezvoltării reprezintă beneficiul primar al utilizării unui motor specializat. Prin abstractizarea funcționalităților complexe comune și oferirea unor sisteme pre-construite, motoarele reduc semnificativ timpul necesar implementării componentelor fundamentale. Utilizarea unui motor modern poate reduce timpul de dezvoltare semnificativ comparativ cu implementarea de la zero a funcționalităților echivalente.

Standardizarea și transferabilitatea competențelor reprezintă consecințe importante ale adoptării largi a motoarelor majore. Familiarizarea cu arhitectura și paradigmele specifice unei platforme precum Unity sau Godot creează un set de competențe transferabile între proiecte și chiar între companii, facilitând mobilitatea profesională și reducând timpul de onboarding pentru noii membri ai echipelor.

Cicluri iterative accelerate sunt facilitate de capacitățile de rapid prototyping implementate în motoarele moderne. Funcționalități precum hot-reloading, preview în timp real și instrumentele de debugging specializate permit testarea imediată a modificărilor și iterarea rapidă a designului.

Specializarea vs. versatilitatea reprezintă o alegere fundamentală în evaluarea rolului motoarelor. În timp ce unele motoare specializate optimizează workflow-ul pentru genuri specifice (simulatoare de curse, FPS, RPG), platformele generaliste precum Unity și Godot prioritizează versatilitatea și adaptabilitatea. Această tensiune influențează direct deciziile strategice ale studiourilor legate de selecția platformei în funcție de portofoliul anticipat de proiecte .

Comunitatea și ecosistemul extins asociate unui motor reprezintă resurse critice care transcend funcționalitățile tehnice. Disponibilitatea documentației, tutorialelor, forumurilor active și bibliotecilor de asseturi pre-construite accelerează semnificativ dezvoltarea și rezolvarea problemelor. Unity beneficiază de unul dintre cele mai extinse ecosisteme cu peste 11 milioane de dezvoltatori înregistrați, în timp ce Godot a dezvoltat o comunitate remarcabil de activă pentru un proiect mai recent [18].

2. PLATFORMA UNITY - ANALIZĂ TEHNICĂ ȘI APLICAȚII

2.1. Arhitectura și funcționalitățile platformei Unity

Unity reprezintă un ecosistem complex de dezvoltare, a cărui arhitectură reflectă evoluția sa continuă de la lansarea inițială în 2005. Structura fundamentală a platformei implementează o arhitectură componentă bazată pe GameObject-uri și Components, organizată într-o ierarhie care facilitează modularitatea și reutilizarea [19].

Arhitectura de bază a Unity este structurată în jurul conceptului de scene ca containere principale pentru conținut. Fiecare scenă reprezintă un spațiu distinct care poate conține un număr nelimitat de GameObject-uri - entități fundamentale care servesc ca containere pentru componente funcționale. Acest model component-based diferă de abordările tradiționale orientate pe obiecte prin separarea explicită a datelor (GameObject) de comportament (Components), facilitând compoziția flexibilă a entităților complexe din module reutilizabile [19].

Scriptingul în Unity se realizează predominant în C#, un limbaj puternic tipizat care oferă beneficiile programării orientate pe obiecte într-un context performant. Scripturile se atașează GameObject-urilor ca și componente, implementând comportamente specifice prin metode precum Start(), Update() și FixedUpdate(), care sunt invocate automat de motorul de execuție conform ciclului de viață predefinit. Această abordare simplă dar versatilă permite atât implementări rapide pentru dezvoltatori începători, cât și arhitecturi sofisticate pentru proiecte complexe [19].

Render Pipeline în Unity a evoluat semnificativ, oferind în prezent multiple opționale: Built-in Render Pipeline (legacy), Universal Render Pipeline (URP) pentru performanță și compatibilitate largă, și High Definition Render Pipeline (HDRP) pentru grafică avansată pe hardware performant. Această diversificare permite optimizarea pipeline-ului grafic pentru cerințele specifice ale proiectului și platformele țintă, de la dispozitive mobile la console high-end [20].

Subsistemele specializate implementează funcționalități esențiale pentru dezvoltarea modernă:

1. Terrain System: facilitează crearea și manipularea terenurilor vaste cu instrumente specializate pentru sculptare, texturare și vegetație procedurală.
2. Animation System: integrează rigging, blending și state machines într-un framework unificat (Mecanim), capabil să gestioneze animații complexe și tranzițiile între acestea.
3. Audio System: oferă mixing și spațializare 3D prin integrarea native cu middleware specializat precum FMOD.
4. Navigation System: implementează pathfinding și obstacle avoidance pentru entități autonome prin NavMesh și NavMesh Agents.

5. Physics Systems: integrează multiple motoare de fizică specializate pentru diferite scenarii: PhysX pentru simulare 3D comprehensivă, Box2D pentru fizică 2D optimizată și Cloth Physics pentru simularea țesăturilor.
6. Particle System: permite creația efectelor vizuale complexe prin manipularea masivă a particulelor cu parametri extensivi pentru emisie, mișcare și randare [21].

Unity Editor reprezintă interfața vizuală care agregează aceste subsisteme într-un mediu integrat de dezvoltare (IDE). Organizat în ferestre modulare customizabile (Scene View, Game View, Inspector, Hierarchy, Project), editorul facilitează manipularea directă a conținutului prin WYSIWYG și implementează un sistem extensibil de instrumente specializate. Această extensibilitate permite dezvoltarea de editor tools personalizate care optimizează workflow-ul pentru cerințele specifice ale proiectului [20].

2.2. Programarea în C# și integrarea scripturilor în Unity

Unity utilizează C# ca limbaj principal de scripting o decizie arhitecturală care influențează fundamental modelul de programare și capabilitățile platformei. Această alegere combină beneficiile unui limbaj mature, puternic tipizat și orientat pe obiecte cu optimizările specifice necesare pentru dezvoltarea jocurilor [20].

Integrarea cu .NET Framework oferă acces la un ecosistem extensiv de biblioteci și funcționalități standard. Unity a evoluat de la o implementare proprietară (Mono) către .NET Standard și ulterior către versiuni moderne ale .NET, îmbunătățind continuu compatibilitatea cu ecosistemul .NET și toolchain-ul asociat. Această evoluție a extins semnificativ resursele disponibile dezvoltatorilor, de la pattern-uri arhitecturale la biblioteci specializate [20].

Modelul de scripting în Unity implementează paradigma component-based prin intermediul claselor derivate din MonoBehaviour - clasa fundamentală care facilitează interacțiunea cu motorul. Acest model definește un ciclu de viață standardizat pentru scripturi, cu metode predefinite invocate automat în momente specifice:

1. Metode de inițializare: Awake() și Start() pentru setup inițial, cu diferențe subtile legate de momentul execuției.
2. Metode de actualizare: Update(), FixedUpdate() și LateUpdate() pentru logica care necesită execuție constantă.
3. Metode de callback: OnTriggerEnter(), OnCollisionExit() pentru reacția la evenimente de fizică.
4. Metode de UI: OnMouseDown(), OnButtonClick() pentru interacțiunea cu utilizatorul [46].

Comunicarea între componente utilizează predominant trei mecanisme distincte:

- Referințe directe obținute prin GetComponent<>();

- mesaje între componente prin SendMessage() și BroadcastMessage();
- evenimentele și delegatele native C# pentru implementări mai complexe și decuplate.

Această varietate de opțiuni permite arhitecturi adaptate complexității proiectului, de la abordări simple și directe pentru prototipare rapidă la implementări sofisticate cu coupling redus pentru proiecte extensive [20].

Programarea asincronă în Unity a evoluat semnificativ, integrând coroutines și async/await pentru gestionarea operațiunilor care se extind peste multiple frame-uri. Coroutines utilizează yield instructions specifice (WaitForSeconds, WaitForEndOfFrame) pentru a suspenda și relua execuția fără blocarea thread-ului principal, o abordare esențială pentru operațiuni precum animații procedurale, loading asincron și secvențierea evenimentelor [20].

Optimizarea performanței în scripturile C# necesită considerații specifice pentru contextul de gaming. Managementul memoriei reprezintă o provocare particulară, garbage collection-ul .NET putând genera stuttering vizibil dacă nu este gestionat corespunzător. Practicile recomandate includ:

- object pooling pentru entități frecvent instanțiate/distruse;
- minimizarea alocărilor în metode invocate frecvent (Update);
- utilizarea structurilor pentru tipuri de date cu lifetime scurt;
- implementarea cache-ing-ului pentru rezultate computaționale constante [20].

Extensia editorului reprezintă un aspect distinctiv al programării în Unity, permițând customizarea și automatizarea workflow-ului de dezvoltare. Custom Inspectors, Editor Windows și Custom Property Drawers facilitează crearea instrumentelor specializate pentru nevoile specifice ale proiectului. Aceste extensii utilizează un API dedicat (UnityEditor namespace) și sunt executate exclusiv în contextul editorului, fiind eliminate automat din build-urile finale [20].

Scriptable Objects oferă o soluție elegantă pentru gestionarea datelor persistente și partajate între scene. Aceste containere scriptabile persistă independent de ierarhia GameObject și facilitează arhitecturi data-driven care separă explicit logica de conținut. Utilizările comune includ configurații globale, inventare și sisteme de quest-uri, reducând duplicarea datelor și facilitând modificările centralizate [20].

Instrumentele moderne de debugging disponibile în Unity includ Visual Studio integration, runtime debugging cu break points, profiling și memory snapshot analysis. Aceste capabilități, combinate cu instrumentele specifice C# precum LINQ for collections și expresiile lambda, facilitează dezvoltarea și întreținerea codebase-urilor complexe. Unity oferă de asemenea posibilitatea utilizării reflection pentru scenarii avansate, deși cu considerente importante de performanță [20].

2.3. Sistemul de fizică și detectarea coliziunilor

Sistemul de fizică reprezintă o componentă critică în Unity, oferind simulare realistă a corpurilor rigide, detectarea coliziunilor și algoritmi de răspuns optimizați pentru aplicații în timp real. Implementat pe baza middleware-ului NVIDIA PhysX, acest subsistem balansează acuratețea simulării cu performanța computațională necesară pentru rate de cadre fluide [20].

Arhitectura sistemului fizic în Unity este structurată în jurul componentelor dedicate care pot fi atașate GameObject-urilor:

1. Colliders: definesc forma geometrică utilizată pentru detectarea coliziunilor, cu variante primitive (Box, Sphere, Capsule) și complexe (Mesh Collider). Aceste componente pot opera în două moduri distincte – ca rigid bodies complete sau ca triggere pentru detectarea overlapping-ului fără răspuns fizic.
2. Rigidbody: marchează un GameObject ca participând activ la simularea fizică, adăugând proprietăți precum masa, fricțiunea și amortizarea. Distincția între rigidbody kinematic (controlat programatic) și dynamic (controlat de motorul fizic) permite diferite grade de interacțiune cu simularea.
3. Joints: implementează constrângeri între corpuri rigide, de la simple articulații cu balama (Hinge Joint) la conexiuni complexe cu șase grade de libertate (Configurable Joint) [20].

Pipeline-ul de procesare fizică operează într-un step fix de 0.02 secunde (50Hz) implicit, independent de framerate-ul randării, asigurând consistența simulării indiferent de variațiile performanței. Această separare permite optimizarea diferențiată a subsistemelor vizuale și fizice, cu parametri de configurare distincti pentru fiecare [20].

Sistemul de detectare a coliziunilor implementează o abordare multi-fază pentru eficiență computațională:

1. Broad phase: utilizează structuri de date spațiale optimizate (quad trees, AABB trees) pentru identificarea rapidă a perechilor de obiecte potențial colidente, reducând dramatic numărul de teste necesare.
2. Narrow phase: calculează geometria exactă a coliziunii pentru perechile identificate în faza anterioară, generând informații detaliate precum punctele de contact, normălele și adâncimea penetrării.
3. Resolution phase: aplică impulsuri și forțe pentru rezolvarea penetrărilor și simularea răspunsului fizic corespunzător proprietăților materialelor implicate [20].

Raycasting și shape casting reprezintă instrumente esențiale pentru interogarea spațială a lumii fizice. Aceste operațiuni proiectează raze sau forme geometrice în scenă, returnând informații

detaliate despre obiectele intersectate. Aplicațiile includ detectarea țintelor, implementarea sistemelor de line-of-sight și calcularea traiectoriilor balistice [20].

Optimizarea performance fizice necesită considerații specifice datorită naturii intensive computațional a simulărilor:

1. Layer-based collision matrix: permite controlul granular al interacțiunilor între categorii de obiecte, eliminând calculele inutile pentru perechi care nu necesită detecție.
2. Collision meshes simplificate: utilizarea mesh collider-elor simplificate (convex) sau aproximarea cu primitive compuse reduce semnificativ costul computational al detectării.
3. Sleep thresholds: obiectele staționare sunt automat "adormite", fiind eliminate temporar din calculele fizice până la reactivarea prin forțe externe [20].

Fizica 2D specializată a fost introdusă în Unity 4.3, oferind un subsistem optimizat pentru jocuri bidimensionale. Acest sistem paralel implementează componente dedicate (Rigidbody2D, Collider2D) și utilizează middleware-ul Box2D pentru simulare eficientă. Separarea completă între sistemele 2D și 3D permite optimizări specifice fiecărui domeniu, rezultând în performanță superioară pentru proiectele specializate [20].

2.4. Managementul resurselor și optimizarea performanței

Managementul eficient al resurselor reprezintă un aspect critic în dezvoltarea jocurilor moderne, influențând direct performanța, timpii de încărcare și scalabilitatea proiectelor. Unity implementează multiple sisteme și metodologii pentru optimizarea utilizării memoriei, CPU și GPU, adaptate diverselor platforme țintă [20].

Asset Pipeline reprezintă infrastructura fundamentală pentru procesarea, importul și gestionarea resurselor în Unity. Acest sistem transformă fișierele sursă (modele 3D, texturi, audio) în formate optimizate pentru runtime, generând asset-uri serializate cu metadate asociate. Pipeline-ul implementează procese automatizate de compresie, mipmapping, și optimizare specifice platformei target, reducând semnificativ dimensiunea build-ului final [20].

Streaming Assets oferă o abordare optimizată pentru resursele voluminoase, permițând încărcarea asincronă și descărcarea controlată a conținutului. Addressable Assets System, introdus în versiunile recente, extinde acest concept implementând un framework comprehensiv pentru managementul dinamic al resurselor, cu beneficii substanțiale pentru proiecte de mari dimensiuni:

1. Content-addressable resources: identificarea resurselor prin hash-uri unice, decuplând referințele de locația fizică.
2. Remote hosting: separarea conținutului în pachete descărcabile post-instalare.
3. Dependency tracking: managementul automat al dependențelor între resurse.

4. Memory management: API-uri explicite pentru încărcarea și descărcarea controlată a resurselor [20].

Optimizarea memoriei reprezintă o provocare permanentă, în special pentru platformele cu resurse limitate. Unity oferă instrumente specializate pentru identificarea și rezolvarea problemelor de memorie:

1. Memory Profiler: analizează alocarea memoriei pe categorii (texturi, meshes, audio), identificând resurse redundante sau excesiv de voluminoase.
2. Object pooling patterns: reutilizarea instanțelor pentru entități frecvent create/distruse, reducând fragmentarea heap-ului.
3. Asset Bundle compression: compactarea și optimizarea resurselor pentru distribuție eficientă.
4. Mipmap streaming: încărcarea dinamică a nivelelor de rezoluție pentru texturi în funcție de distanța de vizualizare [20].

Level of Detail (LOD) systems implementează tranziții automate între variante de complexitate diferită ale modelelor în funcție de distanța față de cameră. Această tehnică reduce dramatic numărul de poligoane randate pentru obiectele distante, menținând fidelitatea vizuală pentru elementele apropiate. Unity oferă suporturi native pentru LOD groups și tranziții configurabile între nivele, facilitând implementarea acestei optimizări esențiale [20].

Draw call batching reprezintă o tehnică de optimizare critică pentru performanța GPU, combinând multiple obiecte în operațiuni de randare unificate. Unity implementează două variante complementare:

1. Static batching: combină meshurile imobile la build time, eliminând overhead-ul per-object pentru scenele statice.
2. Dynamic batching: grupează în runtime obiectele dinamice cu materiale identice, sub anumite limite de complexitate.

Profiling tools oferă vizibilitate detaliată asupra performanței aplicației, identificând bottleneck-urile și facilitând optimizarea țintită:

1. Unity Profiler: analizează performance per subsystem (CPU, GPU, rendering, physics, memory).
2. Frame Debugger: deconstruiește procesul de randare în pași individuali pentru analiza detaliată.
3. GPU Profiler: măsoară timpul de execuție pentru operațiunile grafice individuale.
4. Custom profiling API: permite instrumentarea codului pentru măsurarea precisă a secțiunilor critice [20].

2.5. Studii de caz: jocuri de succes dezvoltate în Unity

Analiza jocurilor de succes dezvoltate în Unity oferă perspective valoroase asupra capabilităților practice ale platformei în diverse contexte de producție. Aceste studii de caz ilustrează atât versatilitatea motorului cât și strategiile specifice de optimizare implementate pentru diferite genuri și platforme țintă.

Genshin Impact (miHoYo, 2020) reprezintă un exemplu remarcabil de implementare cross-platform la scară masivă. Acest action RPG open-world cu stilizare anime a generat peste 3 miliarde USD în primul an, stabilind un nou standard pentru jocurile mobile premium. Implementarea tehnică utilizează extensiv:

1. Custom render pipeline optimizat pentru stilizarea vizuală distinctă și eficiența cross-platform.
2. Streaming terrain system pentru lumea vast și diversă fără ecrane de încărcare vizibile.
3. Sophisticated culling techniques care maximizează performanța pe dispozitive mobile. menținând densitatea vizuală.
4. Client-server architecture scalabilă pentru funcționalitățile multiplayer [21].

Cuphead (Studio MDHR, 2017) demonstrează capacitatea Unity de a implementa viziuni artistice distinctive și neconvenționale. Acest run-and-gun platformer aclamat pentru estetica sa inspirată de animațiile din anii 1930 utilizează:

1. Custom 2D animation framework pentru animațiile hand-drawn (peste 50.000 de frame-uri desenate manual) [22].
2. Shader-based visual effects pentru emularea imperfecțiunilor filmului vintage.
3. Deterministic physics system pentru gameplay consistent și precis.
4. Optimized audio management pentru coloana sonoră orchestrală înregistrată live.

Hearthstone (Blizzard Entertainment, 2014) ilustrează scalabilitatea Unity pentru jocuri service-based cu audiență globală. Acest card game competitiv cu peste 100 milioane de jucători implementează:

1. Robust client-server synchronization pentru gameplay competitiv.
2. Efficient asset bundling system facilitând actualizări frecvente de conținut.
3. Cross-platform UI framework adaptat multipelor dispozitive.
4. Analytics integration pentru balance și feedback cantitativ [23].

Among Us (InnerSloth, 2018) reprezintă un caz interesant de scalare neașteptată, jocul evoluând de la un titlu indie obscur la un fenomen global cu peste 500 milioane de jucători. Implementarea tehnică remarcabil de eficientă include:

1. Lightweight networking solution optimizată pentru latență minimă și sincronizare precisă.
2. Minimalist art style care maximizează compatibilitatea cross-platform.

3. Efficient procedural map generation pentru variabilitate între sesiuni.
4. Optimized mobile controls accesibile unei audiențe largi [24].

Monument Valley (ustwo games, 2014) demonstrează capacitatea Unity de a facilita experiențe vizuale și conceptuale inovatoare. Acest puzzle game bazat pe iluzii optice și geometrie utilizează:

1. Custom geometry manipulation tools pentru leveluri bazate pe perspectivă.
2. Shader-based visual design pentru estetica minimalistă distinctivă.
3. Touch input optimization pentru interacțiune intuitivă.
4. Efficient level streaming pentru tranziții fluide între puzzle-uri [25].

Beat Saber (Beat Games, 2019) ilustrează versatilitatea Unity în domeniul VR, devenind unul dintre cele mai profitabile titluri pentru această platformă. Implementarea tehnică adresează provocările specifice realității virtuale prin:

1. High-performance rendering optimizat pentru framerate constant 90+ FPS.
2. Precise collision detection pentru feedback tactil consistent.
3. Efficient audio synchronization între gameplay și muzică.
4. Cross-platform VR optimization pentru HMD-uri diverse [26].

Aceste studii de caz evidențiază versatilitatea Unity ca platformă de dezvoltare, capacitatea sa de adaptare la cerințe diverse și potențialul de a susține atât producții independente de scară mică, cât și titluri AAA cu audiență globală. Succesul acestor implementări confirmă maturitatea ecosistemului și robustețea arhitecturală a motorului în contexte variate de producție.

3. PLATFORMA GODOT - ANALIZĂ TEHNICĂ ȘI APLICAȚII

3.1. Arhitectura și specificațiile tehnice ale platformei Godot

Godot Engine reprezintă o abordare arhitecturală distinctă în peisajul motoarelor de joc contemporane, implementând un design modular centrat pe conceptul de "noduri" și "scene". Această filosofie de design, inspirată parțial de arhitecturile orientate pe date (DOD - Data-Oriented Design), diferă fundamental de abordările tradiționale bazate pe moștenire și oferă flexibilitate remarcabilă pentru structurarea proiectelor [27].

Arhitectura fundamentală a Godot este organizată în jurul conceptului de "scene tree" - o structură ierarhică de noduri care încapsulează toate elementele jocului. Fiecare nod reprezintă o entitate atomică cu funcționalitate specifică (reprezentare vizuală, comportament, transformare spațială), iar scenele funcționează ca templates reutilizabile compuse din multiple noduri. Această abordare facilitează:

- composition over inheritance: entitățile complexe sunt create prin combinarea nodurilor specializate, evitând ierarhiile profunde de moștenire;
- scene instancing: reutilizarea eficientă a structurilor complexe prin instanțiere;
- hot-reloading: modificarea structurii în timpul execuției pentru iterație rapidă [28].

Sistemul de rendering implementează o arhitectură multi-backend adaptată diverselor contexte de utilizare:

- GLES2/GLES3 renderers: optimizate pentru compatibilitate și performanță pe dispozitive mobile și hardware mai vechi;
- vulkan renderer: implementat în Godot 4.0, oferind performanță și funcționalități moderne pentru hardware high-end;
- Forward+ rendering pipeline: balansează eficiența cu suportul pentru lighting dinamic și materiale complexe [28].

Subsistemele principale care constituie arhitectura Godot includ:

- rendering: implementează tehnologii moderne precum deferred rendering, real-time GI (Global Illumination), post-processing, și material system bazat pe PBR (Physically Based Rendering);
- audio: motor audio multicanal cu DSP (Digital Signal Processing), spațializare 3D, și streamer optimizat pentru memorie;
- physics: implementări specializate pentru fizică 2D (bazată pe Box2D) și 3D (bazată pe Bullet Physics) cu API unificate;
- animation: sistem complex cu support pentru skeletal animations, blend trees, și animation state machines;

- UI (User Interface): framework complet pentru interfețe bazat pe containere, cu suport pentru theming, styling și adaptare la rezoluții diverse;
- input system: abstractizare multi-device cu mapping configurabil și input contexts;
- networking: implementare high-level pentru multiplayer bazată pe RPC (Remote Procedure Calls) și node synchronization [28].

Sistemul de scripting în Godot este remarcabil prin diversitatea opțiunilor disponibile:

- GDScript: limbaj propriu, dinamic, cu sintaxă inspirată de Python, optimizat pentru integrarea cu motorul;
- C#: suport pentru ecosistemul .NET prin integrarea Mono;
- visual scripting: sistem node-based (Visual Shader Editor și Visual Script) pentru dezvoltare no/low-code;
- GDNative/GDExtension: interfață pentru integrarea codului nativ (C/C++, Rust) cu performanță optimă [28].

3.2. Limbajul GDScript și paradigma orientată pe noduri

GDScript reprezintă limbajul de scripting principal al platformei Godot, dezvoltat specific pentru a complementa arhitectura node-based și pentru a optimiza dezvoltarea jocurilor. Această creație originală, inspirată sintactic de Python dar cu adaptări specifice contextului, oferă un echilibru între accesibilitate și performanță [28].

Caracteristicile fundamentale ale GDScript includ:

- sintaxă clară și minimalistă: utilizează indentarea pentru definirea blocurilor și elimină punctuația excesivă, facilitând citirea și scriere rapidă;
- tipare dinamică cu opțiuni statice: suportă atât variabile tipate dinamic pentru flexibilitate în prototipare, cât și type hints pentru verificare statică în proiecte mature;
- integrare nativă cu motorul: acces simplificat la funcționalitățile motorului fără boilerplate sau binding-uri complexe;
- performanță optimizată: compilat în bytecode și executat de o VM (Virtual Machine) optimizată pentru scenariile specifice dezvoltării jocurilor;
- duck typing și signals: facilitează programarea event-driven și reduce coupling-ul între componente [28].

Paradigma orientată pe noduri definește fundamental abordarea de dezvoltare în Godot, diferențiind-o de sistemele entity-component tradiționale precum Unity. Câteva concepte cheie includ:

- nodurile ca unități funcționale: Fiecare nod implementează o funcționalitate specifică (Camera, Sprite, CollisionShape) și expune proprietăți configurabile relevante;
- scene ca template-uri compozite: O scenă agregă multiple noduri într-o structură reutilizabilă care poate fi instanțiată, extinzând conceptul tradițional de prefab;
- ierarhie cu propagare de proprietăți: Transformările (poziție, rotație, scală) se propagă automat de la părinte la copii, simplificând organizarea entităților complexe;
- traversare automată a tree-ului: Funcțiile speciale _process(), _physics_process(), _ready() sunt invocate recursiv pe întreaga ierarhie [28].

Mecanismul de comunicare între noduri implementează multiple strategii pentru facilitarea interacțiunilor decuplate:

- signals (semnale): implementarea pattern-ului observer pentru comunicare event-driven între noduri fără dependențe directe;
- NodePath și get_node(): referințe relative sau absolute către noduri în ierarhie pentru acces direct;
- groups: tagging-ul nodurilor pentru referențiere și procesare batch fără cunoașterea explicită a structurii;
- autoloads/singletons: noduri globale accesibile din orice context pentru servicii și state partajate [28].

3.3. Funcționalități avansate și optimizarea performanței

Godot Engine oferă un set comprehensiv de funcționalități avansate destinate dezvoltării jocurilor complexe și performante. Aceste capabilități, combinate cu instrumentele de optimizare integrate, facilitează implementarea proiectelor ambițioase menținând eficiența execuției pe hardware diverse.

Sistemul de rendering implementează tehnologii moderne adaptate diverselor contexte:

- PBR (Physically Based Rendering): material system standardizat pentru reprezentări realiste ale suprafețelor, cu suport pentru albedo, metalness, roughness și normal mapping;
- Global Illumination: tehnici pentru iluminare indirectă realistă, inclusiv lightmapping pentru surse statice și SDFGI (Signed Distance Field GI) pentru iluminare dinamică în Godot 4.0;
- Post-processing stack: efecte avansate precum bloom, tone mapping, depth of field și screen-space reflections pentru îmbunătățirea calității vizuale;
- GPU Particles: sistem de particule accelerat hardware capabil să gestioneze mii de elemente simultan pentru efecte vizuale complexe [28].

Animație avansată pentru personaje și obiecte interactive:

- skeletal animation system: rigging și animație completă pentru modele 3D, cu suport pentru weight painting și multiple deformation modes;
- animation tree: sistem vizual pentru blend-ul dinamic între animații bazat pe parametri și tranziții configurabile;
- animationPlayer: controlul precis al proprietăților și transformărilor pentru scene și UI [28].

Sisteme procedurale pentru generarea și manipularea conținutului:

- CSG (Constructive Solid Geometry): Crearea și editarea mesh-urilor prin operații booleene între primitive (uniune, intersecție, diferență);
- terrain system: generarea și manipularea terenurilor 3D cu height mapping și multiple layers pentru texturare [28].

Strategii de optimizare specifice platformei Godot includ:

- occlusion culling: eliminarea automată din pipeline-ul de randare a obiectelor obstrucționate de geometrie;
- object pooling: reutilizarea instanțelor pentru entități frecvent create/distruse, implementată nativ prin NodePool în Godot 4.0;
- sprite atlasing: combinarea texturilor multiple într-o singură imagine pentru reducerea draw calls și optimizarea memory bandwidth;
- LOD (Level of Detail) system: tranziția automată între variante de complexitate diferită în funcție de distanța față de cameră;
- multithreading: API-uri pentru distribuirea task-urilor computaționale pe multiple thread-uri, inclusiv ThreadGroup pentru procesare paralelă [28].

3.4. Studii de caz: jocuri reprezentative dezvoltate în Godot

Platforma Godot, deși mai nouă decât Unity, a câștigat popularitate datorită capacităților sale și naturii open source, atrăgând dezvoltatori independenți și studiouri mici. În această secțiune vom analiza câteva jocuri reprezentative dezvoltate cu motorul Godot, evidențiind diversitatea și potențialul platformei.

Sonic Colors Ultimate Portare (2023). Deși dezvoltarea originală a jocului nu a folosit Godot, portarea acestui titlu celebru al francizei Sonic pentru diverse platforme a fost parțial realizată folosind Godot Engine. Acest caz demonstrează capacitatea motorului de a gestiona proiecte de amploare cu grafică 3D de înaltă calitate și mecanici de joc complexe. Godot s-a dovedit capabil să

reproducă fidel experiența de joc originală, oferind în același timp optimizări pentru platformele moderne [29].

The Interactive Adventures of Dog Mendonça & Pizzaboy (2016), a fost dezvoltat de OKAM Studio și lansat în 2016, acest joc de aventură point-and-click bazat pe o serie de benzi desenate populare reprezintă un exemplu excelent de utilizare a Godot pentru crearea unei experiențe narative bogate. Jocul demonstrează capacitățile 2D ale motorului, sistemul de dialog și capacitatea de a crea experiențe interactive complexe. A fost lansat pe Steam și a primit recenzii pozitive pentru atmosfera sa unică și implementarea fidelă a universului original [30].

Kingdoms Of The Dump (TBA), Roach Games este un joc RPG inspirat de titluri clasice precum Earthbound și Chrono Trigger. Kingdoms of the Dump reprezintă un exemplu de utilizare a Godot pentru crearea unui joc cu mecanici complexe, sisteme de luptă turnate și o lume vastă. Deși încă în dezvoltare, jocul a atras atenția comunității prin calitatea sa artistică și potențialul de gameplay. Este un exemplu excelent al capacității Godot de a gestiona proiecte RPG elaborate [31].

Wargroove (2019) Chucklefish, dezvoltatorul jocului Wargroove, a menționat utilizarea Godot pentru anumite componente ale jocului, deși dezvoltarea principală a folosit un motor proprietar. Acest joc de strategie turn-based a fost bine primit pentru mecanicile sale rafinate și stilul artistic distinct. Contribuția Godot în dezvoltarea acestui titlu demonstrează versatilitatea motorului și capacitatea sa de a fi integrat în fluxuri de lucru complexe, alături de alte tehnologii [32].

Cruelty Squad (2021), unul dintre cele mai neconvenționale jocuri dezvoltate cu Godot, Cruelty Squad este un FPS cu o estetică deliberat respingătoare și mecanici experimentale. Jocul a atras atenția pentru stilul său artistic unic și comentariul său satiric, demonstrând că Godot poate fi utilizat pentru crearea unor experiențe artistice îndrăznețe. Recenziile critice au fost extrem de pozitive, lăudând originalitatea jocului și implementarea sa tehnică [33].

Cassette Beasts (2023), dezvoltat de Bytten Studio, Cassette Beasts este un RPG open-world cu elemente de colecționare de monștri. Acest joc ambițios folosește Godot pentru a crea o lume deschisă în stil pixel art, cu sisteme complexe de luptă și fuziune a creaturilor. Jocul demonstrează capacitatea Godot de a gestiona proiecte complexe cu lumi deschise și sisteme de joc elaborate [34].

Aceste studii de caz ilustrează versatilitatea motorului Godot și capacitatea sa de a susține o gamă largă de genuri și stiluri de joc. De la jocuri de acțiune la RPG-uri complexe, de la experiențe artistice experimentale la platformere clasice, Godot s-a dovedit capabil să satisfacă nevoile dezvoltatorilor independenți și ale studiourilor mici.

Succesul acestor jocuri în diverse genuri demonstrează că, deși Godot poate fi considerat mai puțin matur decât Unity sau alte motoare etablate, platforma este perfect capabilă să producă jocuri de calitate comercială care pot concura pe piața actuală. Aceste exemple oferă dovezi concrete ale punctelor forte ale Godot discutate în secțiunile anterioare: flexibilitatea sistemului bazat pe noduri,

eficiența limbajului GDScript și capacitatea de a crea experiențe personalizate fără limitările impuse de alte motoare comerciale.

3.5. Analiză comparativă și criterii de selecție pentru proiecte

Până acum au fost descrise în detaliu cele două platforme, mai jos a fost realizat un tabel pentru a afișa comparația lor (Tabelul 1).

Tabelul 1. Analiza comparativă a motoarelor Unity și Godot

Caracteristică	Godot	Unity	Descriere
Licență	Open-Source (MIT)	Licență comercială cu plan gratuit	Godot oferă libertate totală fără restricții
Limbaj de Programare	GDScript (similar Python), C#, C++	C#, UnityScript (deprecated)	Godot mai flexibil din punct de vedere lingvistic
Platforme de Dezvoltare	Windows, macOS, Linux	Windows, macOS, Linux	Suport complet cross-platform pentru ambele platforme
Platforme de Export	Desktop, Mobile, Web, Console	Desktop, Cloud Mobile, Web, Console, VR/AR	Unity are ecosistem mai vast de platforme
Performanță 2D	Excelentă, optimizată nativ	Bună, necesită optimizări	Godot este superior pentru jocuri 2D
Performanță 3D	Îmbunătățire continuă	Foarte bună, stabilă	Unity e dominant în producții 3D complexe
Cloud Gaming	Suport foarte limitat, în dezvoltare	Integrare completă cu servicii cloud	Unity oferă soluții cloud mai mature
Suport Console	PlayStation, Xbox, Nintendo (în dezvoltare)	PlayStation, Xbox, Nintendo, Switch	Unity are certificări și suport oficial
Mărime Motor	~40 MB	~1-2 GB	Godot extrem de ușor și compact
Comunitate	În creștere rapidă, open-source	Foarte mare, profesională	Unity are comunitate și resurse mai extinse
Grafică	Motor grafic modern	Motor grafic avansat	Unity oferă instrumente grafice superioare
Randare Fizică	PBR de bază	PBR avansat	Unity superior în randare realistă
Sistemul de Particule	Sistem propriu simplu	Sistemă complexă de particule	Unity oferă efecte vizuale mai sofisticate
Animație	Sistem propriu robust	Animator profesional	Unity mai avansat pentru animații complexe
Realitate Virtuală	Suport experimental	Suport complet VR	Unity e unul din lidereri în dezvoltare VR
Realitate Augmentată	Capacități de bază	Platforme dedicate AR	Unity dominant în AR

Tabel 1. (continuare)

Caracteristică	Godot	Unity	Descriere
Cloud Rendering	Minimal	Servicii integrate AWS, Azure	Unity oferă soluții enterprise
Inteligență Artificială	Suport basic	NavMesh, sisteme AI avansate	Unity oferă instrumente AI mai complexe
Multiplicare Rețea	Suport built-in	Netcode, Multiplayer HLAPI/LLAPI	Unity are soluții de rețea mai mature
Cost Dezvoltare	Complet gratuit	Gratuit sub $200k venituri	Godot total gratuit indiferent de scară
Monetizare Joc	Deschis, fără restricții	Opțiuni integrate de monetizare	Unity oferă unelte comerciale built-in
Instrumente Audio	Suport audio integrat	FMOD, Wwise integration	Unity superior pentru audio profesional
Asset Store	Magazin propriu mic	Asset Store vast	Unity are bibliotecă enormă de resurse
Scalabilitate	Bună pentru proiecte mici/medii	Excelentă pentru orice scară	Unity mai potrivit pentru proiecte mari
Documentație	În continuă îmbunătățire	Foarte cuprinzătoare	Unity are documentații și tutoriale extinse
Curba de Învățare	Mai ușoară	Mai complexă	Godot prietenos pentru începători

Analiza celor două platforme a evidențiat diferențe fundamentale în filozofie, arhitectură și model economic:

- Aspecte tehnice: Unity oferă capacități grafice avansate și suport extins pentru multiple platforme, excelând în proiecte 3D complexe și dezvoltare pentru console. Godot prezintă avantaje în eficiență, timp de iterație și arhitectura bazată pe noduri, fiind deosebit de potrivit pentru dezvoltarea 2D și proiecte cu resurse limitate.
- Ecosisteme și comunități: Unity beneficiază de un ecosistem vast și matur, cu peste 70.000 de asset-uri comerciale și suport tehnic structurat. Godot, deși cu un ecosistem mai restrâns, se bucură de o comunitate entuziastă, în creștere rapidă, și o orientare puternică spre open-source și colaborare.
- Modele economice: Diferența fundamentală între modelul comercial al Unity, bazat pe abonamente și taxe pe venit, și modelul open-source al Godot, cu licență MIT, determină considerente importante pentru dezvoltatori, în special pentru studiourile independente și proiectele cu buget limitat.

Aplicabilitate contextualizată a platformelor

Cercetarea demonstrează că alegerea platformei optime depinde în mod crucial de contextul specific al proiectului:

Unity este preferabil pentru:

- jocuri 3D complexe care necesită capacități grafice avansate;
- proiecte care vizează consola sau VR/AR, unde Unity oferă suport nativ extins;
- dezvoltarea aplicațiilor care beneficiază de integrarea cu servicii de analiză și monetizare;
- echipe cu experiență anterioară în C# și arhitectura componentelor;
- proiecte cu buget mediu și mare, unde costul licențelor poate fi amortizat.

Godot este mai potrivit pentru:

- dezvoltarea jocurilor 2D, unde arhitectura nodurilor și fluxul de lucru sunt optimizate;
- proiecte cu resurse limitate sau studii independente;
- dezvoltatori care preferă libertatea completă în privința drepturilor de proprietate intelectuală;
- aplicații care necesită particularizări semnificative ale motorului.

Atât Unity cât și Godot reprezintă soluții viabile pentru dezvoltarea jocurilor, fiecare cu avantaje distincte. Alegerea optimă depinde de contextul specific al proiectului, de obiectivele echipei și de resursele disponibile. Tendințele observate sugerează că ambele platforme vor continua să evolueze, iar complementaritatea lor va contribui la diversificarea și îmbogățirea ecosistemului de dezvoltare a jocurilor.

4. REALIZAREA JOCURILOR ÎN UNITY ȘI GODOT

4.1. Dezvoltarea jocurilor demonstrative

În cadrul acestei lucrări au fost cercetate și descrise motoarele de joc: Godot și Unity. Ulterior au fost create 2 jocuri cu ajutorul Asset Store al ambelor motoare. Joaca va este de tipul „Survivor Like” asemănător cu „Vampire Survivors”. Pentru Unity există multe variante contra-plată și versiuni gratuite limitate. Pe când Godot oferă un singur asset în 3D. Pentru teză vor fi folosite assetul Undead Survivors în Unity 2D[35] și Survivors Starter Kit în Godot 3D [36].

Pentru aceasta la început a fost creat jucătorul (Fig. 1). A fost folosit RigidBody2D, Sprite nr.0 și player input system din Unity.

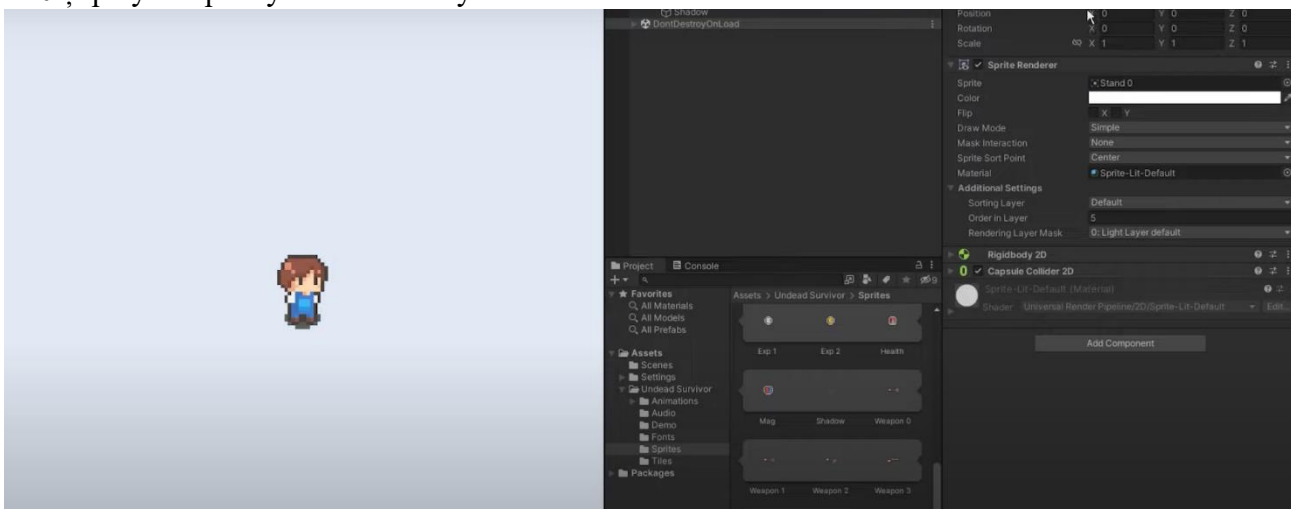

Fig. 1. Crearea jucătorului.

În continuare a fost realizată mișcarea pentru aceasta a fost creat primul C# script în folder nou (Fig. 2).

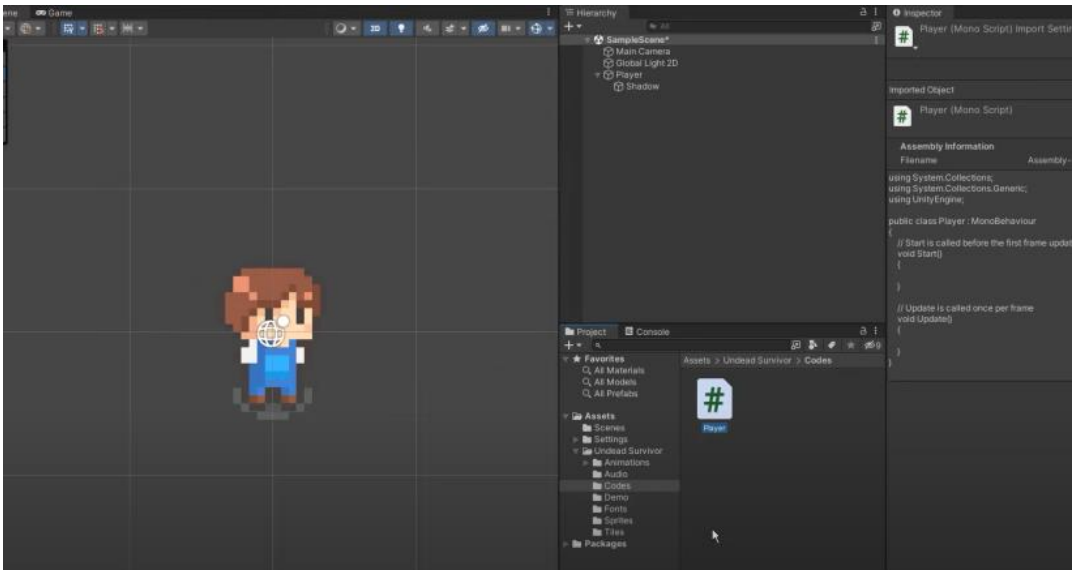

Fig. 2. Crearea scriptului.

Codul inițial care a fost făcut pentru mișcare este prezentat mai jos:

using System.Collections;

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

public class Player : MonoBehaviour
{
    public Vector2 inputVec;
    public float speed;
    Rigidbody2D rigid;

    void Awake()
    {
        rigid = GetComponent<Rigidbody2D>();
    }

    void Update()
    {
        inputVec.x = Input.GetAxisRaw("Horizontal");
        inputVec.y = Input.GetAxisRaw("Vertical");
    }

    void FixedUpdate()
    {
        Vector2 nextVec = inputVec.normalized * speed * Time.fixedDeltaTime;
        rigid.MovePosition(rigid.position + nextVec);
    }
}
```

În continuare a fost realizată animația jucătorului (Fig. 3). Au fost folosite asseturile din folderul farmer 0. Ulterior a fost creat animation override ceea ce permite refolosirea animațiilor pentru restul proiectului, doar vor fi schimbate spriteurlile.

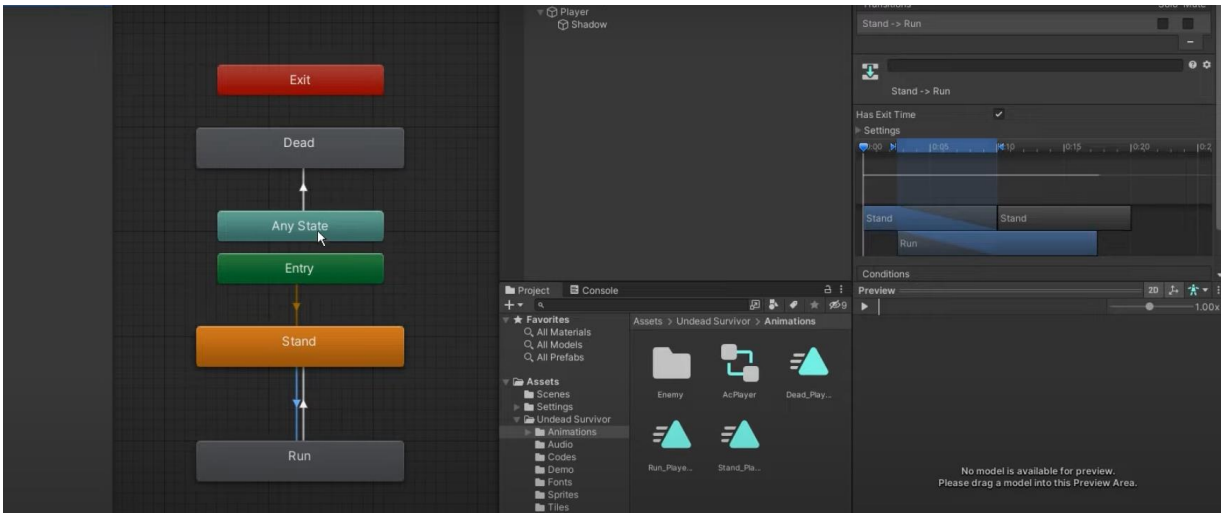

Fig. 3. Crearea animațiilor pentru jucător.

În continuare s-a creat terenul de joacă (Fig. 4) prin multiplicarea spriteurilor Tile 0, 1, 2, 3, 4 și 5 – terenul verde care are diferite efecte.

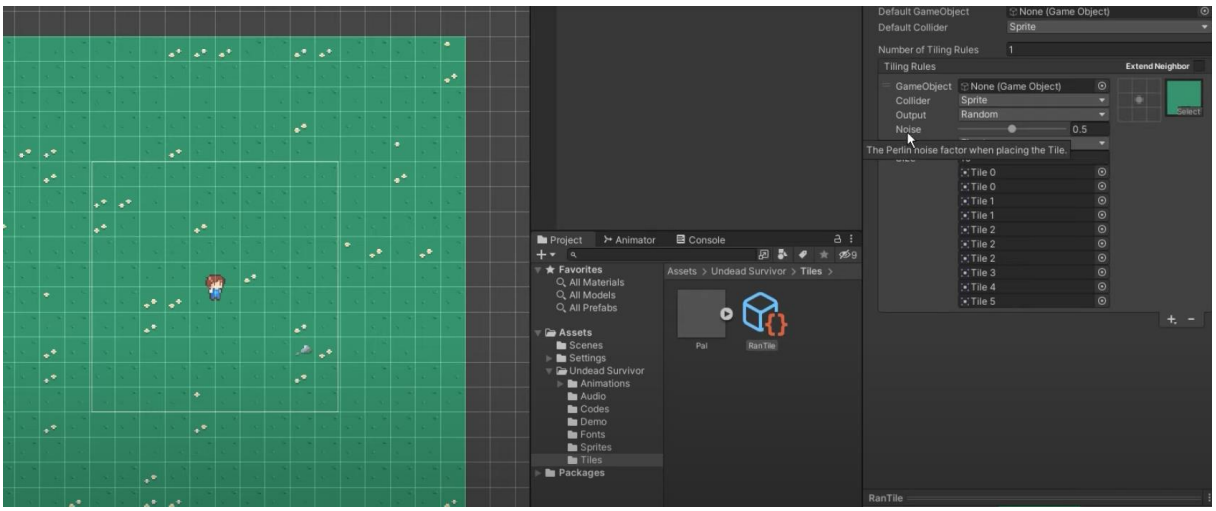

Fig. 4. Crearea terenului de joacă.

Mai departe au fost create spriturile inamicii, animațiile acestora și logica de căutare a jucătorului (Fig.5). La fel a fost realizată și logica de creare (spawn) a inamicilor. Au fost folosite spriteurile din folderul enemy 1 și 2.

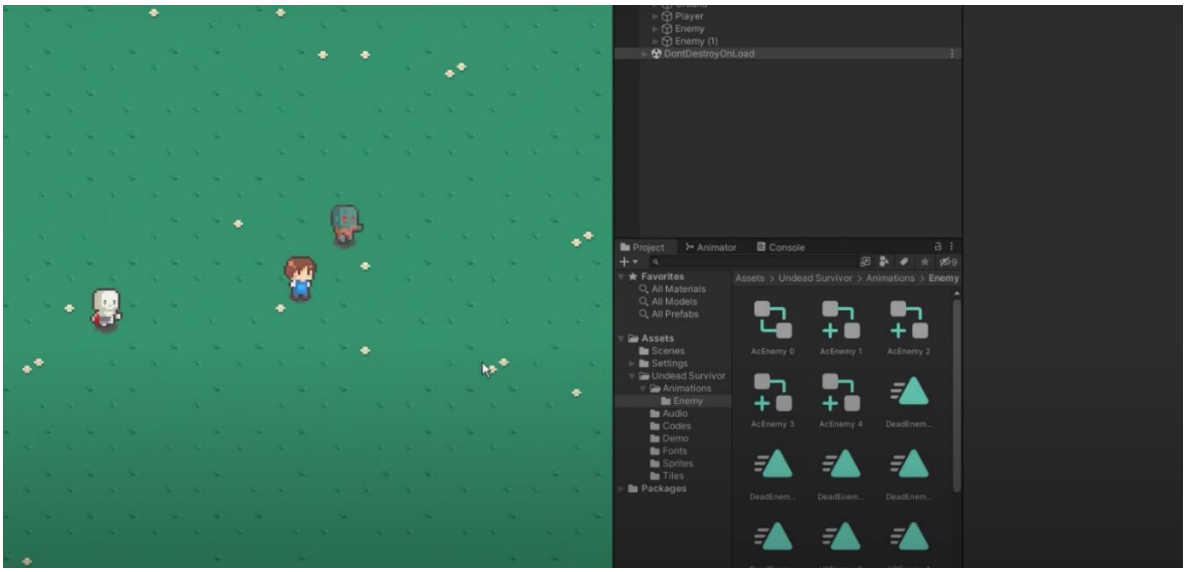

Fig. 5. Crearea inamicilor.

Mai jos este prezentat codul pentru comportamentul inamicilor:

```
public class Enemy : MonoBehaviour
{
    public float speed;
    public float health;
    public float maxHealth;
    public RuntimeAnimatorController[] animCon;
    public Rigidbody2D target;
    bool isLive = true;
    Rigidbody2D rigid;
    Collider2D coll;
    Animator anim;
    SpriteRenderer spriter;
    WaitForFixedUpdate wait;
```

```
void Awake()
{
    rigid = GetComponent<Rigidbody2D>();
    coll = GetComponent<Collider2D>();
    anim = GetComponent<Animator>();
    spriter = GetComponent<SpriteRenderer>();
    wait = new WaitForFixedUpdate();
}

void FixedUpdate()
{
    if (!GameManager.instance.isLive)
        return;

    if (!isLive || anim.GetCurrentAnimatorStateInfo(0).IsName("Hit"))
        return;

    Vector2 dirVec = target.position - rigid.position;
    Vector2 nextVec = dirVec.normalized * speed * Time.fixedDeltaTime;
    rigid.MovePosition(rigid.position + nextVec);
    rigid.velocity = Vector2.zero;
}

void LateUpdate()
{
    if (!GameManager.instance.isLive)
        return;

    if (!isLive)
        return;

    spriter.flipX = target.position.x < rigid.position.x;
}

void OnEnable()
{
    target = GameManager.instance.player.GetComponent<Rigidbody2D>();
    isLive = true;
    coll.enabled = true;
    rigid.simulated = true;
    spriter.sortingOrder = 2;
    anim.SetBool("Dead", false);
    health = maxHealth;
}

public void Init(SpawnData data)
{
    anim.runtimeAnimatorController = animCon[data.spriteType];
    speed = data.speed;
    maxHealth = data.health;
    health = data.health;
```

```
    }

    void OnTriggerEnter2D(Collider2D collision)
    {
        if (!collision.CompareTag("Bullet") || !isLive)
            return;

        health -= collision.GetComponent<Bullet>().damage;
        StartCoroutine(KnockBack());

        if (health > 0)
        {
            anim.SetTrigger("Hit");
            AudioManager.instance.PlaySfx(AudioManager.Sfx.Hit);
        }
        else
        {
            isLive = false;
            coll.enabled = false;
            rigid.simulated = false;
            spriter.sortingOrder = 1;
            anim.SetBool("Dead", true);
            GameManager.instance.kill++;
            GameManager.instance.GetExp();
            Invoke("Dead", 1f);//Death animation

            if (GameManager.instance.isLive)
                AudioManager.instance.PlaySfx(AudioManager.Sfx.Dead);
        }
    }

    IEnumerator KnockBack()
    {
        yield return wait;
        Vector3 playerPos = GameManager.instance.player.transform.position;
        Vector3 dirVec = transform.position - playerPos;
        rigid.AddForce(dirVec.normalized * 3, ForceMode2D.Impulse);


    }

    void Dead()
    {
        gameObject.SetActive(false);
    }
}
```

În continuare au fost create armele și logica de atacă (Fig. 6). Pentru aceasta au fost folosite spriteurile bullet 0 și 3, de asemenea și weapon 0 și 3. A fost luată în considerație viteza de mișcare a jucătorului și a inamicilor pentru a crea o senzație de impact.

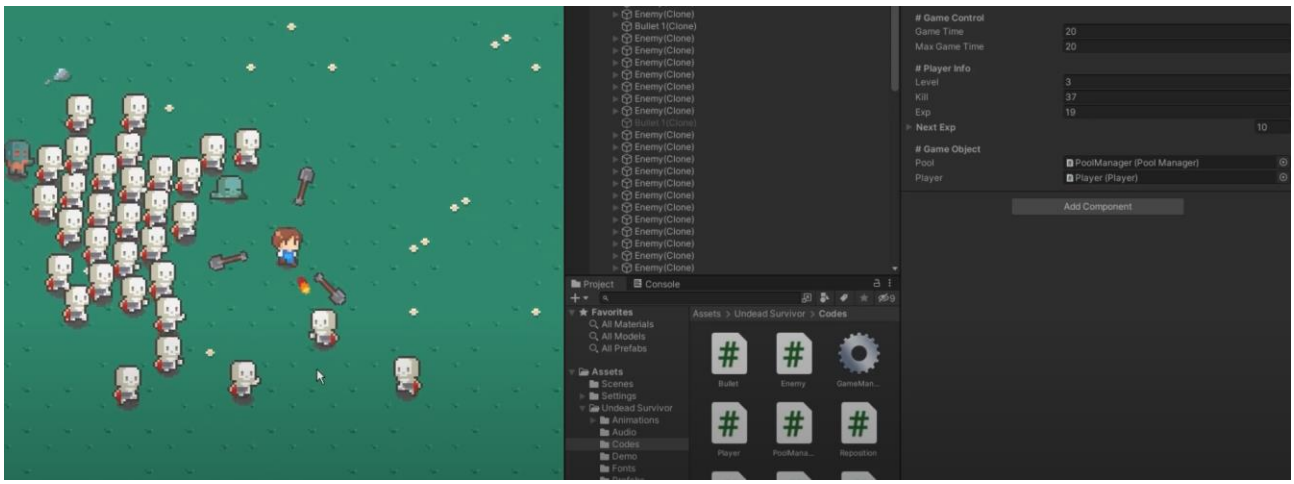

Fig. 6. Crearea armelor.

Mai jos este prezentat o bucată de cod care răspunde pentru arma de foc.

```
void Fire()
    {
        if (!player.scanner.neareastTarget)
            return;

        Vector3 targetPos = player.scanner.neareastTarget.position;
        Vector3 dir = targetPos - transform.position;
        dir = dir.normalized;

        Transform bullet = GameManager.instance.pool.Get(prefabId).transform;
        bullet.position = transform.position;
        bullet.rotation = Quaternion.FromToRotation(Vector3.up, dir);
        bullet.GetComponent<Bullet>().Init(damage, count, dir);

        AudioManager.instance.PlaySfx(AudioManager.Sfx.Range);
    }
}
```

În continuare a fost creat HUD-ul pentru jucător (Fig. 7). S-a folosit pentru crearea HUDU-lui spriteurile panel, icon 0, front 0, back 0 și title 0.

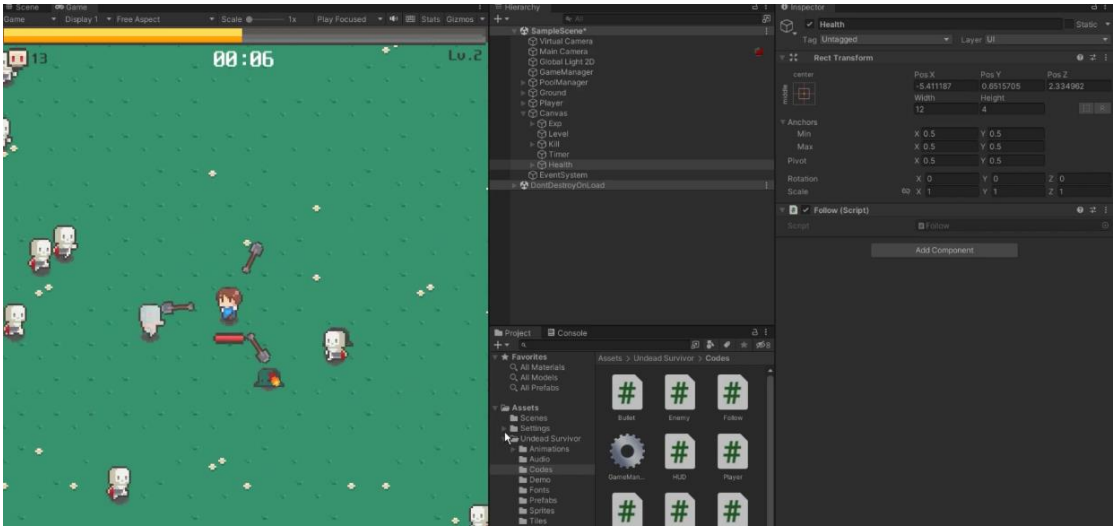

Fig. 7. HUD-ul jucătorului.

Mai departe a fost realizat posibilitatea de a putea alege un power up (Fig. 8). Pentru aceasta au fost folosite spriteurile select 0, 3, 6, 7 și 8 precum și title 1 și 2.

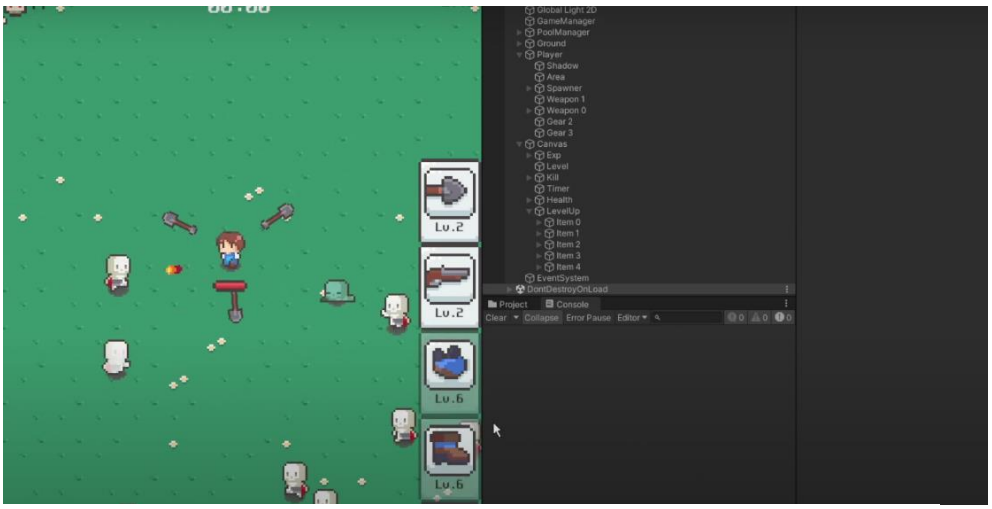

Fig. 8. Power Ups.

În continuare a fost creat fereastra de startu și finisarea jocului cu implementarea power up la ridicarea nivelului jucătorului (Fig. 9). Au fost folosite aceleași spriteuri, doar că a fost realizată logica acesteia la level up al jucătorului.

Fig. 9. Implementarea power up la ridicarea nivelului jucătorului.

Mai jos este dat codul pentru implementarea acestei logicii:

```
public class LevelUp : MonoBehaviour
{
    RectTransform rect;
    Item[] items;

    void Awake()
    {
        rect = GetComponent<RectTransform>();
        items = GetComponentsInChildren<Item>(true);
    }

    public void Show()
```

```
{
    Next();
    rect.localScale = Vector3.one;
    GameManager.instance.Stop();
    AudioManager.instance.PlaySfx(AudioManager.Sfx.LevelUp);
    AudioManager.instance.EffectBgm(true);
}

public void Hide()
{
    rect.localScale = Vector3.zero;
    GameManager.instance.Resume();
    AudioManager.instance.PlaySfx(AudioManager.Sfx.Select);
    AudioManager.instance.EffectBgm(false);
}

public void Select(int index)
{
    items[index].OnClick();
}

void Next()
{
    // Disable all items first
    foreach (Item item in items)
    {
        item.gameObject.SetActive(false);
    }

    int[] ran = new int[3]; // You want 2 or 3, so let's aim for 3 potential slots

    while (true)
    {
        ran[0] = Random.Range(0, items.Length);
        ran[1] = Random.Range(0, items.Length);
        ran[2] = Random.Range(0, items.Length);

        if (ran[0] != ran[1] && ran[1] != ran[2] && ran[0] != ran[2])
            break;
    }

    for (int index = 0; index < ran.Length; index++)
    {
        Item ranItem = items[ran[index]];

        if (ranItem.level == ranItem.data.damages.Length)
        {
            items[4].gameObject.SetActive(true);
        }
        else
        {
            ranItem.gameObject.SetActive(true);          }       }   }}
```

După aceasta a fost creată alegerea personajelor și logica de deschidere a personajelor noi (Fig. 10). Pentru realizarea acestora au fost folosite spriteurile farmer 1, 2 și 3. Pentru început jucătorii blocați sunt ascunși și pentru deschiderea acestora sunt realizate anumite obiective.

Fig. 10. Alegerea personajelor.

Mai jos este dat codul care răspunde de logica obiectivelor:

```
public class AchiveManager : MonoBehaviour
{
    public GameObject[] lockCharacter;
    public GameObject[] unlockCharacter;
    public GameObject uiNotice;

    enum Achive { UnlockBanan, UnlockBean }
    Achive[] achives;
    WaitForSecondsRealtime wait;
    void Awake()
    {
        achives = (Achive[])Enum.GetValues(typeof(Achive));
        wait = new WaitForSecondsRealtime(5);
        if (!PlayerPrefs.HasKey("MyData"))
        {
            Init();
        }

    }

    void Init()
    {
        PlayerPrefs.SetInt("MyData", 1);

        foreach (Achive achive in achives)
        {
            PlayerPrefs.SetInt(achive.ToString(), 0);
        }
```

```
    }

    void Start()
    {
        UnlockCharacter();
    }

    void UnlockCharacter()
    {
        for (int index = 0; index < lockCharacter.Length; index++)
        {
            string achiveName = achives[index].ToString();
            bool isUnlock = PlayerPrefs.GetInt(achiveName) == 1;
            lockCharacter[index].SetActive(!isUnlock);
            unlockCharacter[index].SetActive(isUnlock);
        }
    }

    void LateUpdate()
    {
        foreach (Achive achive in achives)
        {
            CheckAchive(achive);
        }
    }

    void CheckAchive(Achive achive)
    {
        bool isAchive = false;
        switch (achive)
        {
            case Achive.UnlockBanan:
                isAchive = GameManager.instance.kill >= 10;
                break;
            case Achive.UnlockBean:
                isAchive = GameManager.instance.gameTime == GameManager.instance.maxGameTime;
                break;
        }

        if (isAchive && PlayerPrefs.GetInt(achive.ToString()) == 0)
        {
            PlayerPrefs.SetInt(achive.ToString(), 1);

            for (int index = 0; index < uiNotice.transform.childCount; index++)
            {
                bool isActive = index == (int)achive;
                uiNotice.transform.GetChild(index).gameObject.SetActive(isActive);
            }

            StartCoroutine(NoticeRoutine());
```

```
    }
  }

  IEnumerator NoticeRoutine()
  {
    uiNotice.SetActive(true);

    AudioManager.instance.PlaySfx(AudioManager.Sfx.LevelUp);

    yield return wait;
    uiNotice.SetActive(false);
  }
}
```

În mod asemănător a fost realizată și joaca în Godot. La început, de asemenea, a fost creat jucătorul. În proiectul pentru Godot materialele 3D deja sunt create și sunt puse în folderul kaykit_characters. Pentru jocul dat totul a fost folosit de acolo: jucătorul, inamicii și terenul de joacă. În figura 11 se poate vedea terenul de joacă și jucătorul.

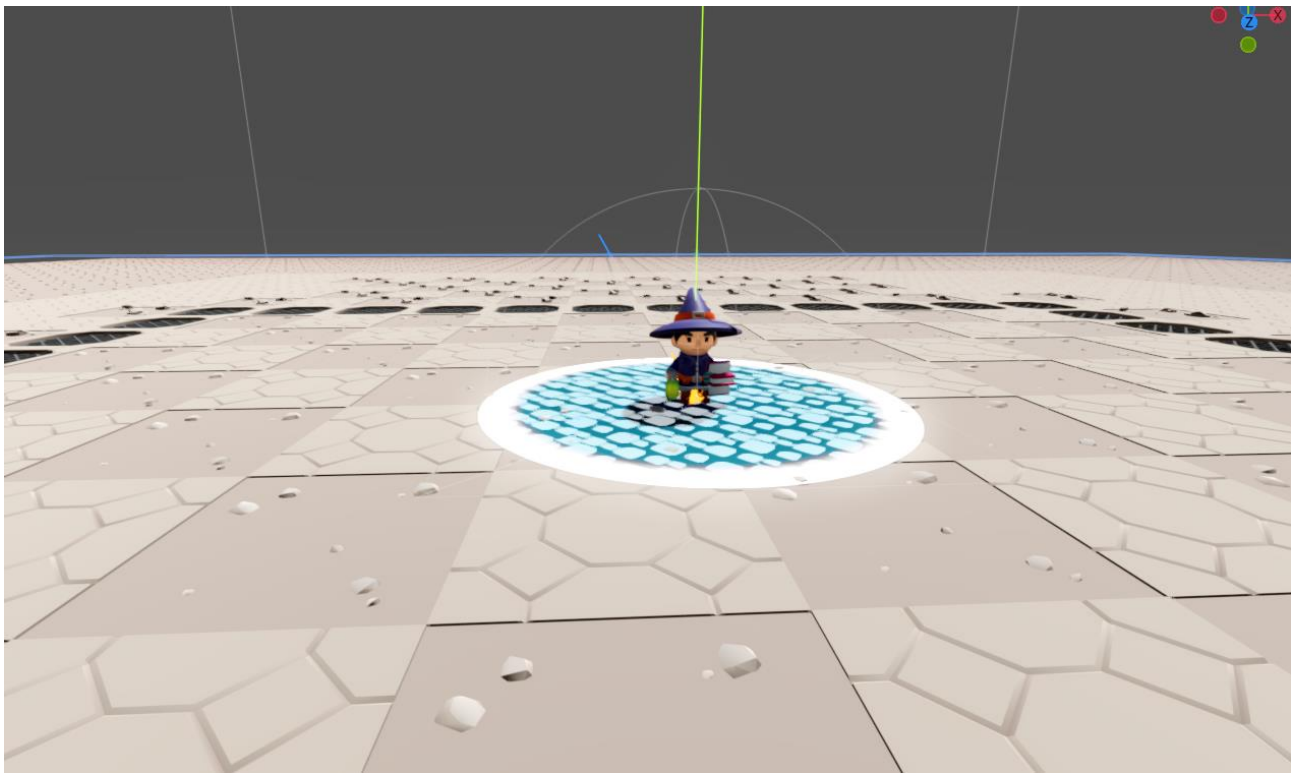

Fig. 11. Jucătorul și terenul de joacă.

Mai jos este prezntat codul care răspunde de mișcare:

```
public override void _PhysicsProcess(double delta)
                    {
                        Vector3 velocity = Velocity; // Add the gravity.
                        if (!IsOnFloor())
                            velocity.Y -= gravity * (float)delta;

                        // Get the input direction and handle the movement/deceleration.
                        Vector2 inputDir = Input.GetVector("left", "right", "up", "down");
```

```
                    Vector3 direction = (Transform.Basis * new Vector3(inputDir.X, 0,
inputDir.Y)).Normalized();
                    if (direction != Vector3.Zero)
                    {                       velocity.X = direction.X * Speed;
                        velocity.Z = direction.Z * Speed;               }
                    else
                    {                       velocity.X = Mathf.MoveToward(Velocity.X, 0,
Speed);
                        velocity.Z = Mathf.MoveToward(Velocity.Z, 0, Speed);      }

                    Velocity = velocity;
                    MoveAndSlide();

                    velocity.Y = 0;
                    _animationTree.Set("parameters/walking/blend_amount",
velocity.LimitLength(1).Length());
                    if (Mathf.IsZeroApprox(velocity.Length())) return;

                    _visual.LookAt(Position + 10 * velocity, Vector3.Up, true);}
```

De asemenea au fost realizați și inamicii (Fig. 12.), precum și logica acestora.

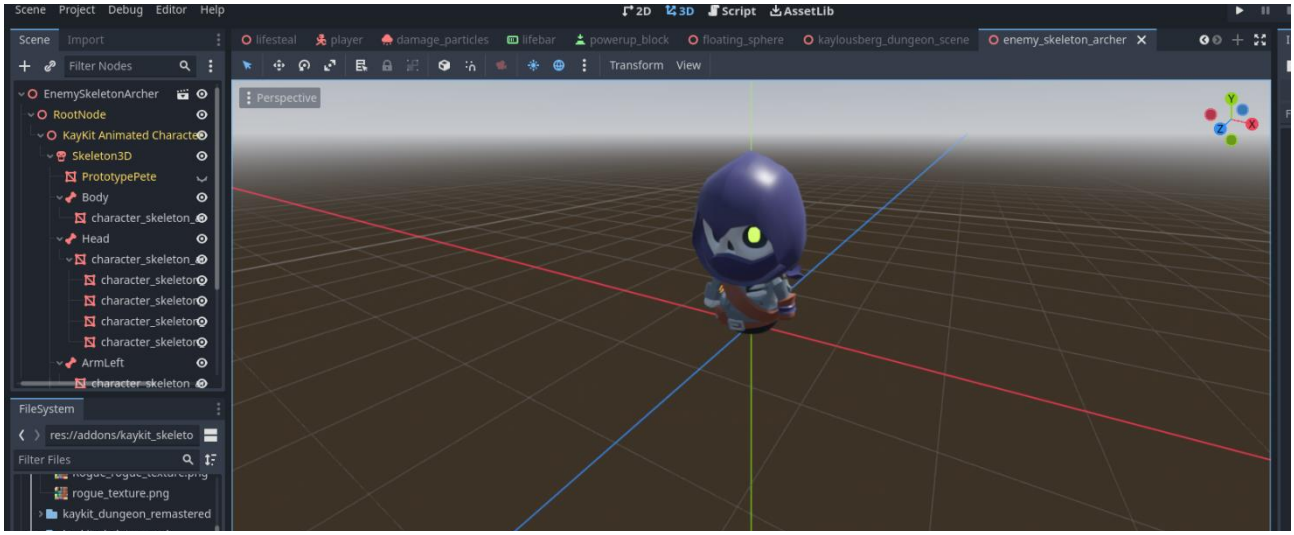

Fig. 12. Crearea inamicilor.

Mai jos este prezentată logica inamicilor:

```
using Godot;
public enum EnemyClass
{
                    Minion = 0,
                    Warrior = 1,
                    Archer = 2,
                    Mage = 3,

                    Boss = 100
}

public partial class Enemy : RigidBody3D
{
                    public const uint MaxLifepoints = 10;
```

```
    [Export]
    public int Lifepoints { get; set; } = (int)MaxLifepoints;

    [Export]
    public uint Damages { get; set; } = 5;

    [Export]
    public float MovementSpeed = 4;

    [Export]
    public int Experience { get; private set; } = 1;

    private GameManager _gameManager;
    private GpuParticles3D _damageParticles;

    private ProgressBar _lifebar;
    private Camera3D _camera;
    private Label _username;
    private Timer _attackCooldown;

    [Signal]
    public delegate void OnEnemyHitEventHandler(Enemy enemy, int damages);

    public bool IsPlayerInAttackRange => (_gameManager.Player.GlobalPosition - GlobalPosition).Length() <= 2f;


    public override void _Ready()
    {
        _gameManager = GetNode<GameManager>("/root/GameManager");

        _attackCooldown = GetNode<Timer>("AttackCooldown");
        _damageParticles = GetNode<GpuParticles3D>("DamageParticles");

        _camera = GetNode<Camera3D>("../Player/Camera3D");

        // var hud = GetNode<Control>("../HUD");
        // _lifebar = GD.Load<PackedScene>("res://Prefabs/UI/enemy_life_bar.tscn").Instantiate<ProgressBar>();
        // _lifebar.MaxValue = Lifepoints;
        // hud.AddChild(_lifebar);
    }

    public override void _Process(double delta)
    {
        base._Process(delta);

        if (_attackCooldown.TimeLeft <= 0) Attack();
```

```
            LookAt(_gameManager.Player.GlobalPosition, Vector3.Up, true);

            if (_lifebar != null)
            {
                Vector2 lifeBarPos = _camera.UnprojectPosition(GlobalPosition);
                lifeBarPos -= _lifebar.Size / 2;
                lifeBarPos.Y -= 50;
                _lifebar.Position = lifeBarPos;
                _lifebar.Value = Lifepoints;
            }

            if (_username == null) return;

            Vector2 usernamePos = _camera.UnprojectPosition(GlobalPosition);
            usernamePos -= _username.Size / 2;
            usernamePos.Y -= 40;
            _username.Position = usernamePos;
        }

        public override void _PhysicsProcess(double delta)
        {
            base._PhysicsProcess(delta);

            var playerPos = _gameManager.Player.GlobalPosition;
            var direction = (playerPos - GlobalPosition).LimitLength();

            LinearVelocity = direction * MovementSpeed;
        }

        public override void _ExitTree()
        {
            base._ExitTree();

            _lifebar?.QueueFree();
            _username?.QueueFree();
        }

        private void Attack()
        {
            if (!IsPlayerInAttackRange) return;
            _attackCooldown.Start();
            _gameManager.Player.TakeDamages(Damages);
        }

        internal void SetName(string username)
        {
            _username = new() { Text = username };
            GetNode<Control>("../HUD").AddChild(_username);
        }
```

```
        internal void TakeDamages(uint damages = 1)
        {
            EmitParticles();

            EmitSignal(SignalName.OnEnemyHit, this, damages);
            Lifepoints -= (int)damages;
            if (Lifepoints > 0) return;
            QueueFree();
        }

        internal async void EmitParticles()
        {
            var particles = _damageParticles.Duplicate() as GpuParticles3D;
            AddChild(particles);
            particles.Emitting = true;
            await ToSignal(GetTree().CreateTimer(particles.Lifetime),
SceneTreeTimer.SignalName.Timeout);
            particles.QueueFree();
        }
    }
```

La fel a fost realizată și interfața pentru power ups când jucătorul va avansa la un nivel nou (Fig . 13.).

Fig. 13. Interfața pentru lvl up.

4.2. Analiza comparativă a platformelor Unity și Godot

Pentru crearea tezei a fost folosit laptopul Zenbook 14 UM425. Toate specificațiile acestui model sunt disponibile pe site-ul oficial Asus [37]:

- Sistem de operare: Windows 11 Home;
- Procesor: AMD Ryzen™ 7 4700U Mobile Processor (8C/8T, 12MB Cache, 4.1 GHz Max Boost);
- Grafica: AMD Radeon™ Graphics;

- Display: 14.0-inch, 100% sRGB color gamut, 400nits, FHD (1920 x 1080) 16:9 aspect ratio, IPS-level Panel, Anti-glare display, LED Backlit, (Screen-to-body ratio)90%, 60Hz refresh rate, Non-touch screen;
- Memoria: 16GB LPDDR4X;
- Memoria pe disk: 512GB M.2 NVMe™ PCIe® 3.0 SSD.

În tabelul 2 sunt prezentate analiza comparativă a motoarelor Unity și Godot obținută în urma evaluării jocurilor demonstrative dezvoltate.

Tabelul 2. Analiza comparativă a motoarelor Unity și Godot în baza rezultatelor practice

Caracteristică	Godot	Unity	Descriere
Timpul de deschidere	în jur de 20 secunde	până la 2 minute	Godot este cu mult mai rapid
Timpul de compilare a proiectului	până la 20 de secunde	până la 2 minute	Godot deși realizat în 3D arată o performanță mult mai bună
Timpul de încărcare a proiectului în timpul editării	momentan	până la 1 minută	Chiar și pentru niște schimbări mici Unity necesită mult timp
Resursele ocupate pe calculator	141 MB	1.54GB	Godot cântărește extrem de puțin
Resursele ocupate pe calculator în timpul lucrului	1.5 GB RAM inactiv/ 2 GB RAM cu proiectul deschis	1.5 GB RAM inactiv/ 2 GB RAM cu proiectul deschis	Godot afișează o performanță remarcabilă pentru jocul 3D

CONCLUZII

În această lucrare au fost cercetate și analizate platformele și instrumentele existente disponibile pentru dezvoltarea jocurilor utilizând motoarele Godot și Unity.

După cercetarea și analiza surselor informative se poate afirma că compararea și înțelegerea platformelor de dezvoltare a jocurilor este esențială pentru luarea deciziilor informate în industria dezvoltării de jocuri.

Analiza detaliată a ambelor motoare de joc a permis înțelegerea atât a punctelor forte, cât și a limitărilor fiecăruia. Godot excelează prin natura sa open-source, dimensiunea redusă și performanța ridicată pe dispozitive cu resurse limitate, în timp ce Unity oferă un ecosistem matur, instrumente avansate și o comunitate extinsă de dezvoltatori.

Investigarea limbajelor și instrumentelor de programare specifice (GDScript pentru Godot și C# pentru Unity) a furnizat cunoștințele necesare pentru a alege platforma potrivită în funcție de complexitatea proiectului și experiența echipei de dezvoltare. Această înțelegere profundă a capacităților și limitărilor tehnologice a condus la o arhitectură solidă și eficientă a jocului.
Testele de performanță au demonstrat că, deși ambele motoare sunt capabile să livreze experiențe de joc de înaltă calitate, există diferențe notabile în ceea ce privește consumul de memorie, timpul de încărcare și performanța pe diverse platforme hardware.

Evaluarea comparativă a relevat că alegerea între Godot și Unity nu poate fi redusă la o simplă decizie de "mai bun sau mai rău", ci depinde de cerințele specifice ale proiectului, bugetul disponibil, experiența echipei și platformele țintă.

Analiza comparativă a evidențiat punctele forte ale fiecărei platforme:

- Godot se remarcă prin ușurința în utilizare, licența open-source, eficiența resurselor, arhitectura bazată pe noduri și performanța excelentă pentru jocuri 2D;
- Unity excelează prin ecosistemul matur, suportul comunității, instrumentele avansate de 3D, documentația extinsă și integrarea cu platforme multiple.

Provocările identificate includ: curba de învățare pentru GDScript, comunitatea mai restrânsă pentru Godot, costurile de licențiere pentru Unity, și dependența de actualizări care pot modifica fluxul de lucru stabilit.

În concluzie, lucrarea elaborată prezintă o analiză comparativă valoroasă pentru dezvoltatorii de jocuri, oferind informații esențiale pentru alegerea platformei optime în funcție de cerințele specifice ale proiectului. Prin implementarea ambelor jocuri pe ambele motoare, s-a demonstrat practic cum fiecare platformă poate fi utilizată eficient pentru a obține rezultate similare, dar cu abordări și resurse diferite. Această cercetare contribuie la înțelegerea mai profundă a ecosistemului de dezvoltare a jocurilor și oferă un ghid practic pentru dezvoltatorii începători și avansați în luarea deciziilor informate privind alegerea tehnologiei potrivite pentru proiectele lor.

BIBLIOGRAFIE

1. *Video game industry - Statistics & Facts* [online] [accesat 10.02.2024]. Disponibil: https://www.statista.com/topics/868/video-games/#topicOverview
2. *What was the Great Video Game Crash of 1983?* [online] [accesat 10.02.2024]. Disponibil: https://blog.bugsplat.com/great-video-game-crash-1983/
3. *The history of game engines* [online] [accesat 10.02.2024]. Disponibil: https://dev.to/ispmanager/the-history-of-game-engines-from-assembly-coding-to-photorealism-and-ai-d9h
4. GREGORY, JASON, *Game Engine Architecture.* Third Edition. Boca Raton: CRC Press, 2018. 1240 p. ISBN 978-1138035454.
5. ERIC LENGYEL, *Foundations of Game Engine Development, Volume 2: Rendering,* Terathon Software LLC, 2019. 412 p. 978-0985811754.
6. IAN MILLINGTON, *Game Physics Engine Development* (Series In Interactive 3D Technology). Boca Raton: CRC Press, 2007. 480 p. ISBN 978-0123694713.
7. *Analyzing Strengths and Weaknesses of Modern Game Engines* [online] [accesat 10.02.2024]. Disponibil: https://www.ijcte.org/vol15/IJCTE-V15N1-1330.pdf
8. *Top Game Development Trends in 2025* [online] [accesat 10.02.2024]. Disponibil: https://maticz.com/game-development/trends#:~:text=The%20intersection%20of%20technology%20and,experience%20immersive%20entertainment%20and%20socialization.
9. *Cross-Platform Game Development:When Versatility Matters* [online] [accesat 10.02.2024]. Disponibil:https://kevurugames.com/blog/cross-platform-game-development-when-versatility-matters/
10. *DevOps Practices in the Gaming Industry — Differences with Traditional Industries* [online] [accesat 10.02.2024]. Disponibil: https://medium.com/globant/devops-practices-in-the-gaming-industry-differences-with-traditional-industries-1146d0d21ca3
11. *What Are Live Service Games (And Why Are They So Polarizing)?* [online] [accesat 10.02.2024]. Disponibil: https://www.howtogeek.com/what-are-live-service-games-and-why-are-they-so-polarizing/
12. *Indie Games: A Triple Win for Telcos, Developers, and Users* [online] [accesat 10.02.2024]. Disponibil:https://mondia.com/indie-games-a-triple-win-for-telcos-developers-and-users/#:~:text=According%20to%20VGI%2C%20the%20number,the%20games%20on%20the%20platform.
13. *What is virtual reality gaming (VR gaming)?* [online] [accesat 10.02.2024]. Disponibil: https://www.techtarget.com/whatis/definition/virtual-reality-gaming-VR-gaming

14. *Global games market forecast to decline in 2022* [online] [accesat 10.02.2024]. Disponibil: https://www.ampereanalysis.com/insight/global-games-market-forecast-to-decline-in-2022#:~:text=The%20latest%20outlook%20from%20Ampere,first%20six%20months%20of%202022.
15. *Differences between playing games on mobile versus on desktop* [online] [accesat 10.02.2024].Disponibil:https://end3r.com/p/mobile-vs-desktop.html#:~:text=mobile%20and%20desktop.-,While%20both%20platforms%20have%20their%20own%20unique%20features%20and%20advantages,graphics%2C%20performance%2C%20and%20control.
16. *Unity engine maker says sorry after runtime fee price plan backlash* [online] [accesat 10.02.2024]. Disponibil: https://www.bbc.com/news/newsbeat-66810296
17. *Godot License* [online] [accesat 10.02.2024]. Disponibil: https://godotengine.org/license/
18. *Godot Vs Unity: Which is the Ideal Choice for You* [online] [accesat 10.02.2024]. Disponibil: https://hennyjones.medium.com/godot-vs-unity-which-is-the-ideal-choice-for-you-8dd36beb71a5
19. *Advanced programming and code architecture* [online] [accesat 10.02.2024]. Disponibil: https://unity.com/how-to/advanced-programming-and-code-architecture
20. *Unity 6 User Manual* [online] [accesat 10.02.2024]. Disponibil: https://docs.unity3d.com/Manual/UnityManual.html
21. *Technology Stack Genshin Impact* [online] [accesat 10.02.2024]. Disponibil: https://fork.ai/lookup/genshin-impact-59381
22. *Cuphead - A Game | Made with Unity* [online] [accesat 10.02.2024]. Disponibil: https://unity.com/made-with-unity/cuphead
23. *How to Develop a Game Like Hearthstone - SDLC Corp* [online] [accesat 10.02.2024]. Disponibil:https://sdlccorp.com/post/how-to-develop-a-game-like-hearthstone/#:~:text=Which%20technologies%20are%20used%20for,C%23%3A%20Primary%20language%20in%20Unity.
24. *Among Us | Unity Case Study* [online] [accesat 10.02.2024]. Disponibil: https://unity.com/case-study/innersloth-among-us
25. *Monument Valley – a fresh look at AR implementation in existing games* [online] [accesat 10.02.2024]. Disponibil: https://www.linkedin.com/pulse/monument-valley-fresh-look-ar-implementation-existing-tao-xu-%E5%BE%90%E6%B6%9B-
26. *Keeping the Beat: Porting Beat Saber to OpenXR for An Improved Developer Experience* [online] [accesat 10.02.2024]. Disponibil: https://www.khronos.org/blog/keeping-the-beat-porting-beat-saber-to-openxr-for-an-improved-developer-experience#:~:text=Beat%20Saber%20was%20originally%20developed,manage%20four%20different%20VR%20implementations.

27. *Why Godot is right for you* [online] [accesat 12.11.2024]. Disponibil: https://godotengine.org/features/#design
28. *Godot Engine 4.4 documentation in English* [online] [accesat 02.02.2024]. Disponibil: https://docs.godotengine.org/en/stable/
29. *Sonic Colors: Ultimate* [online] [accesat 12.11.2024]. Disponibil: https://tcrf.net/Sonic_Colors:_Ultimate
30. *The Interactive Adventures of Dog Mendonça and Pizzaboy* [online] [accesat 10.02.2024]. Disponibil: https://www.pcgamingwiki.com/wiki/The_Interactive_Adventures_of_Dog_Mendon%C3%A7a_and_Pizzaboy
31. *Kingdoms Of The Dump* [online] [accesat 11.11.2024]. Disponibil: https://kingdomsofthedump.com/
32. *Wargrove* [online] [accesat 11.11.2024]. Disponibil: https://wargroovewiki.com/Wargroove_Wiki
33. *Cruelty Squad review* [online] [accesat 11.11.2024]. Disponibil: https://www.pcgamer.com/cruelty-squad-review/
34. *Cassette Beasts* [online] [accesat 11.11.2023]. Disponibil: https://www.cassettebeasts.com/
35. *Undead Survivor* [online] [accesat 11.11.2024]. Disponibil: https://assetstore.unity.com/packages/2d/undead-survivor-assets-pack-238068
36. *Survivors Starter Kit* [online] [accesat 11.11.2024]. Disponibil: https://godotengine.org/asset-library/asset/2336
37. *Asus Zenbook 14 UM425 – Tech Specs* [online] [accesat 11.11.2024]. Disponibil: https://www.asus.com/laptops/for-home/zenbook/zenbook-14-um425/techspec/

DECLARAȚIA PRIVIND ASUMAREA RĂSPUNDERII

Subsemnatul, Cazacu-Condrat Dumitru, declar pe răspundere personală că materialele prezentate în teza de master sunt rezultatul propriilor cercetări și realizări științifice. Conștientizez că, în caz contrar, urmează, să suport consecințele în conformitate cu legislația în vigoare.

„___”„_____” 2025 Cazacu-Condrat Dumitru