MINISTERUL EDUCAȚIEI ȘI CERCETĂRII AL REPUBLICII MOLDOVA
UNIVERSITATEA TEHNICĂ A MOLDOVEI
FACULTATEA CALCULATOARE, INFORMATICĂ ȘI MICROELECTRONICĂ
DEPARTAMENTUL INGINERIA SOFTWARE ȘI AUTOMATICĂ

DOCUMENTATIE

la lucrarea de laborator nr. 4

la Programarea Orientata pe Obiecte

Tema:Simulare Ecosistem

Elaborat: st. gr. TI-231 Maxian Alexandru

Verificat: asist. univ., Coșeru Cătălin

Chişinău – 2024

Descrierea claselor şi ierarhiilor

Acest cod implementează un ecosistem virtual utilizând Pygame, în care animalele şi plantele interacţionează într-un mediu definit de condiţii meteorologice şi resurse disponibile.

1. Clasa Abstractă: Ecosistem

Aceasta este clasa de bază abstractă pentru toate entităţile din ecosistem, precum plantele, animalele şi condiţiile meteorologice. Conţine:

Atribute comune precum: `nume`, `pozitie`, `energie`, `total_age`, `age_remaning`, `energie_ecosistem`, etc.
Metoda abstractă `actioneaza()`, care trebuie implementată în clasele derivate.

---

2. Clase Derivate din Ecosistem

a. Clasa `StatusEcosistem`

Rol: Monitorizează şi furnizează un rezumat al stării ecosistemului (număr de plante, animale, energie totală).
Atribute:
`numar_animale`, `numar_iepuri`, `numar_arici`, `numar_vulpi`, `cantitate_plante`.
Metode:
`status()`: Returnează un rezumat al stării ecosistemului.
`actioneaza()`: Afişează starea curentă.

b. Clasa `weather_conditions`

Rol: Modelează condiţiile meteorologice care influenţează ecosistemul.
Atribute:
`state` (ex. "normal"), imagini pentru ploaie (`imagine_rain`) şi secetă (`imagine_seceta`), `image_to_show` (imaginea curentă).
Metode:
`weather_type(nume, rata_supravetuire)`: Modifică rata de supravieţuire în funcţie de vreme şi adaugă entităţi în ecosistem.
`draw(window)`: Afişează starea meteorologică pe ecran.

---

3. Clasele Animalelor (Derivate din Ecosistem)

Clasa Abstractă: `Animal`

Este clasa de bază pentru toate animalele.
Atribute:
`x`, `y` (poziţia), `speed`, `imagine` (imaginea asociată).
Metode:
`deplaseaza()`: Schimbă poziţia animalului aleatoriu.
`reproduce()`: Permite reproducerea între animale de acelaşi tip, cu limitări de timp şi populaţie.
`status():` Returnează starea animalului.

Metode abstracte: `tip_hrana()` **şi** `mananca()`.

b. Clasele Specifice Animalelor

Erbivor:
Mănâncă plante și le reduce energia sau le elimină.
Creşte energia proprie cu fiecare hrană consumată.

Carnivor:
Mănâncă erbivore sau omnivore, eliminându-le din ecosistem.
Creşte energia proprie cu fiecare pradă consumată.

Omnivor:
Se hrănește atât cu plante, cât și cu erbivore, eliminându-le din ecosistem.

---

4. Clasa Plante (Derivată din Ecosistem)

Rol: Modelează plantele din ecosistem.
Atribute:
`humidity` (umiditate necesară pentru creştere).
`imagine` (imaginea plantei).
Metode:
`status()`: Returnează starea plantei.
`draw(window)`: Afişează planta pe ecran.

---

5. Funcţii Auxiliare

`coliziune(animal1, animal2)`: Verifică dacă două animale sunt la o distanţă suficient de mică pentru a interacţiona.
`plant_grow(plante)`: Adaugă plante noi în ecosistem, dacă umiditatea este suficientă.

Ierarhia Claselor

```
Ecosistem (abstractă)
├── StatusEcosistem
├── weather_conditions
├── Plante
└── Animal (abstractă)
    ├── Erbivor
    ├── Carnivor
    └── Omnivor
```

Codul este o simulare complexă a unui ecosistem, cu entități și interacțiuni definite. Clasele sunt organizate pe categorii de entități (plante, animale, condiții meteorologice), fiecare având roluri și comportamente specifice.

Explicația fiecărei metode

Clasa `Ecosistem`

Aceasta este o clasă abstractă care definește un ecosistem general. Este baza pentru toate entitățile din ecosistem (plante, animale, vreme, etc.).

`__init__(self, nume, pozitie, energie, total_age)`:
Inițializează proprietăți comune: nume, poziție, energie, vârsta totală, rata de supraviețuire. Valori implicite pentru energie și supraviețuire sunt setate.

`actioneaza(self)`:
Metodă abstractă ce trebuie implementată de toate clasele derivate. Definește ce acțiuni execută entitatea respectivă în ecosistem.

Clasa `StatusEcosistem`

Această clasă gestionează și afișează starea ecosistemului, inclusiv energia totală, numărul de plante și animale.

`__init__(self, numar_plante, numar_animale, numar_iepuri, numar_arici, numar_vulpi)`:

Inițializează starea ecosistemului cu date despre plante, animale și energia totală.

`status(self)`:

Returnează un șir de caractere ce conține starea curentă a ecosistemului.

`actioneaza(self)`:

Afișează starea ecosistemului folosind metoda `status()`.

Clasa `weather_conditions`

Reprezintă condițiile meteorologice ce pot influența ecosistemul.

`__init__(self, nume, x, y, imagine_rain, imagine_seceta)`:

Inițializează poziția și imaginile asociate stărilor meteo (ploaie și secetă).

`actioneaza(self)`:

În acest moment, este o metodă goală care poate fi extinsă pentru implementări viitoare.

`weather_type(self, nume, rata_supravetuire)`:

Modifică starea vremii (ploaie sau secetă) în funcție de rata de supraviețuire a unei specii. Generează efecte vizuale și ajustează populația animalelor.

`draw(self, window)`:

Afișează imaginea curentă a vremii pe fereastra principală, dacă este activată.

Clasa `Animal`

Clasă abstractă ce reprezintă baza pentru ierbivore, omnivore și carnivore.

`__init__(self, nume, x, y, energie, total_age, speed, imagine)`:

Inițializează poziția, energia, viteza și imaginea animalului.

`actioneaza()`:

Metodă abstractă, goală, ce va fi suprascrisă în subclase.

`reproduce(self, animale, animale_noi)`:

Permite reproducerea animalelor de același tip dacă sunt aproape și timpul necesar s-a scurs.
Adaugă noul animal în ecosistem și actualizează populația.

`tip_hrana(self)`:

Metodă abstractă ce definește tipul de hrană consumat de animal.

`mananca(self, animale, plante)`:

Metodă abstractă ce definește comportamentul de hrănire.

`deplaseaza(self)`:

Permite mișcarea animalului în spațiul ecosistemului într-o direcție aleatorie.

`status(self)`:

Returnează un șir cu detalii despre energia și vârsta rămasă a animalului.

`draw(self, window)`:

Afișează animalul pe fereastra principală.

Clasa `Erbivor`

Subclasă a clasei `Animal`. Reprezintă animalele care consumă doar plante.

`__init__(self, nume, x, y, energie, total_age, speed, imagine)`:

Apelarea constructorului clasei `Animal`.

`tip_hrana(self)`:

Returnează tipul de hrană: "ierbivor".

`mananca(self, animale, plante)`:

Animalul consumă plante dacă este suficient de aproape de ele. Crește energia și reduce durata de viață a plantei.

Clasa `Carnivor`

Subclasă a clasei `Animal`. Reprezintă animalele care consumă doar alte animale.

`__init__(self, nume, x, y, energie, total_age, speed, imagine)`:

Apelarea constructorului clasei `Animal`.

`tip_hrana(self)`:

Returnează tipul de hrană: "carnivor".

`mananca(self, animale, plante)`:

Animalul vânează ierbivore sau omnivore dacă acestea sunt suficient de aproape. Crește energia și elimină prada din ecosistem.

Clasa `Omnivor`

Subclasă a clasei `Animal`. Reprezintă animalele care consumă atât plante, cât și alte animale.

`__init__(self, nume, x, y, energie, total_age, speed, imagine)`:

Apelarea constructorului clasei `Animal`.

`tip_hrana(self)`:

Returnează tipul de hrană: "omnivor".

`mananca(self, animale, plante)`:

Consumul poate fi atât plante cât și ierbivore, în funcție de proximitate.

Clasa `Plante`

Reprezintă plantele din ecosistem.

`__init__(self, nume, pozitie, energie, age_remaning, imagine)`:

Inițializează poziția, energia, vârsta rămasă și imaginea plantei.

`status(self)`:

Returnează un șir cu detalii despre energie și vârsta rămasă.

`draw(self, window)`:

Afișează planta pe fereastra principală.

`actioneaza(self)`:

Metodă goală pentru viitoare implementări.

Funcţii externe:

`coliziune(animal1, animal2)`:

Verifică dacă două animale se află suficient de aproape pentru a interacționa.

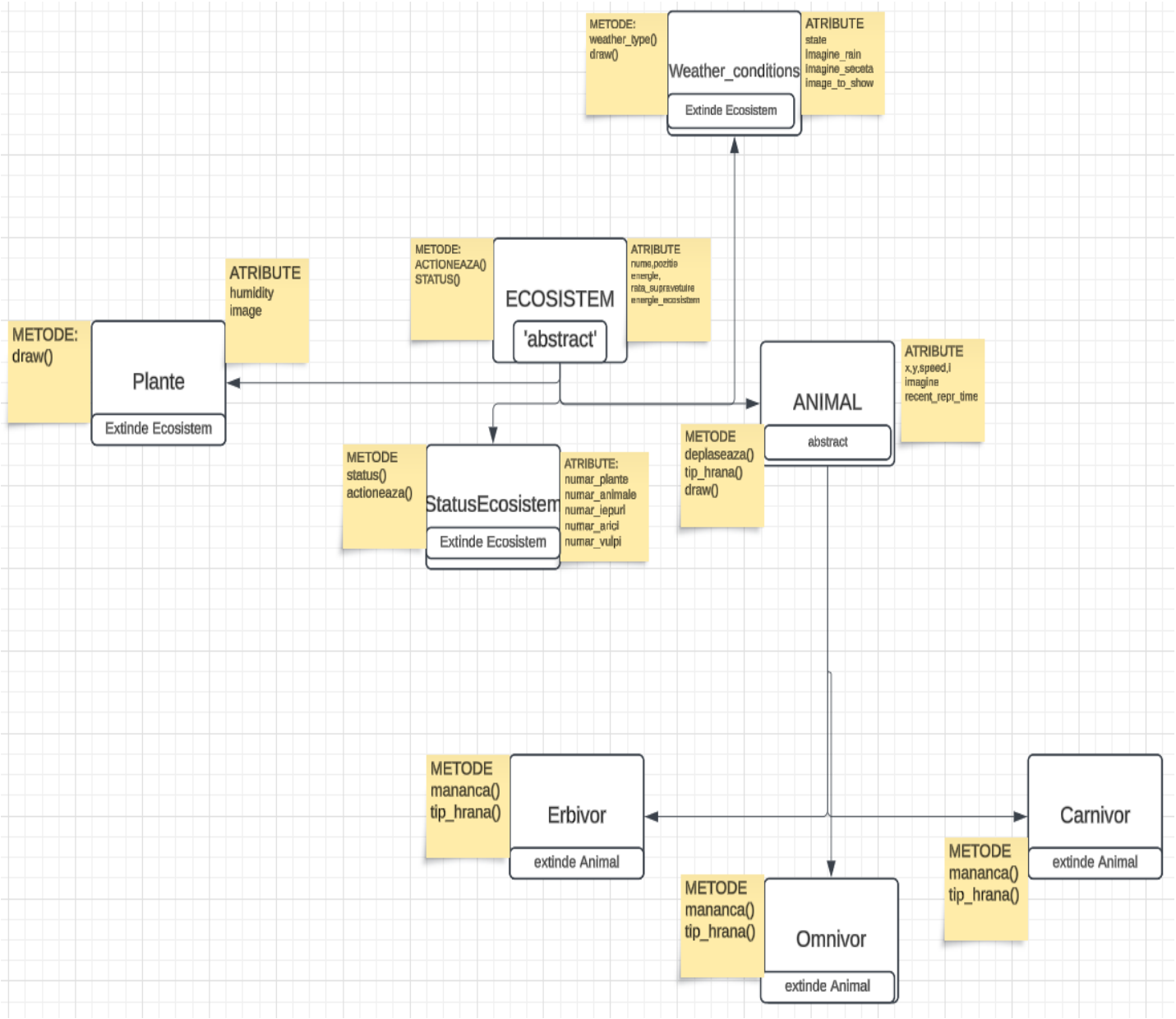
`plant_grow(plante)`:

Permite creșterea plantelor în ecosistem la intervale regulate.

`redraw_game_window()`:

Reîmprospătează fereastra principală, afișând fundalul, plantele, animalele şi vremea.

Diagrama UML



Scenarii de utilizare (exemple concrete de rulare a simulării).

Simularea "Ecosistem" oferă un poxibilitate pentru gestionarea unui ecosistem artificial în care animalele, plantele și condițiile meteo interacționează dinamic.

Scenariul 1: Ecosistem echilibrat

Configurație inițială:

5 iepuri, 3 arici, 2 vulpi, 10 plante.
Umiditate inițială: 100.

Evoluție:

Animalele se deplasează aleatoriu, iepurii consumă plante, iar vulpile vânează iepuri și arici.
Plantele cresc constant dacă umiditatea rămâne peste 30.
Populația se stabilizează deoarece rata reproducerii este controlată de limitele maxime de populație.

Scenariul 2: Criză de hrană pentru erbivore

Configurație inițială:

2 iepuri, 2 arici, 3 vulpi, 3 plante.
Umiditate inițială: 50.

Evoluție:

Iepurii și aricii consumă rapid plantele disponibile.
Plantele nu cresc din cauza scăderii umidității sub 30.
Populația erbivorilor scade drastic. Odată ce numărul iepurilor scade sub 3, apare fenomenul de ploaie (vizualizat) și umiditatea ecosistemului crește, favorizând regenerarea plantelor.

Scenariul 3: Supraîncărcare cu carnivore

Configurație inițială:

1 iepure, 1 arici, 5 vulpi, 10 plante.
Umiditate inițială: 80.

Evoluție:

Vulpile vânează rapid singurul iepure și ariciul.
Lipsa hranei pentru vulpi duce la scăderea energiei lor, provocând dispariția treptată a populației de carnivore.
Plantele rămân neconsumate, crescând în exces.

Scenariul 4: Reechilibrarea ecosistemului

Configurație inițială:

2 iepuri, 2 arici, 1 vulpe, 5 plante.
Umiditate inițială: 100.

Evoluție:

Ecosistemul începe cu o populație mică și un nivel moderat de hrană.
Animalele se reproduc treptat: iepurii cresc în număr și consumă plante, dar sunt controlați de vulpe.
Când populația vulpilor scade sub 2, vulpile reproduc noi indivizi, reechilibrând ecosistemul.

Scenariul 5: Deșertificare și efecte meteo severe

Configurație inițială:

5 iepuri, 3 arici, 2 vulpi, 10 plante.
Umiditate inițială: 20.

Evoluție:

Datorită umidității scăzute, plantele nu se pot regenera, iar ecosistemul suferă un colaps treptat.
Apar fenomene meteo extreme (secetă vizualizată).
Reechilibrarea ecosistemului este forțată de condițiile meteo care cresc umiditatea după un prag critic, reluând ciclul de viață.

Scenariul 6: Studiu al reproducerii

Configurație inițială:

4 iepuri (2 perechi), 2 arici, 1 vulpe, 8 plante.
Umiditate inițială: 100.

Evoluție:

Iepurii se reproduc rapid dacă găsesc parteneri și energia lor rămâne peste un nivel minim.
Vulpea limitează populația de erbivore consumând iepuri.
Experimentatorul poate ajusta variabilele precum rata de reproducere sau viteza animalelor pentru a observa impactul asupra ecosistemului.

Aceste scenarii pot fi folosite pentru a înțelege mai bine interacțiunile din natură sau pentru a testa diferite ipoteze privind echilibrul ecologic. De asemenea, ele pot fi extinse prin adăugarea unor noi specii sau prin ajustarea condițiilor de mediu pentru a observa impactul diverselor variabile.

Dificultăţile întâlnite şi soluţiile adoptate.

1. Gestionarea interacţiunilor între entităţile din ecosistem

Problemă: Implementarea logicii pentru coliziuni, interacțiuni (hrănire, reproducere) și gestionarea stării fiecărei entități s-a dovedit complexă, mai ales pentru a sincroniza comportamentele diferitelor tipuri de entități (erbivore, carnivore, omnivore, plante).
Soluţie:

Am creat clase bine definite cu metode abstracte și suprascrise, ceea ce a permis extinderea logicii pentru fiecare entitate în mod separat.
Am utilizat o funcție comună `coliziune()` pentru a evalua distanța dintre două entități și am implementat logica de interacțiune direct în clasele specifice.

2. Gestionarea reproducerii animalelor

Problemă: Prevenirea unei creșteri necontrolate a populației. În cazul în care reproducerea nu avea limite temporale sau numerice, populațiile creșteau exponențial.
Soluţie:

Am introdus un mecanism de verificare a timpului (`pygame.time.get_ticks()`) și am impus o limită de timp (cooldown de 5 secunde) pentru reproducere.
Am definit populații maxime pentru fiecare tip de animal și am blocat reproducerea dacă populația a atins limita.

3. Simularea condiţiilor meteorologice

Problemă: Afișarea corectă a schimbărilor meteorologice și sincronizarea lor cu alte procese ale ecosistemului, precum creșterea plantelor sau scăderea umidității.
Soluţie:

Am creat o clasă separată `weather_conditions` pentru a gestiona stările meteorologice și a afișa vizual schimbările (ploaie, secetă).
Am folosit un timer pentru a controla durata afișării imaginilor meteorologice și pentru a reseta condițiile după un interval.

4. Sincronizarea diferitelor procese

Problemă: Asigurarea că procesele precum creșterea plantelor, deplasarea animalelor, interacțiunile și schimbările meteorologice nu interferează sau nu creează conflicte în timpul execuției.
Soluţie:

Am folosit bucla principală de joc pentru a apela metodele relevante într-un anumit ritm definit de FPS.
Am prioritizat procesele mai frecvente (deplasare, interacțiuni) și am stabilit intervale mai mari pentru evenimente mai rare (creșterea plantelor, schimbările meteorologice).

5. Optimizarea performanţei

Problemă: Pe măsură ce numărul entităților din ecosistem creștea, performanța aplicației începea să scadă.
Soluţie:

Am implementat o listă temporară `animale_noi` pentru a adăuga noi animale după ce ciclul curent s-a terminat, prevenind modificarea simultană a listei de animale.
Am redus complexitatea verificărilor utilizând liste derivate pentru fiecare tip de entitate.

6. Aspectul vizual şi scalarea imaginilor

Problemă: Unele imagini pentru animale și plante nu erau bine scalate, iar poziționarea pe hartă era haotică.

Soluţie:

Am redimensionat imaginile folosind `pygame.transform.scale()` pentru a avea dimensiuni uniforme.

Am implementat limite pentru coordonatele entităților, astfel încât să nu iasă din ecran.

7. Gestionarea resurselor ecosistemului

Problemă: Echilibrarea consumului de resurse (plante) cu populația de animale şi simularea unui ciclu realist de viață.

Soluţie:

Am introdus umiditatea ecosistemului ca un factor care influențează creșterea plantelor. Dacă umiditatea scade sub un prag, plantele încetau să crească.

Am ajustat rata de supraviețuire a animalelor pe baza resurselor disponibile şi a condițiilor meteorologice.

8. Testarea simulării în diferite scenarii

Problemă: Verificarea comportamentului ecosistemului în scenarii precum populații minime, umiditate scăzută sau colapsul resurselor.

Soluţie:

Am creat scenarii de testare (cum ar fi reducerea umidității la valori extreme sau reducerea populației de erbivore) pentru a verifica dacă logica implementată se adaptează corect.

Prin rezolvarea acestor dificultăți, simularea a devenit mai robustă, realistă şi plăcută vizual

Concluzii:

Codul proiectului demonstrează o aplicare coerentă și bine structurată a principiilor programării orientate pe obiecte (OOP), integrând concepte fundamentale precum **moştenirea**, **polimorfismul**, **suprascrierea** și **interfețele** pentru a crea o simulare robustă și scalabilă.

---

1. Moştenirea

Am utilizat moştenirea pentru a defini o ierarhie clară între clase, ceea ce a permis reutilizarea codului şi extinderea funcţionalităților. De exemplu:

Clasa de bază `Entitate` a fost moştenită de clase specifice precum `Animal` şi `Planta`.
Fiecare tip de animal (erbivor, carnivor) extinde clasa de bază `Animal`, moştenind atributele şi metodele comune precum `deplasare()` sau `coliziune()`, ceea ce a redus duplicarea codului.

---

2. Polimorfismul

Proiectul a demonstrat utilizarea polimorfismului prin intermediul metodelor care au fost redefinite pentru a răspunde nevoilor fiecărei clase derivate:

> De exemplu, metoda `interactiune()` a fost implementată diferit în clasele derivate pentru erbivore și carnivore, adaptând comportamentul acestora în funcție de tipul lor.
> Acest principiu a permis tratarea uniformă a tuturor entităților în cadrul simulării, indiferent de tipul lor concret, folosind metode comune apelate din liste mixte de entități.

---

3. Suprascriere și supraîncărcare

> **Suprascriere:** Metodele moștenite din clasele de bază au fost suprascrise pentru a adăuga funcționalități specifice. De exemplu:
>
> > Metoda `deplasare()` a fost redefinită pentru carnivore, pentru a include logica de urmărire a prăzii, spre deosebire de erbivore, care se deplasează aleatoriu.
>
> **Supraîncărcare:** Deși limbajul Python nu suportă supraîncărcarea nativă, s-a utilizat flexibilitatea argumentelor implicite pentru a crea metode care pot primi parametri diferiți, adaptând comportamentul metodei.

Proiectul realizat reprezintă o implementare bine gândită a unui ecosistem virtual, combinând concepte avansate de programare orientată pe obiecte (OOP) cu o structură modulară și scalabilă. Am urmărit să creez un mediu în care diferitele entități să interacționeze într-un mod realist, punând accent pe reutilizarea codului, extensibilitate și claritate în design.

Implementarea principiilor OOP, precum moștenirea, polimorfismul, suprascrierea, supraîncărcarea și utilizarea interfețelor, a permis dezvoltarea unui proiect flexibil și ușor de extins. Aceste abordări au redus duplicarea codului și au asigurat posibilitatea adăugării ulterioare a unor noi tipuri de entități sau funcționalități fără a afecta structura generală a proiectului.

Totodată, dificultățile întâlnite pe parcurs – de la gestionarea interacțiunilor complexe până la implementarea logicii dinamice – au fost rezolvate printr-o abordare metodică și prin utilizarea tehnicilor adecvate de programare. Rezultatul final este un ecosistem virtual funcțional, care nu doar simulează comportamentele de bază ale entităților, dar oferă și o platformă ideală pentru viitoare optimizări și extinderi.

În concluzie, proiectul reprezintă o demonstrație practică a conceptelor teoretice studiate, reușind să îmbine creativitatea cu rigurozitatea tehnică. Acesta nu doar că își îndeplinește obiectivele propuse, dar constituie și o bază solidă pentru explorări și proiecte ulterioare în domeniul simulărilor și modelărilor software.