Sewer Escape

Samachiș Eduard-Iulian
1208A

Proiectarea contextului:

Cookie, protagonistul nostru, este un simplu motan portocaliu care nu are parte de prea multă aventură. Însă lucrurile urmau să se schimbe.

Iată că, într-o zi obișnuită, pe când Cookie abia ce făcuse încălzirea penrtu antrenamentele de prins șoareci, pe lângă o gură mică de canalizare se afla a fi, la prima vedere, un șoricel. O pradă perfectă pentru un prim exercițiu. Al nostru motan nu a stat pe gânduri și, cu ghearele pregătite, a alergat înspre el. Șoricelul, observând atacul, a alergat repede în canalizare și, după el, Cookie.

Motanul portocaliu se trezește amețit de jos. Ce s-a întâmplat? Canalizarea în care nimerise era punctul principal de legare a întregii rețele de salubritate a orașului. Era uriașă. Căzând de la câțiva metri pe o conductă, era aproape să încheie una din cele 9 vieți.

Uitându-se pierdut împrejur, vede în depărtare șoricelul care de fapt era un pui de șobolan. Nu orice șobolan, ci fiul conducătorului celor din canalizare. Atât a auzit de la el: ”Nu ieși viu de aici!”.

În ziua aceea Cookie își începea marea aventură a vieții sale. Va reuși să treacă de inamici și să găsească o cale de ieșire?

Proiectarea sistemului:

Jocul este un platformer cu perspectiva 2D din lateral. În joc vor apărea zone cu diferite interacțiuni:

- Zonele cu apă permit personajului să înoate;
- Zonele cu țepi iau o viață personajului;
- Atacul inamicilor iau o viață personajului.

Fiecare viață pierdută a personajului (din cele 3 care se dau la începutul unui nivel) vor reseta poziția caracterului la zona de start, sub conducta principală. Dacă se vor pierde toate viețile pe parcursul unui nivel, jocul se va relua de la nivelul 1.

Jucătorul va trebui să treacă fiecare nivel, ca în final să evadeze din canalizare. Antagoniștii – șobolanii din canal – vor reprezenta piedica principală în traversarea unui nivel, aceștia trăgând cu arma și îngreuna drumul spre ieșire.

Inamicii trag cu arma atunci când personajul intră în raza lor vizuală, în restul timpului aceștia patrulând pe o anumită zonă. Jucătorul poate sări peste focurile inamicilor, sau se poate pune jos (acțiunea de CRAWL) și nu va primi damage.

Fiecare nivel conține o superputere simbolizată printr-o lăbuță de pisică. Odată culeasă, superputerea permite jucătorului să omoare un inamic normal dintr-o lovitură, iar pentru Boss scade viața cu 2 unități/lovitură. Totodată, la ultimul nivel, superputerea îi oferă abilitatea jucătorului să sară foarte sus. Altfel, sunt necesare 3 lovituri pentru ca un inamic normal să fie doborât și 10 în cazul Boss-ului. În dreapta-sus va apărea iconița superputerii, dacă a fost găsită.

Pentru a trece la nivelul următor, e necesară găsirea cleștelui ce va fi folosit pentru deblocarea conductei, prin tăierea grilajului. Acesta se află la unul din inamici, astfel jucătorul va fi nevoit să se lupte până găsește cleștele. În dreapta-sus va apărea iconița lui, dacă a fost găsit.

La nivelul final va trebui să se confrunte cu regele șobolanilor. Acesta va avea 10 vieți, însă dacă jucătorul găsește superputerea, va muri din 5 lovituri. Dificultatea constă în faptul că împușcă cu o rapiditate mai mare, astfel fiind mai greu de ucis. Dacă regele va fi învins, motanul va ieși din canalizare și se va întoarce acasă, jucătorul câștigând.

Scorul reprezintă durata de finalizare a jocului, exprimată în secunde. Astfel, un scor cât mai mic poziționează jucătorul în topul clasamentului.

Personajul principal se va controla cu ajutorul tastaturii și va avea abilități precum: săritul, mersul simplu, alergatul, înotatul, târâtul, atacatul.

Tastă	***Comandă***
W ***Space***	***Săritură*** ***Cățărat*** ***Ridicare în picioare***
A	***Deplasare stânga***
D	***Deplasare dreapta***
S	***Poziție de mers târâș***
Left Mouse Button	***Atac***
Esc	***Pornire/Oprire pauză joc***
Shift + A / S	***Alergatul în stânga / dreapta (în poziția normală, în picioare)***

Proiectarea conținutului:

Personajul principal – Cookie – și animațiile pentru mers în picioare și atac:

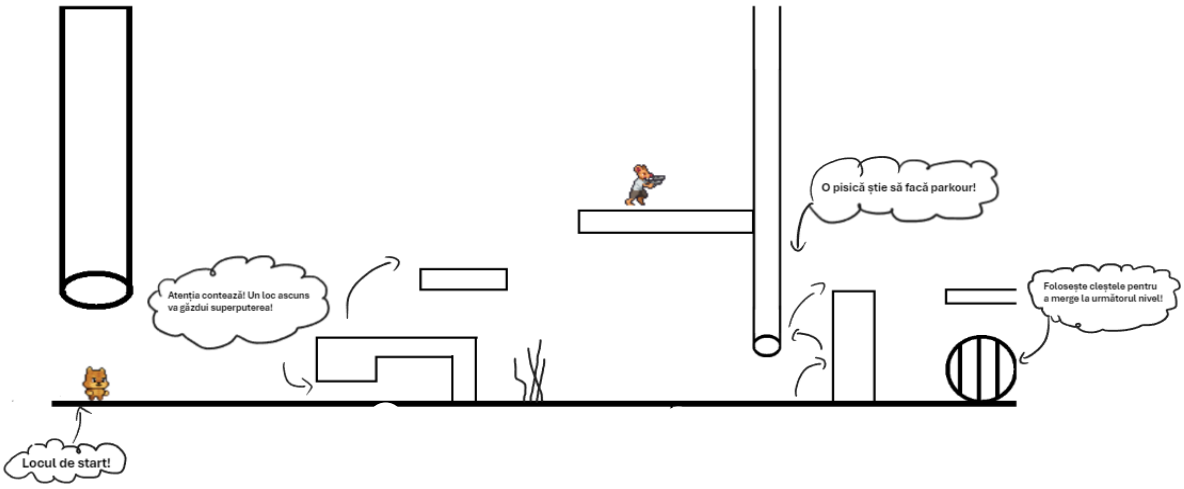
Inamicii – animația pentru mers și atac:

- Șobolanii normali:

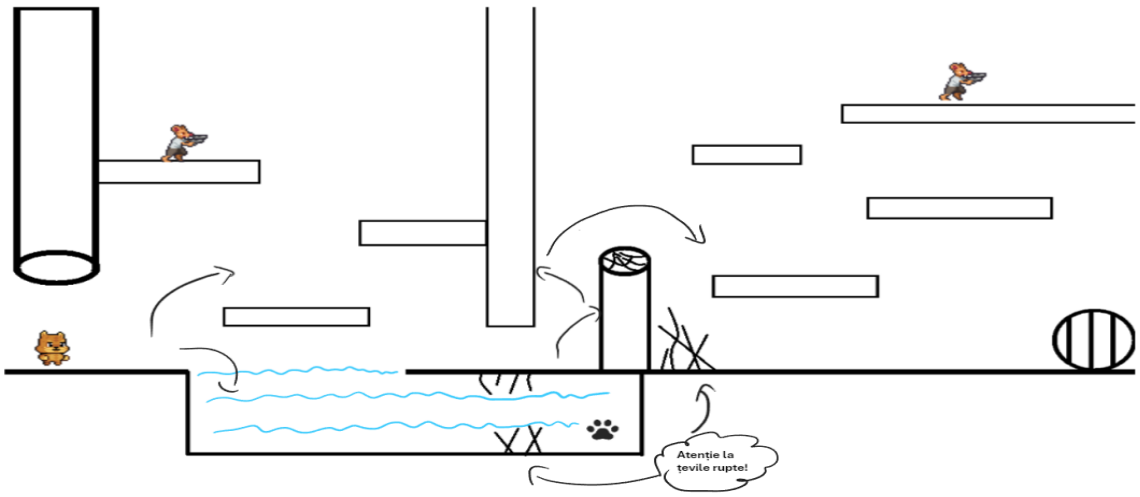
- Boss-ul:

Proiectarea nivelurilor:

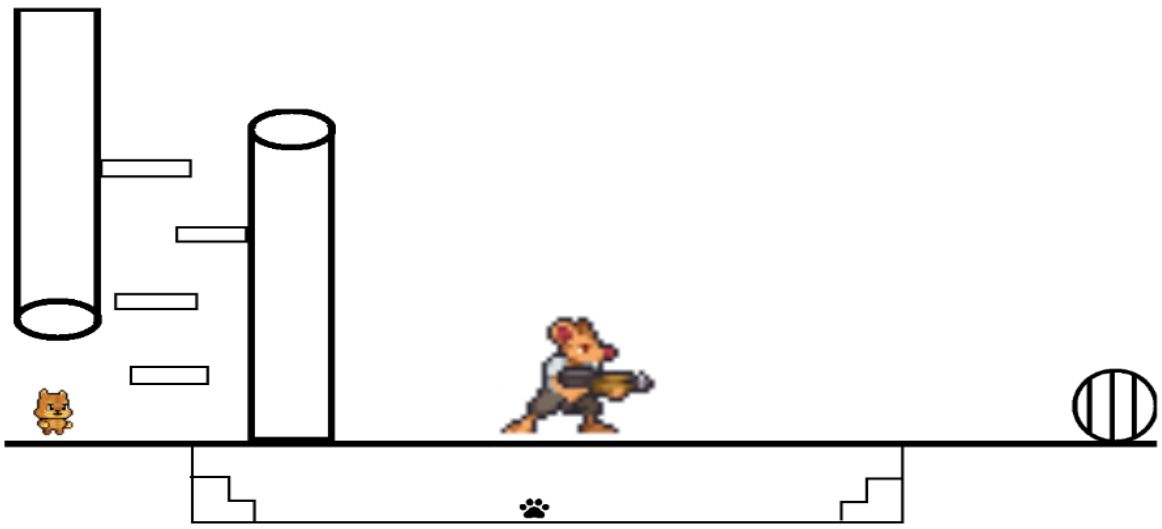
Nivelul 1:



Nivelul 2:



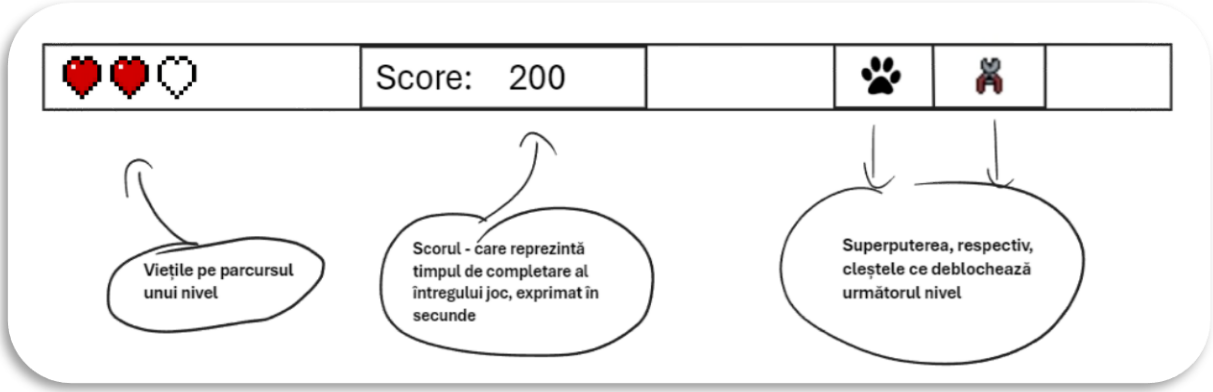
Nivelul 3 – final:



Proiectarea interfeței cu utilizatorul:

- Afișul jocului -

Va exista un meniu principal cu butoanele: New Game, Load Game, Settings, Leaderboard și Exit. Fereastra din timpul gameplay-ului va avea: viețile caracterului principal, casetele pentru superputere și clește și scorul (adică timpul de completare al jocului, exprimat în secunde).



Descrierea detaliată a claselor și pachetelor:

Pachetul **entities**:

Clasa **Entity** reprezintă abstractizarea conceptului de entitate din joc. Metodele principale și comune entităților sunt:

- updateAnimation() – pentru a afișa animația pe ecran;
- testGravity() – ce testează gravitația unei entități în corespondență cu harta nivelului;
- updateXPos() – ce actualizează poziția X a entității dacă aceasta se poate mișca;
- Metodele update(), updatePos(), setAnimation() și render() suprascrise în clasele derivate, au rol de actualizare a poziției, respectiv animației entității. Iar metoda takeDamage() reprezintă modul de în care viața entității scade, în funcție de anumiți factori.

Câmpuri importante ale clasei:

- hitBox – dreptunghiul cu rol în gestionarea coliziunilor;
- levelData – matricea nivelului;
- health – instanță a clasei HealthBar. Modelează viața entității.

Clasa **Player**, extinsă din Entity, reprezintă modelarea jucătorului. Metodele principale sunt:

- updatePos() – actualizează poziția player-ului in funcție de mai mulți factori: dacă pe hartă se detectează apă, dacă sare ori dacă player-ul se mișcă normal sau cu viteză;
- setAnimation() – actualizează animația player-ului în funcție de acțiunea lui sau de mediul în care se află (în apă va înota);
- getSpikeDamage() – gestionează cazul în care player-ul se află în coliziune cu țepii de pe hărți, primind damage prin metoda takeDamage();

- detectWater() – gestionează cazul în care player-ul se află în apă, gravitația schimbându-se și player-ul putând să se miște în sus și în jos, împachetate în metoda WaterBehaviour();
- LoadFromSave() – se apelează atunci când se face o încărcare din baza de date, pentru a actualiza tot ce ține de player direct și indirect (scor, iteme, poziția camerei, viață).

Clasa **SimpleEnemy**, extinsă din Entity, reprezintă modelarea inamicului simplu. Metodele principale sunt:

- updatePos() - inamicul patrulează o anumită zonă, mișcându-se constant ori în dreapta, ori în stânga.
- setAnimation() – setează animația inamicului, variind între animația de IDLE, WALK, SHOOTING și DEATH;
- shoot() – modelează acțiunea de atac a inamicului. Atunci când player-ul intră în raza de atac a inamicului, se activează animația și funcția de shooting, care se împletește cu cea a clasei **Bullet**, prin instanța **b**, descrisă mai jos;
- takeDamage() – modelează metoda de damage dată de player către inamic. Atunci când superputerea nu e selectată, viața inamicului scade cu 1, în caz contrar, moare dintr-o lovitură. Metoda testează dacă inamicul e instanță a clasei SimpleEnemy sau BossEnemy, pentru a gestiona comportamentul corect.
- die_if_attack() – metoda conturează contextul în care takeDamage() are loc, depinzând de poziția player-ului. Atunci când hitBox-ul unui inamic e în contact cu cel al player-ului, și player-ul este în modul de atac (variabila attacking), atunci inamicul primește damage;
- deathAnimation() – funcția desenează animația de DEATH a inamicului, într-un mod lent.

Clasa **BossEnemy**, extinsă din SimpleEnemy, reprezintă conceptualizarea Boss-ului din joc, cel care va fi la ultimul nivel și va trebui înfrânt pentru a câștiga jocul. Fiind o extindere a clasei inamicilor simpli, comportamentul e aproape identic. Diferențele constau în faptul că este mai greu de înfrânt: are 10 vieți, are o rază de atac mai mare și un atac mai rapid.

Clasa **Bullet** abstractizează conceptul de glonț, cel prin care atacă inamicul. Fiecare inamic are o instanță a clasei Bullet, prin care dobândește abilitatea de a ataca. Metodele principale și importante sunt:

- movement() – modelează mișcarea glonțului, în funcție de tipul de inamic. Atunci când inamicul nu atacă, glonțul este purtat de inamic, poziția lui fiind direct dependentă de cea a inamicului. Când atacul este pornit, glonțul pleacă pe o anumită distanță dată de variabila **shootingRange**.
- die_if_attack() – metoda se apelează atunci când inamicul atacă player-ul. Prin câmpul **hitBox**, care are implicații în coliziunile dintre obiecte, se testează dacă glonțul intră în contact cu player-ul, în caz afirmativ acesta luând damage.

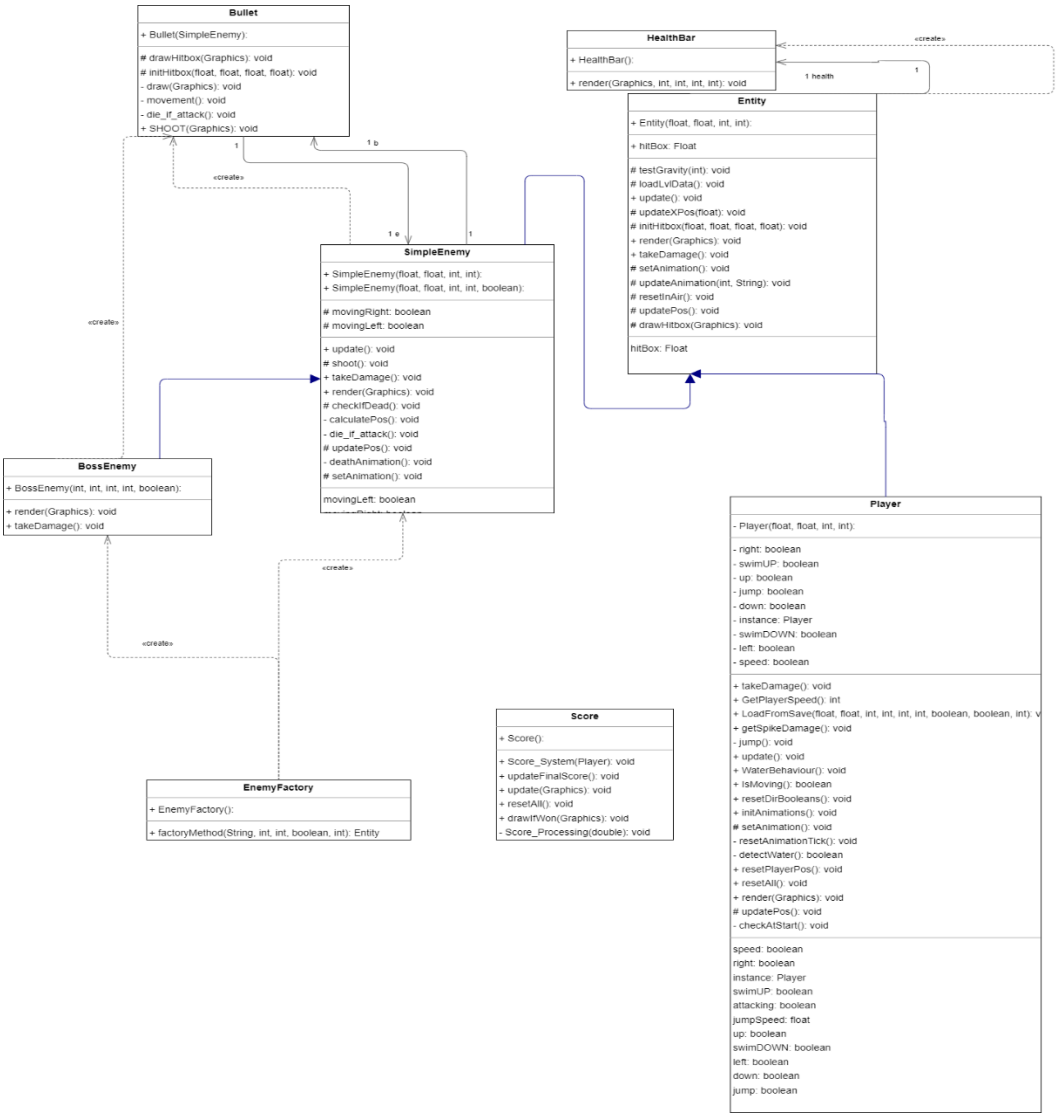
Clasa **HealthBar** încapsulează conceptul de viață a unei entități. Clasa este simplă, având doar un câmp, lifeCount, ce este inițializat implicit cu valoarea 3, putând fi modificat, și metoda render() – folosită la afișarea vieții unei entități.

Clasa **Score** se ocupă cu gestionarea și conceptul de scor din joc. Prin câmpul current_score, se păstrează scorul curent din joc, iar prin finalScore cel final, care va fi afișat la completarea jocului. Metodele importante sunt:

- Score_System() – dependentă de player, incrementează cu 1/60 scorul curent, deoarece sunt 60 frame-uri/secundă. În final scorul se incrementează cu 1/secundă. Deci cu cât mai mic scorul final, cu atât mai bine;
- Score_Processing() – descompune scorul dintr-un întreg într-un vector de întregi, cifră cu cifră, pentru a putea fi afișat;
- update() – cu rol în afișarea scorului pe ecran.

Clasa **EnemyFactory** implementează șablonul de proiectare Factory Method pentru gestionarea ușoară a celor două tipuri de inamici: simplu și Boss.

Diagrama UML a pachetului:



Pachetul **Inventory**

Clasa abstractă **ItemAbstractClass** modelează conceptul de item în joc. Implementarea este una simplă:

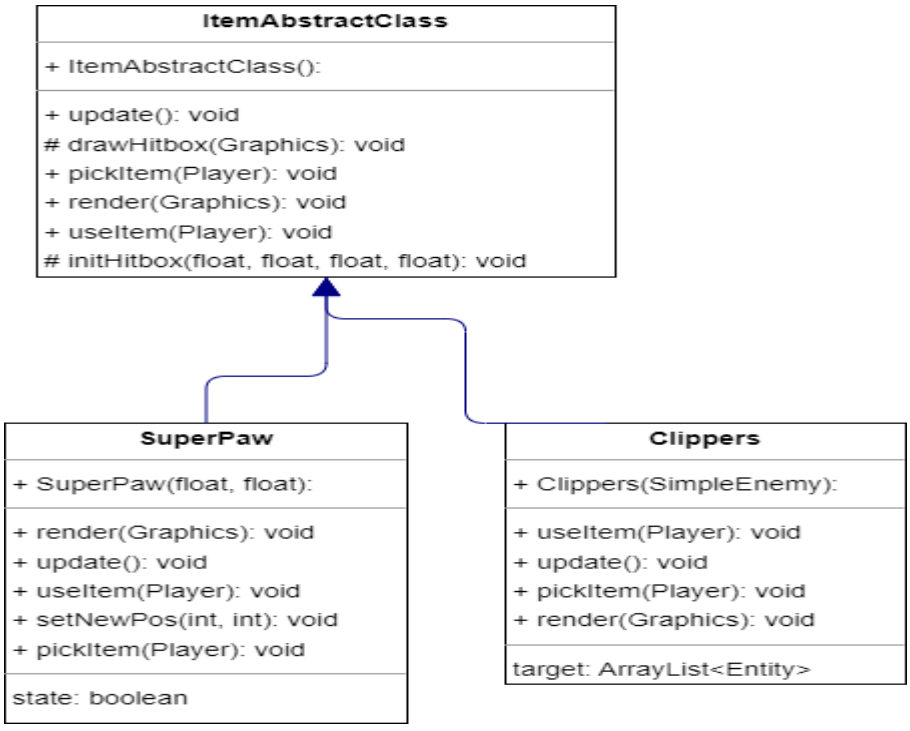
- metoda render() se ocupă cu afișatul itemului pe ecran în cazul în care a fost cules sau nu (câmpul collected);
- metodele pickItem() și useItem(), dependente de player, gestionează cazurile în care itemul încă nu a fost cules și cel în care, fiind cules, este folosit;
- metoda update() este suprascrisă în fiecare item separat;
- orice item are o coliziune bazată pe același sistem amintit și mai sus – câmpul **hitBox**.

Clasa **Clippers**, extinsă din ItemAbstractClass, implementează cleștele necesar pentru a trece de la un nivel la altul. O instanță a unui clește este dependentă de un inamic, care va trebui ucis pentru ca player-ul să colecteze cleștele.

Clasa **SuperPaw**, extinsă din ItemAbstractClass, implementează superputerea pe care player-ul o poate colecta și cu care poate ucide inamicii mai ușor. Instanța superputerii nu depinde de altă entitate, având o poziție fixă în cadrul unui nivel, setată prin metoda publică setNewPos().

În cadrul ambelor clase amintite mai sus, interacțiunea instanțelor lor cu player-ul este asemănatoare: pentru a putea fi colectat, coordonatele hitBox-ului trebuie să intre în conctact cu cele ale player-ului, iar după aceasta, când player-ul va ataca sau ajunge la ușa spre nivelul următor, itemele se vor folosi automat. Singura diferență semnificativă o reprezintă poziția fiecărui item, unul având o poziție statică, celălalt aflându-se la un inamic.

Diagrama UML a pachetului:



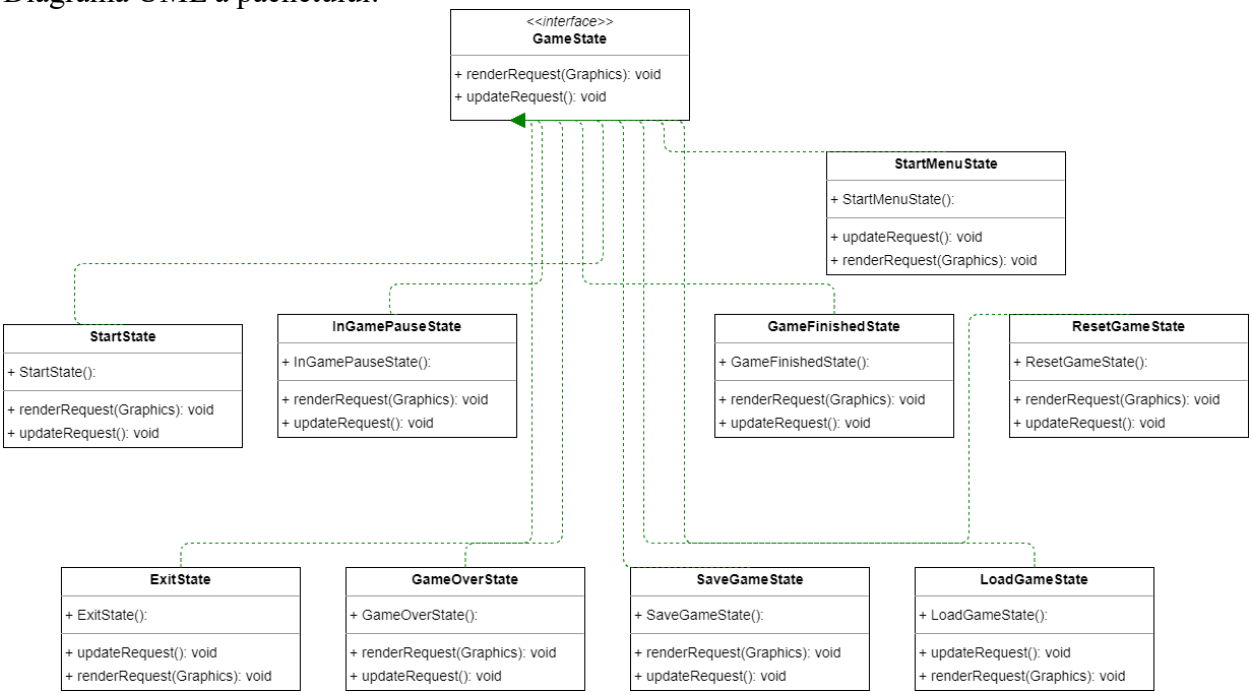
Pachetul **GameState**:

Interfața **GameState** definește două metode care vor fi implementate de fiecare stare a jocului în parte: renderRequest() – pentru afișaj și updateRequest() – pentru actualizarea entităților, obiectelor etc.

Clasele care au implementat GameState și rolul lor minimal (gestionarea lor se realizează în clasa MenuControl din pachetul UserInterface):

- **StartMenuState** – starea în care jocul se află în meniul de start;
- **StartState** – starea în care jocul rulează și se poate juca;
- **LoadGameState** – starea în care jocul începe inițializat din baza de date;
- **SaveGameState** – starea în care jocul este salvat în baza de date;
- **ResetGameState –** starea în care poziția player-ului se resetează; // ar trebui scoasă? Sau poate sa reseteze doar pozitia, nimic altceva?
- **ExitState –** starea în care jocul se închide;
- **GameOverState -** starea în care jocul este pierdut;
- **GameFinishedState –** starea în care jocul este câștigat.

Diagrama UML a pachetului:



Pachetul **UserInterface**:

Aici are loc modelarea interfeței cu utilizatorul în ceea ce privește meniul și butoanele din el, precum și gestionarea stărilor din joc.

Clasa abstractă ButtonInterface modelează butonul din meniu. Metodele importante și funcționalitatea lor:

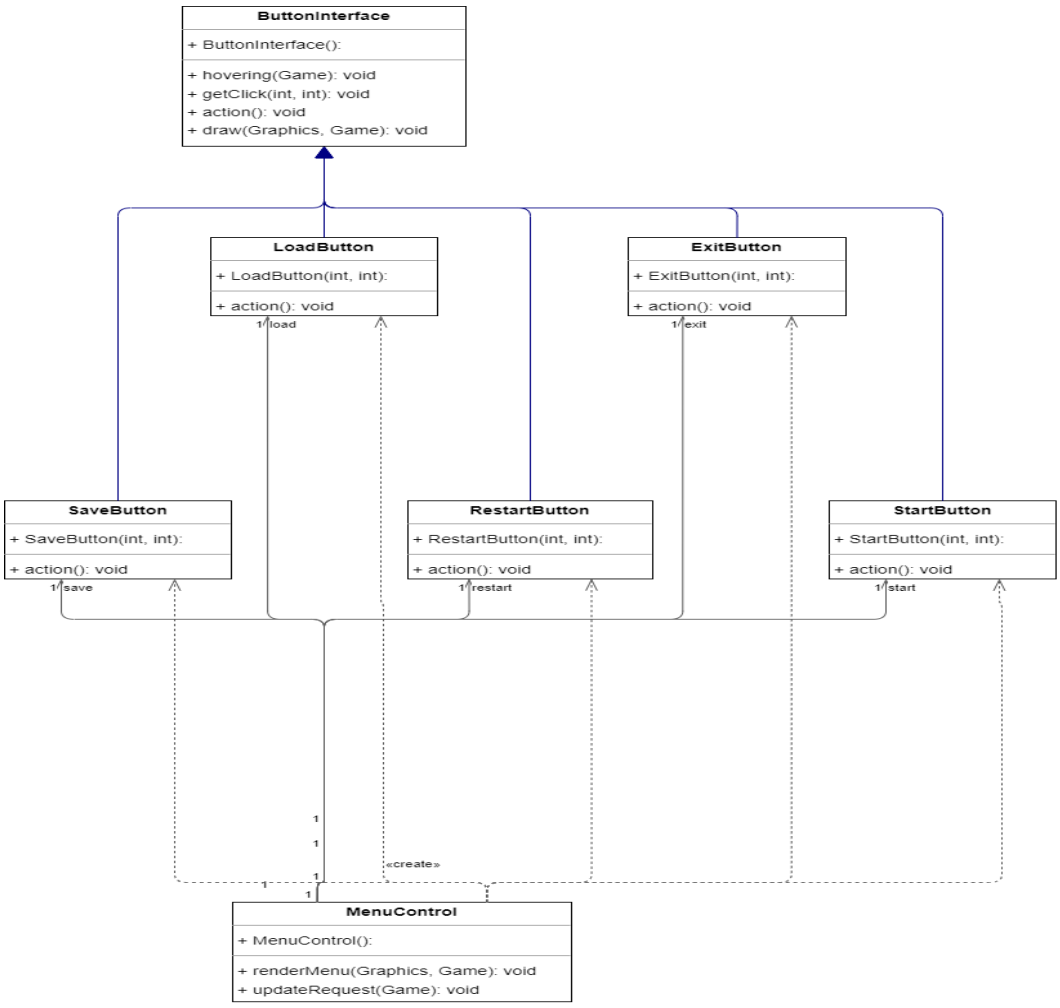
- draw() – desenează pe ecran butonul la o poziție specificată, actualizând imaginea în cazul în care pointer-ul mouse-ului este deasupra butonului – efectul de hovering;

- hovering() – metoda stabilește dacă pointer-ul mouse-ului este deasupra butonului sau nu. Se folosește SwingUtilities pentru a reduce aria de captare a poziției pointer-ului la fereastra jocului.
- getClick() – metoda este apelată în clasa de MouseInput, detectând evenimentul de apăsare a click-ului din stânga. In cazul în care coordonatele mouse-ului sunt in zona butonului, acesta devine activ prin variabila isPressed, gestionată ulterior în metoda action(), implementată de fiecare buton în parte.

Clasele butoanelor implementate și rolul lor:
- StartButton – pornește jocul;
- ExitButton – iese din joc;
- LoadButton – încarcă jocul din baza de date și îl pornește;
- SaveButton – salvează starea curentă a jocului;
- RestartButton – resetează poziția player-ului în cadrul nivelului.

Clasa **MenuControl** are rol de control al jocului la nivel abstract. Cele două metode, renderMenu() și updateRequest(), se ocupă cu gestionarea acțiunilor butoanelor, precum și cu stabilirea stării jocului.

Diagrama UML a pachetului:



Pachetul **utils**:

Acest pachet conține clase ce implementează diferite concepte și abilități ale jocului.

Clasa **Camera** modelează conceptul de cameră din joc. Acesta se bazează pe existența imaginară a unui dreptunghi ce ocupă aproximativ 80% din fereastra jocului, plasat strategic în centru. Atunci când player-ul ajunge la una din extremitățile dreptunghiului, camera se mișcă în direcția respectivă.

Clasa **Constants** stochează variabile ce au rol în gestionarea animațiilor, precum și metoda GetSpriteAmount(), care returnează lungimea unei animații în funcție de tip și entitate.

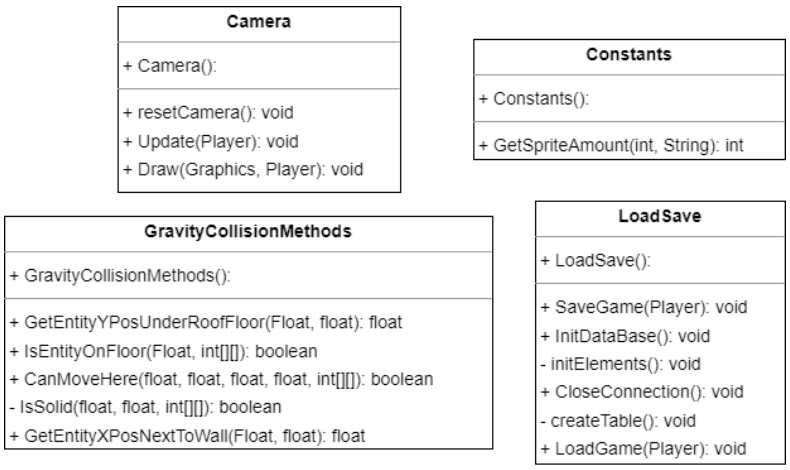
Clasa **GravityCollisionMethods** gestionează tot ce ține de conceptul de gravitație în joc. Metodele au următoarele funcționalități:

- IsSolid() –returnează o valoare boolean, testând dacă entitatea se poate mișca în locul unde se află, în concordanță cu matricea nivelului curent și valoarea de IsSoldi() a id-urilor tile-urilor din hartă;
- CanMoveHere() – apelând metoda IsSolid() pentru toate cele 4 colțuri ale hitBox-ului unei entități, stabilește dacă entitatea se poate mișca;
- GetEntityXPosNextToWall() – atunci când o entitate se află lângă un perete, metoda stabilește ca rama hitBox-ului să fie lipită de perete;
- GetEntityYPosUnderRoofFloor - metoda stabilește poziția entității pe podea sau sub tavan;
- IsEntityOnFloor – metoda returnează o valoare de adevăr ce semnalează dacă entitatea se află pe podea.

Clasa **LoadSave** implementează metodele de save și load legate de baza de date. Metodele sunt statice și au următoarele funcționalități:

- createTable() – creează tabelul bazei de date;
- initElements() – inițializează baza de date cu valorile implicite de început de joc: nivelul 1, scor curent și final 0, poziția inițială a jucătorului, viață completă, iteme colectate 0, inamici uciși 0, pozițiile inițiale ale camerei;
- InitDataBase() – verifică dacă fișierul bazei de date există deja, în caz controar apelează metodele createTable() și initElements(), conectând câmpul **Connection c** la baza de date;
- CloseConnection() – închide conexiunea la baza de date;
- SaveGame() – salvează starea jocului în baza de date;
- LoadGame() – încarcă starea jocului din baza de date.

Diagrama UML a pachetului:



Pachetul **Tiles**:

Pe lângă clasele care abstractizează diferite tipuri de dale: țepi, tepi în apă, blocuri solide, țevi solide, apă etc., în pachet se găsește și clasa LevelManager.

Clasa **LevelManager** se ocupă cu gestionarea nivelelor și entităților unui nivel. Metode principale:

- update() – metoda de update a obiectelor unui nivel: entități și iteme;
- render() – metoda de desenare a obiectelor din nivel, precum și a hărții;
- setLevel() – metoda setează harta și fundalul hărții fiecărui nivel;
- addEntities() – metoda adaugă entitățile în fiecare nivel, precum și superputerea;
- getEscapePos() – metoda caută dala care reprezintă ușa spre nivelul următor și salvează poziția în câmpurile gateXpos și gateYpos;
- initALevel() – metoda inițializează un nivel;
- passLevel() – metoda ce gestionează cazul în care player-ul a ajuns la poziția ușii spre nivelul următor și are cleștele colectat, trecând la nivelul următor.

Pachetul **GameWindow**:

Clasa KeyboardInput și MouseInput gestionează input-urile de la tastatură și mouse astfel:

- controlul player-ului este gestionat în metodele keyPressed() și keyReleased() din KeyboardInput. Când o tastă este apăsată, variabila booleană a comportamentului este pusă pe true (spre exemplu – tasta **A** setează pe true câmpul **left** din Player), iar când nu mai este apăsată, o setează pe false. De asemenea aici se gasesc câteva taste cu rol penrtu gestionarea meniului, precum și tasta **B** pentru debugging (vizualizarea hitBox-ului entităților).
- Controlul atacului player-ului și apăsarea butoanelor din meniu sunt gestionate în MouseInput, mai exact în mouseClicked() și mousePressed(). Comportamentul identic cu cel descris anterior la KeyboardInput, butonul care este luat în vedere fiind **BUTTON1**, adică **Click Stânga**.

Clasa **Game**:

În clasa Game au fost introduse câteva metode importante și funcționalități care au următoarele roluri:

- initInput() – inițializează input-urile, adăugând MouseListener și KeyListener la fereastra jocului;
- initClasses() – inițializează baza de date, texturile, nivelurile, grafica, entitățile și setează jocul pe StartMenuState;
- Update() și Draw() – apelează funcțiile de update() și render() din MenuControl și ale variabilei **state**;
- setState() – metodă pentru a seta starea jocului (apelată în clasa MenuControl).

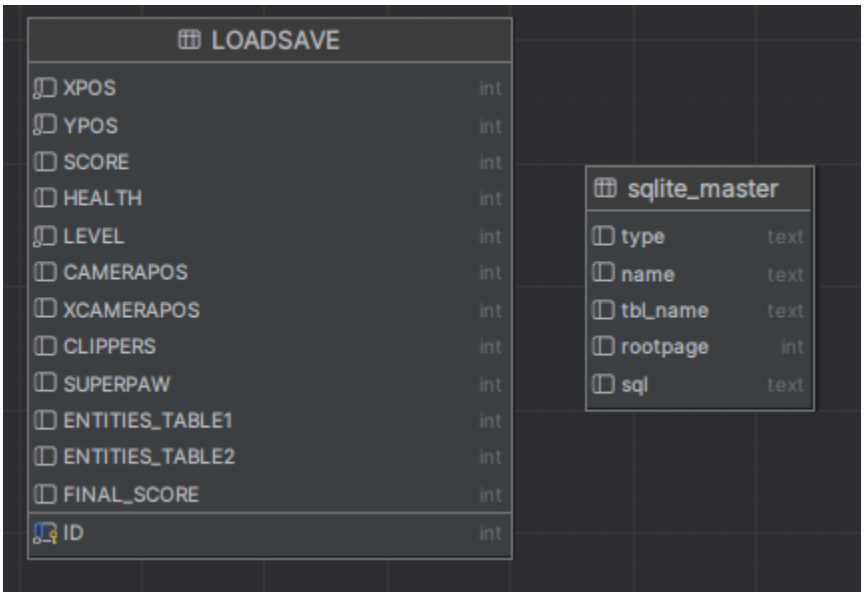
Metoda de victorie și de înfrângere

Jocul se câștigă prin parcurgerea celor 3 niveluri. Pentru a trece de la un nivel la celălalt, player-ul trebuie să găsească cleștele, care se află la unul din inamicii de pe hartă. În cadrul ultimului nivel, cleștele este la Boss.

Jocul este pierdut și resetat complet în cazul în care player-ul pierde toate cele 3 vieți pe care le are în timpul unui nivel.

În cazul în care jocul este câștigat, se va afișa un ecran cu scorul final. Ținând cont că scorul reprezintă durata de completare a jocului, un scor final cât mai mic este mai bun.

Structura bazei de date:



Descrierea tabelei:

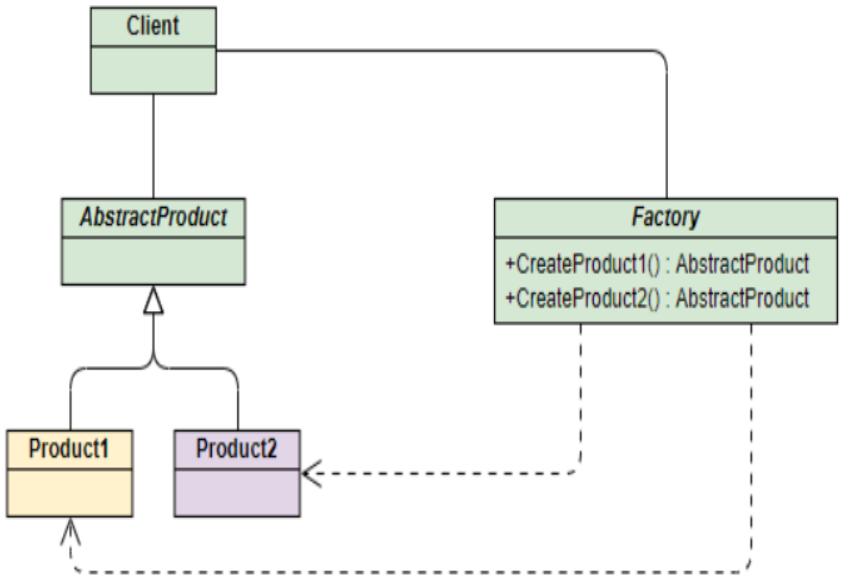
- XPOS – poziția X a player-ului;
- YPOS – poziția Y a player-ului;

- SCORE – scorul curent;
- HEALTH – viața jucătorului;
- LEVEL – nivelul;
- CAMERAPOS – poziția camerei în stânga;
- XCAMERAPOS – poziția camerei în dreapta;
- CLIPPERS – clește colectat sau nu;
- SUPERPAW – superputere colectată sau nu;
- ENTITIES_TABLE1 – primul inamic de pe hartă ucis sau nu;
- ENTITIES_TABLE2 – al doilea inamic de pe hartă ucis sau nu;
- FINAL_SCORE – scorul final;

Conceptele de SAVE și LOAD sunt implementate în clasa LoadSave din pachetul utils, descris mai sus. Salvarea în baza de date se poate face doar pe un slot.

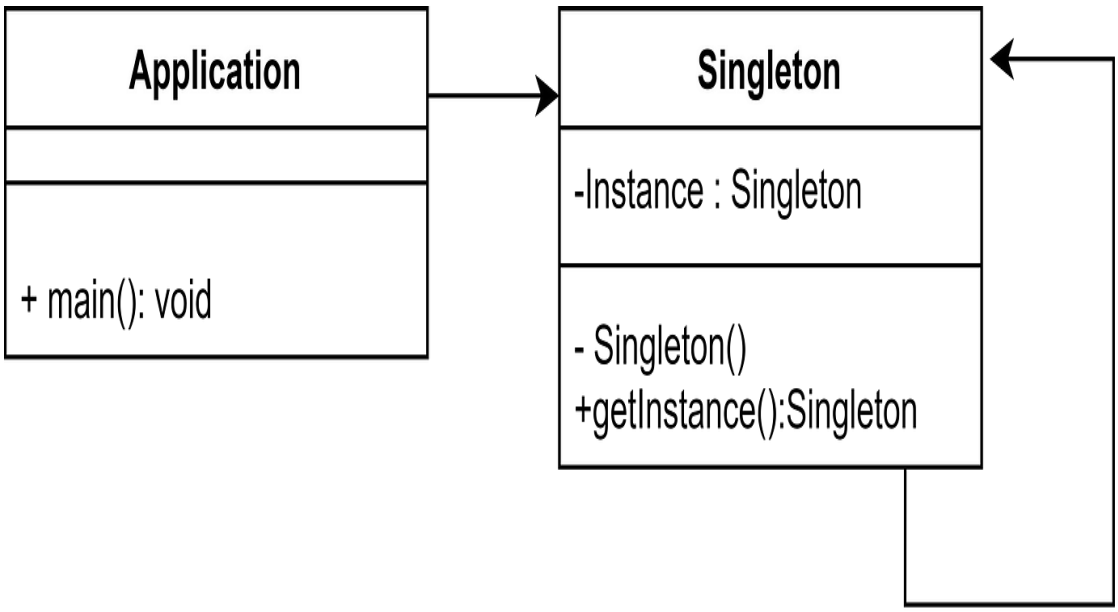
Șabloane de proiectare:

Factory Method:



Șablonul Factory a fost implementat în clasa EnemyFactory, prin metoda **factoryMethod()** ce are ca tip de return **Entity** și returnează inamici simpli sau inamici Boss.

Singleton:



Șablonul Singleton a fost implementat în cadrul clasei Player. Funcționalitatea șablonului constă în existența unei singure instanțe statice a clasei Player, accesibilă doar prin metoda publică getInstance().

Instanța privată se află în clasa Player, iar constructorul este unul privat. Astfel că player-ul se instanțiază odată cu folosirea funcției getInstance(), concept cunoscut ca lazy initialization.

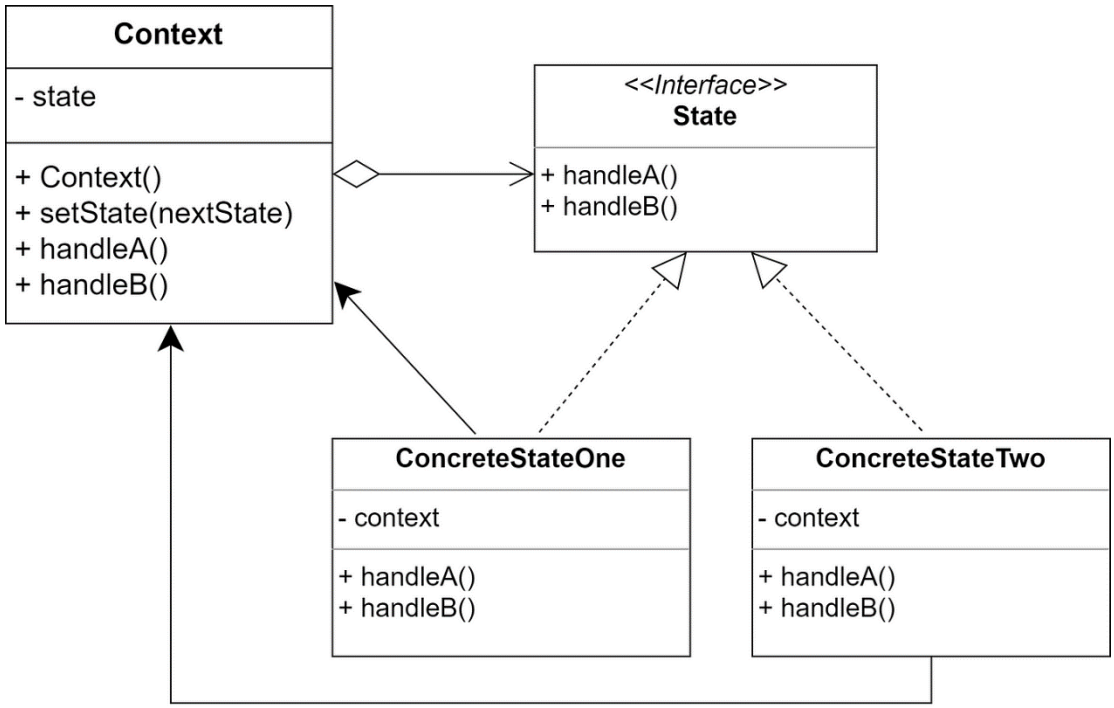
Codul ce implementează acest șablon:

```
private static Player instance;

public static Player getInstance()
{
   if(instance == null)
      instance = new Player(90,450,64,64);

   return instance;
}
```

State Pattern:



Șablonul State a fost implementat în pachetul GameState. Interfața unei stări este reprezentată prin interfața GameState ce are două metode de gestionat: renderRequest() și updateRequest().

Clasele concrete care implementează această interfață și rolul lor au fost descrise la descrierea detaliată a pachetelor.

Contextul este clasa **Game**, prin câmpul private **GameState state,** inițializat la pornire cu **new StartMenuState()**. Fiecare stare se ocupă de o anumita actualizare a jocului (updateRequest) și de un anumit afișaj(renderRequest).

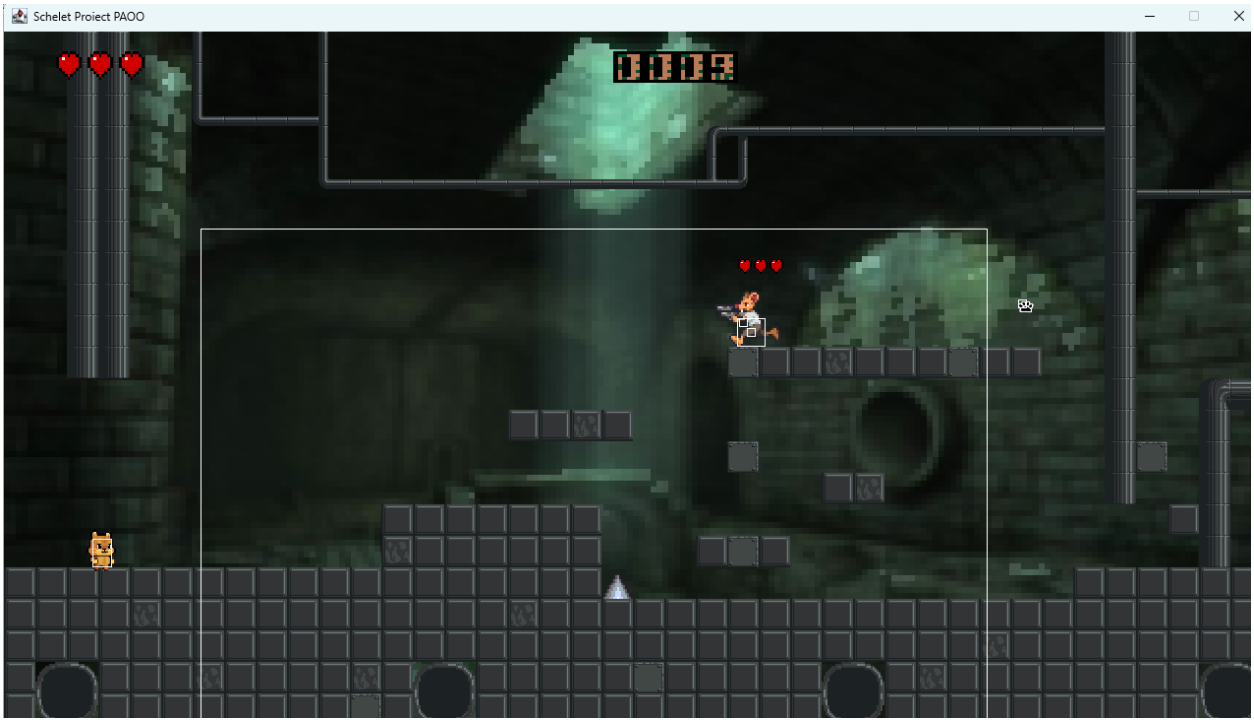
Pentru a controla trecerea printre stări, s-a folosit clasa MenuControl și câmpurile statice booleene din clasa Game: **START_PRESSED, EXIT_PRESSED, LOAD_SELECTED, SAVE_SELECTED, RESET_PRESSED, GAME_FINISHED, InGamePause_PRESSED.**

Implementarea coliziunilor:

Coliziunile au fost implementate printr-un mecanism simplu. Fiecare entitate, dar și itemele, conțin un câmp denumit **hitBox**, care este de tip **Rectangle2D.Float**.

Prin metodele din clasa GravityCollisionMethods, descrise mai sus, apelate mereu pe parcursul gameplay-ului, se testează coliziunea dintre entități și cea cu harta.

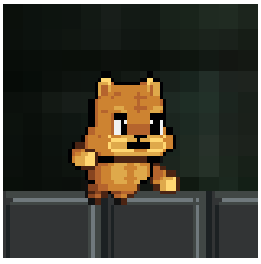
Tasta **B** permite vizualizarea dreptunghiului de coliziuni:



A se observa hitBox-ul:
- Dreptunghiul din jurul player-ului;
- Dreptuntghiul mai mare reprezintă aria camerei;
- Dreptunghiul din jurul inamicului;
- Pătratele mici din hitBox-ul inamicului reprezintă: cleștele (cel din mijloc) și glonțul (cel din stânga sus);
- Pătratul de la superputere;

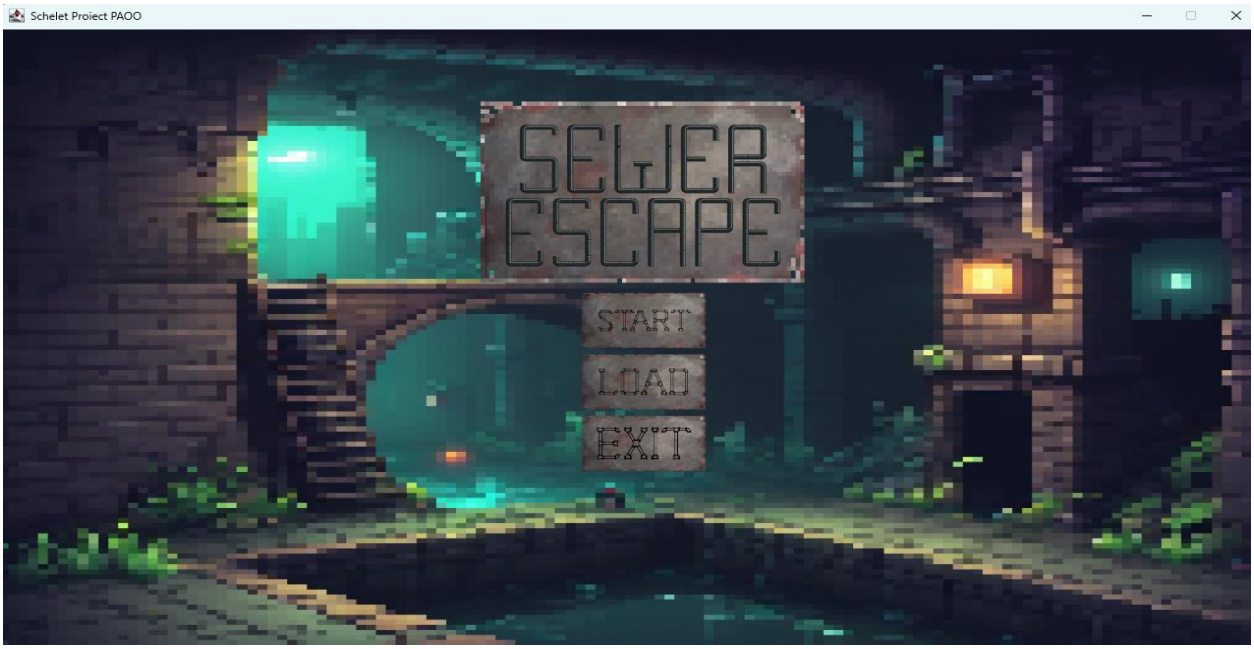
Funcționalități Player:

Playerul poate executa mai multe acțiuni: atac, săritură, mers târâș, înot.

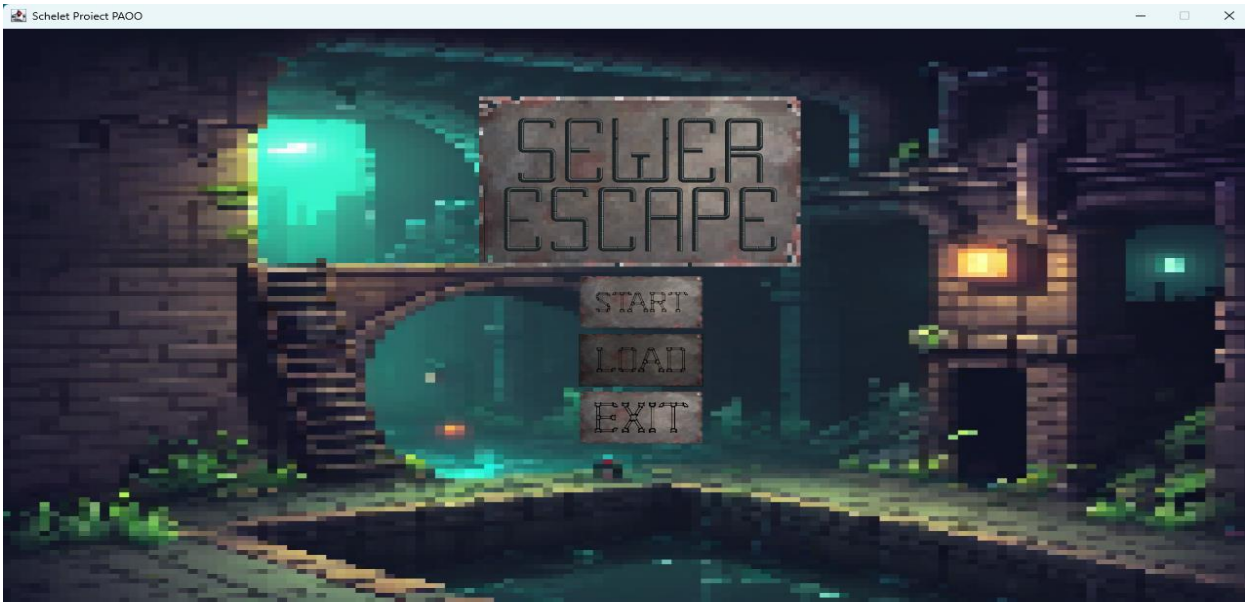



User Interface Design:

La deschidere, jocul se află în starea StartMenuState, care se afișează utilizatorului în felul următor:



Butoanele interacționează cu poziția cursorului pe ecran, umbrindu-se:

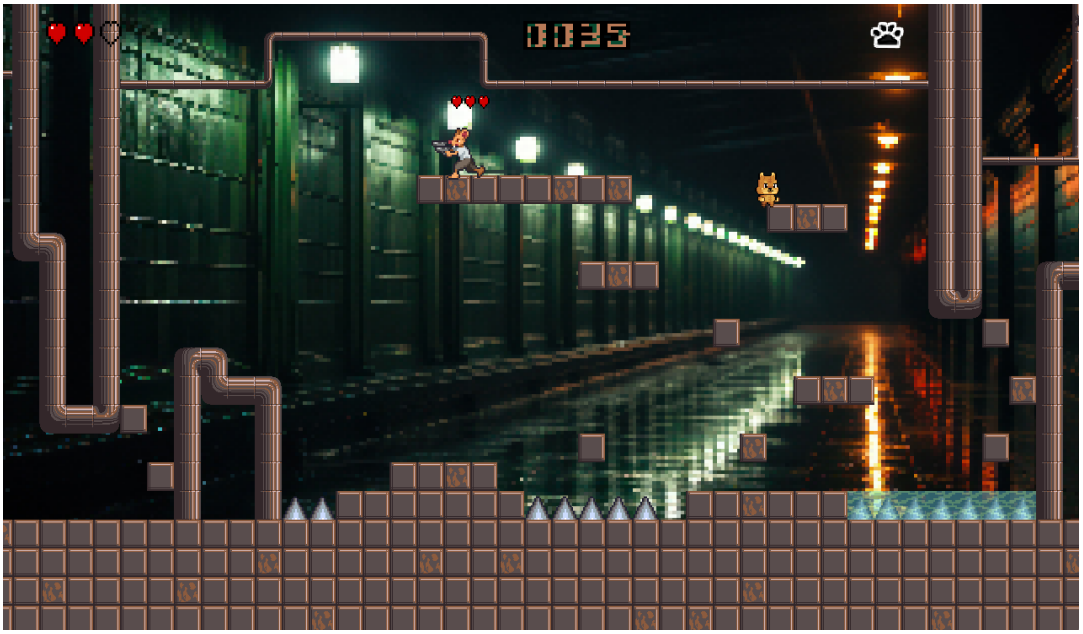


În timp ce jocul rulează, există un meniu de pauză, accesibil prin tasta **ESCAPE**, care arată în felul următor:

Screenshot-uri cu diferite ipostaze din joc:

- nivelul 1:

- nivelul 2:



- nivelul 3:

- joc terminat:

- joc pierdut

Alte funcționalități:

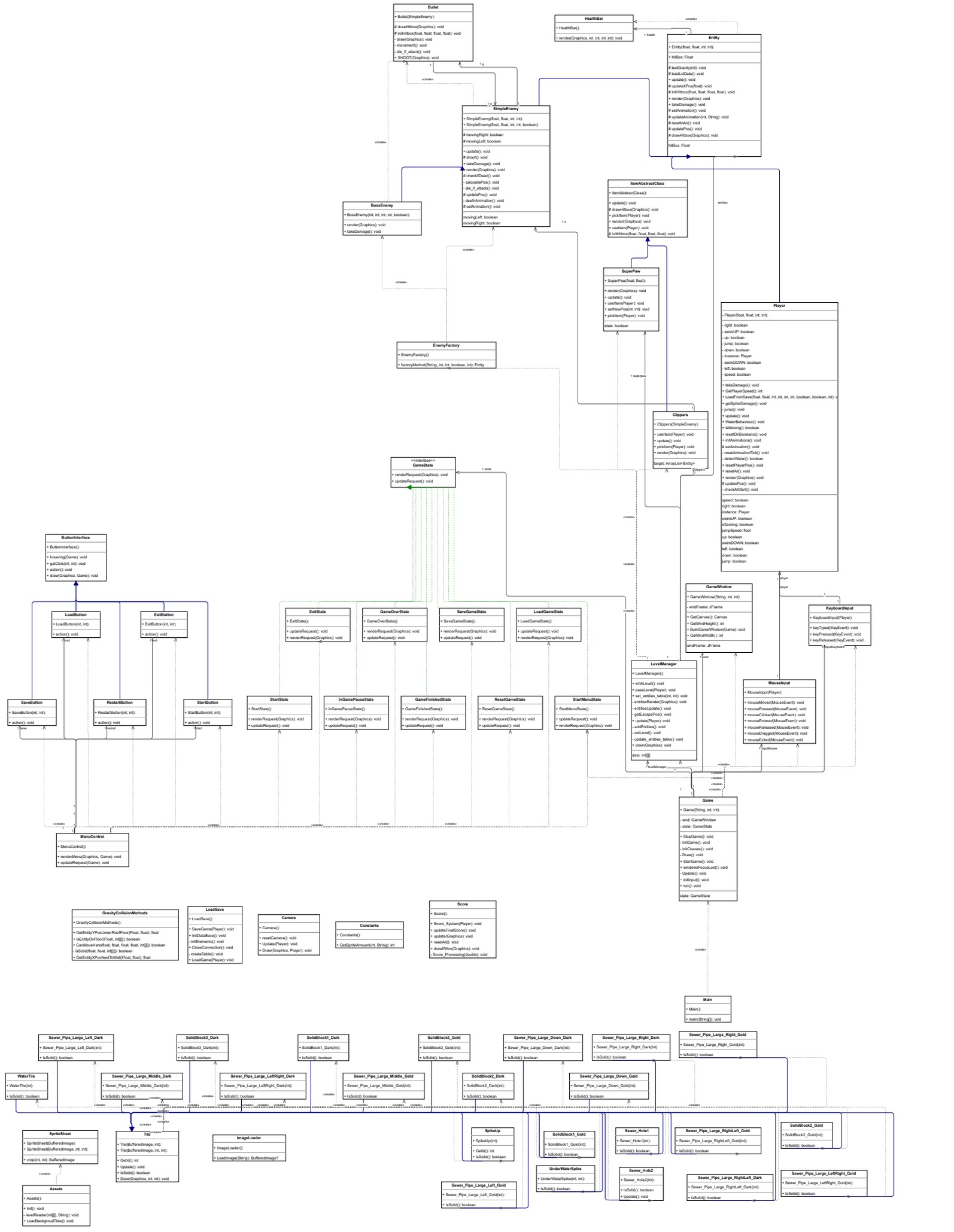
Funcțiile de meniu pot fi accesate cu următoarele butoane:

Tastă	Funcție
ESC	Oprire/Pornire pauză de joc
K	Save game
L	Load game
E	Exit game
R	Reset
ENTER	Start game

Bibliografie:

https://actg.itch.io/old-pipes-tileset
https://bowpixel.itch.io/cat-50-animations
https://sagak-art-pururu.itch.io/ratman
https://slashdashgamesstudio.itch.io/2d-platformer-sewer-assets
https://kingkelp.itch.io/sewer-tileset
https://gencraft.com/
https://ivoryred.itch.io/gardens-16x16-icon-pack?download
https://clipart-library.com/clipart/8TEjzoe7c.htm
https://swooshwhoosh.itch.io/heartsui
https://omniclause.itch.io/spikes?download
https://giventofly.github.io/pixelit/
https://medium.com/
Software utilizat: FireAlpaca, Paint, Microsoft Whiteboard.

Diagrama UML a întregului proiect:


Bullet
+ Bullet(SimpleEnemy):
# drawHitbox(Graphics): void
# initHitbox(float, float, float, float): void
- draw(Graphics): void
- movement(): void
- die_if_attack(): void
+ SHOOT(Graphics): void
HealthBar
+ HealthBar():
+ render(Graphics, int, int, int, int): void
Entity
+ Entity(float, float, int, int):
+ hitBox: Float
# testGravity(int): void
# loadLvlData(): void
+ update(): void
# updateXPos(float): void
# initHitbox(float, float, float, float): void
+ render(Graphics): void
+ takeDamage(): void
# setAnimation(): void
# updateAnimation(int, String): void
# resetInAir(): void
# updatePos(): void
# drawHitbox(Graphics): void
hitBox: Float
SimpleEnemy
+ SimpleEnemy(float, float, int, int):
+ SimpleEnemy(float, float, int, int, boolean):
# movingRight: boolean
# movingLeft: boolean
+ update(): void
# shoot(): void
+ takeDamage(): void
+ render(Graphics): void
# checkIfDead(): void
- calculatePos(): void
- die_if_attack(): void
updatePos(): void
- deathAnimation(): void
setAnimation(): void
movingLeft: boolean
movingRight: boolean
BossEnemy
+ BossEnemy(int, int, int, int, boolean):
+ render(Graphics): void
+ takeDamage(): void
ItemAbstractClass
+ ItemAbstractClass():
+ update(): void
# drawHitbox(Graphics): void
+ pickItem(Player): void
+ render(Graphics): void
+ useItem(Player): void
# initHitbox(float, float, float, float): void
SuperPaw
+ SuperPaw(float, float):
+ render(Graphics): void
+ update(): void
+ useItem(Player): void
+ setNewPos(int, int): void
+ pickItem(Player): void
state: boolean
Player
- Player(float, float, int, int):
- right: boolean
- swimUP: boolean
- up: boolean
- jump: boolean
- down: boolean
- instance: Player
- swimDOWN: boolean
- left: boolean
- speed: boolean
+ takeDamage(): void
+ GetPlayerSpeed(): int
+ LoadFromSave(float, float, int, int, int, int, boolean, boolean, int): v
+ getSpikeDamage(): void
- jump(): void
+ update(): void
+ WaterBehaviour(): void
+ IsMoving(): boolean
+ resetDirBooleans(): void
+ initAnimations(): void
# setAnimation(): void
- resetAnimationTick(): void
- detectIfWater(): boolean
+ resetPlayerPos(): void
+ resetAll(): void
+ render(Graphics): void
# updatePos(): void
- checkAtStart(): void
speed: boolean
right: boolean
instance: Player
swimUP: boolean
attacking: boolean
jumpSpeed: float
up: boolean
swimDOWN: boolean
left: boolean
down: boolean
jump: boolean
EnemyFactory
+ EnamyFactory():
+ factoryMethod(String, int, int, boolean, int): Entity
Clippers
+ Clippers(SimpleEnemy):
+ useItem(Player): void
+ update(): void
+ pickItem(Player): void
+ render(Graphics): void
target: ArrayList<Entity>
<<interface>>
GameState
+ renderRequest(Graphics): void
+ updateRequest(): void
ButtonInterface
+ ButtonInterface():
+ hovering(Game): void
+ getClick(int, int): void
+ action(): void
+ draw(Graphics, Game): void
LoadButton
+ LoadButton(int, int):
+ action(): void
ExitButton
+ ExitButton(int, int):
+ action(): void
SaveButton
+ SaveButton(int, int):
+ action(): void
RestartButton
+ RestartButton(int, int):
+ action(): void
StartButton
+ StartButton(int, int):
+ action(): void
ExitState
+ ExitState():
+ updateRequest(): void
+ renderRequest(Graphics): void
GameOverState
+ GameOverState():
+ renderRequest(Graphics): void
+ updateRequest(): void
SaveGameState
+ SaveGameState():
+ renderRequest(Graphics): void
+ updateRequest(): void
LoadGameState
+ LoadGameState():
+ updateRequest(): void
+ renderRequest(Graphics): void
StartState
+ StartState():
+ renderRequest(Graphics): void
+ updateRequest(): void
InGamePauseState
+ InGamePauseState():
+ renderRequest(Graphics): void
+ updateRequest(): void
GameFinishedState
+ GameFinishedState():
+ renderRequest(Graphics): void
+ updateRequest(): void
ResetGameState
+ ResetGameState():
+ renderRequest(Graphics): void
+ updateRequest(): void
StartMenuState
+ StartMenuState():
+ updateRequest(): void
+ renderRequest(Graphics): void
GameWindow
+ GameWindow(String, int, int):
- wndFrame: JFrame
+ GetCanvas(): Canvas
+ GetWindHeight(): int
+ BuildGameWindow(Game): void
+ GetWindWidth(): int
wndFrame: JFrame
KeyboardInput
+ KeyboardInput(Player):
+ keyTyped(KeyEvent): void
+ keyPressed(KeyEvent): void
+ keyReleased(KeyEvent): void
LevelManager
+ LevelManager():
+ initALevel(): void
- passLevel(Player): void
+ set_entities_table(int, int): void
- entitiesRender(Graphics): void
- entitiesUpdate(): void
- getEscapePos(): void
+ update(Player): void
- addEntities(): void
- setLevel(): void
- update_entities_table(): void
+ draw(Graphics): void
data: int[][]
MouseInput
+ MouseInput(Player):
+ mouseMoved(MouseEvent): void
+ mousePressed(MouseEvent): void
+ mouseClicked(MouseEvent): void
+ mouseEntered(MouseEvent): void
+ mouseReleased(MouseEvent): void
+ mouseDragged(MouseEvent): void
+ mouseExited(MouseEvent): void
Game
+ Game(String, int, int):
- wnd: GameWindow
- state: GameState
+ StopGame(): void
- InitGame(): void
- initClasses(): void
- Draw(): void
+ StartGame(): void
+ windowsFocusLost(): void
- Update(): void
~ initInput(): void
+ run(): void
state: GameState
MenuControl
+ MenuControl():
+ renderMenu(Graphics, Game): void
+ updateRequest(Game): void
GravityCollisionMethods
+ GravityCollisionMethods():
+ GetEntityYPosUnderRoofFloor(Float, float): float
+ IsEntityOnFloor(Float, int[][]): boolean
+ CanMoveHere(float, float, float, float, int[][]): boolean
- IsSolid(float, float, int[][]): boolean
+ GetEntityXPosNextToWall(Float, float): float
LoadSave
+ LoadSave():
+ SaveGame(Player): void
+ InitDataBase(): void
- initElements(): void
+ CloseConnection(): void
- createTable(): void
+ LoadGame(Player): void
Camera
+ Camera():
+ resetCamera(): void
+ Update(Player): void
+ Draw(Graphics, Player): void
Constants
+ Constants():
+ GetSpriteAmount(int, String): int
Score
+ Score():
+ Score_System(Player): void
+ updateFinalScore(): void
+ update(Graphics): void
+ resetAll(): void
+ drawIfWon(Graphics): void
- Score_Processing(double): void
Main
+ Main():
+ main(String[]): void
Sewer_Pipe_Large_Left_Dark
+ Sewer_Pipe_Large_Left_Dark(int):
+ IsSolid(): boolean
SolidBlock3_Dark
+ SolidBlock3_Dark(int):
+ IsSolid(): boolean
SolidBlock1_Dark
+ SolidBlock1_Dark(int):
+ IsSolid(): boolean
SolidBlock3_Gold
+ SolidBlock3_Gold(int):
+ IsSolid(): boolean
Sewer_Pipe_Large_Down_Dark
+ Sewer_Pipe_Large_Down_Dark(int):
+ IsSolid(): boolean
Sewer_Pipe_Large_Right_Dark
+ Sewer_Pipe_Large_Right_Dark(int):
+ IsSolid(): boolean
Sewer_Pipe_Large_Right_Gold
+ Sewer_Pipe_Large_Right_Gold(int):
+ IsSolid(): boolean
WaterTile
+ WaterTile(int):
+ IsSolid(): boolean
Sewer_Pipe_Large_Middle_Dark
+ Sewer_Pipe_Large_Middle_Dark(int):
+ IsSolid(): boolean
Sewer_Pipe_Large_LeftRight_Dark
+ Sewer_Pipe_Large_LeftRight_Dark(int):
+ IsSolid(): boolean
Sewer_Pipe_Large_Middle_Gold
+ Sewer_Pipe_Large_Middle_Gold(int):
+ IsSolid(): boolean
SolidBlock2_Dark
+ SolidBlock2_Dark(int):
+ IsSolid(): boolean
Sewer_Pipe_Large_Down_Gold
+ Sewer_Pipe_Large_Down_Gold(int):
+ IsSolid(): boolean
SpriteSheet
+ SpriteSheet(BufferedImage):
+ SpriteSheet(BufferedImage, int, int):
+ crop(int, int): BufferedImage
Assets
+ Assets():
+ Init(): void
- levelReader(int[][], String): void
+ LoadBackgroudTiles(): void
Tile
+ Tile(BufferedImage, int):
+ Tile(BufferedImage, int, int):
+ GetId(): int
+ Update(): void
+ IsSolid(): boolean
+ Draw(Graphics, int, int): void
ImageLoader
+ ImageLoader():
+ LoadImage(String): BufferedImage?
SpikeUp
+ SpikeUp(int):
+ GetId(): int
+ IsSolid(): boolean
SolidBlock1_Gold
+ SolidBlock1_Gold(int):
+ IsSolid(): boolean
Sewer_Hole1
+ Sewer_Hole1(int):
+ IsSolid(): boolean
Sewer_Pipe_Large_RightLeft_Gold
+ Sewer_Pipe_Large_RightLeft_Gold(int):
+ IsSolid(): boolean
SolidBlock2_Gold
+ SolidBlock2_Gold(int):
+ IsSolid(): boolean
UnderWaterSpike
+ UnderWaterSpike(int, int):
+ IsSolid(): boolean
Sewer_Hole2
+ Sewer_Hole2(int):
+ IsSolid(): boolean
+ Update(): void
Sewer_Pipe_Large_RightLeft_Dark
+ Sewer_Pipe_Large_RightLeft_Dark(int):
+ IsSolid(): boolean
Sewer_Pipe_Large_LeftRight_Gold
+ Sewer_Pipe_Large_LeftRight_Gold(int):
+ IsSolid(): boolean
Sewer_Pipe_Large_Left_Gold
+ Sewer_Pipe_Large_Left_Gold(int):
+ IsSolid(): boolean
«create»
1 health
1 b
1 e
1 state
1 superpaw
1 player
1 keyboard
1 mouse
1 levelManager
1 wnd