

**UNIVERSITATEA DIN BUCUREȘTI
FACULTATEA DE MATEMATICĂ ȘI INFORMATICĂ**

**LUCRARE DE LICENȚĂ
Joc 2D în C++ pentru platforma
PlayStation Portable**

**COORDONATOR ȘTIINȚIFIC
Conf. Dr. PĂUN Andrei**

**STUDENT
IFTODE Bogdan-Marius**

**BUCUREȘTI
Iunie 2013**

CUPRINS

1. Introducere	3
1.1. Motivație	3
1.2. Sumar	4
2. Consola <i>PlayStation Portable</i>	6
2.1. Istoric.....	6
2.2. Specificații tehnice	7
2.3. Aplicații homebrew	9
3. Tehnologii folosite	10
3.1. Limbajul C++	10
3.1.1 Programarea orientată pe obiecte	10
3.1.2 Biblioteca Standard Template Library (STL).....	13
3.2. Kitul de dezvoltare software <i>PSPSDK</i>	17
3.3. Librăria <i>OldSchool Library</i> pentru <i>PlayStation Portable</i>	19
3.4. Depanarea aplicației	20
3.4.1. Configurarea aplicațiilor <i>Eclipse</i> și <i>PSPLink</i>	20
3.4.2. Testarea performanței.....	22
3.4.3. Înțelegerea mesajelor de eroare.....	23
4. Arhitectura jocurilor și a jocului <i>N.A.S.S.</i>	24
4.1 Motorul jocului (<i>game engine</i>).....	24
4.2 Arhitectura pe trei nivele.....	31
4.2.1 Nivelul de comandă și control.....	32
4.2.2 Nivelul de logică	34
4.2.3 Nivelul interfețelor	37
4.3 Arhitectura multiplayer online	40
5. Prezentarea jocului <i>N.A.S.S.</i>	43
5.1 Meniul principal	43
5.2 Mecanica jocului	44
5.3 Abilități	45
5.4 Detectarea coliziunilor	47
6. Concluzii	49
Bibliografie	50
Anexa nr. 1: Listingul programului sursă	

1. INTRODUCERE

Lucrarea de față prezintă pe parcursul a șase capitole dezvoltarea unui joc 2D pentru consola *PlayStation Portable* și tehnologiile folosite în acest scop.

1.1. Motivație

Am ales să dezvolt un joc arcade 2D deoarece mecanica simplă ce stă la baza lor poate fi înțeleasă de către oricine, însă nu am vrut să fie doar o clonă a unui joc vechi, ci să aducă și un element de noutate. Spre deosebire de jocurile clasice cu nave spațiale și inamici, *N.A.S.S.* nu îți pune la dispoziție nicio abilitate ofensivă. Jucătorul este nevoit să se folosească de scuturi, de teleportări, de o scurtă perioadă de invulnerabilitate și de timpii săi de reacție pentru a aduna cât mai multe puncte. De asemenea, pe când jocurile clasice arcade puteau fi „câștigate” prin distrugerea unui inamic foarte puternic (și în general foarte mare) la final, în *N.A.S.S.* dificultatea jocului este sporită în timp până când devine imposibilă evitarea coliziunilor. Asta, împreună cu limitarea numărului de îmbunătățiri ce pot fi aduse abilităților forțează jucătorii să își cheltuiască punctele *XP* într-un mod strategic, astfel încât să obțină un punctaj cât mai mare cu putință.

Alegerea consolei *PlayStation Portable* ca și mediu de dezvoltarea a jocului a fost datorată faptului că este o consolă portabilă. Acest lucru, împreună cu durata scurtă a unui joc, fac posibilă jucarea oriunde și oricând. Un alt motiv este performanța acestei console. Se pot randa un număr foarte mare de obiecte pe ecran la o viteză constantă de 60 de cadre pe secundă și la o calitate vizuală superioară datorată ecranului lat ce poate afișa 16 milioane de culori.

Folosirea arhitecturii modulare prezentate în capitolul 4 mi-a permis ca oricând aveam o idee nouă pentru un element nou să îl pot adăuga cu ușurință fără a fi nevoit să fac modificări în codul deja existent. Programarea motorului de joc pe care s-a bazat ulterior jocul final, deși a mărit durata dezvoltării, m-a ajutat să evit rescrierea de cod și va permite în viitor portarea aplicației pe alte platforme doar prin modificarea codului motorului.

Jocurile video, fie ele pe console, tablete, telefoane mobile sau calculator, au evoluat pe parcursul a câtorva decenii, din stadiul de a fi considerate simple activități de ocupare a timpului liber, într-o industrie de miliarde de dolari care rivalizează *Hollywood*-ul în termeni de dimensiune și popularitate.

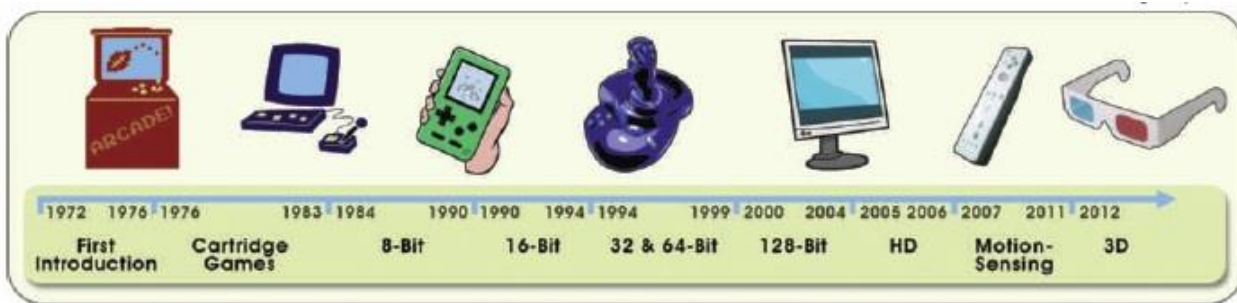


Fig. 1.1. Evoluția tehnologiei folosite pentru jocuri [13]

Deși tehnologia s-a îmbunătățit considerabil (vezi figura 1.1), permițând realizarea unor aplicații multimedia 3D foarte complexe și detaliate, ce rulează în timp real pe dispozitive accesibile publicului larg, jocurile 2D clasice și-au păstrat farmecul, atât pentru cei nostalgici după vremurile *Atari* și *Nintendo Entertainment System*, cât și pentru cei ce doresc să se recreeze în timpul lor liber.

1.2. Sumar

Următoarele capitole cuprind descrierea consolei *Playstation Portable* și a tehnologiilor folosite în vederea realizării jocului, precum și prezentarea aplicației propriu-zise.

Capitolul 2, *Consola PlayStation Portable (PSP)*, prezintă istoricul și componentele hardware ale consolei precum și detalii referitoare la dezvoltarea de aplicații *homebrew*.

Capitolul 3, *Tehnologii folosite*, începe cu o descriere a limbajului de programare C++ împreună cu avantajele pe care le oferă. Sunt prezentate resursele software necesare compilării de cod în vederea executării sale pe consolă și librăria *OldSchool Library*, care cuprinde funcții utile pentru desenarea pe ecran și redarea de sunete, printre altele. La sfârșitul capitolului se discută despre metodele de depanare pe care le asigură *Eclipse* împreună cu programul *PSPLink* fără de care un astfel de proiect ar fi incredibil de dificil, dacă nu chiar imposibil.

Capitolul 4, *Arhitectura unui joc*, prezintă avantajele unui motor și conceptele arhitecturii pe trei nivele. Realizând un motor peste care se construiește restul aplicației, oferă atât o metodă de a evita rescrierea inutilă de cod, cât și portabilitatea programului. Astfel, logica și interfața cu utilizatorul sunt păstrate în stare intactă, iar modificând doar codul pe care îl conține motorul, putem porta jocul pe o altă platformă.

Capitolul 5, *Jocul N.A.S.S*, prezintă aplicația din punctul de vedere al utilizatorului și abilitățile pe care acesta le are la dispoziție pentru a supraviețui.

Capitolul 6, *Concluzii*, expune direcția în care aplicația poate fi extinsă în viitor și concluziile la care s-a ajuns pe parcursul dezvoltării acesteia.

2. CONSOLA PLAYSTATION PORTABLE

Dimensiunea redusă a consolelor portabile permite utilizatorului să le ia oriunde. Consola *Game Boy* de la *Nintendo* a avut ca țintă publicul tânăr, însă începând cu *Game Boy Advance SP* (*GBA SP*) era popular și pentru publicul cu vârsta de peste 18 ani. Consola *GBA SP* este cu mult îmbunătățit față de modelele precedente, cu un ecran cu iluminare în spate și avea baterie reîncărcabilă cu autonomie mare. Folosind aplicația *Majesco's GBA video player* utilizatorii consolei *GBA SP* puteau să vizioneze filme. Consola *Nintendo DS* (o consolă portabilă cu două ecrane) inovează experiența jucătorilor prin folosirea simultană a două perspective diferite.

În 2011, *Nintendo* lansează *3DS*, o consolă portabilă autostereoscopică, care poate proiecta imagini 3D stereoscopice fără a fi necesară folosirea unor ochelari speciali. Stereoscopia este tehnica de creare a imaginilor 3D prin prezentarea a două imagini 2D corespunzătoare ochiului stâng, respectiv drept. Creierul procesează aceste două imagini, creând iluzia de adâncime.

Consola *PlayStation Vita* de la *Sony*, lansată tot în anul 2011, este succesorul consolei *PlayStation Portable*, având două joystick-uri analogice, un ecran OLED multi-touch de 5 țoli, Bluetooth, Wi-Fi și 3G.

2.1. Istoric

PlayStation Portable (figura 2.1) este o consolă de jocuri portabilă produsă de către *Sony*. La expoziția *E3* din 2003, *Sony Computer Entertainment* au anunțat că au o consolă portabilă în stadiul de proiect, urmând să fie dezvăluită pe 11 mai 2004. Dispozitivul a fost lansat inițial în Japonia pe data de 12 decembrie 2004, în America de Nord pe 24 martie 2005 și pe 1 septembrie 2005 în restul regiunilor globului.

Este singura consolă de jocuri portabilă care folosește discuri optice *Universal Media Disc (UMD)*, un format proprietar *Sony*. Consola se mai diferențiază prin dimensiunea mare a ecranului, a capacităților multimedia și a posibilităților de conectare cu *PlayStation 2*, *PlayStation 3*, alte *PlayStation Portable* și la Internet.

În septembrie 2007, *Sony* a lansat o versiune mai subțire și mai ușoară a consolei, pe care a denumit-o *PlayStation Portable Slim & Lite*. După doi ani, acest model a fost înlocuit de *PlayStation Portable Brite*, o versiune cu ecranul îmbunătățit și cu microfon. În paralel cu modelul *Brite*, se vindea și un model complet refăcut, numit *PSP Go* (vezi figura 2.3), având

dimensiuni reduse, ecran rabatabil, memorie internă, însă fără dispozitiv optic de citire a discurilor *UMD*.

În 2011 s-a lansat o versiune de buget, modelul *Street* sau *PSP-E1000*, căreia îi lipseau componentele *Wi-Fi* și dispozitivul optic. În a opta generație de console de jocuri, *PlayStation Portable* a fost înlocuit de consola *PlayStation Vita*, lansată în decembrie 2011 în Japonia și în februarie 2012 în America de Nord, Europa și Australia [1].



Fig. 2.1. Consola *PlayStation Portable*

Modelul original *PSP-1000* [1]

2.2. Specificații tehnice

Următoarele specificații tehnice se aplică pentru prima generație de console *PlayStation Portable*, sau *PSP-1000*, restul având mici diferențe. Cea mai importantă diferență între prima generație și următoarele este dimensiunea memoriei RAM, care este 32 MB la prima și 64 MB la următoarele [1].

- Dimensiuni:
 - Lățime: 170mm
 - Înălțime: 74mm
 - Adâncime: 23mm
- Ecran TFT LCD:
 - 4.3 țoli
 - 480 x 272 (aspect 16:9)
 - 16,77 milioane de culori

- Comunicarea datelor:
 - Wireless LAN 802.11b
 - IrDA
 - Mini-USB 2.0
- Stocare date:
 - Memory Stick Pro Duo
 - Memorie internă NAND flash de 32 MB pentru sistemul de operare (vezi figura 2.2)
 - Universal Media Disc (UMD) Dual-Layer de până la 1.8GB
- Audio:
 - Difuzoare stereo
 - Suportă redarea formatelor ATRAC3 plus, AAC, WMA, MP3 și WMA
- CPU (vezi figura 2.2)
 - Bazat pe arhitectura MIPS R4000 pe 32 de biți
 - 1-333 MHz
 - 16 KB cache pentru instrucțiuni și 16 KB cache pentru date
- GPU
 - 1-166 MHz
 - Rată de umplere cu pixeli: 600 Megapixeli / s
 - 33 milioane de poligoane / s
 - Culori RGBA pe 24 de biți
- Memorie:
 - 32 MB DDR SDRAM (8 MB rezervați pentru Kernel)
 - 2 MB eDRAM (VRAM)



Fig. 2.2. CPU, PSP Media Engine și NAND flash [1]



Fig. 2.3. Consola *PSP Go* [1]

2.3. Aplicații homebrew

Termenul de *homebrew* se referă la procesul de a folosi exploatari ale vulnerabilităților sistemului de operare ale consolei *PlayStation Portable* în scopul executării de cod fără semnătură digitală (vezi figura 2.4).

La scurt timp de la apariția consolei, hackerii au început să descopere astfel de vulnerabilități, urmând ca Sony să lanseze versiunea 1.51 a sistemului de operare al consolei în mai 2005 care rezolva problemele de securitate pe care le găsiseră hackerii. Până în 2006, singura metodă de a rula cod nesemnat era să păstrezi versiunea 1.50 a sistemului de operare, însă pe la jumătatea aceluși an a fost postat pe Internet un fișier care permitea utilizatorilor de PlayStation Portable cu versiunea de sistem 2.6 să revină la versiunea 1.5.

Probabil cel mai cunoscut individ din scena *homebrew* este *Dark_AleX*, un programator de origine spaniolă care a creat și a făcut public un sistem de operare personalizat, intitulat „*Open Edition firmware*” sau „*Custom Firmware (CFW)*”. Acest sistem de operare permitea utilizatorilor să ruleze cod nesemnat și în același timp să aibă acces la toate funcțiile celei mai noi versiuni oficiale a sistemului software.



Fig. 2.4. Cod fără semnătură digitală
rulând pe consola PlayStation Portable [1]

3. TEHNOLOGII FOLOSITE

3.1. Limbajul C++

C++ este un limbaj de programare dezvoltat de dr. Bjarne Stroustrup în laboratoarele *AT&T Bell* care are la bază limbajul de programare C la care a adăugat orientarea pe obiecte și alte facilități. Se poate considera limbajul C++ ca fiind un superset a lui C, deoarece C++ acceptă caracteristicile limbajului de programare C. C++ este mai mult decât „un C orientat pe obiecte” – C++ adaugă, de fapt, multe noi caracteristici care sporesc capacitățile programelor [2].

3.1.1 Programarea orientată pe obiecte

Programarea orientată pe obiecte este un model de programare foarte puternic și important. Programele orientate pe obiecte, atunci când sunt proiectate și implementate corespunzător, oferă mai multă flexibilitate pentru a face față cerințelor livrării de produse software pe o piață aflată în continuă schimbare. Toate aplicațiile, de la procesoare de texte la programe de calcul tabelar, aplicații grafice și sisteme de operare, sunt scrise în limbaje orientate pe obiecte, iar marea lor majoritate sunt scrise în C++ [3].

Pentru programatori, un obiect este o colecție de date și un set de operații, numite metode, care instrumentează datele. Programarea orientată pe obiecte este calea prin care programele sunt gândite în termeni de obiecte (lucruri) care alcătuiesc un sistem. Programarea pe obiecte are mai multe avantaje:

- Ușurința proiectării și a reutilizării codului – După ce codul operează corespunzător, utilizarea obiectelor mărește posibilitatea de reutilizare a proiectului sau codului creat pentru o aplicație, într-o altă aplicație.
- Fiabilitate crescută – O dată corect testate bibliotecile de obiecte, utilizarea codului existent va mări fiabilitatea programelor.
- Ușurința înțelegerii – Utilizarea obiectelor îi ajută pe programatori să înțeleagă și să se concentreze asupra elementelor cheie ale sistemului. Utilizarea obiectelor permite proiectanților și programatorilor să se concentreze asupra fragmentelor cele mai mici ale sistemului. Astfel se creează un cadru în care proiectanții se pot concentra mai

mult asupra operațiilor executate de program asupra acestor obiecte, asupra informației stocate în aceste obiecte și asupra altor componente cheie ale sistemului.

- Creșterea abstractizării – Abstractizarea permite proiectanților și programatorilor „o privire asupra matricei în ansamblu”, ignorând temporar detaliile elementare, pentru a opera cu elementele de sistem pe care le înțeleg mai ușor.
- Creșterea capacității de încapsulare – Încapsularea grupează toate părțile unui obiect într-un pachet unic și bine organizat.
- Ascunderea crescândă a informației – ascunderea informației este capacitatea unui program de a trata o funcție, o procedură sau un obiect drept o „cutie neagră”, utilizând elementul pentru a efectua o operație specifică, fără cunoașterea a ceea ce se întâmplă în interior [2].

Cele trei concepte care formează modelul orientat spre obiecte sunt: moștenirea, încapsularea și polimorfismul. Moștenirea implică o relație părinte/copil (descendent), exact așa cum se întâmplă în natură. Încapsularea include interfața și abstractizarea unei clase. Despre o clasă încapsulată se spune că este coezivă și de sine stătătoare. Încapsularea asigură o separare clară între interfața și implementarea unei clase. Polimorfismul înseamnă, literal, multe forme.

Un obiect prezintă un comportament polimorfic în funcție de poziția în care se află în ierarhia de moștenire. Dacă două sau mai multe obiecte au aceeași interfață, dar au componente diferite, se spune că sunt polimorfe. Polimorfismul este o caracteristică foarte puternică a limbajelor de programare orientate pe obiecte. Acesta permite ca o funcție membru să aibă comportamente diferite în funcție de tipul obiectului [3].

Moștenirea

Moștenirea este cunoscută ca o relație de tip *este un*. Un golden retriever *este un* câine, un șarpe *este o* reptilă și o floare *este o* plantă. Fiecare dintre acestea este o specializare a părintelui. De exemplu, deși atât șarpele cât și șopârla sunt reptile, între ele există mari diferențe.

Clasele dintr-un program furnizează utilizatorilor funcționalitatea de bază, prestabilită. Utilizatorii, în acest context, sunt legați de alte obiecte din aplicație care creează și folosesc instanțe ale claselor definite în program. O instanță (formă de manifestare) a unei clase este numită și obiect. Schița unei clădiri este analogă unei clase. Clădirea este obiectul. Schița este doar reprezentarea statică a clădirii. Clădirea este obiectul dinamic, real.

Interfața unui obiect este partea pe care o văd clienții. O interfață este considerată un contract între clasă și clienții acesteia. De exemplu, un automobil are o interfață (un contract) cu șoferul. Interfața unui automobil include un volan, pedale de accelerație și frână, vitezometru, eventual o pedală de ambreiaj și un schimbător de viteze. Interfața asigură șoferului funcționalitatea de care are nevoie fără ca acesta să știe nimic despre modul intern de funcționare a automobilului – doar bagă cheia în contact, o răsuțește și conduce. Singurele funcții de care se folosește șoferul sunt manevrarea volanului, frânarea și accelerarea. Pentru el nu contează dacă mașina are motorul în față sau în spate ori dacă motorul are patru sau opt cilindri [3].

Încapsularea

Încapsularea specifică includerea funcționalității (serviciilor) și a părților într-un articol. Conceptul de încapsulare precizează că ar trebui ascunse părțile unui articol. Numai articolul trebuie să-și cunoască propriile părți interne.

Un alt aspect al încapsulării este asumarea responsabilității pentru părțile și serviciile unui articol. Fiecare articol are un contract prin care își specifică responsabilitățile față de clienți. Un articol este considerat coeziv dacă este complet încapsulat.

Este necesară protejarea stării interne a obiectelor, altfel obiectele respective vor deveni instabile sau corupte. Un obiect instabil este un obiect în cărui stare nu se poate avea încredere [3].

Polimorfismul

Comportamentul polimorfic se bazează pe implementarea corespunzătoare a moștenirii și încapsulării [3]. Polimorfismul permite programelor să aplice aceeași operație la obiecte de tipuri diferite. Deoarece polimorfismul permite programatorilor aplicarea aceleiași operații pentru mai multe tipuri, polimorfismul permite utilizarea aceleiași interfețe de acces la obiecte diferite. În C++, accesul la polimorfism este furnizat de funcțiile virtuale. În cel mai simplu înțeles, o funcție virtuală este un pointer către o funcție pe care compilatorul o rezolvă în cursul execuției. În raport de funcția către care indică funcția virtuală, operația efectuată de program va prezenta diferențe. În consecință, o singură interfață (funcția virtuală) poate furniza accesul la diferite operații [2].

3.1.2 Biblioteca Standard Template Library (STL)

STL este o bibliotecă generică ce furnizează soluții pentru manipularea colecțiilor de date, precum și algoritmi eficienți care acționează asupra acestor date. Toate componentele *STL* sunt șabloane. Când vorbim de șabloane, ne referim la șabloane de funcții sau la șabloane de clase. Funcțiile și clasele șablon sau generice au proprietatea remarcabilă că acceptă tipuri de date generice sau, într-o altă formulare, acceptă tipuri de date parametrizate. Tipurile generice permit folosirea aceleiași clase sau funcții cu tipuri de date diferite, fără a mai fi nevoie de a rescrie codul de fiecare dată când se trece la alt tip de date.

STL se bazează pe trei componente fundamentale: containere, algoritmi și iteratori. Containerele sunt obiecte care găzduiesc alte obiecte.

Containerele *STL* sunt omogene, adică pot păstra numai valori de același tip. Există în *STL* câteva tipuri predefinite de clase container, numite pe scurt containere. Spre exemplu, clasa șablon *vector* definește un vector dinamic, capabil să rețină un set de valori de același tip, care pot fi accesate în mod aleatoriu. Clasa *list* definește o listă liniară dublu înlănțuită, *deque* creează o listă cu două capete (*double ended queue*). Aceste clase container sunt grupate în conceptul de containere secvențiale, întrucât, în terminologia *STL*, o secvență este văzută ca o succesiune liniară de elemente.

STL definește și o a doua categorie de containere, așa numitele containere asociative. Un asemenea tip de container asociază o valoare cu o cheie, iar apoi folosește acea cheie pentru a regăsi în mod eficient valoarea. Containerele asociative predefinite ale *STL* sunt: set, multiset, map și multimap. Calitatea principală a containerelor asociative este rapiditatea accesului la elementele conținute.

Fiecare clasă container furnizează un set de funcții care operează asupra elementelor sale, realizând diverse operații cum ar fi inserarea, ștergerea, căutarea.

Iteratorii sunt o generalizare a conceptului de pointer, specifică *STL*. Sunt folosiți în scopul parcurgerii elementelor unei colecții, într-un fel similar în care pointerii o fac la parcurgerea unui tablou. Iteratorii sunt de fapt obiecte care acționează asemănător pointerilor. Este remarcabil faptul că iteratorii, deși sunt definiți în mod separat pentru fiecare container, oferă o interfață uniformă pentru orice tip de container. Spre exemplu, o operație de incrementare `it++`, comună atât pointerilor, cât și iteratorilor, prin care se avansează la următorul element din colecție, se scrie în același fel și are același efect, indiferent de structura de date internă a containerului: șir, listă liniară înlănțuită sau arbore.

Algoritmii acționează asupra containerelor. Cu ajutorul acestora puteți manipula conținutul containerelor. Algoritmii permit căutarea, copierea, modificarea, respectiv sortarea elementelor containerelor. Algoritmii folosesc iteratori. Grație acestui fapt, algoritmii devin independenți de tipul containerului utilizat. O operație, cum ar fi căutarea unui element în mai multe colecții, presupune existența unei funcții care se va apela în același mod pentru tablouri, pentru liste sau pentru oricare alt tip de container. *STL* oferă asemenea funcții șablon, numite algoritmi, independente atât de tipul datelor depozitate în containere, cât și de tipul structurilor interne de date specifice fiecărui container [4].

În vederea realizării aplicației m-au ajutat componente precum clasele *list* și *string* cât și iteratorii, componente pe care le voi descrie în continuare.

Clasa template list

După cum sugerează numele, *list* este un container secvențial care implementează o listă dublu înălțuită cu următoarele proprietăți:

- Suportă iteratori bidirecționali, prin urmare poate fi parcursă în ambele direcții. Așadar, *list* este un container reversibil.
- Permite inserări și ștergeri în timp constant, oriunde în interiorul secvenței.
- Spre deosebire de clasa *vector* și de cozile *deque*, *list* nu suportă accesul rapid la elementele listei, fapt care nu este întotdeauna un neajuns, întrucât unii algoritmi nu necesită decât accesul secvențial.

Ca o consecință a faptului că *list* nu permite accesul aleatoriu la elemente, clasa nu oferă nici operatorul `[]` pentru accesul indexat. Pentru clasa *list* este destul de evident că atât specificațiile, cât și numele, sugerează o implementare bazată pe o listă dublu înălțuită, cu alocare dinamică (vezi figura 3.1).



Fig. 3.1. Schema unei liste dublu înălănțuite

Parcurgerea unei liste se face cu ajutorul iteratorilor. Aceștia trebuie să fie iteratori bidirecționali, pentru a permite deplasarea în ambele direcții. Pentru *list*, iteratorii sunt generalizări ale pointerilor la noduri [4].

Clasa *string*

C++ suportă șiruri de caractere în două forme. Prima ca șiruri de caractere terminate cu caracterul nul „\0”, numite uneori stringuri C sau șiruri standard C. A doua formă o reprezintă obiectele de tip *string*.

Biblioteca C++ *Standard* definește tipul *string* în fișierul antet *string*. Clasa funcționează ca un container secvențial. Obiectele de tip *string* încapsulează șiruri de caracter,

Clasa *string* oferă operații pentru manevrarea stringurilor. Stringurile pot fi accesate indexat, atribuite, copiate sau comparate. Din punct de vedere tehnic, tipul *string* satisface cerințele unui container secvențial, dar nu face parte din *STL* deoarece nu este clasă șablon.

Clasa *string* este un container secvențial reversibil ce se aseamănă în multe privințe cu clasa *vector*:

- Suportă iteratori cu acces aleatoriu.
- Implementează `push_back()` ca operație de concatenare a unui caracter.
- Admite operatorul de indexare `[]` și funcția `at()`, pentru accesarea aleatorie a elementelor secvenței.
- Lungimea unui *string*, ca și memoria ocupată se pot modifica dinamic.
- Implementează funcțiile `reserve()` și `capacity()`, ca și *vector*

Există și deosebiri: *string* nu este clasă șablon. Containerizează numai șiruri de caractere. Nu implementează `pop_back()`, dar implementează operatorii de inserție `<<` și de extracție `>>`. Deoarece tipul *string* este o specializare, nu veți găsi pentru aceasta o definiție separată de clasă. Există o singură clasă, `basic_string<char>`. Un alt aspect important este că `basic_string` este o clasă container care satisface toate cerințele unei secvențe. Pentru că suportă iteratori cu acces aleatoriu, este și un container reversibil [4].

Iteratori

Fiecare container definește propriul tip de iterator astfel încât, în general, nu este necesară includerea în mod separat a unui fișier antet pentru iteratori. Totuși, există unele tipuri speciale de iteratori pentru care trebuie inclus antetul `<iterator>`.

Orice se comportă ca un iterator, este iterator. De exemplu, un pointer este un iterator. Dar un iterator nu este obligatoriu pointer. Dimpotrivă, de regulă, iteratorii sunt obiecte ale unor clase, pentru care s-au definit operațiile specifice pointerilor.

Iteratorii diferă între ei prin abilitățile lor. De exemplu, un algoritm rapid de sortare necesită iteratori care să permită accesul aleatoriu la elemente.

Standardul Internațional C++ definește următoarele cinci categorii de iteratori, în acord cu operațiile definite:

- *Input iterators* (iteratori de intrare)
- *Output iterators* (iteratori de ieșire)
- *Forward iterators* (iteratori de înaintare)
- *Bidirectional iterators* (iteratori bidirecționali)
- *Random access iterators* (iteratori cu acces aleatoriu)

Input iterators admit cele mai puține operații. Aceștia pot să se deplaseze doar înainte și se deplasează doar un pas o dată. Prin urmare, admit operatorul de incrementare `++` pentru avansarea la următorul element, dar nu și `--`. Pot numai să citească elementele, dar nu să le modifice, deoarece operatorul de dereferențiere `*` returnează o *right-value* și nu o *left-value*. Pot să citească doar o singură dată ceea ce indică. Modelează pointeri care citesc din *stream*-uri de intrare. Iteratorii de tip `istream_iterator` din *Biblioteca C++ Standard* sunt reprezentativi pentru această categorie. Datorită limitărilor, se pot folosi numai cu algoritmi care necesită o singură trecere prin secvență.

Output iterators sunt analogi, dar sunt adaptați pentru ieșire: aceștia se deplasează numai înainte, se deplasează un pas o dată, pot numai să scrie ceea ce indică și pot să scrie doar o dată într-o anumită locație. Modelează pointeri care scriu în *stream*-uri de ieșire. Iteratorii de tip `ostream_iterator`, din *Biblioteca C++ Standard*, sunt reprezentativi pentru această

categorie. Datorită limitărilor se pot folosi numai cu algoritmi care necesită o singură trecere prin secvență.

Forward iterators satisfac toate cerințele iteratorilor de intrare și de ieșire. Permit doar acces unidirecțional în secvență. În plus, pot să citească sau să scrie ceea ce indică, mai mult decât o dată. Containerul nestandard *slist* care implementează o listă simplu înlănțuită, are iteratori din această categorie.

Bidirectional iterators adaugă *forward iterator*-ilor abilitatea de a se mișca înapoi în secvență. Suportă operatorul de decrementare ($--$). Pot fi folosiți oriunde este specificat un iterator de intrare, de ieșire sau unul de înaintare. În această categorie intră iteratorii containerelor *list*, *set*, *multiset*, *map*, *multimap*.

Random access iterators reprezintă categoria cea mai „puternică” de iteratori. Satisfac toate cerințele iteratorilor bidirecționali, dar suportă în mod suplimentar operatorul $[]$, pentru accesul indexat la elementele secvenței. De asemenea, suportă aritmetica iteratorilor, similară cu aritmetica pointerilor. Astfel, se poate adăuga sau scade un întreg pozitiv unui iterator, pentru a „sări” înainte sau înapoi o distanță arbitrară, în timp constant. Containerele care definesc astfel de iteratori sunt *vector*, *string*, *deque* [4].

3.2. Kitul de dezvoltare software *PSPSDK*

Un kit de dezvoltare software (*Software Development Kit – SDK*) este un set de unelte software ce permit unui programator să creeze aplicații pentru un pachet software, framework, platformă hardware, consolă de jocuri video, sistem de operare etc.

Poate fi un simplu *API* (*application programming interface* – interfață pentru programarea de aplicații), care asigură accesul unei aplicații la serviciile unui sistem, sau hardware specializat de comunicare cu un sistem înglobat. În general, uneltele dintr-un astfel de pachet includ soluții de depanare și utilități folosite într-un mediu de dezvoltare (*integrated development environment – IDE*). De asemenea, un SDK mai cuprinde și exemple de cod, alături de alte documente cu rol de sprijin în folosirea corectă a kitului [5].

Kitul de dezvoltare software pentru *PlayStation Portable* (*PSPSDK*) este o colecție de unelte și librării cu sursă deschisă scrisă pentru consola *PlayStation Portable*. Conține, de

asemenea, și documentație și alte resurse de care programatorii se pot folosi pentru a-și crea propriile aplicații pentru *PlayStation Portable*.

Printre facilitățile kitului de dezvoltare software *PSPSDK* se numără:

- Librării și fișiere antet pentru comunicarea cu sistemul de operare a consolei (procese, fișiere, driver video, Wi-Fi)
- Suport în timpul rulării (*crt0*) pentru librării și executabile
- Librăria Standard C (*libc*) cu suport pentru alocarea memoriei, formatarea șirurilor de caractere etc.
- Implementarea librăriei *libGU* ce asigură interfața cu procesorul grafic *Graphic Engine* al consolei
- Implementarea librăriei *libGUM* ce asigură interfața necesară manipulării matricilor în vederea folosirii acestora în software 3D
- Librărie audio pentru redarea sunetelor
- Unelte pentru dezvoltarea de librării statice sau dinamice

Printre uneltele găsite în kitul *PSPSDK* se regăsesc:

- *bin2c*, *bin2o* și *bin2s* pentru convertirea fișierelor binare în cod sursă C, fișiere obiect, respectiv cod mașină
- *mksfo* pentru crearea fișierelor *PARAM.SFO*
- *pack-pbp* și *unpack-pbp* pentru adăugarea, respectiv eliminarea fișierelor dintr-un *EBOOT.PBP*
- *psp-prxgen* pentru convertirea fișierelor ELF în fișiere PRX
- *psp-build-exports* pentru crearea de tabele de export
- *psp-fixup-imports* pentru repararea tabelelor de import după link-editare și înlăturarea funcțiilor ce nu sunt folosite de către executabil [6].

MinPSPW

Este o portare a kitului de dezvoltare software *PSPSDK* pe platforma *Windows*. Cu ajutorul acestuia, un programator care dorește să dezvolte aplicații pentru *PlayStation Portable* nu mai este nevoit să folosească platforma *Linux* sau să folosească emulatorul *Cygwin*, compilarea realizându-se direct din *Command Prompt*. În plus față de asta, *MinPSPW* conține și

librării utile dezvoltării jocurilor și aplicațiilor *homebrew*, precum *Allegro 4*, *Bullet Physics*, *CubicVR*, *Open Dynamics Engine*, *OldSchool Library* și multe altele.

3.3. Librăria *OldSchool Library* pentru *PlayStation Portable*

O librărie este o colecție de funcții sau clase ce alcătuiesc o interfață prin intermediul căreia o aplicație poate executa rutinele librăriei. Un „înveliș” pentru o librărie (*wrapper library*) este un nivel de cod adăugat peste librăria originală ce are rol de „traducere” a interfeței librăriei într-una compatibilă. Motivul poate fi oricare din următoarele:

- Pentru a perfecționa o interfață slab implementată sau care este prea complicată
- Pentru a asigura buna funcționare a codului (ex. evitarea incompatibilităților de tipuri de date)
- Pentru a facilita scrierea de cod portabil [7].

Librăria *OldSchool Library* este un exemplu de astfel de „înveliș” de librărie și este proiectat pentru a ajuta în dezvoltarea de aplicații 2D pe consola *PlayStation Portable*. Pune la dispoziția programatorului întreaga putere de calcul a procesorului grafic al consolei fără ca acesta să fie nevoit să învețe să utilizeze funcțiile complicate ale librăriilor din kitul de dezvoltare software *PSPSDK*. Librăria *OldSchool Library* permite:

- Afișarea imaginilor *PNG* sau *GIF* cu canal alfa
- Deformarea imaginilor prin operații de tip mărire/micșorare, rotație, transparentă, nuanțare, filtrare biliniară etc.
- O rată de umplere a ecranului de peste 100.000 de imagini pe secundă
- Citirea și scrierea pixel cu pixel a imaginilor
- Afișarea mesajelor de depanare în dialoguri sau în consolă
- Redarea de fișiere audio
- Citirea tastelor apăsate
- Fonturi personalizate
- Stabilirea frecvenței cadrelor



Fig. 3.1. Exemple de aplicații ce pot fi realizate folosind
 librăria *OldSchool Library*

3.4. Depanarea aplicației

3.4.1. Configurarea aplicațiilor *Eclipse* și *PSPLink*

Depanarea este procesul prin care se găsesc și se repară defectele unui program în încercarea de a-l face să se comporte cum este de așteptat. Fără o soluție de depanare a aplicațiilor realizate pentru consola *PlayStation Portable*, programatorul ar fi nevoit ca după fiecare modificare în cod să recompileze, să copieze executabilul și eventualele fișiere pe care acesta le folosește și să spera că totul funcționează. În cazul în care este descoperit un defect, programatorul nu poate decât să încerce să ghicească de unde provine eroarea.

Depanatorul GNU, sau *GDB*, este un depanator portabil capabil de a rula pe mai multe sisteme de tip *Unix* și funcționează cu mai multe limbaje de programare, printre care și C/C++. *GDB* oferă o gamă largă de facilități de urmărire și modificare a execuției unui program. Utilizatorul poate monitoriza și modifica variabile și poate chiar să apeleze funcții indiferent de comportamentul obișnuit al aplicației [8].

PSPLink este o aplicație pentru consola *PlayStation Portable* care permite rularea programelor dezvoltate direct de pe calculatorul care realizează compilarea, eliminând astfel nevoia de a copia executabilul în memoria consolei în vederea testării rezultatului. Aplicația are o interfață peste USB sau Wi-Fi ce permite comunicarea cu o consolă de *Linux* sau *Command*

Prompt. Astfel se poate controla comportamentul aplicației direct din consolă, se pot realiza dump-uri ale memoriei sau se poate depana o aplicație folosind depanatorul *Remote GDB*.

Adăugarea opțiunii „-g” la variabila *CFLAGS* din fișierul *Makefile* îi spune compilatorului să includă informații de depanare când compilează codul sursă. Programul *PSPLink* ne permite numai depanarea fișierelor *PRX* (librării statice), deci tot în fișierul *Makefile* se mai adaugă linia *BUILD_PRX=1* pentru a crea și un *PRX* pe lângă executabilul propriu-zis, anume fișierul *PBP*. După următoarea compilare, aplicația va fi pregătită pentru depanare.

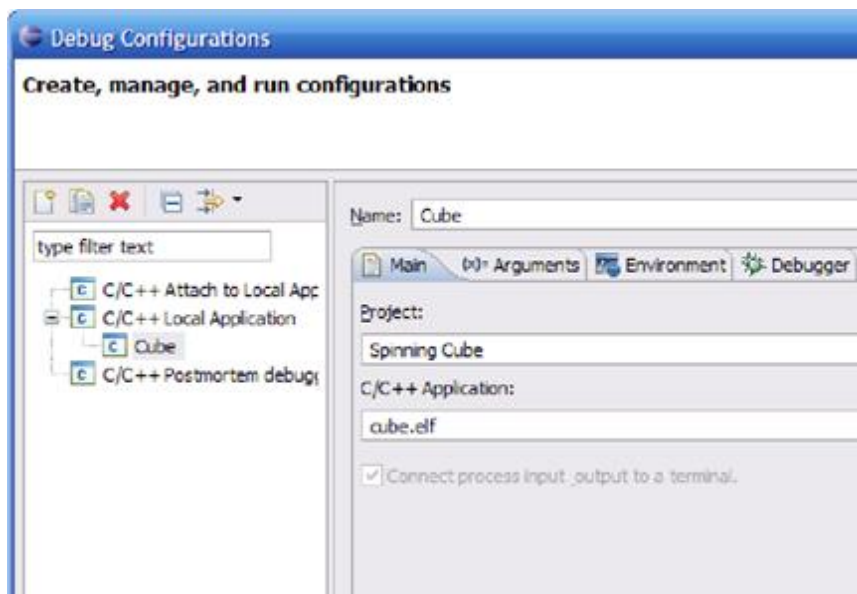


Fig. 3.2. Configurarea depanatorului *Eclipse* 1

Se conectează consola *PlayStation Portable* la calculator folosind un cablu *USB* și se rulează aplicația *PSPLink*. Pe calculator se rulează aplicația *USBHostFS* și *PSPSh* din același director cu executabilul ce urmează a fi depanat. În consola corespunzătoare aplicației *PSPSh* se scrie *debug app.prx*, unde *app.prx* este fișierul *PRX* corespunzător aplicației ce se dorește a fi depanată, și se apasă tasta *Enter*. În *Eclipse* se configurează depanatorul după cum urmează: în meniul *Debug Configurations* se creează o configurație nouă sub tipul *C/C++ Local Application* (vezi figura 3.2), se înlocuiește *gdb* cu *psp-gdb* (vezi figura 3.3), se selectează opțiunea *Debugger*, iar la port se trece *10001* (vezi figura 3.4). Depanatorul este pregătit și va permite inspectarea codului în timp ce rulează aplicația pe consola *PlayStation Portable*.

3.4.2. Testarea performanței

Câteodată, depanarea nu ajunge pentru a dezvolta o aplicație performantă. Există posibilitatea ca jocul să ruleze cu o frecvență a cadrelor nesatisfăcătoare, deși nu există erori sau defecte vizibile. În continuare voi prezenta un program de analiză statistică a codului care furnizează informații precum de câte ori a fost apelată o anumită funcție și cât timp a durat execuția sa. Pentru asta se va folosi librăria *gprof*

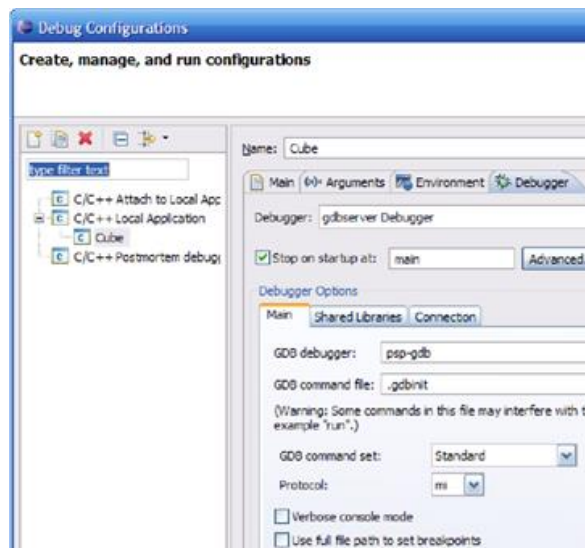


Fig. 3.3. Configurarea depanatorului Eclipse 2

care este inclusă în kitul de dezvoltare software *PSPSDK*. Prin adăugarea opțiunii *-pg* în fișierul *Makefile* și adăugarea unui apel la funcția *gprof_cleanup()* în cod printre ultimele linii executate se va scrie în directorul aplicației un fișier cu date statistice despre rularea acesteia. Fișierul intitulat *gmon.out* se analizează folosind programul *psp-gprof*, rezultatul fiind prezentat în figura 3.5.

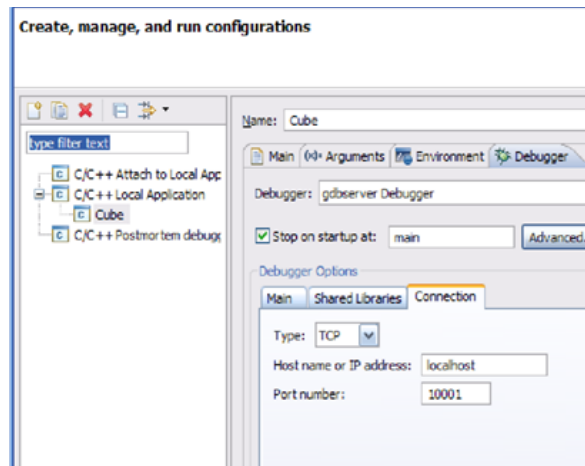


Fig. 3.4. Configurarea depanatorului Eclipse 3

Acest lucru ne oferă o privire de ansamblu asupra performanței codului în scopul optimizării acestuia.

```
Flat profile:
Each sample counts as 0.001 seconds.
% cumulative self self total
time seconds seconds calls s/call s/call name
49.78 2.00 2.00 2 1.00 1.00 wait1second
49.78 4.00 2.00 1 2.00 2.00 wait2seconds
0.42 4.02 0.02 1 0.02 4.02 main
0.02 4.02 0.00 1 0.00 0.00 SetupCallbacks
0.00 4.02 0.00 10 0.00 0.00 loops_10_times
0.00 4.02 0.00 8 0.00 0.00 nested
0.00 4.02 0.00 1 0.00 0.00 gprof_cleanup
```

Fig. 3.5. Rezultatul analizei performanței unei aplicații

3.4.3. Înțelegerea mesajelor de eroare

În cazul în care aplicația întâmpină o eroare și se oprește, în consola programului *PSPSh* va apărea ceva asemănător cu figura 3.6.

```
host1:/> Exception - Bus error (data)
Thread ID - 0x048E6D07
Th Name   - user_main
Module ID - 0x00CD497B
Mod Name  - "PSPWebCam"
EPC       - 0x08805518
Cause     - 0x1000001C
BadVAddr  - 0x0B089540 Status - 0x20088613
zr:0x00000000 at:0xDEADBEEF v0:0x00000000 v1:0x000000FF a0:0x00000000
a1:0x4400C754 a2:0x00000008 a3:0x4400BF54 t0:0x00000008 t1:0x00000080
t2:0x00000000 t3:0x00000001 t4:0x00000000 t5:0x4400BF54 t6:0x000000DB
t7:0xDEADBEEF s0:0x00000015 s1:0x0BBBF34 s2:0x00000001 s3:0x0BBBFEE0
s4:0x00000015 s5:0x00000013 s6:0xDEADBEEF s7:0xDEADBEEF t8:0xDEADBEEF
t9:0xDEADBEEF k0:0x0BBBF00 k1:0x00000000 gp:0x0882B2E0 sp:0x0BBBFDf0
fp:0x0BBBFDf0 ra:0x08804CC0 0x08805518: 0xA0400000 '..@.' - sb $zr,
```

Fig. 3.6. Eroare în execuția aplicației

La prima vedere pare prea criptic pentru a se putea înțelege. Folosind adresa din *EPC* putem calcula unde se află în cod instrucțiunea care a cauzat eroarea. Deoarece nu rulăm executabilul *PBP* propriu-zis, ci fișierul *PRX*, nu cunoaștem adresa de baza a aplicației (spre deosebire de fișierele *PBP*, cele *PRX* sunt relocabile, adică nu sunt încărcate mereu în același loc în RAM, ci unde se găsește memorie liberă). Pentru a calcula adresa la care este încărcat fișierul *PRX* executăm instrucțiunea următoare în consola *PSPLink*:

```
calc $epc-$mod
```

După ce obținem adresa de memorie, putem folosi un program inclus în kitul de dezvoltare software *PSPSDK*, numit *psp-addr2line*. Se deschide o fereastră *Command Prompt* și se execută comanda:

```
psp-addr2line -fe app.elf 0x08805518
```

înlocuindu-se *app.elf* cu numele fișierului *ELF* și numărul hexazecimal cu adresa de memorie obținută mai devreme [9].

4. ARHITECTURA JOCURILOR ȘI A JOCULUI N.A.S.S.

Dezvoltarea jocurilor video este analogă disciplinelor arhitecturale și de inginerie. Așa cum un arhitect proiectează o clădire, la fel procedează și un designer de jocuri în realizarea lor, folosind în parte aceleași unelte software. Iar asemeni un inginer care construiește clădirea proiectată de arhitect, un dezvoltator software realizează jocul proiectat de designer. Dezvoltarea de jocuri implică atât artă cât și inginerie.

4.1 Motorul jocului (*game engine*)

Pentru crearea unui motor de joc performant, chiar și pentru jocuri 2D, este foarte importantă scrierea de cod care nu depinde de un anumit program de dezvoltare software. În alte cuvinte, motorul poate fi compilat de către mai multe compilatoare fără a fi necesare modificări la nivel de cod.

De ce avem nevoie de un motor de jocuri?

Care sunt avantajele folosirii unui motor de jocuri spre deosebire de scrierea codului de care avem nevoie în momentul în care avem nevoie de acesta? De ce să scriu un motor de jocuri când pot în tot acest timp să dezvolt direct jocul?

Cel mai simplu răspuns ar fi: Nu avem nevoie de un motor de jocuri pentru a realiza un joc. Însă asta implică fie că jocul e foarte simplu, fie că deja avem la dispoziție cod din alte proiecte pe care l-am putea refolosi. Un motor de jocuri permite realizarea foarte simplă a unui joc, ușurință pe care numai librării API precum *Direct3D* sau *OpenGL* pentru grafică, *DirectInput* pentru citirea intrărilor de la mouse și tastatură, *Winsock* pentru rețele, *FMOD* pentru sunet, și altele [10].

Ce este un motor de jocuri?

De cele mai multe ori, jocul și motorul său nu sunt două entități separate. Într-un joc, funcțiile care se ocupă de randare pot „știi” cum să deseneze un orc, pe când în alt joc, alte funcții de randare ar putea asigura doar niște baze de care s-ar folosi programatorul pentru randarea mai multor modele printre care s-ar afla și un orc.

Fără îndoială că o arhitectură orientată pe date face diferența dintre un motor de jocuri și un simplu joc. Când aplicația are regulile și logica hard-codate, sau implementează cod specializat pe randarea unor anumite obiecte, re folosirea codului în alte proiecte devine aproape imposibilă. Vom rezerva termenul de „motor de jocuri” soluțiilor software care pot fi extinse și care pot fi folosite ca și fundație pentru o gamă largă de jocuri fără a fi necesare modificări majore asupra codului [11].

În continuare vor fi prezentate interfețele claselor care alcătuiesc motorul jocului *Not Another Space Shooter (N.A.S.S.)*.

Clasa *Game*

```
class Game
{
    Game () ;
    ~Game () ;
    void Run () ;
};
```

Inițializează motorul jocului și jocul în sine. Constructorul `Game ()` inițializează componentele sistemului de operare (grafica, sunetul etc.), cât și subsistemele motorului și încarcă în memorie resursele necesare jocului. Destructorul `~Game ()` eliberează memoria și închide aplicația, iar `Run ()` conține bucla principală a jocului.

Clasa *Drawable*

```
class Drawable
{
    Drawable(string pathToImgFile);
    Drawable(const Drawable *drawableImg);
    ~Drawable();

    int GetPositionX();
    int GetPositionY();
    int GetWidth();
    int GetHeight();

    unsigned GetPixel(int coordX, int coordY);
};
```

```

    void SetPositionX(int coordX);
    void SetPositionY(int coordY);
    void SetPositionXY(int coordX, int coordY);

    void MoveX(int offsetX);
    void MoveY(int offsetY);
    void MoveXY(int offsetX, int offsetY);

    void Draw();
    void Draw(int coordX, int coordY);
};

```

Această clasă conține un set de metode utile pentru manipularea și afișarea imaginilor pe ecran. Constructorul `Drawable()`, în funcție de tipul parametrului, fie încarcă o imagine dintr-un fișier, fie copiază o imagine care este deja încărcată.

Metodele `GetPositionX/Y()` și `GetWidth/Height()` întorc informații referitoare la poziția imaginii pe ecran, respectiv dimensiunile sale în pixeli. `SetPositionX/Y/XY()` mută imaginea la poziția specificată de către programator, pe când `MoveX/Y/XY()` mută imaginea cu un număr de pixeli specificat în parametrii metodei pe o anumită axă.

Metoda `GetPixel()` returnează valoarea *RGBA* a unui pixel aflat la coordonatele specificate de programator, iar `Draw()` desenează pe ecranul consolei, la coordonatele specificate, imaginea.

Clasa `AnimatedSprite`

```

class AnimatedSprite : public Drawable
{
    AnimatedSprite(
        string pathToImgFile,
        int frameWidthSize,
        int frameHeightSize,
        int framesPerSecond);
}

```

```

AnimatedSprite(const AnimatedSprite *animatedSprite);

int GetFrameWidth();
int GetFrameHeight();

unsigned GetFramePixel(int coordX, int coordY);

void ResetAnimation();
void InvertAnimation(bool isInverted);
bool IsAnimationInverted();
void SetAnimationSpeed(int framesPerSecond);
int GetAnimationSpeed();

int GetCurrentFrame();
int GetFrameCount();

void Render();
};

```

Clasa `AnimatedSprite` extinde clasa `Drawable` cu metode utile pentru randarea de imagini animate. La instanțierea obiectului se încarcă o imagine ce conține toate cadrele animației și se stabilesc dimensiunile cadrelor și viteza de animație în cadre pe secundă. O parte din metodele clasei sunt asemănătoare cu cele din clasa `Drawable`, cu diferență că se aplică pe cadrul curent la care s-a ajuns.

Se permite resetarea animației, adică poziționarea iteratorului la cadrul 1, prin intermediul unui apel al metodei `ResetAnimation()`, cât și inversarea sensului de deplasare a iteratorului asupra cadrelor cu metoda `InvertAnimation()`.

Clasa *Font*

```

class Font
{
    Font(string fontFile);
    ~Font();

```

```

    void LoadFont(string fontFile);
    void DrawText(string text, int posX, int posY);
    void DrawTextCentered(string text, int posY);
    void DrawTextAlignedRight(string text, int posY);
    int GetTextWidth(string text);
};

```

Clasa `Font` permite încărcarea și utilizarea fonturilor de tip *SFont*. Asemănător imaginilor compuse din cadre ale unei animații, fiecărui cadru din imagine îi corespunde câte o literă. Când se apelează metoda `DrawText()`, fiecare literă din șirul de caractere `text` este analizată și cadrul corespunzător acelei litere este afișat pe ecran.

Clasa Sound

```

class Sound
{
    Sound(string pathToSndFile, bool isStreamed);
    ~Sound();

    bool Play();
    bool PlayLooped();
    void Pause();
    void Resume();
    void PlayPause();
    void Stop();
    bool IsPlaying();
};

```

Clasa `Sound` pune la dispoziție o serie de operații asupra unei resurse audio. La apelarea constructorului, în funcție de valoarea de adevăr a parametrului `isStreamed`, fișierul audio este fie încărcat în memorie, fie citit direct de pe *Memory Stick*. Sunt asigurate metode pentru redarea, redarea repetată, pauzarea, reluarea, comutarea între stările de redare și pauză și oprirea resursei audio. Consola *PlayStation Portable* asigură, prin intermediul librăriei *OldSchool*

Library, numai 8 canale pe care pot fi redate sunete pe care le gestionează o altă clasă numită `Audio`. Astfel, un sunet nu îl va înlocui niciodată pe altul din greșeală.

Clasa `Controller`

```
class Controller
{
    enum Keys
    {
        DPAD_UP,
        DPAD_LEFT,
        DPAD_DOWN,
        DPAD_RIGHT,
        TRIANGLE,
        SQUARE,
        CROSS,
        CIRCLE,
        L,
        R,
        START,
        SELECT,
        HOLD
    };

    enum RemoteKeys
    {
        PLAYPAUSE,
        FORWARD,
        BACK,
        VOL_UP,
        VOL_DOWN,
        RM_HOLD
    };
};
```

```

Controller();

void ReadKeys();
void ReadRemoteKeys();
bool IsPressed(Keys pressedKey);
bool IsHeld(Keys heldKey);
bool IsPressedRemote(RemoteKeys pressedRemoteKey);
bool IsHeldRemote(RemoteKeys heldRemoteKey);
int AnalogX();
int AnalogY();
void WaitAnyKey();
};

```

Clasa `Controller` pune la dispoziție o colecție de metode pentru citirea stării joystick-ului analogic și a tastelor. Metoda `ReadKeys()` trebuie apelată la fiecare ciclu a jocului pentru a putea citi starea tastelor, iar metodele `IsPressed()` și `IsHeld()` indică dacă o tastă a fost apăsată, respectiv dacă încă este apăsată. Metodele `AnalogX()` și `AnalogY()` returnează o valoare corespunzătoare poziției joystick-ului analogic pe axele Ox, respectiv Oy.

Clasa `BinaryFileIO`

```

class BinaryFileIO
{
    BinaryFileIO(string pathToFile);
    ~BinaryFileIO();

    bool Write(const void* data, size_t size);
    bool Read(void* data, size_t size);
    bool IsFileOpen();
};

```

Clasa `BinaryFileIO` pune la dispoziție metode pentru manipularea fișierelor binare. Se pot scrie și citi structuri întregi de date, ceea ce permite salvarea stării jocului și revenirea la aceasta mai târziu.

Clasa Timer

```
class Timer
{
    Timer();
    unsigned long getStartTimeMs();
    void Sleep(int ms);
    void ResetStartTime();
    void ResetStopwatch();
    bool Stopwatch(int ms);
};
```

Clasa Timer implementează operații pe variabile de tip `clock_t`. Cu ajutorul acesteia se poate calcula timpul în milisecunde dintre două evenimente, sau se pot sincroniza evenimente în funcție de timp folosind metoda `Stopwatch()`. Clasa este utilă în calcularea frecvenței cadrelor sau a animării imaginilor, printre altele.

4.2 Arhitectura pe trei nivele

Se poate lua fiecare subsistem al jocului și se poate clasifica ca aparținând uneia din trei categorii: nivelul de comandă și control, nivelul de logică și nivelul interfețelor. Nivelul de comandă și control se ocupă de hardware și de sistemul de operare, nivelul de logică gestionează stările jocului și felul în care acestea se modifică, iar nivelul interfețelor prezintă starea jocului prin sunet și grafică.

Această arhitectură este similară cu arhitectura *MVC (Model-View-Controller)*, care dorește să separe logica unui sistem de interfața folosită pentru a prezenta sau a solicita modificări ale datelor. Arhitectura propusă mai sus înglobează arhitectura *MVC* și îi adaugă un „strat” pentru gestionarea hardware-ului și a componentelor sistemului de operare.

Nivelul de comandă și control este dependent de platforma pe care rulează jocul. În cazul unei eventuale portări pe altă platformă (ex. de pe *PlayStation 3* pe *Xbox 360*) aproape întregul cod asociat acestui nivel trebuie rescris, iar celelalte două nivele ar rămâne (în mare parte) intacte. Nivelul de comandă și control gestionează dispozitive precum mouse-ul, tastatura sau

gamepad-ul sau servicii ale sistemului de operare precum comunicarea în rețea sau firele de execuție.

Nivelul de logică este jocul în sine, care este complet separat de platforma pe care rulează sau de felul în care este prezentat acesta jucătorului. Într-o lume ideală, codul asociat acestui nivel ar putea fi recompilat pentru orice platformă sau sistem de operare fără a fi necesare modificări asupra sa. Nivelul de logică înglobează subsistemele necesare gestionării stărilor jocului și comunicării cu celelalte sisteme. Un exemplu bun ar fi un motor de fizică care decide cum interacționează obiectele din joc.

Al treilea și ultimul nivel este nivelul interfețelor. Acest nivel e responsabil de prezentare stării actuale a jocului și traducerea datelor de intrare în comenzi pe care le înțelege nivelul de logică. Nivelul interfeței poate avea implementări diverse, iar un joc poate avea mai multe interfețe. Un tip de interfață poate fi cea pentru utilizatori, care desenează starea actuală a jocului pe ecran, redă sunete și primește comenzi prin intermediul perifericelor de intrare. Alt tip de interfață poate fi cea pentru inteligența artificială implementată în joc, iar un al treilea tip de interfață poate fi pentru un jucător de pe rețea. Toate primesc aceleași stări din partea nivelului de logică, doar că le interpretează în moduri diferite [12].

4.2.1 Nivelul de comandă și control

Conținutul nivelului de comandă și control este alcătuit din mai multe subsisteme care gestionează periferice, sistemul de operare și „durata de viață” a jocului (vezi figura 4.1).

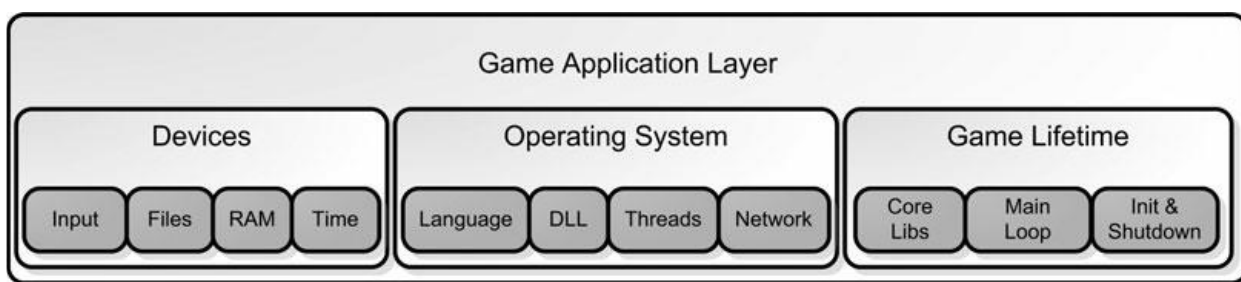


Fig. 4.1. Componentele nivelului de comandă și control [12]

Citirea datelor de intrare

Jocurile pot avea o varietate de periferice de intrare: tastatură, mouse, gamepad, joystick, volan pentru simulatoare auto, cameră, accelerometru, *GPS* etc. Metoda de citire a datelor de intrare de la aceste periferice diferă de la o platformă la alta, depinzând de drivere și de sistemul de operare. Stare acestor periferice sunt „traduse” pentru a fi trimise nivelului de logică (ex.

„lansează racheta”) sau nivelului interfeței (ex. „deschide meniul cu setări”). Acest subsistem este implementat în clasa `Controller`. La fiecare buclă se citește starea butoanelor consolei. Ulterior, în nivelul de logică se verifică dacă vreun buton a fost apăsat, dacă încă este ținut apăsat sau dacă a fost mișcat joystick-ul analogic.

Sistemul de citire/scriere fișiere

Acest sistem definește operațiile de citire și scriere a fișierelor de pe dispozitive de stocare a datelor, precum DVD-ROM-ul, hard disk-ul și cardul de memorie. Codul din acest subsistem este responsabil de gestionarea resurselor jocului și de salvare și încărcarea stării jocului. Acest subsistem este implementat în clasa `FileIO`. De această clasă se folosește la rândul ei clasa `SaveLoad`. Sunt salvate informații privind starea actuală a jocului într-un fișier binar pentru a se putea reveni mai târziu la aceasta.

Gestionarea memoriei

Gestionarea memoriei este un subsistem foarte important pentru un joc. Structurile de date folosite în jocuri au în general dimensiuni reduse și sunt păstrate fie în RAM, fie în memoria video. Un gestionar de memorie ar trebui să cunoască când și de câtă memorie are nevoie jocul pentru alocări și dealocări dinamice. Un exemplu poate fi găsit în clasa `EnemyList`, care alocă memorie pentru un obiect de tip `Enemy` când acesta urmează să apară pe ecran și îl șterge din memorie când acesta este distrus sau părăsește ecranul.

Inițializarea, bucla principală și închiderea unui joc

Aplicații precum *Microsoft Excel* așteaptă ca utilizatorul să facă ceva înainte ca acestea să execute cod. Dacă nu se mișcă mouse-ul sau dacă nu se scrie la tastatură, acestea rămân într-o stare inactivă. Un astfel de comportament este benefic procesorului, deoarece pot fi rulate mai multe aplicații fără ca acesta să fie suprasolicitat. Însă jocurile sunt simulări care au propria „durată de viață”. Chiar și fără acțiuni din partea utilizatorului, nivelul de logică tot poate să trimită o creatură care să înceapă să lovească personajul jucătorului, motivând astfel jucătorul să reacționeze cu niște apăsări de butoane.

Sistemul care controlează astfel de activități este bucla principală și are trei mari componente: citirea datelor de intrare de la periferice, avansarea cu un pas a logicii jocului și prezentarea stării jocului folosind grafică și sunet.

4.2.2 Nivelul de logică

În nivelul de logică este definit universul jocului, obiectele din acel univers și cum interacționează acestea. Cuprinde, de asemenea, reguli de schimbare a stărilor în funcție de stimuli externi, cum ar fi un jucător care apasă tastă sau deciziile unui proces de inteligență artificială [12]. În figura 4.2 este prezentat nivelul de logică și subsistemele sale.

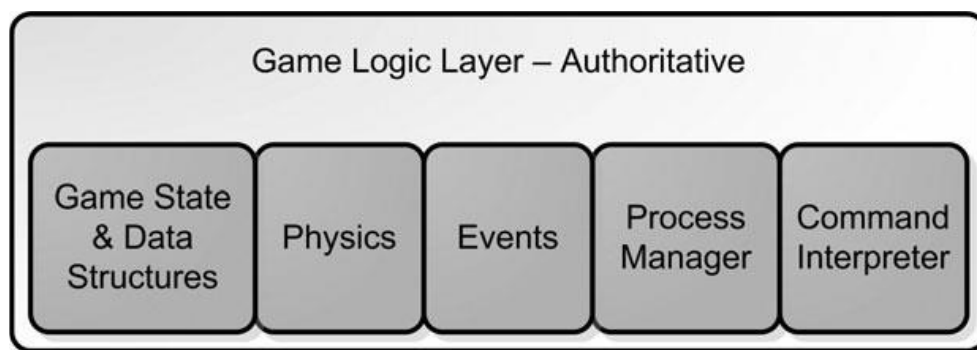


Fig. 4.2. Nivelul de logică și subsistemele sale [12]

Stări și structuri de date

Orice joc are un container pentru obiectele sale. Jocurile mai simple pot folosi o structură de tip listă, pe când jocurile complexe au nevoie de ceva mai flexibil și optimizat, astfel încât motorul jocului să o poată parcurge rapid pentru a modifica stările obiectelor. Proprietățile unor obiecte, precum punctele de viață, sunt în general stocate în structuri de date a căror eficiență pot varia de la joc la joc. *Ultima Online* se folosea de stringuri pentru a defini proprietățile obiectelor, astfel asigurând flexibilitate și ușurință în dezvoltarea jocului, dar utilizând mai multă memorie. Pe când *Thief: Deadly Shadows* avea un sistem de proprietăți foarte complicat ce era orientat pe obiecte. Astfel puteau fi definite un set de proprietăți pentru un arhetip, cum ar fi un scaun, iar prin supraîncărcare se putea crea un anumit scaun care apărea într-o singură locație [12]. Acest sistem era eficient din punct de vedere al memoriei, însă necesita o putere de calcul sporită, întrucât sistemul de proprietăți erau în esență o structură arborescentă.

N.A.S.S. folosește o structură de tip listă dinamică pentru a se putea ține cont de asteroizii prezenți pe ecran și pentru ștergerea lor din memorie în momentul în care aceștia au fost distruși sau dispar de pe ecran.

Fizică și coliziuni

Fizica face parte din subsistemele logicii unui joc și reprezintă regulile universului jocului. Un joc nu necesită un motor de fizică complicat pentru a fi distractiv, dar un motor prost implementat va deveni frustrant pentru un jucător. Când obiectele unui joc sunt abstracte, precum un „stick figure” dintr-un joc *Flash*, jucătorul poate ierta animațiile nerealiste, însă pentru un personaj din *Battlefield 3*, până și cea mai mică eroare de animației faciale poate fi deranjantă.

Acest concept are de-a face cu psihologia umană și de felul în care percepem realitatea, devenind foarte important în momentul în care un joc se apropie de realitate din punct de vedere al prezentării sale. Implementarea sistemului de coliziune a jocului *N.A.S.S.* se află în clasa *CollisionDetection*. O descriere detaliată a algoritmului de detectare a coliziunilor se poate găsi la capitolul 5.

Evenimente

Când logica jocului efectuează o schimbare de stare a jocului alte subsisteme vor reacționa într-un fel sau altul. Spre exemplu, inserarea unui radio cauzează desenarea de poligoane și texturi de către sistemul responsabil de randare, iar sistemul de sunet va reda muzică. Toate aceste sisteme trebuie să fie știe faptul că acest radio există și care este starea sa curentă, iar acest lucru este realizat cu ajutorul evenimentelor.

Majoritatea jocurilor implementează un sistem de evenimente care le definește pe ele și datele care le acompaniază. Un exemplu ar fi sistemul de sunet care redă un sunet de ciocnire când „vede” un eveniment de coliziune în sistemul de evenimente. O arhitectură bazată pe evenimente are tendința de a îmbunătăți eficiența unui joc. În loc să se facă apeluri *API* în patru sau cinci sisteme diferite când două obiecte se lovesc unul de altul, se trimite un eveniment la sistemul de evenimente, iar toate sistemele care trebuie să reacționeze vor fi notificate de către acesta.

Un exemplu de eveniment în jocul *N.A.S.S.* este momentul în care jucătorul activează abilitatea *Dematerialize*, determinând dezactivarea algoritmului de coliziune pe durata folosirii abilității.

Managerul de procese

Orice simulare a lumii unui joc este în general alcătuit din fragmente cod, precum codul necesar mișcării unui personaj pe o direcție liniară. Acționând asupra unui singur obiect permite alcătuirea unui sistem complex din simple schimbări de stări. Fragmentele respective de cod sunt organizate în clase care pot fi instanțiate. Dacă am avea clasele `FugiPeTraiectoria` și `ÎmpușcăÎnDirecția` pe care le-am instanția pe același obiect am obține un efect interesant și complex din interacțiunea celor două clase.

Managerul de procese se ocupă de astfel de operații. Stochează o listă de procese și le apelează la fiecare buclă a jocului. În cazul funcției `FugiPeTraiectoria`, managerul de procese îi execută codul, iar când personajul ajunge la destinație, îi termină execuția.

Pentru Ultima VIII, dezvoltatorii software ai acestui joc au dorit să îmbunătățească managerul de procese folosit în Ultima VII. Au realizat că orice fragment de cod care se executa asupra unui obiect sau a unui grup de obiecte putea fi încapsulat într-un proces secundar care să fie responsabil de propria sa durată de viață. După ce codul ar fi aplicat efectul dorit asupra obiectului, acesta și-ar fi încheiat singur execuția. Avantajul este că tot sistemul putea să fie manageriat folosind o singură clasă care conținea o listă a proceselor ce trebuiau executate, asemănător unui sistem de operare simplist [12].

Procesele pot depinde unul de altul, astfel încât unul să poată aștepta terminarea altuia înainte de a-și începe propria execuție. Un exemplu bun este logoul principal. Se urmărește starea animației acestuia de ieșire de pe ecran, iar la finalizarea execuției se intră în starea `IN_GAME`, iar nava și asteroizii își încep randarea.

Interpretorul de comenzi

Sistemul logic din spatele unui joc trebuie să fie capabil de a răspunde la comenzi externe. În cazul unui simulator auto, aceste comenzi (care pot proveni atât de la un jucător, cât și de la un proces IA) ar fi reprezentate la nivel logic ca și „acelerează”, „frânează”, „întoarce volan”. Spre exemplu, jucătorului îi este prezentată o interfață care interpretează starea „butonul A este apăsat” în comanda „acelerează”, însă nu o execută el, ci o trimite la sistemul logic.

Dacă sistemul logic poate accepta comenzi printr-o interfață bazată pe evenimente în locul unor apeluri API, atunci se poate crea un limbaj de programare în joc. Jucătorul ar putea

scrie într-o consolă `SeteazăProprietateMașină(2, PROPRIETATE_FRÂNĂ, true)`, rezultatul fiind că mașina 2 va apăsa frâna și se va opri.

4.2.3 Nivelul interfețelor

Interfața este o colecție de sisteme care comunică cu sistemul logic pentru a prezenta jocul unui observator. Acest observator poate fi un jucător sau un agent IA, a cărui vedere asupra stării jocului determină următoarea sa decizie. Interfața pentru un jucător trebuie să răspundă evenimentele jocului și să deseneze scena, să redea sunete etc. (vezi figura 4.3).

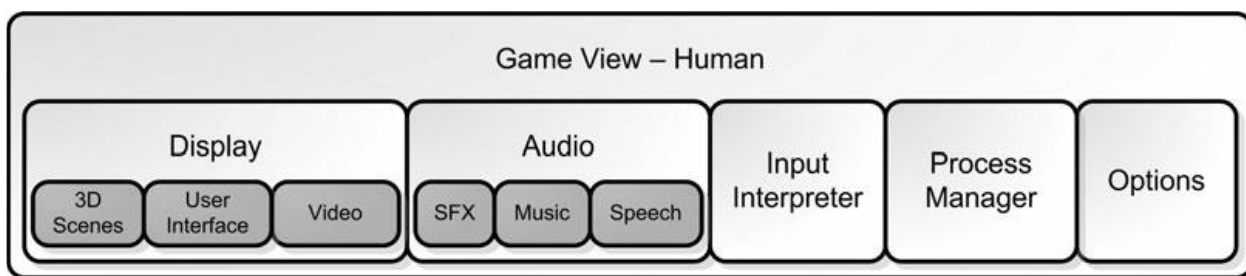


Fig. 4.3. Interfața cu jucătorul [12]

Afișarea grafică

Acest proces randează obiectele care alcătuiesc scena jocului și interfața de pe ecran, și este unul dintre cele mai intensive subsisteme ale unui joc din punct de vedere al folosirii resurselor procesorului. Din acest motiv, motorul jocului ar putea dezactiva efecte neesențiale (precum efectul de *bloom* sau *HDR*) în scopul măririi performanței randării.

O problemă care apare la motoarele 3D este deciderea căror poligoane ar trebuie desenate pe ecran pentru a crea cea mai bună scenă fără să scadă viteza de randare. Pentru a rezolva o astfel de problemă se folosesc diferite nivele de detaliu pe terenuri sau obiecte în funcție de distanța lor față de spectator. Însă e dificilă realizarea unei tranziții fluide în nivelele de detaliu astfel încât să nu apară efectul de *popping*. Altă problemă este evitarea scrierii de prea multe poligoane pe ecran în cazul unei scene complexe, cum ar fi o intersecție aglomerată a unui oraș mare. Cea mai mare viteză de randare se obține prin desenarea exclusivă a poligoanelor vizibile spectatorului scenei.

Sunetul

Sunetele unui joc pot fi categorisite astfel: efecte sonore, muzică și vorbire. Efectele sonore sunt simple de implementat. Pur și simplu sunt încărcate fișiere *WAV* și trimise sistemului audio pentru redare. Se pot crea efecte 3D în funcție de poziția unui observator față de obiectul care produce sunetul prin furnizarea locației obiectului respectiv.

Muzica poate fi ori foarte ușor de implementat, ori foarte greu. Nu e mare diferență față de implementarea unui efect sonor decât în cazul în care se dorește schimbarea muzicii de fundal în funcție de evenimentele jocului (ex. când jucătorul urmează să intre într-o luptă).

Librăria *OldSchool Library* permite redarea a maxim 8 sunete simultan. Pentru a evita înlocuirea unui sunet cu altul din greșeală, am implementat clasa `Audio` care verifică singură starea canalelor și le marchează ca fiind ocupate sau libere. Clasa `Sound` este cea care face apelurile funcțiilor din librăria *OldSchool Library* pentru redarea sunetelor, însă clasa `Audio` decide pe ce canale sunt redade acele sunete.

Interfața *Heads Up Display* (HUD)

Interfața HUD este metoda prin care un jucător primește informații referitoare la stările unor obiecte și interacționează cu jocul. Ea cuprinde butoane pentru accesarea meniurilor, punctele de viață a unui personaj, lista de abilități etc., preferabil într-un mod creativ, elegant și unic pentru fiecare joc în parte. Un exemplu foarte bun de creativitate în design este jocul *Dead Space* al cărei interfață HUD nu există, jucătorii putând să țină cont de punctele de viață ale personajului prin intermediul unui dispozitiv de pe spatele armurii sale, iar harta, notificările etc. sunt vizibile pe o imagine holografică generată tot de sistemele de pe armură (vezi figura 4.4).



Fig. 4.4. Interfața HUD a jocului *Dead Space*

N.A.S.S. folosește o interfață simplă care oferă jucătorului informații precum scorul curent, cel mai mare scor înregistrat și nivelul la care se află energia folosită pentru activarea abilităților. În capitolul 5 se vorbește mai mult despre interfața jocului *N.A.S.S.*

Manager de procese

Asemănător nivelului de logică, interfețele pot beneficia și ele de un manager de procese care să gestioneze orice de la animațiile butoanelor până la redarea conținutului video și audio.

Opțiuni configurabile

Majoritatea jocurilor au opțiuni ce pot fi configurate de către jucător precum volumele efectelor sonore, dacă axa Oy a mouse-ului este sau nu inversată, rezoluția video etc. Astfel de opțiuni pot fi stocate în fișiere XML pentru a putea fi ușor modificate, mai ales pe perioada dezvoltării jocului.

Jocuri multiplayer

Arhitectura logică/interfață permite implementarea cu ușurință a unui sistem multiplayer prin atașarea interfețelor mai multor jucători la același sistem de logică.

Interfața pentru agenți IA

Un avantaj al separării sistemului logic de sistemul de interfețe este că atât jucătorii cât și agenții IA pot interacționa cu nivelul logic prin exact aceeași interfață bazată pe evenimente. Interfața unui agent IA are în general componentele din figura 4.5.

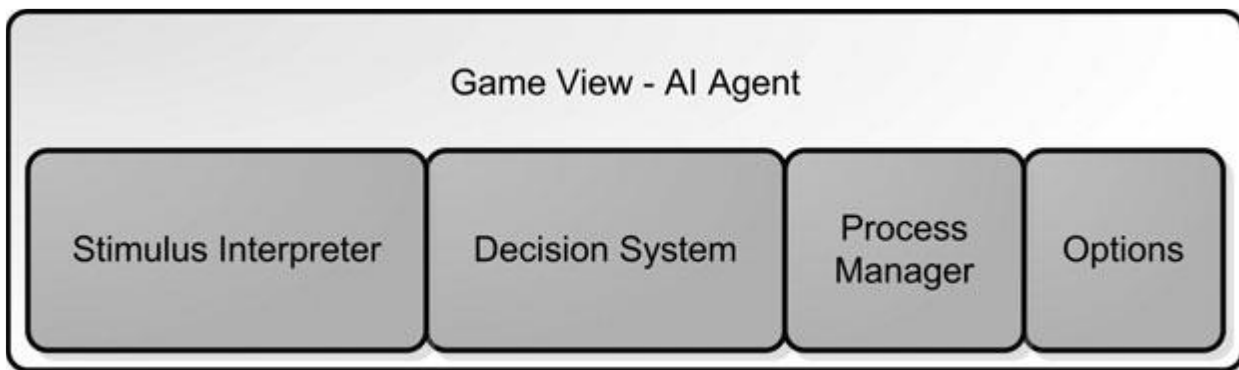


Fig. 4.5. Interfața unui agent IA [12]

Interpretorul de stimuli primește aceleași evenimente pe care le primesc restul interfețelor: mișcarea obiectelor, coliziuni etc, iar agentul IA decide cum va reacționa la aceste evenimente.

A doua parte a interfeței agentului IA este sistemul de decizie care traduce stimulii în acțiuni. Agentul IA ar putea trimite comenzi jocului pe care un jucător nu ar putea (sau viceversa), precum deschiderea ușilor încuiate. Dacă agentul IA trebuie să rezolve probleme dificile, cum ar fi navigarea printr-un labirint, atunci se poate folosi un manager de procese pentru eficientizarea procesului. Astfel se poate distribui evaluarea stimulilor pe o perioadă mai lungă de timp, amortizând astfel impactul unor calcule complexe asupra vitezei de randare.

În final, este utilă implementarea unei liste de configurații ce pot fi citite dintr-un fișier text. Un filtru pentru stimuli și un set de decizii pot determina un fișier mare, însă se asigură astfel o ușurință de configurare a comportamentului agentului IA pe durata dezvoltării jocului.

4.3 Arhitectura multiplayer online

Arhitecturii prezentată mai sus permite implementarea unui sistem multiplayer prin adăugarea a două clase (vezi figura 4.6). Această arhitectură presupune noi implementări puțin diferite pentru sistemul logic și sistemul interfețelor în plus față de cele descrise anterior.

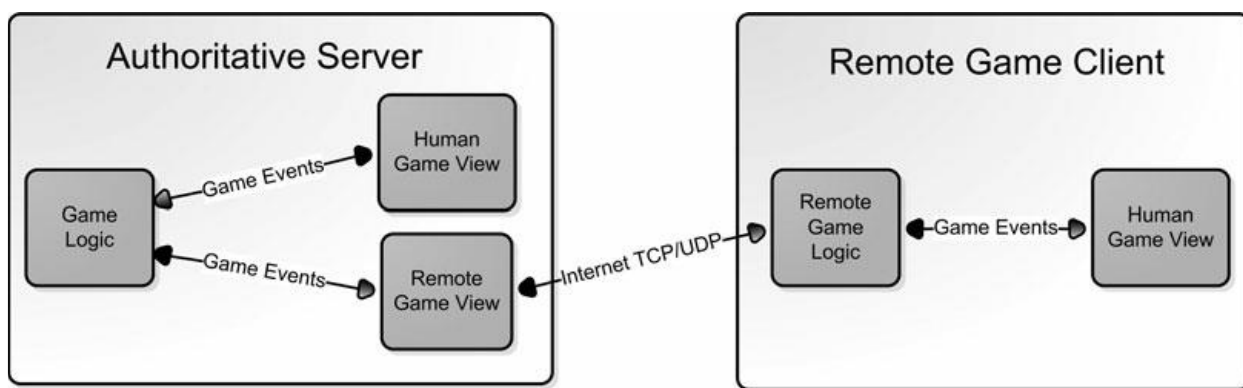


Fig. 4.6. Arhitectura client/server pentru un joc multiplayer online [12]

Interfața de rețea

Pe mașina server, jucătorul de pe rețea ar trebui să pară ca un agent IA. Interfața de rețea primește evenimente din partea sistemului logic al jocului și îi răspunde cu comenzi. Evenimentele primite sunt împachetate și trimise via TCP sau UDP către o mașină client prin intermediul rețelei. Mesajele sunt evaluate înainte ca ele să fie trimise unui client. În primul rând, cele redundante sunt eliminate. Nu are sens să se trimită două evenimente „obiect mișcat” când singurul care contează este ultimul. În al doilea rând, mai multe evenimente sunt împachetate și trimise împreună. Dacă pachetul este prea mare, ar trebui să fie comprimat pentru a se economisi lățime de bandă.

Interfața de rețea primește date de la client, anume comenzile jucătorului de pe rețea și verifică aceste date pentru comenzi imposibile sau alte metode prin care jucătorul de pe rețea ar putea trișa, după care sunt trimise la sistemul de logică de pe mașina server.

Sistemul logic de rețea

În acest model, sistemul logic de pe server are putere de decizie asupra evenimentelor jocului. Desigur, mașina client are nevoie de o copie a stării jocului și de un mod de a compensa întârzierile cauzate de o conexiune slabă și Internet. Aceasta este sarcina sistemului logic de rețea.

Sistemul logic de rețea este asemănător sistemului logic de pe server. Conține tot ce îi trebuie pentru a crea stimuli, chiar și cod care simulează decizii când este necesar. Are două componente pe care sistemul logic de pe server nu le are: cod pentru anticiparea deciziilor finale ale serverului și pentru corectarea acelor decizii. Spre exemplu, dacă de pe server se trimite comanda „lansează racheta”, iar clientul are o conexiune proastă la Internet, mașina clientului va primi mesajul abia după câteva sute de milisecunde după ce racheta a fost lansată. Deoarece nu

poate fi anticipat acest eveniment, mașina client trebuie să compenseze mărin viteza rachetei pentru se sincroniza cu serverul, deci să corecteze decizia serverului.

5. PREZENTAREA JOCULUI N.A.S.S.

Jocurile din genul *arcade* au în general niveluri scurte, simple, controale intuitive și dificultate care crește în timp, deoarece primele jocuri din acest gen au apărut pe consolele din sălile *Arcade* (utilizatorii puteau juca până murea personajul din joc sau până rămâneau fără jetoane). Jocurile pe console sau PC pot fi numite jocuri *arcade* dacă au aceleași calități sau dacă sunt portări directe ale acestora [13].

Un *shoot 'em up* este un joc în care protagonistul luptă împotriva unui număr mare de inamici, împușcându-i și ferindu-se de focul lor. Jucătorul trebuie să se bazeze pe timpii săi de reacție pentru a reuși să facă acest lucru. Acest gen este specific jocurilor *arcade* anilor 1980.

Jocul *Not Another Space Shooter*, sau *N.A.S.S.*, aparține genului *arcade shoot 'em up*, însă cu o diferență de design. Spre deosebire de jocurile *shoot 'em up* clasice care oferă jucătorilor abilități ofensive, *N.A.S.S.* oferă abilități defensive, însă restul elementelor acelui gen sunt păstrate. Perspectiva este de sus în jos, jucătorul controlând o navă spațială care trece printr-o centură de asteroizi. Țelul e de a supraviețui o perioadă cât mai mare folosind abilitățile puse la dispoziție și un timp de reacție foarte bun. O coliziune cu un asteroid va distruge nava, jocul măbind viteza asteroizilor și numărul lor într-un timp foarte scurt.

5.1 Meniul principal



Prin apăsarea butonului  se deschide meniul principal prin intermediul căruia se poate accesa ecranul cu abilități, ecranul cu controale, resetarea salvării jocului, modul de depanare sau terminarea jocului (vezi figura 5.1). Pentru începerea jocului, utilizatorul apasă



Fig. 5.1. Meniul principal al jocului *N.A.S.S.*

butonul  când se află la ecranul cu logoul jocului.

În meniul de abilități se pot cheltui punctele de experiență (*XP*) pentru cumpărarea abilităților și îmbunătățirea lor. În fiecare abilitate se pot investi un maxim de 5 puncte, până la


un maxim de 10 puncte per total, moment în care *Drive Core*-ul devine supraîncărcat (vezi figura 5.2). Astfel, jucătorul este forțat să își folosească strategic punctele *XP* pentru a-și maximiza șansele de supraviețuire, deoarece prin rambursarea unui punct de abilitate se primesc mai puține puncte *XP* decât cât s-a cheltuit pentru cumpărarea punctului respectiv. Apăsarea butonului  determină apariția unui ecran cu informații despre abilitatea selectată (mecanica, costul de energie, scalarea pe niveluri).



Fig. 5.2. Ecranul cu abilități

Ecranul cu controale îi prezintă utilizatorului o imagine a consolei *PlayStation Portable* și butoanele corespunzătoare abilităților și cele responsabile de mișcarea navei pe ecran. Opțiunea *Reset Progress* resetează abilitățile la nivelul 0 și șterge toate punctele *XP* pe care jucătorul le deținea. Opțiunea *Debug Mode* activează sau dezactivează informațiile de depanare de pe ecran (frecvența cadrelor, memoria RAM disponibilă etc.). Opțiunea *Quit* închide jocul, iar consola *PlayStation Portable* revine la meniul sistemului său de operare.

5.2 Mecanica jocului

Scopul jucătorului este de a supraviețui cât mai mult timp folosind abilitățile pe care le are la dispoziție (pe care le-a cumpărat cu puncte *XP*). Cu cât trece mai mult timp, cu atât probabilitatea de a apărea un inamic nou la fiecare cadru (bucă a jocului) crește, iar viteza de bază a asteroizilor (la care este adăugat un modificador generat aleatoriu) crește cu fiecare inamic nou care apare.

La fiecare secundă se actualizează scorul în funcție de viteza de bază a asteroizilor din acel moment, deci cu cât jucătorul supraviețuiește mai mult, cu atât câștigă mai multe puncte pe secundă. Astfel, se răsplătește timpul de reacție a jucătorului și folosirea strategică a punctelor de abilitate. La finalul jocului, jucătorul primește puncte *XP* în funcție de scorul acumulat ($XP \text{ primit} = \text{scor acumulat} / 10$).

5.3 Abilități



În scopul supraviețuirii cât mai îndelungate, jucătorul are la dispoziție 4 abilități ce devin din ce în ce mai potente pe măsură ce se investește mai mult *XP* în ele. Inițial, abilitățile sunt blocate, iar singura modalitate de a supraviețui și a aduna *XP* este printr-un timp de reacție foarte bun.

Pentru ca jocul să nu devină un chin încă din primele 5 minute, primul punct din fiecare abilitate are un cost redus de *XP*, anume 3750, iar rambursarea primului punct din orice abilitate este tot 3750. În acest fel, jucătorul poate testa mecanica tuturor abilităților fără să se teme de *XP* pierdut. Restul punctelor costă mai mult *XP* și restituie mult mai puțin decât costul lor, ajungând la 63750 pentru al cincilea punct al unei abilități și 18750 pentru rambursarea sa (vezi figura 5.2).

Drive Core este sursa de energie a navei, care limitează numărul de puncte care pot fi cumpărate în total la maxim 10 (sau 10% supraîncărcare per punct de abilitate). Această limitare forțează jucătorul să își aleagă strategic abilitățile și numărul de puncte investit în fiecare, astfel încât să aibă o șansă cât mai mare de supraviețuire. Limitarea impusă de *Drive Core* este și motivul principal pentru care există un sistem de rambursare a punctelor: pentru ca jocul să nu ajungă niciodată să pară un chin, sau să fie necesară resetarea progresului pentru că jucătorul a investit într-o abilitate care nu îl ajută pe cât de mult dorea.

Abilitatea Warp

Prima abilitate se numește *Warp*. Folosind această abilitate se poate evita o coliziune iminentă prin teleportarea navei în spatele asteroizilor (vezi figura 5.3). Distanța de teleportare este cu atât mai mare cu cât sunt investite mai multe puncte în abilitate, după cum urmează: 50 / 70 / 100 / 140 / 190 de pixeli.

Pentru a activa abilitatea, energia trebuie să fie la 100%. La apăsarea tastei  este consumat 50% din energie și apare o țintă care indică poziția la care se va teleporta nava la a doua apăsare a tastei. Teleportarea reduce energia la 0%, însă dacă nu se apasă a doua oară tasta  într-un interval de 2,5 secunde, se anulează

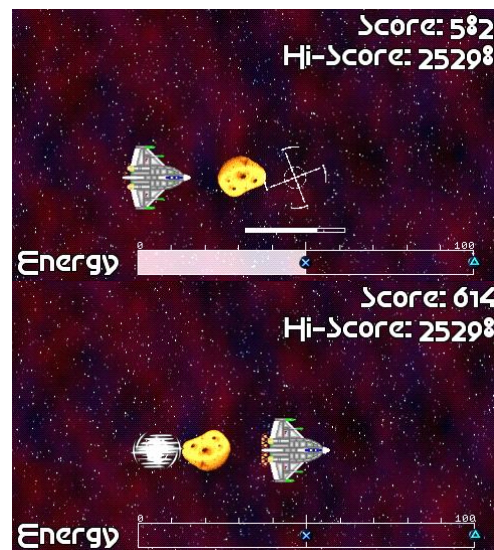



Fig 5.3. Abilitatea Warp

activarea abilității, iar restul energiei nu mai este consumată. Întrucât la activarea abilității *Warp* se consumă numai jumătate din energie, restul de 50% poate fi folosit pentru activarea altei abilități până se realizează teleportarea propriu zisă.

Abilitatea *Dematerialize*

Această abilitate oferă navei invulnerabilitate pe durata activării sale (vezi figura 5.4). Activarea abilității se realizează prin ținerea butonului  apăsat. În funcție de câte puncte au fost investite în abilitate, aceasta consumă din ce în ce mai puțină energie pe durata activării sale astfel: 100 / 90 / 80 / 70 / 50% din maximul de energie pe secundă.

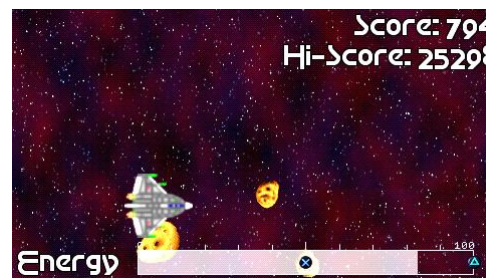


Fig. 5.4. Abilitatea *Dematerialize*

Abilitatea *Overdrive*



Apăsarea și ținerea butonului  cauzează activarea abilității *Overdrive*. Această abilitate consumă 50% din energie pe secundă și mărește viteza navei în funcție de numărul de puncte investite în aceasta după cum urmează: 1,5 / 2 / 2,5 / 3 / 3,5 ori viteza de bază a navei.



Fig. 5.5. Abilitatea *Overdrive*

Abilitatea *Force Field*

Abilitatea *Force Field* se activează prin apăsarea butonului . Se generează un scut care absoarbe impactul cu asteroizii și regenerează 10% din energia maximă la fiecare lovitură. Scutul rezistă pentru o perioadă de timp sau pentru un număr limitat de coliziuni după cum urmează: 1 / 2 / 3 / 4 / 5 secunde sau 1 / 2 / 3 / 4 / 5 coliziuni. Costul energiei variază, de asemenea, cu numărul de puncte: 50 / 55 / 60 / 65 / 70% din energia maximă.

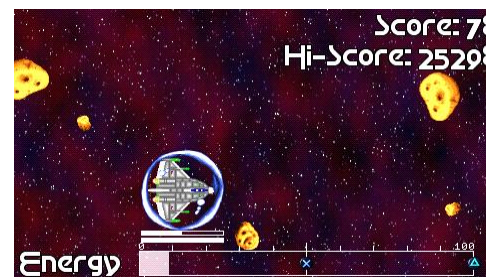


Fig. 5.6. Abilitatea *Force Field*

5.4 Detectarea coliziunilor

Pixel Perfect Collision este un algoritm foarte precis de detectare a coliziunilor obiectelor 2D care folosește o analiză la nivel de pixel pentru a determina dacă două obiecte se suprapun în cel puțin un punct. Este, însă, un algoritm foarte lent în comparație cu cel al obiectelor înscrise într-un cerc, spre exemplu, care calculează distanța centrelor cercurilor care înscriu obiectele și le compară cu suma razelor acestora.

Pentru a testa dacă două obiecte se ciocnesc se verifică fiecare pixel al texturilor celor două obiecte, iar dacă se găsesc doi pixeli netransparenți care se suprapun, înseamnă că cele două obiecte se ciocnesc. Complexitatea acestui algoritm este $O(n^4)$ în cazul în care nici un obiect nu se ciocnește cu altul, deoarece sunt folosite patru instrucțiuni `for` pentru iterarea fiecărui pixel (pozițiile pe axele O_x și O_y) al celor două texturi. O astfel de complexitate nu poate fi folosită într-un joc care trebuie să ruleze la o viteză de 60 de cadre pe secundă.

Plecând de la ideea algoritmului de mai sus, putem verifica mai întâi coliziunea cu un algoritm mai puțin precis, însă mai simplu. Obiectele sunt încadrate într-un dreptunghi, iar dacă se detectează o suprapunere a celor două dreptunghiuri se aplică algoritmul *Pixel Perfect Collision* asupra zonei în care se intersectează dreptunghiurile. Descrierea în pseudocod a algoritmului dreptunghiurilor este următoarea:

Date de intrare: dreptunghiurile A și B

Dacă $A.x + A.lățime \geq B.x$ și

$A.x \leq B.x + B.lățime$ și

$A.y + A.înălțime \geq B.y$ și

$A.y \leq B.y + B.înălțime$

atunci

Avem coliziune.

Algoritmul *Pixel Perfect Collision* poate fi descris în felul următor: pentru fiecare pixel al intersecției celor două dreptunghiuri se verifică dacă pixelii desenați la acele coordonate, corespunzători celor două texturi, sunt transparenți. Dacă nu sunt atunci avem o coliziune. Acesta este algoritmul *Pixel Perfect Collision* implementat în jocul *N.A.S.S.* și are o complexitate de $O(n^2)$ în cazul în care texturile sunt complet transparente.

Algoritmul *Pixel Perfect Collision* poate fi observat în modul de depanare care se poate activa din meniul principal, opțiunea *Debug Mode*, și apoi apăsarea butonului **R**. Dreptunghiurile care încadrează jucătorul și asteroidul sunt desenate cu un alb semitransparent, intersecția lor cu negru semitransparent, iar primul pixel unde se găsește o coliziune este indicat de către capătul unei linii albe ce pornește din colțul din dreapta jos al ecranului (vezi figura 5.7).

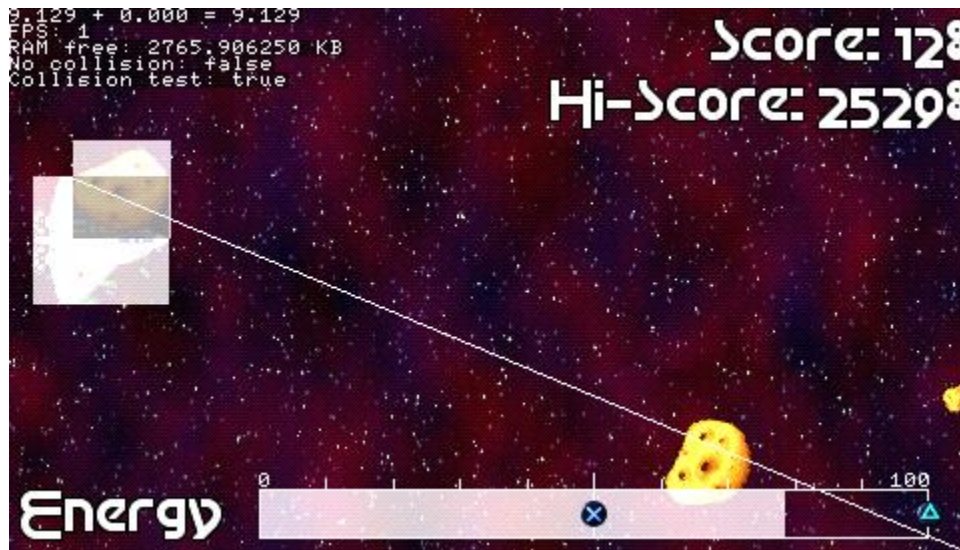


Fig. 5.7. Reprezentarea în joc a calculelor efectuate de
algoritmul de coliziune

6. CONCLUZII

Dezvoltarea unui joc necesită cunoștințe din mai multe domenii ale informaticii și matematicii, precum structuri de date, algoritmică, inginerie software, geometrie etc. În final, pentru acest joc s-a dorit o implementare originală a modulelor, folosind conceptele unei arhitecturi întâlnite și multe dintre produsele studiourilor de dezvoltare de jocuri consacrate (*Ubisoft, Naughty Dog, SCE Santa Monica Studio, Riot Games*, etc.).

Datorită arhitecturii modulare a aplicației, aceasta este ușor de extins cu noi caracteristici. Noi abilități, mai multe tipuri de inamici sau un motor de inteligență artificială pot fi adăugate fără prea multe modificări aduse codului deja existent. Jocul va fi extins în viitor în mai multe direcții: efecte vizuale, conectivitate cu rețele de socializare și optimizare.

Pentru efecte vizuale mai interesante se poate implementa un sistem de particule. Astfel, se pot crea animații mai interesante pentru explozii sau se pot adăuga prelungiri asteroizilor pentru transformarea acestora în comete. Conectivitatea cu o rețea de socializare precum *Facebook* ar permite sincronizarea salvării jocului cu alte dispozitive pe care ar putea fi portat, cumpărarea de puncte de abilități și oferirea acestora drept cadou prietenilor sau primirea și lansarea de provocări de a depăși scorurile acestora.

Hardware-ul consolei *PlayStation Portable*, deși nu mai este de actualitate, are în continuare performanțe surprinzătoare. Dezactivarea coliziunilor în modul de depanare va determina la un moment dat apariția unui asteroid nou la randarea fiecărui cadru, deoarece șansa ca așa ceva să se întâmple variază în funcție de viteza de bază a inamicilor. Chiar și sub aceste condiții de stres, în care sunt randate atât de multe obiecte și verificarea coliziunii lor cu nava (prin dezactivarea coliziunilor, funcția responsabilă de acest lucru returnează fals tot timpul, însă algoritmul încă rulează în fundal), aplicația rulează constant la 60 de cadre pe secundă. Randarea unui cadru durează în acele condiții aproximativ 10,5 milisecunde, de unde deducem că fără limitarea vitezei cadrelor, jocul ar putea rula cu ușurință la 95 de cadre pe secundă.

Optimizări încă pot fi făcute, spre exemplu la memoria folosită de resursele jocului. În situația de față, dimensiunea redusă a resurselor a permis încărcarea lor în întregime în RAM, fără a fi necesare dealocări dinamice de memorie când anumite resurse nu mai sunt folosite. În schimb, când aplicația va fi extinsă, ar putea apărea probleme serioase de memorie. Implementarea unui sistem care să gestioneze încărcarea resurselor dinamic într-un alt fir de execuție și redarea fișierelor audio direct de pe cardul de memorie vor rezolva aceste probleme.

BIBLIOGRAFIE

1. *PlayStation Portable*, Wikipedia, 13 iunie 2013
(http://en.wikipedia.org/wiki/PlayStation_Portable)
2. *Totul despre C și C++*, Dr. Kris Jamsa, Lars Klander, Ed. Teora, 2004, ISBN 973-601-911-X, 1328 pag.
3. *C++*, Danny Kalev, Michael J. Tobler, Jan Walter, Ed. Teora, 2000, ISBN 973-20-0429-0, 496 pag.
4. *C++ Introducere în Standard Template Library*, Constantin Gălățan, Ed. ALL, 2008, ISBN 978-973-571-798-8, 322 pag.
5. *Software development kit*, Wikipedia, 13 iunie 2013
(http://en.wikipedia.org/wiki/Software_development_kit)
6. *PSPSDK*, GitHub, 13 iunie 2013
(<https://github.com/pspdev/pspsdk#readme>)
7. *Wrapper library*, Wikipedia, 13 iunie 2013
(http://en.wikipedia.org/wiki/Wrapper_library)
8. *GNU Debugger*, Wikipedia, 13 iunie 2013
(http://en.wikipedia.org/wiki/GNU_Debugger)
9. *PSP Homebrew Game Development with Eclipse*, Paulo Lopes, 13 iunie 2010
(<http://www.jetdrone.com/static/articles/pspeclipse.pdf>)
10. *Advanced 2D Game Development*, Jonathan S. Harbour, Ed. Course Technology, 2009, ISBN 978-1-59863-342-9, 330 pag.
11. *Game Engine Architecture*, Jason Gregory, Ed. A K Peters, 2009, ISBN 978-1-4398-6526-2, 853 pag.
12. *Game Coding Complete, Fourth Edition*, Mike McShaffry, David Graham, Ed. Course Technology, 2013, ISBN 978-1-133-77657-4, 959 pag.
13. *Game Development Essentials An Introduction Third Edition*, Jeannie Novak, Ed. Delmar Cengage Learning, 2012, ISBN 978-1-1113-0765-3, 514 pag.