GDROMag™

Romanian Game Developer's Magazine

Issue #01, July 2010 www.gamedev.ro

Articole din acest numar:

- Finite state machines
- Tehnici de partitionare spatiala
- UX (user experience) design in dezvoltarea de jocuri
- Abstractizarea sistemelor unui motor de joc, plugin system

Cuvant de inainte si inapoi

Dragi tovarasi cititori si cititoare,
Bine v-am gasit, in aceasta prima editie a e-zine-ului GDROMag speram sa gasiti cateva lucruri interesante de citit si aprofundat, incercam sa va trezim interesul, nu neaparat sa prezentam articole academice, prin acest mic experiment si efort avem in gand sa materializam cumva peisajul creatorilor de jocuri din Romania, atat ***indie*** cat si angajati la firme de jocuri autohtone. Mai mult ca sigur ca articolele prezentate aici le gasiti pe Internet fara probleme, insa in viitor o sa incercam sa avem articole cat mai specifice scenei romanesti, cu interviuri, galerii ale unor artisti din industrie, articole mai elaborate si mai complexe. Pana atunci, aveti in fata editia princeps! Enjoy!

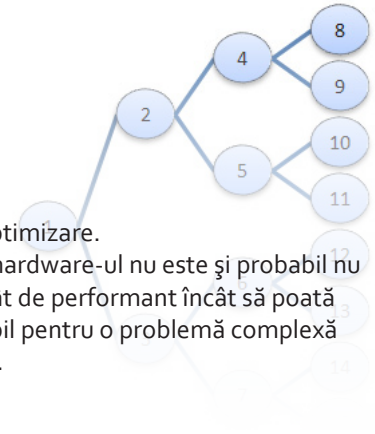
The GDRO Staff

http://www.gamedev.ro

10 July 2010
1:43 AM

Adrian Manolache

ARBORI SPAȚIALI BINARI



MOTIVAȚIE

Calculatorul modern (bazat pe arhitectura lui ***John von Neumann***), gestionează cel puțin două resurse: memorie şi putere de procesare. Scopul principal al maşinii este de a face calcule, pe baza memoriei, folosind puterea de procesare disponibilă, efectuând comunicații între două componente complementare si inutilizabile separat: componenta hardware şi cea software. Memoria, element central în arhitectura von Neumann, specifică circuitelor secvențiale, aduce cu sine o noțiune remarcabilă: timpul. Timpul este poate cea mai de preț resursă în funcționarea unui sistem informatic în timp real.

Pentru că cele trei resurse amintite sunt prezente în cantități limitate dar şi datorită complexității problemelor cerute spre rezolvare s-a cultivat noțiunea de optimizare. Astfel s-a apelat la metode matematice care să modeleze cu rigoare acest concept: din punct de vedere matematic optimizarea se ocupă cu minimizarea sau maximizarea unei funcții variind argumentele sale. Din perspectiva informatică lucrurile stau similar: având la dispoziție o instanță a unei probleme, fie ea **p**, definim o funcție numită:

$$cost : P \rightarrow R,$$

(unde **P** reprezintă mulțimea tuturor instanțelor problemei), care exprimă cât de mare consumatoare de resurse este rezolvarea lui **p**. De obicei, se va dori minimizarea funcției cost. Această abordare simplistă, este ilustrativă pentru conceptul de optimizare (în practică se doreşte determinarea consumului de resurse caracteristic tuturor instanțelor unei probleme găsind funcții matematice având ca argumente proprietăți cantitative ale datelor de intrare). Deşi poate mulți dintre noi se sperie de teoretizarea si abstractizarea conceptelor folosind ca unelte de modelare ştiințe ca, matematica, este de ținut minte că tocmai aceste unelte ne ajută să înțelegem profund şi să găsim caracteristici mai puțin intuitive ale problemelor care ne sunt puse.

Deseori, programatorii începători au senzația că nu trebuie să adopte o poziție fermă față de optimizare, pentru că oricum hardware-ul devine din ce în ce mai bun. În acest context, legile lui Moore descriu un trend de lungă durată în istoria hardware-ului computațional şi anume faptul că numărul de tranzistori ce pot fi aşezați pe un circuit integrat se dublează la aproximativ 2 ani (odată cu aproape toate caracteristicile calculatorului: memorie, putere de procesare). Aceste legi nu trebuie confundate cu *„la fiecare aproximativ 2 ani pot scrie programe de două ori mai ineficiente fără a se observa scăderi de performanță"*. Odată cu dezvoltarea hardware-ului, software-ul este împins până la limite şi invers: odată ce software-ul se dezvoltă, împinge utilizarea hardware-ului la extrem. Astfel este nevoie atât de hardware eficient dar mai ales, de software performant, iar performanța este dată de gradul de optimizare.

Cu alte cuvinte hardware-ul nu este şi probabil nu va fi pentru ceva timp atât de performant încât să poată da soluții în timp rezonabil pentru o problemă complexă implementată neeficient.

OPTIMIZARE

În virtutea introducerii, se va discuta în continuare o structură de accelerare care facilitează efectuarea unor operații specifice într-o manieră eficientă, cu alte cuvinte: optimizat. Este de menționat că optimizarea acționează în principal asupra minimizării efortului calculatorului în a rezolva o problemă. Avem nevoie astfel să măsurăm într-un fel acest efort. Putem să calculăm numărul de adunări, de scăderi, apeluri de funcții sau chiar cicli de procesor (cât de uşor ?!) însă una din cele mai utilizate metode de analiză a codului este analiza asimptotică a complexității. Vom privi astfel codul ca un mare algoritm, adică o secvență finită de paşi înzestrată cu logică, având un scop bine definit de a rezolva o problemă. Desigur, putem defini o problemă ca fiind un cuplu format din două mulțimi: *intrarea şi ieşirea*. Un algoritm rezolvă o problemă corect dacă furnizează o ieşire corectă pentru intrarea furnizată (atenție: pentru o intrare pot exista mai multe ieşiri corecte).

Astfel, algoritmizarea rezolvării unei probleme (adică găsirea unui algoritm care rezolvă problema) este o cerință fundamentală în găsiriea soluției sale cu ajutorul calculatorului(de remarcat ca există probleme pentru care nu se cunoaşte algoritm de rezolvare, ele numindu-se nedecidabile). Analiza asimptotică se referă la analiza ratei de creştere a timpului de execuție a unui algoritm pe măsura ce dimensiunea datelor de intrare creşte la infinit. Se va găsi deci pentru un algoritm, o funcție care să descrie rata de creştere a timpului în funcție de proprietăți cantitative şi variabile ale setului de date. Exemplu:

*Algoritmul de **ridicare la putere***:

```
subprogram calculează( a, n )
      putere_a_la_n = 1; // 1 operatie
      pentru i = 1 pana la n
      // 2 operatii (initializare sau incre-
mentare si comparare cu n)
      puterea_a_la_n *= a; // 1 operatie
      returnează putere_a_la_n;  // 1 operatie
sfârşit subprogram
```

Timpul său de executie este descris de funcția **f(n) = 1 + 3 * n - 1 + 1 = 1+3*n**. Rata sa de creştere va fi dată de **T(n)** unde, **T(n) = n**. De remarcat că analiza asimptotică compară algoritmii independent de numărul de instrucțiuni specifice unui limbaj (care pot varia deci în funcție de limbajul de programare şi stilul programatorului) şi de puterea de procesare a unui calculator. Mai rămâne un singur lucru de lămurit: de ce se numeşte analiza asimptotică ? Pentru că lui **T(n)** îi va

fi asociată o clasă de funcţii care se apropie din ce în ce mai mult de **f(n)** cu cât **n** tinde la infinit. Astfel vom spune în cazul nostru că **T(n)** apartine lui **O(n),** unde **O(n)** clasa de funcţii (a algoritmilor liniari) care sunt mărginite superior de **f(n)**. În practică se foloseşte deseori semnul de egalitate între **T(n)** şi **O(n)**.

Analiza asimptotică este utilă pentru că cele mai mari optimizări se pot face la nivelul complexităţii studiate astfel. A doua mare optimizare are loc la nivelul implementării algoritmului şi în final al treilea tip de optimizare este cea de nivel jos (eng. low-level). În cazul problemei de faţă se poate obţine acelaşi rezultat calculând funcţia recursivă:

$$putere(a,n) = \begin{cases} 1, dacă\ n = 0 \\ putere\left(a, \frac{n}{2}\right) \cdot putere\left(a, \frac{n}{2}\right), dacă\ n\ este\ par \\ putere\left(a, 1 + \left[\frac{n}{2}\right]\right) \cdot putere\left(a, \left[\frac{n}{2}\right]\right), dacă\ n\ este\ impar \end{cases}$$

T(n) pentru acest algoritm va fi log *n* (utilizând memoizarea), o îmbunătăţire majoră faţă de algoritmul liniar. Aceste noţiuni se vor dovedi utile la analiza complexităţii algoritmilor ce vor fi discutaţi în cele ce urmează.

PARTIŢIONAREA SPAŢIALĂ

Mediile 3D complexe sunt formate dintr-un număr impresionant de obiecte. Pe de altă parte hardware-ul 3D specializat fie el de ultimă generaţie, sau nu, poate gestiona un număr finit de astfel de obiecte. Funcţia principală a unui procesor grafic este de rasteriza. În acest sens, pentru a elimina obiectele pentru care procesarea ar fi redundantă avem nevoie să determinăm doar pe acelea care ar contribui prin rasterizare la imaginea finală şi să le eliminăm pe cele care sunt ascunse de geometria altora. În această situaţie metodele intuitive/simple sunt foarte lente.

Problema aceasta este doar una din cele pe care le rezolvă partiţionarea spaţială. Un al doilea exemplu este specific simulărilor fizice. Avem la dispoziţie **N^2** obiecte ce interacţionează conform legilor fizicii si vrem să determinăm toate interacţiunile dintre obiecte la momentul actual şi să acţionăm corect pentru a le rezolva. Putem aborda situaţia testând fiecare obiect cu fiecare pentru a determina dacă se intersectează în vreun fel sau nu. În acest caz complexitatea este de ordinul lui **N^2**. Cu siguranţă se poate şi mai bine decât atât. Intuitiv, putem partiţiona spaţiul folosind un grid având **L x C** celule. Vom insera toate obiectele în acest grid, în celula corespunzătoare poziţiei obiectului. Pentru a determina toate obiectele pe care le poate intersecta obiectul curent vom interoga doar celulele apropiate acestuia.

În cele ce urmează vom vedea că o structură ierarhică de partiţionare a unui mediu 3D permite rezolvarea eficientă a mai multor tipuri probleme, cele mai importante fiind: determinarea aproximativă sau exactă a suprafeţelor vizibile, detecţia coliziunilor, modelarea solidelor.

ARBORII SPAŢIALI BINARI

Arborii spaţiali binari sunt structuri ierarhice de partiţionare şi clasificare ce au rolul de a împărţi şi sorta geometria unei scene de la intrare. Pentru a construi un arbore binar spaţial procedăm astfel:

1. Selectăm un plan de partitionare
2. Împărţim setul de poligoane curent după acest plan
3. Trimitem cele două seturi de poligoane rezultate în subarborii stâng si drept

Procedeul este foarte simplu dar rămân câteva aspecte neclarificate: condiţia de oprire şi modalitatea în care alegem planul de partiţionare. Adevărul este că există atâtea tipuri de arbori binari spaţiali, câte moduri de a selecta planul de partiţionare şi condiţii de oprire la care ne-am putea gândi găsim. Însă de interes foarte mare sunt arborii binari spaţiali construiţi prin autopartiţionare, adică cei pentru a căror construcţie se folosesc planele suport ale poligoanelor de la intrare.

DETECŢIA VIZIBILITĂŢII

Sortarea poligoanelor pentru rasterizarea fără *Z-Buffer* poate fi rezolvată de unul din cei mai simpli arbori binari spaţiali. Această problemă mai apare şi sub denumirea de eliminarea suprafeţelor ascunse (eng. *Hidden Surface Removal*) sau detecţia suprafeţelor vibile (eng. *Visible Surface Detection*). Cele două motivaţii ale existenţei unui asemenea algoritm sunt: corectitudine (dacă suprafaţa **A** se află în spatele supfafeţei **B**, atunci **A** nu ar trebui să fie vizibilă) şi viteză (dacă suprafaţa **A** nu este vizibilă nu ar trebui irosit timp pentru a o procesa).

Descrierea problemei: la intrare avem **n** poligoane şi desigur, poziţia şi orientarea observatorului, iar ieşirea va consta în găsirea unei ordini în care va trebui să rasterizăm aceste poligoane astfel încât să determinăm corect suprafeţele vizibile şi să le eliminăm pe cele ascunse. Probabil una din metodele intuitive care ne vin în minte este să efectuăm o sortare a poligoanelor în funcţie de adâncimea (distanţa) lor faţă de observator (prin distanţa între un poligon şi un punct **P** ne putem referi la distanţa între centrul poligonului şi **P**). Această abordare, denumită "*algoritmul pictorului*" (eng. *Painter's Algorithm*), eşuează în cazul suprapunerii ciclice:

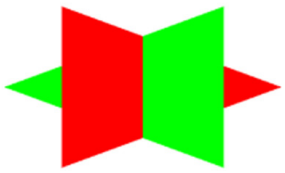


Figura 0.1 – Imposibil de sortat

Deşi un astfel de aranjament este rezonabil dacă folosim un buffer de adâncime problema ar fi rezolvată doar pe jumătate. Dacă cele doua triunghiuri ar fi transparente şi s-ar folosi o funcţie de combinare a culorilor necomutativă nici măcar buffer-ul de adâncime nu ar fi de ajuns. Un alt lucru care face acest mod de abordare impractic este timpul foarte mare pe care îl necesită sortarea. Precalcularea nu ne-ar ajuta prea mult pentru că poziţia observatorului în scena este arbitrară. Un arbore binar spaţial este capabil să rezolve problema sortării poligoanelor de la cel mai depărtat la cel mai apropiat, faţă de observator, într-un timp foarte scurt având ca intrare orice poziţie posibilă şi validă (în cazul arborilor binari spaţiali cu noduri solide), a observatorului. Timpul este atât de mic

încât scena poate fi sortată de câteva sute sau mii de ori pe secundă din poziții diferite ale observatorului.

Modul de funcționare al unui arbore binar spațial

Fie următorul aranjament de poligoane:

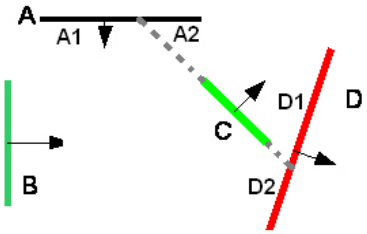


Figura 0.2 – scenă de test

Exemplul reprezintă o trivializare a unei scene tridimensionale. Se presupune ca poligoanele sunt privite de sus (proiecție paralelă ortografică verticală). Săgețile indică sensul normalei poligonului.

Poligonul **C** creează o problemă în construcția arborelui spațial. Pentru că într-un arbore binar spațial toate poligoanele sunt complet în fața ori complet în spatele oricărui alt poligon sau, pe același plan, nu putem construi arborele fără sa facem niște modificări datelor de mai sus. Modificarea pe care o vom face este sa tăiem poligoanele **A** și **D**, după planul poligonului **C**. Astfel poligoanele rezultate vor fi ori complet în față, în spate sau pe același plan cu **C**. De remarcat ca această modificare nu se va face la inițializare, tăierea poligoanelor având loc în timpul construcției arborelui, de fiecare dată când va fi nevoie. Va trebui astfel sa găsim un echilibru între numărul de poligoane noi introduse și adâncimea arborelui, ceea ce înseamnă ca dintr-un anumit set de date se pot obține mai mulți arbori binari spațiali perfect valizi.

CONSTRUCȚIE

În continuare vom construi arborele binar autopartiționat pentru setul de poligoane din *Figura 0.2*. Vom presupune deocamdată că alegerea planului de partiționare se face aleator. Ne vom opri din procesul recursiv atunci când vom fi folosit planul suport al fiecărui poligon ca plan de partiționare iar în timpul construcției următoarea convenție va fi utilizată: fiul stâng din fiecare subarbore memorează poligoanele din spatele planului corespunzător rădacinii subarborelui şi fiul drept pe cele clasificate ca fiind în față.

Începem prin alegerea lui **C** ca **plan de partiționare**: acesta va tăia poligoanele **A** şi **D** şi va apela procedeul recursiv de creare pentru listele de poligoane **A1, B, D2** şi **A2, D1**. Discutăm întâi lista **A1, B, D2** (*Figura 0.4*). Alegem din nou aleator ca plan de partiționare pe cel al lui **B**. În spatele lui **B** nu se află niciun poligon (din lista poligoanelor din care face parte) deci fiul stâng va fi setat pe **NULL**, indicând acest lucru. În fiul drept al subarborelui cu rădăcina în **A1, B, D2** vor ajunge **A1** şi **D2**.

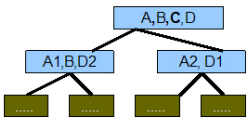


Figura 0.3 – pas 1

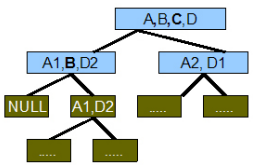


Figura 0.4 – pas 2

La *pasul 4* am evidențiat trimiterea lui **D1** în fiul drept al subarborelui cu rădăcina **A2, D1,** alegând ca splitter pe **A2**. În spatele lui **A2** nu se află niciun poligon deci fiul său stâng va fi setat pe **NULL**.

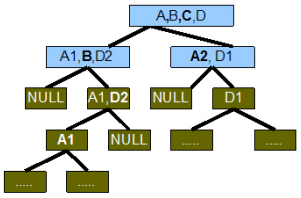


Figura 0.5 – pas 3

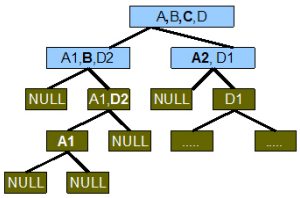


Figura 0.6 – pas 4

În final arborele devine următorul:

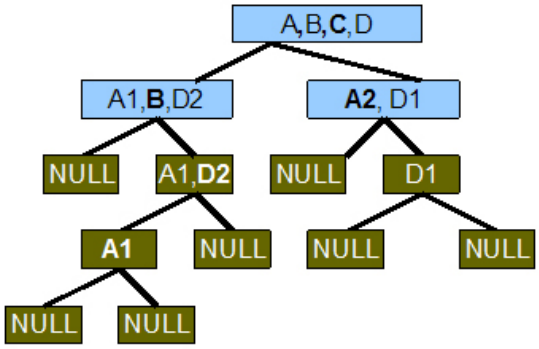


Figura 0.6 – Arborele Binar Spațial complet

De reținut că în nodurile interne ale arborelui vor fi memorate doar poligoanele îngroşate, celelalte poligoane doar „*trec*" prin acel nod şi se vor opri mai jos în ierarhie.

Următorul pseudocod construiește un arbore spațial simplu dar suficient de complex încât să ilustreze modul general de creare al arborilor binari spațiali. Poligoanele vor fi reținute în nodurile interne, iar cele ale căror plan va fi folosit pentru partiționare nu vor fi trimise mai departe în fiul drept sau stâng ci se vor opri în nodul curent. Pornind de la acest exemplu vom rafina ulterior modul de construcție.

Algoritm ***construcție arbore binar spatial simplu***:

```
subprogram BSPTSimplu( node*& nodCurent, lista poligoane )
dacă lista_vida( poligoane )
    nodCurent = NULL;
altfel
    nodCurent =  new node;
    nodCurent->lafel = 0;
    nodCurent->opus = 0;
    lista_spate = 0;
    lista_fata = 0;
    polygon ps = allege_splitter(poligoane);

    pentru fiecare p != ps din poligoane
        cod = clasifica_poligon(ps, p);
        dacă cod = IN_SPATE atunci
            lista_spate = lista_spate U p;
        altfel dacă cod = IN_FATA atunci
            lista_fata = lista_fata U p;
        altfel dacă cod = INTERSECTEAZA atunci
            taie_poligon(ps, p, spate, fata)
            lista_spate = lista_spate U spate;
            lista_fata = lista_fata U p;
        altfel  atunci // p şi ps sunt pe acelaşi plan
            dacă p are aceeaşi direcție ca ps atunci
```

```
            nodCurent->lafel = nodCurent->lafel U p;
            altfel atunci
               nodCurent->opus = nodCurent->opus U p;
            sfârșit dacă
         sfârșit dacă
      sfârșit pentru
   sfârșit dacă
   BSPTSimplu(nodCurent->stânga,   lista_spate);
   BSPTSimplu(nodCurent->dreapta, lista_fata);
sfârșit subprogram
```

COMPLEXITATEA CONSTRUCȚIEI ARBORELUI

Complexitatea funcției este **O(N^2)**. Cei curioşi să ştie cum am ajuns la această concluzie pot citi în continuare, ceilalti poti sări peste fără să-şi facă griji.

Complexitatea funcției depinde în bună măsură de topologia scenei ce este partiționată. Astfel, deși la fiecare apel alegem un plan de partiționare ce obține cel mai bun rezultat al funcției de optimizare nu putem ști dacă acesta va împărți scena în două părți egale ca volum și nici că numărul poligoanelor din spate este egal cu numărul poligoanelor din față. Pentru a ușura calculul vom presupune că cel mai bun plan împarte scena în cantități aproximativ egale, din punct de vedere al poligoanelor. Apelul **alege_splitter** are complexitatea de ordinul lui **O(N^2)**, unde **N** este dimensiunea listei de poligoane, de la intrare. Operația de găsire a listelor din spatele splitter-ului și din fața acestuia are complexitatea **O(N)**. Obținem următoarea recurență:

$$T(N) = 2T\left(\frac{N}{2}\right) + N^2$$

O vom aborda prin metoda iterației (deşi se poate aplica si teorema lui Masters):

$$T(N) = N^2 + 2T\left(\frac{N}{2}\right) = N^2 + 2\left(\left(\frac{N}{2}\right)^2 + 2T\left(\frac{N}{4}\right)\right) = \frac{N^2}{2^0} + \frac{N^2}{2^1} + 2T\left(\frac{N}{2^2}\right) = \frac{N^2}{2^0} + \frac{N^2}{2^1} + \cdots + \frac{N^2}{2^k} + 2^k T\left(\frac{N}{2^k}\right) = 2^k T\left(\frac{N}{2^k}\right) + \sum_{j=0}^{k} \frac{N^2}{2^j} = 2^k T\left(\frac{N}{2^k}\right) + N^2 \sum_{j=0}^{k} \frac{1}{2^j}.$$

Concluzionăm astfel complexitatea funcției.

Subprograme utilitare

Am folosit în algoritmul de mai sus o serie de funcții utilitare. În continuare le vom discuta pe rând începând cu funcția alege_splitter . Aceasta funcție implementează o euristică de alegere a poligonului după planul căruia se va face clasificarea celorlalte poligoane din listă, astfel încât sa fie optimizate două aspecte: adâncimea(echilibrarea arborelui) și numărul de poligoane noi, introduse față de setul inițial de date. O posibilă implementare a acestei funcții este următoarea:

*Algoritm **alege splitter***

```
Poligon alege_splitter(lista poligoane)
    Poligon best;
    scor_maxim = ∞;
    pentru fiecare p din poligoane
       spate = 0;
       fata = 0;
       intersecteaza = 0;

    pentru fiecare g != p din poligoane
       cod = clasifica_poligon(p, g);
       dacă cod = IN_SPATE atunci
          spate = spate + 1;
       altfel dacă cod = IN_FATA atunci
          fata = fata + 1;
       altfel dacă cod = INTERSECTEZA atunci
          intersecteaza = intersecteaza + 1;
       sfârșit dacă
    sfârșit pentru
    dacă scor_maxim > abs(spate - fata) + intersectează*3
       atunci
          scor_maxim = abs(spate - fata) + intersectează*3
          best = p;
       sfârșit dacă
    sfârșit pentru
    return best;
sfârșit subprogram
```

Pentru minimizarea adâncimii arborelui și a numărului de poligoane noi introduse (prin procesul de tăiere) am folosit funcția

```
f(in_spate,in_fata,intersecteaza) = abs(in_spate - in_
fata) + intersecteaza * 3
```

unde **în_spate** reprezintă numărul de poligoane complet în spatele planului poligonului candidat pentru partiționare, **în_față** reprezintă numărul de poligoane complet în fața acestuia și intersectează, numărul poligoanelor care intersectează planul.

Este evident de ce am vrea sa minimizăm numărul de poligoane noi introduse: acestea necesita timp în plus pentru rasterizare dar și pentru detecția coliziunilor. Adâncimea și echilibrarea arborelui sunt importante deoarece, pentru a obține un număr de cadre pe secundă puțin variabil, avem nevoie ca parcurgerea arborelui să consume un timp nu mult variabil pentru fiecare poziție în care s-ar afla observatorul. Astfel nu am avea cazuri când framerateul ar scădea sau creşte considerabil dacă ne-am afla în fața sau spatele unui aceluiași poligon. Prima parte din funcția de optimizare (*abs(in_spate - in_fata)*) este destinată minimizării diferenței între adâncimile sub-arborilor nodului curent iar a doua parte (*intersecteaza * 3*) se ocupă de minimizarea numărului de poligoane noi introduse.

Funcția **clasifică_poligon** e printre cele mai simple: dacă toate punctele poligonului de testat se află în spatele planului poligonului dat la intrare se returnează *ÎN_SPATE*, dacă toate punctele poligonului se află în fața planului se returnează *ÎN_FAȚĂ*, dacă unele puncte se află în spate şi altele în față se returnează *INTERSECTEAZĂ* iar altfel se va returna *IDENTIC*, cu semnificația că planele poligonului de testat și planul după care se face împărțirea sunt identice cu o anumită marjă de eroare.

COMPLEXITATEA FUNCȚIEI alege_splitter

Complexitatea funcției **clasifică_poligon** este de ordinul lui **O(N)** unde *N = numărul de puncte al poligonului care se testează*. Având discutată această funcție putem calcula complexitatea funcției **alege_splitter**. Există două bucle pentru iar în bucla cea mai interioară un apel la funcția **clasifica_poligon**. Rezultă că avem o complexitate de ordinul lui **O(M2*N)** unde *M = numărul de poligoane din listă* (cele două bucle *pentru*). În cazuri foarte rare **N** va fi mai mare ca **10**. Putem să-l considerăm pe **N** constant și astfel am avea o complexitate de ordinul lui **O(M2)**.

FUNCȚIA taie_poligon

Pentru că funcția **taie_poligon** este una din cele mai importante și este folosită de aproape orice sistem de partiționare spațială vom discuta în continuare algoritmul folosit.

Pentru a tăia un poligon ne vom folosi de o altă funcție utilitară care va returna intersecția unei linii cu un plan. Algoritmul de tăiere a poligonului va returna *2 poligoane*, unul pentru fiecare din cele *2 spatii*, delimitate de plan, fiecare poligon fiind specificat de puncte în ordine trigonometrică. Vom numi aceste spații **p+** și **p-**. Vom parcurge punctele poligonului și vom clasifica punctul curent și pe cel următor în funcție de plan. Astfel vom ști dacă cumva această muchie intersectează planul sau nu. Dacă există o intersecție o calculăm și adăugăm acest punct de intersecție atât la **p+** cât și la **p-**, altfel adăugăm punctul curent doar la **p-** sau **p+**, în funcție de semiplanul în care se află. Următorul desen ilustrează un triunghi ce trebuie tăiat după un plan arbitrar:

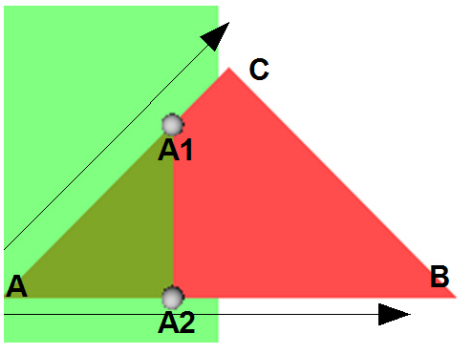


Figura 0.7

Algoritmul de tăiere a poligonului va parcurge punctele în ordinea **A,B,C,A** și va construi incremental cele două poligoane (le vom nota cu **p-** și **p+**) corespunzătoare semiplanului pozitiv și negativ. Parcurgând segmentul de la **A** la **B** vom găsi o intersecție cu planul în punctul **A2**. Vom adaugă la **p-** pe **A** și pe **A2**, iar la **p+** doar pe **A2**. Punctul **B** va fi adăugat la **p+**, la fel și **C**. Segmentul **CA** intersectează planul în punctul **A1** care va fi adăugat atât la **p-** cât și la **p+**. Obținem următoarea lista de puncte:

p- : A, A2, A1
p+ : A2, B, C, A1

De remarcat ca algoritmul funcționează în asa fel încât păstrează ordinea trigonometrică a punctelor, caracteristică poligonului de tăiat, la poligoanele rezultate. Astfel nu este nevoie să aplicăm algoritmi de sortare convexă a punctelor după ce am tăiat un poligon.

Pentru a calcula punctul de intersecție dintre un segment și un plan vom proceda ca în cele ce urmează. Definim planul că

$plan(x,y,z) = ax+by+cz+d$

sau

$plan(n) = p \cdot n + d$

unde ***n*** reprezintă normala planului și ***d*** este distanța până la origine în direcția normalei. Un segment este format din două puncte. Vom rescrie ecuația dreptei suport a acestuia sub formă parametrică:

$d(t) = s + (e - s) \cdot t$

unde cu ***s*** și ***e*** am notat extremitățile segmentului. Variind ***t*** între ***0*** și ***1*** vom obține orice punct de pe segment. Știm că punctul de intersecție verifică atât ecuația dreptei suport cât și pe cea a planului, deci avem:

$plan(d(t)) = d(t) \cdot n + d = 0 \rightarrow$
$(s + (e - s) t) \cdot n + d = 0 \rightarrow$
$s \cdot p + (e - s) \cdot pt + d = 0 \rightarrow$
$t = -\frac{s \cdot n + d}{(e - s) \cdot n}$

unde operația „·" reprezintă produsul scalar. Dacă **0 ≤ t ≤ 1** atunci ştim sigur că punctul de intersecție cu planul se va afla pe segment şi va fi dat de **d(t)**. Altfel punctul de intersecție se află undeva pe dreapta suport și considerăm ca segmentul nu a intersectat planul. Mai trebuie avut grijă la cazul când segmentul este paralel cu planul adică dreapta suport este perpendiculară cu normala. Putem determina ușor dacă ne aflăm în aceasta situație verificând dacă:

$(e-s) \cdot n = 0$

Pe lângă calcularea punctului de intersecție (fie acesta ***p***) putem folosi parametrul ***t*** calculat ca mai sus și pentru interpolarea liniară a altor atribute ale lui ***p***, de-a lungul muchiei, cum ar fi coordonatele de textură și normala. Pentru calcularea coordonatelor de textură se procedează astfel: avem $t1 \in R^2$ și $t2 \in R^2$, coordonatele de textură ale segmentului pe care se p află (fie $pt \in R^2$ coordonatele sale de textura). Avem: $pt=t1+(t2-t1)*t$. La fel de simplu se calculează și normala lui p. Poate nu la fel de evident este de ce acest mod de găsire a atributelor este corect. Răspunsul vine de la modul în care funcționează hardware-ul 3D modern. Deși ar fi putut folosi funcții de interpolare mai complicate hardware-ul specializat interpolează toate atributele unui punct (vertex) liniar pentru rapiditate. Astfel nu am făcut decât sa procedăm la fel cum procedează și hardware-ul și deci rezultatele obținute vor furniza rasterizări corecte din punct de vedere vizual. În continuare este prezentat pseudocodul funcției de tăiere a poligoanelor după planul suport al unui poligon:

*Algoritm **taie poligon***

```
subprogram taie_poligon(Poligon ps, Poligon p, Poligon&
spate, Poligon& fata)
    pentru i de la 0 la p.nrPuncte-1
       urmator = (i + 1) mod p.nrPuncte;
       poz1 = clasifica(ps.plan, p.puncte[i]);
       poz2 = clasifica(ps.plan, p.puncte[urmator]);
       spate.puncte = 0; spate.nrPuncte = 0;
       fata.puncte = 0; fata.nrPuncte = 0;
       dacă (poz1=IN_SPATE și poz2=IN_SPATE) sau
            (poz1=IN_SPATE și poz2=PE_PLAN)
             AdaugăPunct(spate, p.puncte[i]);
       altfel
       dacă (poz1=IN_FATA și poz2=IN_FATA) sau
            (poz1=IN_FATA și poz2=PE_PLAN)
             AdaugăPunct(fata, p.puncte[i]);
       altfel
       dacă (poz1=PE_PLAN și poz2 = IN_FATA) sau
            (poz1=PE_PLAN și poz2=IN_SPATE) sau
            (poz1=PE_PLAN și poz2=PE_PLAN)
             AdaugăPunct(spate, p.puncte[i]);
             AdaugăPunct(fata, p.puncte[i]);
       altfel
       dacă (poz1=IN_SPATE și poz2=IN_FATA ) sau
            (poz2=IN_SPATE și poz1=IN_FATA )
             intersectează(ps.plan, p.puncte[i],
```

```
p.puncte[următor], t);
       pi = p.puncte[i]+(p.puncte[urmator]-p.puncte[i])*t;
       ti  = p.puncte[i].texcoord +  (p.puncte[urmator].
texcoord-p.puncte[i])*t;
       dacă poz1=IN_SPATE
          AdaugăPunct(spate, p.puncte[i]);
       altfel
          AdaugăPunct(fata, p.puncte[i]);
       sfârșit dacă
       AdaugăPunct(spate, p.puncte[i]);
       AdaugăPunct(fata, p.puncte[i]);
    sfârșit dacă
    sfârșit pentru
sfârșit subprogram
```

RASTERIZAREA ARBORELUI

În acest moment avem toate informațiile necesare construirii unui *arbore binar spațial simplu*. Îl putem utiliza pentru a sorta poligoanele de la cel mai depărtat la cel mai apropiat, pentru a rasteriza fără ***Z-Buffer*** sau în cazul necesității unei funcții de combinare a culorilor (eng. blending) necomutativă. Operația de rasterizare a setului de poligoane din spate în față necesită foarte puține linii de cod şi este foarte rapidă, odată construit arborele binar spațial. La intrare avem poziția observatorului iar la ieşire ordinea corectă a poligoanelor. Pseudocodul ar putea arăta astfel:

```
subprogram RenderSimpleBSPTree(node, position)
    daca node == NULL
       return;
    result = clasifica_punct(node->plane, position);
    daca result == IN_FATA
       RenderSimpleBSPTree (node->Back, Position);
       RenderPolygon(node->polygon);
       RenderSimpleBSPTree (node->Front, Position);
    altfel
       RenderSimpleBSPTree (node->Front, Position);
    // dacă setul de date este convex nu e nevoie să
rasterizăm acest poligon
       RenderPolygon(node->polygon);
       RenderSimpleBSPTree (node->Back, Position);
sfarsit subprogram
```

EXEMPLU DE RULARE A SUBPROGRAMULUI DE RASTERIZARE

Presupunem arborele binar spațial pentru setul de date din *Figura o.8*, şi poziția observatorului marcată cu bulina zâmbitoare:

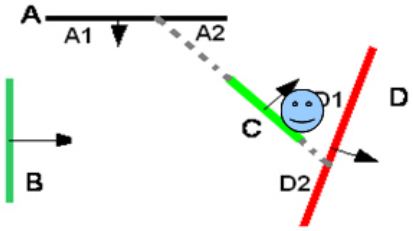


Figura o.8

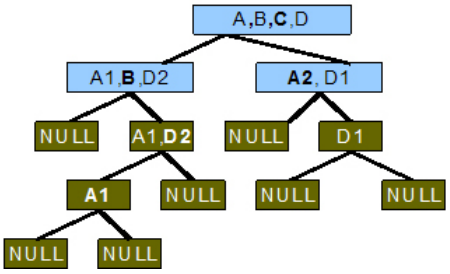


Figura o.9

Funcția ***RenderSimpleBSPTree*** se va apela cu aceeaşi poziție de fiecare dată iar la inceput va primi ca nod de intrare radacina: ***A,B,C,D***. Observatorul se află în acest moment în fața lui ***C*** iar ordinea de rasterizare a poligoanelor va arăta astfel:

```
RenderSimpleBSPTree(A1, B, D2), C, RenderSimpleBSPTree(A2,
D1)
```

Am evidențiat apelurile recursive care vor completa ordinea de rasterizare corectă. În continuare vom discuta subarborele stâng. Poligonul de partiționare este ***B***, observatorul se află în fața acestuia deci vom expanda ca în cazul radacinii:

```
RenderSimpleBSPTree (NULL), B, RenderSimpleBSPTree(A1,
D2), C,
RenderSimpleBSPTree(A2, D1)
```

Expandăm subarborele ***A1, D2***: observatorul se află în spatele lui ***D2*** deci inversăm procedeul de până acum:

```
RenderSimpleBSPTree (NULL), B, RenderSimpleBSPTree(NULL),
D2, RenderSimpleBSPTree(A1), C, RenderSimpleBSPTree(A2,
D1)
```

În acest moment putem face câteva simplificări: este evident că ***RenderSimpleBSPTree (NULL)*** nu va contribui cu niciun poligon iar singurul poligon contribuit de ***RenderSimpleBSPTree(A1)*** va fi ***A1***. Rezultatul este:

```
B, D2,  A1, C, RenderSimpleBSPTree(A2, D1)
```

Pentru expandarea ultimului apel recursiv observăm că observatorul se află în fața lui A2 deci ordinea devine:

```
B, D2,  A1, C, RenderSimpleBSPTree(NULL), A2,
RenderSimpleBSPTree(D1)
```

Iar ordinea finală va fi:

```
B, D2, A1, C, A2, D1
```

O primă observație este că această ordine nu este unică, iată încă o soluție posibilă:

```
D2, A1, B, C, D1, A2
```

A doua observație se referă la complexitatea algoritmului. Acesta are complexitatea **O(N)** – recurența care dă acest ordin fiind: **T(N) = 2T(N/2)**. Pentru cei interesați, demonstrația este următoarea:

$$T(N) = c + 2T\left(\frac{N}{2}\right) = c + 2c + 4T\left(\frac{N}{4}\right)$$
$$= 3c + 4\left(c + 2T\left(\frac{N}{8}\right) = 7c + 2^3T\left(\frac{N}{2^3}\right) = \cdots = (2^k - 1)c + 2^kT(\frac{N}{2^k})\right)$$

În acest moment trebuie să alegem un caz pentru care ştim T(x). Pentru ***k=log$_2$N*** obținem ***T(1)***:

$$T(N) = 2^{\log_2 N}c + 2^{\log_2 N}T\left(\frac{N}{2^{\log_2 N}}\right) = Nc + NT(1)$$

Cum ***T(1) = O(1)*** reiese că într-adevăr complexitatea funcției este de ordinul lui ***O(N)***, unde ***N*** este numărul poligoanelor.

CONCLUZII

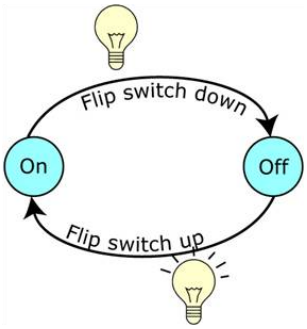
Am parcurs drumul construirii primului arbore spațial. Deşi am discutat multe lucruri nu am vorbit deloc despre cele mai importante tipuri de arbori binari spațiali: *arborele binar cu noduri solide şi arborele binar cu informații în frunze*. Aceşti doi arbori combinați definesc o structură de date ce a fost utilizată intens în jocuri ca ***Doom***, ***Quake***, ***Heretic***, ***Wolfenstein***, ***Return to Castle Wolfenstein*** şi multe altele. Pe baza arborelui binar spațial cu noduri solide şi informații în frunze se pot genera automat portale şi cu ajutorul lor calcula celebrele seturi potențial vizibile (***PVS***). Deşi aceste tehnici nu mai reprezintă state of the art pe PC, ele nu şi-au pierdut utilitatea şi se pot folosi cu succes pe dizpozitive *handheld*, unde grafica nu a evoluat aşa mult (de exemplu *iPhone*).

Quake 3

BSP Level

Liviu Emanuel

FINITE STATE MACHINES



În jocuri, Finite State Machines (FSM) sunt folosite tot timpul. Aproape fiecare loc unde se foloseşte o instrucțiune switch avem o formă de maşină de stare.

Ele pot fi utilizate pentru tot felul de lucruri incluzând, inteligența artificială, contrlolul animației, stările jocului, sistemul de salvare, motoarele de rețea, şi multe altele. În continuare voi vorbi de FSM folosite pentru inteligența artificială în jocuri.

FSM au fost folosite înca de la începuturile jocurilor video în aproape toate jocurile aparute pe piață. Desi au apărut foarte multe alte tehnici pentru implementarea inteligenței, FMS şi-au pastrat un loc important în industria jocurilor video.

Principalele puncte forte ale FSMurilor, care au ajutat-o să supraviețuiască toți aceşti ani, sunt: simplitatea codului, rapiditatea cu care pot fi scrise, uşurinta debuginguLui, folosesc puține resurse, sunt intuitive, sunt flexiile.

Acum că v-am trezit interesul asupra acestui subiect, sau cel putin aşa sper, să definesc ce înseamnă o FSM.

FSM este un model de comportament compus dintr-un număr finit de stări, tranziții între acele stări şi acțiuni. FSM ține minte ultimul lucru pe care l-a făcut (ultima stare) şi evalueaza numai decizia de a părăsi sau nu starea curentă. Dacă o părăseşte, poate apela un cod la ieşire şi un cod la intrarea în altă stare, dar acest lucru nu este obligatoriu.

Sunt mai multe moduri de a implementa o FSM. Unul dintre acestea fiind un switch:

```
switch (CurrentState)
{
        case: EVADE:
                Evade();
                if ( isSafe() )
                        ChangeState(PATROL);
        break;

        case PATROL:
                FollowPath();
                if ( isThreatened() )
                {
                    if ( StrongerThanEnemy() )
                        ChangeState(ATTACK);
                    else
                        ChangeState(EVADE);
                }
        break;

        case ATTACK:
                if ( ! StrongerThanEnemy() )
                    ChangeState(EVADE);
                else
                    EngageEnemy();
                }
        break;
}
```

Acest mod de implementare al unui FSM pare rezonabil, şi probabil mult mai simplu de înteles şi implementat decât metoda pe care o voi explica mai jos. Problema cu acest mod de implementare este că odată cu adăugarea mai multor stări, veți observa faptul că acest cod va arată foarte îmbârligat, greu de citit şi modificat. Din fericire, s-a inventat OOPul, care ne va scapă de toate problemele legate de aranjarea codului.

În primul rând vom organiza stările într-o tabelă numită "Tabelă de tranziție a stărilor" (State transition table). Luând tot exemplul de mai sus, aceasta ar arata astfel:

Stare curentă	Condiție	Tranziția stării
EVADE	isSafe	PATROL
ATTACK	! StrongerThanEnemy	EVADE
PATROL	isThreatened && is-StrongerThanEnemy	ATTACK
PATROL	isThreatened && ! is-StrongerThanEnemy	EVADE

În continuare vom construi 3 clase reutilizabile pentru a folosi FSM.
Prima clasă se va numi IStates şi este o interfață ce defineşte funcțiile ce trebuiesc suprascrise pentru fiecare stare în parte.

```
template <typename state_class>
class IState
{
public:
        virtual ~IState() {}

        virtual void Enter(state_class*) = 0;

// Se executa cand se intra in stare
        virtual void Execute(state_class*) = 0;

// Functie de update a starii
        virtual void Exit(state_class*) = 0;

// Se executa cand se iese din stare

        virtual bool ProcessMessage(state_class*,
const MessageStruct&) = 0;     // Proceseaza un mesaj
primit (daca exista vreunul)
};
```

Următoarea clasă se numeşte CStateMachine şi se foloseşte pentru a updata state-urile:

```
template <class machine_type>
class CStateMachine
{
        machine_type*  m_pOwner;
// Un pointer la clasa care detine aceasta instanta
        IState<machine_type>*   m_pCurrentState;
// Un pointer la starea curenta
        IState<machine_type>*   m_pGlobalState;
// Este chemat de fiecare data cand masina de stare
este updatata, pe cand celelalte stari sunt chemate
numai daca sunt curente

public:
        CStateMachine(machine_type* owner):m_
pOwner(owner), m_pCurrentState(NULL), m_
pGlobalState(NULL) {}
        virtual ~CStateMachine(){}

        IState<machine_type>* GetCurrentState() const
{ return m_pCurrentState; }
        void SetCurrentState(IState<machine_type>*
newState) { m_pCurrentState = newState; }

        IState<machine_type>* GetGlobalState() const
{ return m_pGlobalState; }
        void SetGlobalState(IState<machine_type>*
newState) { m_pGlobalState = newState; }

void Update() const
{
        if(m_pGlobalState)
   // Daca exista o stare globala, atunci ii executam
metoda
                m_pGlobalState->Execute(m_pOwner);

        if (m_pCurrentState)
// Daca exista o stare curenta, atunci ii executam
metoda
                m_pCurrentState->Execute(m_pOwner);
}

bool HandleMessage(const MessageStruct& pMsg)const
{
        if (m_pCurrentState && m_pCurrentState-
>ProcessMessage(m_pOwner, pMsg))
                return true;
        if (m_pGlobalState && m_pGlobalState-
>ProcessMessage(m_pOwner, pMsg))
                return true;
        return false;
}

void  ChangeState(IState<machine_type>* pNewState)
{
        assert(pNewState && "CStateMachine - Chang-
eState: starea care se doreste a fi atribuita este
null");

        m_pCurrentState->Exit(m_pOwner);

        m_pCurrentState = pNewState;
        m_pCurrentState->Enter(m_pOwner);
}

};
```

Ultima clasă este folosită pentru a trimite mesaje de la stari catre alte stări sau clase ce trebuie să fie anunțate că s-a produs un anumit eveniment, pentru a-l trata. Clasa sau starea ce va primi mesajul, va trebui să răspundă la acesta. Acest sistem de mesaje se poate folosi de exemplu pentru a anunta un jucator că s-a trimis o pasă către el, un inamic că trebuie să se fereasta din calea atacului.

Pentru implementarea mesajelor vom folosi o structură (MessageStruct) ce va ține toate datele necesare pentru a identifica mesajul şi cine l-a trimis. De asemenea vom folosi o clasă singleton (CMessageSender) ce va ansambla şi va trimite mesajul către destinatar.

```
struct MessageStruct
{
        IGameObject *pSender;
        // Entitatea ce trimite mesajul
        IGameObject *pReceiver;
        // Entitatea ce primese mesajul
        int                     iMsg;
        // Mesajul identificat printr-un integer
        MessageStruct():pSender(NULL),
pReceiver(NULL), iMsg(-1) {}
        MessageStruct(IGameObject *sender, IGameOb-
ject *receiver, int msg):
        pSender(sender), pReceiver(receiver),
iMsg(msg) {}
};

class CMessageSender
{
private:
        CMessageSender(){}

public:
        static CMessageSender* Instance();
        // Clasa este singleton
        // Trimite un mesaj (iMsg) de la o entitate
(pSender) catre o alta entitate (pReceiver)
        void SendMessage(IGameObject *pSender,
IGameObject *pReceiver, int iMsg)
        {
                MessageStruct strMsg(pSender, pReceiv-
er, iMsg);
                if (!pReceiver->HandleMessage(strMsg))
                // Trimite un mesaj obiectului ce tre-
buie sa il primeasca
                        cout << "Message could not be
handled!";
        }
};
```

Cam acestea ar fi clasele pentru a implementa un FSM. Acum voi explica programul demo pe care l-am scris pentru a va arata un exemplu de FMS.

Este vorba despre un program simplu ce va tipari pe ecran anumite acțiuni făcute de un cavaler şi de scutierul lui. Clasele din program sunt:

- **CKnight.h/cpp** – clasa cavalerului. Aici sunt variabilele ce definesc nevoile lui, Update, aici este definit un opbiect al lui CStateMachine;
- **CKnightStates.h/cpp** – defineşte toate sările cavalerului în clase singleton;
- **CMessage.h/cpp** – trimite mesaje între o stare şi altă stare sau o clasă;
- **CSquire.h/cpp** – clasa scutierului. Aici sunt variabilele ce definesc nevoile lui, Update, aici este definit un opbiect al lui CStateMachine;
- **CSquireStates.h/cpp** – defineşte toate sările scutierului în clase singleton;
- **IGameObject.h** – o interfață ce defineşte un obiect;
- **IState.h** – o interfață ce defineşte o stare;
- **CStateMachine** – folosită pentru a updata stările;
- **main.cpp** – updatează cavalerul şi scutierul de 20 de ori fiecare;

Acțiunile cavalerul vor fi: odihnă, mancat, băut, antrenament şi luptă. Când luptă sau când se antrenează, cavalerul va cere printr-un mesaj armele scutierului. Acțiunile scutierul vor fi: odihnă, mancat, băut şi îi va da armele cavalerului când aceasta i le va cere. Scutierul răspunde la mesajele cavalerului printr-o stare globala CSquireGlobalState. Această va fi în CStateMachine pusă în m_pGlobalState. Starea globală va fi updatată la fiecare update, faată de celelale stări care vor fi updatate numai dacă sunt stare curentă.

Nu uitați să studiați codul sursă şi să îl îmbunătățiți după bunul plac. Dacă aveți întrebări, sugestii, propuneri etc, m-ă puteți contacta la ***liviuemanuel@yahoo.com***.

Bibliografie:

Programming Game AI by Example - Ed. Worldware Publishing - Mat Buckland - 2005
Game Institute (www.gameinstitute.com) - Artificial Intelligence for Games - curs

Florin Anghel

Designul experienței utilizatorului în dezvoltarea de jocuri

Pentru a putea vorbi despre UX Design, trebuie pentru început să înțelegem ce înseamnă UI, UX, dar şi în ce constă proiectarea fiecăreia în parte.

UI este acronimul de la „User Interface" şi se referă la toate componentele vizuale cu care utilizatorul poate sau nu să interacționeze, ceea ce înseamnă că deşi acesta poate să acționeze asupra sa (de exemplu să dea un click pe componentă), aceasta nu reacționează neapărat. De exemplu, o componentă cu care se poate interacționa într-un joc este un obiect (buton) din meniu, iar o componentă vizuală care nu prezintă nici un fel de reacție la orice stimul din partea utilizatorului este fundalul meniului (deşi în unele jocuri, ca TumbleBugs, şi această componentă permite interacțiunea).

UX este acronimul de la „User eXperience" şi reprezintă experiența pe care o are utilizatorul în urma stimulilor primiți. Aceşti stimuli sunt receptați de către simțurile utilizatorului, astfel că pot fi de trei tipuri: vizuali (de la monitor prin ochii), auditivi (de la sistemul audio prin urechi), respectiv tactili (de la sistemele de input – ex. gamepad – prin piele).

În cazul dezvoltării de jocuri, UX-ul din timpul jocului propriu-zis este cunoscut şi ca „gameplay". Pentru a oferi o experiență plăcută utilizatorului, un game designer trebuie să găsească o combinație optimă de stimuli pe care să-i transmită jucătorului. De asemenea, acesta trebuie să fie conştient de faptul că odată receptați, aceşti stimuli pot crea o varietate de sentimente.

Când vorbim de design, fie el de jocuri, produse, sau pur şi simplu grafică, totul ține de psihic, iar un designer priceput ştie să se folosească de acest lucru. Poate ați mai auzit de expresia „nu contează ce vinzi, important e cum prezinți produsul" (sau „nu contează ce vinzi atâta timp cât are un ambalaj frumos"). Este cât se poate de adevărată... dar nu şi în cazul jocurilor! O grafică atractivă nu înseamnă tot, dar efectele vizuale potrivite la timpul potrivit completează cu siguranță jocul. Nişte sunete nu distrează jucătorul de unele singure, dar un soundtrack sincronizat cu acțiunea o să-i crească fără îndoială ritmul cardiac jucătorului.

Aşa că la următorul proiect, acordați ceva mai mult timp proiectării experienței utilizatorului şi nu uitați că aveți puterea de a decide ce sentimente au jucătorii, iar voi trebuie să le dați ce vor. Mai precis ce sentimente vor să aibă jucătorii? Veți afla în ediția viitoare a revistei.

Nicusor Nedelcu

Abstractizarea subsistemelor unui motor de joc, plugin system

Acest articol vă va ghida în proiectarea unei arhitecturi software bazate pe module „plug-in". Este nevoie de ceva experiență în C + +, folosirea bibliotecilor dinamice DLL precum şi o bună înțelegere a conceptelor OOP fundamentale, cum ar fi interfaces şi class factory. Dar, înainte de a începe, sa vedem ce avantaje putem obține de la plugin-uri şi de ce ar trebui să le utilizăm:

AVANTAJE

- Claritate sporită şi uniformitate in cod – un plugin poate encapsula cod din alte surse 3rdparty, asa ca tot codul specific unei implementari ramâne într-un loc separat.
- Îmbunătățeşte modularizarea proiectului - codul vostru este curat, separat în module distincte. Acest proces de decuplare a componentelor, creează cod care poate fi refolosit mai uşor.
- Compilare mai rapidă - compilatorul nu este obligat să parseze anteturile bibliotecilor externe, folosite in plugin-uri. Acest lucru poate reduce drastic compilarea.
- Înlocuirea sau adăugarea de componente - Dacă aveți de trimis patch-uri pentru utilizatorului final, este de multe ori suficient pentru a actualiza plugin-urile în loc să trimiteți un installer mare. Un renderer nou sau unele tipuri noi de unitati pentru un add-on la jocul dvs. (inclusiv mods făcute de către utilizatorii finali) ar putea fi uşor adăugate doar prin furnizarea unui set de plugin-uri pentru jocul dvs. sau motorul jocului

DEZAVANTAJE

- În schema de funcționare a acestui sistem se va adăuga, logic, un overhead prin chemarea de metode virtuale ale claselor definite in engine si implementate in plugin
- Sistemul de plugins este mai complicat un pic, fata de un sistem de clase intr-un sigur loc, însă odată înțeles cum funcționează şi cum poate fi folosit la maximum, este uşor de extins si intretinut.

Incarcarea si folosirea unei biblioteci dinamice DLL

Pentru a incarca un fisier DLL (Dynamic Link Library) dintr-un fisier EXE, folosim functia Win32 API : LoadLibrary
HMODULE WINAPI LoadLibrary(LPCTSTR lpFileName);
care va returna un handle catre acel DLL incarcat, lpFileName fiind numele/calea fisierului DLL de incarcat.
Pentru a obtine un pointer catre o functie exportata de acel DLL, folosim functia GetProcAddress:
WINAPI GetProcAddress(HMODULE hModule, LPCSTR lpProcName);

Exemplu:

```
// incarcam un DLL
HMODULE hDLL = LoadLibrary( "my\\path\\some.dll" );
// definim un tip pointer la o functie, care returneaza un pointer la o instanta a clasei SomeClass
typedef SomeClass* (*PFN_FUNC) ( int someArg );
// returnam in pFunc adresa functiei din DLL, createSomeClass
PFN_FUNC pFunc = GetProcAddress( hDLL, "createSomeClass" );
// utilizam functia intr-un call normal
SomeClass* pObj = pFunc( 120 );
```

Arhitectura sistemului de plugin-uri

Este formata din cateva clase: PluginClassInfo, PluginInfo, Plugin si PluginManager
Plugin-urile sunt defapt DLL-uri care sunt incarcate de PluginManager, acesta cauta o functie speciala in acel DLL, de exemplu:

Clasa interfata PluginClassInfo, care tine informatii despre o clasa-plugin, sa ii zicem un fel de plugin-class-descriptor (surse din motorul meu Nytro):

Cod sursă:

```
//! This class holds the info about a C++ class from inside a plugin
class NYTRO_API PluginClassInfo
{
    public:

    //! returns am instance of the class
    virtual void*           newInstance()       = 0;

    //! returns the super class ID, see nyCommon.h
    virtual long            getSuperClassID()   = 0;

    //! returns the class title string
    virtual char*           getTitle()          = 0;

    //! returns the class name as it is in C++
    virtual char*           getClassName()      = 0;

    //! returns the class description, comments
    virtual char*           getDescription()    = 0;

    //! returns the class version number
    virtual int             getVersion()        = 0;

    //! returns user data pointer
    virtual void*           getUserData()       = 0;
};
```

- cand se apeleaza newInstance prin intermediul PluginManager vei obtine o instanta la acea clasa plugin
- super class ID e un ID de genul: ID_ENTITY, ID_EDITOR, ID_UTILITY etc, care specifica clasa de baza din care plugin-

class-ul e derivat
- classname este numele C++ al clasei, exact asa cum se numeste in codul sursa, trebuie specificat neaparat deoarece asta e un fel de UID (unique ID), prin care acea clasa este identificata in sistemul de plugins
- restul sunt diverse informatii despre clasa.

Urmatoarea clasa este PluginInfo care tine informatii despre acel DLL-plugin si clasele plugin exportate de acesta:

Cod sursă:

```
//! Plugin info class that holds the info about a dy-
namic library plugin
class NYTRO_API PluginInfo
{
    public:

   //! get the number of classes hosted by the plugin
   virtual int     getClassCount() = 0;

   //! get the class info object for the class from
the aIndex
   virtual PluginClassInfo*    getClassInfo( int aI-
ndex ) = 0;

   //! get the plugin title string
   virtual char*   getTitle()                  = 0;
  //! get the plugin description string
  virtual char*    getDescription()            = 0;
  //! get the plugin copyright string
  virtual char*    getCopyright()              = 0;
  //! get the plugin version
  virtual int      getVersion()                = 0;
  //! get the plugin package name, some kind of plu-
gin grouping,
  //! for example provider plugins have "PROVIDER_
PACKAGE"
  virtual char*    getPackageTypeName()        = 0;
  //! get this plugin's user data
  virtual void*    getUserData()               = 0;
};
```

Se intelege de la sine ce semnificatie are fiecare membru.

Apoi clasa Plugin care tine toate aceste date, dupa ce pluginul a fost incarcat de PluginManager:

Cod sursă:

```
//! A plugin holder class, from a dynamic library
class NYTRO_API Plugin
{
    public:

        Plugin();
        Plugin( const char* pFile );
        virtual ~Plugin();

        //! returns the dynamic library handle
        LibraryHandle           getLibHandle();

        //! returns the dynamic library file name
        string&                 getLibFile();

        //! returns this plugin's info class
        PluginInfo*             getPluginInfoClass();

        //! loads a dynamic library plugin
        Result                  load( const char*
pFile );

        //! free the dynamic library plugin
        Result                  free();

        //! sorts and retrieves the plugin classes
which have the specified super class ID, into rClasses
        //! return the number of classes found, or
zero if none found
        int                     sortBySuperClassID(
long aSuperClassID, vector& rClasses );

    private:

        LibraryHandle           m_hLib;
        PluginInfo*             m_pPluginInfo;
        vector m_classesInfo;
        string                  m_libFile;

        //! the plugin entry point exported function
from the dynamic library, retrieved with getProcAd-
dress
        PluginInfo*             (*getPluginInfo)();
};
```

La fel e o clasa simpla, cu diverse metode utilitare care nu necesita vreo explicatie.

In cele din urma managerul central care incarca si tine lista plugin-urilor, si care arata cam asa:

Cod sursă:

```
//! A plugin manager, holds an array of Plugin ob-
jects
class NYTRO_API PluginManager
{
    public:

        PluginManager();
        //! creates the object and loads all plugins
from pDir using the pFileMask, ex.: "*.nyp"
        PluginManager( const char* pDir, const char*
pFileMask );
        virtual ~PluginManager();

        //! load all plugins from pDir using the
pFileMask, ex.: "*.nyp"
        int                 loadPlugins( const char*
pDir, const char* pFileMask );

        //! free the loaded plugins and their dynamic
libraries
        Result              freePlugins();

        //! sorts and retrieves from all the plugins
the classes that have super class ID as aSuperID
        //! return the number of found classes
        int                 sortBySuperClassID( long
aSuperID, vector& rList );

        //! get the plugins array
        vector&    getPlugins();

        //! get the class info for a specified class
name residing in one of the plugins, or NULL if none
found
        PluginClassInfo*    getClassInfo( const char*
pClass );

    private:

        vector<Plugin*>   m_plugins;
};
```

Inca odata, explicatiile sunt in comentariile metodelor.
In continuare sa facem un exemplu de DLL plugin cu 2 clase exportate:

MyPluginClass1.h
Cod sursă:

```
class MyPluginClass1 : public Entity
{
    public:
        ..................
}
```

MyPluginClass2.h
Cod sursă:

```
class MyPluginClass2 : public Entity
{
    public:
        ..................
}
```

MyPluginPack.cpp
Cod sursă:

```
class NYTRO_API MyPluginClass1Info : public Plugin-
ClassInfo
{
    public:
    void* newInstance(){ return new MyPluginClass1; }
    long getSuperClassID(){ return SUPER_ID_ENTITY; }
    char* getTitle(){ return "My Cool Entity 1"; }
    char* getClassName(){ return "MyPluginClass1"; }
    char* getDescription(){ return "No smart descrip-
tion here"; }
    int   getVersion() { return 1; }
    void* getUserData(){ return NULL; }
};

class NYTRO_API MyPluginClass2Info : public Plugin-
ClassInfo
{
    public:
    void* newInstance(){ return new MyPluginClass2; }
    long getSuperClassID(){ return SUPER_ID_ENTITY; }
    char* getTitle(){ return "My Cool Entity 2"; }
    char* getClassName(){ return "MyPluginClass2"; }
    char* getDescription(){ return "No smart descrip-
tion here"; }
    int   getVersion() { return 1; }
    void* getUserData(){ return NULL; }
};

// declaram cele doua class-info
MyPluginClass1Info g_MyPluginClass1Info;
MyPluginClass2Info g_MyPluginClass2Info;

// vectorul cu lista de clase ale plugin DLL-ului
PluginClassInfo* g_thisPluginClassInfos[] = { &g_My-
PluginClass1Info, &g_MyPluginClass2Info };

// clasa care contine informatiile despre acest DLL
plugin
class NYTRO_API ThisPluginInfo : public PluginInfo
{
    public:
        int getClassCount(){ re-
turn sizeof( g_thisPluginClassInfos ) /
sizeof( g_thisPluginClassInfos[0] ); }
    PluginClassInfo* getClassInfo( int aIndex )  {
return g_thisPluginClassInfos[aIndex]; }
    char* getTitle(){ return "My Plugin Pack"; }
    char* getDescription(){ return "This plugin pack
contains 2 classes! yay!"; }
    char* getCopyright(){ return "(C) None"; }
    int   getVersion(){ return 1; }
    char* getPackageTypeName(){ return "ENTITY_PACK-
AGE"; }
  void* getUserData(){ return NULL; }
};

// declaram obiectul pentru plugin info
ThisPluginInfo g_thisPluginInfo;

// entry-point care va fi chemat de PluginManager,
daca nu e gasit atunci acest DLL nu va fi considerat
plugin
PluginInfo* NYTRO_API __getPluginInfo() { return (Pl-
uginInfo*) &g_thisPluginInfo; }
```

* NYTRO_API - __declspec(dllexport)

Pentru definirea acelor clase se pot face si niste #define-uri care sa usureze viata, cu ele ar arata cam asa:

Cod sursă:

```
PLUGIN_CLASS_INFO( MyPluginClass1, SUPER_ID_ENTITY,
"My Cool Entity 1", "No smart description here", 1,
NULL );
PLUGIN_CLASS_INFO( MyPluginClass2, SUPER_ID_ENTITY,
"My Cool Entity 2", "No smart description here", 1,
NULL );

BEGIN_PLUGIN_CLASSES
    ADD_PLUGIN_CLASS( MyPluginClass1 )
    ADD_PLUGIN_CLASS( MyPluginClass2 )
END_PLUGIN_CLASSES;

PLUGIN_INFO( "My Plugin Pack", "This plugin pack con-
tains 2 classes! yay!", "(C) None", 1, "ENTITY_PACK-
AGE", NULL );
```

Utilizarea PluginManager si instantierea unui plugin intr-un game.exe de exemplu:

Cod sursă:

```
pluginMan->loadPlugins( "/plugins" );
.....................
PluginClassInfo* classInfo = pluginMan->getClassInfo(
"MyPluginClass1" );

// daca exista
if( classInfo )
{
    // clasa de baza este entity, pentru ca din ea am
derivat MyPluginClass1
    Entity* pluggedEntity = (Entity*)classInfo-
>newInstance();

    // voila!, new plugin entity, use it!
}
```

Cam asta ar fi, sper ca s-a inteles cat de cat :)

MY GDRO PERSONAL NOTES AREA

the thinking place