

Dezvoltarea de jocuri multidevice și multiplatformă folosind XNA

Mogoș Dumitru Radu
radu.mogos@pixelplant.ro

2010

Cuprins

1. Introducere.....	3
2. Informații generale despre XNA.....	4
2.1.Diferențe între .NET și .NET versiunea compactă.....	4
2.2.Dezvoltarea pentru platforme multiple.....	5
2.3.Planificarea unui joc.....	5
2.4.Concepte generale de programare ale jocurilor.....	7
2.5.Structura XNA.....	9
2.5.1. Procesatori vs Importatori de conținut.....	9
2.5.2. Componente. GameComponents și DrawableGameComponents.....	10
2.5.2.1 Tipuri de componente de joc.....	10
2.5.2.2 Când și ce tip de componentă folosim.....	11
2.5.2.3 Cum sunt apelate componentele?.....	11
2.5.3. Loop-ul principal și XNA.....	12
3. Construirea unui joc simplu în XNA.....	15
3.1.Inspectarea codului generat.....	17
3.2.Modificarea metodelor Update și Draw	18
4. Jocul de BlackJack.....	23
4.1.Structurarea proiectului.....	23
4.2.Metode de input.....	23
4.3.Structura codului.....	24
4.3.1. Directoarele de cod sursă.....	25
4.3.2. Locul unde se întâmplă magia.....	34
5. Platforme diferite. Device-uri multiple.....	37
5.1.Windows Phone 7.....	37
5.2.Platforma PC.....	39
5.3.Sugestii de optimizare.....	40
5.3.1. Administrarea memoriei.....	40
5.3.2. Blocarea metodelor virtuale.....	41
5.4.Cazul BlackJack.....	41
5.4.1. Rezoluția.....	42
5.4.2. Input	42
5.4.3. Shadere	43
6. Concluzie.....	47
7. Bibliografie.....	48

1. INTRODUCERE

Cartea conține fragmente din lucrarea mea de disertație și dezbate modalitatea de dezvoltare a jocurilor pe platforme multiple (telefoane, PC, console) folosind o bază de cod comună, precum și metode de interacțiune între utilizatorii finali și aplicația în sine. Soluția dezvoltată reprezintă un studiu de caz în ceea ce privește limbajul de programare C# și platforma de dezvoltare jocuri XNA, ajunsă în prezent la versiunea 4.0

XNA¹ este un framework ce permite scrierea de cod managed folosind C# și DirectX, codul rezultat putând fi rulat pe Windows Phone 7, pe PC, pe consola XBOX 360 sau pe dispozitivele Zune. În funcție de cerințele și specificul jocului, același cod poate rula fără prea mari modificări pe toate platformele, reducând astfel drastic din timpul de dezvoltare și din costurile aferente portării unei soluții pe platforme diferite.

Pentru aplicația practică am ales dezvoltarea jocului 21 (BlackJack²), un joc de cărți destul de popular al cărui scop principal este adunarea unei mâini de cărți care să totalizeze cel mult 21 de puncte. Jocul rulează atât pe PC cât și pe Windows Phone 7 (fiind testat în emulator doar), codul folosit de ambele soluții fiind în proporție de 95% același.

O altă diferență majoră în dezvoltarea de aplicații multi device constă în mecanismele diferite pe care le avem la îndemână pentru a interacționa cu jucătorul. De exemplu, pentru PC avem acces la tastatură și mouse (și eventual un gamepad), pe consola XBOX 360 avem acces doar la controllerul de XBOX, iar pe Windows Phone 7 avem acces la ecranul tactil și la accelerometru, de aceea inputul și modul de interacționare cu jucătorul pentru un joc ce rulează pe diferite platforme, trebuie gândit astfel încât să ofere aceeași ușurință și plăcere, pe orice platformă ar rula acesta. Modul creativ de a folosi inputuri atât de diferite pentru a rula același joc, devine deci, cel mai important factor pentru succesul jocului, pe lângă gameplay-ul în sine.

1 <http://creators.xna.com/> - site oficial XNA

2 <http://ro.wikipedia.org/wiki/Blackjack> - informații generale despre jocul de cărți BlackJack

2. INFORMAȚII GENERALE DESPRE XNA

XNA este o librărie ce permite crearea de jocuri care să ruleze pe PC, XBOX 360, Zune HD și Windows Phone 7. C# este limbajul folosit pentru a dezvolta jocuri XNA, ceea ce face din XNA nu doar o unealtă grozavă, ci și o alternativă grozavă pentru oricine de a se apuca să învețe sau să aprofundeze limbajul C#.

Avantajul principal XNA este că permite oricui, de la hobby-ști la dezvoltatori de jocuri profesioniști, să creeze jocuri multi platformă într-un timp foarte scurt, fără să fie necesare cunoștințe avansate în domeniul programării jocurilor.

XNA folosește aceeași bază de cod pentru toate platformele, astfel că aceeași metodă de desenare va avea același nume și aceiași parametri și pe Zune și pe PC, însă unele funcționalități nu vor rula pe toate platformele (de exemplu suportul pentru shadere), iar principala diferență între platforme va fi versiunea librăriei .NET pe care acestea o au instalată și o pot rula. De exemplu, consola XBOX 360, Zune și Windows Phone 7 rulează o versiune compactă a librăriei .NET, spre deosebire de PC, astfel chiar dacă accesul la resurse se face prin aceleași metode, nu înseamnă că vor avea același randament, sau că o metodă optimă pe PC va fi optimă și pe un dispozitiv mobil.

2.1. Diferențe între .NET și .NET versiunea compactă

După cum îi spune și numele, versiunea compactă .NET folosește un subset al librăriei .NET, fiind folosită pe dispozitive cu posibilități hardware reduse față de PC. XBOX 360, Zune și Windows Phone 7 folosesc o versiune compactă a librăriei .NET.

XNA este scris însă în așa fel încât să nu fim nevoiți să cunoaștem dedesubturile nivelurilor joase de dezvoltare, însă informațiile acestea ne pot ajuta să scriem cod optim, iar atunci când optimizezi pentru versiunea compactă, va merge la fel de bine și pe versiunea completă .NET.

Chiar dacă denumirea de compact ne duce în principal cu gândul la minusuri, în ce privește XBOX, de exemplu, .NET compact framework are incluse și funcționalități adiționale, ce îi permit să acceseze multiple core-uri pe consolă, funcționalități ce nu pot fi disponibile pe .NET PC, datorită modului în care este construit și configurat hardware-ul folosit de ambele platforme.

Un exemplu de diferențe în care optimizarea are un rol important o reprezintă Garbage Collectorul. Pe PC, .NET, Garbage Collectorul poate să treacă un obiect prin 3 generații, până să îl colecteze și să îl distrugă. Versiunea compactă .NET nu are noțiunea de generații, Garbage Collectorul tratând fiecare obiect ca fiind la ultima generație.

2.2. Dezvoltarea pentru platforme multiple

Înainte de a trece la cod, trebuie să vedem de ce anume avem nevoie pentru a putea dezvolta pentru platforme multiple. În cazul nostru avem librăria XNA și partea de deployment este și ea inclusă. Am putea trece direct la cod, însă întrebarea principală este ce abordare luăm? Ne apucăm din start și scriem cod diferit pentru toate platformele? Sau folosim o bază comună, și apoi, în funcție de platformă, extindem codul după necesități și limitările platformei respective?

În general, e bine să folosim cât de mult cod comun putem pentru toate platformele, însă dacă nu avem la dispoziție un XBOX 360, nu prea are rost să dezvoltăm o versiune pentru consolă. Nu vom avea nici cum să testăm jocul pe consolă, și nici cum să facem optimizări de cod. Dacă acel cod nu va fi scris decât pentru versiunea de PC, atunci probabil că e bine să ne gândim din start la cod ce va rula optim pe PC, și să uităm de cod comun și structuri de date complexe.

În final, totul se rezumă la cerințele respective ale jocului, la limitările platformei pentru care se dezvoltă și cel mai important, la timpul pe care îl poți aloca dezvoltării.

2.3. Planificarea unui joc

Există numeroase genuri de jocuri care pot fi dezvoltate, iar la rândul lor, aceste genuri pot fi implementate folosind tehnologii 2D sau 3D. Deși exemplul prezentat în această lucrare are caracter demonstrativ, aspectele privind planificarea jocului trebuie să aibă în vizor mai ales piața căruia jocul îi este adresată și genul de bază căruia jocul îi aparține.

Planificarea implică trecerea peste următoarele aspecte:

- **Scopul jocului** – ce are de făcut jucătorul și de ce ar continua să joace acest joc.
- **Povestea** – este cea care contribuie la contopirea jucătorului cu lumea virtuală, care îl face să se simtă un jucător de fotbal de neînvincut sau un erou grec ce ține soarta lumii în mâinile sale. Povestea influențează gameplay-ul și poate contribui la inovativitatea jocului.
- **Gameplay** – scopul principal al oricărui joc este să distreze. Este cel mai important aspect care trebuie luat în vedere atunci când dezvoltați jocuri. Jucătorul nu trebuie taxat aspru când greșește, punctele de respawn trebuie să îi permită să reîncearcă ultima mișcare greșită făcută, AI-ul dușmanilor trebuie să fie coordonat cu abilitățile dobândite de jucător până în acel moment (ca să nu fie nici prea dificilă trecerea, dar nici prea ușoară), ținând astfel în priză jucătorul și menținându-i un nivel ridicat de interes pentru lumea virtuală în care se află.
- **Varietate** – deși e greu să fi inovativ într-un domeniu în care idei noi sunt

promovate de la joc la joc, varietatea este un alt aspect ce contribuie la menținerea unui interes crescut pentru joc. Dacă dezvolți un joc „action-adventure”³, vei îmbina scenele de luptă cu diferite scene în care poți introduce mici puzzle-uri, deblocând astfel jucătorul de la acțiunile de luptă mecanice, și cerându-i să își folosească creativitatea și inteligența. Echilibrarea acestor etape poate duce la succesul sau eșecul lamentabil al jocului la care lucrezi.

- **Premii** – o poveste bună cu un gameplay inteligent va duce la mulți jucători mulțumiți. Pe măsură ce înaintează însă în joc, jucătorul va dori să primească, cumva, o recunoaștere a abilităților dobândite, a experienței câștigate pe parcursul jocului, așa că de-a lungul jocului e bine ca traseul să fie presărat cu multiple premii pe care jucătorul le poate câștiga: diferite upgrade-uri atunci când nivelul eroului crește, power-ups pentru arme, viață, magie, deblocarea de arme, armuri sau diferiți alți eroi, etc. O altă metodă de a premia jucătorul într-un joc de acțiune, este de a-l pune în situația de a lupta la finalul rundei cu un boss de nivel. Jocul devine mai variat astfel, și jucătorul poate să își pună mai bine în valoare abilitățile adunate până în acel moment (vezi **fig 2.2**).



Fig 2.1 – Exemplu de power-up de viteză din jocul „Avioane”, pentru Windows Phone 7, disponibil pe www.xna.ro

3 Action Adventure - http://en.wikipedia.org/wiki/Action-adventure_game



Fig 2.2 – Lupta cu boss-ul de nivel în jocul „Avioane”, disponibil pe www.xna.ro

- **Puncte de salvare bine gândite** – Nimeni nu vrea să reia secțiuni mari de joc de la început, în cazul în care ratează un anumit punct de trecere. Jucătorul trebuie să poată salva ușor (atât cât îi permite și structura jocului) progresul său în anumite momente din timp, ca apoi să poată relua acțiunea cât mai recent cu putință.

Odată echilibrate aceste aspecte, putem trece la următorii pași în dezvoltarea unui joc. Vom trece peste concepte generale în programarea de jocuri apoi vom vedea modul în care aceste concepte pot fi integrate în XNA.

2.4. Concepte generale de programare ale jocurilor

Structura unui joc este una repetitivă. Jocul în sine este un loop principal care trece

printr-o serie de faze (preluare input, calcule inteligență artificială, calcule fizică joc, prelucrare animații și sunet și într-un final, afișarea rezultatului final pe ecran), ce se repetă până în momentul în care jucătorul revine la meniul principal sau părăsește aplicația.

Scopul principal al optimizărilor este de a rula acest loop (sau cel puțin părți din el) la o viteză cât mai mare, generând astfel o rată de desenare cât mai mare (FPS⁴). Am specificat mai sus „cel puțin părți din el” deoarece valoarea FPS returnează de câte ori reușim să apelăm metoda de desenare a jocului, pe secundă. FPS ne spune de câte ori pe secundă am apelat metoda de desenare, și doar această metodă. Metoda de desenare este cea pe care întotdeauna vom încerca să o apelăm cât mai rapid, pentru un gameplay fin, și pentru o experiență de joc în timp real. Celelalte metode, în funcție de cerințele jocului, pot fi apelate la intervale mai mari.

Ca să exemplificăm, într-un joc în care jucătorul interacționează cu puține obiecte, putem apela metoda de fizică (ce include și verificările de coliziune), de zece ori pe secundă, lăsând metoda de desenare să se execute de cel puțin 60 de ori pe secundă. Cum metoda de fizică include și deplasarea obiectelor, cadrele pierdute în acea secundă pot fi calculate folosind interpolări.

Reprezentat într-o structură de cod simplă, loop-ul principal al unui joc ar putea arăta așa:

```
metoda principala()
{
    atata timp cat jocul ruleaza
    {
        proceseazaJoc()
    }
}

metoda proceseazaJoc()
{
    proceseazaStareObiecte() // procesarea obiecte joc (masini, oameni,..)
    proceseazaInput() // preluare tastatura, mouse, controller, touchpad
    proceseazaInteligențaArtificiala()
    proceseazaFizica() // miscare obiecte, coliziuni, etc
    proceseazaAnimatii() // animatii 2d sau 3d
    proceseazaSunet() // ruleaza muzica d0cor, sunete coliziuni, arme, etc
    deseneazaTotPeEcran() // odata calculele terminate, desenam pe ecran
}
```

Bineînțeles că în funcție de tipul jocului, unele părți ar putea să lipsească. De exemplu, în cazul unui joc X și 0, partea de inteligență artificială ar lipsi.

4 FPS = Frames Per Second (frecvența de desenare a cadrelor, raportată la secunde)

2.5. Structura XNA

Asset-urile unui joc sunt reprezentate de texturi, sunete, modele 3D, fișiere pentru shading, fișiere pentru hărți, fișiere de configurare, etc. XNA include o clasă (ContentManager) ce poate procesa automat diferite tipuri de conținut, încât ele să fie folosite direct în joc, fără să fie nevoie ca dezvoltatorul să scrie metode de importare imagini sau modele 3D. XNA include clasele Content Processor, Content Reader, Content Writer și Content Importer care îi spun cum să importe conținutul în Content Pipeline și cum să compileze acest conținut ca să poată fi încărcat fără probleme pe toate platformele pe care XNA rulează.

Fiecare tip de conținut are propriul lui Content Importer și Content Processor; în timpul compilării XNA invocă clasa de import conținut și procesare specifică tipului de conținut ce trebuie încărcat.

2.5.1. Procesatori vs Importatori de conținut

- Un **importator de conținut** (Content Importer) preia asseturile și le convertește în obiecte ce pot fi consumate de procesatorii de conținut standard (cei care vin deja cu XNA), sau într-o altă formă customizată ce poate fi prelucrată de importatorii de conținut scriși de noi.
- Un **procesator** ia un tip specific de asset-uri importate (de exemplu un set de mesh-uri 3D, un set de fonturi), le compilează în cod obiect managed care poate fi încărcat și folosit de XNA pe platforma Windows, XBOX 360, Zune sau Windows Phone.

Pentru jocul prezentat în această lucrare vom folosi doar importatorii și procesatorii de conținut standard XNA. Informații despre ce tip de conținut avem încărcat, putem vedea din IDE-ul Visual Studio C#, selectând asset-ul dorit.

În exemplul de mai jos, când se selectează o imagine PNG putem vedea că atât importatorul cât și procesatorul de conținut sunt setate la valoarea „**Texture – XNA Framework**”. Extinzând căsuța de selecție, putem vedea și lista de procesatori și importatori standard, care sunt oferți de XNA.

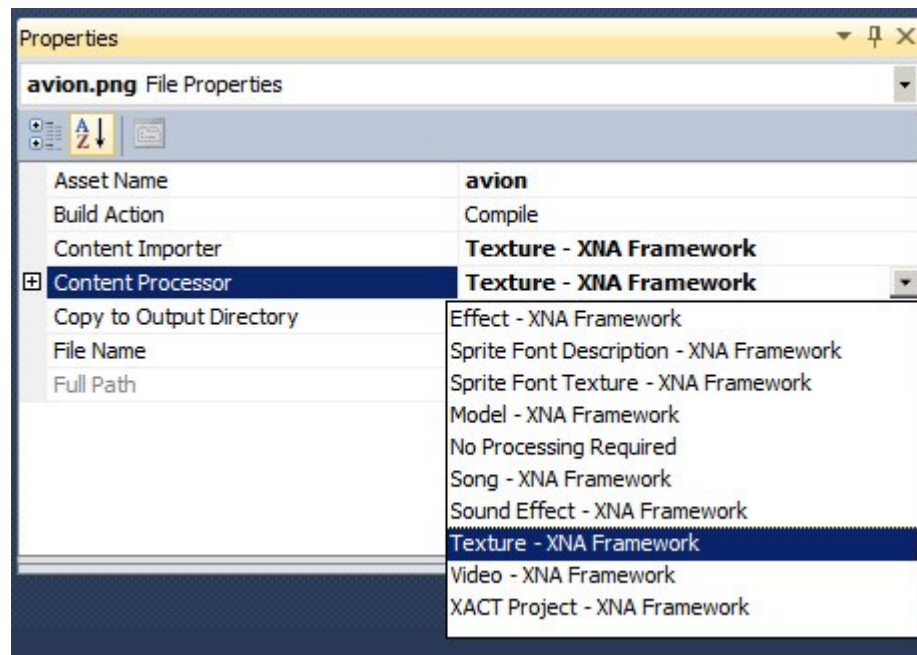


Fig 2.3 – Content Importer și Content Processor folosit de un sprite⁵ în format PNG.

2.5.2. Componente. GameComponents și DrawableGameComponents

XNA este structurat pe componente de joc (game components), care includ deja metode pentru desenare, inițializare variabile, încărcare conținut și procesare logică joc.

Principalul scop al componentelor este structurarea codului și re folosirea acestuia. Dacă ai de scris cod ce se ocupă cu desenarea modelelor sau cu rularea sunetelor în joc, poți structura acest cod într-o componentă de joc. Aceasta s-ar putea ocupa cu încărcarea sunetelor, rularea lor și diferitele efecte care pot fi aplicate acestor sunete. Atunci când ai de scris un alt joc ce are nevoie de o componentă care să ruleze sunete, o să poți folosi componenta deja scrisă, deoarece fiecare metodă de care ai nevoie este deja la locul ei.

Poți scrie și niște clase proprii pe care să le extinzi, și să obții aceeași funcționalitate, dar avantajele componentelor e că vin cu toate metodele de bază gata scrise. Referințele la managerul de conținut sunt scrise, astfel încât poți încărca orice tip de conținut, referința la instanța principală a jocului e deja acolo, iar toate metodele apelabile în clasa principală a jocului pot fi apelate și în componenta de joc.

2.5.2.1 Tipuri de componente de joc

XNA vine cu două tipuri mari de componente de joc:

1. `GameComponent`⁶ – include metode pentru încărcat conținut și pentru metoda `Update`, în care are loc logica jocului, iar în acest caz, logica componentei.

5 Sprite - http://en.wikipedia.org/wiki/Sprite_%28computer_graphics%29

6 Dezvoltarea de componente de joc - <http://msdn.microsoft.com/en-us/library/bb199634.aspx>

2. `DrawableGameComponent` – extinde `GameComponent` și include în plus o metodă pentru desenarea conținutului pe ecran.

2.5.2.2 Când și ce tip de componentă folosim

În următoarele rânduri voi da un exemplu pentru fiecare tip de componentă în parte. Să zicem că avem de dezvoltat un joc simplu în care avem de controlat un avion pe ecranul de joc. Controlul se va face cu ajutorul tastaturii, iar desenarea o vom face 2D.

În această situație, ar fi o practică bună să scriem, dacă nu avem deja, o componentă care să se ocupe cu prelucrarea diferitelor metode de input pe care le va primi jocul (tastatură, mouse, controller, touchpad, etc). Din moment ce această metodă se va ocupa doar de input, vom extinde clasa `GameComponent`, deoarece clasa de input nu va desena nimic pe ecran, ci doar va prelua starea interfețelor de intrare.

Managementul și desenarea avionului sau mai multor avioane o vom face cu o clasă ce extinde `DrawableGameComponent`, deoarece, evident, vom avea nevoie și de metoda de desenare a obiectelor pe ecran.

O practică bună o reprezintă crearea unei componente care să se ocupe de prelucrarea mai multor obiecte, și nu doar a unuia singur. De exemplu, vom scrie o componentă care poate prelucra și încărca toate sunetele unui joc, nu doar un sunet. La fel, vom scrie o componentă care se va ocupa de toate avioanele, sau de o categorie mare de avioane, într-un astfel de joc, și nu doar de unul singur.

2.5.2.3 Cum sunt apelate componentele?

Fiecare instanță a jocului nostru va extinde clasa `Microsoft.Xna.Framework.Game`. Această clasă are incluse niște metode virtuale care sunt definite și pentru componente (similaritatea se rezumă la numele metodelor și parametrii primiți, bineînțeles că cele 2 clase vor extinde clase diferite). Metoda `Initialize` este apelată prima dată când se inițializează jocul. În general, asset-urile sunt încărcate aici. După ce execută codul scris de noi pentru inițializare, va apela metoda `base.Initialize()`; care va trece prin fiecare componentă scrisă de noi și va apela apoi metoda `Initialize` a componentei respective.

Logica jocului va fi preluată din metoda `Update`. La finalul acestei metode, ca și în exemplul de mai sus, va fi apelată funcția `base.Update()` ce va apela metoda `Update` pentru fiecare componentă a jocului în parte. Desenarea are loc în metoda `Draw`, care la fel va apela metoda `base.Draw()` pentru a executa desenarea pe fiecare componentă.

Componentele sunt adăugate de obicei în constructorul instanței jocului, cu apelul `Components.Add(umeComponenta)`, toate componentele adăugate fiind stocate într-o instanță a clasei `GameComponentCollection`⁷.

7 `GameComponentCollection` pe MSDN - <http://msdn.microsoft.com/en->

2.5.3. Loop-ul principal și XNA

Bucula principală a oricărui joc, despre care am vorbit în subcapitolul „Concepte generale de programare ale jocurilor”, are echivalentul său în XNA. Inițializarea jocului și intrarea în bucla principală se face la apelarea metodei `Main`.

```
static void Main(string[] args)
{
    using (BlackjackGame game = new BlackjackGame())
    {
        game.Run();
    }
}
```

Metoda `Run` se ocupă de inițializarea jocului (apelarea metodei `Initialize`, care la rândul ei apelează metoda `Initialize` pentru toate componentele jocului), apoi are loc încărcarea conținutului (apelul metodei `LoadContent`), apoi aplicația intră în loop-ul principal, executând repetat 2 metode, metoda `Update` și metoda `Draw`.

Dacă revenim la pseudocodul prezentat la secțiunea de concepte generale, și anume:

1. `processeazaStareObiecte()`
2. `processeazaInput()`
3. `processeazaInteligentaArtificiala()`
4. `processeazaFizica()`
5. `processeazaAnimatii()`
6. `processeazaSunet()`
7. `deseneazaTotPeEcran()`

vom observa că metoda 1-6 din secvența de mai sus vor corespunde metodei `Update` din XNA, iar metoda 7 va corespunde metodei `Draw` din XNA. În cazul nostru, preluarea `Input` va fi realizată cu ajutorul unei componente, fizica, animațiile și sunetele de asemenea, deci apelul `Update` din clasa jocului va corespunde apelului `Update` pe toate componentele respective. Pentru clarificare consultați imaginea de mai jos.

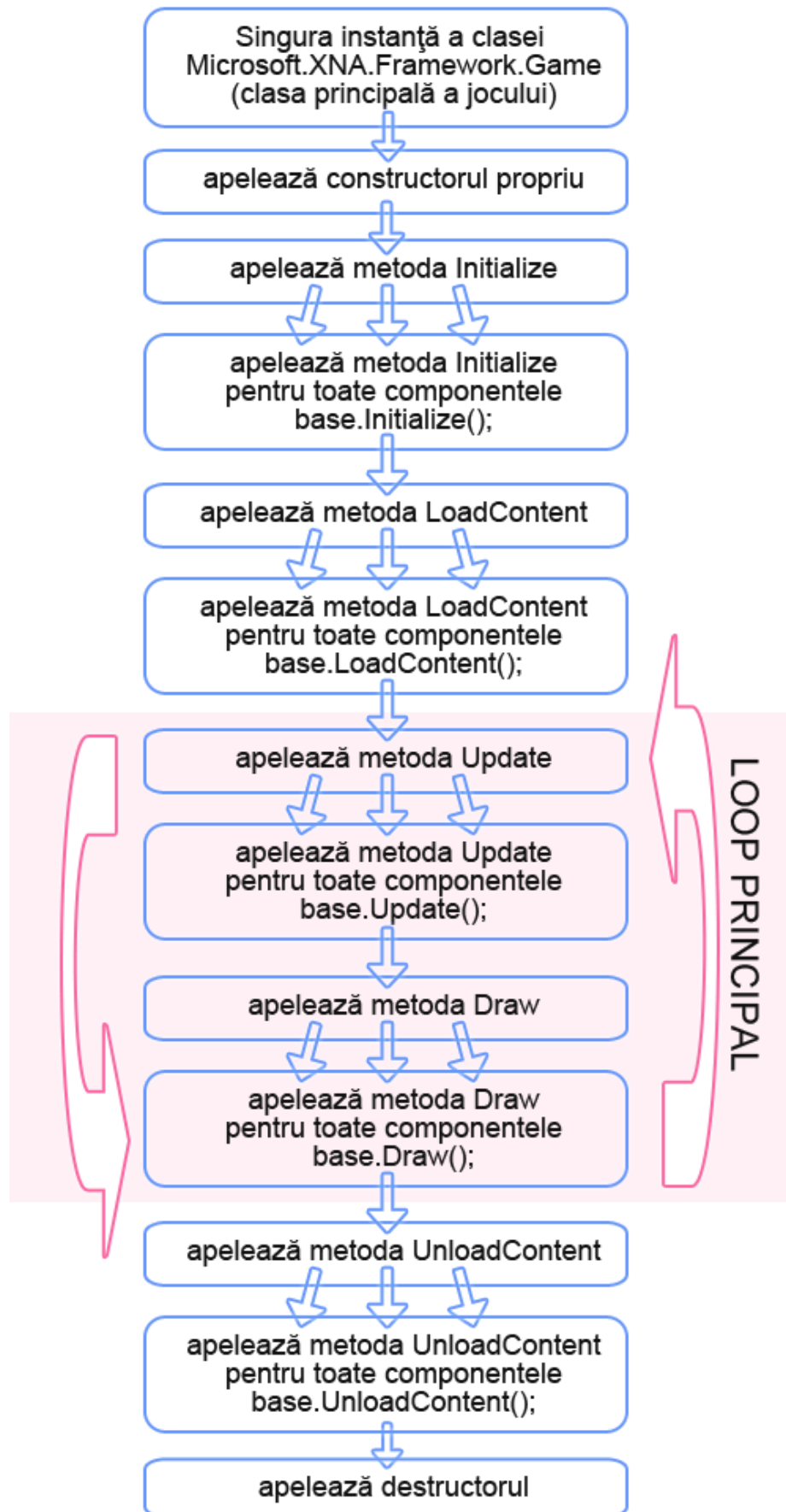


Fig 2.4 – O structură simplă a modului în care este inițializat un joc XNA și a componentelor loop-ului principal dintr-un joc XNA.

Un singur aspect trebuie luat în vedere însă. Deși metodele `Update` și `Draw` sunt apelate în același loop, ele nu sunt apelate cu aceeași frecvență. Standardul de desenare a cadrelor pe secundă pe XBOX este de 60 FPS, iar pe Windows Phone de 30 FPS, ceea ce înseamnă că metoda `Draw` va fi apelată de numărul de ori specificat mai sus (sau cel puțin se va încerca asta). Pe de altă parte, metoda `Update` va fi apelată la intervale diferite, mult mai des decât metoda `Draw`. Ochiul nu va detecta modificări ale gameplay-ului la viteze FPS mai mari decât cele specificate, așa că desenând la vitezele standard de mai sus, vom avea mai multe cicluri libere pentru metoda `Update`, și deci, mai multe cicluri pentru fizica jocului, coliziuni, preluare input, rulare sunete, etc.

3. CONSTRUIREA UNUI JOC SIMPLU ÎN XNA

Cea mai bună metodă de a explica modul de funcționare al unui sistem implică exemple practice de a construi soluții folosind sistemul respectiv. Pentru acest scurt exemplu, cât și pentru jocul prezentat în această lucrare vom folosi Visual Studio 2010 Express pentru Windows Phone și XNA 4.0 CTP. Primul lucru pe care va trebui să îl faceți va fi să descărcați aceste aplicații de pe situl oficial: www.creators.xna.com

Odată instalate aplicațiile, porniți Visual Studio 2010 Express, și alegeți să creați un proiect nou XNA. **File > New Project**

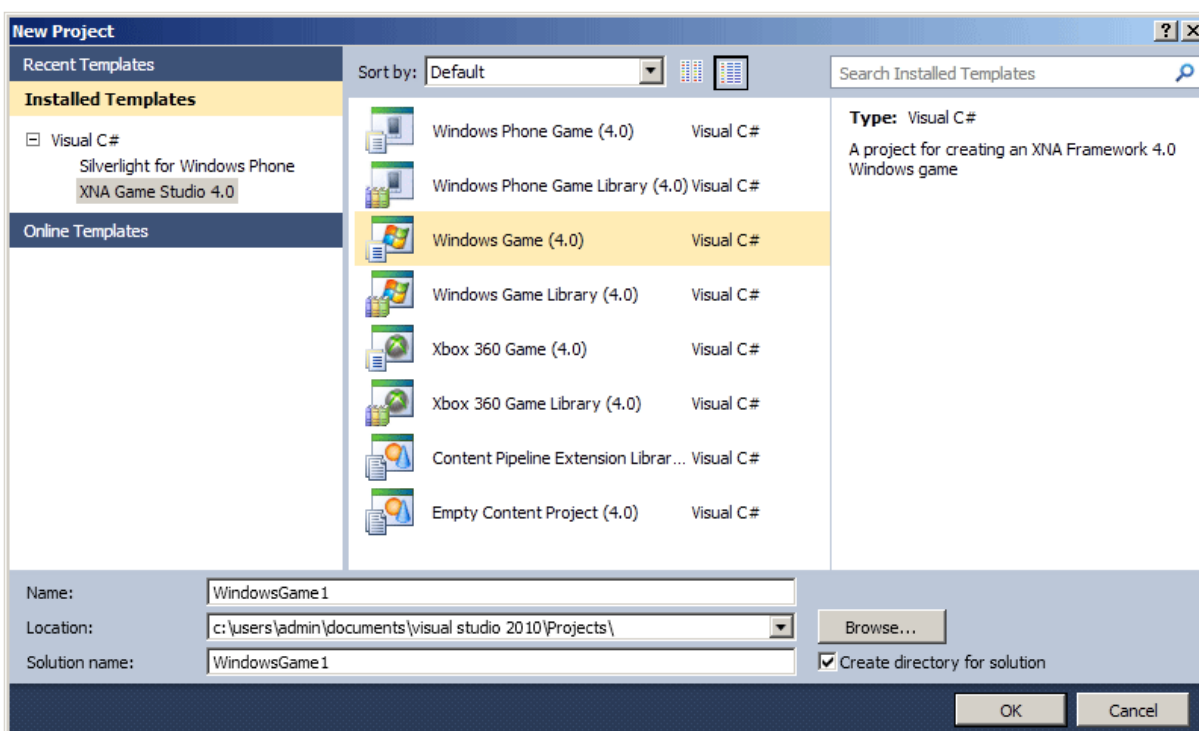


Fig 3.1 Alegerea tipului de proiect

Selectați la tipul de proiect, „Windows Game (4.0)”, dați-i un nume proiectului și alegeți OK. Proiectele de tipul „Windows Phone Game (4.0)” sunt folosite pentru a crea jocuri pentru telefon, „XBOX 360 Game (4.0)” pentru a crea jocuri pentru XBOX, iar cele ce conțin „Library” în denumirea lor sunt folosite pentru a crea librării de cod ce pot fi împărțite între diferite jocuri (similar DLL-urilor din Windows). „Empty Content Project (4.0)” va conține doar partea de asset-uri ale unui proiect, fără codul jocului respectiv. Despre Content Project vom vorbi și în rândurile următoare.

Dacă veți privi cu atenție în „Solution Explorer” (fig 3.2) veți observa că ați creat defapt 2 proiecte: **WindowsGame1** și **WindowsGame1Content(Content)** (dacă ați denumit proiectul inițial *WindowsGame1*). **WindowsGame1** va conține partea de cod a proiectului, iar **WindowsGame1Content** va conține partea de asset-uri (imagini, sunete, modele 3d, etc) și

procesatorii, importatorii de conținut utilizați de aceste asset-uri.

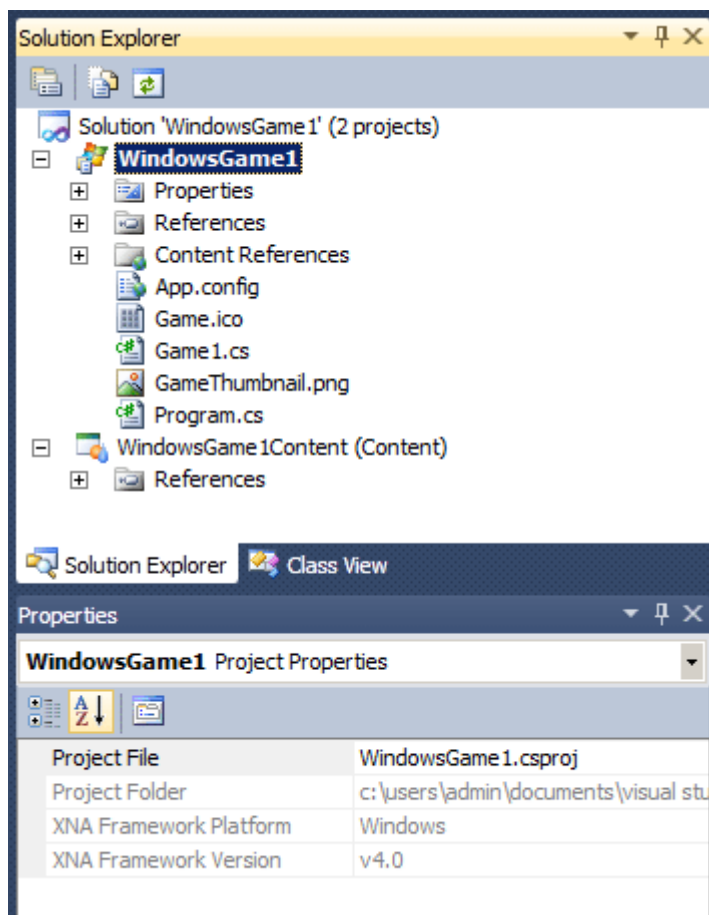


Fig 3.2 Cele 2 proiecte create, în „Solution Explorer”

Dacă veți extinde referințele proiectului **WindowsGame1Content** (vezi fig 3.3) vă veți convinge că acesta se va ocupa doar de partea de conținut (procesare și importare) deoarece toate referințele sale vor fi la Content Pipeline-ul folosit de XNA (Microsoft.Xna.Framework.Content.Pipeline.*)

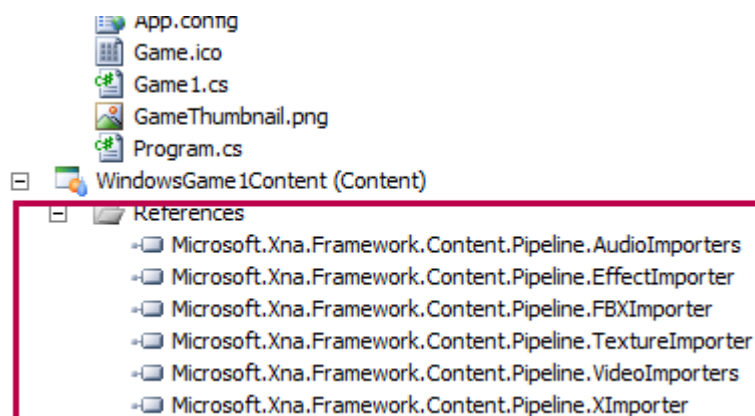


Fig 3.3 Referințele proiectului WindowsGame1Content extinse

Concluzia acestei introduceri este următoarea: **toate asset-urile** vor fi incluse în **WindowsGame1Content**, iar **codul** va merge doar în proiectul **WindowsGame1**.

3.1. Inspectarea codului generat

Al doilea proiect, **WindowsGame1** conține referințele necesare pentru proiectul nostru precum și un fișier *Game1.cs* ce include un schelet de bază pentru jocul nostru, incluzând apelul metodelor de inițializare, `LoadContent`, `Update`, `Draw` și `UnloadContent`.

```
public class Game1 : Microsoft.Xna.Framework.Game
{
    GraphicsDeviceManager graphics;
    SpriteBatch spriteBatch;

    public Game1()
    {
        graphics = new GraphicsDeviceManager(this);
        Content.RootDirectory = "Content";
    }
    ...
}
```

Prima parte declară 2 variabile, `spriteBatch`, de tip `SpriteBatch` și `graphics`, de tip `GraphicsDeviceManager`. Variabila `spriteBatch` se ocupă cu desenarea unei liste de sprite-uri. Setările pentru această variabilă vor fi aplicate pentru desenarea tuturor sprite-urilor incluse între apelul `spriteBatch.Begin()` și `spriteBatch.End()`. Variabila `graphics` se ocupă cu referirea plăcii video (clasa `GraphicsDevice`) și include metode pentru setarea rezoluției jocului (metodele `PreferredBackBufferWidth` și `PreferredBackBufferHeight`) și pornirea jocului în fullscreen sau mod fereastră (metoda `IsFullScreen`).

În constructorul jocului are loc inițializarea managerului video și setarea locației unde poate fi găsit `ContentManager`-ul. Acesta este de fapt o referință la proiectul de **Content WindowsGame1Content** despre care am discutat mai sus. Încărcarea de texturi, sunete, se va face astfel apelând metoda `Content.Load<TipulAssetului>("locatie\nume");`. De exemplu, o textură 2D va fi încărcată cu apelul `Content.Load<Texture2D>("numele_texturii");` și va returna un obiect de tip `Texture2D`.

Metoda de `Update` momentan verifică doar dacă jucătorul apasă tasta `Back` de pe controllerul de XBOX, iar metoda de desenare șterge ecranul și apelează metoda de desenare pentru toate componentele inițializate în joc (momentan nu avem nici o componentă definită).

3.2. Modificarea metodelor Update și Draw

Pentru exemplul nostru simplu, vom adăuga la codul deja generat un sprite care va fi controlat de la tastatură. Pentru asta vom avea nevoie de o variabilă `Texture2D` care va reține textura (fig 3.5) folosită pentru sprite, de o variabilă `Vector2` care va reține poziția pe ecran a sprite-ului (coordonatele X și Y, vezi fig. 3.4), și o variabilă ce va reține starea tastaturii și butoanele deja apăsate, mișcarea sprite-ului fiind controlată de la tastatură.

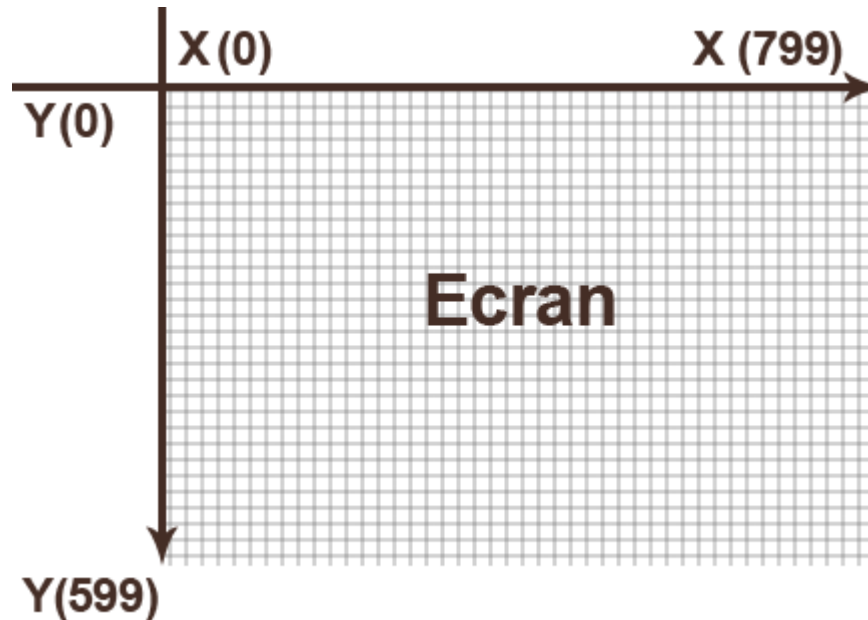


Fig 3.4 Coordonatele ecranului, în spațiu 2D, pentru o rezoluție a ecranului de 800x600



Fig 3.5 Textura pe care o vom folosi pentru exemplu

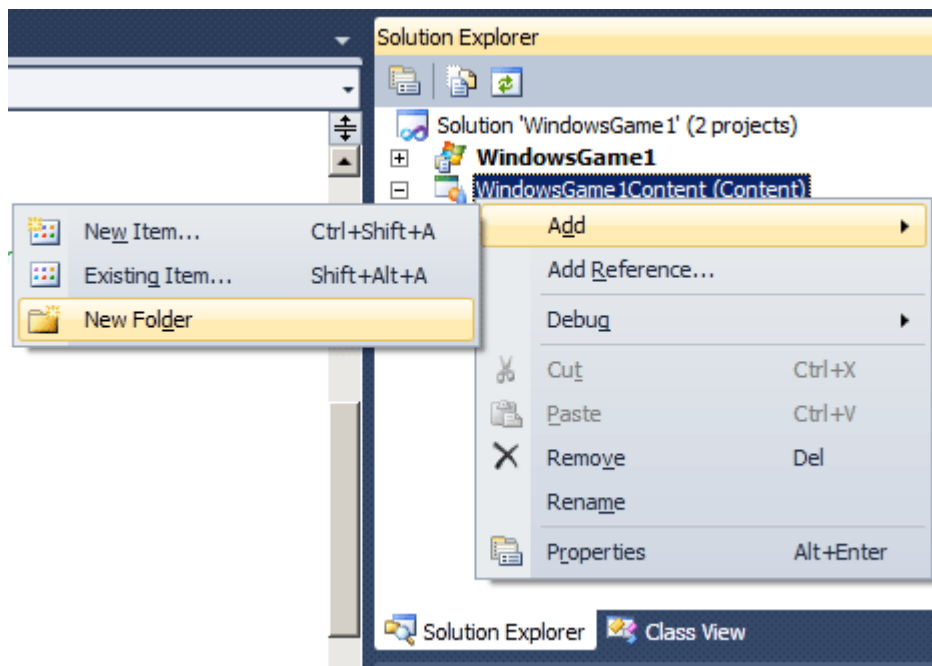
Pornind de la informațiile de mai sus, vom face următoarele modificări în cod:

- Vom defini următoarele 3 variabile pentru clasa `Game1` (clasa principală a jocului)

```
public class Game1 : Microsoft.Xna.Framework.Game
{
    GraphicsDeviceManager graphics;
    SpriteBatch spriteBatch;

    //--- variabile noi ---
    // textura pe care o vom desena pe ecran
    Texture2D textura;
    // plasam avionul la pozitia 100, 100 pe ecran
    Vector2 pozitia = new Vector2(100, 100);
    // preluam starea tastaturii
    KeyboardState keyboardState;
}
```

- Vom crea directorul „**Texturi**” în proiectul de conținut (**WindowsGame1Content**) - click dreapta pe **WindowsGame1Content** > *Add* > *New Folder*, și redenumim directorul „**Texturi**”.



**Fig 3.6 – Adăugarea unui nou director în proiectul de conținut (Content Project)
XNA**

Odată creat directorul, vom adăuga textura avionului pe care o vom folosi pentru acest exemplu – click dreapta pe noul director creat “**Texturi**” > *Add* > *Existing Item...* și apoi selectăm fișierul PNG cu textura avionului.

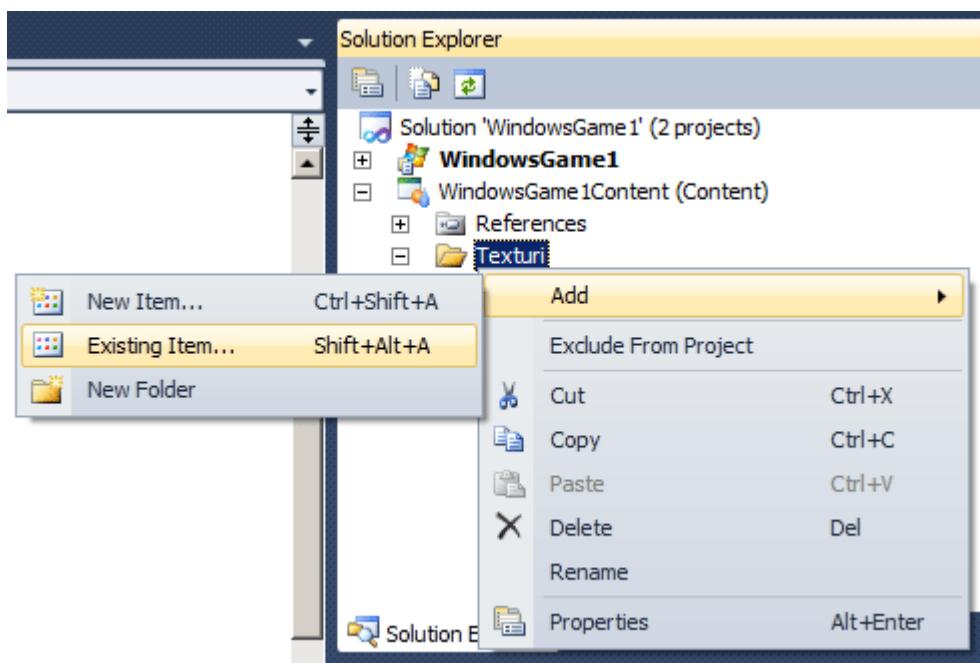


Fig 3.7 - Importarea unui asset existent în proiectul de conținut, pentru a fi procesat de Content Pipeline.

Fișierul nostru folosit pentru textură are numele **avion.png**, de aceea XNA va denumi

asset-ul nostru ca „avion”, urmând ca să îl încărcăm în joc folosind calea „Texturi/avion”.

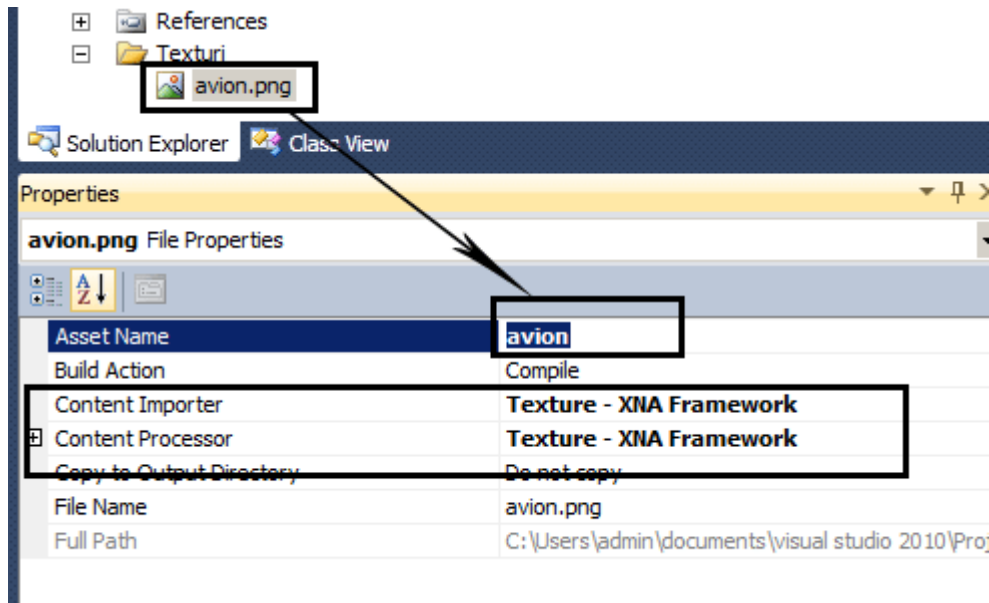


Fig 3.7 – Selectând assetul dorit, putem vedea în secțiunea „Properties” numele acestuia, precum și Content Importer și Content Processor-ul setat acestui asset.

- Vom încărca apoi acest asset din ContentManager, și îl vom atribui texturii noastre definite mai sus (`Texture2D textura`). Conținutul îl vom încărca în metoda `LoadContent`.

```
protected override void LoadContent()
{
    // Create a new SpriteBatch, which can be used to draw textures.
    spriteBatch = new SpriteBatch(GraphicsDevice);
    //incarcam textura noastra
    textura = Content.Load<Texture2D>("Texturi/avion");
}
```

- Vom adăuga codul pentru preluarea stării tastaturii și pentru deplasarea avionului atunci când jucătorul apasă tastele săgeți de pe tastatură (fig 3.8).

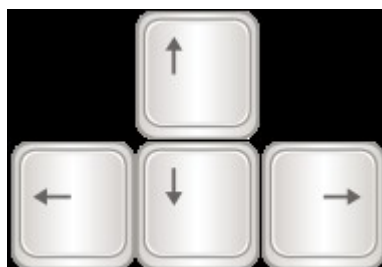


Fig 3.8 – Tastele săgeți de pe tastatură

Logica jocului vine plasată în metoda `Update`, așa că aici vom plasa și noi prelucrarea input-ului.

```
protected override void Update(GameTime gameTime)
{
```

```

// Allows the game to exit
if (GamePad.GetState(PlayerIndex.One).Buttons.Back == ButtonState.Pressed)
    this.Exit();
// preluam starea tastaturii si miscam avionul in functie de tastele apasate
keyboardState = Keyboard.GetState();
if (keyboardState.IsKeyDown(Keys.Left))
    pozitia.X -= 40f * (float)gameTime.ElapsedGameTime.TotalSeconds;
if (keyboardState.IsKeyDown(Keys.Right))
    pozitia.X += 40f * (float)gameTime.ElapsedGameTime.TotalSeconds;
if (keyboardState.IsKeyDown(Keys.Up))
    pozitia.Y -= 40f * (float)gameTime.ElapsedGameTime.TotalSeconds;
if (keyboardState.IsKeyDown(Keys.Down))
    pozitia.Y += 40f * (float)gameTime.ElapsedGameTime.TotalSeconds;

// apelam apoi Update pentru toate componentele
base.Update(gameTime);
}

```

Toată acțiunea unui joc este dependentă de timp. Într-un final scopul nostru este să desenăm acțiunea pe ecran într-un interval cât mai rapid, sau într-un interval destul de rapid încât să nu pară că jocul nostru stagnează în unele părți. Apelurile `Draw`, care desenează totul pe ecran, nu se apelează la intervale fixe de timp (deși XNA ne oferă și această posibilitate), de aceea mișcarea obiectelor în joc trebuie să fie dependentă de aceste variații de timp. Putem prelua diferențele de timp între apeluri succesive ale metodei `Update` și `Draw` folosind proprietăți ale variabilei `gameTime` pe care cele 2 metode o primesc ca parametru. În exemplul de mai sus deplasarea de 40 pixeli a avionului depinde de timpul în secunde a apelului metodei `Draw/Update` precedente (preluată în `gameTime.ElapsedGameTime.TotalSeconds`), astfel avionul se va deplasa cu 40 de pixeli pe secundă, indiferent de rata de desenare a ecranului (FPS). La un FPS fix, de 60 frame-uri pe secundă, de exemplu, `gameTime.ElapsedGameTime.TotalSeconds` va returna întotdeauna valoarea `0.1666666f` ($1 \text{ secunda} / 60 \text{ fps} = 0.1666666f$)

Desenarea avionului la noua poziție:

```

protected override void Draw(GameTime gameTime)
{
    GraphicsDevice.Clear(Color.CornflowerBlue);

    // apelurile de desenare se fac intre Begin() si End()
    spriteBatch.Begin();
    spriteBatch.Draw(textura, pozitia, Color.White);
    spriteBatch.End();

    base.Draw(gameTime);
}

```

În acest caz desenăm `textura` la noua poziție, folosind culoarea albă pentru nuanță. Nuanța este amestecul dintre o culoare și alb, având scopul de a mări luminozitatea acelei

culori. În cazul nostru nuanțăm textura cu alb, deci, afișăm de fapt textura în culorile ei originale. Mai jos puteți vedea exemple de nuanțări ale aceleiași texturi folosind culori diferite (fig 3.9).



Fig 3.9 – Textura avionului, cu nuanță alb (culori originale), cu nuanță roșie, cu nuanță verde, cu nuanță galbenă și cu nuanță albastru deschis (de la stânga la dreapta)

4. JOCUL DE BLACKJACK

Jocul ales pentru implementare este un joc de cărți numit BlackJack, sau 21. Jucătorul joacă împotriva calculatorului, iar scopul jocului este adunarea unei mâini de cărți care să totalizeze cel mult 21 de puncte și să fie mai mare decât totalul mâinii de cărți adunate de calculator.

Jocul are 2 versiuni, una pentru PC și una pentru Windows Phone 7 (testat pe emulator), de aceea am creat 2 soluții Visual Studio pentru joc, **BlackjackWindows.sln** și **BlackjackWindowsPhone.sln**.

4.1. Structurarea proiectului

Legat de partea de cod, există 2 modalități de a structura codul pentru un proiect multi platformă:

1. Scriem din start cod diferit pentru fiecare platformă în parte, luând în considerare configurarea hardware pe care acestea o au și funcționalitățile la care avem acces pe fiecare platformă în parte.
2. Folosim același cod (în mare parte) pentru toate platformele, urmând a optimiza unele rutine pentru fiecare platformă în parte.

În cazul nostru vom folosi cea de-a doua metodă, dezvoltarea multi platformă fiind principalul scop al librăriei XNA, datorită accesului similar la resurse pe toate platformele.

Dezvoltarea folosind cod shared poate fi împărțită în 2 categorii:

1. dezvoltarea folosind librării de joc (XNA le numește **Game Library**⁸, și ca funcționalitate, sunt similare DLL-urilor din Windows),
2. dezvoltarea folosind directive preprocesor, verificând astfel pe ce platformă compilăm jocul. (condiții `#if Windows`, `#if WindowsPhone`, `#if XBOX`)

Proiectul nostru nu este unul complex, așa că vom folosi directivele preprocesor pentru a verifica platforma pentru care compilăm jocul, situațiile în care vom verifica platforma curentă fiind de altfel puține (în constructorul jocului, pentru a seta rezoluția, la desenarea decorului, pentru a verifica suportul pentru shadere)

4.2. Metode de input

Rulând pe platforme diferite, jocul nostru va avea acces la input-uri diferite. Atunci când avem diferite metode de acces la controlul jocului, trebuie să fim atenți să oferim același

⁸ Crearea unui joc Windows sau al unui proiect Librărie (Game Library) - <http://msdn.microsoft.com/en-us/library/bb203928.aspx>

mod familiar de control al jocului și același gameplay, pentru ca jucătorul să nu fie nevoit să învețe taste noi, sau să piardă timp configurând butoane, când trece de la o platformă la alta.

De exemplu, un joc de avioane ar putea fi controlat de săgeți, pe versiunea de PC, iar folosind tasta Control ar trage cu gloanțe în inamici. Pe versiunea de Windows Phone însă, deplasarea avionului ar fi controlată de accelerometru⁹, iar avionul ar trage tot timpul, constant, fără a fi nevoiți să apăsăm o tastă pentru asta. În cazul acesta, folosirea accelerometrului și apăsarea ecranului ar fi prea dificil de făcut în același timp, ar distra jucătorul de la joc și acesta s-ar plictisi repede de creația noastră.

Diferitele metode de input pe numeroasele platforme existente trebuie folosite astfel ingenios, folosirea lor să vină într-un fel natural pentru jucător.



Fig 4.1 – Input-uri disponibile pentru PC și Windows Phone

4.3. Structura codului

Ambele soluții, pentru PC și Windows Phone, folosesc același cod, însă au proiecte de conținut diferite. Folosirea de proiecte diferite de conținut e logică, deoarece în general, pentru versiunea de telefon vom avea nevoie de mai puține asset-uri, sau de asset-uri la dimensiuni mult mai mici deoarece rezoluția telefonului este de 800x600, iar suportul pentru shading pe cele 2 dispozitive diferă. Windows Phone 7 nu suportă shading programabile¹⁰, și

9 Preluarea mișcării folosind accelerometrul - http://en.wikipedia.org/wiki/Accelerometer#Motion_input

10 Lista de suport a shaderelor built-in pentru Windows Phone 7 -

vine cu 4 shadere built-in: **SkinnedEffect**, **EnvironmentMapEffect**, **DualTextureEffect**, **AlphaTestEffect**

În jocul de BlackJack folosim un shader programabil pentru versiunea de PC, celelalte asset-uri fiind comune ambelor proiecte, deși sunt grupate în proiecte separate.

4.3.1. Directoarele de cod sursă

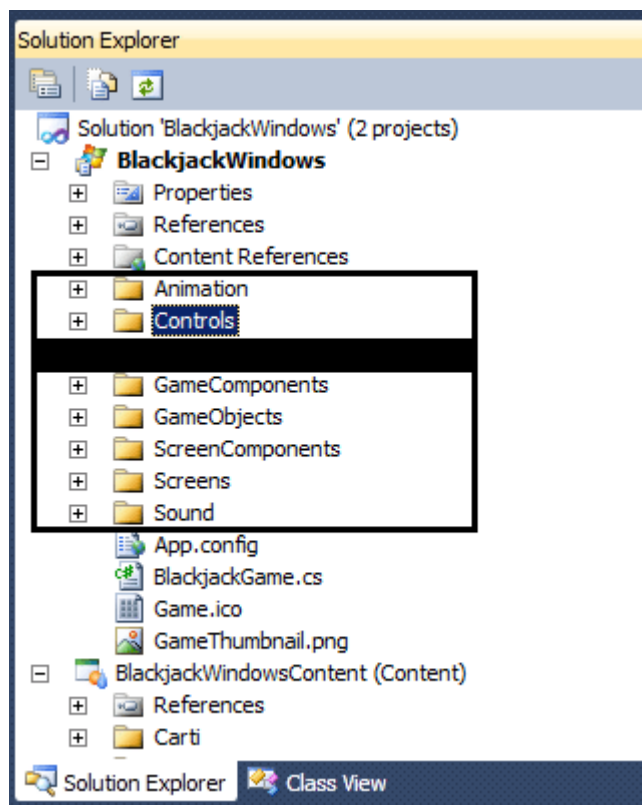


Fig 4.2 – Codul structurat pe directoare

Directorul **Animation** conține 3 clase:

1. `Animation.cs` – Conține clasa de bază pentru animații. Momentan este folosită doar pentru animațiile textului.
2. `AnimationText.cs` – Extinde clasa `Animation` și adaugă suport pentru animarea textului. Textul animat este folosit la afișarea sumelor pierdute sau câștigate de jucător.

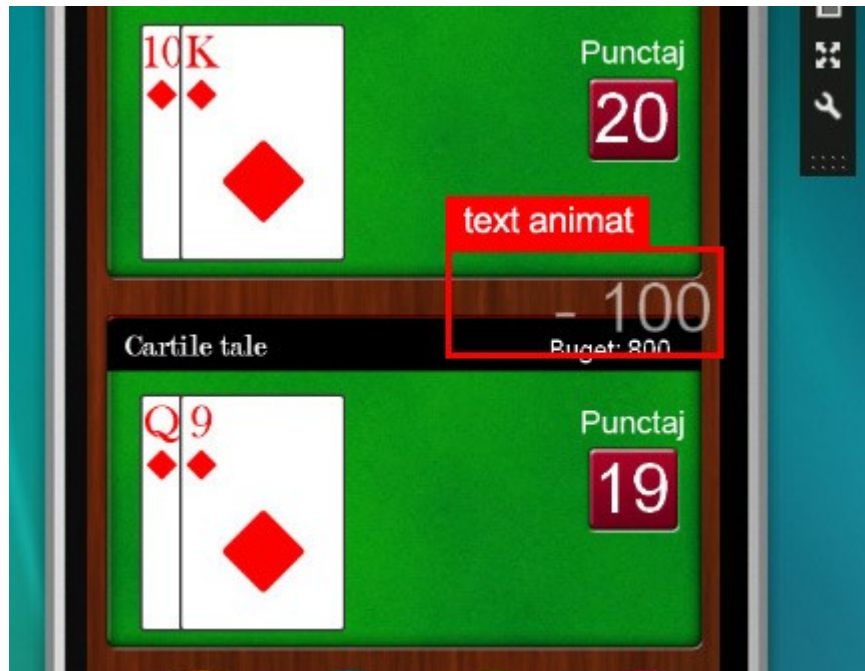


Fig 4.3 – instanță a clasei AnimationText în acțiune

3. `AnimationManager.cs` – Clasa se ocupă cu prelucrarea tuturor animațiilor curente. Fiecare animație nouă este adăugată într-o listă fiind prelucrată și distrusă ulterior.

Directorul **Controls** conține 4 clase:

1. `Button.cs` – Clasă abstractă, este folosită de butoanele cu care jucătorul interacționează pe masa de joc (jetoanele de pariu, butoanele de pariu).
2. `GameButton.cs` – Extinde clasa `Button`, adăugând suport pentru efectul de `hover` (schimbarea imaginii când mouse-ul este deasupra butonului).
3. `TokenButton.cs` – Extinde clasa `Button`, adăugând suport pentru opțiunea `checked` (dacă butonul este bifat sau nu) și suport pentru valoarea pe care jucătorul dorește să o parieze.



Fig 4.4 – instanțe ale claselor GameButton și TokenButton

4. `ButtonManager.cs` – Clasa conține o listă de butoane pe care le administrează și desenează pe ecran. Toate butoanele (`GameButton` și `TokenButton`) aparțin unui manager de butoane, care la rândul lui aparține unui `GameScreen`; clasa `GameScreen` va fi prezentată în curând.

Directorul **GameComponents** conține 2 clase:

1. `Input.cs` – Clasa `Input` este atât o componentă XNA pentru accesarea metodelor de input, cât și un serviciu (`Game Service`¹¹) ce permite accesul ușor la input de către celelalte componente XNA (prin componente înțelegem instanțe ale clasei `GameComponent` sau `DrawableGameComponent`) sau celelalte clase ale jocului. Toate componentele scrise ca și servicii trebuie să extindă și o interfață, ce definește metodele accesibile din acea componentă. Clasa `input` extinde clasa `GameComponents`, și implementează metodele interfeței `IInputHandler`.

```
public interface IInputHandler
{
    KeyboardState KeyboardState { get; }
    GamePadState GamePadState { get; }
    MouseState MouseState { get; }
    bool KeyHold(Keys key);
    bool KeyPressed(Keys key);
    bool MouseLeftClick();
    bool MouseInRectangle(Rectangle rect);
    bool IsMenuUp();
    bool IsMenuDown();
    bool IsMenuLeft();
}
```

¹¹ Componente și Servicii XNA - <http://www.nuclex.org/articles/4-architecture/6-game-components-and-game-services>

```

        bool IsMenuRight();
        bool IsMenuSelect();
        bool IsMenuCancel();
        bool IsPauseGame();
        bool AreaClicked(Rectangle area);
        Vector2 MousePosition { get; }
};

```

Înregistrarea unei componente ca și serviciu se face în constructorul componentei, adăugând componenta respectivă la lista de servicii curente ale jocului și definind interfața prin care componenta poate fi accesată.

```

public Input(Game game)
    : base(game)
{
    // inregistram clasa ca si un serviciu, pe interfata IInputHandler
    Game.Services.AddService(typeof(IInputHandler), this);
}

```

2. `TextureManager.cs` – Asemeni clasei `Input`, clasa `TextureManager` este atât o componentă XNA cât și un serviciu, extinzând clasa `GameComponent` și implementând metodele interfeței `ITextureManager`.

```

public interface ITextureManager
{
    void AddTexture(string textureName, string texturePath);
    void RemoveTexture(string textureName);
    Texture2D GetTexture(string textureName);
};

```

Clasa se ocupă cu stocarea texturilor folosite, pentru ca acestea să poată fi referite dintr-un singur loc; astfel prevenim și încărcarea de mai multe ori a aceleiași texturi. Atât componenta `Input` cât și `TextureManager` nu se ocupă cu desenarea pe ecran a vreunei informații, de aceea extind doar clasa `GameComponent`, și nu `DrawableGameComponent`.

Adăugarea acestor componente la joc o facem în constructorul jocului (`BlackjackGame.cs` – linia 70).

```

input = new Input(this);
Components.Add(input);

textureManager = new TextureManager(this);
Components.Add(textureManager);

```

Directorul **GameObjects** conține 6 clase:

1. `Card.cs` – Pentru jocul 21 folosim un pachet de cărți anglo-american¹² standard, cu 52 de cărți în total. Cărțile pot fi de 4 tipuri: inimă roșie, romb, inimă neagră și treflă. Clasa `Card` stochează tipul și valoarea uneia dintre cele 52 de cărți

¹² Istoria cărților de joc - http://en.wikipedia.org/wiki/Playing_card

posibile.

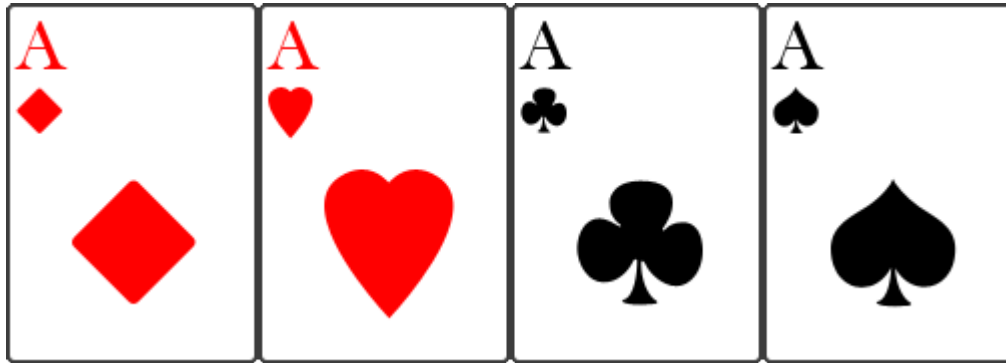


Fig 4.5 – Exemplu de carte de tip romb, inimă roșie, treflă și inimă neagră (de la stânga la dreapta).

2. `CardDeck.cs` – Fiecare carte aparține unui singur pachet de cărți. Un pachet de cărți conține în total 52 de cărți, câte 13 de fiecare tip. Clasa `CardDeck` conține o listă cu cărțile dintr-un pachet, și o metodă `Shuffle` care se ocupă cu amestecarea cărților.
3. `GameState.cs` – Momentan fișierul `GameState.cs` conține doar o enumerare cu rezultatul mâinii curente, cine a pierdut, cine a câștigat sau dacă a fost egal.

```
/// structura tine minte rezultatul rundei curente. cine a castigat sau a
pierdut
public enum GameResult
{
    PlaceBet, WaitingInput, Playing, DealerBlackJack, PlayerBlackJack, DealerBust,
    PlayerBust, DealerWin, PlayerWin, Push
}
```

4. `Hand.cs` – clasa `Hand` stochează lista de cărți extrase de un jucător (fie el dealerul sau jucătorul principal).
`protected List<Card> cards;`
5. `Player.cs` – Clasa `Player` reprezintă un jucător, fie el dealer, jucătorul principal sau alți jucători (în cazul în care jocul va fi implementat cu suport multiplayer). Fiecare jucător are o referință la pachetul principal de cărți (clasa `CardDeck`), din care vor fi extrase cărți pentru mâna curentă (clasa `Hand`) a jucătorului.
6. `CardGame.cs` – Cea mai importantă clasă a jocului, se ocupă cu inițializarea meselor de joc, cu crearea celor doi jucători (dealer și jucătorul principal), cu determinarea rezultatului rundei curente și cu afișarea acestor date pe ecran.

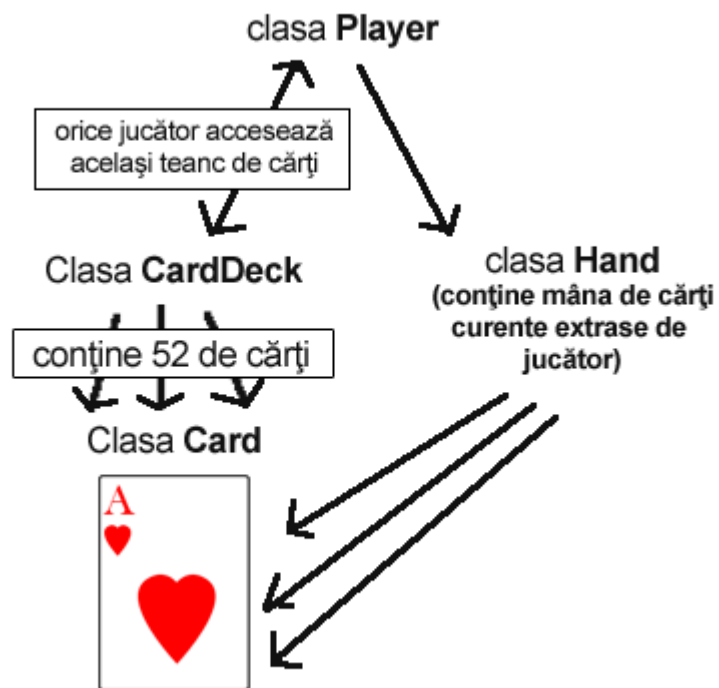


Fig 4.6 – Relațiile între clasele din directorul GameObjects.

Directorul **ScreenComponents** conține 2 clase:

1. `GameScreen.cs` – implementează clasa abstractă `GameScreen`, ce stă la baza fiecărui ecran pe care facem desenarea jocului sau a meniurilor. Implementarea este o modificare a soluției **Game State Management**¹³ de pe creators.xna.com
2. `ScreenManager.cs` – conține clasa `ScreenManager` ce se ocupă cu administrarea tuturor ecranelor existente. Clasa este o componentă XNA și deoarece se ocupă cu desenarea ecranelor curente, extinde clasa `DrawableGameComponent`. Variabila `List<GameScreen> screens` stochează lista curentă de ecrane ce trebuie desenată, astfel, meniul principal este desenat pe un ecran, căsuțele modale de dialog aparțin propriilor ecrane (stocate asemeni unei stive), iar jocul principal este desenat pe propriul lui ecran. Folosirea de ecrane pentru fiecare parte a jocului permite folosirea de efecte de tranziție la trecerea de la un ecran la altul.

Directorul **Screens** conține 7 clase, 6 din cele 7 clase extind clasa abstractă `GameScreen` și sunt administrate de o instanță a clasei `ScreenManager`.

¹³ Soluția originală *Game State Management*, pentru XNA - <http://creators.xna.com/en-us/samples/gamestatemanagement>

1. `BackgroundScreen.cs` – Pe Windows, acest ecran încarcă un shader (*Whirl.fx*) ce creează un efect similar apei pe textura folosită ca decor. Shaderul definește o funcție HLSL pentru pixel shader, versiunea 2.0. Pe Windows Phone, clasa `BackgroundScreen` desenează textura de decor fără nici un efect (Windows Phone nu suportă shadere personalizate).
2. `GameplayScreen.cs` – clasa ce se ocupă cu rularea jocului în sine (creează o instanță a clasei `CardGame`, prezentată mai sus).
3. `MainMenuScreen.cs` – după cum sugerează și numele, clasa `MainMenuScreen` este o clasă ce extinde `MenuScreen` (ce extinde la rândul ei clasa `GameScreen`) și se ocupă cu afișarea și prelucrarea meniului principal. Pe Windows, meniul principal trebuie centrat în funcție de rezoluția curentă. Pe Windows Phone, din moment ce lucrăm la rezoluție fixă, putem plasa meniul la o zonă absolută.

```
#if WINDOWS
    Position = new Vector2((ScreenManager.GraphicsDevice.Viewport.Width - 180)
/ 2, 240);
#endif
```



Fig 4.7 - Ecranul MainMenuScreen în acțiune

4. `MenuEntry.cs` – Fiecare ecran ce conține un meniu, creează o listă de instanțe ale clasei `MenuEntry`. Fiecare buton din meniul principal sau din căsuțele de dialog este o instanță a clasei `MenuEntry`.
5. `MenuScreen.cs` – clasa extinde `GameScreen` și este creată pentru a fi folosită la crearea oricărui tip de ecran ce conține meniuri, cu 1 sau mai multe elemente. Elementele pot fi accesate atât folosind tastatura, cât și mouse-ul, respectiv

touchpad-ul.

6. `MessageBoxScreen.cs` – clasa `MessageBoxScreen` este folosită pentru orice ecran ce are de afișat o căsuță de dialog. Tipurile de căsuțe de dialog suportate momentan sunt cele ce afișează informația și un buton Ok, și cele de confirmare simple, cu 2 butoane, Ok și Anulare.

```
public enum MessageBoxType
{
    MessageBoxOk, MessageBoxYesNo
}
```

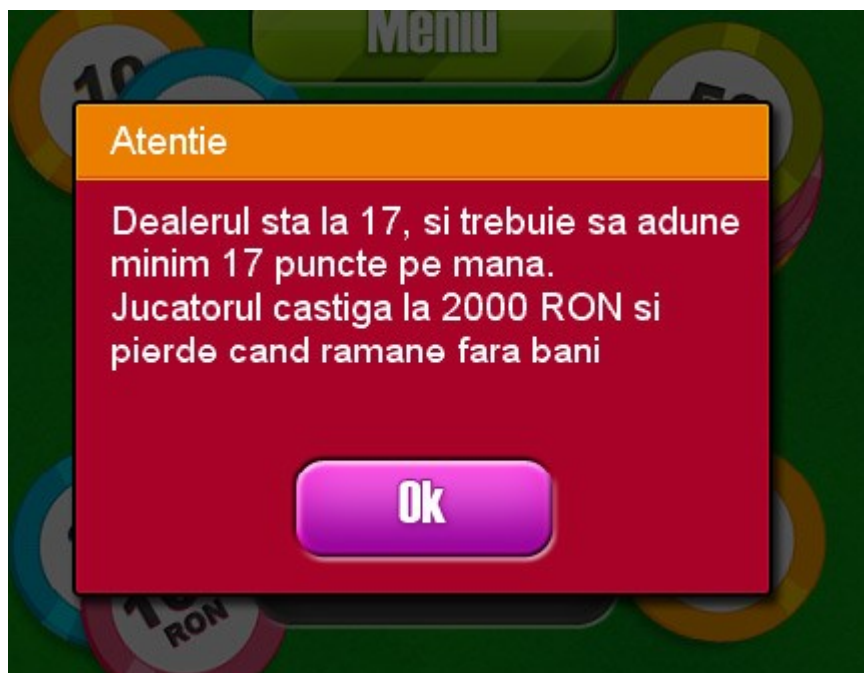


Fig 4.8 – Căsuță de mesaj de tip `MessageBoxOk`

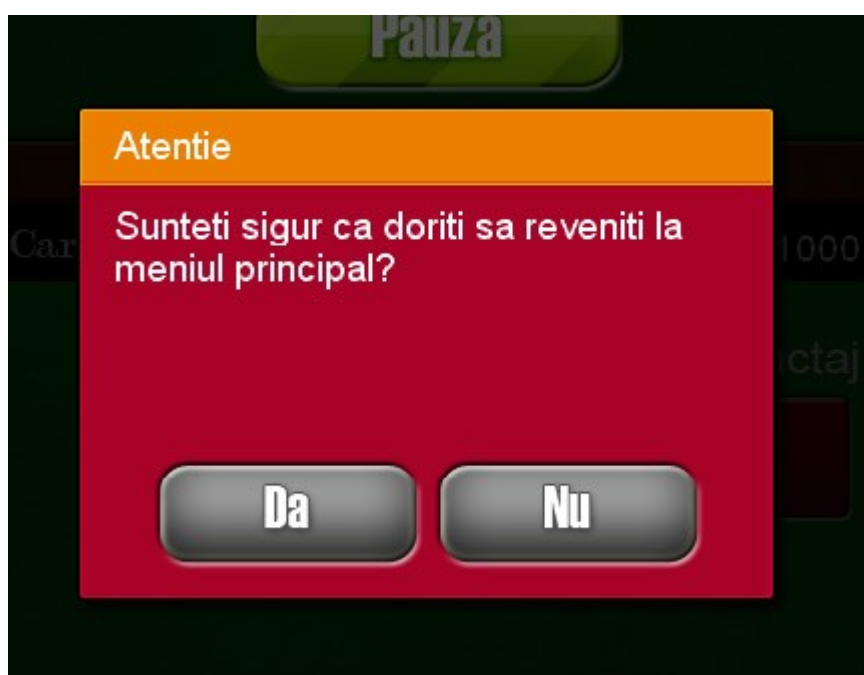


Fig 4.9 – Căsuță de mesaj de tip `MessageBoxYesNo`

Ecranul pentru căsuțele de dialog conține o variabilă de tip `ButtonManager` (vezi clasa `ButtonManager` mai sus) ce se ocupă cu administrarea butoanelor create pe acestea.

7. `PauseMenuScreen.cs` – Clasa `PauseMenuScreen` extinde clasa `MenuScreen` și afișează meniul de pauză când jucătorul apasă tasta **ESC** sau când face click pe butonul **X** în timpul jocului.



Fig 4.10 – Ecranul `PauseMenuScreen`, cu 2 `MenuEntry`, „Rezumare joc” și „Meniul Principal”.

Adăugarea celor 2 butoane, și atașarea unor funcții pe evenimentul `onclick`:

```
MenuEntry resumeGameMenuEntry = new
MenuEntry(ScreenManager.TextureManager.GetTexture("meniuRezumare"), this);
MenuEntry quitGameMenuEntry = new
MenuEntry(ScreenManager.TextureManager.GetTexture("meniuPrincipal"), this);

resumeGameMenuEntry.Selected += OnCancel;
quitGameMenuEntry.Selected += QuitGameSelected;

MenuEntries.Add(resumeGameMenuEntry);
MenuEntries.Add(quitGameMenuEntry);
```

Directorul **Sound** conține o singură clasă:

1. `SoundManager.cs` – Clasa `SoundManager` este o clasă minimă pentru stocarea și prelucrarea sunetelor. Pentru prelucrarea avansată a sunetului, XNA vine cu proiecte XACT¹⁴, ce suportă crearea de liste de sunete și aplicarea de diferite efecte pe acestea. Din păcate, XNA pentru Windows Phone nu suportă proiecte XACT ci doar simple sunete încărcate prin clasa `SoundEffect`. Clasa noastră poate stoca și rula astfel de sunete, iar din moment ce pentru acest joc

¹⁴ Crearea sunetelor cu XNA Game Studio - <http://msdn.microsoft.com/en-us/library/bb203895.aspx>

avem nevoie doar de sunete simple, vom folosi această clasă pentru ambele proiecte, atât pe PC cât și pe Windows Phone.

4.3.2. Locul unde se întâmplă magia

Fișierul de intrare în aplicație este `BlackjackGame.cs`. Aici are loc inițializarea și setarea dispozitivului grafic (accesarea plăcii video, crearea contextului pentru desenare), inițializarea componentelor folosite de joc și crearea managerului de texturi și de ecrane (`screenManager`).

Codul este destul de sumar deoarece logica jocului are loc în componentele XNA create sau în diferitele ecrane care sunt active la momentul respectiv, și pe care le-am prezentat în rândurile precedente. Constructorul setează rezoluția jocului în funcție de dispozitiv (800x480 pe Windows Phone), apoi sunt create cele 3 componente XNA folosite de joc, `input` pentru preluarea dispozitivelor de intrare, `textureManager` pentru stocarea texturilor folosite în joc și `screenManager` pentru prelucrarea ecranelor active în joc; `input` și `textureManager` sunt înregistrate și ca servicii (Game Services) pentru a putea fi accesate cu ușurință de orice clasă (ce are acces la instanța principală a `Microsoft.Xna.Framework.Game`) sau componentă XNA.

```
public BlackjackGame()
{
    graphics = new GraphicsDeviceManager(this);
    graphics.PreferredBackBufferWidth = 1024;
    graphics.PreferredBackBufferHeight = 768;
    graphics.ApplyChanges();

    Content.RootDirectory = "Content";
    IsMouseVisible = true;

    // vrem ca jocul pe winphone sa ruleze fullscreen, iar pe windows intr-o fereastră
    simulata
    #if WINDOWS_PHONE
        IsMouseVisible = false;
        graphics.IsFullScreen = true;
        graphics.PreferredBackBufferWidth = 480;
        graphics.PreferredBackBufferHeight = 800;
        graphics.ApplyChanges();
    #endif

    // Frame rate este 30 fps standard pe Windows Phone.
    TargetElapsedTime = TimeSpan.FromSeconds(1 / 30.0);

    input = new Input(this);
    Components.Add(input);

    textureManager = new TextureManager(this);
    Components.Add(textureManager);

    screenManager = new ScreenManager(this);
    Components.Add(screenManager);
}
```

Are loc apoi apelul funcției `Initialize()` ce se ocupă cu încărcarea texturilor, încărcarea sunetelor și crearea primelor ecrane de joc, ecranul pentru imaginea/imaginile de decor – `BackgroundScreen` – și ecranul pentru meniul principal – `MainMenuScreen`.

Urmează apoi apelul metodei `Initialize()` pentru toate componentele XNA atașate jocului: `base.Initialize()`

```
protected override void Initialize()
{
    // incarcam toate texturile
    textureManager.AddTexture("decor", "Carti\Masa\decor");
    textureManager.AddTexture("masaDealer", "Carti\Masa\masaDealer");
    textureManager.AddTexture("masaJucator", "Carti\Masa\masaJucator");
    // incarcam texturile pentru toate cartile
    for (int cardTypeValue = 0; cardTypeValue < 4; cardTypeValue++)
    {
        CardType cardType = (CardType)cardTypeValue;
        for (int cardFaceValue = 2; cardFaceValue < 14; cardFaceValue++)
        {
            CardFace cardFace = (CardFace)cardFaceValue;
            textureManager.AddTexture(cardType.ToString() +
cardFace.ToString(), "Carti\\" + cardType.ToString() + "\\" + cardType.ToString() +
"- " + cardFace.ToString());
        }
    }
    textureManager.AddTexture("carteIntoarsa", "Carti\Spate\carteIntoarsa");
    textureManager.AddTexture("casutaMesaj", "Controale\casutaMesaj");
    textureManager.AddTexture("meniuStart", "Controale\meniuStart");
    textureManager.AddTexture("meniuOk", "Controale\meniuOk");
    // numeroase apeluri de incarcare texturi ..

    // incarcam apoi si sunetele
    screenManager.AddSound("sunetStart",
Content.Load<SoundEffect>("Sunete\start"));
    screenManager.AddSound("sunetArataCartile",
Content.Load<SoundEffect>("Sunete\arataCarti"));
    screenManager.AddSound("sunetButon",
Content.Load<SoundEffect>("Sunete\buton"));
    screenManager.AddSound("sunetJeton",
Content.Load<SoundEffect>("Sunete\jeton"));

    screenManager.AddScreen(new BackgroundScreen());
    screenManager.AddScreen(new MainMenuScreen());

    base.Initialize();
}
```

Metoda de desenare nu face nimic important, ci doar șterge ecranul și apoi apelează metoda `Draw` pentru toate componentele atașate jocului. Desenarea acțiunii curente este făcută de ecrane/ecranul care sunt active momentan, deoarece `screenManager`-ul este o componentă de tip `DrawableGameComponent`. În cazul nostru, deoarece în metoda `Initialize` am adăugat 2 ecrane în lista de `GameScreens`, vom avea desenat decorul, apoi peste decor meniul principal. În momentul în care jucătorul apelează o opțiune din meniu, un alt ecran va lua locul meniului principal.

```
protected override void Draw(GameTime gameTime)
{
    GraphicsDevice.Clear(Color.CornflowerBlue);

    base.Draw(gameTime);
}
```

Punctul de intrare în joc, pe versiunea de PC și XBOX 360, este prezentat mai jos.

```
#if WINDOWS || XBOX
static class Program
{
    /// <summary>
    /// Punctul principal de intrare in program
    /// </summary>
    static void Main(string[] args)
    {
        using (BlackjackGame game = new BlackjackGame())
        {
            game.Run();
        }
    }
}
#endif
```

5. PLATFORME DIFERITE. DEVICE-URI MULTIPLE

Crearea unui joc simplu și care să ruleze de exemplu pe PC, nu este o sarcină atât de mare. Problema se rezumă în final la optimizarea codului acestui joc, mai ales când el urmează să fie distribuit și în versiunile pentru platforme mobile (Windows Phone sau Zune HD) ce dispun de resurse limitate.

Ne lovim apoi de interactivitate. Pe PC și pe XBOX jucătorul are acces la o sumedenie de butoane și controale. Pe telefon și pe Zune HD va trebui să ne limităm la touchscreen și la accelerometru. Se pune astfel problema păstrării gameplay-ului, folosind input-uri diferite.

Când vorbim de platforme, ne referim la sistemul/sistemele de operare pe care jocul rulează și la librăriile pe care jocul le folosește pe acea platformă. Când vorbim de device-uri, ne referim la capacitățile lor grafice și la metodele disponibile de a prelua input.

5.1. Windows Phone 7

Ce e grozav la Windows Phone 7 și la telefoanele ce vor suporta acest sistem este legat de placa video, toate telefoanele vor include o placă video gpu standard, având aceleași funcționalități și performanțe pe toate telefoanele¹⁵, de la orice producător. Asta se traduce în final în deployment fără probleme și în performanțe similare pe orice telefon WP7.

Arhitectura platformei de aplicații Windows Phone este formată din patru componente principale:

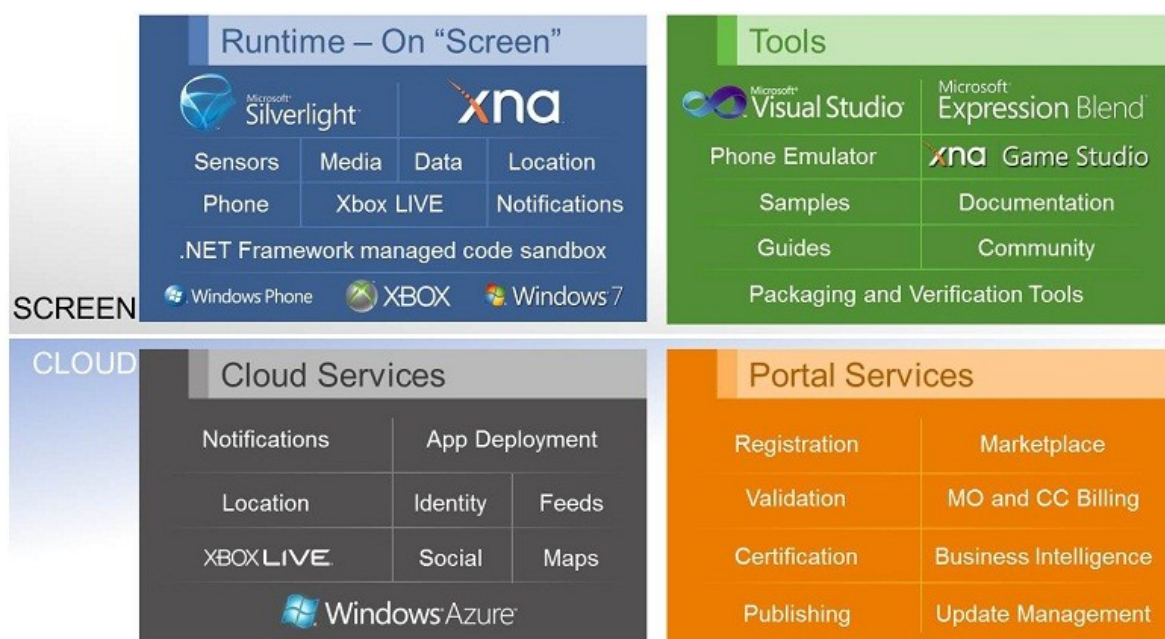


Fig 5.1 - Arhitectura Windows Phone Application Platform

15 Video: GDC 2010 – Interviu Shawn Hargreaves despre performanțele WP7 - http://ecn.channel9.msdn.com/o9/ch9/9/1/8/6/3/5/GDC2010XNA_2MB_ch9.wmv

În rândurile următoare vom dezbate doar componenta **Runtime**, componenta **Unelte**, și anume Visual Studio 2010 IDE și XNA Game Studio fiind dezbătute la începutul acestei documentații.

Dezvoltarea de aplicații grafice pentru Windows Phone 7 se poate face fie folosind Silverlight, fie XNA. Dezvoltarea se face folosind cod managed, în mediu protejat, permițând astfel dezvoltarea rapidă și sigură a aplicațiilor. Datorită structurii XNA, aplicațiile scrise pentru această librărie vor rula pe Windows Phone folosind un număr minim de modificări, de exemplu pentru specificarea rezoluției sau pentru accesul senzorilor specifici device-ului.

Silverlight și XNA, alături de componentele specifice Windows Phone și librăria de clase comune de bază contribuie la un set substanțial de componente ce pot fi folosite de dezvoltatori.

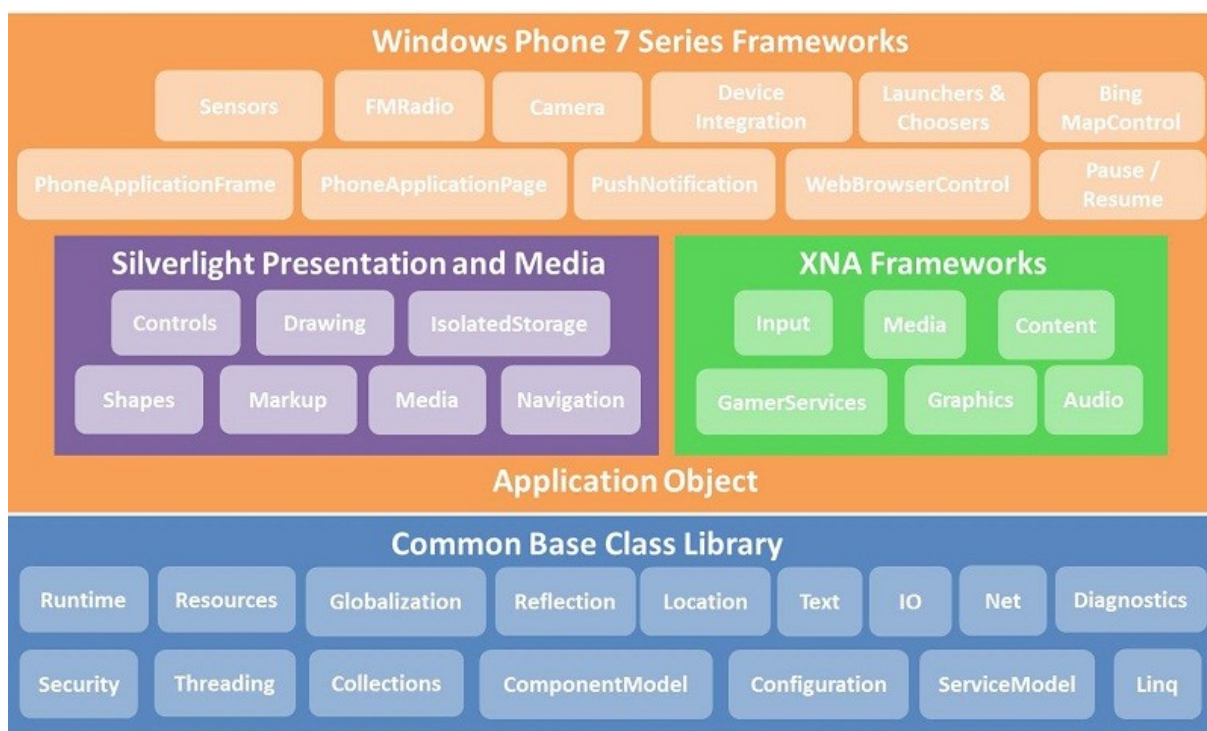


Fig 5.2 – Toate componentele accesibile dezvoltatorilor

Dintre componentele importante XNA pe care nu le-am discutat până acum fac parte:

- **Sensors** – componenta returnează date de la o multitudine de senzori, cum ar fi: preluare date multitouch¹⁶, accelerometru și preluarea datelor de la senzorii microfonului.
- **Media** – XNA permite integrarea și consumarea ușoară a imaginilor, animațiilor și diferitelor tipuri de conținut media (filme, audio). Funcțiile API media permit

¹⁶ Returnare informații de la dispozitive de intrare - <http://msdn.microsoft.com/en-us/library/bb203899%28XNAGameStudio.40%29.aspx>

descoperirea și listarea media de pe dispozitivul curent și bineînțeles prelucrarea datelor media (astfel poți rula în joc melodiile pe care le ai stocate în librăria Media de pe PC, XBOX sau telefon, sau poți include filme pe care tu le crezi).

- **Location** – Serviciul de Localizare Microsoft pentru Windows Phone permite dezvoltatorilor să acceseze locația fizică a utilizatorului telefonului folosind API-ul de localizare. Se pot accesa datele locației curente, setarea acurateții acestor date, se poate accesa direcția și viteza de deplasare și se poate calcula distanța dintre diferite puncte.

Senzorii și localizarea sunt importanți deoarece aceste date pot fi folosite la crearea unor jocuri inovative, folosind microfonul putem crea jocuri/aplicații ce se bazează pe recunoașterea vocală, folosind localizarea putem crea de exemplu o simplă aplicație ce ne oferă informațiile importante despre împrejurimi sau în cazul unui joc, am putea încărca hărți diferite specifice zonei în care ne aflăm, dacă acestea sunt disponibile (imaginați-vă că jucați Quake 2 și odată ajunși în Alba Iulia vi se schimbă avatarul cu un erou îmbrăcat în tricolor și vă luptați într-o hartă ce surprinde zona catedralei Reîntregirii Neamului¹⁷)

5.2. Platforma PC

Marele avantaj al programării folosind cod managed îl reprezintă faptul că nu trebuie să ne mai facem griji în ceea ce privește pierderile de memorie, în sensul pierderii pointerilor la memoria referită (putem numi totuși „memorie pierdută” variabilele ce nu sunt distruse în timp util, sau handle-urile la un pointer pe care uităm să îl setăm ca nul). Am vorbit la începutul acestei cărți puțin despre diferențele între platforme, în ce privește librăria .NET pe care acestea o au disponibilă. Am văzut că pe PC avem acces la întreaga librărie, iar pe XBOX, telefon și Zune avem acces doar la o variantă compactă a librăriei .NET. Legat de acest lucru, am explicat sumar și faptul că Garbage Collector (memoria fiind administrată), administrează memoria grupată în anumite nivele numite „generații”, care pot fi 3 la număr pe PC, iar pe .NET compact doar într-un singur nivel (obiectele create pe .NET compact au o singură generație).

Pe PC, atunci când creăm obiecte, ele sunt puse în coada managed. .NET calculează memoria necesară pentru obiect și confirmă dacă este destulă memorie liberă în coadă. Este apelat apoi constructorul obiectului și codul rulat returnează o referință la acel obiect (locația din coadă). Memoria este creată în mod contiguu, obiectele fiind plasate în ordinea creării, unul după celălalt.

Dacă memoria nu poate fi alocată pentru un obiect, este executat garbage collector-ul

17 Catedrala Reîntregirii Neamului - http://enciclopediaromaniei.ro/wiki/Catedrala_Re%C3%AEntregirii_Neamului

pentru a elibera memoria nefolosită. Ce memorie eliberăm și în ce situație, ne spune generația în care se află obiectele respective. Prima generație stochează memoria recent adăugată, astfel memoria este alocată când variabilele sunt create și apoi e setată ca și inactivă când variabilele nu mai sunt folosite sau sunt setate explicit la nul. Când setează memoria ca și inactivă, .NET setează defapt rădăcina obiectului (care este un pointer la locația memoriei) la valoarea nul. Când coada de memorie se umple, va rula garbage collector-ul ce va trece toate obiectele active în generația 1, eliberând memoria generației 0 (obiectele setate ca inactive). La umplerea memoriei cu obiecte din generația 1 și 2 va rula același proces, doar că pentru ultima generație, vor trebui verificate toate obiectele din coadă, determinându-se dacă sunt active sau nu, ceea ce se traduce în solicitarea intensă a procesorului, când acesta ar trebui să fie preocupat cu procesarea jocului nostru.

Ideea principală este să fim atenți cum și când creăm obiecte. Acestea nu trebuie să fie foarte mari (în ce privește consumul de memorie) și pe cât posibil să aibă o durată de viață limitată, mai ales pe .NET compact, unde coada de memorie are o singură generație.

5.3. Sugestii de optimizare

5.3.1. Administrarea memoriei

În .NET există 2 tipuri de obiecte: valoare și referință. Exemple de tipuri valoare sunt: enumerările, tipurile întregi (byte, short, int, long), tipurile flotante (single, double, float), tipurile primitive (bool, char) și structurile. Exemple de tipuri referințe sunt: array-urile, excepțiile, atributele, delegates și clasele. Tipurile valoare au stocate informațiile pe stiva thread-ului curent, iar tipurile referință stochează informația pe coada managed. Când transmitem variabile la metode, ca și parametrii, le trimitem prin valoare. Pentru tipurile valoare, asta înseamnă că de fapt trecem o copie a informației de pe stivă, astfel orice schimbare vom face asupra acelei variabile în acea metodă nu va afecta variabila originală din memorie. Când transmitem un tip referință transmitem de fapt o referință la acea variabilă/informație, astfel că modificăm direct variabila originală. Informația originală nu este copiată, noi trimitem doar adresa de memorie unde aceasta poate fi accesată, de aceea obiectele mari ar trebui întotdeauna transmise prin referință, deoarece este mult mai rapid să trimiți o adresă de memorie decât să copiezi un obiect întreg în stivă și apoi să accesezi acel obiect.

Pentru a transmite tipuri valoare (și doar tipuri valoare, deoarece tipurile referință sunt accesate din start prin referință) ca și referință în C# putem folosi cuvântul cheie `ref`¹⁸ și numele variabilei.

18 Transmiterea parametrilor de tip valoare - <http://msdn.microsoft.com/en-us/library/9t0za5es%28VS.80%29.aspx>

```

class PassingValByRef
{
    static void SquareIt(ref int x)
    // The parameter x is passed by reference.
    // Changes to x will affect the original value of x.
    {
        x *= x;
        System.Console.WriteLine("The value inside the method: {0}", x);
    }
    static void Main()
    {
        int n = 5;
        System.Console.WriteLine("The value before calling the method: {0}", n);

        SquareIt(ref n); // Passing the variable by reference.
        System.Console.WriteLine("The value after calling the method: {0}", n);
    }
}

```

5.3.2. Blocarea metodelor virtuale

Metodele și clasele virtuale sunt benefice atunci când vrem să extindem funcționalitatea oferită de acestea, dar aduc cu ele și penalizarea performanțelor deoarece metodele virtuale previn anumite optimizări să nu mai aibă loc în timpul rulării. Metodele virtuale necesită verificări ale tabelii virtuale în timpul rulării, blocând însă aceste clase/metode (în cazul în care nu avem nevoie să le extindem) compilatorul va ști astfel că nimeni nu va avea voie să extindă acea clasă sau metodă și va genera un apel direct al metodei în loc să facă verificarea în tabela de lookup.

Blocarea claselor/metodelor se face folosind cuvântul cheie `sealed`¹⁹.

```

sealed class ClasaMea
{
    public int x;
    public int y;
}

```

5.4. Cazul BlackJack

Am discutat despre structura codului din jocul de BlackJack, dar nu am explicat pe deplin locurile unde am avea nevoie să separăm codul pe platforme. În cazul nostru, operația e mai simplă deoarece discutăm de 2 platforme doar, PC, care putem spune că are acces la resurse nelimitate (comparat cel puțin cu Windows Phone) și telefonul, ce rulează pe o versiune compactă a librăriei .NET și evident, se bazează pe o placă grafică mai slăbuță și resurse limitate de memorie și stocare.

Ca o recapitulare, am explicat la începutul acestei documentații că vom merge pe o abordare folosind exact același cod pentru ambele platforme, dar vom separa codul specific platformelor folosind directive preprocesor (`#if WINDOWS, #if WINDOWS_PHONE`)

¹⁹ Referință MSDN cuvântul cheie `sealed` - <http://msdn.microsoft.com/en-us/library/88c54tsw%28VS.71%29.aspx>

5.4.1. Rezoluția

Vom începe cu constructorul clasei principale din fișierul `BlackjackGame.cs`. Rezoluția telefonului este setată la 480x800 (în poziție verticală), pe când pe PC putem folosi orice rezoluție cu înălțimea mai mare de 760 pixeli (pentru că folosim aceeași poziționare ca pe telefon, iar controalele de pe ecran sunt vizibile până la 760 de pixeli în jos), de aceea pe PC folosim rezoluția de 1024x768, deși putem folosi orice rezoluție mai mare ca aceasta, sau putem rula jocul într-o fereastră chiar la aceleași dimensiuni cu ale telefonului, de 480x800.

Deoarece folosim rezoluții diferite pe cele 2 dispozitive și am hotărât să folosim directive preprocesor, vom include setările pentru Windows Phone 7 în propria lor secțiune:

```
#if WINDOWS_PHONE
    IsMouseVisible = false;
    graphics.IsFullScreen = true;
    graphics.PreferredBackBufferWidth = 480;
    graphics.PreferredBackBufferHeight = 800;
    // Frame rate este 30 fps standard pe Windows Phone.
    TargetElapsedTime = TimeSpan.FromSeconds(1 / 30.0);
    graphics.ApplyChanges();
#endif
```

Standardul FPS pe PC este de 60, dar pe telefon va trebui să ne rezumăm la 30 FPS datorită capacităților dispozitivelor mobile. `TargetElapsedTime` setează valoarea FPS la care dorim să reapelăm metoda `Draw`, și va încerca să redeseneze cadrele jocului la această valoare.

5.4.2. Input

Datorită faptului că dispozitivele au input-uri diferite, vom avea nevoie de directive preprocesor și în clasa de input:

```
#if WINDOWS_PHONE
    /// <summary>
    /// Folosit pentru Windows phone
    /// </summary>
    private TouchCollection touches;
#endif
```

Telefonul cu WP7 include un ecran tactil și de aceea va trebui să adăugăm suport pentru touchscreen doar pentru acesta. Asta înseamnă că va trebui să folosim directive preprocesor și pentru metodele de preluare input din interfața `IInputHandler`:

- În metoda `Update` a componentei `Input`, unde trebuie să preluăm și starea ecranului:

```
#if WINDOWS_PHONE
    // si touchscreenul
    touches = TouchPanel.GetState();
#endif
```

- În metoda `MouseLeftClick` unde verificăm dacă a fost apăsat ecranul sau butonul

stânga de la mouse:

```
#if WINDOWS_PHONE
    return (touches.Count > 0 && touches[0].State ==
TouchLocationState.Released);
#else
    return (mouseState.LeftButton == ButtonState.Pressed &&
previousMouseState.LeftButton == ButtonState.Released);
#endif
```

- În metoda `MousePosition` ce returnează poziția unde facem click sau unde atingem ecranul cu degetul:

```
#if WINDOWS_PHONE
    get
    {
        if (touches.Count > 0)
            return new Vector2(touches[0].Position.X, touches[0].Position.Y);
        else
            return Vector2.Zero;
    }
#else
    get { return new Vector2((float)mouseState.X, (float)mouseState.Y); }
#endif
```

În metoda `MouseInRectangle` ce verifică dacă butonul sau degetul pe ecranul tactil se află într-un anumit dreptunghi:

```
#if WINDOWS_PHONE
    if (touches.Count > 0 && touches[0].State == TouchLocationState.Released)
    {
        Rectangle mouseArea = new Rectangle((int)touches[0].Position.X,
(int)touches[0].Position.Y, 1, 1);
        return rect.Intersects(mouseArea);
    }
    return false;
#else
    Rectangle mouseArea = new Rectangle(mouseState.X, mouseState.Y, 1, 1);
    return rect.Intersects(mouseArea);
#endif
```

Aceste metode au nume generice, ca să poată fi folosite cu orice dispozitiv ce are funcționalitatea similară cu a mouse-ului. Dacă vom avea nevoie să preluăm datele unui laser sau a camerei, în cazul în care vrem să le folosim ca și metode de input le vom putea adăuga la aceste metode.

5.4.3. Shadere

Am discutat inițial sumar despre suportul de shadere pe cele 2 platforme. Dar ce sunt până la urmă shaderele. Un shader este reprezentat de cod asamblare, care poate fi consumat direct de placa grafică. Pe vremuri, exista un set de funcții fixe ce puteau fi apelate pentru a configura setările și proprietățile plăcii video; odată cu apariția plăcilor grafice tot mai avansate, s-a trecut la permiterea scrierii de cod ce să fie executat direct de placa video. Plăcile video au instrucțiuni pentru vertex shaders și pixels shaders. Funcțiile vertex shader

sunt executate pe fiecare vertex din scena curentă, iar funcțiile pixel shader pe fiecare pixel.

Microsoft a creat un limbaj standard high-level numit HLSL²⁰ ce ne permite să scriem cod de limbaj înalt fără să fim nevoiți să utilizăm codul greoi de asamblare pentru a comunica cu placa video.

Pe XNA, codul HLSL se stochează în fișiere cu extensia `.fx`, ce pot fi încărcate direct în Content Pipeline și compilate automat de acesta.

XNA pe Windows Phone nu suportă shadere programabile, de aceea am folosit un simplu shader custom doar pe versiunea de PC, pentru a exemplifica folosirea acestuia.

Fișierul poate fi găsit doar în soluția de Windows, în proiectul de conținut, la `Shadere\Whirl.fx`.

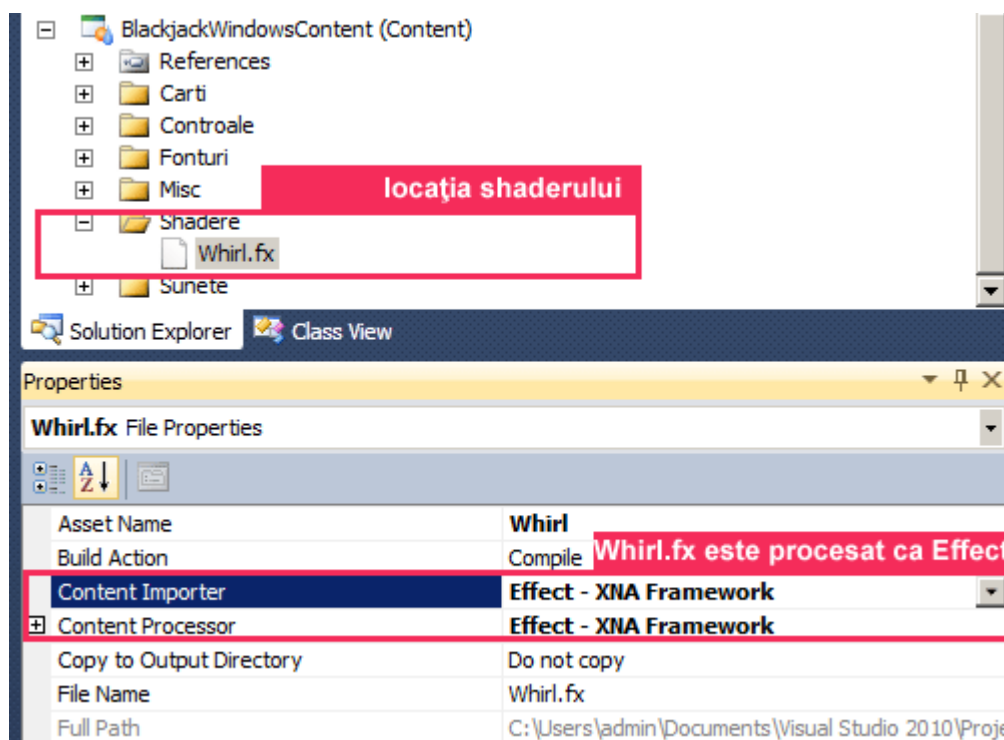


Fig 5.3 – Locația fișierului `fx` și tipul de importator și procesator de conținut atașat acestuia.

Acest shader îl folosim doar pentru ecranul de decor (`BackgroundScreen`) unde creăm un efect similar apei pe textura de lemn din decor. Codul pentru acest shader este următorul:

```
// simplu shader (postprocesare) pentru un efect de apa
sampler ColorMapSampler : register(s0);
float fTimer;

float4 PixelShaderFunction(float2 TextureCoord:TEXCOORD0) : COLOR0
{
    TextureCoord.x += sin(fTimer + TextureCoord.x * 10) * 0.01f;
```

²⁰ Introducere în HLSL făcută de Riemer Grootjans - http://www.riemers.net/eng/Tutorials/XNA/Csharp/Series3/HLSL_introduction.php

```

        TextureCoord.y += cos(fTimer + TextureCoord.y * 10) * 0.01f;

        float4 Color = tex2D(ColorMapSampler, TextureCoord);
        return Color;
    }

    technique PostProcess
    {
        pass P0
        {
            PixelShader = compile ps_2_0 PixelShaderFunction();
        }
    }

```

În exemplu nostru folosim doar un pixel shader, reprezentat de funcția `PixelShaderFunction()`. Jocul fiind 2D, nu avem nevoie de un vertex²¹ shader.

Linia aceasta ne spune să compilăm shaderul pe o placă video ce suportă minim pixel shader (`ps_X_Y`) versiunea 2.0:

```
PixelShader = compile ps_2_0 PixelShaderFunction();
```

Așteptăm de altfel și 1 parametru din partea aplicației, declarat la început de variabila `fTimer` de tip `float`:

```
float fTimer;
```

Tehnica noastră de desenare primește un nume – `PostProcess` – și îi specificăm că are un singur pas de parcurs – `pass P0` – deoarece putem trece o scenă/un model prin mai multe tehnici sau mai mulți pași de procesare.

Pentru a încărca în cod un fișier HLSL și pentru a-l procesa, XNA folosește clasa `Effect`²². Fiind procesat de Content Pipeline, îl vom încărca folosind `ContentManager`. Codul din `BackgroundScreen.cs` arată în felul următor:

```

#if WINDOWS
    /// <summary>
    /// Efectul folosit la postprocesare, disponibil doar pentru PC (si XBOX, daca
    /// voi face o versiune pt consola)
    /// </summary>
    private Effect effect;

    private float effectTimer = 0f;
#endif

```

Inițial declarăm variabila `effect` de tip `Effect`, apoi variabila de tip `float` `effectTimer` pe care o vom transmite variabilei `fTimer` din shaderul de mai sus.

În funcția `LoadContent` pe lângă textura de decor, vom încărca și shaderul pentru

21 un vertex este un punct reprezentat în spațiu 3D de 3 coordonate, X, Y și Z.

22 MSDN – clasa `Effect` - <http://msdn.microsoft.com/en-us/library/microsoft.xna.framework.graphics.effect.aspx>

decor:

```
public override void LoadContent()
{
    texture = ScreenManager.TextureManager.GetTexture("decor");
#if WINDOWS
    effect = ScreenManager.Game.Content.Load<Effect>("Shadere\\Whirl");
#endif
}
```

În funcția `Update` vom prelua timpul trecut între apelurile succesive ale metodei `Update`, și vom folosi valoarea pentru a seta variabila `effectTimer`:

```
public override void Update(GameTime gameTime, bool otherScreenHasFocus, bool
coveredByOtherScreen)
{
#if WINDOWS
    effectTimer += (float)gameTime.ElapsedGameTime.TotalMilliseconds / 500;
#endif
    base.Update(gameTime, otherScreenHasFocus, false);
}
```

Tot ce mai rămâne de făcut este să transmitem valoarea timerului la shader, și să-i spunem obiectului de desenare (`SpriteBatch sb`) să folosească acest efect la procesarea imaginii:

```
public override void Draw(GameTime gameTime)
{
    if (texture == null)
        throw new Exception("e goala");
    SpriteBatch sb = ScreenManager.SpriteBatch;
    Viewport viewport = ScreenManager.GraphicsDevice.Viewport;
    Rectangle fullscreen = new Rectangle(0, 0, viewport.Width,
viewport.Height);
    byte fade = TransitionAlpha;

#if WINDOWS
    effect.Parameters["fTimer"].SetValue(effectTimer);
    sb.Begin(0, BlendState.Opaque, null, null, null, effect);
#else
    sb.Begin();
#endif

    sb.Draw(texture, fullscreen, new Color(fade, fade, fade));
    sb.End();
}
```

Din moment ce aplicăm efectul acesta doar la desenarea decorului, doar imaginea de decor va fi afectată, restul elementelor din joc desenându-se normal. Codul HLSL executându-se direct pe GPU, nu doar că am obținut un efect interesant folosind foarte puțin cod, ci am reușit asta și fără să afectăm performanța jocului.

6. CONCLUZIE

Microsoft XNA este o unealtă grozavă pentru jocuri multi platformă, dezvoltarea de astfel de jocuri nefiind nicicând mai ușoară și mai accesibilă ca acum. Deși printre neajunsuri s-ar putea număra faptul că suportul există doar pentru sistemele Windows și dispozitivele create de Microsoft, este mai mult decât de ajuns pentru studenți și cei pasionați de domeniul dezvoltării jocurilor.

Să poți dezvolta un joc pe care să îl poți transpune pe mai multe dispozitive cu efort minim și în timp cât mai scurt este un lucru ce îți oferă multă satisfacție personală.

Jocul de BlackJack surprinde doar părți din dezvoltarea multiplatformă și nu apucă să dezbată și grafica 3D, elemente avansate de shading, crearea de procesatori și importatori de conținut customizați, dar așterne calea pentru cei ce doresc să înceapă programarea de jocuri simple.

Înainte de toate, un joc este o metodă de entertainment, de aceea în dezvoltarea jocurilor trebuie să punem tot timpul accentul pe elementul distractiv și pe un gameplay inovativ. Având acces la microfon și alte dispozitive (gen camera video) putem crea noi moduri de a interacționa cu dispozitivele respective și mai ales cu jocurile pe care le creăm. Într-un final, singura limită este imaginația fiecărui dezvoltator și cunoștințele acestuia de a pune în practică aceste idei.

7. BIBLIOGRAFIE

Cărți

1. *Beginning XNA 3.0 Game Programming*, Apress, 2009
2. *Learning XNA 3.0: XNA 3.0 Game Development for the PC, Xbox 360, and Zune*, O'Reilly Media, 2009
3. *Microsoft XNA Game Studio 3.0 Unleashed*, Sams Publishing, 2009
4. *XNA 3.0 Game Programming Recipes: A Problem-Solution Approach*, Apress, 2009
5. *Professional XNA Game Programming: For Xbox 360 and Windows*, Wiley Publishing, 2007
6. *Beginning Game Level Design*, Thomson Course Technology, 2005
7. *Game Architecture and Design*, New Riders Publishing, 2004

Referințe internet

8. Carter, C., *XNA Advanced debugging tutorial*, partea 1 și 2
<http://xnaessentials.com/archive/2009/05/06/advanced-debugging-tutorial-part-one.aspx> | <http://xnaessentials.com/archive/2009/07/27/advanced-debugging-part-two.aspx>
9. Grootjans, R., *Tutoriale XNA ce acoperă toate aspectele programării 2D și 3D folosind XNA* <http://www.riemers.net/>
10. Zima, C., *Crash course in HLSL*, http://www.catalinzima.com/?page_id=575
11. *Optimizations: Particles and High-Frequency Code*, http://creators.xna.com/en-US/tutorial/optimization_highfrequency
12. *Best practices for Indie Games 3.1*, <http://creators.xna.com/en-US/education/bestpractices31>
13. *Sprite Sheet – combinarea mai multor imagini într-o singură textură*, <http://creators.xna.com/en-US/sample/spritesheet>
14. *Sprite Effects*, <http://creators.xna.com/en-US/sample/spriteeffects>
15. *Shader Series: Introduction*, http://creators.xna.com/en-US/article/shader_primer
16. *Shader Series: Fixed Function to Programmable Pipeline*, http://creators.xna.com/en-US/article/shader_ff2pp