Ron Penton:

Kezdő C# játékprogramozás

Magyarra fordította: Ipacs Tamás (2017-18-ban)

A könyv fordítójának előszava

Ez egy nem hivatalos fordítása a könyvnek tőlem. Tulajdonképpen magamnak csináltam, hogy elsajátítsam az alapismereteket a C# játékprogramozásról, de szíves-örömest megosztom mással – veled – ezt a fordítást; szabadon letöltheted a honlapomról (http://www.programozzunk.ucoz.hu), de ha van, elhelyezheted a sajátodon is, hiszen minél több helyen van meg, annál többen érhetik el. Át is írhatod, de ezesetben tüntesd fel a (bece)neved, mint kiegészítő, és amit átírtál. Lehet, te nálam jobb fordítást készítesz, hiszen én nem vagyok egy hivatásos szakfordító, és ezért néhol lehetnek eléggé nehezen érthető mondatok a könyvben – elnézést érte. Mindent nem is tudtam lefordítani vagy éppen nem akartam, mert csupán szószaporítás volt. Sajnos a képeket sem tudtam áttenni az eredeti műből, és ha nagyon nem értesz valamit, javaslom, hogy emellett azt is nézd át.

Mindezek ellenére remélem, hogy neked is hasznodra válik a tanulmányozása!

Bevezető

Csupán néhány rövid évvel ezelőtt is, mindenki C-ben programozott játékokat. Afelől nem volt kérdés – ha élvonalbeli játékokat akartál programozni, úgy azt C-ben csináltad. Persze, a C++ táján, de az túl „lassú” volt. A haladó tulajdonságok, amiket a C++ kínált, túl sok számítási teljesítményt igényeltek, és ez egyszerűen elfogadhatatlan volt egy játékprogramozó számára.

Ahogy telt az idő, a számítógépek gyorsabbá és gyorsabbá váltak, a videójátékok pedig nagyobbá és nagyobbá. Hamarosan az emberek ráébredtek, hogy a játékok túl naggyá váltak ahhoz, hogy C-ben íródjanak. Mikor a programok még kicsik voltak, a C egy nagyszerű nyelv volt ennek használatára, mert nem igazán volt szükség különösebb kezelésre a kódodban. Egy ember meg tudott írni egy programot, és könnyen megértett mindent, amit csinált. De a programok nagyobbá válásával a C is egy problémává vált: a benne írt programot túl nehézkes volt kezelni. Nem szándékozom itt kifejteni, hogy miért – ha valaha használtál C-t, akkor tudod.

A C++ megoldott sok C-s problémát, de fenntartva a visszafelé-kompatibilitást a C-nyelvvel, volt egy fő probléma, és ennek következtében a C++ a legnagyobb nyelvi mutációként végződött. Szintén egy nagyszerű nyelv, de igen hosszú hiányosságlista van vele társítva.

A számítógéped gyakran már abban a percben elavul, amikor kisétálsz vele a bolt ajtaján. Fejlesztettem a videókártyám egyszer egy évben; az igazi hardcore játékosok kétszer vagy háromszor is megteszik ugyanezt egy évben! A dolgok már nem olyanok többé. A számítógépem itt ült másfél évig, és én egy új merevlemez beszerelésén kívül nem nyúltam a belsejéhez.

A számítógépek olyan jellemvonást kaptak, ahol elég gyorsak kezelni a legtöbb tőlük elvárható dolgot ésszerű időmennyiségen belül, és igazán nem jár hatalmas előnnyel fejleszteni a számítógépedet azért, hogy futtathasd a legújabb játékokat, mert azok olyan közel vannak a fotorealisztikus minőségű megjelenítéshez, hogy nagyobb előrelépések alig lesznek már.

Nem csoda, hogy a „lassú” nyelvek, mint a C# és mások, amelyek részei a .NET-nek, újra népszerűvé válnak. A kezelt nyelvek, mint a C#, a régieknél sokkal többet felvállalnak, de a védelem tekintetében olyan sokat nyújtanak, hogy statisztikailag sokkal kevesebb az esélye hibák vétésének a programjaidban, mert a nyelv tervezett. Ezek a nyelvek bizonyára több számítási teljesítményt igénybe vesznek, hogy további ellenőrzéseket csináljanak számodra, de az emberek megértik, hogy ez végül is megéri, mert lehetővé teszi számodra, hogy kevesebb idő alatt készíts játékokat, kis hibák miatti aggódás nélkül.

Kinek íródott ez a könyv

Ez a könyv mindazoknak szól, akik meg akarják tanulni, hogy hogyan programozzanak C#-ben és DirectX 9-ben. A könyv elolvasásához nem szükséges, hogy legyen bármilyen szintű C# ismereted, de némi programozási háttér (bármilyen nyelven) hasznos lehet.

Továbbá, nem kell, hogy elmenj és vásárolj bármilyen eszközt a C#-ban való elmélyedéshez, mert minden elérhető ingyen, ami kell hozzá!

Ez a könyv nem lesz egy teljes átfogó útmutató a C#-hoz, DirectX-hez vagy a játékprogramozáshoz általában. Szándéka az, hogy kiindulópontot adjon a témához. Lehetetlenség lenne teljes útmutatót kínálni azon témák bármelyikéhez egy ilyen méretű könyvben (és lehetetlenség lenne teljes útmutatót adni a játékprogramozáshoz egy akármilyen méretű könyvben), úgyhogy végigmentem a C#-on és DirectX-en és felszínre hoztam az alapvető témákat, ahogyan a többit is, amelyek alapvető fontosságúak a játékprogramozáshoz.

I. rész: Tanulás a C#-ról

1. fejezet

A C# története

A történelem mindig a kedvenc tantárgyaim közé tartozott. Nagyon hasznosnak tartom tudni, hogy az események hogyan és miért történtek a múltban. A történelem ismerete segít megmagyarázni, hogy miért vannak most úgy a

dolgok, ahogy vannak és ötletet adnak, hogy a jövőben hogyan lesznek. Ez az, amiért ha tanulok egy új technológiát, akkor először annak történetét próbálom kitalálni; így cselekedve ötleteket kapok arról, hogy milyen problémák megoldására volt tervezve és milyeneket nem tud megoldani. Ebben a fejezetben tanulni fogsz arról:

- hogy a gépi nyelvek elmondják egy számítógépnek, hogy mit csináljon.
- hogy az assembly nyelvek elmondják egy számítógépnek, hogy mit csináljon, mégpedig olvasható, emberi kifejezésmódon.
- hogy a magas szintű programozási nyelvek lehetővé teszik neked, hogy elvonatkoztasd programjaidat az alacsony szintű gépi nyelvtől és leírják azokat egy könnyedebb módon.
- hogy a virtuális gépek lefordítják a képzeletbeli gépi kódot aktuális gépi kódra.
- hogy a virtuális gépek segítenek hordozhatóvá tenni a programokat többféle platformra.
- hogy az összes program lecsökkenthető gépi kódú alakba.
- hogy a .NET felgyorsítja a VM (virtuális gép) folyamatot a kód lefordítása által csak a futás első idején.

A számítógépek története röviden

Egyszer volt, hol nem volt, egy varázslatos földön messze, messze innét, néhány őrült ember elhatározta, hogy felfedezi a matematikát. Természetesen azokban az időkben még nem voltak olyan dolgok, mint számológépek vagy számítógépek, ezért az emberek a matematikát kézzel, papíron csinálták. Bárki tudja tanúsítani, aki számológép nélkül volt matekórán, hogy ez nem mókás. Ezenkívül az agyadat is kell használni (ez maga a horror!), és néhány száz számolás után a kezed begörcsöl. Hol van ebben a móka?

A probléma megoldására néhány szórakoztató ember feljött azzal a ragyogó ötlettel, hogy készít egy gépet, amely tudott matematikai számításokat végezni neked, az összes bosszantó gondolkodás és írás nélkül. Az ember létrehozta a számítógépet, és szólt, hogy ez jó. Most nekünk nem kell várnunk néhány szegény lélekre, hogy párszáz számítást elvégezzen papíron; helyette van egy gépünk, amely sokkal gyorsabban és pontosabban megcsinálja ezt.

Gépi és Assembly nyelvek

Azokban az ősi időkben a számítógépes programok egyszerűek voltak. A legkorábbi számítógépek némelyike csak nyolc különböző parancsot támogatott és csak néhány tucatot tudott végrehajtani, mielőtt egy új programot kellett megalkotni. Alapvetően egy programozó csinált egy számlistát, betáplálta egy számítógépbe és futtatta; a számok jelképezték a parancsokat. Egy feltételezett példában a 0 szám jelképezheti az összeadás parancsot, az 1 pedig a szorzást. Az ilyen módon írt programokat úgy nevezik, hogy gépi kódban íródott programok.

Az olyan egyszerű gépekkel, mint a korai számítógépek, meglehetősen könnyen lehetett emlékezni arra, hogy mely szám mely parancsot jelenti – végül is csak nyolc utasítás volt. Ám a számítógépek egyre összetettebbekké váltak. Az emberek elkezdtek adni még több parancsot, úgyhogy hamarosan volt vagy fél tucat, vagy még száznál is több elérhető. Nagyon kevés ember tudott csak emlékezni ennyi parancsra, és feltekintve a használati utasításból minden perc nagyon unalmas lehetett, így feltalálták az assembly nyelveket. Egy assembly nyelv lényegében egy olyan nyelv, amely közvetlenül fordít szó-alapú parancsokat gépi nyelvre. Például az előzőleg említett feltételezett gépben a gépi nyelvű kód valahogy így nézne ki, amely a 6-ot megszorozza 7-tel:

```
1 6 7
```

ahol az 1 jelképezi a parancsot, az azt követő két szám pedig az adatokat. Természetesen a kiírt sokszáz számsor bánthatja a szemedet és az agyadat, ezért egy assembly nyelvű parancs valahogy így nézhet ki:

```
mul 6, 7
```

Ah, most már sokkal csinosabb a szemnek! Legalább most már megfelelően el tudod mondani, hogy a 6-ot akarod megszorozni 7-tel. A számítógépeknek van egy assembler-nek nevezett programjuk, amelyek fogják az assembly nyelvű kódot és közvetlenül lefordítják gépi kódra. Az assembler-ek nagyon egyszerű programok; alapvetően azt csinálják, hogy megkeresik a parancs nevét és kicserélik az azt jelképező számmal.

Hordozhatóság

A hordozhatóság kifejezés utal egy program azon képességére, hogy átrakható egy másik számítógépre. Egészen a legutóbbi időkig egy nagy fájdalmat jelentett az ülep számára. Sok ember csinált számítógépeket azokban a rossz régi időkben, és majdhogynem egyik gép sem működött együtt a másikkal. Tehát lehetett egy géped, amely megértette, hogy az 1 a szorzást jelenti, de egy másiknál meg mondjuk a 2 jelentette ugyanezt.

Az assembly nyelvek segítettek megoldani ezen problémák némelyikét. Feltételezheted, hogy a legtöbb gépnek megvoltak az alapvető összeadási, kivonási, szorzási és osztási parancsai, így alapvetően ami neked kellett az egy assembler az A gépnek a „MUL”-t 1-re fordítani és egy assembler a B gépnek a „MUL”-t 2-re fordítani.

Elméletileg tudtál hordozni, vagyis portolni egy assembly programot sok különböző gépre, feltételezve, hogy mindegyik gépnek van egy assembler programja, amely megérti a használt assembly nyelv nyelvtanát.

De a dolgok csúnyán felgyorsultak. A számítógépek meglehetősen összetettekké váltak és az összes gyártó elhatározta, hogy annyi utasítást rak egy processzorba, amennyit csak tud. De egyikük sem egyezett meg abban, hogy milyen parancsokat kellene használnia! Ezért néhány számítógépnek voltak utasításai a lebegőpontos-matematika végrehajtására, a többinek meg nem. Néhány végre tudta hajtani a binárisan kódolt decimális (BCD) számításokat, a többi meg nem. Néhány adott neked egy tucat különböző módot a memória elérésére, míg a többi csak egyet!

Megjegyzés: ne nagyon aggódj a BCD számítás jelentése miatt, mert ez nem igazán használt sokszor a játékprogramozásban.

Houston, van egy problémánk. Az assemblereket többé nem lehetett hordozni egyik platformról a másikra, mert a platformok összevisszaságokká váltak. Így ahelyett, hogy az összes gépre próbálták volna írni a programokat, a legtöbb programozó megtanulta, hogyan használjon egy gépet, és készítse a programjait csak arra a gépre. Akarod az A gépre írt programot a B gépen futtatni? Sajnos az nem történhet meg.

A magas-szintű nyelvek megmentik a napot

Belépés a magas-szintű programozási nyelvek világába. Ezek voltak azok a magasan összetett nyelvek, amelyek leírták, hogy hogyan kell végrehajtani matematikai számításokat, de nem mentünk bele azokba a zavaró részletekbe, hogy valójában hogyan csinálták ezeket. Mondhatsz valami ilyesmit:

int i = 6 * 7;

Egy C nyelvben (egyike a legkorábbi és legnépszerűbb magas-szintű programozási nyelveknek) egy *fordító*-nak (compiler) nevezett program fogná azt a szöveget és lefordítaná gépi kódra neked. Neked igazán nem kell tudnod, hogy hogyan történik – az egész amit tudsz, hogy csináltál egy számot, amely tárolja a 6-szor 7 eredményét.

Sajnos a magas-szintű programozási nyelvek elbuktak a tökéletesen hordozható programok létrehozásának terén. A probléma az, hogy mindegyik fordító különböző, és máshogy csinálja a dolgokat. Minden operációs rendszernek különböző *Application Programming Interface*-e (API) van, amelyek más gépeken nem használhatók. Ha csinálsz egy Windows programot, akkor a WIN32 API-val fogsz bánni, de sok szerencsét ennek egy Macintosh-on futtatásához.

Hordozhatóság virtuális gépekkel

Aztán valakinek az a nagyszerű ötlete támadt, hogy feltalálja a virtuális gépet (virtual machine – VM). Egy virtuális gép egy számítógép processzor, amely szoftverben van szimulálva. Például, mondjuk megalkotod a saját gépi nyelvedet. Ez nagyszerű, de ha nincs meg a saját processzorod a nyelv végrehajtására, akkor az egész használhatatlan. Tehát előremész és megalkotsz egy szoftverdarabot, ami a virtuális géped lesz. Ez a szoftver fog utasításokat beolvasni a saját gépi nyelvedből, és fordítja parancsokká a futtató számítógépnek.

Tehát mi ennek a lényege? Miért nem csak megírod a programodat a jelenlegi gépi nyelven az első helyen? A válasz a hordozhatóság. Képzeld el, hogy kimennél és csinálnál VM-eket tíz különböző platformra. Most csak egy programot tudtál csinálni a VM nyelvedben, és futtatni azt tíz teljesen eltérő gépen (pl. Windows, Linux, Macintosh és Sparc rendszereken)!

Egyike a legnépszerűbb virtuális gépeknek, amely betört a számítógép iparba, a Java Virtual Machine (JVM) volt, kifejlesztve a Java programozási nyelvvel párhuzamosan. Az ötlet az volt, hogy készítenek egy számítógép nyelvet, amely futna bármilyen számítógépen, bárhol – 100 százalék hordozhatóság. Ez lehetővé tenné a fejlesztőknek, hogy készítsenek egy programot és eladhassák bármely gépre, amelyen rajta van a JVM, idő és pénz költése nélkül, hogy próbálgassák, működik-e másik platformon. A fejlesztők így sokkal nagyobb célközönséget érnek el. Programjaid nemcsak Windows, hanem Macintosh és Linux gépeken is futhatnak, további erőfeszítés nélkül részedről.

Bár mindezek jól hangzanak elméletben, és a Java nagyon népszerű nyelvvé vált, a játékiparban mégis megbukott. Az első probléma természetesen a sebesség. Egy virtuális gép költséges, ami azt jelenti, hogy minden végigmegy a virtuális gépen, mielőtt végrehajtódik az aktuális gépen. A játékprogramozásnak majdnem mindig gondja van a sebességgel: mindenki a határig! Fogni akarod amid van, és a lehető legmesszebbre lökni.

Virtuális gép megléte esetén volt egy nagy probléma: miért programoznál egy játékot Javaban, hogy fele olyan gyors legyen, mintha C++-ban írtad volna? Nyilván kis játékoknál és különösen Web-alapú játékoknál a sebesség nem igazán nagy dolog (és a Java valóban Web-alapú alkalmazásokkal és játékokkal indult), de minden másra elég nagy és a Java nem igazán vált tényezővé.

Egy egyszerű nyelv nem válasz minden problémára. Vannak időszakok, amikor Java-szerű nyelvben akarsz programozni egy játékot, de máskor a Javanak nincs meg az a tulajdonsága, amit igényelsz. Nem szándékozom túl mélyen elmerülni ebben, de teljes nyelvek léteznek még, amelyek teljesen más programozási paradigmákat használnak és sokkal könnyebben képesek megoldani a problémákat (például a funkcionális programozási nyelvek, mint a LISP, meglehetősen gyakran használatosak mesterséges intelligencia programozására), mint ahogy a Java tudja. Egyszerűen nem jó ötlet kötni egy nyelvet egy virtuális géphez, mert erőlteted az embereket, hogy egy olyan nyelven programozzanak, amit nem szeretnek (és hidd el, rengetegen vannak, akik ki nem állhatják a Javat).

.NET a megmentéshez

Így jön a .NET. A Microsoft figyelemmel kísérte a hibákat, amelyeket a Sun vétett a Javaval, és megpróbálta kijavítani azokat a .NET-ben. Nem mindet, de összességében a .NET hatalmas javítás a Javan, és sokat megvalósít azokból, amiknek kiadásán a Java megbukott.

A Microsoft .NET platform lényegében egy nagyon összetett eszközháló, amely magábafoglal mindent a biztonságtól a Web telepítésig. A .NET legérdekesebb része a Common Language Runtime (CLR), amely egy ál-virtuális gép, ami végrehajtja a Microsoft Interpreted Language (MSIL) kódot. Kicsit bővebben kifejtem a jelentését.

A .NET nem kötött semmilyen egyéni nyelvhez sem. A Microsoft hivatalosan négy különböző .NET nyelvet támogat:

- Managed C++
- C# (kiejtve: *szí sharp*)
- Visual Basic.NET
- J# (kiejtve: *dzséj sharp*)

Nem hivatalosan még szó szerint tucatnyi további nyelv van, amelynek olyan fordítója van, amely MSIL kódot állít elő. Ezen nyelvekbe beleértendő a LISP, a PERL, a Python és még (lélegzetelállításul) a COBOL is.

Figyelmeztetés: minthogy sok nyelv van, amelyek képesek fordítani .NET-be, és a .NET-nek hozzáférése van a DirectX-hez, elméletileg lehetséges programozni játékokat COBOL-ban is. De ezzel csak képesített szakemberek próbálkozzanak; más szóval, ne próbáljátok ki otthon, gyerekek! Megsérthettek valakit.

A legjobb része a .NET-nek, hogy benne minden megoszt egy hasonló szerkezetet, aminek a neve *Common Type System*. Alapvetően, ha létrehozol egy osztályt egy nyelvben (mint például Visual Basic), adsz neki két egész számot és lefordítod, akkor létre tudod hozni az azonos osztályt C#-ban azonos adatokkal, és mindez feltételezhetően azonos MSIL kódba fordítódik.

A másik, hogy ami .NET-be van fordítva, az hozzáférhet másik .NET modulokhoz, melynek érdekes mellékhatása, hogy lehetővé teszi különféle nyelveknek, hogy beszélhessenek egymással. Például ha C#-ot használsz, akkor elmondhatod annak, hogy használjon olyan osztályokat, amelyek Visual Basic.NET-ben jöttek létre. Tudsz örökölni belőlük és kiterjesztheted a képességeiket, ami azt jelenti, hogy lehetnek olyan osztályaid, amelyek egynél

több nyelvet használva jöttek létre! A .NET rendszer ezokból hihetetlenül rugalmas; sosem volt még olyan rendszer fejlesztve, amely lehetővé teszi neked, hogy ilyen könnyen építs be olyan sok paradigmát.

„Éppen Időben” fordítás

Ahogy előzőleg említettem, minden virtuális gépnek vannak költségei. Bár a .NET rendszer nem pontosan egy igazi virtuális gép. A .NET rendszer valamit igazán ügyesen csinál: egy Just In Time fordításnak (JIT, „Éppen Időben”) nevezett módszert használ a kód végrehajtásának felgyorsítására. A JIT rendszer nyomon követi a kódodat, és mikor először futtatsz egy modult, fogja az MSIL kódodat és átalakítja a géped natív kódjára. Tehát amikor először futtatsz egy .NET modult a Windows gépeden, a JIT betölt az MSIL kódba, lefordítja közvetlenül x86 kódba, és aztán elmenti azt a kódot. Attól a ponttól kezdve, valahányszor a modulod fut, a számítógép végrehajtja a natív x86 kódot, és teljesen elkerüli bármiféle használatát a virtuális gépnek, tehát ez majdnem olyan, mintha fordítottál volna egy programot közvetlenül egy magas szintű nyelvből gépi nyelvbe – de nem egészen.

Csökkentő elmélet

Az ötlet a .NET és a virtuális gépek mögött általában az, hogy a programok magas szintű nyelvekben mindig tudnak „leméreteződni” vagy „lecsökkenni”. Vegyük például azt az ötletet, hogy kiírunk szavakat a monitorodra. C# nyelven ez elvégezhető egy kódsorral:

```
System.Console.WriteLine(„ Szeretem a sütiket. „);
```

De mit csinált ez valójában? Belsőleg a számítógép alapvetően csak mozgat némi memóriát felé, és közli a bemeneti/kimeneti busszal, hogy küldjön némi adatot a képernyőre. Elméletben bármely összetett parancs bármely nyelvben lecsökkenthető egyszerűbb utasítások kötegére.

Itt egy párhuzam: amikor egy autóban elfordítod az indítókulcsot, az autó beindul; ez olyan mint egy magas-szintű nyelv. A kocsi motorján belül események sorozata történik:

1. Az akkumulátor elkezdi forgatni a dugattyúkat.
2. Az akkumulátor begyújtja a gyújtógyertyát.
3. A gyújtógyertya berobbantja az üzemanyagot a hengerekben.
4. A berobbanó üzemanyag még gyorsabban forgatja a dugattyúkat.

Minden nagyobb parancs (mint az autó motorjának elindítása) lebontható jellegzetes utasításkészletre (mint az a lista feljebb). Csak néhány különböző kisebb utasítástípus van, és ezek azok, amelyekre a virtuális gép számít. Létrehozhatsz néhány szuper összetett programozási nyelvet, amelynek van olyan függvénye is, hogy MostEgySzuperjoJatekotCsinal(), de a végén a számítógép lecsökkenti azt egy utasítássorra, amely matematikai műveleteket csinál és memóriát mozgat hozzá. A valóságban ez minden, amit a számítógép végez – végrehajt matematikai műveleteket és memóriát mozgat hozzá.

Így ha minden, ami egy virtuális gépnek szükséges az, hogy tudja hogyan kell matematikai műveleteket végrehajtani és hozzá memóriát mozgatni, azt jelenti, hogy elég egyszerűek ezt csinálni és könnyen hordozható különböző platformokra.

Megjegyzés: egy egész számítógép tudományág létezik arra szánva, hogy problémákat egyszerűbb alakba csökkentsen. Vannak valójában teljes számítógép probléma osztályok, melyeket *NP-Complete* problémáknak hívnak, amelyekben minden egyszerű probléma lecsökkenthető egy problémára, amely részletez minden NP-Complete problémát a világon.

A jövő

A C# a Microsoft zászlóshajója a .NET platformra. A társaság fogni akarta a C++-t és megállapítani, hogy mi a baj vele; ez meglehetősen nehéz célkitűzés, de ha valakinek van elég erőforrása megbirkózni ezzel a problémával, akkor az a Microsoft.

Nincsenek még olyan nagyobb játékstúdiók, amelyek közzétesznek fejlesztéseket C#-pal, de ez érthető. A nyelv még a kezdeti szakaszában van és egy nagy társaság nem akar dollármilliókat költeni olyan projektre, amelyben nem száz százalékig biztos. Idővel valószínűleg ez meg fog változni. Tény, hogy az egyetlen legnagyobb plusz dolog egy .NET rendszerről amit tud nyújtani az a hordozhatóság. Most, ha írni akarsz egy játékot PC-re és játékkonzolra, gyakorlatilag két játékot kell írnod, mert esélyes, hogy a rendszereknek nincs

meg minden közösben. Ez egy óriási gond azon társaságoknak, amelyek nem túl gazdagok és nem engedhetik meg maguknak két játék írását, így ők valószínűleg eldönteni szándékoznak azt, hogy a játékot PC-re vagy konzolra írják. A jövőben a konzolok, mint az Xbox 2 valószínűleg támogatják majd a .NET-et, tehát lehetségesnek kellene lennie, hogy egy játék a megírása után tökéletesen működjön PC-n és konzolon egy időben. Amint a magas szintű nyelvek bevezették egy teljes új szintjét a félig-hordozhatóságnak a számítógép világba, a .NET meglebegtetett egy még nagyobb hatást.

2. fejezet

Első C# programod

Van egy ősi hagyomány (oké, nem olyan öreg) a számítógép programozásban, ami azt mondja, hogy bármilyen nyelven írt első programodnak illene egy „Szervusz világ” programnak lennie, olyannak, ami egy egyszerű üdvözlő üzenetet ír ki a számítógéped monitorára. Ennek C#-ban valahogy így kellene kinéznie:

```
class HelloCSharp
{
	static void Main( string[] args )
	{
		System.Console.WriteLine( “Szervusz, C#!” );
	}
}
```

Első pillantásra azt láthatod, hogy ez körülbelül négy vagy öt sorral hosszabb, mintha C-ben vagy C++-ban írtad volna; ez azért van, mert a C# egy még összetettebb nyelv.

Osztályok

A C# egy *objektum-orientált* programozási nyelv, amely ennél a pontnál még nem biztos, hogy sokat mond neked. „Az osztályok rövid bemutatója” című fejezetben részletesebben végig fogok menni a fogalmakon, de most elég annyit tudnod, hogy a C# objektumokként ábrázolja a programjait.

Az ötlet az, hogy főnevekbe és igékbe különítődnek el a programjaid, ahol minden főnév egy objektumként ábrázolható. Például ha készítesz egy játékot, melyben űrhajók repkednek körbe, akkor gondolhatsz úgy az űrhajókra, mint objektumokra.

Egy osztály egy C# programban meghatároz egy főnevet; elmondja a számítógépnek, hogy milyenfajta adatai lesznek az objektumaidnak és milyenfajta cselekvéseket lehet rajtuk végrehajtani. Egy űrhajó osztály részletezheti a számítógépnek, hogy mekkora létszámú személyzet van benne, mennyi üzemanyaga van még és hogy milyen gyorsan megy.

C#-ban valójában az egész programod egy osztály.

A belépő pont

Minden programnak van egy *belépő pont*ja, a hely a kódban, ahol a számítógép el fogja kezdeni a végrehajtást. A régebbi nyelvekben, mint a C és C++, a belépő pont tipikusan egy *main* nevezetű globális függvény volt, de C#-ban ez egy kicsit másképpen van. A C# nem engedi meg neked, hogy legyenek globális függvényeid, inkább rákényszerít arra, hogy függvényeidet osztályokba helyezd. A C# ebben a tekintetben Java szerű; a belépő pont minden C# programnak egy *Main* nevezetű statikus függvény egy osztályon belül.

Minden C# programnak kell lennie egy osztályának, aminek van egy statikus *Main* függvénye; ha nincs, akkor a számítógép nem fogja tudni, hogy honnan kezdje futtatni a programot. Továbbá csak egy *Main* függvényt tudsz meghatározni a programodban; ha egynél több van, akkor a számítógép nem fogja tudni, hogy melyikkel kell kezdeni.

Megjegyzés: technikailag el tudsz helyezni egynél több *Main* függvényt a programodban, de az csak rendetlenné teszi a dolgokat. Ha egynél több *Main* függvényt építesz be, akkor el kell mondanod a C# fordítódnak, hogy melyik tartalmazza a belépő pontot – ez igazán bajos és meg tudsz lenni nélküle is.

Szia, C#!

Ez az a sor, amely végrehajtja konzolos C# programnál a kiírást:

```
System.Console.WriteLine( “Szia, C#!” );
```

Ez a sor fogja a System.Console osztályt – mely be van építve a .NET keretrendszerbe – és megmondja neki, hogy írja ki a „Szia, C#!”-ot, használva a WriteLine függvényt.

Fordítás és futtatás

Van néhány módja, hogy lefordítsd és futtasd ezt a programot. A legegyszerűbb, ha nyitsz egy konzolos ablakot, megkeresed ennek a programnak a mappáját és használod a parancssori C# fordítót, hogy lefordítsa az állományt, mint itt:

csc HelloCSharp.cs

A másik módszer, hogy a SharpDevelop vagy Visual Studio.NET rendszeren belül végzed el a fordítást és futtatást az általad használt fejlesztőeszköz megfelelő parancsikonja segítségével.

Tehát most, amikor futtatod ezt a programot, ilyen kimenetet kapsz a képernyőn:

Szia, C#!

Ta-da! Most már megvan az első C# programocskád, ami kiír némi szöveget a képernyőre.

Az alapok

Majdnem minden programozási nyelvnek vannak közös tulajdonságai. Egy dolgot: az adattárolás módját általában tudniuk kell a programozási nyelveknek. Ezenkívül műveleteket kell végrehajtani rajtuk a mozgatásuk és rajtuk való számítások végzése által.

Alapvető adattípusok

A legtöbb programozási nyelvhez hasonlóan a C#-nak nagyszámú beépített adattípusa van. Ezek a következők:

Típus	**Méret (byte-okban)**	**Érték**
bool	1	igaz vagy hamis
byte	1	0-tól 255-ig
sbyte	1	-128-tól 127-ig
char	2	Alfanumerikus írásjelek (Unicode-ban)
short	2	-32,768-tól 32,767-ig
ushort	2	0-tól 65,535-ig
int	4	-2,147,483,648-tól 2,147,483,647-ig
uint	4	0-tól 4,294,967,295-ig
*float	4	$-3.402823x10^{38}$-tól $3.402823x10^{38}$-ig
long	8	-9,223,372,036,854,775,808-tól 9,223,372,036,854,775,807-ig
ulong	8	0-tól 18,446,744,073,709,551,615-ig

*double 8 -1.79769313486232x10^{308} -tól 1.79769313486232x10^{308}
**decimal16 -79,228,162,514,264,337,593,543,950,335-tól
79,228,162,514,264,337,593,543,950,335-ig

*Ezek lebegőpontos számok, amelyek nem pontos tizedes értékeket reprezentálnak.

**Ezek fixpontos számok, amelyek pontos tizedes értékeket reprezentálnak 28 számjegyig.

Az egész alapú típusok (byte, short, int, long és így tovább) csak egész számokat tudnak tárolni, mint 0, 1, 2 és így tovább; tizedesjegyű számok tárolására, mint 1.5 vagy 3.14159, alkalmatlanok. Hogy tizedesszámokat is tudj tárolni, ahhoz egy lebegő- vagy egy fixpontos formátumba kell váltanod. Hogy a számok ezen fajtái hogyan tárolódnak, annak pontos leírása túlmutat ezen könyv hatáskörén, de van egy vékony különbség, ami hatással lesz tudósokra és matematikusokra (de valószínűleg nem a játékprogramozókra).

Megjegyzés: alapvetően a lebegőpontos számok nem tudnak pontos számokat tárolni, csak megközelíteni tudják a tizedesszámokat bizonyos mennyiségű hibán belül. Például lebegőpontosokat használva tudod ábrázolni az 1.0 és 1.00000012 számokat, de nem tudsz ábrázolni bármely számot köztük. Tehát ha te beállítasz egy lebegőpontosat egyenlővé 1.00000007-tel, akkor a számítógép önműködően felkerekíti 1.00000012-vé. A double-k hasonlóan csinálják, de még nagyobb a pontosságuk (15 számjegyig). A decimal-ok más módon vannak kódolva, és annak ellenére, hogy a .NET dokumentáció fix-pontos számoknak nevezi őket, azok technikailag még lebegőpontos számok, és pontosságuk 28 számjegyig tart.

Műveleti jelek

A műveleti jelek szimbólumok, amik felbukkannak egy programozási nyelvben; elmondják a számítógépnek, hogy végezzen bizonyos számításokat az adaton. A műveleti jelek közönségesen használatosak matematikai egyenletekben, úgyhogy biztos vagyok abban, hogy jól ismertek számodra.

A C# nyelvnek számos beépített műveleti jele van, és ha valaha is használtál C++-t vagy Java-t, akkor valószínűleg legtöbbjüket már ismered.

Matematikai műveleti jelek

A C#-nek öt alap matematikai műveleti jele van beépítve a nyelvbe, amit a következő táblázat is mutat:

Operátor	Szimbólum	Példa	Eredmény
Összeadás	+	5 + 6	11
Kivonás	-	6 – 5	1
Szorzás	*	6 * 7	42
Osztás	/	8 / 4	2
Modulus	%	9 % 3	0
Növelés	++	10++	11

Csökkentés — 10— 9

Az első négy egyértelmű, vagy legalábbis azoknak kellene lenniük. Az ötödik műveleti jel viszont új lehet számodra, ha ezelőtt még nem programoztál sokat. A modulus néha „maradék operátor” vagy „óra operátor” néven is ismert. Alapvetően az eredmény egy modulusból a maradékkal azonos, ha fogtad az első számot és elosztottad a másodikkal. A fenti táblázat példájában 3-mal osztottuk a 9-et, tehát a maradék 0. Ha veszed a 10 % 3-at, akkor 1 lesz az eredmény, mivel a 10/3 maradéka 1.

Megjegyzés: a modulust gyakran óra operátornak is nevezik, mert könnyen kiszámíthatod az eredményt egy óra használatával. Például vegyük a 13 % 12 számítást. Tételezzük fel, hogy egy 12-től kezdődő óránál van a kezed, és mozgatod előre egy órával, minden alkalommal 1-et számítva. Tehát mikor 1-et számítasz, a kéz 1-nél lesz, amikor 2-t, akkor 2-nél és így tovább. Végül is amikor 12-t kapsz, a kéz 12-nél lesz újra, és mikor 13-at számítasz, a kéz visszamozog 1-re. Tehát a 13 % 12 eredménye 1.

Megjegyzés: a növelő és csökkentő operátoroknak valójában két különböző változata van: a post- vagyis utótagos és a pre- vagyis az előtagos változat. Például a ++x a pre-inkrementáló (növelő) változat, az x++ pedig a post-inkrementáló. A különbség akkor van, mikor az operátorok végrehajtják a számításaikat. Például ha x az 10 és azt írod, hogy y = x++, akkor a számítógép először elhelyezi x értékét y-ban és aztán növeli x-et, hagyva y-t egyenlő 10-zel és x-et egyenlő 11-gyel, mikor a kód befejeződik. Másrészről y = ++x véghezviszi először a növelést, később pedig a hozzárendelést, hagyva x és y értékét is 11-nek. Ez egy másik maradvány a C-ből, és csúnyává és bonyolulttá teheti az olvasást, úgyhogy nem igazán ajánlom ezen műveleti jelek túl sokszori használatát.

Illene megjegyezned, hogy az összes matematikai műveletnek van alternatív változata, ami lehetővé teszi, hogy közvetlenül módosíts egy változót. Például ha 10-et akarsz hozzáadni x-hez, csinálhattad így:

```
x = x +10;
```

De ez valahogy nehézkes és szükségtelen. Helyette írhatod ezt:

```
x += 10;
```

Az összes többi matematikai műveletnek van hasonló változata:

```
x *= 10; //szorzás 10-zel

x /= 10; //osztás 10-zel
```

x -= 10; //10 kivonása

x %= 10; //modulus 10-zel

x >>= 2; //eltolás 2-vel jobbra

x <<= 2; //eltolás 2-vel balra

Bitenkénti matematikai műveletek

Az alapszintű matematikai műveleteken kívül vannak bitenkénti matematikai műveletek is, melyek bináris (kettes számrendszerbeni) matematikai műveleteket hajtanak végre számokon. Az alapvető bitenkénti matematikai műveletek C#-ban a következők:

Művelet	**Jele**	**Példa**	**Eredmény**
Bináris És	&	6 & 10	2
Bináris Vagy	|	6 | 10	14
Bináris Kizáró Vagy (Xor)	^	6 ^ 10	12
Bináris Nem	~	~7*	248

* - ez a példa egy bájton hajtódott végre

A bitenkénti matematikai műveleteknek szintén van alternatív változata:

```
x &= 10; // és 10-zel
x |= 10; // vagy 10-zel
x ^= 10; // xor 10-zel
```

Eltoló műveletek

Két eltoló művelet van: << és >>. Ezek a műveletek eltolják a biteket egy számban felfelé vagy lefelé, a következő egyenleteket eredményezve:

x<<y azonos ezzel: $x*2^y$

x>>y azonos ezzel: $x/2^y$

Megjegyzés: a biteltolás sokkal gyorsabb, mint a szorzás vagy osztás, de ritkán használt. A sebességi nyereség nem olyan látványos, és nehézkessé teszi a programod olvashatóságát.

Logikai műveletek

Van néhány közönséges logikai művelet, amelyek végrehajtanak összehasonlításokat a dolgokon, és visszatérnek Boolean, azaz logikai típusú *true* (*igaz*) vagy *false* (*hamis*) értékkel, az eredménytől függően. Ezek a következők:

Művelet	**Jel**	**Példa**	**Eredmény**
Egyenlő	==	1 == 2	false
Nem egyenlő	!=	1 != 2	true
Kisebb mint	<	1 < 2	true
Nagyobb mint	>	1 > 2	false
Kisebb vagy egyenlő mint	<=	1 <= 2	true
Nagyobb vagy egyenlő mint	>=	1 >= 2	false
Logikai És	&&	true && false	false
Logikai Vagy	||	true || false	true
Logikai Nem	!	!true	false

* - ez a példa egy bájton hajtódott végre

Változók

C#-ban, mint szinte bármely más programnyelvben, el tudod készíteni az alap adattípusok példányait, melyeket *változó*knak neveznek, és végrehajthatsz rajtuk matematikai műveleteket.

Könnyű megadni egy adatdarabot a programodban. Ehhez mindössze meg kell adnod az adattípusod nevét, aztán a változó nevét, amit utána létre akarsz hozni, és aztán (nem kötelezően) kezdőértéket adhatsz neki. Itt vannak példák:

```
int x = 10;
float y = 3.14159;
decimal z;
```

Figyelmeztetés: megjegyzendő, hogy ha megpróbálsz úgy használni egy változót, hogy előzőleg nem adtál neki kezdőértéket (például ha megpróbálod használni a *z* változót az előző kódmintából), akkor C#-ban fordítási hibát fogsz kapni. Régebbi nyelvekben, mint a C és C++, megengedhetnéd, hogy használj egy változót érték hozzáadása nélkül, mely sok hibát okozhatott volna, mert sosem tudod, hogy mi volt a változóban, ha sohasem állítottad be!

Itt egy példa, amely változókat használ matematikai függvényekkel:

```
int x = 10 + 5; // 15
int y = 20 * x; // 300
int z = x / 8; // 1
float a = (float)x / 8.0; // 1.875
x = (int)a; // 1
```

Szánj különös figyelmet a két utolsó sorra. Ezek megmutatják neked, hogy hogyan használj *typecast*-okat a programodban, melyekről bővebb kifejtést hamarosan olvashatsz itt.

Állandók

Megadhatsz állandókat, ál-változókat, amelyek értéke nem változhat meg a kódodban. Ez csak egy másik biztonsági tulajdonság, amely évek óta megtalálható a számítógép nyelvekben. Például:

```
const float pi = 3.14159;
```

Most használhatod a pi-t a számításaidban, de nem változtathatod meg az értékét (mert például 3.0 értékűre állítása a pi-nek abszolút értelmetlen!). Az fordítási hibát okozna:

```
pi = 3.0; //Hiba!!!
```

Tanács: az állandók javítják programod olvashatóságát a bűvös számok kiküszöbölése által. A *bűvös számok* számok a programodban, melyeknek nincs azonnali jelentése akárki olvasónak. Például írhatod valahová, hogy: x = 103; de nincs, aki igazán tudná, hogy mit is jelent ez a 103. Jelenthette volna a lőszerek számát egy tölténytárban, de valami teljesen mást is. Helyette használhatsz állandókat, hogy pontosan megmutasd, mire gondolsz, egy

```
const int loszerek = 103;
```

állandó megadásával korábban a programodban, és később használva azt az állandót pl. x = loszerek; alakban. Látod, mennyire olvashatóbb így?

Typecast-ok

Nézd át ezt a kódot:

```
float a = 1.875;
int x = (int)a;
```

Tekints az utolsó sorra: az *a* értéke 1.875, egy tört szám, és a kód utolsó sora megpróbálja elhelyezni *a* értékét *x*-be, amely egy egész szám. Nyilván nem tudod csak úgy átvinni *a* tartalmát *x*-be, így veszítened kell némi pontosságot. Régebbi nyelvek, mint a C/C++, ezt önműködően csinálnák neked, és levágja 1.875-öt 1-re azért, hogy illeszkedjen az integer értékbe (ezt a folyamatot *csonkítás*nak nevezik). Ha megpróbálnád begépelni ezt egy C# programba, akkor fordítási hibát kapnál:

x = a; //Hiba! Nem lehet ’float’-ot ’int’-be átalakítani.

Természetesen ez a kód tökéletesen működik régebbi nyelvekben, úgyhogy sokan automatikusan elutasítanák a C#-ot azzal, hogy „bonyolult használni”. Hallom is most őket: „Nem tudod automatikusan konvertálni a floatot integerbe, te buta fordító?”

Nos, a fordító valójában nem buta; az csak megpróbál neked némi hibakeresési időt megspórolni. Lehet, hogy nem vagy tudatában, de egy közönséges hibaforrás a programokban a véletlen csonkítás. Elfelejthetted, hogy az egyik típus egy egész szám, és némi fontos adat valahol elveszhet az átalakítás során. Tehát a C# elvárja, hogy pontosan jelezd, mikor akarsz adatot csonkolni. A következő két táblázat megmutatja, hogy mely átalakítások igényelnek Explicit (határozott) és Implicit (magától értetődő) konverziót.

Táblázat 1.

Ebből	**byte**	**sbyte**	**short**	**ushort**	**int**	**uint**
byte	I	E	I	I	I	I
sbyte	E	I	I	E	I	E
short	E	E	I	E	I	E
ushort	E	E	E	I	I	I
int	E	E	E	E	I	E
uint	E	E	E	E	E	I
long	E	E	E	E	E	E
ulong	E	E	E	E	E	E
float	E	E	E	E	E	E
double	E	E	E	E	E	E
decimal	E	E	E	E	E	E

Táblázat 2.

Ebből	**long**	**ulong**	**float**	**double**	**decimal**
byte	I	I	I	I	I
sbyte	I	E	I	I	I
short	I	E	I	I	I
ushort	I	I	I	I	I
int	I	E	I	I	I
uint	I	I	I	I	I
long	I	E	I	I	I
ulong	E	I	I	I	I
float	E	E	I	I	E
double	E	E	E	I	E
decimal	E	E	E	E	I

A táblázatok első látásra zavaróan nézhetnek ki, de valójában egészen egyszerűek. Például ha int-et akarsz double-ba konvertálni, nézd meg a 2. táblát, keresd ki az „int”-et a bal oldalán, majd a „double”-t a tetején. Azon a helyen egy I van, ami azt jelenti, hogy implicit konverziót hajthatsz végre:

```
int a = 10;

double b = a; //rendben
```

Ezután mondjuk double-ból int-be akarsz konvertálni. Tekints az 1. táblázatra, annak bal oldalán a „double”-ra és jobb oldalán az „int”-re. Egy E van azon a helyen, ami explicit konverzió szükségességét jelenti:

```
double a = 10.0;

//int b = a ←hiba lenne!

int b = (int)a; //ez viszont már jó
```

Megjegyzés: egy float vagy double értékből az átalakítás decimal-ba explicit számítást igényel. Ez azért van, mert a decimálisok más módon kódolnak adatokat, mint ahogy a float-ok vagy a double-ok csinálják, tehát határozott lehetőség van némi adatvesztésre a konverzió végrehajtásakor. Valószínűleg jelentéktelen, amivel nekünk, játékprogramozóknak törődnünk kellene, de azért neked tisztában kellene lenni vele.

Elágazás

Ha valóban tanultál programozási nyelveket, akkor tudod, hogy három különböző jellegzetesség van, aminek alapján egy nyelvet igazi programozási nyelvnek tekinthetünk.

Ezek a

- sorrendiség
- elágazás
- ismétlés

A sorrendiség lényegében azt jelenti, hogy a nyelvnek képesnek kell lennie parancsok végrehajtására adott sorrendben.

Alapjában véve az elágazás lehetővé teszi egy számítógép programnak, hogy megvizsgáljon egy adott változókészletet, és eldöntse, vajon folytatnia kell-e a végrehajtást, vagy el kell ágaznia a program egy másik részére.

A C#-nak van néhány feltételes kifejezése beépítve a nyelvbe; ezek mindegyike a C-ből örökölve, úgyhogy ismerős lehet számodra.

Az if kifejezés

Meglehetősen gyakori egy programban, hogy le akarod ellenőrizni, hogy egy feltétel igaz-e vagy nem, és ennek eredményétől függően végrehajtani cselekvéseket. Például ha egy olyan eseményt akarsz végrehajtani, amely egy feltétel igaz volta esetén futhat le, akkor írhatsz egy ilyesfajta kódot:

```
if (x ==10)
{
	//csinál valamit
}
```

A kód ellenőrzi, hogy valami x-nek nevezett változónak 10 e az értéke, és aztán végrehajtja a kapcsos zárójeleken belüli kódot, de csak ha x értéke 10. Ezt hívják *if* (*ha*) kifejezésnek.

Ezenkívül hozzáadhatod az *else* (*különben*) kikötést a végéhez, azért, hogy végrehajthass kódot minden olyan esetre, amikor *x* értéke **nem** 10:

```
if (x ==10)
{
	//csinál valamit
}
else
{
	//csinál valamit
}
```

Tehát a számítógép végrehajt mindent az első blokkban amikor *x* egyenlő 10-zel, és végrehajt mindent a második blokkban akkor, ha *x* értéke nem 10.

Továbbá hozzáadhatod az *elseif* kifejezést a végéhez, hogy végrehajthass többszörös lekérdezéseket:

```
if (x < 10)
{
	//csinálni valamit, ha x < 10
}
elseif (x < 20)
{
	//csinálni valamit, ha 10 <= x < 20
}
elseif (x < 30)
{
	//csinálni valamit, ha 20 <= x < 30
}
```

Megjegyzés: amennyiben használtál már valaha egy olyan nyelvet, mint a C++, akkor tudod, hogy egy ilyen kód előállításához, mint ez: *if (x)*, ahol *x* egy integer típus, számokat

használhatsz a feltételen belül. Régebbi nyelvekben a számítógép a 0-t *hamis* értékűként kezeli és bármely más értéket *igaz*-nak vesz, jelentve azt, hogy ha *x* értéke 0, akkor az *if* blokk nem fog végrehajtódni, de minden más esetben igen. A C# nem olyan, mint ez, és tulajdonképpen Boolean típus használatát igényli tőled minden feltételes kifejezésben. Tehát a kód jelenlegi alakjában fordítási hibát fog adni. Ha végiggondolod, a régi módszer nem igazán biztonságos, mert nem fejti ki pontosan, hogy mit tesztelsz. A C# biztonságosabbá és olvashatóbbá teszi a programjaidat.

A switch kifejezés

A *switch* kifejezés használata egy praktikus mód egy változóból a többszörös kimenetek gyors ellenőrzésére. Például ha van egy *x* változód, amely tartalmazhatja az 1, 2, 3 és 4 értékeket, és a programod minden egyes értékhez különböző cselekvés folyamatot vesz, akkor lekódolhatod ezt a *switch* kifejezéssel így:

```
switch ( x )
{
        case 1:
        //csinál valamit, ha x értéke 1
        case 2:
        //csinál valamit, ha x értéke 2
        case 3:
        //csinál valamit, ha x értéke 3
        case 4:
        //csinál valamit, ha x értéke 4
        default:
        //csinál valamit, ha x egyéb értékű
}
```

Tehát ha *x* értéke 2, akkor a kód a 2-es esetű blokkra fog ugrani, és így tovább.

Bár van itt egy becsapás. A kód jelenlegi állapotában az előző blokkban, ha x egyenlő 2-vel, akkor a kód helyesen a 2-es blokkra ugrik, de folytatódni fog, és végrehajt minden alatta levő blokkot is. Ez azt jelenti, hogy a kód végrehajtja a 3-as és 4-es blokkot, valamint a *default* (*egyébként*) ágat is. Néha ezt a viselkedést akarod, de legtöbbször nem, ezért szükséged van a *break* kulcsszóra a *switch*-ből való kilépéshez minden blokk után:

```
switch ( x )
{
        case 1:
        //csinál valamit, ha x értéke 1
        break; //kiugrás a switchből
        case 2:
        //csinál valamit, ha x értéke 2
```

```
        break; //kiugrás a switchből
        default:
        //csinál valamit, ha x egyéb értékű
        break; //itt nem kötelező
}
```

Tipp: a *break* a *switch* kifejezés utolsó blokkjában nem kötelező, elhagyható, mert alatta már nincsen kód. De mégis jó ötlet kitenni mindenhová – nehogy később további blokkok esetleges hozzáadásakor elfelejtsd.

Rövidzárlat kiértékelés

Hadd térjek itt egy másik tárgyra, amely eléggé fontos, amikor feltételes utasításokat értékelsz ki.

Az összes C-alapú nyelv támogat valamit, amit *rövidzárlat kiértékelésnek* neveznek. Ez egy nagyon hasznos eszköz, de tud néhány problémát okozni neked, ha végre akarsz hajtani néhány különleges kód trükköt.

Ha ismered a bináris matematikai szabályokat, akkor tudod, hogy egy *és* kifejezésnél ha az egyik operandus hamis, akkor az egész dolog hamis. A következő lista mutatja a logikai *és* és *vagy* eredménytáblázatát:

x	**y**	**x és y**	**x vagy y**
igaz	igaz	igaz	igaz
igaz	hamis	hamis	igaz
hamis	igaz	hamis	igaz
hamis	hamis	hamis	hamis

Nézd meg a táblázatot, különösen azt a két sort, ahol *x* értéke hamis. Az „x és y” művelet számára nem számít, hogy *y* milyen, mert az eredmény mindig hamis lesz. Ugyanúgy ha az első két sort nézed, megjegyezheted, hogy valahányszor *x* igaz, az „x vagy y” művelet igaz, és nem számít *y* értéke.

Tehát ha van valami ilyesmi kódod:

```
if( x && y )
```

akkor a számítógép kiértékeli *x*-et, és ha az hamis, akkor nem fog még *y* kiértékelésével is alkalmatlankodni.

Ugyanúgy:

```
if( x || y )
```

itt is ha *x* igazzá válik, akkor *y* nem számítódik ki. Ez egy kis optimalizáció, amely megfelelő körülmények között felgyorsíthatja a programodat. Például:

```
if( x || ( ( a && b ) && ( c || d ) ) )
```

Ha ez a kód hajtódik végre, és úgy találja, hogy *x* értéke igaz, akkor az a nagy halom a jobb oldalon soha nem fog kiszámítódni.

Ez hibák nagyon trükkös forrása lehet, de csak ha trükkösen kinéző kódot írsz. Nézd ezt a sort például:

```
if( x || ( ( y = z ) == 10 ) )
```

Ha az első reakciód ezt a kódot látva az, hogy „Mi a fene folyik itt?”, akkor megérdemelsz egy sütit. Ez a kód hihetetlenül csúnya, és nem tudod megmondani, hogy a szerzőjének mi volt vele a szándéka. De sajnos ez egy tökéletesen érvényes C# kód, és valaki valahol gondolhatja azt, hogy elég dörzsölt ahhoz, hogy ilyen anyagot írjon.

Mindenesetre, ha *x* értéke igaz, akkor a számítógép figyelmen kívül hagyja a kód második felét. De ha *x* hamis, akkor ami *z*-ben van, az hozzárendelődik *y*-hoz, aztán az eredmény összehasonlítódik 10-zel, adva a kódnak hasonló szerkezetet, mint ez a jobban olvasható változat:

```
if( x == false )
y = z;
if( x || ( y == 10 ) )
```

Ez a második változat szinte teljesen másképp néz ki, mint az első, így láthatod, hogyan próbál néhány ügyes trükk igen sok problémává válni számodra egy napon.

Ismétlések

A harmadik jellemző vonás, hogy egy számítógép nyelvben van *ismétlés* vagy *ciklus*. Lényegében az ismétlés lehetővé teszi számodra, hogy egy meghatározott feladatot újra és újra végrehajts. Az a három, amelyekről ebben a fejezetben lesz szó, a C programnyelvből öröklődött. A negyedikről később lesz szó.

A while ciklus

Az első és legegyszerűbb ismétlési szerkezet a *while* ciklus. Íme egy példa:

```
while ( x < 10 )
{
	// csinál valamit
}
```

Bármi is van a zárójeleken belül, az újra és újra végrehajtódik, amíg *x* értéke 10-nél kisebb. Ha *x* sosem lesz egyenlő vagy több mint 10, akkor a ciklus a végtelenségig ismétlődik.

A for ciklus

Másik népszerű ismétlési szerkezet a *for* ciklus, amely csak egy másik mód egy *while* ciklus végrehajtására. Az alap nyelvtana egy *for* ciklusnak a következő:

```
for ( inicializáció; feltétel; cselekvés )
```

A kód inicializációs része egyszer hajtódik végre, amikor belép a *for* ciklusba. Lehetővé teszi számodra, hogy beállíts bármilyen változót, amire szükséged lehet.

A feltétel rész minden ismétlődés kezdetekor hajtódik végre, és ha hamis értékűként tér vissza, akkor a ciklus kilép.

A cselekvés rész minden ismétlődés végekor hajtódik végre.

Általában *for* ciklusokat használsz, hogy létrehozz ismétléseket, amelyek végigmennek egy számsorozaton egy bizonyos változó számára. Például ha 10 számítást akarsz végrehajtani *x*-en, ahol *x* értéktartománya 0-tól 9-ig tart, csinálhatsz egy ciklust, mint ez:

```
for ( int x = 0; x < 10; x++ )
{
	// csinál valamit
}
```

Először amikor a ciklus végrehajtódik, akkor *x* értéke 0, aztán következőleg 1, és így tovább, míg el nem éri 9-et.

Csinálhatsz néhány különleges dolgot is, mint több változó inicializálása vagy több cselekvés végrehajtása:

```
for ( int x = 0, int y = 0; x < 10; x++, y += 2 )
```

Ez a ciklus létrehoz két változót, *x*-et és *y*-t, ahol *x* 0-tól 9-ig ismétlődik, *y* pedig 0-tól 18-ig, kihagyva minden egyéb számot.

A do-while ciklus

Néha programozáskor olyan helyzet is keletkezhet, amikor teljes mértékben bizonyos akarsz lenni abban, hogy egy ciklus legalább egyszer végrehajtódik. Nézd ezt a kódot például:

```
int x = 0;
while( x > 0 )
{
// ez a ciklus soha nem jut el a végrehajtásig
}
```

A *for* és *while* ciklusoknál mindig megvan az esély arra, hogy ha a feltétel kiértékelése *hamis*, akkor a kód a cikluson belül soha nem hajtódik végre. A *for* és *while* ciklusok helyett használhatod a *do-while* ismétlést, mely végrehajt mindent és ellenőrzi a feltételt a ciklus végrehajtása után. Itt egy példa:

```
do
{
        // ciklus kód itt
} while( feltétel );
```

Break és Continue

A *break* és *continue* két hasznos utasítás, amiket felhasználhatsz, amikor csinálsz valamit a ciklusokon belül és meg akarod változtatni a folyásukat.

Break

Az első a *break* kulcsszó, amelyet láttál már használva a *switch* blokkokon belül. Alapvetően elhelyezve egy break-et a programban azt okozza, hogy a ciklus végére ugrik és kilép belőle. Itt egy példa:

```
for ( int x = 0; x < 10; x++ )
{
        if ( x == 3 )
        break;
}
```

Ez a ciklus 0, 1, 2 és 3 értékekkel végigmegy *x*-en, aztán kilép, amikor *x* egyenlő 3-mal.

Continue

A másik ciklus változtató a *continue* kulcsszó. Ez a ciklus végrehajtásának megállását okozza, és a visszaugrást a tetejére majd az újbóli elindulást. Itt egy példa:

```
for( int x = 0; x < 10; x++ )
{
	FunctionA();

	if( x == 3 )
	continue; // visszaugrás a tetejére, kihagyva mindent ami alatta van
	FunctionB();
}
```

Tételezzük fel egy pillanatra, hogy a *FunctionA()* és *FunctionB()* függvények léteznek. Ez a ciklus 0-tól 9-ig értékekkel végigmegy *x*-en, de amikor *x* értéke 3 lesz, akkor a kód kihagyja a *FunctionB()* függvényt, és a ciklus elejére ugrik újra.

Érvényesség

A *hatáskör* (*scope*) szakkifejezés, mikor egy számítógépes programról van szó, utal a helyre egy programban, ahol egy változó érvényes. Például tételezzük fel, hogy van ez a programkódod:

```
class ScopeDemo
{
	static void Main( string[] args )
	{ // zárójel A
	int x = 10;
	} // zárójel B
	static void blabla()
	{
	// x = 20; <—- EZT NEM CSINÁLHATOD!
	}
}
```

Az *x* változó azt mondta, hogy van egy érvényesség az A és B zárójel között. Ha próbálnál hivatkozni arra az *x*-re azon két zárójelen kívül, akkor a C# fordító furcsán nézne rád, és megkérdezné, hogy mi a fenéről beszélsz.

Elég egyszerűnek tűnik, ugye? Itt egy másik példa:

```
static void Main(string[] args)
{
	if( 2 == 2 )
	{ // zárójel A
	int y;
	} // zárójel B
	// y = 10; <—- EZT NEM CSINÁLHATOD!
}
```

Az *if* blokk ebben a kódban mindig végrehajtódik, mert 2 nyilván mindig egyenlő 2-vel; de ez most lényegtelen. Az A és B zárójeleken belül egy új változó, az *y* jön létre, és aztán az *if* blokk véget ér. De ennek az *y*-nak csak azon két zárójel között van érvényessége, ami azt jelenti, hogy a zárójeleken kívül nem tudja semmi elérni; tehát ha az *y*-t az *if* blokkon kívül próbálod használni, akkor a számítógép hibaüzeneteket hány ki neked, mert fogalma sincs, hogy mi az az *y*.

De van még egy példa, amit szeretnék neked megmutatni:

```
static void Main(string[] args)
{
        for( int x = 0; x < 10; x++ )
        {
        // valamit csinálni itt
        }
        // x = 10; <—- EZT NEM CSINÁLHATOD!
}
```

Ebben az utolsó példában létrehoztál egy új *x* változót a *for* cikluson belül, és hozzáférhetsz *x*-hez bárhol a zárójeleken vagy a *for* cikluson belül, de azonkívül sehol.

3. fejezet

Egy rövid bevezetés az osztályokba

Eddig a pontig már eléggé felkészültnek kellene érezned magad ahhoz, hogy készíts egy nagyon egyszerű C# programot. De még igazán nem tudod, hogy hogyan használd a C# további erőteljes tulajdonságait, ezért ebben a fejezetben megmutatom ezeket. Nem szándékozom mindenen végigmenni a nyelvben – ahhoz egy sokkal nagyobb könyvet kellene írnom, és nem lenne elég helyem, hogy a játékprogramozási anyagot belegyömöszöljem. Most főleg azon dolgokat szándékozom végigvenni, amelyek a játékprogramozáshoz szükségesek.

Értékek a hivatkozásokkal szemben

Különböző módokon lehet egy fordítónak az adatokról beszélni, és a C#-nak két módja van ezt véghezvinni. Az összes adattípus a C#-ban a következő két kategória egyikébe kerül:

- érték típusok
- referencia típusok

A következő szakaszokban kifejtésre kerülnek ezen különböző adattípusok.

Érték típusok

Egy érték típus jellemzően egy kicsi adat darab, amelyre a rendszer nagyon kis időgazdálkodást költ. A 2. fejezetben már használtál érték típusokat az összes beépített szám adattípussal. Minden, ami a táblázatban fel volt sorolva – az *int*-ek, a *float*-ok – egy-egy érték típus.

Megjegyzés: az érték típusok a rendszer vermen (stack) jönnek létre. Nem szükséges tudnod, hogy mi az, de ha érdekel, akkor erősen ajánlom, hogy nézz utána. Ez a téma túlmegy ezen könyv hatókörén, tehát nincs elég helyem ezt kifejteni itt, de neked nagy segítség annak megértése, hogy pontosan hogyan működnek a számítógépek, mert így programjaidat még gyorsabbá és hatékonyabbá teheted.

Az értéktípusokat elég egyszerű és egyértelmű használni, ahogy láthatod is ebben a kódban:

```
int x = 10, y = 20;
x = y; // y értéke átmásolva x-be
y = 10; // y 10-re állítása
```

A beépített adattípusokkal együtt a szerkezetek szintén érték típusok, melyet később részletesebben felfedek ebben a fejezetben.

Referencia típusok

A referencia típusok (hivatkozások) teljesen különböznek az érték típusoktól. Az osztályok mindig referencia típusok. Ezek az adat közvetlen tárolása helyett egy címet tárolnak magukon belül, és az a cím mutat az aktuális adatra valahol a számítógépben.

Egy referencia típus megadása

A legnagyobb különbségek egyike az értékek és referenciák között a deklarálásuk módja. Egy referencia típust a *new* kulcsszóval kell létrehozni (tételezzük fel, hogy van egy *valami* nevű osztályunk):

```
Valami x = new Valami();
```

Első látásra nagy munkának látszik, de hozzá fogsz szokni. Alapjában véve a kód két feladatot hajt végre. Ezek:

1. egy *x* nevű új referencia típus létrehozása, és

2. egy *valami* nevű új objektum létrehozása a halmon és *x* rámutatása.

Megjegyzés: a halom egy másik része a számítógépnek, ami memóriát tárol. Nincs itt elég helyem bővebben kifejteni; ez egy másik olyan dolog, amelynek magadtól kell utánanézned, ha érdekel.

Természetesen nem kell egyszerre csinálnod az egészet. Könnyen részekre bonthattad volna, például így:

```
Valami x;
x = new Valami();
```
Ez tőled függ.

Játék a referenciákkal

Most itt az ideje a játéknak a referenciák körül, amelyet még nem csináltál ezelőtt. Sajnos a hivatkozások nem pontosan olyan módon működnek, mint az érték típusok, és ez először zavarba ejtő lehet.

Ez az, ahol a referenciák egy kis trükk kapása felé haladnak. Muszáj emlékezned mindenkor, hogy hivatkozásokat használsz, különben végezni fogsz a programozással azzal, hogy nem azt csinálja, amit elvársz tőle, hogy csináljon. Például próbáld meg kitalálni, hogy mit csinál ez a kód:

```
Valami x = new Valami();
Valami y = new Valami();
y = x;
// némi művelet végrehajtása itt, hogy y megváltozott
```
Bizonyára azt gondolod, hogy miután ez a kód végrehajtódik, akkor *x* maradna az eredeti állapotában, *y* pedig megváltozna, igaz? Rosszul gondolod! Mindkettő megváltozott. Légy elnéző velem szemben – első látásra egy kissé bonyolult, de van értelme.

Alapvetően a sor, amely igazán összekuszál mindent, a következő:

```
y = x;
```
Mit hajt ez valójában végre? Valószínűleg *x*-ből *y*-ba akartad másolni az adatot, de az nem történt meg. Helyette, mivel mindkettő referencia típus, a számítógép rámutatja *y*-t ugyanarra az adatra, amire az *x* is rámutat. Tehát *x* és *y* most ugyanarra az adatra mutatnak a memóriában, és bármilyen művelet végrehajtása az *y*-on ugyanazt eredményezi *x*-nek is.

Megjegyzés: ha tényleg másolni akarsz egy referencia típust egy új referencia típusba, akkor ahelyett, hogy csinálsz két hivatkozást ugyanarra az adatra mutatva, használnod kell egy beépített C# függvényt, hogy végrehajtsd az osztály egy klónozását. Ezt a 10. fejezetben láthatod a JatekObjektum osztállyal.

Szemét gyűjtés

Az iménti referenciás példával kapcsolatban megjegyezhetnéd, hogy *y*-nak volt adva némi memória, mely aztán figyelmen kívül hagyódik, miután hozzárendelődik *x*-hez. Mi történik azzal a memóriával, amire y mutat?

Régebbi nyelvekben, mint a C, a memória elveszne örökre. Valami olyasmit csinálnál, amit *lógó mutató*-nak (*dangling pointer*; a mutatók hasonlítanak a referenciákhoz) neveznek: a számítógép tudja, hogy a memória használt, de a programod elfelejtette, hogy hol volt, és nem leszel képes visszanyerni azt a memóriát, míg be nem zárod a programodat.

A C# a szemét gyűjtéssel (garbage collection) oldja meg ezt a problémát. Minden alkalommal, amikor egy új adatdarabot hozol létre C#-ban, a .NET futási idő nyilvántartja, hogy mennyiszer mutatott a programod arra az adatra, és ha az a szám valamikor lecsökken 0-ra, akkor a szemét gyűjtő érzékelni fogja ezt és felszabadítja azt a memóriát valami más használatra.

Lehetetlen memóriafolyást okozni C#-ban.

Megjegyzés: oké, lehetséges memóriafolyást okozni C#-ban. De nem szándékozom megmutatni, hogyan. Bi-bí. Különben sem akarod tudni, higgy nekem.

null

Van egy különleges érték, amelyet a referenciatípusokkal használhatsz. Ezt *null*-nak hívják. A *null* érték lényegében azt jelenti, hogy „semmi”. Ha egy hivatkozást *null*-ra állítasz, akkor azt mondod a számítógépnek, hogy az semmire sem mutat. Régebbi nyelvekben a 0 érték volt használva ezt jelezni, de a *null* sokkal olvashatóbb.

A struktúrák és osztályok alapjai

A számítógép programozás régebbi napjaiban a programozási nyelvek meglehetősen egyszerűek voltak, és csak korlátozott számú változót tudtál létrehozni. Ez nyilvánvalóan eléggé korlátozottá és csúnyává tette a programokat. Például csinálhatnál programokat a következő módon egy régebbi nyelvben:

```
int UrhajoPajzs;
int UrhajoEnergia;
int UrhajoUzemanyag;
int EllensegPajzs;
int EllensegEnergia;
int EllensegUzemanyag;
```

Ahogy el tudod képzelni, ez kusza rendetlenséget okoz, és nehezen kezelhetővé teszi a kódodat.

Az osztályok és struktúrák használata megkönnyíti az életedet az adatok bekapszulázásával könnyen használható adatcsomagokba.

Osztályok és struktúrák létrehozása

Az ötlet az osztályok és struktúrák mögött alapvetően az, hogy létrehozd a nagyon sajátos objektumfajtádat, felhasználva azokat a darabokat, amelyek már léteztek a nyelvben. Egy *struktúra* lényegében egy adattípus, amely tud hordozni más adatdarabokat belül, lehetővé téve, hogy felépítsd saját adattípusaidat; olyanfajta, mint egy építőblokk. Például itt egy struktúra, amely meghatároz egy egyszerű űrhajó objektumot C#-ban:

```
struct Urhajo
{
        public int uzemanyag;
        public int pajzs;
        public int energia;
}
```

Megjegyzés: a *public* kulcsszó közli a fordítóval, hogy bármely függvény bárhol hozzáférhet az adathoz a struktúrán belül. De emiatt most nem kell aggódnod; el fogok menni még mélyebbre erről ebben a fejezetben. Ha a *public* szó elmarad, akkor a számítógép feltételezi, hogy nem akarod, hogy az osztályon kívüli dolgok hozzáférhessenek az adataihoz.

Megjegyzés: osztály létrehozásához egyszerűen cseréld le a *struct* szót *class*-ra a fenti példában.

Most a programodon belül létrehozhatod a sajátos űrhajó változóidat:

```
Urhajo jatekos;
Urhajo ellenseg;
jatekos.uzemanyag = 100;
ellenseg.uzemanyag = 100;
```

Meglehetősen egyszerű, ugye?

Különbségek struktúrák és osztályok között

C#-ban van néhány alapvető különbség az osztályok és struktúrák között. A struktúrák könnyűsúlyú konstrukciókat jelentenek, rendszerint nagyon egyszerűek, és nincs sok összetett funkció bennük. A struktúrák rendszerint kisebbek, mint az osztályok, így a C# mindig értéktípusként fogja létrehozni ezeket (ez azt jelenti, hogy mindig a veremben fognak létrejönni).

Az osztályok a struktúrákkal ellentétben mindig referencia típusok, és így mindig a halmon jönnek létre. Az osztályoknak sok funkciójuk van, de ahelyett, hogy most mindet bemutatnám neked, kifejtem őket amikor szó esik róluk.

Függvények elhelyezése osztályaidban és struktúráidban

Az osztályoknak és struktúráknak nemcsak adattárolási képességük van, hanem bizonyos műveleteket is végre tudnak hajtani, ha megadod nekik ezt a lehetőséget. Például akarhatnád könnyen és gyorsan visszaállítani az adatok összességét az űrhajóban 100-ra; egy függvény nélkül ez a következőképpen nézhetne ki:

```
jatekos.uzemanyag = 100;
jatekos.pajzs = 100;
jatekos.energia = 100;
```

Természetesen ez nem olyasvalami, amit mindig csinálni akarsz mindenhol, tehát miért nem helyezed el inkább egy függvényen belül, és így megváltoztatva az *Urhajo* osztályt ilyen kinézetűre (az új rész félkövéren):

```
struct Urhajo
{
	public int uzemanyag;
	public int pajzs;
	public int energia;

	public void Ujratoltes()
	{
		uzemanyag = 100;
		pajzs = 100;
		energia = 100;
	}
}
```

Most már csak hívnod kell az Ujratoltes függvényt egy űrhajón, mikor vissza akarod állítani annak összes változóját:

```
jatekos.Ujratoltes();
```

Visszatérő értékek

A függvények nemcsak feladatokat hajtanak végre, de visszatérhetnek értékekkel is. Például, mondjuk van egy űrhajód; tudod, hogy mennyi üzemanyaga és energiája van, de nem vagy biztos abban, hogy az energia meddig fog kitartani. Ennek kiszámítására felhasználsz egy képletet – mondjuk kapsz két órányit minden egyes energiaegységből. Hogy kitaláld, mennyi időd van hátra a jelenlegi energiaszinten, csinálhatnál valami ilyesmit:

```
int orakhatra = jatekos.energia * 2;
```

Nos, ez egy módja a probléma megoldásának, de nem igazán egy nagyszerű megoldás. Később a játékban elhatározhatod, hogy minden energiaegység három órányi energiát ad kettő helyett. Hogy véghezvidd ezt a változtatást, végig kellene menned az egész kódodon, megtalálni a helyeket, ahol 2-t használtál és átírni 3-ra. Nem túl mókás.

Tehát csináljuk ezt a folyamatot egy függvényen belül!

```
int MaradekEnergiaIdeje()
{
	return energia * 2;
}
```

Ta-da! Az *int* szócska a függvénynév előtt elmondja a fordítónak, hogy milyen típus tér vissza a függvényből, és a *return* kulcsszót használod az érték visszaadására. Ha nem akarsz semmilyen értéket visszaadni, akkor használd a *void*-ot az *int* helyett.

Illene megjegyezned, hogy a *return* kifejezés az azonnali kilépést okozza a függvényből. Ha ránézel a következő kódra, látni fogod, hogy egy része soha nem hajtódik végre:

```
int Fuggveny()
{
        return 0;
int x = 10; //ezt soha nem hajtja végre
}
```

Megjegyzés: a C# fordító elég okos, hogy tudja, a kód soha nem fog végrehajtódni, ezért rádordít az ilyen kódírás miatt.

Szintén megengedett többszörös *return* kifejezések használata a kódodban:

```
int Fuggveny()
{
        if (valami)
                return 0;
        return 1;
}
```

Ebben a kódban ha *valami* értéke igaz (persze tételezzük fel, hogy létezik), akkor 0 tér vissza, egyébként 1.

Paraméterek

Lehetővé van téve, hogy adj egy függvénynek paramétert, hogy dolgozzon vele. Példám a hátralévő idő számításáról az elérhető energiaszinteddel, egy eléggé egyszerű számítás, amely nem használ paramétereket, de meg tudom ezt változtatni, hogy még rugalmasabbá tegyem.

Mi van akkor, ha egy űrhajón az energiaszint használata néhány külső tényezőtől függ, mint például hogy mekkora a sugárzás a csillagrendszerben (nézd el nekem – én csak kitaláltam valamit)? Mondjuk az alacsonyabb sugárzási szint a rendszerben alacsonyabb energiaszintet igényel. Tehát ha átírod az előző függvényt az elgondolásom alapján, az így nézhet ki:

```
int MaradekEnergiaIdeje( int sugarzasiszint )
{
	return (energia * 2) / sugarzasiszint;
}
```

Ha a sugárzási szint 1 volt (teljesen kitalált adatmértékeket használok itt) és az energiád 100, akkor a hátralévő órák száma 200. Ha a sugárzási szint 2, akkor 100 órád van hátra, és ha 3, akkor pedig 66 órád marad.

Hívhatnád a függvényt eképpen:

```
int maradekido = jatekos.MaradekEnergiaIdeje( 1 );
```

Többszörös paraméterek

Eljön majd az idő, amikor egynél több paramétert akarsz átadni, és a C# lehetővé teszi ezt:

```
int Fuggveny1( int parameter1, float parameter2, double parameter2 );
```

Érték Paraméterek a Referencia Paraméterekkel szemben

Ez az, ahol a dolgok kaphatnak egy kis trükkösséget. Mondjuk van egy osztályod két függvénnyel, amik így néznek ki:

```
class Osztalyom
{
	public void Fuggveny1( int parameter )
	{
		parameter = 10;
	}
	public void Fuggveny2()
	{
		int x = 0;
		Fuggveny1( x );
		// Micsoda  x?
	}
}
```

Tehát mi is *x*, miután ez a kód lefut? 0 vagy 10? A válasz az, hogy 0, mert átadtad *x*-et érték szerint (by-value). Ez azt jelenti, hogy a számítógép fogja *x* értékét, másolja és elhelyezi egy *parameter* nevű új változóba. Most, mikor *parameter* megváltozik, semmi nem történik *x*-szel.

Tehát hogyan tudod megcsinálni a referencia szerinti átadást? Csak csinálj két dolgot. Először is, változtasd meg a *Fuggveny1* deklarációját:

```
public void Fuggveny1(ref int parameter )
```

Másodszor, változtasd meg a függvényhívást ilyen kinézetűre:

```
Fuggveny1( ref x );
```

Most át fogsz adni egy referenciát *x*-nek, és az *x* értéke meg fog változni.

Illene megjegyezned, hogy az osztályok mindig referencia által adódnak át. Például:

```
public void Fuggveny1(Int32 parameter )
```

Ebben az osztályban bármely Int32 (ez csak egy általam kitalált osztály) amit átadsz abba, mindig referencia szerint fog átadódni, nem pedig érték szerint.

Függvény túltöltés

Lehet egy osztályod vagy struktúrád számos függvénnyel, amelyeknek azonos a neve. Elsőre ez hülyén hangozhat, de a való életben nagyon jól működik. Például, mondjuk van két különböző módszered egy űrhajó utazási távolságának kiszámítására, amihez adott a jelenlegi üzemanyag készlete; egy módszer számításba veszi az űrhajó terhét, amit szállít, egy másik pedig csak kihagyja a terhet, és ad neked egy „legjobb forgatókönyv” eredményt. Csinálhattad a függvényeket ilyen kinézetűre:

```
public int TavolsagHatra( int tehertomege ) // teherrel
{
// csinálni itt számításokat
}
public int TavolsagHatra() // optimális, nincs teher
{
// csinálni itt számításokat
}
```

így később a kódodban hívhatnád ezeket így:

```
int tavolsag;
tavolsag = jatekos. TavolsagHatra ( 100 );
tavolsag = jatekos. TavolsagHatra ();
```

Túltöltheted a függvényeket amennyiszer csak akarod; az egyetlen megkötés, hogy minden egyes túltöltött függvénynek különböző aláírása kell legyen. Az aláírás a beadott paraméterek által van meghatározva, nem a visszatérési érték által.

Tehát csinálhatod ezt:

```
void Fuggveny1();
void Fuggveny1( int p1 );
void Fuggveny1( float p1, int p2 );
```

És csinálhatod ezt is:

```
int Fuggveny1();
void Fuggveny1( int p1 );
float Fuggveny2( float p1, int p2 );
```

De egyáltalán nem csinálhatod ezt:

```
int Fuggveny1();
float Fuggveny1(); // azonos aláírás! HIBA!
```

Konstruktorok

A konstruktorok („létrehozók”) igazán hasznos tulajdonságai a legtöbb modern programozási nyelvnek. Lehetővé teszik számodra osztályaid és struktúráid automatikus inicializálását.

Vissza a régi rossz időkbe, amikor létrehoztál egy új szerkezetet vagy osztályt, igazán nem volt ötleted, hogy mi volt benne. Látod, a számítógép nem törli a memóriát, így mikor leálltál egy memóriadarab használatával, a számítógép csak megjegyezte, hogy nincs többé használatban, és aztán feltálalta az első dolognak, aminek kell, azonos adattal benne. Ez azt jelenti, hogy gyakran lennének struktúráid tele hulladék adatokkal és ez sok problémát okozhat, ha csak anélkül kezded a struktúrát használni, hogy megbizonyosodnál annak érvényes értékeiről.

A konstruktorok alapvetően függvények, amelyeket a C# automatikusan hív, amikor létrehozol egy új osztályt.

Alapértelmezett konstruktorok

Hadd mutassak egy egyszerű példát egy konstruktorról egy osztályon az elinduláshoz:

```
class Urhajo
{
	public int uzemanyag;
	public Urhajo() // alapértelmezett konstruktor
	{
		uzemanyag = 100;
	}
}
```

A félkövér kód az, amit alapértelmezett konstruktornak hívnak. Valahányszor létrehozol egy új űrhajót, az Urhajo függvény (jegyezd meg, hogy azonos a neve, mint az osztályé) önműködően hívódik.

Ez a kód létrehoz egy űrhajót, hozzá 100 egységnyi üzemanyaggal:

```
Urhajo u = new Urhajo();
```

Nem-alapértelmezett konstruktorok

Mint a szabályos függvényeknél, a konstruktoroknak szintén lehetnek túltöltött változataik, ami hasznos, mikor szükséged van extra adat nyújtására, mikor létrehozol egy osztályt. Itt egy példa, ami megmutatja, hogyan hozz létre egy nem-alapértelmezett konstruktort, amely üzemanyag mennyiséget vesz át egy változóban, hogy hozzárendelje egy űrhajóhoz:

```
public Urhajo( j_uzemanyag )
{
        uzemanyag = j_uzemanyag;
}
```

Most, mikor létrehozol egy új űrhajót, hívhatod ezt az új konstruktort így:

```
Urhajo u = new Urhajo(50); //létrehoz egy űrhajót 50% üzemanyaggal
```

Annyi konstruktort hozhatsz létre, amennyit akarsz, ameddig mindegyiknek különböző aláírása van.

Struktúrák és konstruktorok

A struktúráknak lehetnek konstruktoraik is, de van egy csalafintaság: a struktúráknak nem lehetnek alapértelmezett konstruktoraik. A Microsoft igényli, hogy legyen ez elvégezve a hatékonyság érdekében, mert a struktúrák feltételezhetően könnyűsúlyúak. Tehát a struktúráknak lehetnek nem-alapértelmezett konstruktoraik, de nem lehetnek alapértelmezett konstruktoraik.

Destruktorok

Ha konstruktorok hívódnak valahányszor egy osztály létrejön, akkor destruktorok („megsemmisítők”) hívódnak valahányszor egy osztály megsemmisül. Régebbi nyelvekben, mint a C++, a destruktorok nagyon fontosak voltak, mert mindig fel kellett szabadítanod azt a memóriát, amit nem használtál többé. De mióta a szemétgyűjtés megjelent, a destruktorok leginkább a dinoszauruszok útján mennek. Igazán nincs rájuk többé szükség, de azért még benne vannak a nyelvben, ha netán mégis kellenének.

Egy alap példa

Ez az, ahogy egy destruktor kinéz:

```
class Osztalyom
{
        ~Osztalyom()
        {
                // kód itt
        }
}
```

Egy olyan nyelvben, mint a C++, ez az, ahol automatikusan csinálná az osztály a kért memóriája törlését. De minthogy a .NET futási környezet gondot visel az összes memóriára, neked igazán nem szükséges azt itt csinálnod.

Tipp: a leghasznosabb célja egy destruktornak a C#-ban a *példány számlálás*, ahol tudni akarod, hogy egy általad létrehozott egyéni osztálynak mennyi példánya van bármely adott pillanatban. Alapvetően az ötlet az, hogy egyet kell hozzáadni a számlálóhoz valahányszor a konstruktor hívódik, és egyet kell kivonni a számlálóból a destruktor minden egyes hívásakor. Tehát ha egy osztálynak nincsenek már példányai, akkor a példány számláló 0 lehet.

Késleltetett megsemmisítés

Másik tény a destruktorokról, hogy nem hívódnak azonnal. Nézd ezt a kódot például:

```
Urhajo u = new Urhajo(); // új űrhajó
u = null; // a hivatkozás az űrhajóra elveszett
```

Ebben a kódban egy új űrhajó jött létre és egy hivatkozás rá, hozzárendelve *u*-hoz. A sokkal későbbi sorban a hivatkozás *null* értékűre állítódik, tehát most van valahol a rendszeredben egy űrhajó, de nincs ötleted, hogy hogyan juthatsz hozzá.

Ez igazán nem nagy ügy neked, mert tudod, hogy később a hulladékgyűjtő megtalálja és megsemmisíti. De tud okozni némi problémákat, amikre azonnal nem is gondolnál.

A dolog a C#-ban az, hogy igazán nincs ötleted, mikor szándékozik egy osztály példány megsemmisíthetővé válni. Beállíthatod az összes hivatkozást annak *null*-ra, de a rendszer hosszú ideig megtarthatja a példányt, miután beállítottad a referenciákat.

Ezért a fő probléma destruktorokkal az, hogy igazán nem tudod, mikor lesznek hívva. Nem számíthatsz egy osztály azonnali megsemmisítésére, tehát ne tételezd fel, hogy megsemmisül, mikor törlöd az összes referenciát.

Vannak módszerek eköré, de minthogy valószínűleg nem fogod a destruktorokat sokszor használni, ezért nem foglak terhelni annak elmondásával.

Megjegyzés: a struktúráknak nincsenek destruktoraik, mert feltételezhetően igazán könnyűsúlyú konstrukciók.

Még fejlettebb osztály trükkök

Elég magabiztosságot kellene már érezned az osztályokkal most, hogy létrehoztál egy egyszerűt kisebb feladatok végrehajtására, mint adattárolás és egyszerű számítások végrehajtása. Ámbár az osztályoknak sokkal több tulajdonsága van, mint amit eddig láttál.

Az öröklés alapjai

Egy objektum-orientált programozási nyelv egyik legnagyobb előnye az *öröklés*. Nem szándékozom nagyon mélyen belemerülni a haladóbb öröklési témába, de e könyv elolvasása után képesnek kellene lenned értened és használnod az öröklést.

Az öröklés lehetővé teszi, hogy valós módon modellezd a programjaidat képességek rangsorának meghatározásával, lehetővé téve osztályaidnak a hasonlatosságot a való-világ objektumaihoz. A legegyszerűbb módja az öröklés elképzelésének, ha az állatok tudományos besorolására gondolsz.

Az öröklés egy példája

Ha visszaemlékszel a középiskolai biológiára, akkor tudod, hogy az emlősöknek vannak bizonyos közös tulajdonságaik, mint például az elevenszülés és a négy kamrás szív. Egy számítógépes programban létrehozhattad az emlős osztályodat és hozzáadhattad azokat a jellegzetességeket.

De annak ellenére, hogy az összes emlős megoszt néhány jellemzőt, ők nem osztják meg azonos jellemzőik mindegyikét. Az embereknek két lábuk van, a teheneknek négy – magától értetődően nem használhatsz egy emlős osztályt az

emberek és tehenek képviselésére. Ugyanabban az időben butaság létrehozni két teljesen különböző osztályt az emberek és tehenek reprezentálására. A kód, amit írsz a közös tulajdonságok képviselésére egy ember és egy tehén között, meg lenne kettőzve a két osztály között, és feladod a tervezést, ami így hibákhoz vezet.

Figyelmeztetés: a megkettőzött kód mindig egy rossz ötlet. Valamikor szükséged lesz arra, hogy megváltoztasd a kód egy darabját, amit írtál, és ha az a kód különböző helyeken van, akkor garantálom, hogy el fogod felejteni, hogy hol vannak a részei.

Ez az a pont, ahol az öröklés belép a játékba. Az öröklés lehetővé teszi, hogy létrehozz egy új osztályt és önműködően használja másik osztály összes tulajdonságát. Ezt nevezik „az egy" kapcsolatnak (a tehén *az egy* emlős).

Az öröklés használata

Az öröklés meglehetősen egyszerűen használható a C#-ban. Először létre kell hoznod egy *alap osztály*t, mely az öröklési rangsorod tetejére kerül (az emlős osztály az előző szakaszban, például).

Mondjuk létre akarsz hozni egy alap űrhajó osztályt, egy osztályt, amely le fogja írni a közös jellemzőit az összes létező űrhajónak. Minthogy az összes űrhajó tartalmaz üzemanyagot, létrehozhatod azt az alap osztályodban:

```
class Urhajo
{
        public int uzemanyag;
};
```

Megjegyzés: tartsd észben, hogy ezek a példák nagyon egyszerűek, minimálisak a kóddal, azért, hogy a lényegre összpontosítsanak. Megpróbállak nem összezavarni téged.

Tehát most van egy űrhajód, de minden amije van, az az üzemanyag. Most talán létre akarsz hozni egy hadihajót, melyen vannak fegyverek:

```
class Hadihajo : Urhajo
{
        public int fegyverek;
}
```

Elmondtad a fordítónak, hogy egy hadihajó az egy űrhajó, mégpedig azáltal, hogy az osztálynév után elhelyeztél egy kettőspontot és a nevét az osztálynak, amelyből öröklődik.

Például létrehozhatsz egy teherhajót is:

```
class Teherhajo : Urhajo
{
        public int rakomany;
}
```

Most használhatod a megadott jellegzetességeket és az alap osztály tulajdonságait is:

```
Hadihajo h = new Hadihajo();
h.fegyverek = 100; //új tulajdonság
h.uzemanyag = 100; //az Urhajo-tól öröklött tulajdonság
Teherhajo t = new Teherhajo();
t.rakomany = 100; //új tulajdonság
t. uzemanyag = 100; //az Urhajo-tól öröklött tulajdonság
```

És nagyjából ez minden az alap öröklésről.

Hozzáférési szintek és adat rejtés

Eddig a *public* szót láttad a kód példákban, de még nem fejtettem ki, hogy mit jelent vagy miért van ott. Alapvetően, mikor valamiről azt mondod, hogy *public*, akkor elmondod a fordítónak, hogy minden hozzáférhet. Ha adsz egy osztálynak egy public integer-t, akkor azt bármi olvasni tudja vagy megváltoztatni. Nagyjából ezen a módon dolgozott az összes számítógép nyelv, míg fel nem bukkant az *adat rejtés* ötlete.

Az adatrejtés mögötti ötlet

Az adatrejtés egy elgondolás, amely lehetővé teszi neked lényegében adat elrejtését a programod más részeitől. Csodálkozhatsz azon, hogy vajon mi a fenéért is akarnál adatot elrejteni. Nos, erre a válasz megér egy kis kifejtést.

Nem akarod engedni, hogy akárki érinthesse az adatodat; nem tudod, hogy valaki akar-e valami károsat tenni vele. Vegyünk egy űrhajót, például. Egy űrhajóban amikor az üzemanyagszint 0 lesz, a hajtóműveknek le kellene állniuk. Most képzeld el az összes helyet a kódodban, ahol függvényeid vannak az űrhajód üzemanyagszintjének módosítására. Talán a hajódból kifolyik némi üzemanyag, mikor eltalálja egy lövedék. Vagy talán kapsz egy ráadás üzemanyagtartályt, némi üzemanyag hozzáadásának a tartályodhoz. Milliónyi lehetőség van.

Tonnányi hely van, amely megváltoztatja az üzemanyagot, és mindegyiküknek le kell állítaniuk a hajtóművet, ha az üzemanyagszint eléri a 0-t, ami egy rossz dolog. Továbbá valaki megpróbálhat több üzemanyagot elhelyezni a tartályaidba, mint amennyit jelenleg tárolni képes, melyet nem akarod, hogy megtörténjen. Lényegében, ha engeded bármely függvényednek, hogy módosítsa az adatodat, azok a függvények véletlenül (vagy még céltudatosan is) elronthatják, és ez határozottan nem jó dolog.

Másik kérdés, amit fel kell tenned, hogy vajon a többi kódodnak igazán szükséges-e mindent tudni az adatodról az első helyen. Az űrhajód ellenőrizheti a Weebul kondenzátor beömlő tömítés hőmérsékletét, de az űrhajón kívül az összes osztálynak tudnia kell, hogy az űrhajónak van Weebul kondenzátor beömlő tömítése? Valószínűleg biztosan feltételezed, hogy semmi más nem törődik vele, kivéve az űrhajót, és ezért bármi, ami próbál törődni valami ilyesmivel, valószínűleg valami rossz lesz. A legjobb elrejteni ezt az adatot, és így nem zavarod össze később a dolgokat.

Hozzáférési szintek

A C#-nak van számos meghatározott hozzáférési szintje. Egyiküket már láttad, ez a *public*. Ahogy kitalálhatod, a *public* azt jelenti, hogy bármi elérheti azt a funkciót.

A másik két népszerű hozzáférési szint a *protected* és a *private*.

Megjegyzés: van még két további hozzáférési szint, az *internal* és a *protected internal*, de ezek közel sem olyan közönségesek, mint a másik három, és valószínűleg nem lesz szükséged rájuk, hacsak nem valami igazán összetett programokat készítesz.

A *private* hozzáférés azt jelenti, hogy nem érheti el a kódod többi része a funkciót, kivéve az osztályt magát. Még az öröklött osztályok sem. Vizsgáld meg a következő kódot:

```
class Urhajo
{
        private int uzemanyag;
};
class Hadihajo :  Urhajo
{
        public void valamiFuggveny()
        {
```

```
            uzemanyag = 10; //HIBA! Nem tudja látni az üzemanyagot!
        }
};
Urhajo u = new Urhajo();
u.uzemanyag = 10; //HIBA! Nem tudja látni az üzemanyagot!
```

A hadihajón belül a *valamiFuggveny* függvény megpróbálja elérni az *uzemanyag*-ot. Előzőleg nem volt gondod ennek elvégzésével; végül is egy hadihajó egy űrhajó, és ezért van üzemanyaga és módosítható.

De most, hogy az üzemanyag bizalmas hozzáférésű, a hadihajó nem tudja többé elérni. Az üzemanyag természetesen még ott van, de a hadihajónak nincs megengedve az érintése.

A kód többi részében sincs megengedve, hogy szabályos űrhajók érinthessék az üzemanyagot; az adat rejtve van.

Megjegyzés: ha elfelejted megadni egy függvényhez vagy változóhoz a hozzáférési szintet, akkor a C# fordító önműködően feltételezni fogja, hogy az *private* elérést használ.

A *protected* (*védett*) hozzáférési szint hasonló a *private*-hez, egy kisebb eltéréssel: bármi, ami védett, az még rejtett a kódnak az osztályon kívül, de az osztályok, amelyek öröklődnek az alap osztályból, még láthatják a funkciókat. Nézd ezt a példát:

```
class Urhajo
{
        protected int uzemanyag;
};
class Hadihajo :  Urhajo
{
        public void valamiFuggveny()
        {
            uzemanyag = 10; //Ez most rendben, mert védett!
        }
};
Urhajo u = new Urhajo();
u.uzemanyag = 10; // HIBA! Nem tudja látni az üzemanyagot!
```

Ez a példa nagyon hasonló az előzőhöz, de most az üzemanyag védett, és a hadihajó egész jól el tudja érni.

Statikus tagok

Mostanáig minden, amit láttál az osztályokon belül, azok példány tagok. Egy *példány tag* egy osztálynak egy része, amely azon osztály egyetlen példányán belül létezik. Ha van két űrhajód, és azoknak van egy integer típusú értékük, amely az üzemanyagot jelképezi, akkor két integer számod lesz, egy minden egyes hajónak.

Másrészről, lehetnek statikus tagjaid is. Egy *statikus tag* adat (vagy egy függvény) darabja, amely megosztott egy osztály összes példánya között ahelyett, hogy meg lenne kettőzve minden példánynak.

Statikus adat

Nézd a következő kódszegmenst:

```
class Urhajo
{
	public static int szamlalo;
	public int uzemanyag;
	public int teher;
};
```

Ez létrehoz egy osztálydefiníciót egy űrhajónak, ahol minden űrhajónak lesz üzemanyaga és teherszállítmánya, és az osztálydefiníció nyomon fog követni egy *szamlalo* nevű integer értéket. Hozzáférhetsz ehhez az integerhez bármikor a következő (vagy valami hasonló) kód beidézésével:

```
Urhajo.szamlalo = 10;
```

Nem szükséges, hogy legyen bármilyen űrhajó példányod azért, hogy használd ezt a változót; ez mindig létezik. Nem tartozik semelyik különleges űrhajóhoz sem; bármi tudja használni. A statikusság egy könnyű módja annak, hogy azonos funkcionalitást kapjunk, amiket a globális változók nyújtanak az olyan nyelvekben, mint a C vagy C++, ezenkívül ezek tervezési szempontból takarosabbak.

Statikus függvények

Függvények is lehetnek statikusak. Alapvetően, egy statikus függvény hívható annak szükségessége nélkül, hogy egy meghatározott példány működjön rajta. Például:

```
class Urhajo
{
	public static void FuggvenyA()
	{
		//csinál valamit
	}
};
```

Most tudod hívni ezt a függvényt egy későbbi időpontban valahogy így:

```
Urhajo.FuggvenyA();
```

Nem kell, hogy legyen bármennyi űrhajó példányod ahhoz, hogy hívd a függvényt.

Figyelem: statikus függvények nem tudnak működni adat példányon anélkül, hogy egy aktuális példány működjön. Nyilvánvalónak tűnik, mikor efelől gondolkodsz, de néhányaknak nem jut eszébe azonnal ez az elgondolás.

Tulajdonságok

Az előzőekben elmondtam, hogy az adatrejtés egy jó dolog. Valójában rossz voltam és használtam néhány rossz trükköt, hogy a példák kevésbé zavarónak tűnjenek ebben a könyvben. Általánosságban szólva, neked *soha* nem kellenének publikus adatok az osztályaidban. A kísértés néha túl erős, hogy megengedd a közvetlen hozzáférést az adatodhoz, és előbb vagy utóbb belefutsz azokba a problémákba, amikről az előzőekben beszéltem. És amikor elkezdesz hibákat kapni a semmiből, nem akarom, hogy sírva gyere hozzám.

Tartozékok

Általában, a múltban az előnyben részesített módszer publikussá tevés nélküli hozzáférhető adat készítésére az *accessor* („*tartozék*”) függvények használata volt. Ennek megvalósítására két függvényt csinálsz minden változónak: egyet megkapni az értéket, és egyet beállítani, megadni az értéket. Mint így:

```
class Urhajo
{
	protected int uzemanyag;
	public int KapUzemanyag() { return uzemanyag; }
	public void MegadUzemanyag( int p_uzemanyag) { uzemanyag = p_uzemanyag; }
};
```

És hozzáférhetsz az üzemanyaghoz, mint így:

```
Urhajo u = new Urhajo();
u.MegadUzemanyag( 10 );
int ua = u.KapUzemanyag();
```

Megjegyzés: ez a módszer figyelembe vette a biztonságot, mert megváltoztathatod, hogyan történjen később az üzemanyag elérése, anélkül, hogy összezavarnád a többi kódod bármelyikét. Például, ha egy nap elhatározod, hogy nem akarod az embereknek megengedni azt, hogy az üzemanyagot 0 alá állíthassák, akkor megváltoztathatod a *MegadUzemanyag* függvényt, így az önműködően gondot visel rá.

Ez mérnöki nézőpontból nagyon biztonságos. Viszont órákig-gépeltem-és-már-haza-akarok-menni nézőpontból ez a megoldás szívás. Sok extra kódolással járt, hogy a programodat „biztonságosabbá" tedd.

A C# megoldása: tulajdonságok

A „túl sok gépelés" probléma megoldására, ami létezett a korai nyelvekben, a C# egy *tulajdonságok*nak („*properties*") nevezett új elgondolást vezetett be. Egy tulajdonság lehetővé teszi neked, hogy kevesebb kódot használj a változóid még hozzáférhetőbbé tételéhez, míg fennmarad a tartozék függvények védelme. Itt egy példa:

```
class Urhajo
{
        protected int uzemanyag;
        public int Uzemanyag {
                get { return uzemanyag; }
                set { uzemanyag = ertek; }
        }
}
```

A félkövér kód a tulajdonságmező. Alapvetően, létrehoztam egy *Uzemanyag* (nagy U) nevű tulajdonságot, mely pontosan úgy fog cselekedni, mint egy adatdarab kívül:

```
Urhajo u = new Urhajo();
u.Uzemanyag = 10;
int ua = u.Uzemanyag;
```

Bár, ami pillanatnyilag a színhely alatt történik, az a *get* és *set* tulajdonság függvények hívása, és a kódjuk végrehajtása.

Csinálhatsz bármit, amit akarsz, akármelyik tulajdonság függvényben, de valószínűleg legtöbbször a legtöbb kódmunkát a *set* függvényben szándékozol

tenni. Például biztosíthatod, hogy többé senki se állíthassa az űrhajó üzemanyagszintjét 0 alá az *Uzemanyag.set* függvény ilyesmire átírásával:

```
public int Uzemanyag {
        get { return uzemanyag; }
        set {
                if ( ertek < 0 )
                        uzemanyag = 0;
                else
                        uzemanyag = ertek;
                }
}
```

Így most, ha végrehajtod ezt a kódot:

```
u.Uzemanyag = -10;
```

a *set* függvény önműködően 0-ra állítja az üzemanyagot -10 helyett.

Figyelem: a tulajdonságoknak nincs végtelen rekurzió érzékelésük, mely néha fájdalmas lehet a fenéknek. Például ha véletlenül *Uzemanyag = ertek* utasítást gépeltél *uzemanyag = ertek* helyett a *set* függvényen belül, akkor a számítógép automatikusan hívná újra a *set* függvényt, és a program összeomlásáig futtatva tartaná. Legyél ezzel körültekintő.

Felsorolások

Egy tulajdonsága a C#-nak (és más nyelveknek) amely könnyebbé teszi az életed, az a *felsorolt típusok*, vagy *enumerációk* („*felsorolások*”) fogalma. Hányszor találod magad írni egy programot, ahol van néhány fajta adatod, amelyek nem számok, de nem hív egy egyéni adattípust?

Képzeld el: csinálsz egy egyszerű rendszert, amiben minden űrhajó a játékodban tudni fogja, hogy milyen állapotban van – vajon mozog körbe, áll vagy vadul csatázik néhány gonosz kalózzal. Elhatározhattad, hogy használsz egy egész számot, mint ez:

```
class Urhajo
{
        int allapot;
        //többi kód itt
};
```

Rendben, tehát a 0 állapot a mozgás, 1 a megállás és a 2-es a harc. Elég jónak hangzik, ugye? Nem! Ez gonosz. Sose csináld ezt! Ez csúnya és nem leszel képes emlékezni minden egyes kód jelentésére.

A C# megoldása a felsorolások. Egy felsorolt típus magába foglal csoportosított neveket egy típusba, amely könnyen hivatkozható és olvasható. Itt egy felsorolt típus, amely jelképezi az űrhajó állapotait:

```
enum UrhajoAllapot
{
        mozgas,
        helybenallas,
        csata
};
```

Ez a kód létrehoz egy *UrhajoAllapot* nevű felsorolt típust, melynek összesen három különböző értéke van: *mozgas*, *helybenallas* vagy *csata*. Használhatod, mint így:

```
UrhajoAllapot u;
u = UrhajoAllapot.mozgas;
if ( u = UrhajoAllapot.csata )
        //kód itt
//és így tovább
```

Az enumerációk alapvetően egész számok a szavak mögött; nem szükséges azon a módon gondolnod rájuk, de jó, ha tudod. Tipikusan, az első felsorolás kapja a 0 értéket, és a többi növekszik 1-gyel.

```
int i;
i = (int)UrhajoAllapot.mozgas; //0
i = (int)UrhajoAllapot.helybenallas; //1
i = (int)UrhajoAllapot.csata; //2
```

Ha nem tetszenek az alapértelmezett értékek, akkor megváltoztathatod őket, például így:

```
enum UrhajoAllapot
{
        mozgas = 10,
        helybenallas = 12,
        csata //ez automatikusan 13 lesz
};
```

Tehát most a *mozgas*-nak 10 az értéke, a *helybenallas*-nak 12, és minthogy kifejezetten nincs megadva, fogja az előző enumeráció értékét, hozzáad egyet és így 13-at csinál a *csata* számára.

4. fejezet

Haladó C#

Névterek

A névterek viszonylag új fogalom a számítógép nyelvekben, de nagyon hasznosak, és néhányan vitathatják létezésük alapvetőségét napjainkban.

A programozásban az egyik legnagyobb probléma a *név átlapolás* („*name overlapping*”). Mondjuk létrehozol egy csomó osztályt a programodnak, és aztán elhatározod, hogy beemeled valaki más könyvtárát segítségül a programodhoz. Mi történik, ha azon személy néhány osztályának azonos a neve a tieidével, de más dolgokat csinálnak? Meglehetősen gyakran megtörténik, sajnos.

Például a Direct3D-nek és a DirectSound-nak is van *Device* nevű osztálya, és neked nyilvánvalóan nem lehet két osztályod azonos névvel. A névterek könnyebbé teszik ezt, így utalhatsz a különböző eszközökre, mint *Direct3D.Device* és *DirectSound.Device*.

Gondolhatsz úgy a névterekre, mint egy városra. Ha valakinek csupán annyit mondasz, hogy a Fő utcán laksz, akkor ezzel nem mondtál neki túl sokat, mivel ezernyi Fő utca van szerte az országban. Azért, hogy pontosan rámutass arra, hogy hol élsz, el kell mondanod annak a személynek azt is, hogy melyik városban laksz. Egy névtér létrehozása olyan, mint meghatározni a városodat és utcádat, amelyben el tudsz helyezni egy bizonyos osztályt (utca), belül egy megadott névtéren (város), ezért rendesen tudod szegmentálni a programjaidat.

Majdnem bármit elhelyezhetsz egy névtérben, beleértve egy osztályt, egy struktúrát, egy felsorolást és még másik névteret is!

A névterek menők, mert bújtathatod őket – elhelyezhetsz még névtereket a létező névtereken belül. Például a .NET keretrendszer jön a *System* névtérrel, és ezen belül még vannak más névterek, mint a *System.Data* és *System.Collections*.

Kiterjesztve az előző hasonlatot, a bújtatott névterek lehetővé teszik nagyobb rangsorok létrehozását, mint például hogy az Egyesült Államok létezik Észak-Amerikán belül, Kalifornia létezik az Egyesült Államokon belül, Los Angeles létezik Kalifornián belül és a Fő utca létezik Los Angeles-en belül.

Névterek létrehozása

Itt van néhány kód, amely bemutatja egy névtér használatát:

```
namespace Fejezet04
{
        class Urhajo
        {
        //valódi kód mehet itt
        };
        class Urallomas
        {
        //valódi kód mehet itt
        };
}
```

És aztán a névtéren kívül hozzáférhetnél azokhoz az osztályokhoz így:

```
Fejezet04.Urhajo u = new Fejezet04.Urhajo();
```

Másik nagyszerű tulajdonsága a névtereknek, hogy fel lehet őket osztani több szakaszra. Például lehet neked ez egy állományban:

```
namespace Fejezet04
{
        class Urhajo
        //bla bla
}
```

és aztán elhelyezve az űrállomás egy másik fájlban:

```
namespace Fejezet04
{
        class Urallomas
        //bla bla
}
```

A C# fordító önműködően összeilleszti a névtereket neked, így neked nem kell mindent egy nagy állományban elhelyezned.

Névterek használata

Mikor egy névtéren belül vagy, bármit használhatsz abban a névtérben minősítés nélkül. Ha hozzáfértél az *urhajo* osztályhoz az *urallomas* osztályon belül az előzőleg mutatott példában, akkor csak begépelhetted volna, hogy **Urhajo** és a C# feltételezte volna, hogy a *Fejezet04.Urhajo*-ról beszélsz, mert azonos névtérben voltál.

Viszont ha a névtéren kívül vagy, akkor módosítanod kell a névteret úgy, hogy először a *Fejezet04*-et teszed ki.

Persze újra és újra gépelni, hogy *Fejezet04.Urhajo*, egy idő után bosszantó, különösen ha tudod, hogy csak a Fejezet04-ből szándékozol használni űrhajókat, és máshonnan nem. Szerencsére elmondhatod a C# fordítónak a *using* kulcsszó használatával, hogy egy megadott névtéren belül akarsz mindent használni. Ez így nézhet ki:

```
//a forrásfájl tetején:
using Fejezet04;
//később az állományban:
Urhajo u = new Urhajo();
```

Megjegyzés: a *using* kulcsszó csak bizonyos helyeken szerepelhet. A kulcsszót nem lehet elhelyezni osztályokon, struktúrákon vagy felsorolásokon belül, de ezeken kívül majdnem bárhová igen. Rendszerint jó szokás a forráskód állományaid tetején elhelyezni a *using* kifejezéseket, így azonnal tudod, hogy az állományod más könyvtárainak mire van szüksége.

Névtér átnevezés (aliasing)

A bújtatott névterek okozhatnak nagy fájdalmat a fenéknek. Nem láttad még azokat, de amikor találkozol velük a forró és nehéz DirectX-ben, teli tüdődből kiáltasz majd fel, hogy „Átkozott Microsoft!”… vagyis, hacsak nem vagy tájékozott a névtér átnevezésről.

Minden összefüggő a Direct3D-vel a *Microsoft.DirectX.Direct3D* névtéren belül. Tehát ha hozzá akarsz férni egy Direct3D eszközhöz, akkor be kellene gépelned, hogy *Microsoft.DirectX.Direct3D.Device*. Ugh, helyes? Szerencsére a névtér átnevezés sokkal jobbá teszi! Alapvetően, foghatsz egy névteret, és mondhatod a C#-nak, hogy használjon más nevet hozzá.

Itt egy másik név a Direct3D névtérnek:

```
using D3D = Microsoft.DirectX.Direct3D;
D3D.Device d; //a Microsoft.DirectX.Direct3D.Device helyett d;
```

Látod, mennyivel könnyebbé teszi a dolgokat a névtér átnevezés?

Polimorfizmus

A *polimorfizmus* témája nagy és összetett – egyetemek egész kurzusokat szánnak erre. Én csak egy korlátozott futó pillantást adhatok erre a tárgyra ebben a szerény könyvben. De neked nem is kell megtanulnod mindent a polimorfizmus igazán összetett részeiről.

Szó szerint a *polimorfizmus* szó azt jelenti, hogy „sokalakúság”. A számítógép programozásban a polimorfizmus lehetővé teszi az egymásra hatást sok különböző objektummal anélkül, hogy aggódnod kéne azon, hogy mik is azok az objektumok valójában.

Egy való világ példája a polimorfizmusnak számítógép programozás értelemben, ha gondolsz egy autóra. Beszállhatsz egy autóba, elindíthatod, lenyomhatod a gázpedált, és tudod, mi fog történni: az autó elindul! Most szállj ki az autóból, szállj be egy teljesen másik autóba és csináld ugyanazokat a dolgokat: az az autó is elindul! Mindkét autónak azonos felülete van, és neked igazán nem kell azzal törődnöd, hogy belül a motorok hogyan működnek. Ha egy négyhengeres autót, ha egy nyolchengeres verseny bestiát vagy egy politikailag megfelelő elektromos autót vezetsz, tudod, hogy mikor rálépsz a gázpedálra, az autó el fog indulni. Ez a polimorfizmus legjobban. A számítógép elmondja egy objektumnak, hogy dolgozzon, és az objektum, nem számít mi az, elindul dolgozni.

Tekintsünk a polimorfizmusra egy játékhelyzetben. Mondjuk programozol egy egyszerű lövöldözős játékot – egy űrhajóban vagy, röpködsz körbe és lézereket lősz mindenre. Amikor a lézer eltalál valamit, az objektum reagál valahogy, ugye? Egy polimorfikus rendszerben ez csinálna érzékelést minden objektumnak, hogy tudja, hogyan kellene reagálnia, mikor eltalálják. Egy számítógép-vezérelt űrhajó elviselne némi károsodást, és elmondaná a mesterséges intelligenciájának, hogy támadja meg, aki a lézert lőtte. Egy játékos irányította űrhajó elviselné a kárt, és talán küldene némi erő-visszacsatolási jelet a játékos botkormányának. Egy aszteroida széthasadna sok darabra. A lényeg, hogy az aktuális játék motornak igazán nem kell törődnie azzal, hogy az objektumok hogyan reagálnak, mikor lézertalálatot kapnak – az

egész, amit csinálnia kell, hogy elmondja az objektumnak, hogy találatot kapott, és hagyni, hogy az törődjön a részletekkel. Kifejtem ezt az elgondolást később egy kicsit részletesebben.

Alap polimorfizmus

Mondjuk van egy nagyon alapszintű öröklési fád: egy gyökér és két levél. A gyökér az űrhajó, a két levél pedig a hadihajó és a teherhajó. Játszadozhatsz velük, mint általában:

```
Urhajo u = new Urhajo();
Teherhajo t = new Teherhajo();
```

Ebben semmi újdonság nincs, természetesen. De a tény, hogy egy teherhajó az egy űrhajó, lehetővé tesz végrehajtani néhány csinos trükköt. Tekints erre a kódsorra például:

```
Urhajo u = new Teherhajo();
```

Ez a kód tökéletesen érvényes. Végül is egy teherhajó az egy űrhajó, tehát lehetne értelme képesnek lenni csinálni egy űrhajó hivatkozást , mutatva egy teherhajóra, helyes?

Van egy hátrány ehhez: az űrhajó hivatkozásnak nem megengedett a hozzáférés a teherhajó osztály bármely jellemző részéhez, minthogy az nem örökölt az űrhajóból. Tételezzük fel, hogy az űrhajóknak van egy újratöltés függvényük, a teherhajóknak pedig egy berakodás függvénye, aztán nézd ezt a kód példát:

```
Urhajo u = new Teherhajo();
u.Ujratoltes(); //rendben
//u.Berakodas(); //FORDÍTÁSI HIBA. Az űrhajók nem tudnak rakományt berakni.
Teherhajo t = new (Teherhajo)u;
t.Berakodas(); //rendben
```

Bármikor, amikor van egy hivatkozásod egy osztályra, csak azon megadott osztály tulajdonságaihoz férhetsz hozzá, még ha az objektum mutat további tulajdonságok támogatására is.

Megjegyzés: jegyezd meg, hogy nem használhatod a polimorfizmust más módon. Ha azt próbáltad volna beírni, hogy Teherhajo t = new Urhajo(); akkor egy fordítási hibát kaptál volna. Egy űrhajó az nem egy teherhajó.

Virtuális függvények

A polimorfizmus egyik legfontosabb vonatkozása a virtuális függvény ötlete. Egy virtuális függvény alapvetően lehetővé teszi egy függvény meghatározását egy alap osztályban és annak későbbi megváltoztatását. Mondjuk alapértelmezésben az összes űrhajó egy módon kezeli a lézerbecsapódást, így meghatározod azt a viselkedést egy alap űrhajó osztályon belül. Aztán később elhatározhatod, hogy a hadihajóknak másképpen kellene a becsapódást kezelniük, mert jobb páncélzatuk van.

A virtuális függvények lehetővé teszik az ilyen helyzetek könnyű kezelését, melyet a következő néhány szakaszban látni fogsz.

Virtuális függvények nélkül

Itt van néhány kód, amik világossá fogják tenni, hogy mi történne a példámban egy virtuális függvény nélkül:

```
class Urhajo
{
	public void LezerTalalat()
	{
		//sok kár
	}
}
class Hadihajo : Urhajo
{
	public void LezerTalalat()
	{
		//kevesebb kár
	}
}
```

Amit itt csináltam, az, hogy létrehoztam egy űrhajó osztályt, amely kevesebb kárt visel el lézertalálat esetén, a hadihajó viszont többet, mert jobban páncélozott, tehát létrehoztam egy új *LezerTalalat* függvényt, amely kisebb kárt okoz.

Ez a kód pontosan azt csinálja, amit gondolsz, hogy csinál:

```
Urhajo u =  new Urhajo();
Hadihajo h = new Hadihajo();
u.LezerTalalat(); //nagyobb kár
h.LezerTalalat(); //kisebb kár
```

Nincsenek trükkök. De mi a helyzet a következő kóddal?

```
Urhajo u =  new Urhajo();
Urhajo h = new Hadihajo();
u.LezerTalalat(); //nagyobb kár
h.LezerTalalat(); //nagyobb kár… miért?
```

Mi a különbség? Az első példában szereplő *Hadihajo* hivatkozás helyett egy *Urhajo* hivatkozást használtam, tehát miért szenvedne le egy hadihajó ugyanannyi kárt, mint egy szabályos űrhajó? Ennek oka az, hogy az űrhajó *LezerTalalat* függvénye soha nem tűnt el – az még ott van. Mikor azt mondtad, hogy *h.LezerTalalat*, az hívta az *Urhajo.LezerTalalat*-ot, mert tudja, hogy a *h* az egy űrhajó. A fordító miért túl buta ahhoz, hogy felfogja, hogy a *h* jelenleg egy hadihajó, elrejtve? Kiderült, hogy ez az a mód, ahogy a fordító feltételezve dolgozott, és végezte úgy a munkáját, hogy igazán érzed a virtuálisság szükségességét.

Üdvözöl a virtuálisság

A virtuális függvények nagyszerű találmány. És még nem is igazán bonyolultak. Mielőtt hozzákezdek a kifejtéséhez, hadd változtassam meg az osztály meghatározást az űrhajónak és a hadihajónak az előző szakaszból, adva néhány kulcsszót a függvénydeklarációhoz:

```
class Urhajo
{
        virtual public void LezerTalalat()
        {
                //sok kár
        }
}
class Hadihajo : Urhajo
{
        override public void LezerTalalat()
        {
                //kevesebb kár
        }
}
```

Két kis dolog megváltozott: a *virtual* szó hozzáadódott az *Urhajo.LezerTalalat* függvényhez, az *override* pedig a *Hadihajo.LezerTalalat* függvényhez. Most, ha futtatod ezt a kódot, pontosan azt csinálja, amit akarsz, hogy csináljon:

```
Urhajo u =  new Urhajo();
Urhajo h = new Hadihajo();
u.LezerTalalat(); //nagyobb kár
h.LezerTalalat(); //most már kevesebb kár. Hurrá!
```

Miért így működik? A virtuális függvény deklarálása közli a fordítóval, hogy a függvényt fel lehet cserélni egy másik változattal később egy gyermek osztályban. Azt mondja, „Hé, ez a lézertalálat függvény működik az összes űrhajónak, de néhány űrhajót később lehet, hogy fel akarunk cserélni”.

Hasonlóképpen, egy függvénydeklarálás, hogy *override* legyen, közli a fordítóval, hogy ez a függvény hatálytalanít egy korábbi változatot. Azt mondja: „Hé, tudom, hogy ez a függvény korábban meg volt adva, de ez a változat jobb, úgyhogy használom helyette”.

Megjegyzés: ha nem használod az *override* kulcsszót, mikor deklarálod a *Hadihajo.LezerTalalat*-ot, akkor bele fogsz futni ugyanabba a hibába, ami azelőtt volt. Az összes hadihajó, mikor űrhajóként van kezelve, az *Urhajo.LezerTalalat*-ot fogja használni az általad használni kívánt függvény helyett. Neked kifejezetten deklarálni kell, hogy egy függvény hatálytalanít egy korábbi verziót. Az olyan nyelveknél, mint a C++ és Java, nem szükséges ez, ezért először ez zavaró lehet kissé.
Megjegyzés: az *override* kulcsszó ellentettje a *new* kulcsszó. Visszaállítani az eredeti viselkedést, írhatnád az *override public void LezerTalalat()* helyett azt, hogy *new public void LezerTalalat()*. Ez meggátolja az *Urhajo.LezerTalalat* függvény felcserélését az új változattal; a régi verzió hívódik bármikor, amikor az *Urhajo* hivatkozással dolgozol, és az új fog hívódni, amikor a *Hadihajo*val.

Absztrakció

Alkalmanként lesz olyan helyzet, melyben nem tudod, hogy mi az alapértelmezett viselkedés egy alap osztálynak. Talán rádöbbensz, hogy azt mondani, hogy „az összes űrhajó találatot kap lézerekből ezen a módon” butaság, mert minden űrhajó különböző, és fel akarod adni a *LezerTalalat* függvény hatálytalanítását minden gyermek osztályban.

Így miért határozol meg egy *Urhajo.LezerTalalat* függvényt az első helyen? Ezesetben az absztrakciónak nevezett tulajdonságot akarod használni. Az *Urhajo.LezerTalalat* függvény elvont, vagyis absztrakt – azt nem tudod, hogy a hajók hogyan szándékoznak lézertalálatot kapni, de azt igen, hogy minden hajó *képes* lézertalálatot kapni.

Ha egyszerűen csak eltávolítod a *LezerTalalat* függvényt az *Urhajo* osztályból, akkor fájdalmas dolgokat fogsz tenni a feneked számára:

```
Urhajo u = new Teherhajo();
u.LezerTalalat(); //HIBA! Az űrhajók nem tuják, hogyan kapjanak lézertalálatot.
```

Szerencsére a C# ad neked egy módot azt mondani, hogy „Az összes űrhajó tudja, hogyan kapjon lézertalálatot, de én még nem vagyok biztos benne, hogy hogyan.” Itt egy újradefiniált *Urhajo* osztály:

```
abstract class Urhajo
{
        abstract public void LezerTalalat();
}
```

Most van egy *Urhajo* osztályod, és tudod, hogy az összes űrhajó képes lézertalálatot kapni, még akkor is, ha ezen a ponton még nincs ötleted, hogyan. De ez hogyan befolyásolja a dolgokat? A következő kód érvényes?

```
Urhajo u = new Urhajo(); //HIBA!
```

Hoppá. Nem tudsz *Urhajo*kat létrehozni többé. Habár ez rendben, mert valószínűleg nem akarod azt csinálni többé – valószínűleg *Hadihajo*kat vagy *Teherhajo*kat akarsz helyette létrehozni:

```
Urhajo u1 = new Teherhajo();
Urhajo u2 = new Hadihajo();
u1.LezerTalalat();
u2.LezerTalalat();
```

Megjegyzés: nem tudod példával szemléltetni az absztrakt osztályokat. Továbbá, ha bármilyen absztrakt függvényeid vannak az osztályban, akkor az osztályt szintén absztraktként kell megadni.
Megjegyzés: bármely függvényt, amely absztraktként van deklarálva, muszáj felülírtként megadni a gyermek osztályokban. Ha ezt nem teszed meg, fordítási hibát fogsz kapni.

Polimorfizmus és függvények

Gondoltam, adnék egy kis megjegyzést polimorfizmus használatáról függvényparaméterekkel, ha a fogalom még nem egészen tiszta neked.

Mondjuk csinálsz egy függvényt, ami működik az űrhajók minden fajtáján. Valami ilyesmi, mint ez:

```
class Valami
{
        static void FolyamatUrhajo( Urhajo u )
        {
                //némi kód
                u.LezerTalalat();
        }
}
```

Átadhatsz teherhajókat vagy hadihajókat a függvénybe és nem fogja bánni mindaddig, amíg átadsz valamilyen űrhajót:

```
Teherhajo teher = new TeherHajo();
Hadihajo harc = new Hadihajo();
valami .FolyamatUrhajo( teher );
valami .FolyamatUrhajo( harc );
```

A *valami.FolyamatUrhajo* függvény nem törődik azzal, hogy milyenfajta űrhajót használsz, mert az összes űrhajónak ugyanolyanok az alapképességeik. Ez a polimorfizmus ereje.

Objektumok

A C#-ban van egy *object* nevezetű osztály, ahonnan minden önműködően örököl. Ez lehetővé teszi, hogy könnyen tárolj objektumokat konténerekben (látni fogod ezt később a fejezetben).

Nézd ezt a kódot például:

```
object o = new TeherHajo();
o = new int();
o = new float();
o = new WeebulKondenzatorAFluxusTomitesben();
```

Az objektumok bármit képesek tartani.

Az objektumosztály használata egy egyszerű módja érték típusok – mint pl. a beépített számok és a saját struktúráid – referencia típusokba való fordításának. Ez azért van, mert az objektumok használhatók doboz érték típusoknak. Bármikor elhelyezel egy értéktípust egy objektumba, az objektum azonnal kioszt memóriát annak az értéktípusnak, és rámutatja magát az új memórián.

Nézd ezt a kódot:

```
int x = 10;
object o = x; //o most egy új utalás a 10-es egész másolatára
x = 20; //x megváltozott; o-nak nem kellene, mivel másolva volt
x = (int)o; //o kicsomagolása; x most 10 újra
```

Megjegyzés: bármikor kibontasz egy objektumot, vissza kell helyezni az eredet típusába (vagy valami összefüggő típusba, amíg kompatibilis). Implicit konverzió nem lehetséges.

Tömbök

A tömbök konténerek, amik lehetővé teszik sok objektum tárolását bennük. Alapvetően, szükséges néhány mód, hogy sok adatot tárolj, és egyszerű változókat használni az összes játékobjektumod tárolására nagyon gyorsan nagyon unalmassá válik:

```
Urhajo u1; //oké
Urhajo u2; //ehhh…
Urhajo u3; //oké, ez egyre bosszantóbb
…
Urhajo u20; //már fájnak az ujjaim
…
Urhajo u42; //drága uram, állítsd meg!
```

Ez egy szörnyű módja a játékod adatainak tárolására. Ne csináld így. Soha. Különben küldeni fogok egy felbőszült mókust a házadhoz, hogy rágja el a kódoló számítógéped tápkábelét, mielőtt a Mentés gombra kattintasz 10 órányi kódolás után egy Mountain Dew ihlette őrületben. Úgy értem.

Mindent a hülye gépelés által csinálás helyett, hozz létre egy tömböt, mely egy nagy darab adat, amit számok által érhetsz el.

Egy alap tömb példa

Itt egy példa tömb használatára:

```
int[] array = new int[10];
array[0] = 0; //az első elem 0
array[1] = 10; //a második elem 10
…
array[9] = 90; //az utolsó elem 90
```

Most van tíz egész számod, és könnyen hozzájuk férhetsz használva a zárójeles jelölést a tömb neve után.

Megjegyzés: a tömbök nulla alapú indexelést használnak, ami azt jelenti, hogy az első objektum egy tömbben a nulla jelölést kapja egy helyett, mint sokan ezt gyanítják. Ez azt jelenti, hogy az előző példa tömbjében 0-tól 9-ig vannak érvényes indexek, és a 10 már érvénytelen.

Mi egy tömb?

Egy tömb, ahogy előzőleg mondtam, csak adat egy nagy darabja. Mikor a következő kódot írod, akkor létrehozol egy *a* nevezetű változót, mely egy utalás egész számok egy tömbjére:

```
int[] a;
```

Az összes tömb referenciatípus, ami azt jelenti, hogy használnod kell a *new* kulcsszót a tömb tulajdonképpeni létrehozásához:

```
a = new int[8];
```

Ez a kódsor létrehozta nyolc egész szám egy tömbjét.

Minden szándék és cél érdekében, kezelheted *a*-t mint bármely más referenciatípust. Itt van néhány kód, amely bemutatja, hogyan használhatod a tömböket:

```
int[] a = new int[10];
int[] b = a; //b ugyanarra a tömbre mutat most
b[0] = 10; //b megváltoztatása a-t is változtatja
int i = a[0]; //i most 10
b = null; //b nem mutat többé semmire
a = new int[20]; //a régi tömb elveszett, szemét gyűjtve lesz később
object c = a; //még egy „objektum” típusnak is csinálhatod
```

A tömböket elég könnyű használni, amint látod.

Sorban inicializálás

Megadhatod az értékeit egy tömbnek közvetlenül, mikor létrehozod, ezen kód használatával:

```
int[] array = new int[] { 1, 2, 3, 4, 5 };
```

Ez létrehoz egy új integer tömböt öt indexszel, amelyek az 1, 2, 3, 4 és 5 értékeket tartalmazzák.

Referenciák az értékek ellen

Az előző szakaszban mutattam neked egész számok egy tömbjét, amelyek érték típusok. Ezt meglehetősen könnyű megérteni. De mi történik, mikor referenciatípusok egy tömbjét hozod létre, amilyen pl. egy űrhajó?

```
Urhajo[] u = new Urhajo[5];
```

Ez öt darab űrhajó egy tömbjét hozza létre? Nem. Ez valójában öt űrhajó *hivatkozás* egy tömbjét hozza létre. Nézd a következő kódot:

```
Urhajo u = new Urhajo[5];
u[0] = new Urhajo();
u[2] = new Urhajo();
```

A 0 és 2 indexek vesznek fel űrhajókat, míg a maradék az ötből nulla mutatók. Láthatod, hogy referenciatípusok egy tömbje csak hivatkozásokat tartalmaz, és nem az aktuális típusokat. Kézzel kell létrehoznod minden egyes objektumot egy tömbben magadnak, ha szükséges használnod azokat. Bár ahogy a második fejezetben megtanultad, használhatod a for-ciklusokat ezen folyamat nagyon egyszerűvé tételéhez.

Öröklés és tömbök

Egyike a legjobb dolgoknak a tömbökkel kapcsolatban, hogy teljes mértékben támogatják az öröklést. Nézd ezt a kódot például:

```
Urhajo[] u = new Urhajo[4];
u[0] = new HadiHajo();
u[1] = new TeherHajo();
u[2] = new TeherHajo();
u[3] = new HadiHajo();
for (int i=0; i < 4; i++ )
        u[i].LezerTalalat(); //minden hajó találatot kap
```

Ez a kód létrehozza négy űrhajó egy tömbjét, és aztán feltölti két teherhajóval és két hadihajóval. Az utolsó két sor mutat neked egy *for* ciklust, amely végigmegy a tömbön, és eltalál minden hajót egy lézersugárral. Ez gyönyörűen működik, mert a fordító tudja, hogy az összes űrhajó tudja, hogyan kap találatot lézerrel, és nem törődik azzal, hogy az űrhajó vajon hadihajó vagy teherhajó.

Többdimenziós tömbök

Eddig amit mutattam neked, azok egydimenziós tömbök. Ha ismered a geometriádat, akkor tudod, hogy valaminek egy dimenzióban csak a hossza lehet meghatározva; nincs szélesség vagy magasság. Egy dimenzióban eléggé korlátozva vagy egyenes vonalak rajzolására. Ez ugyanaz a tömböknél is: egy egydimenziós tömböt egy egyenes vonalként lehetne látni.

Ha fogod a geometrikus elgondolást és kiterjeszted két és három dimenziókra, akkor elképzelheted a tömböket a következő kinézetekben:

Egy 2D-s tömbre úgy lehet gondolni, mint egy négyzetrácsra, mint egy sakktábla. Egy 3D-s tömbre gondolhatsz úgy, mint egy terjedelmes rácsra, sokkal inkább mint egy Rubik-kockára.

Más dimenziókban is lehetnek tömbjeid, mint például 4D, 5D, de akár 32D-ig is, de a legtöbb embernek az olyan tömbök elképzelése bonyolult lehet, és igazán nem látnak sokat.

A könnyű mód

A C# különbözik a C/C++/Java-tól a többdimenziós tömbök kezelésében. Ha már ismerős vagy azokkal a nyelvekkel, akkor ez kissé eldobhat téged, de igazán nem bonyolult.

Alapvetően, egy 2D-s és 3D-s egészek tömbjének megadásához írhatnád ezt:

```
int[,] tomb2d;
int[,,] tomb3d;
```

És egy 32D-s tömb:

```
int[,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,] tomb32d;
```

Egyszerűen el kell helyezned n-1 darab vesszőt a zárójeleken belül, hogy megadj egy *n*-dimenziós tömböt.

A következő lépés egy tömb tulajdonképpeni létrehozása:

```
tomb2d = new int[5,5]; //5x5-ös tömb
tomb3d = new int[5,5,3]; //5x5x3-as tömb
```

Megváltoztathatod a dimenziókat mindenre, ami szükséges, hogy megfeleljen a szándékaidnak. Az elemekhez való hozzáférés is könnyű a tömbökben:

```
tomb2d[0,0] = 100;
tomb2d[2,2] = 200;
tomb2d[0,4] = 300;
tomb3d[0,0,0] = 400;
tomb3d[2,2,1] = 500;
tomb3d[4,0,2] = 600;
```

A nehéz mód

Van egy másik módja a tömbök létrehozásának, de az nem olyan könnyű, mint az első módszer, amit az előbb mutattam. Ez az a megközelítés, amit az olyan nyelvek, mint a Java, visznek véghez, és néha meglehetősen fájdalmas a fenéknek. Alapvetően az elgondolás az, hogy egy 2D tömb csak egy 1D tömbök tömbje. Különösen hangzik, ugye? De van értelme.

Egy tömb tartalmazhat bármit, tehát miért ne tarthatna más tömböket? Itt van, hogyan deklarálhatnál egy 2D és egy 3D tömböt ezen a módon:

```
int[][] tomb2d;
int[][][] tomb3d;
```

Bár a tömb kiosztása egy trükkös feladat. Nem tudsz csak úgy ilyesmit mondani, mint ez:

```
int[][] tomb2d = new int[5][5]; //HIBA
int[][][] tomb3d = new int[5][5][3]; //HIBA
```

Az első példában megpróbáltál kiosztani hat különböző tömböt egy időben (egy tömbök tömbje és öt egészek tömbje), és a C# nem teszi ezt lehetővé neked. Helyette szükség van még egy kis munkára, először a tömbök tömbjének kiosztásával:

```
int[][] tomb2d = new int[5][];
```

aztán létrehozni minden egész tömböt egyénileg:

```
for (int i = 0; i < 5; i++)
{
        tomb2d[i] = new int[5];
}
```

Amint elkészültél azzal, elkezdheted feltölteni a tömböket:

```
tomb2d[0][0] = 100;
tomb2d[2][2] = 200;
tomb2d[0][4] = 300;
```

Természetesen egy 3D tömbnél ez még rendetlenebb, mert van egy tömbök tömbjeinek tömbje. Így természetesen sok munkába kerül ezt csinálni:

```
int[][][] tomb3d = new int[5][][];
//a tömbök második dimenziójának létrehozása
for( int i = 0; i < 5; i++)
{
        tomb3d[i] = new int[5][][];
```

```
}
//most létrehozni a tömbök 3. dimenzióját
for( int i = 0; i < 5; i++)
{
        for( int j = 0; j < 5; j++)
        {
tomb3d[i][j] = new int[3];
        }
}
```

Sűrítheted a ciklust, ha akarod, de én sűrítetlenül hagytam, hogy tisztán mutassa, mi folyik itt.

Az első dolog, amit csináltam az előző példában, hogy létrehoztam egy tömböt. Aztán a következő ciklus végigment és feltöltötte mind az öt indexet új tömbökkel. Ezen a ponton van egy tömböm öt indexszel, és minden indexnek van egy öt indexes tömbje. Az utolsó ciklus végigmegy mind a 25 indexen a 2D tömbben, és kitölti mindet három egész szám egy tömbjével.

Egy hatalmas rendetlenség, ugye? Ez az, amiért az első módszer, amit mutattam, előnyösebb. Az egyedüli haszon, amit élvezhetsz ezzel a módszerrel, a tény, hogy nem kell csinálnod négyszögletes tömböket.

Mivel tömbök tömbjeit tárolod, az utolsó dimenzióban lévő tömböknek nem kell azonos méretűeknek lenniük. Ez néhány helyzetben meglehetősen hasznos lehet, de ilyenek nem jönnek elő túl gyakran.

Ciklus másik fajtája

Visszatérve a 2. fejezetre, megmutattam neked, hogyan lehet végrehajtani különböző ciklusokat C#-ban, használva a *for*, *while* és *do while* ismétlési szerkezeteket. Tulajdonképpen van még egy ciklus a C#-ban, amiről még nem beszéltem: a *foreach* ciklus.

A *foreach* ciklus tulajdonképpen csodálatosan könnyen használható, de csak adat gyűjteményeken, mint például tömbök.

Megjegyzés: a foreach ciklus használható más gyűjteményeken, amiket még nem mutattam. Ezekhez az 5. fejezetben fogok eljutni.

Itt a kifejezés alap nyelvtana:

foreach(**típus változó** in **gyűjtemény**)
{

```
        //ismétlési kód itt
}
```

A kifejezés kezelni fog minden objektumot a *gyűjtemény*en belül, mint bármi típust megadtál a *típus* résznek, és hozzáférhetsz a változóhoz a *változó* részt használva. Például:

```
int[] tomb = new int[] { 1, 2, 3, 4, 5 };
int osszeg = 0;
foreach( int i in tomb )
{
        osszeg = osszeg + i;
}
```

Ez a kód végigmegy az összes indexen a *tomb* tömbben, és összegzi őket.

Az egyetlen korlátozás a *foreach* cikluson, hogy nem megengedett a fizikai megváltoztatása a gyűjtemény tartalmának. Ha megpróbálsz létrehozni ilyen kifejezést az előzőnek, mint ez, akkor fordítási hibát kaphatsz:

```
foreach( int i in tomb )
{
        i = 0;
}
```

Ez azért van, mert a fordító a változót csak-olvashatóként kezeli, így nem tudod megváltoztatni. Sajnos ez azt jelenti, hogy csak érték típus tömbök értékeit tudod olvasni, és nem tudod megváltoztatni a hivatkozásokat a referencia-típus tömbökben. Bár a jó hír az, hogy meg tudod változtatni magát az aktuális referenciatípust, tehát ha van neked egy osztályok tömbje, előremehetsz és megváltoztathatod az osztályokat amennyire akarod – csak a tömbben az indexeket nem tudod mutatni egy másik osztályra.

Szövegek

Még manapság is, a nagyszerű hangszintézis és fordítás korában a kommunikáció nagy része szövegen keresztül megy. Ez az, amiért majdnem minden programnyelvnek van egy nagyon átfogó sztring, azaz szöveg könyvtára. Nem meglepő módon a C#-nak is.

Szerencsére a szövegeket nagyon egyszerű használni. Ha már játszogattál a C *char* adattípusával, akkor megszereted a C# sztringjeit is, amik a szövegek használatát mókássá teszik!

Hadd ugorjak bele és mutassak néhány példát:

```
string sz = „Helló!”; //”Helló!”
sz = sz + „ Hogy vagy?”; //”Helló! Hogy vagy?”
if ( sz == ”Helló! Hogy vagy?” )
{
        sz = „Szia!”; //”Szia!”
}
if ( sz !=”Szia!” )
{
        //A kiértékelés hamis eredményű, így ez a kód nem számít
}
```

A szövegek egyéni tulajdonsága, hogy csak olvashatók. Nem tudod őket megváltoztatni, nem számít milyen keményen próbálkozol. Ha meg akarsz változtatni egy szöveget, akkor létre kell hoznod egy újat és felülírnod azt (mint az előző példa második sorában). Ez időnként pazarló lehet, de mikor csináltál a legutóbbi időkben a processzort leterhelő szövegfeldolgozást egy játékban? Én is úgy gondolom.

A *string* osztály, a már mutatott alap dolgokon kívül, támogatja a hasznos függvények összes fajtáját:

```
string sz = „Hello”;
sz a;
a = sz.ToUpper(); //Visszaadja: „HELLO”
a = sz.ToLower(); //Visszaadja: „hello”
a = sz.Remove( 0, 2 ); //Visszaadja: „llo”
a = sz.Substring( 1, 3 ); //Visszaadja: „ell”
```

És így tovább. Tonnányi függvény van; a leghasznosabbak ki vannak listázva a lenti táblázatban. Jegyezd meg, hogy az *sz* változó nem módosult az előző példában; minden függvény visszatért egy új sztringgel az *sz* megváltoztatása helyett.

Megjegyzés: ha valami nagyfokú szövegmanipulálás elvégzésére van szükséged,akkor a *string* osztály használata helyett bele kellene nézned a *System.Text.StringBuilder* használatába. Sajnos ez túlmutat a könyv hatáskörén. Én csak azt akartam, hogy a *StringBuilder*-ekről tudj, melyek messze hatékonyabbak a nagyfokú szövegmanipuláláshoz.

Egy szöveg alapvetően írásjelek egy tömbje, ezért vártam, hogy tudj a tömbökről, mielőtt kifejtettem. Használhatsz egy szöveget majdnem pontosan úgy, mint egy tömböt, mikor betekintesz egyéni írásjelekbe:

```
string sz = „helló!”;
char betu = sz[0]; //’h’
betu = sz[3]; //’l’
```

Természetesen, minthogy a szövegek csak olvashatóak, nem tudod megváltoztatni az írásjeleket – egy új szöveget kell létrehoznod. Fájdalmas lehet a fenéknek, de ez így van.

Hasznos sztring függvények:

Függvény	**Leírás**
bool Endswith(**sz**)	Megállapítja, hogy egy szöveg véget ér-e **sz**-szel
string Insert(**index**, **sz**)	Beszúrja **sz** szöveget **index** kezdetű helyre
string PadLeft(**szel**, **toltbetu**)	Növeli a szöveg szélességét **szel**-re, beszúrva **toltbetu**-ket balra.
string PadRight(**szel**, **toltbetu**)	Hasonló,mint a PadLeft, de jobbra.
string Remove(**index**, **ennyi**)	Eltávolít **ennyi** írásjelet **index** helytől kezdve.
string[] Split()	Visszaadja a szövegben lévő összes szavak tömbjét.
bool Startswith(**sz**)	Megállapítja, hogy egy szöveg **sz**-szel kezdődik-e.
string Substring(**index**, **ennyi**)	Vissszatér egy szöveggel **index** helytől, ami **ennyi** írásjel hosszú.
string ToUpper()	A kisbetűket nagybetűkké alakítja.
string ToLower()	A nagybetűket kisbetűké alakítja.
string Trim()	Levágja a szóközöket elölről és hátulról.
string TrimEnd()	Levágja a szóközöket hátulról.
string TrimStart()	Levágja a szóközöket elölről.

5. fejezet

Még egy C# fejezet

Biztos vagyok benne, hogy ez az összes anyag a C#-ról kezd öreggé válni; már öt fejezetben voltunk, és még van némi időszerű és játékkal-összefüggő programozásunk. Argh!

Nos, csak megmutatja neked, hogy milyen összetett a C# - őszintén, a C# egyike a valaha ír legösszetettebb számítógép nyelveknek és mikor a .NET keretrendszerrel kombinálták, szintén egyike a létező legnagyobb számítógép nyelveknek.

Már csak néhány témám van, mielőtt továbblépek a Windows programozásra és megtervezem az alap keretrendszerét a számítógépes játékodnak.

Határfelületek

Egy további téma, ami összefüggésben van az örökléssel és eddig még nem érintettem, az a határfelületek témája. Használtam az *interface* kifejezést ezelőtt, de a C#-nak tulajdonképpen van egy kulcsszava ezzel a névvel, mely meghatároz egy szerkezetet némileg eltérően attól, amit eddig láttál.

A 4. fejezetben megmutattam, hogy mi egy absztrakt osztály, és hogy az absztrakt függvények hogyan határoznak meg egy határfelületet. Számítógép tudományi kifejezésben az megfelelő, de a C#-nak tulajdonképpen van egy *interface* kulcsszava valódi határfelületek meghatározására.

Nézz erre az absztrakt osztályra például:

```
abstract class Urhajo
{
	abstract public void LezerTalalat();
};
```

Bármi, ami örököl ebből az osztályból, muszáj hogy meghatározzon egy *LezerTalalat* függvényt, mivel az összes űrhajónak van az a függvény.

Másrészről újradefiniálhatod az űrhajót, és csinálhatsz egy határfelületet egy osztály helyett:

```
interface IUrhajo
{
	void LezerTalalat();
};
```

Tanács: bevett szokás C#-ban, hogy a határfelületek nevét egy nagy I-vel kezdjük. Ez segít a kódot olvashatóbbá tenni, de nincs megkövetelve, hogy így csináld.

Lényegében az *Urhajo* és az *IUrhajo* azonos célt szolgál: definiálnak egy határfelületet, amit a gyermek osztályoknak később végre kell hajtaniuk. Mielőtt a határfelületek részleteibe mennék, hadd mutassam meg, hogyan zajlik az öröklés ebből a két különböző konstruktorból.

Egy egyszerű referenciapontnak, itt egy hadihajó definíció, amely örökölni fog az absztrakt *Urhajo* osztályból:

```
class HadiHajo : Urhajo
{
	override public void LezerTalalat()
	{
		//valami kód
	}
};
```

Semmi új nincs abban a kódban; csak azért raktam ide, hogy megmutassam a különbséget az öröklés egy absztrakt osztály és egy határfelület között.

Itt van az öröklés egy határfelületből:

```
class TeherHajo : IUrhajo
{
	public void LezerTalalat()
	{
		//valami kód
	}
};
```

Ennyi. Az egyetlen különbség, hogy mikor örökölsz egy határfelületből, akkor nem teheted a *LezerTalalat* függvényt egy felülírássá, mert semmi felülírható nincs – a határfelület függvények nem virtuálisak, hacsak nem tetted azzá egy gyermek osztályban, mint a *TeherHajo*.

Határfelületek az Absztrakt Osztályok ellenében

Mi a különbség egy határfelület és egy absztrakt osztály között? Tulajdonképpen számos különbség van, amelyeket nem jegyeznél meg könnyedén. A következő néhány alfejezet áttekinti ezen különbségek némelyikét.

Függvény definíciók és adat

A fő különbség a határfelületek és az absztrakt osztályok között, hogy míg az absztrakt osztályok tudnak úgy tevékenykedni, mint a határfelületek, attól ők még lényegében osztályok, és tudnak tartani dolgokat, mint adat és függvény definíciók. A határfelületek nem tudnak tartani adatot vagy függvény definíciókat, hanem csak függvény *deklarációkat* (a visszaadott típus, név és egy függvény paraméterei).

Virtuális függvények

Az absztrakt függvények feltételezhetően virtuálisak. Bármely osztály, amely örököl egy absztrakt osztályból, szabadon felülírhat bármilyen absztrakt függvényt a saját implementációjával.

A határfüggvények alapértelmezésben nem virtuálisak. Bármilyen függvény, amelyet meghatározol egy osztályon belül, amit használ egy határfelület, az alapértelmezésben egy szabályos függvény; kifejezetten virtuálissá kell tenned a függvényt azért, hogy megadd neki a képességet a függvény későbbi felülírásához. Nézd a következő kódot:

```
class HadiHajo : IUrhajo
{
	public void LezerTalalat()
	{
		//valami kód
	}
}

class FejlesztettHadiHajo : HadiHajo
{
	new public void LezerTalalat()
	{
		//valami kód
	}
}
```

Ez a kód két osztályt hoz létre, használva az *IUrhajo* határfelületet, végrehajtva a *LezerTalalat* függvényt. Ez a kód nem használ virtuális függvényeket; *HadiHajo.LezerTalalat* csak egy szabályos függvény.

Itt van néhány példakód, amelyek megmutatják, hogy az *IUrhajo* határfelület hogyan működik:

```
IUrhajo u = new HadiHajo();
u.LezerTalalat(); //hívja a HadiHajo.LezerTalalat-ot
u = new FejlesztettHadiHajo();
u.LezerTalalat(); //még hívja a HadiHajo.LezerTalalat-ot
```

A számítógép nem látja a *FejlesztettHadiHajo.LezerTalalat*-ot, mikor használod az *IUrhajo* határfelületet. Hogy azt tegye, virtuálissá kell tenned:

```
class HadiHajo : IUrhajo
{
	virtual public void LezerTalalat()
	{
		//valami kód
	}
}

class FejlesztettHadiHajo : HadiHajo
{
	override public void LezerTalalat()
	{
		//valami kód
	}
}
```

Most a következő kód úgy fog működni, ahogy elvárod tőle, mikor használod az új *HadiHajo* és *FejlesztettHadiHajo* definíciókkal:

```
IUrhajo u = new HadiHajo();
u.LezerTalalat(); //hívja a HadiHajo.LezerTalalat-ot
u = new FejlesztettHadiHajo();
u.LezerTalalat(); // hívja a FejlesztettHadiHajo.LezerTalalat-ot
```

Néha az alapértelmezett viselkedése a határfelületekben lévő nem-virtuális függvényeknek előnyödre működhet, máskor pedig lehet, hogy nem. Csak tartsd észben, hogy határfelületekkel további rugalmasságod van a függvényeiddel, de absztrakt osztályokkal rá vagy kényszerítve, hogy virtuális függvényeket használj.

Hozzáférés

El tudsz rejteni függvényeket és adatot absztrakt osztályok belsejében. Absztrakt függvényeknek nem kell publikusnak lenniük; védetté teheted őket, ha akarod.

Másrészről határfelületeken belüli függvények mindig publikusak. Az ötlet az, hogy a határfelületeket meghatározó függvényeket akarod mindenki által láthatónak.

Többszörös öröklés

A többszörös öröklés egyike azon ötleteknek, amelyek elsőre érzékenynek tűnnek, de ha egyszer eljutsz a használatához, végül is a fenét bosszantja, és ezért a C# nem támogatja közvetlenül. Bár a C# támogatja egy korlátozott formáját a többszörös öröklésnek, így be szándékozom mutatni a mögötte levő elméletet. Alapjában véve az ötlet az, hogy létrehozhatsz egy osztályt, amely örököl két különböző osztályból.

Például lehet, hogy létre akarsz hozni egy zászlóshajót az űrhajóflottádnak; a zászlóshajó egy olyan hajó lesz, amely ötvözi egy hadihajó fegyverzetét és egy teherhajó teherhordó képességét. Elméletben nagyszerűen hangzik, ugye?

Hadd mondjam el neked most: a C# nem támogatja a többszörös öröklést. Az csak egy hatalmas zűrzavar.

Gondolj erre: az összes űrhajó kaphat lézertalálatot. De a hadihajók másféleképpen, mint a teherhajók. Tehát van két különbözőféle hajód, amelyek találatot kaphatnak, de mindkettő teljesen másféleképpen reagálhat rá.

Most csinálsz egy zászlóshajót, mely szintén kaphat lézertalálatot. De hogyan csinálja azt? Úgy kapja, mint egy hadihajó vagy mint egy teherhajó? Ki tudja? Igazán nincs könnyű módja konkrétan meghatározni, hogy minek kellene történnie. A probléma még rosszabbá válik, mikor adatot vonunk be… de elég is ebből – a többszörös öröklés csak egy fájdalom a nyaknak.

A C#-nak vannak határfelületei, hogy megoldják a többszörös öröklés problémáját. C#-ban egy osztály csak egy alapból tud örökölni. Ha a következőt csinálod, akkor hibát fogsz kapni:

```
class ZaszlosHajo : HadiHajo, Teherhajo
```

De mi van, ha megadsz egy határfelületet a teherhajónak helyette?

```
interface ITeher()
{
        void Berakodas();
        void Kirakodas();
};
```

Most próbáljuk meg az öröklést:

```
class ZaszlosHajo : HadiHajo, ITeher
```

Működik. Most egy zászlóshajó lényegében egy teherhajó, amely tud rakományt be- és kirakodni. Sajnos majd újra kell kódolnod a teher függvényeket a saját módodon; ez csak egyike a korlátozásoknak, amelyeket kezelned kell.

Megjegyzés: egy jobb megoldása lehetne a kombinált képességek problémájának, hogy az összes űrhajót képessé tesszük teher kezelésére az első helyen. Például tervezhetnéd a hadihajókat képessé nagyon kis mennyiségű rakomány hordozására, mint pl. a pilóta személyes tárgyai, míg egy teherhajó cuccok megatonnáit képes hordani. Ez rajtad múlik.

Általánosságban szólva, igazán nem sokszor lesz szükséged többszörös öröklésre; más megoldások is léteznek, amelyek valószínűleg jobban működnek.

Határfelületek kiterjesztése és kombinálása

Ez a fogalom annyira egyszerű, hogy valószínűleg már kitaláltad: a határfelületeket lehet kiterjeszteni és kombinálni. Mondjuk van egy harci interfészed, ami tud lézereket és rakétákat lőni, de később egy különleges harci interfészt akarsz csinálni, ami tud lőni atombombákat is.

Csinálhattad így:

```
interface IHarc
{
	void LezerLoves();
	void RaketaLoves();
}
interface IAtomHarc : IHarc
{
	void Atom();
}
```

Most van egy *IAtomHarc* határfelületed, amelynek három függvénye van: egy a lézerek lövésére, egy a rakéták lövésére és egy az orbitális pályáról egy bolygó felrobbantására.

Természetesen kombinálhatod is a határfelületeket:

```
interface IZaszlosHajo : IHarc, ITeher
{
	//helyezd ide a szükséges kódodat
}
```

Most van egy zászlóshajó interfészed, aminek vannak függvényei a harci és teher műveletek végrehajtására. Semmi különleges nincs itt.

Kivételek

Emlékszem a számítógép programozás régi rossz napjaira, mielőtt a kivételek jöttek. Annyira boldog vagyok, hogy azok a napok elmúltak. Emlékszem a kódírásra, amely így nézett ki:

```
hiba = Start Graphics Engine();
if( hiba )
	hibaüzenet mutatása
```

```
hiba = Set Resolution();
if( hiba )
        hibaüzenet mutatása

hiba = Start Sound Engine();
if( hiba )
        hibaüzenet mutatása
```

És így tovább. Talán még írsz kódot, mint ez. Ha igen, meg kellene állnod; ez egy fájdalom a fenéknek. A kivételek a megoldás az olyan csúnya kódra, mint ez. A kivételek különleges események, amelyek kivételes körülményekben történnek. Az ötlet az, hogy meg kellene írnod a kódodat, feltételezve, hogy működni fog, és van hibakezelő rész valahol máshol.

Hadd fejtsem ki. Mikor futtatsz egy programot, sok rossz dolog történhet. Kifogyhatsz a memóriából például, vagy egy hardveres alkatrész meghibásodhat. Vagy talán egy szükséges állomány megmagyarázhatatlanul eltűnhet. Legjobb, ha felkészülsz arra, mikor a dolgok rosszra fordulnak.

Sajnos a programozók lusták. Valld be: lusta vagy és gyűlölsz tonnányi munkát végezni. A hibaellenőrzés bosszantó és nem akarsz ezzel foglalkozni.

A legnagyobb rúgás a fenéknek annak felismerése, hogy minden szükséges dolog 90 százalékát a védekezés teszi ki azok ellen, amelyek majdnem soha nem történnek meg. A lemezterületből való kifogyás egyszer történik meg egy kék hold esetén. Néhányan úgy gondolják: „Nos, minthogy ez alig esik meg, nem szándékozom védekezni ellene.” De ez a fajta gondolat visszajöhet kísérteni téged, mert előbb vagy utóbb meg fog történni. Semmi nem szívat jobban, minthogy van egy program – különösen egy játék – amely összeomlik, miután hosszú idő alatt adatokat halmoz fel. Védened kell a játékaidat, ez minden.

A kivételek alapjai

A kivételek összetett bestiák, úgyhogy először adok egy egyszerű áttekintést róluk, aztán pedig az előrehaladottabb részeket mutatom be.

Kivételek egy példája

Tekints erre a példára:

```
try
{
	StartGraphicsEngine();
	SetResolution();
	StartSoundEngine();
}
catch
{
	//egy hibaüzenet kiírása
}
```

Az első dolog amit észrevehetsz, hogy a kód sokkal tisztább, mint a példa, amit előzőleg mutattam. Nincsen bosszantó hibaellenőrző kódod, keverve mindennel. A fő kód tiszta, és gyorsan átlátod, hogy mit csinál. Elindítja a grafikus motort, beállítja a felbontást, és elindítja a hang motort. Badda-bing, badda-bumm.

Nyilván, ha probléma adódik és azon feladatok egyike nem végrehajtható, akkor a játék nem tud futni. Ez egy kivételes körülmény.

Ami történik, mikor egy kivételes körülmény bekövetkezik, hogy azon függvények mindegyike dobni fog egy kivételt, amikor valami rossz történik (később megmutatom, hogyan dobnak kivételt). Mikor egy kivétel dobódott, a függvény azonnal kilép, és a végrehajtás kiugrást tart minden függvényből, míg egy fogási blokkot nem talál. Tehát a kódban, amit éppen mutattam neked, ha valami rosszra fordul és nem tudod indítani a grafikus motort a *StartGraphicsEngine*-en belül, akkor a függvény dob egy kivételt, és a végrehajtás azonnal ki fog ugrani a függvényből, kihagyja végig a *SetResolution*-t és a *StartSoundEngine*-t azok végrehajtása nélkül, és folytatja a végrehajtást a fogási, vagyis a *catch* blokkon belül.

A try blokk

Az előzőleg mutatott kódpélda első része a *try* blokk. Egy *try* blokk egy kóddarab, amely elmondja a fordítónak: „Végre akarom hajtani ezt a kódot, de tudom, hogy valami rossz történhet azon belül, ezért ügyelj egy kivételre.” Minden *try* blokkot követnie kell egy *catch* blokknak, melyhez a következő szakaszban fogok eljutni.

Mi történik, ha végrehajtasz kódot, amely esetleg dob egy kivételt, mikor nincs egy *catch* blokkon belül? Nézd ezt a kódot például (mint általában, tételezzük fel, hogy egy osztályon belül van):

```
public void Initialize()
{
        StartGraphicsEngine();
        SetResolution();
        StartSoundEngine();
}
```

Most, mondjuk valami hívja az *Initialize*-ot. Az hívja a *StartGraphicsEngine*-t, amely próbál megtalálni egy kijelzőeszközt (ezen kód mindegyike csak színlelt, csupán egy például szolgál). Tételezzük fel, hogy nem talál egy kijelző eszközt. Ez egy csinos kivételezési hiba, tehát a függvény azt mondja: „Feladom, nem tudom elindítani a grafikus motort” és dob egy kivételt. Inkább minthogy folytassa, látja, hogy nincs *try*/*catch* blokk a kivétel kezelésére, és kiugrik újra akárkire, aki hívta az *Initialize*-ot. A kivételek tartani fogják a kiugrást, amíg találni fognak egy *try*/*catch* blokkot, ami kezeli a kivételt. Ha nem találnak egyet sem, akkor végül is az egész program ki fog lépni.

```
public class Class1
{
        static void Main( string[] args )
        {
                int[] array = new int[5];
                array[100] = 20;
                System.Console.WriteLine( „Ezt soha nem kellene végrehajtani” );
        }
}
```

A példa létrehoz egy új tömböt öt indexszel (0-tól 4-ig), aztán megpróbálja hozzárendelni a 20-as értéket a 100-as indexhez.

A C# *array* osztály okos; felismeri, hogy valami olyat csináltál, ami helytelen – a tömb nem olyan nagy! Valami igazán kivételesnek kellett történnie, így az *array* osztály dobott egy kivételt. A kód utolsó sora kihagyódik, és a program kilép. Nem kaptad el a kivételt.

A catch blokk

Hogy elkapd a kivételeket és megelőzd a programodnak az ellenőrzés alóli kifutását, használj *catch* blokkokat.

```
public class Class1
{
	static void Main(string[] args)
	{
		int[] array = new int[5];
		try
		{
			array[100] = 20;
			System.Console.WriteLine( “Ezt soha nem kellene végrehajtani” );
		}
		catch
		{
			System.Console.WriteLine( “A kivétel elkapva!” );
		}
		System.Console.WriteLine( “A végrehajtás normálisan folytatódik...” );
	}
}
```

Most a kivétel megfelelően ugrik a hozzárendelési sorra a *catch* blokkra. Amint a *catch* blokk végrehajtása befejeződik, a futtatás normálisan folytatódik. A rendszer kezelte a kivételt, és tudja, hogy a probléma nem létezik többé. Legalábbis remélhetőleg. Igazán tőled függ, hogy javítsd a problémát a *catch* blokkon belül.

A Finally blokk

Lesz idő, amikor nem akarod a kód ugrást az egész hely körül, mikor egy kivétel dobódott. Például, mondjuk van egy állományod, amit megnyitottál, és valami rossz történt a rajta való munka folyamata alatt, és ezért dobódott egy kivétel. Ahelyett, hogy a végrehajtás azonnal kiugrana, talán szeretnél arról megbizonyosodni, hogy az állomány először megfelelően bezáródott. Nos, akkor talán ezt csinálnád:

```
try
{
	//fájlmegnyitás itt
	…
	//valami kivétel dobódhat itt
	…
	//többi kód
```

```
}
catch
{
        //hibakezelés itt
}
finally
{
        //fájlbezárás itt. Ez a kód mindig végre fog hajtódni.
}
```

Ha a fájl-lezáró kódodat a *try* blokkon belülre helyezed, akkor van esély arra, hogy nem hajtódik végre. Ha egy *catch* blokkban helyezed azt el, akkor pedig csak egy kivétel dobódása esetén hajtódik végre. Természetesen elhelyezhetted volna ugyanazt a kódot a *try* és a *catch* blokkon belül is, de ez kódkettőzés, és az rossz. Helyette helyezd el a *finally* blokkban, ezáltal garantálva, hogy a kód mindig végre fog hajtódni a többi blokk után, nem számít, hogy mik történnek.

Megjegyzés: megjegyzendő, hogy mindhárom blokk hatóköre különböző. Bármit definiálsz egy *try* blokkon belül, az nem hozzáférhető azon blokkon kívül, tehát az idő múlásával eljutva a *catch* vagy *finally* blokkhoz, minden ami a *try* blokkon belül deklarált, az kifut a hatókörből, és nem lesz hozzáférhető. Ezért az előző példában deklarálnod (de nem megnyitni) kellene a fájl objektumot a *try* blokkon kívül.

Haladó kivétel témák

A kivételek tulajdonképpen meglehetősen haladó dolgok, és még van sok tudnivaló velük kapcsolatban. Egy kivétel tulajdonképpen egy objektum. Mindig mikor egy kivétel dobódik, a rendszer létrehoz egy kivétel objektumot, ami információt tartalmaz arról, hogy mi ment rosszul. Az összes kivétel egy közös alap osztályból öröklődik, és ez a: *System.Exception*.

Például a tömbös példáknál a kivétel ami dobódott, az a *System.IndexOutOfRangeException* típusa volt. Az a kivételtípus elmondja neked, hogy próbáltál használni egy olyan indexet, ami egy gyűjtőobjektum értéktartományán kívülre esett.

Különleges kivételek megragadása

Néha egynél több hiba fordul elő a kódodban. Egy egyszerű *catch* blokk használatával az összes kivétel elkapódik és feltételezhetően kezeled őket; a végrehajtás aztán folytatódik. De ez nem mindig egy jó ötlet, és hadd fejtsem ki,

hogy miért. Rendszerint tudni szeretnéd, hogy milyen hibák elvártak bizonyos kódokban. Mikor állományokat kezelsz, akkor tudod, hogy szembe kell nézned azzal a lehetőséggel, hogy kifutsz a helyből. De megvan a lehetősége annak is, hogy valami teljesen váratlan történik. Egy *catch* blokk olyan, mint egy szerződés. Azt mondja, „Szándékozom kezelni ezt a kivételt, és aztán a program szándékozik tartani a futtatást ezután." De ha valami teljesen váratlan történik, akkor hogyan tudsz megbizonyosodni a probléma megoldásáról? Nem tudsz.

Ezért a C# lehetővé teszi kivételek különleges típusainak megragadását. Nézd meg:

```
try
{
        //némi kód, amely dobhat index hibát
}
catch( System.IndexOutOfRangeException e )
{
        //hibaüzenet kiírása:
        System.Console.WriteLine( e.Message );
}
```

Most csak azt csináltad, hogy a kódod képes legyen index kivételek elkapására. Az összes többi kivétel figyelmen kívül lesz hagyva, és a végrehajtás tartani fogja az emelkedő ugrást, míg talál egy *catch* blokkot, amely tulajdonképpen kezelni fogja.

Létrehozhatod *catch* blokkok láncolatát is, mint ez:

```
try
{
        //némi kód, amely dobhat index hibát
}
catch( System.IndexOutOfRangeException e )
{
        //index hiba kezelése
}
catch( System.IndexOutOfMemoryException e )
{
        //memória hiba kezelése
}
```

Minden blokk különbözőképpen fog kezelni minden hibát.

Megjegyezhetnéd, hogy az előző példákban egy *e* nevű változót használtam. Ez jelképezi az aktuális kivétel objektumot. A kivételek a hibához jellemző adatot tudnak tárolni, amely esetében szükséges kitalálnod annak pontos részleteit, hogy mi történt.

Kivételek visszadobása

Lesznek idők, amikor el akarsz kapni egy kivételt, de nem leszel képes azt kezelni. Ezek a helyzetek rendszerint akkor merülnek fel, amikor csak azt akarod tudni, hogy egy kivétel dobódott (és tulajdonképpen nem megoldani a problémát) így aztán feljegyezheted valahová vagy tehetsz valami megjegyzést. Ez esetben, minthogy nem kezelted a kivételt, vissza akarod azt dobni. Ez egy nagyon egyszerű feladat:

```
catch
{
        //csinálni valami megjegyzést itt
        throw; //visszadobni
}
```

A kivétel most visszadobódik, és tartani fogja a felugrást, míg egy másik *catch* blokk elkapja.

Saját kivételeid létrehozása

Létre tudod hozni saját kivétel osztályaidat is. Igazán könnyű csinálni, és illene mindig örökölni az osztályokat az alap *System.Exception* objektumból (vagy más beépített kivétel típusból, ha szükségleteid megkövetelik).

```
class VegzetesJatekKivetel : System.Exception
{
        //helyezz némi adatot a játék hibáról ide
}
```

Ez minden!

A saját kivételeid dobása is meglehetősen könnyű:

```
throw new VegzetesJatekKivetel();
```

Egy még összetettebb programban csinálnád a kivétel konstruktorát, hogy fogjon némi adatot a játékról, de minthogy ez egy egyszerű példa, kihagytam.

Delegáltak

A delegáltak szerintem a legmenőbb részei a C#-nak. Lehet, hogy ebben nem értesz velem egyet, de én szeretem a delegáltakat. Egy delegált az egy objektum, amely egy függvényre mutat. Ez minden róla.

C++-ban függvény mutatóid (function pointer) voltak. Az ötlet az volt, hogy volt egy változód, ami egy függvényre mutatott, és meg tudtad változtatni a változót, hogy más függvényekre mutasson futásidőben! A függvény mutatók hasznosak voltak, de csúnyák, minthogy nehezen használhatók. Nem szándékozom vesztegetni az idődet azzal, hogy mutatok egy példát róluk, de bízz bennem – hihetetlenül csúnyák voltak. A C# szerencsére teljesen megoldotta a csúnyaság-és-bonyolultság problémát a delegáltak létrehozásával.

Egy delegált létrehozása

Azért, hogy elkezdhesd használni a delegáltokat, először létre kell hoznod egy delegált-típus definíciót, hogy elmondd a fordítónak, milyen fajta függvényre szándékozik mutatni. Itt egy példa:

```
public delegate void Delegaltam();
```

Ez elmondja a fordítónak, hogy létrehoztál egy *Delegaltam* nevű delegált típust, amely függvényekre mutat, amelyek semmivel nem térnek vissza és nincsenek paramétereik sem. Itt egy minta *Urhajo* osztály, amely két függvényt definiál, egy statikust és egy nem-statikust:

```
class Urhajo
{
	public static void UrhajokBeallitasa()
	{
		//itt kell inicializálni az összes űrhajót a játékban
	}
	public void LezerTalalat()
	{
		//lézertalálat
	}
}
```

Később, valahol másütt, létrehozhatsz egy azokra a függvényekre mutató delegáltat. Itt a statikus függvényre mutató kód:

```
Delegaltam d; //delegált deklarálása
d = new Delegaltam( Urhajo.UrhajokBeallitasa ); //delegált létrehozása
d(); //hívja az Urhajo.UrhajokBeallitasa(); függvényt
```

Minden amit csináltál, a függvény (ezesetben *Urhajo.UrhajokBeallitasa*) nevének átadása a delegált konstruktorába.

Hogy létrehozz egy delegáltat, ami egy nem-statikus függvényre mutat, csináld ezt:

```
Delegaltam d; //delegált deklarálása
Urhajo u = new Urhajo(); //űrhajó létrehozása
d = new Delegaltam( u.LezerTalalat ); //delegált létrehozása
d(); //u.LezerTalalat(); hívása
```

Egy nem statikus függvényre mutató delegált létrehozása megkövetel némi többletmunkát; kell, hogy legyen jelenleg egy példánya az *Urhajo* osztálynak. Ha azt próbálod írni, hogy *new Delegaltam(Urhajo.LezerTalalat)*, akkor a fordítód kiabálni kezd, hogy: „Nem tudom, melyik űrhajót kell azon végrehajtani, te hülye!”

Minden amit csinálnod kell, hogy hívod a delegáltat, mintha az egy függvény volna, és az hívni fog bármilyen függvényt, amit beleraktál.

Természetesen létrehozhatsz olyan delegáltakat is, amelyek eltérő visszatérési értékeket és paramétereket használnak:

```
class Osztalyom
{
        public static int Fuggvenyem( int a, int b, float c )
        {
                //némi kód
        }
};
public delegate ind Delegaltam( int a, int b, float c );
```

Aztán később használhattad volna a delegáltat így:

```
Delegaltam d = new Delegaltam( Osztalyom.Fuggvenyem );
int x = d( 1, 2, 3.141596 );
```

Vagy kicserélhetted volna az *Osztalyom.Fuggvenyem* –et bármilyen függvénnyel, ami visszatér egy *int* –tel és fog két *int* –et és egy *float* -ot paraméterként.

Delegáltak láncolása

A C# fogta a függvénymutatók ötletét és egy lépéssel távolabbra ment, bemutatva a *delegált multicasting*-ot. Ez egy szép összetett kifejezés, ugye? Ehelyett a *láncolás* szót szeretem használni, mert leíróbb.

A delegált multicasting vagy láncolás azt jelenti, hogy lehetőség van delegáltak egymáshoz láncolására. Nézd ezt a kódot például (mely használja az *Urhajo* osztályt az előzőleg mutatott példából):

```
Delegaltam d; //delegált deklarálása
Urhajo u = new Urhajo(); //űrhajó deklarálása
d = new Delegaltam( Urhajo.UrhajokBeallitasa ); //delegált létrehozása
d += new Delegaltam( u.LezerTalalat ); //egy lézertalálat hívás láncolása
d(); //az UrhajokBeallitasa és a LezerTalalat hívása!
```

Ha végrehajtasz egy láncolt delegáltat, akkor a kód végrehajt minden függvényt a delegálton belül abban a sorrendben, ahogy hozzáadtad. Így az előző kódpélda utolsó sora hívni fogja az *Urhajo.UrhajokBeallitasa* és aztán az *u.LezerTalalat* függvényeket.

El is távolíthatsz delegáltakat:

```
d -= new Delegaltam( Urhajo.UrhajokBeallitasa );
d(); //most csak az u.LezerTalalat függvényt hívja
```

Én igazán szeretem a delegáltak láncolásának hatalmas ötletét. Olyan sok lehetőséget ad a programjaidhoz.

Megjegyzés: ha olyan delegáltakat láncolsz, amelyek visszaadnak értékeket, akkor belefuthatsz néhány problémába. A módszer, amellyel a C# kezeli ezt a helyzetet a csak az utolsó függvény értékének visszaadása a delegálton belül. Az összes többi visszatérési érték eldobódik. Tehát ne csinálj visszatérési értéknek olyan dolgokat, amelyek abszolút létfontosságúak a programodnak, ha tudod, hogy a függvény egy láncolt delegált belsejébe kerül.

Gyűjtemények

Az adatgyűjtemények egy hatalmas téma a számítógép programozásban, és megérdemel egy egész könyvet magának. Szándékaim szerint kell adni neked egy korlátozott leírást a C# gyűjteményeiből itt. Előzőleg láttál már egy gyűjtemény típust ebben a könyvben: a tömböt. Minthogy már átmentem ezen, a következő alfejezetekben további haladó C# konténereket mutatok.

Az ArrayList

A tömbök nagyszerűek az információ tárolásához, de van egy nagy hátrányuk: nem tudod átméretezni őket. Ha át akarsz méretezni egy tömböt, akkor egy újat kell létrehoznod, bele kell másolni mindent, és el kell dobni a régi tömböt. Ez sok elpazarolt erőfeszítés.

Szerencsére a .NET keretrendszer nyújtja neked a *System.Collections.ArrayList* osztályt, mely önműködően átméretezhető. Meglehetősen egyszerű használni:

```
System.Collections.ArrayList list = new System.Collections.ArrayList();

// elemek adása
list.Add( 20 ); // 20
list.Add( 30 ); // 20, 30
list.Add( 40 ); // 20, 30, 40
list.Insert( 1, 60 ); // 20, 60, 30, 40
list.Insert( 0, 90 ); // 90, 20, 60, 30, 40

// betekintés az elemekbe
int x;
x = (int)list[0]; // 90
x = (int)list[2]; // 60

// elemek keresése
bool b;
b = list.Contains( 20 ); // igaz
b = list.Contains( 10 ); // hamis
x = list.IndexOf( 90 ); // 0
x = list.IndexOf( 30 ); // 3

// elemek eltávolítása
list.Remove( 30 ); // 90, 20, 60, 40
list.RemoveAt( 0 ); // 20, 60, 40
list.Clear(); // üres!
```

Ennek a példának meglehetősen jó ötletet kell adnia neked arról, hogy a tömblisták hogyan működnek. Nem igazán nehéz.

Az egyedüli dolog, amit észben kell tartanod, az az elemek eltávolítása és hozzáadása a lista közepéhez. Legyél tisztában azzal, hogy a .NET keretrendszer egy tömbben tárolva tartja a listát, tehát mikor beillesztesz valamit középre, akkor minden fölfelé mozog az azutáni hely után, és ha valamit eltávolítasz

középről, akkor mindennek lefelé kell mozognia. Ez nem olyan nagy dolog, de illene tisztában lenned vele.

A másik dolog, amire emlékezned kell, hogy a tömblisták bármilyen típusú objektumokat képesek tárolni. Ez azt jelenti, hogy végrehajthatsz műveleteket, mint ezt, tárolva több különböző fajta objektumot azonos listában:

```
list.Add( 10 );
list.Add( „Én egy mintaszerű modern vezérőrnagy vagyok!” );
list.Add ( new Urhajo() );
```

Ez azt is jelenti, hogy el kell dobnod az objektumokat, amikor újra kiveszed a listából:

```
int x = (int)list[0];
string s = (string)list[1];
Urhajo uh = (Urhajo)list[2];
```

Ez kicsit unalmas tud lenni, ha nem emlékszel, hogy mit tároltál a listádban.

A *typeof* operátor

Ha nem vagy biztos egy objektum típusában, akkor végrehajthatsz egy próbát rajta, mint ez:

```
if( list[0].GetType() == typeof(int) )
// az elem egy int
if( list[1].GetType() == typeof(string) )
// az elem egy string
if( list[2].GetType() == typeof(Urhajo) )
// az elem egy űrhajó
```

És így biztos lehetsz benne.

Az *is* operátor

A másik művelet, amit használhatsz a típusokon, az az *is* operátor, melyet bámulatosan egyszerű használni:

```
if( list[0] is int )
// az elem egy int
if( list[1] is string )
```

```
// az elem egy string
if( list[2] is Urhajo )
// az elem egy űrhajó
```

Ez a módszer sokkal tisztább, mint a korábban már bemutatott *GetType* eljárás.

Az *as* operátor

Másik mód az objektumok tesztelésére a gyűjteményedben, ha tudod, hogy referenciák és nem érték-típusok, az *as* operátor használata. Itt egy példa:

```
Urhajo u = list[2] as Urhajo;
```

Ha az elem egy *Urhajo*, akkor az *u* tartani fog egy hivatkozást az űrhajó objektumra a listán belül. Ha pedig nem *Urhajo*, akkor az *u* egy *null* értéket fog tartani helyette.

Megjegyzés: nem használhatod ezt az operátort érték-típusokkal, mert hivatkozások használata által működik. Ha hibázik, akkor egy *null* hivatkozással tér vissza, és ilyenek nem lehetnek az érték-típusok.

Hash táblák

A Hash táblák különösen hatékony gyűjtemény típusok. Ezek kulcs-érték párokat tárolnak, ami azt jelenti, hogy hasonlóan cselekednek, mint egy tömb, kivéve hogy bármit használhatsz indexként, nemcsak számokat. Itt egy egyszerű példa, amely szövegeket használ kulcsként:

```
System.Collections.HashTable table = new System.Collections.HashTable();
// némi adat tárolása a táblában:
table[“pi”] = 3.14159;
table[“e”] = 2.71828;
table[“negyvenkettő”] = 42.0;
// most visszaszerezni azt:
double d;
d = table[“e”]; // 2.71828
d = table[“negyvenkettő”]; // 42.0
d = table[“pi”]; // 3.14159
// egy bejegyzés eltávolítása:
bool b;
b = table.Contains( “pi” ); // igaz
table.Remove( “pi” );
b = table.Contains( “pi” ); // hamis
```

Sajnos igazán nincs elég helyem végigmenni egy hash tábla belső részletein. De hadd adjak egy rövid eligazítást arról, hogy hogyan működnek. Akármikor beillesztesz egy kulcs/érték párost egy táblába, a kulcs értékek kivonatolódnak egy számszerű értékbe. A C# ezt a kulcs objektum *GetHashCode* függvénye hívásával végzi el, mely *int* értékként tér vissza. Minden objektumnak megvan beépítve ez a függvény, tehát használhatsz bármit mint egy kulcs értéket. A C# aztán kezeli ezt a hash értéket a táblában, mint egy indexet, és gyorsan hozzáférhetővé teszi neked a tételt.

Megjegyzés: én azt ajánlom, hogy hozd létre a saját hash függvényeidet az osztályaidnak, mivel a beépített módszer nem elég jó az egyéni adataidnak. Minthogy valószínűtlen, hogy szükséged lesz használni a saját kulcs típusaidat egy egyszerű játék számára, el szándékozom hagyni ezt a témát, és tőled függ a továbbiak felfedezése ezzel kapcsolatban. Írtam egy terjedelmes fejezetet a hash táblákról az *Adatszerkezetek játékprogramozóknak* című könyvben; bár e könyv elsődlegesen a C++-szal foglalkozik, az elgondolás ugyanaz maradt, ezért nem számít, hogy milyen nyelvet használsz.

Vermek és Sorok

A .NET keretrendszer vermeket és sorokat is nyújt, melyek egyszerű egyenes irányú tárolók, amik lehetővé teszik a hozzáférést adatokhoz különleges módon. Egy sor az egy első-be-első-ki (first-in-first-out, azaz FIFO) tároló (az elsőnek behelyezett objektum fog elsőként eltávolítódni), egy verem pedig egy utolsó-be-első-ki (last-in-first-out, azaz LIFO) tároló (az utolsóként behelyezett objektum fog elsőként eltávolítódni).

Itt egy példa a sorra:

```
System.Collections.Queue q = new System.Collections.Queue();
q.Enqueue( 10 ); // 10
q.Enqueue( 20 ); // 10, 20
q.Enqueue( 30 ); // 10, 20, 30
q.Enqueue( 40 ); // 10, 20, 30, 40
q.Enqueue( 50 ); // 10, 20, 30, 40, 50
// csak nézd meg a tetejét
int x = (int)q.Peek(); // 10
// vagy nézd meg és távolítsd el a tetejét
x = (int)q.Dequeue(); // 10, q = 20, 30, 40, 50
x = (int)q.Dequeue(); // 20, q = 30, 40, 50
x = (int)q.Dequeue(); // 30, q = 40, 50
x = (int)q.Dequeue(); // 40, q = 50
x = (int)q.Dequeue(); // 50, q = {üres}
```

Ez egy eléggé egyszerű elgondolás, és eléggé könnyen kezelhető.

A vermek majdnem azonosak a sorokkal, kivéve a tételek eltávolításának módját. Ahelyett, hogy az elejéről vennék le a dolgokat, a vermek hátulról veszik le:

```
System.Collections.Stack s = new System.Collections.Stack ();
s.Push( 10 ); // 10
s.Push( 20 ); // 10, 20
s.Push( 30 ); // 10, 20, 30
s.Push( 40 ); // 10, 20, 30, 40
s.Push( 50 ); // 10, 20, 30, 40, 50

// csak nézd meg a tetejét
int x = (int)s.Peek(); // 50

// vagy nézd meg és távolítsd el a tetejét
x = (int)s.Pop(); // 50, s = 10, 20, 30, 40
x = (int)s.Pop(); // 40, s = 10, 20, 30
x = (int)s.Pop(); // 30, s = 10, 20
x = (int)s.Pop(); // 20, s = 10
x = (int)s.Pop(); // 10, s = {üres}
```

Egyéb gyűjtemények

Van még egy csomó más gyűjtemény is a C#-ban, de amiket én bemutattam itt, azokat fogod leginkább használni. Hagyni foglak, hogy felfedezd a magad módján a többi gyűjteménytípust, mivel a Microsoft igen jól dokumentálta őket az MSDN-ben (amely megtalálható online a http://msdn.microsoft.com/ címen). Csak keresd a „System.Collections”-t, és találnod kell több információt, mint amennyire szükséged lesz.

Állomány hozzáférés

A játékok összetett lények. A napok, mikor feltölthettél egy játéktermi játékot, és megverted néhány óra alatt, már elmúltak; most léteznek játékok, amelyek teljesítése több hetes egyfolytában vele foglalkozást igényelnek.

Tulajdonképpen senkinek sincs olyan kitartása, hogy olyan sokáig játsszon egy játékot – ez őrültség lehet. Tehát kell neked egy módszer, hogy tárold a játék adatait azért, hogy a felhasználója visszajöhessen és onnan folytathassa, ahol abbahagyta.

Folyamok

Gyakorlatilag bármely számítógép rendszer, állományok *folyamok*nak (*stream*-ek) nevezett elvont objektumokként kezeltek, mert rendszerint egy időben olvasol vagy írsz adat egy hosszú „folyamát”.

Bár folyamok nemcsak állományoknak vannak, hanem használatosak mindenféle dologra, mint szöveg be- és kimenet és hálózati kommunikáció. Ebben a fejezetben viszont csak az állomány folyamokat érintjük.

Olvasás és írás

Az alap osztály az összes folyam számára a .NET-ben a *System.IO.Stream*, amely egy elvont osztály. A folyamoknak van mindenféle függvényük, de azok a függvények, amiket leginkább érinteni fogsz, a *Write*, *WriteByte*, *Read* és *ReadByte*.

Pontosan azt csinálják, amiket ígérnek csinálni, és lehetővé teszik bájtok tömbjének vagy csak egy egyszerű bájtnak az írását vagy olvasását egy folyamba. Például (tételezzük fel, hogy az *s* folyam már érvényes):

```
byte b;
b = s.ReadByte(); // az első két bájt olvasása
b = s.ReadByte(); // a folyamból

s.WriteByte( 10 ); // 10-es érték írása
s.WriteByte( 255 ); // 255-ös érték írása

byte[] array = new byte[4];

// olvasás a tömbbe a 0 indextől kezdve, és olvasni legfeljebb 4 bájtot
s.Read( array, 0, 4 );
s.Write( array, 0, 4 ); // írni mindent újra
```

Ez a példa eléggé butaság, mert nem fogod gyakran kombinálni az olvasó és író műveleteket a kód azonos általános területén. Legtöbbször az egyik műveletet hajtod végre vagy a másikat, és nem mindkettőt. Ez csak a bemutatása volt annak, hogy a függvények hogyan működnek.

Elárasztás és bezárás

A két másik fő függvény, amit a folyamok támogatnak, az a *Flush* és a *Close*.

Ellentétben azzal, amit gondolhatsz, amikor írsz egy folyamba, amit írsz az tulajdonképpen nem íródik ki azonnal. Ennek oka az, hogy az eszközhozzáférés lassú. A feldolgozás szempontjából idő szükséges, és tulajdonképpen egy bájtnak egy állományba kiírása hosszú időt vesz igénybe. Sok ilyen műveleti költség van, így egy bájt kiírásának időmennyisége csak némileg kisebb, mint sok bájté.

Ezokból a folyamok egy átmeneti tárolót, úgynevezett *cache*-et használnak. Akármikor írsz valamit egy folyamba, a számítógép rajta tartja egy kicsit, és mikor a cache eléggé megtelik, akkor a cache tartalma végül kiíródik a jelenleg csatlakoztatott eszközre. Sajnos ez okozhat némi fájdalmat a fenekedben, mert nincs ötleted arról, hogy ez tulajdonképpen mikor szándékozik megtörténni. Néha szükséges megbizonyosodnod arról, hogy valami azonnal kiírandóvá válik. Ezért van a *Flush* módszer. Nézd meg:

```
s.WriteByte( 20 ); //20-as érték cache-elése
s.Flush(); //megbizonyosodni, hogy a cache manuálisan kiíródik azonnal
```

Megjegyzés: a próbák a számítógépemen azt mutatják, hogy az én állományfolyamaim cache-e 1024 bájt, mielőtt önműködően kiíródnak a lemezre. A te eredményeid ettől eltérőek lehetnek.

És, természetesen, a *Close* függvény egyszerűen bezárja a folyamot, így nem lehet rajta további írást vagy olvasást végrehajtani. Egy folyam bezárása flush-olja is azt.

Olvasók és Írók

Nyers bájtok olvasása és írása egy folyamba gyorsan unalmassá válik. Szándékozni akarsz olvasni és írni még fontosabb dolgokat, mint int-ek és float-ok, vagy még inkább szöveges string-ek. Emiatt a .NET keretrendszer magábafoglal specializált osztályokat, amiket *olvasók*nak és *írók*nak hívnak, melyek önműködően végrehajtják neked az adattípusok olvasását és írását. Ezek az osztályok párosítva vannak egy folyammal, és megengedik neked, hogy olvass vagy írj specializált adatot egy folyamba.

Szövegek és folyamok

A legkönnyebben használható osztálypár a *System.IO.StreamReader* és a *System.IO.StreamWriter* osztályok. Lehetővé teszik számodra szövegek olvasását és írását állományokból és állományokba. A szöveges állományokat rendszerint ASCII fájloknak nevezik, minthogy az ASCII formázási szabványt használják – bár nem mindig ez kell. A .NET keretrendszer lehetővé teszi a formázás megváltoztatását, bár nem igazán szándékozol érintett lenni ezzel, hacsak nem nemzetközi játékfejlesztést végzel.

Olvasás

Mikor olvasol egy szöveges folyamból, van néhány függvény, melyekkel illene tisztában lenned, és ezek a: *Read*, *ReadLine* és *ReadToEnd*.

A *Read* olvas egy meghatározott számú írásjelet egy folyamból, tárolja azokat egy karakter tömbben és visszatér az olvasott írásjelek teljes számával. Itt egy példa (tételezzük fel, hogy a *StreamReader s* már be van állítva egy mögöttes eszközzel; majd megmutatom, hogyan kell azt csinálni):

```
char[] buffer = new char[32];
int x;
x = s.Read( buffer, 0, 32 ); // olvasás 32 írásjelig és tárolni őket egy 0 indextől kezdődő
//bufferben
```

Az *x* változó most tartani fogja a jelenleg olvasott írásjelek számát (ha a folyamnak nincs 32 olvasott karaktere, akkor annyival tér vissza, amennyivel tud), és a *buffer* tömb tartalmaz mindent, ami olvasva volt.

Egy könnyebben használható függvény a *ReadLine*:

```
string str;
str = s.ReadLine(); // a szöveg következő sorát olvassa
```

Ez minden, ami van. A függvény visszatér egy szöveggel, minden dolgot szépen és könnyen csinál neked.

Az utolsó lehetőség az egész folyam egyszeri olvasása:

```
string str;
str = s.ReadToEnd(); // olvas MINDENT
```

Ez a függvény is visszaad egy szöveget. Valószínűleg nem akarod ezt a függvényt hatalmas állományokon hívni, de biztos vagyok benne, hogy ezt már kitaláltad.

Írás

Elsődlegesen két olyan függvény van, amelyekkel tisztában kellene lenned, mikor írsz a *StreamWriter*-ekbe: a *Write* és a *WriteLine* függvény. Lefogadom, kitalálod, hogy mit csinálnak.

```
int x = 10;
double y = 3.14159;
string s = “Szia ott!”;
s.Write( x ); // írása “10”-nek, mint szöveg
s.Write( y ); // írása “3.14159”-nek, mint szöveg
s.Write( s ); // írása “Szia ott!”-nak
// a folyam most ezt tartalmazza: “103.14159Szia ott!”
```

A *Write* függvény túlterhelt, hogy elfogadja a beépített adattípusok nagy részét, a munkádat meglehetősen könnyűvé téve – neked nem kell magadnak mindent kézzel átalakítani stringgé.

A *WriteLine* meglehetősen hasonlóan cselekszik, kivéve hogy minden írás után hozzárak egy újsor karaktert:

```
int x = 10;
double y = 3.14159;
string s = “Szia ott!”;
s.WriteLine( x ); // írása “10”-nek, mint szöveg
s.WriteLine( y ); // írása “3.14159”-nek, mint szöveg
s.WriteLine( s ); // írása “Szia ott!”-nak
// a folyam most ezt tartalmazza:
// 10
// 3.14159
// Szia ott!
```

Elég könnyű, ugye?

Bináris folyamok

Néha a szöveges állományok túl nagyok. Ha belegondolsz, sok időt pazarolsz, ha ASCII adatot írsz ki. Egy integer négy bájt hosszú a számítógépen belül, de ha az 1000.000.000 szám egyéni karaktereit írod ki, akkor lefoglalsz tíz írásjelet, ami több mint kétszer akkora, mint négy bájt, amit a számítógép tárolásra használ. Szerencsére a .NET keretrendszer nyújt olyan folyamokat, amelyek lehetővé teszik bináris, azaz kettős számrendszerbeni értékek olvasását és írását közvetlenül egy folyamba ahelyett, hogy átalakítanád őket szöveggé vagy szövegből. Ezeket az osztályokat *System.IO.BinaryReader* és *System.IO.BinaryWriter* osztályoknak hívják.

Olvasás

A *BinaryReader* osztály sok függvényt nyújt bináris adat egy állományból történő olvasására. Szükséged van ezekre, minthogy minden egyszerű bináris típus különbözően van kódolva. Itt az olvasó függvények listája:

- ReadBoolean
- ReadByte
- ReadBytes
- ReadChar
- ReadChars
- ReadDecimal
- ReadDouble
- ReadInt16
- ReadInt32
- ReadInt64
- ReadSByte
- ReadSingle
- ReadString
- ReadUInt16
- ReadUInt32
- ReadUInt64

Ezen függvények nevei meglehetősen önleíróak, de néhányuk igényel némi kifejtést. Például a *float*-ok olvasására szolgáló függvényt *ReadSingle*-nek hívják *ReadFloat* helyett. Ez bizonyára azért van, mert *System.Single* a neve a *float*-okat képviselő .NET adattípusnak.

A többi függvény, amelyek magyarázatot igényelnek, a *ReadChars*, a *ReadBytes* és a *ReadString*.

Írásjelek és bájtok olvasása meglehetősen egyszerű; elmondod a függvénynek, hogy mennyit akarsz olvasni, és az visszatér egy írásjeltömbbel, amennyit olvastunk:

```
char[] ctomb = s.ReadChars( 20 ); // olvas 20 karaktert, ha lehetséges
byte[] btomb = s.ReadBytes( 20 ); // olvas 20 bájtot (előjeles), ha lehetséges
```

A tömb mérete lehet kevesebb is, mint a kért karakterek vagy bájtok száma, attól függően, hogy mennyi van otthagyva a folyamban.

Egy szöveg olvasása némileg bonyolultabb. Mikor ebben a módszerben olvasunk sztringeket, akkor a függvény feltételezi, hogy a sztringnek két része van: egy leíró részletezi azt, hogy jelenleg milyen hosszú a szöveg, és az aktuális szöveg tartalom. A *ReadString* függvény feltételezi, hogy az első bájt (vagy több, eltérő lehet attól függően, hogy milyen hosszú a szöveg) amit a folyamból olvas, az a leíró mérete, aztán olvassa azt a mennyiségű írásjelet egy szövegbe és visszatér vele. Szükséges lesz használnod ezt a módszert a *System.IO.BinaryWriter.Write* függvénnyel kapcsolatban.

```
string str = s.ReadString();
```

A többi függvény meglehetősen egyszerű:

```
int x = s.ReadInt32();
uint y = s.ReadUInt32();
float z = s.ReadSingle();
```

És így tovább.

Írás

Bináris adat írása hihetetlenül egyszerű, mert a *BinaryWriter* osztály önműködően tudja, hogy milyen fajta adatot akarsz írni, az átadott paraméter típusát figyelembe véve:

```
int x = 20;
float y = 12.32154;
char z = ‘P’;
s.Write( x );
s.Write( y );
```

```
s.Write( z );
```

És ennyit ezekről.

Állomány folyamok

Az összes folyam osztály, amit előzőleg mutattam neked, alapvetően határfelületek, interfészek számodra. C#-ban van egy csomó folyam, mint memória folyamok és hálózati folyamok, de ebben a szakaszban csak az állomány folyamok lesznek érintve.

C#-pal az állomány folyamok elég sok pillanatot jelentenek. Használod a statikus *System.IO.File* osztályt hogy állományokat nyiss meg, és az visszaad egy *System.IO.FileStream* objektumot hogy használhasd az olvasókkal és írókkal, amiket előzőleg bemutattam neked.

Sok függvény van a *File* osztályban, amelyek lehetővé teszik, hogy játszogass az állományokkal, de elsődlegesen azokkal kapcsolatban leszel érintett, amelyek nyitott fájlokkal foglalkoznak: *OpenRead* és *OpenWrite*. Van egy másik, az *Open*, amely lehetővé teszi, hogy beállítsd, pontosan hogyan nyíljon meg egy fájl, bár ebbe nem igazán fogok belemenni; a többi függvény ad elég funkcionalitást a munkához e pillanatban.

Az *OpenRead* és az *OpenWrite* is visszatér egy *FileStream* objektummal:

```
System.IO.FileStream file;
// valami írása a fájlba
file = System.IO.File.OpenWrite( “blabla.txt” );
// <írási műveletek végrehajtása azon itt>
file.Close();
// most visszaolvassuk bele az adatot:
file = System.IO.File.OpenRead( “blabla.txt” )
// < olvasási műveletek végrehajtása azon itt >
file.Close();
```

Az előző kódszelet lehetővé teszi, hogy olvass és írj nyers bájtokat, használva a *ReadByte* és *WriteByte* függvényeket, de nem nagyon leszel érdekelt ezen műveletek végrehajtásában, mert a nyers bájtok olvasása és írása unalmas. Helyette valószínűleg párosítani szándékozol az állomány folyamot egy olvasóba vagy íróba.

Itt egy példa annak bemutatására, hogyan kell írni egy *StreamWriter* objektumot felhasználva:

```
System.IO.FileStream file;
System.IO.StreamWriter writer;
// a fájl megnyitása
file = System.IO.File.OpenWrite( “blabla.txt” );
// egy író létrehozása azzal a fájllal
writer = new System.IO.StreamWriter( file );
writer.WriteLine( “Üdv minden boldog embernek!” );
writer.WriteLine( 542 );
writer.Close();
```

És most írtál két szöveges sort a „blabla.txt” nevű állományba. Itt van, hogy hogyan olvasnád vissza:

```
System.IO.FileStream file;
System.IO.StreamReader reader;
// a fájl megnyitása
file = System.IO.File.OpenRead( “blabla.txt” );
// egy olvasó létrehozása azzal a fájllal
reader = new System.IO.StreamReader( file );
string str;
str = reader.ReadLine(); // “Üdv minden boldog embernek!”
str = reader.ReadLine(); // “542”
int i = Int32.Parse( str ); // az érték integerré alakítása
reader.Close();
```

Ha a bináris állományok használatát részesíted előnyben, akkor a bináris olvasókat és írókat kellene használnod (*BinaryReader* és *BinaryWriter*), mint ahogy előzőleg mutattam.

Véletlen számok

Véletlen számokat meglehetősen gyakran használnak játék programozás során, mivel különböző zavaros, kaotikus körülmények szimulálására használtak, ami megtalálható a való világban.

Azért, hogy elkezdj használni véletlen számokat, az első szükséges dolog létrehozni egy számgenerátort, mint ez:

```
System.Random r = new System.Random( 0 );
```

A konstruktorba átadott integer-t *mag érték*nek (*seed value*) hívják. A véletlen szám generátorok nem igazi véletlenek; helyette algoritmusokat használnak, hogy ál-véletlen számokat hozzanak létre, vagy számokat, amik neked és nekem véletlennek tűnnek, de amelyek matematikailag nem véletlenek (ami azt jelenti, hogy nem teljesen kaotikusak). Játékok számára az ál-véletlen számok is megfelelnek.

Magok

A mag érték meghatározza azt, hogy hol kell elkezdeni a véletlen számok létrehozását. Ha adsz egy magot egy véletlen szám generátornak, és leveszel öt számot róla, aztán pedig adsz egy teljesen eltérő generátort ugyanazon kezdő maggal, annak első öt száma azonos lesz. Például:

```
System.Random r = new System.Random( 0 );
int x = r.Next(); // ezt kellene tartania: 1559595546
System.Random s = new System.Random( 0 );
int y = s.Next(); // szintén ezt kellene tartania: 1559595546
```

Bármikor azt akarod, hogy egy generátor számok azonos sorát hozza létre, adj annak egy közös magot.

Ámbár az nincs megkövetelve, hogy legyen egy mag. Valójában legtöbbször valószínűleg nem akarsz beállítani egy kemény-kódolású magot, mert az kevésbé véletlenszerűnek fogja mutatni a játékodat, mivel azonos véletlen számsort generál minden időben, mikor valaki játszik.

Helyette létre tudsz hozni egy generátort, mint ez:

```
System.Random r = new System.Random();
```

Mikor nem gondoskodsz egy magról, akkor a rendszer a jelenlegi időn alapuló magértéket fog neked használni önműködően, úgyhogy nem kell aggódnod ezzel kapcsolatban.

Számok generálása

Ahogy ki is találhatod, a *Next* függvény visszaadja a következő véletlen számot a sorozatban.

Megjegyzés: a következő generáló példák mindegyike feltételezi, hogy egyik a másik után van futtatva. Ha ugyanezt a kódot futtatod, akkor ugyanazt az eredményt kell kapnod, minthogy ugyanazt a magot használod.

Itt egy példa:

```
System.Random r = new System.Random( 0 );
int x;
x = r.Next(); // 1559595546
x = r.Next(); // 1755192844
x = r.Next(); // 1649316166
x = r.Next(); // 1198642031
x = r.Next(); // 442452829
x = r.Next(); // 1200195957
```

Most azok teljes mértékben hatalmas számok, és valószínűleg nem túl hasznosak számodra. Szerencsére a .NET keretrendszer nyújt néhány más módot számok létrehozására. Például előállíthatsz számokat 0-tól egy megadott mennyiségig:

```
x = r.Next( 100 ); // 90
x = r.Next( 100 ); // 44
x = r.Next( 100 ); // 97
x = r.Next( 100 ); // 27
```

Ez a hívás 0 és 100 közötti számokat állít elő. Van egy másik változata is a függvénynek, mely lehetővé teszi kezdeti határérték megadását is:

```
x = r.Next( 50, 100 ); // 64
x = r.Next( 50, 100 ); // 73
x = r.Next( 50, 100 ); // 81
x = r.Next( 50, 100 ); // 73
```

Ezzel 50-től 100-ig hozol létre számokat.

További előállító módszerek

Van két másik mód, amellyel előállíthatsz véletlen számokat a *Random* osztály használatával: generálhatsz dupla-pontosságú számokat, és előállíthatod véletlen bájtok tömbjét is.

Mindkettő eléggé egyszerű:

```
double d;
d = r.NextDouble(); // 0.98215125314060192
d = r.NextDouble(); // 0.030366990729406004
d = r.NextDouble(); // 0.86237015382497118
d = r.NextDouble(); // 0.99534708121574811
```

A double-ok 0.0 és 1.0 értékek között jöttek létre.

És végül megtölthetsz egy tömböt véletlen bájtokkal:

```
byte[] b = new byte[10];
r.NextBytes( b ); // 173, 80, 209, 217, 253, 8, 179, 134, 239, 176
```

Nagyjából ez minden, ami érdekelhet a *System.Random* osztályból.

Fölött és túl

Ezelőtt már néhányszor megemlítettem, hogy a C#, egyesítve a .NET osztály könyvtárral, egyike a létező legnagyobb és legösszetettebb programnyelveknek. Tulajdonképpen egy ilyen méretű könyv nem reménykedhet, hogy mindent lefed róla. A főbb részeit megmutattam a nyelvnek – az anyag, amit leggyakrabban fogsz használni, és az anyag, ami fontos a játékprogramozáshoz.

Sajnos még tonnányi dolog van, amikről nem esett szó. A következő alfejezetekben végigmegyek néhány témán, amelyeket szerintem érdemes átnézned a szabadidődben.

A preprocesszor

A számítógép programozás régebbi napjaiban néhány programozási nyelvnek (leginkább a C-nek) volt egy preprocesszora, egy különleges szakasza a fordítási folyamatnak, ami végigmenne a kódodon, és végrehajt egyszerű szöveg-áthelyezési műveleteket. Nem akarok részletesen belemenni abba, hogy miért volt erre szükség, elegendő azt mondani, hogy volt.

Sajnos ez egy csúnya fenevad, és örülök, hogy a modern nyelvekben ezt az út szélére dobták. A C# úgy döntött, hogy beépít magába egy nagyon korlátozott preprocesszort, ami lehetővé teszi különböző kóddarabok válogatottan történő

fordítását, a preprocesszor értékeinek értékétől függően. Valószínűleg nem lesz szükséged a használatára, de ha érdekel, akkor elmélyedhetsz benne.

Operátor túlterhelés

A C# örökölt egy meglehetősen szép tulajdonságot a C++-ból: az *operátor túlterhelés*t. Ez az a tulajdonság, ami lehetővé teszi neked, hogy meghatározz új műveleteket az osztályoknak, így csinálhatsz olyasmiket, mint ez:

```
Urhajo s = new Urhajo();
Urhajo t = new Urhajo();
s = s + t;
```

Az operátor túlterhelés lehetővé teszi, hogy használd az alapértelmezett műveleteket (mint +,-,*,/ és így tovább) a saját egyéni osztályaidon.

Természetesen, mit csinál pontosan egy űrhajó hozzáadása egy másikhoz? Ki tudja? Ennyi elég, hogy miért nem mentem át részletesen ezen a témakörön; az operátor túlterhelésnek a matematikai osztályokon kívül igazán kis felhasználási területük van. Ha bele akarsz nézni, menj tovább.

Változó paraméter listák

A C# lehetővé teszi neked, hogy létrehozz függvényeket, amelyek paraméterek változószámát tudják fogni. Nekem személyesen még sosem volt ilyesmire szükségem, de feltételezem, hogy hasznos lehet bizonyos helyzetekben. Ha érdekel, akkor nézz utána a *params* kulcsszónak.

Kockázatos kód

A C# lehetővé teszi azt is, hogy létrehozd ún. *kockázatos kód* (*unsafe code*) blokkjait. Ez elsősorban megengedi neked a hozzáférést a régi C API-khoz, és azt is, hogy legyen közvetlen memória hozzáférésed bármilyen szemét gyűjtési védelem nélkül. Igazán nem ajánlanám neked a kockázatos mód használatát, hacsak nem pontosan tudod, hogy mit csinálsz.

II. rész: Játékprogramozás C#-ban

6. fejezet

Egy keretrendszer felállítása

Amit eddig csináltál, az egyszerű konzol-módbeli anyag. Ez nagyszerű, de nem igazán segít neked játékot csinálni az összes grafikai jósággal, amit a DirectX nyújt. Szándékozom mellőzni az összes hülye GDI és GUI vezérlő anyagot, mert – mint egy játékprogramozó – valószínűleg nincs szükséged ezeket tudni. A GUI összetevőkről tanulni később hasznos lehet, mikor bonyolult játék erőforrás szerkesztőket csinálsz, de sajnos nincs elég hely, hogy beemeljem ezt az információt ebbe a könyvbe.

Egy project létrehozása

A korszerű Egységes Fejlesztői Környezetek (Integrated Development Environment – IDE) legrokonszenvesebb vonása, hogy tartalmaznak varázslókat.

Megjegyzés: egy IDE egy egyszerű program, amit a kódod fejlesztésére és fordítására használsz. A Microsoft Visual C#-ja egy IDE, éppúgy mint a SharpDevelop. A SharpDevelop egy teljesen ingyenes C# IDE, amit letölthetsz a következő címről: http://www.icsharpcode.net/OpenSource/SD/

Egy varázsló csupán egy egyszerű program, amely végrehajt egy összetett feladatot neked, kezeli az összes kis piszkos részletet, tehát nem kell vesztegetned az idődet magadnak a kezelésükkel.

A játékprogramozás régi rossz napjaiban az emberek órákat tölthettek egy keretrendszer összeállításával. Egy keretrendszer programozói értelemben olyasvalami, ami nem tartalmazna kódot amely jellemző volt bármely egyszerű játékra, és így használható volt ismételten különböző játékok változataival. A keretrendszereknek jellemzően van kódjuk kezelni a határfelületet az operációs rendszerrel és így tovább.

Ma, mindkét fő C# IDE-nek (VC# és SharpDevelop) van varázslója, amelyek önműködően létrehoznak neked keretrendszert.

SharpDevelop

SharpDevelop-ban meglehetősen könnyű Direct3D projektet létrehozni. Csupán annyit kell tenned, hogy a *File* menüben a *New*-re kattintasz és a *Combine* lehetőséget választod. Ezután a projekt varázsló fel is bukkan.

Megjegyzés: a SharpDevelop IDE néhány korábbi változatában a *New Project*-et kell választani a *New Combine* helyett, szóval ne riasszon el, ha a *New Combine* lehetőség nem látható. Egy project az egy forrásállományok gyűjteménye, egy összeállítás (combine) pedig projektek gyűjteménye. Mint minden szoftvernél, a verziószámok eltérőek lehetnek, tehát olvasd el a dokumentációját, hogy kitaláld, hogyan kell létrehozni egy összeállítást.

A varázsló meglehetősen erős és sok beállítási lehetőséget ad neked. Az opció, amiben érdekelt vagy, az a Direct3D project. A varázsló megkérdezi tőled a projekted nevét és egy helyet, ahová elhelyezi, és ez minden. Amint begépelted ezeket az adatokat, a varázsló önműködően létre fogja hozni neked a forráskód állományokat és a projekt fájlokat, és ezzel készen is vagy az induláshoz!

Fájlok

A SharpDevelop általam használt változatában (0.99) a varázsló négy állományt hozott létre:

- 01-SharpDevelopFrame1.cmbx
- 01-SharpDevelopFrame1.prjx
- AssemblyInfo.cs
- MainClass.cs

Az első két fájl az összeállítás és a projekt állomány. Ezek egyszerűen információkat tárolnak a projekt fájljaidról. A következő, az AssembyInfo.cs egy C# forrásfájl, ami információkat tartalmaz, melyek elmondják a .NET-nek, hogy van összeállítva a projekted – több, mint valószínű, hogy többé nem kell hozzányúlnod ehhez a fájlhoz. Az utolsó állomány pedig az, amit látsz a képernyőn a varázsló lefutása után. Ez a forrásfájl tartalmazza az összes kódot, ami szükséges azért, hogy elkezdhesd készíteni a saját Direct3D alkalmazásodat.

A kód kifejtése

A kód meglehetősen egyszerű, de mivel nem láttad ezen dolgok nagyját ezelőtt, ezért végigmegyek rajta sorról sorra a következő alszakaszban.

A könyvtárak

A keretrendszer első része a használt könyvtárak:

```
using System;
using System.Collections;
using System.ComponentModel;
using System.Drawing;
using System.Windows.Forms;

using Microsoft.DirectX;
using Microsoft.DirectX.Direct3D;
```

Az első két könyvtárat már használtad ezelőtt, de a következő ötöt nem.

- System.ComponentModel: különböző osztályokat tartalmaz, melyek egy ablakos program alapvető vezérlő viselkedéséhez szükségesek;
- System.Drawing: osztályokat tartalmaz, melyek (magától értetődően) a rajzolási függvényekhez használtak;
- System.Windows.Forms: a kijelző formákhoz használt osztályokat tartja;
- Microsoft.DirectX: DirectX információkat tartalmaz;
- Microsoft.DirectX.Direct3D: a Direct3D-ről tartalmaz információkat.

Az osztály áttekintése

A fő osztály, az összes kivágott kóddal, így néz ki:

```
public class MainClass : Form
{
	Device device = null;
	public MainClass()
	public bool InitializeGraphics()
	virtual void InvalidateDeviceObjects(object sender, EventArgs e)
	virtual void RestoreDeviceObjects(object sender, EventArgs e)
	virtual void DeleteDeviceObjects(object sender, EventArgs e)
	virtual void EnvironmentResizing(object sender, CancelEventArgs e)
	virtual void FrameMove()
	virtual void Render()
	public void Run()
	override void OnPaint(PaintEventArgs e)
	override void OnKeyPress(KeyPressEventArgs e)
	static void Main()
}
```

Megjegyzés: eltávolítottam a *protected* (védett) tagokat a legtöbb függvényből, mert túl hosszúvá tették azokat a lap kitöltéséhez. Bármit, aminek az előző kódlistában nincs hozzáférési módosítója, feltételezhetően védeni kell.

Az első dolog, amit illene megjegyezned erről az osztályról, hogy örököl a *System.Windows.Forms.Form*-ból, amely egy osztály, amit szükséges használnod bármikor, amikor ablakos alkalmazást készítesz. Bármikor felnyitsz egy új Windows programot, akkor az ablakot, amit látsz, úgy hívják, hogy *form*, és a *Form* osztály nyújtja az összes funkcionalitást, ami szükséges egy *form*-mal való művelethez. Tehát bármikor létrehozol egy új ablakos programot, a *Form* alap osztályból kell származtatnod.

A *MainClass* tartalmaz egy Direct3D eszközt, *device* néven. Egy későbbi fejezetben kitérek majd a Direct3D eszközökre.

A konstruktor

Osztályod konstruktora a következő formában áll fel:

```
public MainClass()
{
this.ClientSize = new System.Drawing.Size(640, 480);
this.Text = “Direct3D Project”;
}
```

A konstruktor beállítja az ablak méretét 640x480-ra és aztán beállítja az ablak címét *Direct3D Project*-re. Nyilván a későbbiek során ezeket az értékeket meg szándékozhatsz változtatni.

A grafika inicializálása

A következő függvény inicializálja a grafikus motort. Én csak egy rövid áttekintést szándékozom adni itt a kódról, aztán később egy bővebb elmélyülést ebben a „Direct3D” fejezetben.

```
public bool InitializeGraphics()
{
        try {
        PresentParameters presentParams = new PresentParameters();
        presentParams.Windowed = true;
        presentParams.SwapEffect = SwapEffect.Discard;
```

```
        device = new Device(0, DeviceType.Hardware, this,
                CreateFlags.SoftwareVertexProcessing, presentParams);
```

Az előző kód létrehoz egy új grafikus eszközt. Túl sok információt még nem hordoz számodra, de ne aggódj sokat emiatt. Csak tudd, hogy működik, és egy későbbi fejezetben részletesebben lesz róla szó.

A kód következő része beállítja az eseménykezelőket (amelyek delegáltak):

```
device.DeviceLost += new EventHandler(this.InvalidateDeviceObjects);
device.DeviceReset += new EventHandler(this.RestoreDeviceObjects);
device.Disposing += new EventHandler(this.DeleteDeviceObjects);
device.DeviceResizing += new CancelEventHandler(this.EnvironmentResizing);
```

Az első három esemény azt kezeli, hogy vajon az eszköz nem észlelhető (mondjuk mert a felhasználó másik ablakra kapcsolt), hogy az eszköz visszaállt valami okból, vagy az eszközt törölték. Ez a három esemény szilárd esemény; amikor megtörténnek, akkor kezelned kell őket. Nincsenek ha-k, és-ek vagy hanem-ek erről – az operációs rendszer közli veled, hogy valami történt, és az eseménykezelődnek törődnie kell a helyzettel.

Az utolsó esemény azt kezeli, mikor a grafikus eszköz átméreteződik, amely egy különleges fajtájú esemény, mert lehet törölni. Egy törölhető esemény az egy olyan esemény, amelyről a programod dönthet úgy, hogy elutasítja. Például ha a felhasználó át szándékozik méretezni a játékod ablakát, a programod venni fogja az eseményt, de mondhatod azt az operációs rendszernek: „Nem, ez nem fog megtörténni!”, és az esemény nem hajtódik végre. Ez a viselkedés legtöbbször form-ok bezárásának megelőzésére használt, mielőtt a felhasználó elmenti az adatait a Windows alkalmazásokban.

A kód utolsó része visszaad egy *true* értéket a sikeres inicializáció esetén, vagy kaphat bármilyen *DirectXException*-t és ezesetben visszaad egy *false* értéket:

```
        return true;
        } catch (DirectXException) {
        return false;
        }
}
```

Az eseménykezelők

Négy eseménykezelő van a négy esemény kezelésére, amit láttál az előzőekben:

```
virtual void InvalidateDeviceObjects(object sender, EventArgs e)
virtual void RestoreDeviceObjects(object sender, EventArgs e)
virtual void DeleteDeviceObjects(object sender, EventArgs e)
virtual void EnvironmentResizing(object sender, CancelEventArgs e)
```

Ezek mindegyike védett és egyikük sem tartalmaz kódot. A kód valami olyasmi, amit feltételezhetően magadtól helyezel el bennük attól függően, ahogy a programot akarod reagáltatni. Minden függvény első paramétere a *sender*, az objektum ami küldi az eseményt a kezelőnek. Ezesetben a küldő lehet a Direct3D *device* változó az üzenetek küldésére, de soha nem jó ötlet, hogy ezt tegye.

A második paraméter mindegyiknél az esemény argumentumok (*EventArgs*), amely információkat ír le az eseményről neked. Ebbe részletesebben egy későbbi fejezetben megyünk bele.

Játék feldolgozás

A legtöbb játék keretrendszer nagyon hasonló; vannak elkülönített függvényei a játék logika végrehajtására, és aztán kirajzolják a pillanatnyi helyszínt a felhasználónak. Ez a keretrendszer sem kivétel, és ezt a két függvényt nyújtja neked:

```
protected virtual void FrameMove()
{
        // TODO : Frame movement
}
```

A *FrameMove* függvény teljesen csupasz; neked magadnak kell kitöltened. Ez az a függvény, ahová elhelyeznéd az összes aktuális játék logika számításokat, MI (mesterséges intelligencia) számítást, fizikai modellezést, hálózati kommunikációt és így tovább.

```
protected virtual void Render()
{
	if (device != null) {
		device.Clear(ClearFlags.Target, Color.Blue, 1.0f, 0);
		device.BeginScene();
		// TODO : Scene rendering
		device.EndScene();
		device.Present();
	}
}
```

A következő függvény rendereli a helyszínt. Megbizonyosodik, hogy jelenleg egy Direct3D eszköz van-e létrehozva, és ha igen, kékre törli a képernyőt és közli az eszközzel, hogy szándékozik elkezdeni rajzolni egy helyszínt. (Ahogy említettem előzőleg, a grafikai anyagba egy későbbi fejezetben megyek bele.) Aztán a függvény végez a helyszín rajzolásával, és közli az eszközzel, hogy mutassa a helyszínt a nézőnek.

A következő a *Run* függvény, amely akkor hívódik, mikor valaki futtatni akarja a játékot:

```
public void Run()
{
	while (Created) {
		FrameMove();
		Render();
		Application.DoEvents();
	}
}
```

Ez csak egy egyszerű ciklus, amely tartja az ismétlést, amíg a *form* „létrehozódik”. A *Created* az egy Boolean tulajdonsága a *form*-nak, amely *true* marad amíg a *form* érvényes, és *false* lesz, amint a *form* bezáródik. Minden ciklusban a függvény végrehajtja a játék logikáját, rendereli a helyszínt, és elmondja az alkalmazásodnak, hogy kezelje az összes eseményt.

Megjegyzés: az *Application* osztály az egy statikus osztály, amely képviseli az alkalmazásodat és hasznos függvényeket nyújt, mint például közli a programoddal, hogy kezelje az eseményeket vagy azonnal záródjon be.

Ablak esemény kezelők

Minden *form*-nak vannak beépített esemény kezelői az alapvető ablak események kezelésére, mint egy gombnyomás és a helyszín újrafestése. Mellékesen, a keretrendszernek is vannak függvényei ezen események kezelésére.

Illene megjegyezned, hogy nem szükséges hozzáadnod ezeket az alap eseményeket bármely esemény delegálthoz bárhová – a Windows önműködően tudja, hogy egy cselekvés megtörténtekor hívja ezeket az eseményeket.

Megjegyzés: a *Form* osztálynak már van beépített eseménykezelő függvényei, de alapértelmezésben rendszerint ezek nem csinálnak semmit, amely amiért muszáj létrehoznod a sajátodat és meghatároznod őket mint *override*-okat. Bár időnként végre fognak hajtani extra függvényeket, így emlékezned kell hívni az alap változatát a sajátodnak, ha akarod használni az extra alkalmasságot. Nézd meg egy kicsit később az *OnKeyPress* példát, hogy lásd, miről beszéltem.

```
protected override void OnPaint(PaintEventArgs e)
{
	this.Render();
}
```

Az *OnPaint* esemény olyankor történik, amikor a program átfestést igényel, például mikor egy ablak van a form-od tetején és elmozdult, stb. A keretrendszer egyszerűen csak hívja a renderelőt, hogy újrarenderelje a helyszínt.

A következőben van egy gombnyomás eseményed:

```
protected override void OnKeyPress(KeyPressEventArgs e)
{
	base.OnKeyPress(e);
	if ((int)e.KeyChar == (int)System.Windows.Forms.Keys.Escape) {
		this.Close();
	}
}
```

Az első dolog, amit ez a függvény csinál, hogy hívja az alap gombnyomás függvényt, a *Form.OnKeyPress*-t. Ez azért van, mert a *Form.OnKeyPress*

függvény valójában néhány hasznos számítást végez neked. Például, ha van akármilyen GUI összetevőd a képernyőn, akkor a Tab gomb lenyomása azt kellene, hogy okozza, hogy az ablak fókusza a következő GUI összetevőre megy. A *Form.OnKeyPress* függvény kezeli ezt neked, és kételkedem abban, hogy újra végre akarod hajtani ezt a funkcionalitást a magad módján.

Aztán a *Form.OnKeyPress* ellenőrzi, hogy lenyomtad-e az Escape gombot, és ha igen, akkor a form azt mondja, hogy bezárul.

Megjegyzés: ez később meg fog változni. Rossz ötlet, ha van egy játék, amin belül ha Escape-et nyomnak, akkor automatikusan bezárul.

A belépési pont

A keretrendszer utolsó része a belépési pont:

```
static void Main()
{
        using (MainClass mainClass = new MainClass()) {
                if (!mainClass.InitializeGraphics()) {
                        MessageBox.Show("Error while initializing Direct3D");
                        return;
                }
                mainClass.Show();
                mainClass.Run();
                }
}
```

Ez egyszerűen létrehoz egy új *MainClass* objektumot (az ablakod) és megpróbálja inicializálni azt. Ha ezt nem lehet, akkor egy hibaüzenet jelenik meg a képernyőn és az alkalmazás kilép.

Megjegyzés: ez a példa a *using* kulcsszót olyan módon használja, amit még nem láttál ezelőtt. A kód létrehoz egy using-blokkot, melyben a *mainClass* változó érvényes. Amint a blokk véget ér, a C# azonnal megsemmisíti a *mainClass* változót a szemétgyűjtőre való várakozás helyett.

Ha az ablak sikeresen létrejött, akkor a *Main* függvény közli az ablakkal, hogy mutassa magát, és aztán hívja a *Run* függvényt. Ha betöltöd és futtatod ezt a projektet, valami üres képernyőt fogsz látni, mert még semmi sincs renderelve.

Escape lenyomására kiléphetsz a programból.

Visual C#

A Visual C# a Microsoft által készített nagyon népszerű C# IDE; megtalálható a Visual Studio csomagban vagy önálló termékként. Sokkal összetettebb, mint a SharpDevelop, és míg ez utóbbi ingyenes, a VC# sok pénzbe kerül.

Új projekt létrehozásához a Visual C#-ban nyisd meg a File menüt és válaszd a New Project-et. Felbukkan a VC# projekt varázslója és felkínál neked egy csomó lehetőséget. Mivel egy üres C# projektet akarsz létrehozni, válaszd az Empty Project ikont.

Miután mindezzel készen vagy, az F7 gomb lenyomásával fordíthatod, F5-tel pedig futtathatod a programot.

A Visual C# D3D keretrendszere

Használhatod a Visual C# alkalmazás varázslóját, hogy létrehozz egy eltérő D3D keretrendszert, de én nem szándékozom ezt tenni ebben a könyvben. Ha már használtad valaha, akkor tudni fogod, miért. A Visual C# által létrehozott keretrendszer hihetetlenül összetett és számos fejezetet venne igénybe a kifejtése; én a dolgokat szépnek és egyszerűnek szándékozom tartani.

Megjegyzés: a VC# D3D keretrendszer létrehozó nekem teljes 180 kilobájtnyi kódot hozott létre. Ez sok kód.
Megjegyzés: azt is tudnod illene, hogy a DirectX SDK legutóbbi változata (a 2004 nyári kiadás, ez írás szerint – amit ez a könyv nem használ) eltávolítja a Visual C#-ból a D3D Keretrendszer Varázslót. Helyette találhatsz egy vadonatúj keretrendszert, már kódolva neked.

A jó dolog az, hogy a keretrendszer, amellyel a SharpDevelop tökéletesen működik, független attól, hogy VC#-ot vagy SharpDevelop-ot használsz.

A fejlettebb keretrendszer

A SharpDevelop által nyújtott keretrendszer megfelelő egy generikus keretrendszernek, de nem igazán találkozik a szükségleteiddel, mint játékprogramozó. Más részről pedig a VC# keretrendszer túlzás. Én megalkottam egy másik keretrendszert, mely a SharpDevelop keretrendszerére épült. Ennek összes kódja megtalálható e könyv CD mellékletének Demo 6.3 mappájában.

Van időd?

Majdnem minden játék idő alapú. Ritka az, amely nem igényli némi használatát egy időzítő eszköznek. Rossz hír a játékprogramozók számára, hogy a .NET nem igazán volt velünk tervezve gondolatban. A .NET keretrendszer inkább alkalmazások programozására volt tervezve elgondolás szerint, így csak egy nem túl pontos időzítőrendszert tartalmaz. Különben is, mikor láttál utoljára egy szabályos szövegszerkesztő alkalmazást ezredmásodperc-pontosságú időzítőt igényelve?

Szerencsére neked és nekem, a DirectX csapat a Microsoftnál elhatározta, hogy kijavítja ezt a problémát, és adott nekünk egy ügyes, nagy pontosságú időzítőt, amire épelméjű játékprogramozónak nem is lenne szüksége. Mikroszekundumos pontosságról beszélünk itt. Bár én boldog vagyok csak az ezredmásodpercekkel is.

Megjegyzés: az eredeti Pentiumtól kezdve az összes Intel-kompatibilis CPU-nak van egy *teljesítményszámláló*ja, mely az, amit a Windows használ, hogy hozzájusson ehhez az ultra nagy pontosságú időzítéshez.

A Microsoft DXUtil.cs néven nyújtja ezt az állományt, ami tartalmazza ezt az időzítőt, de ez valami elrejtett az SDK-ban. Megtalálhatod, ha bemész a *Samples\C#\Common* mappába, ahová telepítetted a DirectX 9 SDK-t.

Itt egy példa, hogyan használd az időzítőt:

```
float time = DXUtil.Timer( DirectXTimer.GetAbsoluteTime );
```

A *DXUtil* osztály statikus, és a *Timer* függvény visszaad egy adott időt, attól függően, hogy milyen értékek vannak beadva. Az értékek mindegyike egy *DirectXTimer* nevű felsorolás része.

Az érvényes értékek listája a következő:

Érték	Eredmény
Reset	Visszaállítja az időzítő alkalmazás idő értékét 0-ra
Start	Elindítja az időzítőt, ha az megállt. Csak alkalmazásidőt érint.
Stop	Megállítja az időzítő alkalmazásidejét az előrehaladásból
Advance	0.1 másodperccel előremozgatja az időzítő alkalmazásidejét
GetAbsoluteTime	Az abszolút rendszeridőt adja (rendszerint amióta bekapcsolt a számítógép, de nem garantált)
GetApplicationTime	Az alkalmazási idő értéket adja
GetElapsedTime	Az eltöltött idő utolsó visszakeresése óta eltelt idő

Úgyhogy szép mennyisége van az elérhető lehetőségeknek. Megjegyezheted, hogy az idő float-ként (lebegőpontos) van jelképezve; ez valójában nagyon praktikus. A float egysége a másodperc, ami azt jelenti, hogy 1.0 jelképez egy teljes másodpercet, és 0.001 jelképez egy ezredmásodpercet és így tovább. Az ok, amiért ez legtöbbször praktikus, hogy „egység per másodperc” alakban jelképezel sebességet és más idő alapú értéket a játékokban. Például egy rakéta mozoghat 10 láb per másodpercnyi sebességgel, ezért hogy megkapd az adott időzítőhöz tartozó sebességmennyiséget, csak szorozd meg az időzítő eredményét 10-zel.

A DirectX időzítő két különböző időt tart nyilván, az abszolút időt és az alkalmazás időt. Az abszolút idő rendszerint azt jelképezi, hogy a Windows operációs rendszer az utolsó újraindítása óta mennyi ideje fut (de ez nem garantált), míg az alkalmazás időt módosíthatod.

Hadd mutassam meg neked, hogyan játszhatsz egy kicsit az időzítővel:

```
// megkapni az abszolút időt:
float t;
t = DXUtil.Timer( DirectXTimer.GetAbsoluteTime );
t = DXUtil.Timer( DirectXTimer.GetApplicationTime );
```

A két hívás után a *t* körülbelül 68.000 másodperc volt mindkét hívásnál, csak néhány ezredmásodpercben különböztek. Ez körülbelül 19 óra, amely véletlenül az utolsó újraindítás óta eltelt idő.

Mikor először indítod a programodat, az alkalmazás időzítőnek ugyanaz az értéke, mint az abszolút időzítőé, ami nem igazán hasznos. Hogy hasznossá tedd, ahhoz újra be kell állítanod:

```
DXUtil.Timer( DirectXTimer.Reset );
t = DXUtil.Timer( DirectXTimer.GetApplicationTime );
```

Most a *t* azon másodpercek számát fogja tartani, amik elteltek, mióta az alkalmazás időzítő újraindult (valószínűleg jóval kisebb, mint nulla; az én rendszeremen a kapott eredmény 0.000000838095332, ami azt jelenti, hogy 0.84 mikroszekundum telt el a két hívás között).

Meg is állíthatod az időzítőt:

```
DXUtil.Timer( DirectXTimer.Reset );
DXUtil.Timer( DirectXTimer.Stop );
f = DXUtil.Timer( DirectXTimer.GetApplicationTime );
f = DXUtil.Timer( DirectXTimer.GetApplicationTime );
DXUtil.Timer( DirectXTimer.Start );
f = DXUtil.Timer( DirectXTimer.GetApplicationTime );
```

Az *f* által a gépemen először visszanyert eredmény 0.00000139482561 másodperc volt. A nagyon következő sor végrehajtja a hívást újra, és az eredmény ugyanaz: 0.00000139482561 másodperc. Ez azt jelenti, hogy 1.39 mikroszekundum telt el az időből, ahol az időzítő beállítódott a megállításakori időre, és nyilvánvalóan az időzítő nem haladt, mikor leállt.

Miután az időzítő elindult újra a gépemen, az *f* ki volt töltve 0.00000391111143 másodperccel, vagy 3.9 mikroszekundummal.

Végül az időzítő könnyedén visszanyeri a hívások közt eltelt időt:

```
DXUtil.Timer( DirectXTimer.GetElapsedTime );
f = DXUtil.Timer( DirectXTimer.GetElapsedTime );
```

Az első hívás egyszerűen újraállítja az eltelt idő számlálót. A következő hívás adja nekem, hogy mennyi idő telt el az utolsó alkalom óta, mikor megkaptam az eltelt idő értéket. Esetemben ez 0.000000838095332 másodperc vagy 0.84 mikroszekundum.

Gondok az időzítővel

Én nem szeretem a DirectX alkalmazás időzítőt egy okból: statikus, ami azt jelenti, hogy csak egy időzítő van. Nem nagyon hasznos, ha csak egy nagy felbontású időzítő van az egész programodban, mert valószínűleg sok dolog van, amit időzíteni kell, nem csak egy. Tehát előrementem és megcsináltam az ésszerű dolgot: létrehoztam a saját, bővíthető időzítő osztályomat, melyet megtalálsz a haladó keretrendszerrel (Advanced Framework) a Demo 6.3 mappában a *Timer.cs* állományban. Ez az időzítő osztály a létező DirectX időzítő tetejére épült, de lehetővé teszi, hogy létrehozz sok különböző időzítőt, melyek mindegyike különböző órákon fut. A kód nagy része elég egyszerű és a létező DirectX kód tetejére épült, úgyhogy én csak az *AdvancedFramework.Timer* osztály vázlatát szándékozom neked bemutatni és azt, hogy hogyan kell

használni ahelyett, hogy értékes időt vesztegetnénk hülye időzítő kód dokumentálására.

Itt az osztály vázlat:

```
public class Timer
{
	public static float Now()
	public Timer()
	public void Reset()
	public void Reset( float p_time )
	public float Time()
	public float Elapsed()
	public void Pause()
	public void Unpause()
}
```

A *Now* függvény egy egyszerű csomagolás az abszolút időfüggvény körül a DirectX időzítőn belül. Ez egy statikus függvény, tehát nem szükséges bármilyen tényleges időzítőket hívni:

```
float f = AdvancedFramework.Timer.Now();
```

Van két *Reset* függvény, lehetővé téve számodra, hogy visszaállítsd az időzítőt 0-ra vagy amilyen időre akarod:

```
AdvancedFramework.Timer t = new AdvancedFramework.Timer();
t.Reset(); // 0-ra
t.Reset( 60.0f ); // 60 másodpercre
```

Aztán használhatod az időzítőt, hogy megkapd az utolsó resetelés óta eltelt időmennyiséget:

```
f = t.Time(); // utolsó reset óta eltelt időmennyiség
```

vagy megkapni az eltelt időmennyiséget az utolsó hívása óta *Elapsed*-nek:

```
t.Elapsed(); // eltelt idő időzítő resetelése
f = t.Elapsed(); // utolsó hívás óta eltelt idő
```

És, természetesen, szüneteltetheted vagy folytathatod a futtatását az időzítőnek:

```
t.Pause(); // időzítő szüneteltetése
// ennek az időzítőnek többé nem halad az idő
t.Unpause(); // időzítő folytatása
```

Ez az időzítő sokkal könnyebben használható, mint a DirectX időzítő.

A keretrendszer változásai

Ahogy korábban mondtam, a Fejlettebb keretrendszer a SharpDevelop keretrendszer tetejére épült, sok apró változtatással, amelyek megkönnyítik a használatát és még játék központúbbá teszik.

Névterek

Itt egy példa, hogy az új keretrendszer hogyan teszi tisztábbá a dolgokat: a *Microsoft.DirectX.Direct3D* névtér többé már nem használt hallgatólagosan a *using* parancson keresztül. Ez a dolgokat kicsit zavarossá teheti. Az összes fő DirectX összetevőnek vannak osztályai, ezeket hívják *Device*-eknek (Eszközök), mint pl. *DirectSound.Device* és *DirectInput.Device*. Ezért rossz ötlet, ha ilyesmi kódod van:

```
using Microsoft.DirectX.Direct3D;
using Microsoft.DirectX.DirectSound;
// később:
Device d = null; // HIBA: milyen fajta eszköz? Direct3D vagy DirectSound?
```

Így a fejlettebb keretrendszer, használva a névterek másnevesítését, létrehoz három új névteret:

```
using Direct3D = Microsoft.DirectX.Direct3D;
using DirectSound = Microsoft.DirectX.DirectSound;
using DirectInput = Microsoft.DirectX.DirectInput;
```

Most beszélhetsz csak *Direct3D.Device*-ról vagy *DirectSound.Device*-ról, és a programod sokkal olvashatóbb lesz.

A játék osztály

A Fejlettebb Keretrendszer a fő osztályát *Game*-nek hívja. Ez nem igazán egy fontos változtatás (a régi keretrendszer *MainClass*-nak hívta), de úgy éreztem, hogy a *MainClass* hülyén hangzik.

A névváltoztatás mentén, hozzáadtam egy csomó statikus változót a keretrendszer tetejének közelébe, így, ha szükséges, könnyen megváltoztathatod azokat:

```
static string gametitle = “Advanced Framework”;
static int screenwidth = 640;
static int screenheight = 480;
```

Ha újrahívod, ezek az értékek elrejtettek voltak a *MainClass* konstruktoron belül az első keretrendszerben, így le kellett vadásznod azokat a függvényeket, ha meg akartad őket változtatni.

Néhány más statikus is hozzá lett adva:

```
static Timer gametimer = null;
static bool paused = false;

Direct3D.Device graphics = null;
DirectSound.Device sound = null;
DirectInput.Device keyboard = null;
DirectInput.Device mouse = null;
DirectInput.Device gameinput = null;
```

Egy játék időzítő van definiálva, ami nyomon követi a jelenlegi játékidőt neked, és egy *paused* változó is meg van adva, ami meghatározza, hogy a játék szünetel vagy sem. Végül öt eszköz is definiálva van: a grafikus eszköz; a hang eszköz; és három adatbeviteli eszköz, ami jelképez egy billentyűzetet, egy egeret és bármilyen más játék adatbeviteli eszközt (rendszerint egy botkormányt (joystick) vagy egy játékvezérlőt (gamepad)), ami használható.

Függvény változások

A régi keretrendszer csak a Direct3D rendszert inicializálta az *InitializeGraphics* függvény használatával. Ez az új keretrendszer egy lépéssel távolabb viszi és hozzáad két függvényt: az *InitializeSound*-ot és az *InitializeInput*-ot. Kapsz egy ingyen süteményt, ha kitalálod, hogy mit csinálnak. Nem szándékozom itt közétenni a kódjukat, minthogy most még nem is értenéd – ezt későbbre teszem egy másik fejezetre.

A régi *FrameMove* függvénynek megváltozott a neve; most *ProcessFrame*-nek hívják. Úgy gondolom, ez jobban hangzik és leíróbb arról, amit tulajdonképpen csinál.

```
protected virtual void ProcessFrame()
{
        if( !paused )
        {
                // csinálni feldolgozást itt
        }
        else
                System.Threading.Thread.Sleep( 1 );
}
```

A *ProcessFrame* függvénynek ezidejűleg van egy kis kiegészítése: ellenőrzi, hogy a játék szünetel vagy nem. Ha a játék nem szünetel, akkor mehetsz előre és végrehajthatod a játék számításaidat. Viszont ha a játék szünetel, akkor a függvény közli a rendszerszállal, hogy egy milliszekundumot várjon.

Tipp: van itt valami, amit a legtöbb játékprogramozó hajlamos elfelejteni: nem vagy egy kizárólagos tulajdonosa egy számítógép rendszernek. Napjaink korszerű számítógépein nem ritka, hogy valaki számítások ezreit futtatja azonos időben, és ezért szükséges osztozkodnod az összes többi folyamattal. Mikor azt mondod a jelenlegi szálnak, hogy várjon, akkor azt mondod az operációs rendszernek, hogy „Nem igazán van most bármilyen feldolgozási folyamatom, ezért menj, és dolgozz valami máson, ami szükséges lehet.” Ha ez a sor nincs benne a programodban, akkor a játékod felemészt olyan sok processzor erőforrást, amennyit meg tud ragadni, még akkor is, ha szünetel. Ez egy nagyon rossz dolog.

A *Render* és *Run* függvények nem változtak meg szignifikánsan, ezért kihagyom itt a leírásukat.

Események

Egy dolog, amire figyelned kell egy ablakos játékban, elveszíti a hangsúlyt. Egy többfeladatos operációs rendszerben, mint a Windows, a felhasználó nagyon könnyen dönthet úgy, hogy leállítja a játékoddal való játszást, és valamiért ellenőriz egy másik ablakot. Természetesen a játékod nem lehet önző; mondania kellene, hogy „Rendben, felhasználó, menj csak előre és használj másik programot. Én csak ülök itt *hüpp* és várok a visszatérésedre.” Mikor a programod nem lesz a középpontban (ami azt jelenti, hogy a felhasználó átváltott egy másik programra), az *OnLoseFocus* esemény végrehajtódik:

```
protected override void OnLostFocus( EventArgs e )
{
        base.OnLostFocus( e );
        Paused = true;
}
```

Elmondod az alap Form osztálynak, hogy a program nem hangsúlyos, és aztán beállítja a Paused tulajdonságot *true*-ra (igazra), elmondva a játéknak, hogy szüneteljen. Részletezem ezt a tulajdonságot a következő alfejezetben.

Megjegyzés: úgy tűnhet, hogy jó ötlet a játék folytatására, mikor visszanyeri a fókuszt (az *OnGotFocus* eseménnyel), de igazán nem az a következő okok miatt: mi van, ha a játékos kézzel szünetelteti a játékot, aztán kilép és visszalép? A játéknak önműködően folytatódnia kellene? Valószínűleg nem. Ezért biztonságosabb a játékot szüneteltetve hagyni, hogy aztán a játékos maga kapcsolja vissza a futását.

A szünet tulajdonság

A fejlettebb keretrendszernek van egy *Paused* nevű tulajdonsága, mely a *paused* logikai változót fedi be:

```
public bool Paused
{
        get { return paused; }
        set
        {
                // a játék szünetel
                if( value == true && paused == false )
                {
                        gametimer.Pause();
                        paused = true;
```

```
                }
                // a játék folytatódik
                if( value == false && paused == true )
                {
                        gametimer.Unpause();
                        paused = false;
                }
        }
}
```

A kód meglehetősen egyszerű. A *get* esemény egyszerűen visszatér a *paused* változó értékével. A *set* esemény egy kicsit összetettebb; végrehajtja a tulajdonképpeni szüneteltetését és folytatását a játéknak. Ha a *Paused*-et igazra (true) állítod, akkor a játék ellenőrzi, hogy nem szünetel-e már (ha igen, a függvény nem csinál semmit). Ha nem, akkor a függvény szünetelteti a játék időzítőt és a *paused* változót *true*-ra (igaz) állítja. A folytatás az ellenkezőjét teszi; folytatja az időzítő futtatását és a *paused*-et hamisra (false) állítja.

A belépő pont

A keretrendszer belépő pontja is megváltozott egy kicsit a fejlettebb keretrendszerben:

```
static void Main()
{
        try
        {
                Game game = new Game();
                game.InitializeGraphics();
                game.InitializeSound();
                game.InitializeInput();
                game.Show();
                game.Run();
        }
        catch( Exception e )
        {
                MessageBox.Show( “Error: “ + e.Message );
        }
}
```

A játék létrejött, és a három fő média összetevő inicializálódott, mutatódik a játék helyszín, és futni kezd. Végül van egy kivétel blokk, amely kezel bármilyen

kivételt, és kiírja a hibaüzenetet a felhasználónak. Ez egy kis eltérés a régi keretrendszertől; az logikai visszatérési értéket használt az inicializáló függvényekben annak eldöntésére, hogy volt-e hiba. Itt én átalakítottam teljesen egy kivétel-alapú rendszerré.

Összegzés

Most van egy erős keretrendszered, amellyel elkezdheted tervezni a játékaidat. Sajnos, mivel még nem tudod, hogyan kell használni a DirectX-et, ezért folytatnod kell az olvasást! Ó, a fenébe!

A jó hír az, hogy a következő fejezetek megmagyarázzák az összes zavaró DirectX hókusz-pókuszt, amit láttál a keretrendszereken belül. Bár én nem aggódnék nagyon. A DirectX 9 kezelése hatalmasat javult a régebbi változatoktól, és hihetetlenül könnyebb használni, mint az előzőeket.

7. fejezet

Direct3D

A Direct3D az egyik létező legösszetettebb grafikus API-vá fejlődött. Neked és nekem szerencsére, csodálatosan rugalmas is. A Direct3D nagyon hosszú úton jött a hajdani napok óta, és bár minden kiadás még összetettebb, a Microsoftnak sikerült minden kiadást még egyszerűbben kezelhetővé tennie. Nem tudom megmondani, hányszor hallottam az embereket panaszkodni a kódsorok számáról csak egy egyszerű renderelő eszköz felállítása során, mikor még a DirectX 5 volt a király. Neked nem kell ezzel törődnöd, mert jött a DirectX 9 megmenteni a napot.

DirectX változatok

E könyv írása alatt a Microsoft bemutatta a DirectX SDK egy új verzióját. A legújabb változat ez írás szerint a DirectX9.0c, habár ez a könyv a DirectX9.0b-t használja.

Sajnos ha neked a legújabb verzió van telepítve, akkor a kódpéldák némelyike nem fordítódik le, mivel a Direct3D D3DX könyvtárak megváltoztak néhány kulcsfontosságú helyen.

Egy eszköz mindent szabályozni

A Direct3D mag összetevője a *Direct3D.Device*. Egy Direct3D eszköz egyszerűen jelképez bármilyen eszközt a rendereléshez használva, amely majdnem mindig a videókártyád.

Megjegyzés: a DirectX homályos kifejezéseket használ, mint „device” (eszköz), mert a jövőben lehet, hogy nem lesz olyan dolog, mint egy videókártya, és lehet, hogy valami teljesen más dolgot fogsz használni.

Minden a bemutatóról

A Direct3D eszközök csodálatosan konfigurálhatók. Emiatt a Microsoft létrehozott egy speciális osztályt, ami lehetővé teszi, hogy pontosan leírd, hogy hogyan akarod konfigurálni a grafikus eszközödet. Ezt az osztályt hívják *Direct3D.PresentParameters*-nek. Teljes tulajdonságai vannak, olyan sok, hogy nincs elég helyem részletezni ezeket, ezért csak egy táblázatot mutatok azokról, amelyeket ez a könyv érint:

Tulajdonság	Leírás
Windowed	A program ablakos vagy nem
SwapEffect	Elmondja a Direct3D-nek, hogyan cserélje fel a back buffer-eket
BackBufferCount	Elmondja a Direct3D-nek, hogy mennyi back buffer-nek kellene lennie
BackBufferFormat	Elmondja a Direct3D-nek, hogy milyen formátumú back buffer-t akarsz használni
BackBufferHeight	Elmondja a Direct3D-nek, hogy mekkora ablakmagasságot akarsz használni
BackBufferWidth	Elmondja a Direct3D-nek, hogy mekkora ablakszélességet akarsz használni

Megjegyzés: ha érdekel a tanulás azokról a haladó tulajdonságokról, amiket én nem használok ebben a könyvben, akkor utána kell nézned egy Direct3D-ről szóló könyvnek, mint pl. Wendy Jones: Beginning DirectX 9 („Kezdő DirectX 9”) című kötete.

Pufferek és puffercsere

Ha nem tudsz semmit a grafikus rajzolásról, akkor ez a fejezet neked szól. A játék programozás régi napjaiban a grafikus hardver nagyon kezdetleges volt, és csak arra volt elegendő grafikus memóriád, hogy éppen mi van a képernyőn. A régi rendszerek rajzolnának egy helyszínt, törölnék a képernyőt és aztán rajzolnának egy másik helyszínt. Ez rossz villogó hatást eredményezett, mert a képernyő néhány milliszekundumra fekete színűvé vált, mielőtt az új adatokkal frissült. Nem sok játék készült így, mert a felhasználóknak fejfájást okoztak volna.

Megjegyzés: először az ún. *piszkos téglalapok* módszerét találták ki arra, hogy megoldják ezt a problémát. Mikor valami mozgott a képernyőn, a számítógép nyomon követte a terület téglalapját, ahol valami megváltozott, és aztán újra rajzolta a megváltozott téglalapot. Szerencsére nekünk ezt nem kell többé csinálnunk.

Ahogy a grafikus hardver még fejlettebbé vált, egy új módszer került bevezetésre, a p*uffer csere* vagy *back pufferolás*. Az ötlet az, hogy két puffert kell tartani, egy első, front puffert és egy hátsó, back puffert, nyitva minden időben. Te rajzolsz a back pufferre, és amikor ezt az adatot a képernyőn akarod mutatni, a back puffer felcserélődik a front pufferrel, és a régi front puffer válik a back pufferré.

Egy pontnál meg volt határozva, hogy a puffer cseréje nagyon sokáig tartott, és a játékok tulajdonképpen semmit nem csináltak, csak ott vártak a pufferek cseréjére, mielőtt el tudták indítani az írást a back pufferre újra. Ez különösen igaz volt azokban az időkben, amikor a memóriabuszok hihetetlenül lassúak voltak. A probléma megoldására a játékprogramozás úttörői elhatározták, hogy adnak egy másik puffert a keverékhez – létrehozva a *hármas pufferelés*t. Míg a back puffert kicserélték a front pufferre, a játék tulajdonképpen elkezdett írni egy harmadik pufferre ahelyett, hogy a puffercserére várt volna. Természetesen a memóriabuszok sebessége kezdett felzárkózni a játékok által használt felbontásokhoz, és többé már nem gyakran látod ezt a hármas pufferelést.

Megjegyzés: a DirectX támogatja a három back puffert, megadva neked a lehetőséget a négyszeres pufferelés használatához, ha szeretnéd. Bár személy szerint nem látok sok szükségességet ilyen pufferelésre, de kipróbálhatod, ha akarod.

A következő alfejezetek bemutatják a három különböző módot puffercsere végrehajtására DirectX 9-ben.

Másolás

Az első csere effektus a *Direct3D.SwapEffect.Copy*. Ez a formátum megköveteli, hogy a programodnak legyen pontosan egy back puffere. Minden időben, mikor bemutatod a jelenlegi helyszínt, a DirectX másol mindent a back pufferről a front pufferre. Természetesen ez sokáig tarthat. Gondold át: ha van egy játékod, amely ésszerű felbontáson fut, mondjuk 1280x960, és 32 bites színt használ,

akkor a képernyőd 4,68 megabájt helyet foglal. Egy 1600x1200-as képernyő már 7,32 megabájtot. Tehát mikor végrehajtod a másolást, megpróbálsz 4-8 megányi információt másolni át az elsődleges pufferre, körülbelül 30-60-szor másodpercenként. Az átviteli sebesség 120-ról 480 megára másodpercenként.

Flipping-ezés

A második csere effektus a *Direct3D.SwapEffect.Flip*. A flipping-ezés többszörös back puffereken működik, bármelyik puffer mutatódik is jelenleg a képernyőn. Ha az alkalmazásod teljes képernyős, akkor általában nem lesz semmilyen tényleges memóriamásolás folyamatban. Ha az alkalmazásod ablakos, akkor lesz memóriamásolás, mivel a programod csak egy részét ellenőrzi a képernyőnek.

Eldobás

Az utolsó csere effektus a *Direct3D.SwapEffect.Discard*.

A közelmúltban a videókártyák nagyon összetettek lettek; a gyártóik mindenféle módot kitaláltak, hogy a grafika gyorsabb legyen, mint a hagyományos módszerekkel. Néhányuk igazán csinos algoritmusokat alkotott, hogy csak akkor frissüljön a képernyő része, amikor tulajdonképpen megváltozott, és ők nem igazán mondják el nekünk, hogy hogyan csinálják. A végeredmény az lett, hogy a videókártya hardver sokkal kevesebb munkát végez, mint korábban és mindent önműködően csinál. Ezesetben használnád az eldobási módszert a puffereid felcserélésére.

Egy eszköz létrehozása

A konstruktor a *Direct3D.Device* osztály számára sajnos egy behemót. Öt paraméter van a konstruktornak, amelyek a következő táblázatban vannak listázva:

Paraméter	Leírás
int adapter	Az adapter azonosítója, amit használni akarsz
DeviceType deviceType	Az eszköz fajtája, amit létre akarsz hozni
Control renderWindow	Hivatkozás az ablakra, amit renderelni fogsz
CreateFlags behaviorFlags	Az eszköz viselkedését jelképező flag-ek
PresentParameters presentation Parameters	A bemutató paraméterek

Az adapter

A számítógépedben minden grafikus adapterhez hozzá van rendelve egy azonosító szám. A legtöbb számítógép rendszernek csak egy grafikus adaptere van, ez a fő videókártyád. A fő grafikus adapternek mindig a 0 van adva azonosítóként, tehát majdnem mindig ez az, amit a grafikus eszközöd konstruktorába szándékozol átadni.

Az eszköz típus

Lehetőséged van létrehozni számos különböző fajta grafikus eszközt. Általában két értékkel leszel érintett: *Direct3D.DeviceType.Software* és *Direct3D.DeviceType.Hardware*. Van egy harmadik érték, a *Direct3D.DeviceType.Reference*, de ezt csak hibakeresési célokra használják és soha nem kellene használni bármilyen tényleges kiadáshoz.

A szoftveres eszközök általában támogatják a legtöbb jellemzőt, de a CPU-dnak magának kell az összes számítást végrehajtania, mint a megvilágítás és a dolgok átalakítása.

A hardveres eszközök viszont tehermentesítik a nehéz grafikai számítástól a videókártyádat, mely párhuzamos processzorként működik, ezért felszabadítja a CPU-dat más fontos dolgokra, mint mesterséges intelligencia (MI) és fizikai számítások.

A hardveres eszközök meglehetősen ritkák voltak, de nagyjából feltételezheted, hogy a legtöbb számítógép manapság már támogatja az alapvető hardveres gyorsítást, és valószínűleg szándékozni fogsz egy hardver eszközt létrehozni.

Viselkedési flag-ek

Számos flag („zászló”) van, amelyek meghatározzák, hogy egy grafikus eszköz hogyan fog viselkedni, és ezek leginkább a csúcspont számításra vonatkoznak. A következő táblázat a számodra és nekem legfontosabb értékeket mutatja:

Érték	Jelentés
HardwareVertexProcessing	Grafikus hardvered törődni fog a csúcspont átalakításokkal
SoftwareVertexProcessing	A CPU-d törődni fog a csúcspont átalakításokkal
MixedVertexProcessing	Grafikus kártyád és CPU-d osztozni fognak a csúcspont átalakítási számításon

Egyéb lehetőségek is léteznek, de csak ezekkel lennél érintett, ha egy igazán összetett programot készítenél. Valószínűleg nem kell idegeskedned miattuk (nekem sosem kellett).

A hardveres csúcspont számítás egyike a fő forradalmi újdonságoknak a korszerű grafikus hardverben; lehetővé teszi, hogy tehermentesítsd a videókártyádat a nagyon nehéz grafikus számítástól, így a CPU olyan dolgokra összpontosíthat, mint a fizika vagy MI számítása. Az eredeti NVIDIA GeForce kártya volt az első, amely támogatta a hardveres csúcspont számítást, és olyan pontra jutott, ahol majdnem feltételezheted, hogy most már mindenki támogatja. Azért mondtam, hogy *majdnem*, mert vannak olyan naplopók, akik még nem frissítették a vidókártyájukat.

A menedzser

Előreszaladhatsz és készíthetsz olyan fajta eszközt, amilyet akarsz – a fordító nem fog megállítani. Mondhatod: „Hé, számítógép, adj nekem egy hardveres renderelőt teljes csúcspont számítással és egy nagy felbontással!” és a számítógéped azt mondja: „Rendben!” Talán az a kód még futni is fog a fejlesztési platformodon, amire életed megtakarítását költötted. Aztán odaadod a játékodat a kevésbé szerencsés barátaidnak, mondván nekik, hogy milyen fantasztikus… és ők azt mondják neked, hogy szívás, mert nem működik az ő számítógépeiken. Miért?

Nos, nem tételezheted fel, hogy mindenkinek a legkorszerűbb eszközei lesznek. Egy számítógépnek szó szerint milliónyi hardverváltozata lehet, és majdnem lehetetlen valamit működőre csinálni minden egyes számítógépre, hacsak vissza nem térsz 100 százalékban a szoftveres számítási módra.

Ellenőrizned kell, hogy vajon megvannak-e azok az eszközök, amelyek létrehozására készülsz. A jó emberek a Microsoftnál a praktikus *Direct3D.Manager* osztályt nyújtják, adva nekünk ezt a funkcionalitást. Mint minden DirectX dolognál, az osztály nagy és egy csomó függvénye van, amelyeket egyszer fogsz használni egy kék hold alatt, ezért csak azokat a részeit mutatom, amelyek fontosak neked és nekem.

Az eszköz meglétének ellenőrzése

Az első dolog, amit tenni akarsz a játékodban, annak ellenőrzése, hogy egy bizonyos fajta eszköz elérhető-e számodra. Ez végrehajtható a *Direct3D.Manager.CheckDeviceType* függvény használatával:

```
public static bool CheckDeviceType(
        int adapter,
        DeviceType checkType,
        Format displayFormat,
        Format backBufferFormat,
        bool windowed
);
```

Az *adapter* az adapter azonosítója, amelyről az előzőekben szóltam neked, és ami azt is jelenti, hogy valószínűleg 0 értékként fogod használni. A *checkType* jelképezi az eszköz fajtáját, amelyet ellenőrzöl, hogy vajon az adapter támogatja-e. A *displayFormat* és a *backBufferFormat* argumentumok határozzák meg, hogy milyen fajta back puffert és kijelző módot akarsz használni; erre még visszatérek egy pillanat múlva. Végül az utolsó paraméter meghatározza, hogy ablakos módot akarsz-e használni vagy sem.

Kijelző formátumok

A kijelző formátumok trükkös dolgok, mert olyan sok különböző formátum van olyan sok különböző videókártyán. Így a Direct3D nyújtja ezt a szép *Direct3D.Format* felsorolást, amely listáz minden ismert lehetséges kijelző formátumot. Oké, talán nem minden formátumot, de azok tonnáit. Én 46-ot számoltam a dokumentációba belenézve. Azta!

Megjegyzés: a régi időkben voltak szabványos formátumaink, mint az EGA és VGA, de hé – kinek kellenek már szabványok? Uhhh. Ez az egyik olyan dolog, amit hiányolok a régi időkből, de te sose bánd.

Tehát mi is az a kijelző formátum? Alapvetően egy kijelző formátum elmondja neked, hogy egy képernyőn hogyan van jelképezve minden képpont. A nagyon első kijelző formátumok monokrómok voltak, jelezvén, hogy minden pixel üres volt vagy egy színű (rendszerint fehér, a monitortól függően) és minden képpontot egy bit reprezentált. Később a számítógépek még összetettebbekké

váltak és 2 (négy szín) vagy 4 (16 szín) bitet tudtak használni egy színinformáció tárolására. A következő forradalmi lépés a palettázott szín volt, melyben egy 8 bites pixel formátumod volt (256 szín), ahol minden egyes színérték egy paletta színére mutatott.

Ezek egyike sem igazán fontos már többé, mert a világ továbblépett (harsonák, kérem) a true color felé.

A true color megajándékozta a játékprogramozókat azzal a képességgel, hogy buja világokat mutassanak, tele élénk színekkel. Az első időben a kijelző formátumok egy időben 16 ezer vagy 24 millió különböző szín mutatására voltak képesek.

Egy true color képpont 16 vagy 32 adatbiten van ábrázolva, és R5G6B5 (16 bit), X8R8G8B8 (32 bit) vagy valami hasonló formátumban tárolódnak. Az R5G6B5 esetén azt jelenti, hogy a képpont öt bit vörös információn tárolódik (32 érték), hat bit zöld információn (64 érték) és öt bit kék információn (32 érték). Az aktuális pixelszín az által a szín által van meghatározva, amely azon három összetevő kombinálásakor van létrehozva. Például ha volt egy 16-bites pixeled, aminek a (31,63,31) értékei voltak, akkor egy fehér képpontot fogsz kapni, mivel az összes szín a teljes erősségén van, és mikor kombinálod azokat, akkor fehéret kapsz. Hasonlóképpen, ha (0,0,0)-d volt, akkor feketét kapnál, és ha (31,31,0), akkor pedig narancsszínt (teljes vörös, félig zöld, nincs kék).

Vannak 16-bites formátumok is, mint az X1R5G5B5, amelyek öt bitet használnak minden színhez, és az utolsó bit nem használt.

A 32-bites formátumok inkább egy kicsit formalizáltak. Főképpen két változatuk van: X8R8G8B8 és A8R8G8B8. Az első alakban az X azt jelenti, hogy nyolc bit figyelmen kívül van hagyva és egyáltalán nem használt. A második alakban az A az *alfa*-t támogatja, mely egy további nyolc bitje az adatoknak, ami pixelenként tárolható és rendszerint az áttetszőségi hatást jelképezi. Egy képpont 0 alfával teljesen áttetsző, 255 alfa a teljesen színezett, és 127 alfa jelenti, hogy keverve van 50 százaléknyi áttetszőséggel az alatta lévő pixellel. Az alfa értékekre később egy kicsit részletesebben visszatérek.

Valószínűleg 32-bites színformátumot akarsz majd használni. Néhány évvel ezelőtt nagy teljesítménybeli különbség volt a 16- és 32-bites színek között, de

többé ez már nem igazán probléma. Egyedül akkor kell jól tájékozottnak lenned a 16-bites szín használatával, amikor tudod, hogy egy régebbi grafikus kártyát fogsz használni és minden bitnyi sebesség számít. A 16-bites szín használatának legfőbb hátránya, hogy néha rossz csíkozó hatást eredményez.

Egy eszköz ellenőrzés végrehajtása

Most végrehajthatsz egy eszközellenőrzést:

```
bool b;
b = Direct3D.Manager.CheckDeviceType(
        0, // alapértelmezett adapter
        Direct3D.DeviceType.Hardware, // akarunk hardveres renderelést!
        Direct3D.Format.X8R8G8B8, // 32 bites szín
        Direct3D.Format.X8R8G8B8, // backbuffer ugyanaz
        true );
```

Ha a *b* értéke *true* (igaz), akkor tudod, hogy megvan a hardver eszköz, ami tudja támogatni a kívánt formátumot. Ha *b* értéke *false* (hamis)… nos, bocsi. Próbálkozz egy másik formátummal.

A jelenlegi formátum használata

Ha lusta vagy (és melyik önérzetes játékprogramozó nem az?), akkor a legkönnyebb módja egy kijelző formátum felvételének, ha előremész és azt az egyet használod, amit a felhasználó már használ a Windows számára. Ezt elérheted a *Manager* osztály használatával, hogy megkapd a jelenlegi kijelző formátumot:

```
Direct3D.Format current;
current = Direct3D.Manager.Adapters[0].CurrentDisplayMode.Format;
```

Most a *current* fogja tárolni a jelenlegi kijelző módot. És az a mód garantáltan működik, hacsak a felhasználó nem csinál valami hülyeséget, mint pl. a Windowst olyan grafikus módban használni, amit a videókártyája nem támogat.

Eszközlehetőségek ellenőrzése

Néha lehet, hogy le akarsz ellenőrizni egy eszközt hogy lásd, vajon támogatja-e azokat a képességeket, amiket akarsz tőle. Figyelmeztetnem kell téged, hogy ha megnézed a *Direct3D.Caps* szerkezetet, akkor megrémülhetsz. Teljes 63 tulajdonsága van, melynek még sok olyan szerkezete, amelyeknek még több saját tulajdonságai vannak! A Direct3D-nek tonnányi lehetősége van. A jó dolog, hogy majdnem minden, amit én használok ebben a könyvben, eléggé alapértelmezett a számítógépeken manapság, tehát nem kell, hogy elmenj matatni az eszközlehetőségek körül. De itt van, hogy hogyan nyerj vissza egy *Caps* szerkezetet egy egyéni eszköznek:

```
Direct3D.Caps caps =
        Direct3D.Manager.GetDeviceCaps( 0, Direct3D.DeviceType.Hardware );
```

És most a *caps* szerkezet tartalmazni fog minden dolgot, amit valaha akartál (vagy nem akartál) tudni az eszközről.

A keretrendszer frissítése

Most, hogy tudod, hogyan hozz létre eszközöket, frissítheted a keretrendszert a 6. fejezetből, hogy hozzáadd a képességet teljes képernyős játékok készítéséhez. Könnyűnek tűnik, ugye? Nos, nem az. Egy közös téma, amit látni fogsz a programozásban, hogy feltételezhetően jól játszol az operációs rendszerrel. Rég elmúltak azok a napok, amikor egy játék az egész operációs rendszer felett átvette az ellenőrzést, és soha nem látjuk már újra (hacsak nem dolgozol egy konzolos játék rendszeren, mint az Xbox vagy a Gamecube vagy amit következőnek csinálnak).

Az operációs rendszer tucatnyi programot futtat, és a felhasználó bármely percben dönthet arról, hogy egy kicsit elkapcsol a programodról és megnézi az e-mail-jeit vagy valami hasonló. Ha ezt a lehetőséget nem teszed lehetővé a felhasználó számára, akkor lehet, hogy dühös lesz rád. Ablakos módban a programod egy ablak, és szépen játszik a grafikus rendszerrel. Teljes képernyős módban viszont egy zűrzavar.

Mikor egy teljes képernyős alkalmazást készítesz, akkor mohóvá válsz. Elmondod az operációs rendszernek, hogy az egész képernyőt magadnak

akarod. Ez nem egy indokolatlan kérés – mindannyian tudjuk, hogy a legjobb teljes képernyőben játszani a játékokat. De amikor egy felhasználó elkapcsol, akkor a programod még fut, de a Windows szétrombolja a grafikus eszközödet (hogy meri!), és rád hagyja, hogy javítsd ki, mikor a felhasználó visszatér a programodhoz.

Fájdalom a fenékben. Amikor új direct3D felhasználókkal beszéltem, egyike a legközönségesebb kérdéseknek, amiket feltettek, hogy „Hogyan tilthatom le az Alt+Tab-ot?" nem pedig „Hogyan kezelhetem jobban a többfeladatos alkalmazásokat?" Ők bizonyára csak bezárnák a felhasználót a programba, és ezzel készen is lennének. Mennyire szeretném, hogy az élet ilyen egyszerű legyen.

Egy eszköz felállítása

Az új keretrendszer tudja támogatni az ablakos vagy teljesképernyős módot. Ennek elősegítésére egy új logikai változó adódott a keretrendszerhez:

```
static bool windowed = false;
```

Amikor *false* (hamis), akkor a keretrendszer egy teljesképernyős alkalmazást fog készíteni, ha pedig igaz, akkor egy ablakost.

A következő változás a régi keretrendszerből az *InitializeGraphics* függvényen belül történik, amely most létrehoz egy eszközt, amely azon alapul, hogy ablakosra akarod vagy sem. Az ablakos rész könnyű:

```
public void InitializeGraphics()
{
        // egy eszköz felállítása
        Direct3D.PresentParameters p = new Direct3D.PresentParameters();
        p.SwapEffect = Direct3D.SwapEffect.Discard;
        if( windowed == true )
        {
                p.Windowed = true;
        }
```

Minden amit tenned kell, hogy a paraméterek *Windowed* tulajdonságát *true* értékűre állítsd; minden másnál lehet hagyni az alapértelmezett értékeket.

Egy teljesképernyős eszköz felállítása egy kissé eltérő:

```
        else
        {
                Direct3D.Format current =
                        Direct3D.Manager.Adapters[0].CurrentDisplayMode.Format;
                p.Windowed = false; // teljesképernyő
                p.BackBufferCount = 1; // egy back puffer
                p.BackBufferFormat = current; // jelenlegi formátum használata
                p.BackBufferWidth = screenwidth;
                p.BackBufferHeight = screenheight;
        }
```

Egy érvényes pixelformátum keresése helyett használtam a rendszer jelenlegi formátumát (korábban már megmutattam, hogyan kell csinálni), *false*-ra állítottam a *Windowed*-et, és adtam az eszköznek egy back puffert. Egy teljes képernyős alkalmazás számára be kell állítanod a back puffer méretét is, ami a képernyő szélessége (*screenwidth*) és magassága (*screenheight*).

Tipp: most láthatod, hogy miért csináltam a *screenwidth* és *screenheight* statikus változókat a *Game* osztálynak: ezek legalább két különböző helyen vannak használva, és ha egyiken megváltoztatod az értéküket, elfelejtheted megváltoztatni a másikon.

Most, hogy a paramétereid felállítódtak, létre tudod hozni az eszközt, mint általában:

```
        graphics = new Direct3D.Device(
                0, Direct3D.DeviceType.Hardware, this,
                Direct3D.CreateFlags.SoftwareVertexProcessing, p );
        // Az eseménykezelők beállítása az eszköznek
        graphics.DeviceLost
                += new EventHandler( this.InvalidateDeviceObjects );
        graphics.DeviceReset
                += new EventHandler( this.RestoreDeviceObjects );
        graphics.Disposing
                += new EventHandler( this.DeleteDeviceObjects );
        graphics.DeviceResizing
                += new CancelEventHandler( this.EnvironmentResizing );
}
```

A kód utolsó szakasza semmi újat nem mutathat neked.

Jegyezd meg, hogy ez a kód feltételezi, hogy szoftveres csúcspont számítással fogok használni egy hardver eszközt. Nem szándékozom mást csinálni ebben a könyvben, mint ami megköveteli a hardveres csúcspont számítás sebességét, de szeretném használni a színárnyékolást, amit a hardvereszköz nyújt.

Ennek a kódnak 99 százalékos valószínűséggel futnia kell minden számítógépen. Komolyan – bármely számítógépnek, ami ezt nem tudja futtatni, annak a 3DFX Voodoo 1 videókártyánál régebbi videókártyája van, amelyet 1996-ban adtak ki.

A többfeladatosság kezelése

A többfeladatú műveletek elvégzéséhez számos kódfrissítésre van szükség, mivel ez egy elég összetett folyamat. Bármikor elkapcsol valaki a teljes képernyős alkalmazásodról, a Windows érvényteleníti a kijelző eszközödet, úgyhogy jobban teszed, ha nem próbálsz akármit rajzolni rá, hacsak nem akarod, hogy a programod felrobbanjon.

Az első változás a régi keretrendszerből, hogy egy másik logikai változó adódik:

```
static bool graphicslost = false;
```

Ez *true*-ra (igaz) áll bármikor, amikor a grafikus eszköz elvész, így tudod, hogy ne rajzolj semmit, és tudod azt is, hogy meg kell próbálnod újra venned a grafikus eszközt, amint képes vagy rá.

A következő változtatás az, hogy az ablakodat nem átméretezhetővé tesszük:

```
protected virtual void EnvironmentResizing( object sender, CancelEventArgs e )
{
        e.Cancel = true;
}
```

A *Cancel* tulajdonság *true*-ra (igaz) állítása közli az operációs rendszerrel, hogy törlöd az átméretezési eseményt, mert nem szeretnéd átméretezni az ablakot. Ha engedélyezed az átméretezést, akkor a Windows megpróbálja, de valami furcsa okból összetöredezi az ablakodat, mikor többfeladatosan működik.

Az utolsó módosítás a *Render* függvény belsejében történik.

Megjegyzés: az összes következő kód az *if(graphics != null)* sorral kezdődő blokkon belül történik. Én eltávolítottam azt a részt, mert ennek a könyvnek a margói nem elég szélesek, hogy mutassák az egész dolgot.

Most, hogy megvannak az alapjai a felállításnak, itt az idő megnézni a kódot a *Render* függvényen belül:

```
if( graphicslost )
{
	try
	{
		graphics.TestCooperativeLevel();
	}
```

Ha a grafikus eszköz elveszett, akkor hívom a *TestCooperativeLevel* függvényt, mely tesztel, hogy megbizonyosodj az eszköz érvényességéről és együttműködik az operációs rendszerrel. Nyilvánvalóan amint a *graphicslost* az *true* (igaz) értékű, tudod, hogy az eszköz nem működik együtt, de van egy kis információ, amit nem tudsz: lehet az eszköz szükséges vagy sem? E függvény hívása lehetővé teszi, hogy kitaláld.

Mikor az eszköz elveszett és hívod ezt a függvényt, akkor dob egy *DeviceLostException* vagy *DeviceNotResetException* típusú kivételt. Az első kivétel azt jelenti, hogy az eszköz elveszett, és még nem lehet visszaszerezni. Abban az esetben csak visszatér és nem csinál semmilyen renderelést:

```
	catch( Direct3D.DeviceLostException )
	{
		// az eszközt nem lehet még visszaszerezni, ezért csak visszatérés:
		return;
	}
```

Másfelől ha a *DeviceNotResetException* dobódik, akkor tudod, hogy az eszköz még elveszett, de most biztonságos a visszaállítása, tehát az, amit pontosan szándékozol csinálni:

```
	catch( Direct3D.DeviceNotResetException )
	{
		// device has not been reset, but it can be reacquired now
		graphics.Reset( graphics.PresentationParameters );
	}
```

```
        graphicslost = false;
}
```

Nem olyan jó? A *Direct3D.Device* osztály nyújt egy praktikus *Reset* funkciót. Minden, amit tenned kell, hogy átadj néhány megjelenítési paramétert. Szintén szerencsés számodra, hogy az eszközöd emlékszik a megjelenítési paraméterekre, amiket utoljára használtál, tehát ez az, amit vissza akarsz állítani az eszköznek. Aztán beállítjuk a *graphicslost* logikai változót *false*-ra (hamis), mivel az eszköz többé már nem elveszett.

Az utolsó kóddarab egy változtatás a jelenlegi renderelési kódhoz (a változtatások félkövérek a kódban):

```
try
{
        graphics.Clear( Direct3D.ClearFlags.Target, Color.Blue, 1.0f, 0 );
        graphics.BeginScene();
        // CSINÁLNI : Helyszín renderelés
        graphics.EndScene();
        graphics.Present();
}
// az eszköz elveszett, és nem lehet még újra inicializálni
catch( Direct3D.DeviceLostException )
{
        graphicslost = true;
}
```

Egy try-blokk adódott a renderelő ciklus köré, és egy catch-blokk adódott a végéhez, hogy elkapja, ha az eszköz elveszett. Ha elvész, akkor a *graphicslost* logikai változó *true*-ra állítódik, és a renderelő függvény megpróbálja visszanyerni a ciklus megtörténtének újabb alkalmával.

Most a keretrendszer megfelelően kezeli a többfeladatosságot teljesképernyős módban.

Jelenlegi rajzolási anyag

A Direct3D-t sokkal könnyebb beállítani, mint korábban. A Direct3D beállítása hatalmas erőfeszítés volt, és így sok programozó sikoltotta le a fejét. Az a tény, hogy körülbelül 15 oldalon keresztül mutattam be, hogy hogyan kell beállítani, illene, hogy beláttassa veled, milyen összetett volt a használata és hogy hálás legyél azért, mert már ez sokkal könnyebb. De biztos vagyok abban, hogy már betegre tanultad magad arról, hogyan kell beállítani egy eszközt, és most már csak rajzolni akarsz. Nem mondhatom, hogy hibáztatlak.

A Direct3D egy háromszög-raszteres rendszer. Tényleg ez minden: háromszögeket rajzol. Ennek oka az, hogy a háromszög a legegyszerűbb geometriai alakzat, amelyhez van egy terület. Majdnem bármely három-dimenziós tárgyat ki lehet rajzolni egy 3D képernyőre csupán azáltal, hogy száz, ezernyi, vagy milliónyi háromszöggé alakítod.

Megjegyzés: Oké. Technikailag nem tudsz bármilyen objektumot háromszögek használatával megrajzolni. Sok tárgy van, mint gömbök és más kanyargós dolgok, amelyeket nem lehet pontosan meghatározni háromszögek használatával. Bár, ha elegendő háromszöget használsz, akkor úgy fog kinézni, mint egy kanyargós tárgy. Ez a mi kis titkunk; nem kell elmondani a felhasználóidnak, hogy a gömbjeid tulajdonképpen kis háromszögek nagy gyűjteménye.
Megjegyzés: sajnos nincs elég helyem, hogy belevágjak teljes 3D témákba, mint 3D transzformációk, pixel shaderek, és így tovább. Ez azt jelenti, hogy ragaszkodom néhány meglehetősen egyszerű grafikai lehetőséghez a könyv hátralévő részében. Ha egy még részletesebb betekintést keresel a grafikus programozásba, akkor figyelmedbe ajánlom Wolfgang Engel: Kezdő Direct3D játék programozás című könyvét (második kiadás).

Csúcspontok

Most rajzolni akarsz néhány háromszöget! Az első dolog, amit csinálni akarsz, hogy létrehozz valamit, amiben tárolod a geometriádat. Mivel a Direct3D háromszögeket rajzol, ezért neked magától értetődően kell valami mód háromszög adatok tárolására. A háromszög adat csúcspontoknak (vertex) nevezett objektumokban tárolódik. Egy csúcspont legegyszerűbben a 3D térben meghatározott pont. A Direct3D támogatja a csúcspont formátumok összes fajtáját, bár sokuk további adatokat is tartalmaz.

A Direct3D a *Direct3D.CustomVertex* osztállyal nyújtja, mely meghatároz sok csúcspont szerkezetet. Ezek listája a következő táblázatban található:

Csúcs	Átalakított	Szín	Textúra	Normál
PositionOnly	nem	nem	nem	nem
PositionColored	nem	igen	nem	nem
PositionTextured	nem	nem	igen	nem
PositionNormal	nem	nem	nem	igen
PositionColoredTextured	nem	igen	igen	nem
PositionNormalColored	nem	igen	nem	igen
PositionNormalTextured	nem	nem	igen	igen
Transformed	igen	nem	nem	nem
TransformedColored	igen	igen	nem	nem
TransformedTextured	igen	nem	igen	nem
TransformedColoredTextured	igen	igen	igen	nem

Az első dolog, amit észrevehetsz, hogy két fő típusa van a csúcspontoknak: helyi (position) és átalakított (transformed) csúcspontok. A helyi csúcspontok nem transzformáltak, ami azt jelenti, hogy a helyzetük valahol a világodban van meghatározva, és nekik szükséges átmenniük a Direct3D átalakító csővezetéken azért, hogy át legyenek alakítva képernyő koordinátákra. Az átalakított csúcspontok már képernyő koordináták, tehát ha egy átalakított csúcspont x és y értékeit elhelyezed (100,200)-ra, akkor a csúcspont 100 képponttal a képernyő bal oldalától és 200 képponttal a tetejétől lefelé fog kirajzolódni.

A szín érték meglehetősen magától értetődő; meghatározza a csúcspont színét. A textúra koordináták valami összetett ötlet, amelyre kitérek még később ebben a fejezetben. A normálok egy összetett téma, bár ennek részletes kifejtésére nincs elég helyem.

Megjegyzés: lehet, hogy ezelőtt még nem hallottál a normál adatról. Egy normál egyszerűen egy 3D vektor, amely merőleges a háromszög homlokzatára; ez az adat segít a Direct3D-nek kitalálni, hogyan érinti a fény egy háromszög homlokzatát. Illene megjegyezned, hogy csak azok a csúcspontok nem transzformáltak, amelyeknek van normál adatuk. Ez azért van, mert a D3D transzformációs csővezeték a megvilágítási számításokra is gondot visel. Ez feltételezi, hogy az átalakított csúcspontokon már végre vannak hajtva megvilágítási számítások; ezért egy normál vektor nem szükséges átalakított csúcspontokon.

Néhány csúcspont meghatározása

Egy egyszerű bemutatóhoz be szándékozom mutatni, hogy hogyan kell rajzolni egy egyszerű háromszöget. Az első dolog hozzá létrehozni egy szerkezetet, ami tárolja a csúcsokat. Ehhez a demóhoz átalakított színezett csúcsokat használok:

```
Direct3D.CustomVertex.TransformedColored[] vertexes = null;
```

Egyszerűen csak deklaráltam transzformált és színezett csúcsoknak egy üres tömbjét.

A következő lépés létrehozni három csúcspontot és inicializálni őket:

```
public void InitializeGeometry()
{
        vertexes = new Direct3D.CustomVertex.TransformedColored[3];
        // a tetejének a csúcspontja:
        vertexes[0].X = screenwidth / 2.0f; // félúton a képernyőn keresztül
        vertexes[0].Y = screenheight / 3.0f; // 1/3-dal le a képernyőn
        vertexes[0].Z = 0.0f;
        vertexes[0].Color = Color.White.ToArgb();
```

A tömb létrejött elegendő hellyel, hogy tároljon három csúcspontot (három csúcspont képez egy háromszöget), és aztán kitöltődik az X, Y, Z és szín (Color) információ minden csúcsponthoz.

Megjegyzés: biztos vagyok abban, hogy érted az X és Y információt, de a Z megzavarhat, ezért hadd fejtsem ki ezt neked érintőlegesen. Egy 3D programban, mikor képpontokat alakítasz át 2d képernyő-térbe, akkor *mélység* információt tartasz meg, így tudod, hogy mely képpontok vannak közelebb a nézőhöz. Ha van létrehozva egy *mélység puffer*ed (melyet nem tudok megmutatni neked ebben a könyvben), akkor a D3D mélység-ellenőrzést végez. Amikor rajzolsz egy képpontot, akkor a D3D ellenőrzi a mélység puffert, hogy lássa, van-e már odarajzolva egy másik pixel. Ha az előző képpont közelebb van a nézőhöz, akkor az új pixel nem rajzolódik ki, hogy ne láthassák. Nekem nincs szükségem a mélység pufferre, ezért a Z-t mindig 0-ra állítom.

A következő két csúcspont megadása:

```
        // jobb csúcspont:
        vertexes[1].X = (screenwidth / 3.0f) * 2.0f; // 2/3 a képernyőn keresztül
        vertexes[1].Y = (screenheight / 3.0f) * 2.0f; // 2/3 le a képernyőn
        vertexes[1].Z = 0.0f;
        vertexes[1].Color = Color.White.ToArgb();
        // bal csúcspont:
        vertexes[2].X = screenwidth / 3.0f; // 1/3 a képernyőn keresztül
        vertexes[2].Y = (screenheight / 3.0f) * 2.0f; // 2/3 le a képernyőn
        vertexes[2].Z = 0.0f;
        vertexes[2].Color = Color.White.ToArgb();
}
```

És készen is vagy a háromszöged meghatározásával.

Megjegyzendő, hogy a háromszög csúcspontjainak meghatározása jobbra, az óramutató járása szerint van, mégpedig azért, mert a Direct3D alapértelmezésben nem rajzol ki háromszögeket, ahol a csúcsok az óramutató járásával ellentétesek. Ezt hívják *hátlap levágás*nak (*backface culling*), amit egy kicsit részletesebben kifejtek majd a következő szakaszban.

Végső simítások

Van még néhány lépés, ami szükséges a demo befejezéséhez. Az első lépés a hátlap levágás kikapcsolása. A hátlap levágás egy olyan vonás, amely nagyon hasznos 3D alkalmazásokban, mert a legtöbb háromszöget csak egyik oldalról lehet látni. Képzelj el egy kockát: nem láthatod egy szilárd kocka belsejét, tehát a lapok a kocka belsejében soha nem látszódhatnak. A Direct3D tudja ezt, és nem vesztegeti az idejét nem látható dolgok kirajzolására. Ezokból kellene az összes 3D modelledet az összes háromszögeikkel az óramutató járása szerint megtervezned, hogy mikor arra az oldalra nézel, az feltételezhetően látható lesz. Mikor a másik oldalról nézel a háromszögre, a csúcspontok óramutató járásával ellentétesek lesznek, és a Direct3D tudni fogja, hogy nem kell kirajzolni azokat.

Sajnos, ha csupán 2D munkát végzel a Direct3D-ben, ez probléma lehet. Nem akarom, hogy mindig emlékezni kelljen az óramutató szerinti háromszögeim csinálására – micsoda fájdalom a fenékben! Tehát kikapcsolom ezt az opciót az *InitializeGraphics* függvényben:

```
graphics.RenderState.CullMode = Direct3D.Cull.None;
```

Az utolsó dolog, amire szükség van, a háromszög tulajdonképpeni kirajzolása, amely a *Render* függvényen belül van:

```
graphics.Clear( Direct3D.ClearFlags.Target, Color.Black, 1.0f, 0 );
graphics.BeginScene();
graphics.VertexFormat =
        Direct3D.CustomVertex.TransformedColored.Format;
graphics.DrawUserPrimitives(
        Direct3D.PrimitiveType.TriangleList,
        1, vertexes );
graphics.EndScene();
graphics.Present();
```

Az első és utolsó két sora ennek a kódnak ismerős lehet számodra, mivel már láttad őket ezelőtt. A grafikus eszköz feketére van törölve (kékről változtattam, mert a kék zavaró volt) és az eszköz azt mondja, hogy szándékozol elkezdeni megjeleníteni a helyszínedet.

A kód félkövér része az összes új anyag, amit hozzáadtam. Elmondom a grafikus eszköznek, hogy milyen fajta csúcspontokat rendereljen (transzformált színezett csúcsok), és aztán hívom a *DrawUserPrimitives* függvényt, hogy kirajzolja a háromszögek egy listáját (csak egy háromszöget tartalmaz a listám, de attól az még egy lista). Az utolsó paraméter a csúcsok tömbje.

Az utolsó kóddarab elmondja az eszköznek, hogy készen vagyok a rajzolással, és mutatni kellene már a helyszínt a felhasználónak.

Színek és alfa

Rendben, most tudsz rajzolni háromszögeket. Biztos vagyok abban, hogy boldog vagy, amiért sikeresen teljesítettél valamit, de ha megpróbálod ezt megmutatni valaki másnak, akkor valószínűleg csak néznek majd rád, mint egy őrültre és azt mondják: „Micsoda? Ez csak egy háromszög.” A nem-programozók nem értik és nem értékelik ezt.

Akkorhát gyerünk némi haladóbb anyaggal, mint a színek és alfa áttetszőség.

Játék a színekkel

Egyike a Direct3D legtisztább tulajdonságainak, hogy van önműködő háromszög árnyékolója beépítve. Nem szereted a tömör fehér háromszöget? Csináljuk a csúcsokat különböző színűekre! Személy szerint az akvamarin, a búzavirág kék és a citromszín kombinációját kedvelem:

```
vertexes[0].Color = Color.Aquamarine.ToArgb();
vertexes[1].Color = Color.CornflowerBlue.ToArgb();
vertexes[2].Color = Color.LemonChiffon.ToArgb();
```

Természetesen nem kell az előre beállított Windows színeket használnod. Helyettük létrehozhatod a saját szám szín kombinációidat:

```
vertexes[0].Color = Color.FromArgb( 0, 127, 0, 255 ).ToArgb();
```

Ez a kód a *System.Drawing.Color* osztályt használja a szín létrehozásához és az egész érték visszanyeréséhez. Ez a szín egyfajta indigó-lila, félig intenzív piros, nem zöld és teljes intenzitású kék (127,0,255).

Játék az alfával

Láthattad már az *alfa* kifejezést ezelőtt néhányszor ebben a könyvben, de tulajdonképpen nem volt esélyed látni, hogy mi is az. Egyszerűen fogalmazva az alfa információ csak extra információ, amit fel lehet ragasztani egy színre, és rajtad múlik, hogy kitaláljuk, mit akarsz vele csinálni.

Az alfa csatorna legközönségesebb használata az áttetszőség számára van, egy *alfa keverési* folyamaton keresztül. Alfa keveréssel a számítógépe egy új színt számít ki, ami azon alapul, hogy mit akarsz és mi van már ott. Be kell állítanod a Direct3D eszközt egy algoritmussal, ami végrehajtja ezeket a keveréseket.

Amikor van engedélyezve keverésed, és megpróbálsz rajzolni egy képpontot, a számítógép olvasni fogja a már képernyőn lévő pixelt, és keveri azzal a pixellel, amit rajzolni próbálsz, használva bármely műveletet.

Ez az alapvető algoritmus, amit a keverés követ:

újszín = [forrásszín * forrásfaktor] **művelet** [célszín * célfaktor]

A forrásszín az a szín, amelyet rajzolsz, a célszín pedig az, amely már a képernyőn van. Mondjuk van neked egy 1.0 értékűre állított forrás faktor, egy 0.0 értékű célfaktor és a művelet a hozzáadás. Ezesetben az egyenlőség ilyenné válik:

newcolor = [sourcecolor * 1.0] + [destcolor * 0.0]

vagy, egyszerűsítve, ilyenné:

newcolor = sourcecolor

mely azt jelenti, hogy nem igazán keversz semmit; csak egyszerűen kirajzolod az új színt a képernyőn.

Az áttetszőség keverés egy példája

Most, hogy látod, hogyan működik az alfa keverés egy nagyon alapvető módban, hadd mutassak neked egy fejlettebb módszert, ami tulajdonképpen áttetszőségi számításokat hajt végre. Ha a forrás keverési tényezőt a forrás alfa értékre állítod, és a cél keverési tényezőt a forrás alfa érték fordítottjára (egy mínusz az alfa érték), akkor most engedélyezve van az áttetszőség keverés.

Legyen mondjuk kettő képpontod, egy kék és egy piros. A kék a cél szín – ez már a képernyőn van. Ennél a példánál a pirosnak van egy 191-es alfa értéke (vagy körülbelül 0,75, ha 255 töredékeként nézel rá), ami azt jelenti, hogy 75%-ban tömörnek vagy 25%-ban átlátszónak akarod. A művelet „hozzáadás” marad:

newcolor = [red * 0.75] + [blue * (1.0 - 0.75)]

És egyszerűsítve:

newcolor = [red * 0.75] + [blue * 0.25]

Tehát milyen szín lesz pontosan a végén? Hajtsuk végre a pillanatnyi számításokat és találjuk ki:

newcolor = [(255,0,0) * 0.75] + [(0,0,255) * 0.25]
newcolor = (191,0,0) + (0,0,63)
newcolor = (191,0,63)

Tehát az új szín (191, 0, 63), ami egy kékes-piros; úgy néz ki nekem, mint egy sötét rózsaszín szegfű szín. Ha a piros képpont alfa értékét 255-re állítod, akkor a művelet valahogy így nézne ki:

newcolor = [(255,0,0) * 1.0] + [(0,0,255) * (1.0 - 1.0)]
newcolor = (255,0,0) + (0,0,0)
newcolor = (255,0,0)

Azt jelenti, hogy a piros teljesen tömör, és egyetlen eredeti szín sem jelenik meg rajta. Hasonlóképpen, egy 0 alfa érték azt jelenti, hogy az új képpont teljesen átlátszó, míg egy 127-es (0,5) alfa érték 50%-os áttetszőséget jelent, adva neked (127,0,127) értéket, azaz sötétlilát.

Alfa keverés beállítása

Az alfa keverés alapértelmezésben nem engedélyezett a Direct3D-ben. Magadnak kell megcsinálnod:

```
graphics.RenderState.AlphaBlendEnable = true;
```

De munkád még nem ér itt véget. Ezután be kell állítanod a keverési műveletet:

```
graphics.RenderState.AlphaBlendOperation = Direct3D.BlendOperation.Add;
```

Technikailag nem kell kiírnod ezt az opciót, mert az alapértelmezett keverési művelet a hozzáadás („*add*"), de mindig biztonságos a kódodat olvashatóbbá tenni.

Az elérhető keverési műveleteket a következő táblázat mutatja:

Érték	**Hatása**
Add	A két szín együvé adása
Subtract	Az eredmény a forrás mínusz célszín
RevSubtract	Az eredmény a cél mínusz forrásszín
Min	Sötétebb színt használ
Max	Világosabb színt használ

A következő lépés a keverési faktorok beállítása. Ezesetben áttetszőségi keverés végrehajtására szándékozom beállítani őket:

```
graphics.RenderState.DestinationBlend = Direct3D.Blend.InvSourceAlpha;
graphics.RenderState.SourceBlend = Direct3D.Blend.SourceAlpha;
```

Különböző keverési faktorokat használhatsz, hogy különböző hatásokat érj el, ha nem akarsz áttetszőségi keverést. A következő táblázatban megtalálhatók az elérhető keverési tényezők, bár nem listázza ki az összes keverést, hanem csak a leghasznosabbakat. A táblázatban nem szereplő négy keverési lehetőség elavult, vagy csak különleges körülmények között használható. Én még soha nem használtam ezelőtt, és valószínűleg neked sem kell aggódnod miattuk.

Érték	**Faktor**
Zero	(0, 0, 0, 0)
One	(1, 1, 1, 1)
SourceColor	(As, Rs, Gs, Bs)
InvSourceColor	(1 - As, 1 - Rs, 1 - Gs, 1 - Bs)
DestinationColor	(Ad, Rd, Gd, Bd)

InvDestinationColor	(1 - Ad, 1 - Rd, 1 - Gd, 1 - Bd)
SourceAlpha	(As, As, As, As)
InvSourceAlpha	(1 - As, 1 - As, 1 - As, 1 - As)
DestinationAlpha	(Ad, Ad, Ad, Ad)
InvDestinationAlpha	(1 - Ad, 1 - Ad, 1 - Ad, 1 - Ad)
SourceAlphaSat	(1, min(As, 1 - Ad), min(As, 1 - Ad), min(As, 1 - Ad))

*Az *s* index a forrás összetevőre, a *d* index pedig a cél összetevőre utal.

Egy másik demó

Most, hogy tudod, hogyan színezz csúcspontokat és végezz alfa számításokat rajtuk, tudsz játszani ezen tulajdonságokkal. Ezt bizonyítandó készítettem egy demót, amely bemutatja *két* háromszög létrehozását!

Itt a háromszög beállítása:

```
public void InitializeGeometry()
{
	vertexes = new Direct3D.CustomVertex.TransformedColored[6];
	// 1. háromszög:
	// a felső csúcspont:
	vertexes[0].X = screenwidth / 2.0f; // félúton a képernyőn keresztül
	vertexes[0].Y = screenheight / 3.0f; // 1/3 le a képernyőn
	vertexes[0].Z = 0.0f;
	vertexes[0].Color = Color.FromArgb( 255, 255, 0, 0 ).ToArgb();
	// jobb csúcspont:
	vertexes[1].X = (screenwidth / 3.0f) * 2.0f; // 2/3 keresztül a képernyőn
	vertexes[1].Y = (screenheight / 3.0f) * 2.0f; // 2/3 le a képernyőn
	vertexes[1].Z = 0.0f;
	vertexes[1].Color = Color.FromArgb( 255, 0, 255, 0 ).ToArgb();
	// bal csúcspont:
	vertexes[2].X = screenwidth / 3.0f; // 1/3 keresztül a képernyőn
	vertexes[2].Y = (screenheight / 3.0f) * 2.0f; // 2/3 le a képernyőn
	vertexes[2].Z = 0.0f;
	vertexes[2].Color = Color.FromArgb( 255, 0, 0, 255 ).ToArgb();
	// 2. háromszög:
	// alsó csúcspont:
	vertexes[3].X = screenwidth / 2.0f; // félúton a képernyőn keresztül
	vertexes[3].Y = (screenheight / 3.0f) * 2.0f; // 2/3 le a képernyőn
	vertexes[3].Z = 0.0f;
	vertexes[3].Color = Color.FromArgb( 127, 255, 0, 0 ).ToArgb();
	// jobb csúcspont:
	vertexes[4].X = (screenwidth / 3.0f) * 2.0f; // 2/3 a képernyőn keresztül
	vertexes[4].Y = screenheight / 3.0f; // 1/3 le a képernyőn
	vertexes[4].Z = 0.0f;
	vertexes[4].Color = Color.FromArgb( 127, 0, 255, 0 ).ToArgb();
```

```
        // bal csúcspont:
        vertexes[5].X = screenwidth / 3.0f; // 1/3 a képernyőn keresztül
        vertexes[5].Y = screenheight / 3.0f; // 1/3 le a képernyőn
        vertexes[5].Z = 0.0f;
        vertexes[5].Color = Color.FromArgb( 127, 0, 0, 255 ).ToArgb();
}
```

Az első háromszög azonos helyzetben marad, de megváltoztattam a színeket. A csúcspontok vörös, zöld és kék és minden csúcsnak van egy teljes 255-ös alfa értéke, ami azt jelenti, hogy tökéletesen tömör.

A második háromszög fejjel lefelé fordított; azonos színei vannak, de az alfa érték minden csúcsnak 127, ami azt jelenti, hogy 50 százalék áttetszőséggel akarom kirajzolni.

Az egyedüli másik változás az előző demóhoz képest, hogy most egy helyett a renderelő két háromszöget rajzol ki:

```
graphics.DrawUserPrimitives(
        Direct3D.PrimitiveType.TriangleList, 2, vertexes );
```

Textúrák és egyéb alakok

Nagyon menő annak lehetősége, hogy mutass csinos, színezett háromszögeket, de ha ez minden, amit tudsz csinálni, akkor a játékaid inkább néznek ki úgy, mint az 1980-as években a Kraftwerk együttes zenei videója, semmint egy csúcstechnológiájú számítógépes játék.

Emiatt találták fel a *textúrázás*nak nevezett módszert, amely még részletesebb kinézetűvé teszi a háromszögeidet. A továbbiakban a textúrázáshoz azt is bemutatom, hogyan optimizáld a geometriádat, felhasználva haladó primitív gyűjteményeket, mint például háromszög-szalagok és háromszög-legyezők.

Textúrázás

Hogy objektumaidat jobb kinézetűvé tedd, ahhoz részleteket kell hozzájuk adni. Csinálhatod ezt textúrák, azaz felületi mintázatok használatával, amelyek képek, amik rá vannak húzva a háromszögeidre. Készíthetsz egy sokszögekből álló hegységet a játékodban, de ez meglehetősen hülyén fog kinézni, ha csupán tömör-színezett háromszögekből áll. A hegység jobb kinézetéhez valószínűleg

szándékozol találni szikla kinézetű képeket, és aztán azt a szikla textúrát ráhelyezni minden háromszögre a modellben.

Én nem fogok ennyire merészeket tenni, csupán azt mutatom meg, hogyan kell egy textúrát betölteni egy állományból. A Direct3D ezt hihetetlenül könnyen megteszi a *TextureLoader* osztállyal.

Itt van az, hogy hogyan határozz meg egy textúrát:

```
Direct3D.Texture textura = null;
```

Úgy gondolom, ez nem okozott nagy fejtörést.

Egy textúra betöltése

És most, egy textúra betöltéséhez:

```
textura = Direct3D.TextureLoader.FromFile(
        graphics, “textura.jpg”, 0, 0, 0, 0, Direct3D.Format.Unknown,
        Direct3D.Pool.Managed, Direct3D.Filter.Linear,
        Direct3D.Filter.Linear, 0 );
```

Nos, ez egy csomó lehetőség. Neked szerencsére, valószínűleg csak az alapértelmezett lehetőségeket akarod használni egy textúrához. Hadd fejtsem ki, hogy mik ezek a lehetőségek és mit csinálnak. Itt a teljes meghatározása a *TextureLoader.FromFile* függvénynek, így láthatod az összes lehetőséget:

```
public static Texture FromFile(
        Device device,
        string srcFile,
        int width, int height,
        int mipLevels,
        Usage usage,
        Format format,
        Pool pool,
        Filter filter,
        Filter mipFilter,
        int colorKey
);
```

Az első paraméter magától értetődően a grafikus eszköz, a második paraméter pedig az állomány neve, amit be akarsz tölteni. Ez a függvény különböző grafikus állomány formátumok tonnáit tudja betölteni: .BMP, .JPG, . PNG, .TGA, .DDS, .DIB, .HDR, .PFM és .PPM.

A következő a szélesség és a magasság. Be tudod ezeket kézzel állítani, de miért zavar? Ha 0-n hagyod őket, akkor a program önműködően a kép méretét használja az állományból, amit betöltöttél.

A következő változó meghatározza, hogy mennyi mip-szint van megadva ennek a textúrának. Ez a *mip-mappolás*hoz használt, ami lényegében fogja a textúrádat, és kisebb képek láncolatát állítja elő (ha a textúrád 64x64-es, akkor egy teljes mip-lánc tartani fogja a 32x32, 16x16, 8x8, 4x4, 2x2 és 1x1-es változatait a textúrának), melyet a D3D mutat, amikor a textúrád összezsugorodik. A mip-mappolás igazán jó kinézetet eredményez, és a videókártyádnak nem kell majd a textúráid hatalmas változatait tartania, ha olyan dolgokra nézel, amelyek messze vannak. Igazán nem kell aggódnod a mip-mappolás miatt, csak hagyd 0-n ezt a paramétert, és a D3D önműködően létrehoz egy teljes láncolatot neked.

A *usage* paraméter tudatja a D3D-vel, ha szándékozol használni ezt a textúrát bármilyen különleges célra, mint renderelési anyagot a textúrára dinamikusan, lehetővé téve animált textúrák létrehozását játékodban (ez hasznos lehet víz/plazma/tűz effektusokra). Én nem fogom ezt a funkciót használni, ezért 0-n hagyom.

A *format* paraméter a textúra színformátuma; bármilyenre állíthatod, amilyet szeretnél. Ha átadod *Direct3D.Format.Unknown* alakban, akkor a D3D a fájl azonos színformátumát használja.

A *pool* paraméter meghatározza, hogy a memóriában hol lesz elhelyezve a textúra. Ha a *Direct3D.Pool.Managed* alakot használod, akkor közlöd a Direct3D-vel, hogy önműködően kezelje neked a textúrát. Ez az, amit valószínűleg mindig használnod kellene, hacsak valami különleges munkát nem szándékozol végezni a textúrán és ezért valami egyéni helyen szükséges elhelyezni a memóriában.

Megjegyzés: bármikor elveszíted a Direct3D eszközödet, az *összes* textúrád is el fog veszni. Ha a *managed memory pool*-t használod, a Direct3D gondot visel az összes textúrád automatikus visszatöltésére. A *managed memory pool*-ok a barátaid.

A következő két paraméterek szűrők; meghatározzák, hogyan szűrődik a textúra, mikor kinyújtódik vagy zsugorodik. Ha nincs egy szűrőd, akkor a D3D egy egyenes képpont méretezést végez, és az rossz kinézettel végződhet. Az egyenes irányút a legjobb használni rendszerint mindkét esetben; ez nem állít elő tökéletesen kinéző eredményt, de gyorsabb, mint a többi szűrő legtöbbje, és legtöbbször nem tudod igazán megmondani, hogy milyen fajta fejlesztést nyújt a többi lassabb szűrő.

Az utolsó paraméter a szín kulcs. Ez azt jelenti, hogy bármikor lát egy pixelt a D3D, ami egyezik a szín kulcs színével, akkor nem rajzolja ki azt a képpontot. A bittérképek és textúrák mindig 2D téglalapok, de a képeknek nem mindig téglalap-alakúnak kell lenniük. Például egy személy nem úgy néz ki, mint egy téglalap – lesznek képpontok a személy képében, amiket nem akarsz mutatni. Ezek a pixelek rendszerint egy egyéni színben rajzolódnak ki, így ha szín kulcsnak azt a színt állítod be, akkor azok a képpontok soha nem rajzolódnak ki. A szín kulcs tíltásához csak állítsd az értéket 0-ra.

Textúra koordináták

Van egy alapvető különbség a textúrák és a háromszögek között: a textúrák négyszögletesek, a háromszögek pedig nem. Ezért nehéz azt mondani, hogy „Rajzold ezt a textúrát erre a háromszögre”, mert a számítógép nem tudja, hogy melyik részét kell a textúrának kirajzolni. Ezért vannak használatban a textúra koordináták.

A textúra koordináták egy egyszerű mód annak elmondására a számítógépnek, hogy a textúra mely részét kell kirajzolni a háromszögön.

A textúra koordináták nagyon hasonlatosak az alapértelmezett 2D (x,y) skálához, kivéve, hogy a textúrák használnak két eltérő betűt: (u,v). Tehát pl. egy textúra bal felső sarka vonatkozik a (0.0, 0.0)-ra, a jobb felső az (1.0, 0.0)-ra, a bal alsó (0.0, 1.0)-ra, míg a jobb alsó (1.0, 1.0)-ra. Mindegyik koordináta értéktartomány 0.0-tól 1.0-ig van. Tehát egy háromszög számára csinálhatnál valami ilyesmit:

```
Direct3D.CustomVertex.TransformedTextured[] vertexes =
        new Direct3D.CustomVertex.TransformedTextured[4];
// bal felső csúcs:
vertexes[0].Tu = 0.5f;
vertexes[0].Tv = 0.0f;
// bal alsó csúcs:
vertexes[1].Tu = 0.0f;
vertexes[1].Tv = 1.0f;
// jobb alsó csúcs:
vertexes[2].Tu = 1.0f;
vertexes[2].Tv = 0.0f;
```

A jelenlegi képernyőkoordinátái azoknak a csúcspontoknak nem számítanak eddig a pontig; nem számít, hogyan mozgatod azokat köré, a textúra ki fog tágulni, hogy megfeleljen az egész háromszögre.

A geometria egyéb formái

Mondjuk akarsz csinálni egy négyszöget, amely négy csúcspontot igényel. De a Direct3D csak háromszögeket tud rajzolni, így két háromszög kombinálásával készítesz egy négyszöget. De van egy probléma ezzel a módszerrel: két háromszögnek hat csúcspontja van, nem négy. Két csúcspont meg van osztva a háromszögek között.

Memóriapazarlás, ha vannak ezek a további csúcspontok, és ha aDirect3D-t használod a csúcspontok átalakításához, akkor értékes számítási időt is pocsékolsz. Ha volna egy mód létrehozni két háromszöget csak négy csúcspont használatával…

Ah, de van is! A DirectX csapat gondolt erre a problémára, és beemeltek használni különböző típusú primitíveket neked. Az előző demókban használt primitíveket *háromszög listák*nak hívják, melyek egyszerű listája háromszögeknek, amiben minden három csúcspont meghatároz egy egyedi háromszöget.

A primitívek másik fajtája a *háromszög csík*, mely háromszögek egy csíkja.

Tehát használva egy háromszög csíkot meghatározhatsz sok egymáshoz csatlakoztatott háromszöget, amelyek mindegyike megosztja a közös csúcspontokat. Ez egy nagyon hatékony módja 3D objektumok megadásának.

Még egy másik primitív a *háromszög legyező*. Egy háromszög legyező háromszögek egy gyűjteménye, amelyben mindegyik ugyanahhoz a csúcshoz van csatlakozva.

Ezeknek a más típusú primitíveknek a használata meglehetősen egyszerű, mivel minden amit tenned kell, egy paraméter megváltoztatása a *DrawUserPrimitive* hívásában.

Demo 7.4

A Demo 7.4 a Demo 7.3-ra épül; színezett háromszögek helyett egy textúrázott négyzetet helyez a képernyőre.

A demónak először a csúcspontokat és a textúrát kell deklarálnia:

```
Direct3D.CustomVertex.TransformedTextured[] vertexes = null;
Direct3D.Texture texture = null;
```

Aztán létrehozom a négy csúcspontot és betöltöm a textúrát:

```
public void InitializeGeometry()
{
        vertexes = new Direct3D.CustomVertex.TransformedTextured[4];
        // bal-felső csúcs:
        vertexes[0].X = screenwidth / 4.0f;
        vertexes[0].Y = screenheight / 4.0f;
        vertexes[0].Z = 0.0f;
        vertexes[0].Tu = 0.0f;
        vertexes[0].Tv = 0.0f;
        // jobb-felső csúcs:
        vertexes[1].X = (screenwidth / 4.0f) * 3.0f;
        vertexes[1].Y = screenheight / 4.0f;
        vertexes[1].Z = 0.0f;
        vertexes[1].Tu = 1.0f;
        vertexes[1].Tv = 0.0f;
        // bal-alsó csúcs:
        vertexes[2].X = screenwidth / 4.0f;
        vertexes[2].Y = (screenheight / 4.0f) * 3.0f;
        vertexes[2].Z = 0.0f;
        vertexes[2].Tu = 0.0f;
        vertexes[2].Tv = 1.0f;
        // jobb-alsó csúcs:
        vertexes[3].X = (screenwidth / 4.0f) * 3.0f;
        vertexes[3].Y = (screenheight / 4.0f) * 3.0f;
        vertexes[3].Z = 0.0f;
        vertexes[3].Tu = 1.0f;
```

```
        vertexes[3].Tv = 1.0f;
        texture = Direct3D.TextureLoader.FromFile(
        graphics, “texture.jpg”, 0, 0, 0, 0, Direct3D.Format.Unknown,
        Direct3D.Pool.Managed, Direct3D.Filter.Linear,
        Direct3D.Filter.Linear, 0 );
}
```

És végül, renderelem a négyzetet:

```
graphics.Clear( Direct3D.ClearFlags.Target, Color.White , 1.0f, 0 );
graphics.BeginScene();
graphics.SetTexture( 0, texture );
graphics.VertexFormat = Direct3D.CustomVertex.TransformedTextured.Format;
graphics.DrawUserPrimitives(
Direct3D.PrimitiveType.TriangleStrip, 2, vertexes );
graphics.EndScene();
graphics.Present();
```

A legfontosabb lépés itt az első félkövér sor. Bármikor rajzolsz valami textúrázottat, el kell mondanod a grafikus eszköznek, hogy milyen textúrát akarsz használni. Aztán elmondod az eszköznek, hogy milyen fajta csúcsokat rajzolsz, és végül megrajzolod két háromszögnek egy csíkját.

Sprite-ok

Amikor egyszerű 2D sprite-okat akarsz rajzolni, akkor a Direct3D használata valami túlzás, de ez nyújt sok szép jellemzőt, minthogy a DirectDraw (a DirectX régi 2D rajzoló API-ja) soha nem csinálta.

Egy 2D sprite rendszer felállítása igazán szép munka, ezért a Microsoft előrement, és adott nekünk egy ügyes *Direct3D.Sprite* osztályt a rendetlen részletek kezelésére. Nincsenek rendetlenségek a csúcspontokkal vagy bármi mással kapcsolatban - csak egy szép, átfogó sprite osztály.

Figyelmeztetés: a *Sprite* osztály, együtt a többi segítő osztállyal, amelyek ebben a könyvben használtak – mint például a *Direct3D.TextureLoader* osztály és a *Direct3D.Font* osztály – a D3DX könyvtárhoz tartoznak, mely nem része a Direct3D magjának. A D3DX egy könyvtár, amelyet arra terveztek, hogy az életedet könnyebbé tegye, és tényleg segít.
Sajnos a D3DX sok változáson ment át, és még a mai napig változik. A D3DX függvények, amelyeket ebben a könyvben látsz, a DirectX 9.0b SDK részei, mely 2003 nyarán volt kiadva. Azóta van már másik DirectX SDK frissítés, a DirectX 9.0c, amely sok függvényben megváltozott, és ezek összeférhetetlenek a régebbi változatokkal. Az ok, amiért a DirectX 9.0b használatát választottam az, hogy több fordítóval működik együtt. A DX9.0b működik a

Visual Studio 2002-vel, a SharpDevelop-pal, és a Visual Studio 2003-mal. A DirectX 9.0c csak a Visual Studio 2003-mal működik.

A Sprite osztály

A *Sprite* osztály valójában zavaró megnevezés, mivel valójában nem képvisel egy sprite-ot. Ez csak egy segítő osztály, amely elfogad rajzoló információkat, amit átadsz bele, és létrehozza a sprite-okat neked. Hadd mutassak neked egy példát.

Először létre kell hoznod a változót:

```
Direct3D.Sprite sprite = null;
```

Aztán inicializálnod kell, használva a grafikus eszközt:

```
sprite = new Direct3D.Sprite( graphics );
```

Most, hogy létre van hozva az osztályod, rajzolhatsz vele sprite-okat:

```
graphics.Clear( Direct3D.ClearFlags.Target, Color.White , 1.0f, 0 );
graphics.BeginScene();
sprite.Begin();
sprite.Draw( texture,
        new Rectangle( 0, 0, 512, 512 ),
        new Vector2( 1.0f, 1.0f ),
        new Vector2( 256.0f, 256.0f ),
        0.0f,
        new Vector2( 0.0f, 0.0f ),
        Color.White );
sprite.End();
graphics.EndScene();
graphics.Present();
```

Váó! Ez egy csomó kód a *Draw* függvényen belül! Ez csak meg akarja mutatni neked, hogy milyen erőteljes függvény.

Most hadd fejtsem ki pontosan, hogy mi történik. Az első dolog, amit tenned kell, mikor sprite-okat rajzolsz, hogy a *Begin* hívása által elmondod a *Sprite*

osztálynak, hogy sprite információt szándékozol küldeni kezdeni. Amikor ezzel készen vagy, használhatod a *Draw* függvényt, hogy sprite információt adj.

A *Draw* függvény fog egy csomó betöltési paramétert, ahogy világosan láthatod. Itt a teljes függvény deklaráció:

```
public void Draw(
        Microsoft.DirectX.Direct3D.Texture srcTexture,
        System.Drawing.Rectangle srcRectangle,
        Microsoft.DirectX.Vector2 scaling,
        Microsoft.DirectX.Vector2 rotationCenter,
        System.Single rotation,
        Microsoft.DirectX.Vector2 translation,
        System.Drawing.Color color )
```

Az első paraméter természetesen a textúra, amit a sprite számára akarsz használni. A második paraméter talán nem olyan nyilvánvaló neked.

A második paraméter egy téglalap objektum, mely meghatározza, hogy a textúra mely részét akarod rajzolni. Ez szükséges, mert akarhatod a textúrának csak egy részét rajzolni, mint sprite-odat, és nem a teljes dolgot. A példában, amit fentebb adtam, meghatároztam egy új téglalapot, hogy körülvegyen egy 512x512 képpontnyi területet, mert akkora volt a textúra, amit használtam. Használhatsz kisebb területeket is, ha akarsz.

A következő paraméter egy 2D vektor, amely meghatározza, hogy hogyan méreteződik az objektum. Ha (1.0, 1.0) értéket adsz be, akkor nem méreteződik az objektum. Hasonlóképpen a (0.5, 0.5) a felére zsugorítja, a (2.0, 2.0) pedig megkétszerezné a méretét.

Az ezutáni paraméter szintén egy 2D vektor, amely ezidő szerint az elforgatási középpontját adja a sprite-nak. Képzeld el, hogy fogsz egy szöget, beleszúrod egy papírdarabba, és aztán elforgatod a papírt a szög körül. Ez pontosa az, ami egy forgatási középpont is. Én ezt (256, 256)-ra helyeztem, amely a középpontja az (512, 512)-es textúrának. Bár neked nem kell elforgatnod a sprite-jaidat a középpont körül, ha nem akarod. Ez tőled függ.

Megjegyzés: a sprite forgatási középpontja tulajdonképpen zavaró, és egy kis időbe telt kitalálni, hogy pontosan mit jelent az érték, mert a Managed DirectX dokumentáció eléggé lecsupaszított. A sprite forgatási középpontja nem skálázott, ami azt jelenti, hogy ha méretezed a sprite-ot, akkor a sprite forgatási középpontja nem fog mozogni amentén. Ha a sprite-od méretben 128x128-as, és (64, 64)-re állítod a forgatási középpontot, akkor elfordul a sprite középpontja körül – de csak ha (1.0, 1.0)-t használsz skálázási vektornak. Ha a sprite-ot 64x64-esre méretezed (0.5, 0.5) skálázási vektor használatával, akkor a forgatási középpont még (64, 64)-nél lesz, és a sprite-od végül forogni fog a jobb-alsó sarka, mint a középpontja körül.

Azután egy lebegőpontos szám jelképezi a sprite forgatási szögét, mely radiánban van megadva.

Megjegyzés: egy radián egyszerűen egy szög mértéke, amely a 0-tól 2pi-ig (megközelítőleg 6,28) terjedő értéktartományba esik. Gondolhatod, hogy 360 fok egyenlő 6,28 radiánnal, és 180 fok egyenlő 3,14 radiánnal. Az egyenlet, amely megadja a radiánokat fokokból:

radián=fok/(180/pi)

A következő paraméter egy vektor, amely meghatározza a sprite fordítását a képernyőn. Ha (0,0)-t használsz, akkor a sprite a képernyő bal-felső sarkánál rajzolódik ki, ha (100,0)-t, akkor a képernyő bal oldalától 100 képpontnyira, és így tovább.

Az utolsó paraméter egyszerűen a sprite színe, ha árnyékolni akarod vagy alfa keverést használni. A példámban a *Color.White*-ot használtam, ami egy árnyékolatlan sprite-ot ad neked. Könnyen használhatsz bármilyen más színkombinációt.

Tegyük a kódot jobbá

Az előzőleg mutatott rajzolókód egy zűrzavar. Nincs mód, hogy valaha is használj kódot úgy, ahogy ez kinéz. Nagy és szakszerűtlen, valamint bonyolult a kezelése. A dolgok jobbá tételéhez úgy döntöttem, hogy megcsinálom a saját *Sprite* osztályomat, ami kezelni fogja az olyan dolgokat, mint elhelyezkedés, forgatási szög és méretezés.

Egy jobb Sprite osztály

Ezt a *Sprite* osztályt a Demo 7.5-ön belül találod, a Sprite.cs állományban. Itt az adattagok listája:

```
public class Sprite
{
	Direct3D.Texture texture;
	Drawing.Rectangle rect;
	DirectX.Vector2 scaling;
	DirectX.Vector2 offset;
	DirectX.Vector2 anchor;
	float angle;
	Drawing.Color color;
}
```

Az osztály nyomon fog követni egy textúrát, a forrás négyszöget, egy méretezési vektort, egy eltolási vektort, egy horgony pontot, egy forgatási szöget és a sprite színét.

A horgony pontot kivéve minden közvetlenül viszonyít azokhoz a paraméterekhez, amelyeket átadsz a *Direct3D.Sprite.Draw* függvénybe. Egy horgony pont hasonló egy sprite elforgatási középpontjához, de nem pontosan.

Előzőleg már elmondtam neked, hogy a forgatási középpont nem méreteződik a sprite-od mérete mentén, és ez lehet egy fájdalom az ember fenekének. Továbbá, ha megváltoztatod a sprite skálázási tényezőjét, akkor a bal oldali sarok marad, ahol volt, és az összes többi oldal mérete megnövekszik.

Mikor a horgony pont a bal felső saroknál van, az egész sprite elfordul akörül a pont körül és méreteződik is körülötte, tehát soha nem növekszik felfelé vagy balra – egyszerűen csak méreteződik lefelé és jobbra.

Ha viszont a horgony pont a középpontban van, akkor az egész dolog egyenletesen skálázódik e pont körül.

A dolgok egyszerűbbé tételéhez, az én *Sprite* osztályom textúra koordinátákat fog használni a horgony pont meghatározásához, ami azt jelenti, hogy 0.0 – 1.0 skálázáson lesz meghatározva. Ezért (0.5, 0.5) pontosan a sprite középpontja, (0.0, 1.0) a bal alsó sarok, és így tovább.

Tulajdonságok

Az új *Sprite* osztály majdnem teljes egészében az értékek lekérése és beállítása tulajdonságokon alapszik. A legtöbb tulajdonság meglehetősen unalmas, így

csak a legfontosabbakat szándékozom megmutatni. Itt a listája az összes tulajdonságnak az osztályban:

```
public Direct3D.Texture Texture
public int Width // csak lekérdezhető
public int Height // csak lekérdezhető
public float XScale
public float YScale
public float Scale // csak beállítható
public float X
public float Y
public float XAnchor
public float YAnchor
public float Angle
public Drawing.Color Color
```

Ezek legtöbbje, mint a *Texture, XScale*, *YScale*, *X*, *Y*, *XAnchor*, *YAnchor*, *Angle* és *Color*, nyilvánvaló kell hogy legyen számodra.

A *Width* és *Height* csak olvasható; nem állíthatod be az értékeiket, mert az adott textúrából közvetlenül olvasnak. Ez a két érték elmondja neked a textúra szélességét és magasságát.

A *Scale* csak beállítható; ez egy rövidített parancs a sprite X és Y méretének egyidejű beállítására.

A tulajdonságok némelyike extra háztartási munkáról gondoskodik. Különösen a *Texture* tulajdonság – akármikor beállítasz egy új textúrát, visszaállítja a belső *rect* szerkezetet neked, így nem kell kézzel beállítani mindig egy új négyszöget, valahányszor megváltoztatod a textúrákat.

Rajzolás

A legösszetettebb része az új osztálynak a *Draw* függvény. Itt a deklarációja:

```
public void Draw( Direct3D.Sprite renderer, Camera camera )
```

A függvény első paraméterként fog egy *Direct3D.Sprite*-ot, és egy 2D vektor jelképezi a kamerát.

Egy kamera egy igazán egyszerűen megérthető elgondolás; alapvetően egy kamera koordinátái egy világ térben. Képzeld el a játék világodat egyetlen nagy 2D rácsként; adhatsz neki tetszőleges méreteket – mondjuk -1000-től 1000-ig függőleges és vízszintes irányokba, adva neked 2000x2000-es egységnyi univerzum méretet. A képernyőd nyilván nem tudja egyszerre mutatni az egész dolgot – a kamera csak egy kis darabjára tud mutatni az univerzumnak egy időben.

Például ha a képernyőd 640x480-as, akkor csak ennyi képpontot tudsz mutatni. Tehát egy egyszerű 2D világban egy kamerának rendszerint két koordinátája van, megjelölve azt, hogy mely részét kell mutatni az univerzumnak. Én azt részesítem előnyben, ha ezek a koordináták a képernyő közepére vannak meghatározva, így ha elmondod a kamerádnak, hogy mutasson a (100,200) univerzum koordinátákra, akkor azok a koordináták pontosan a képernyő közepére lesznek mutatva.

Megjegyzés: meg fogom neked mutatni, hogy hogyan használd a *Camera* osztályt, amit meghatároztam a keretrendszer részeként (amely megtalálható a Sprite.cs fájlban a Demo 7.5-ben) a következő szakaszban.

Minden további hűhó nélkül, itt a rajzoló függvény:

```
public void Draw( Direct3D.Sprite renderer, Camera camera )
{
	// a jelenlegi szélesség és magasság számítása:
	float w = XScale * Width;
	float h = YScale * Height;
```

Az első lépés a sprite jelenlegi szélességének és magasságának a számítása, amely a szélesség és magasság szorozva a méretezési tényezőkkel.

Itt a következő lépés:

```
	// a jelenlegi horgony pont számítása a sprite koordinátákban:
	DirectX.Vector2 scaledanchor = anchor;
	scaledanchor.X *= w;
	scaledanchor.Y *= h;
```

Ez a lépés létrehoz egy új vektort, amely tárolni fogja a jelenlegi horgony pontját a sprite-nak. Például ha van egy 0.25-tel méretezett 512x512-es sprite-

od, akkor a *w* (szélesség) és *h* (magasság) is 128. Most, ha a horgony pontod (0.5, 0.5)-re van állítva – amely a textúra közepe – ez a lépés meg fogja szorozni 128*0.5-tel, adva neked 64-et a méretezett horgony *X* és *Y* értékeinek. Ennek van értelme, mivel egy 128x128-as sprite közepe (64, 64).

A következőben eltolom a sprite-ot a méretezett horgony ponttal:

```
DirectX.Vector2 newoffset = offset;
newoffset.X -= scaledanchor.X;
newoffset.Y -= scaledanchor.Y;
```

A horgony pont meghatározza, hogy a sprite-nak hol kell kirajzolódnia. Ha egy sprite a (100, 100)-as koordinátánál van, és a horgony pont a sprite közepében, akkor valahányszor rajzolod a sprite-ot, a sprite közepét (100, 100)-nál akarod majd kirajzolni. Sajnos a *Direct3D.Sprite* a sprite bal felső sarkánál rajzolja a sprite-okat, tehát ha azt mondod neki, hogy (100, 100)-nál rajzoljon egy sprite-ot, akkor annak bal felső sarka lesz (100, 100)-nál. Ennek megoldására módosítom a sprite eltolását a méretezett horgony értékei által. Tehát használva az adatot, amit előzőleg mutattam, a horgony pont elmozdult volna balra és felfelé 64 képpontnyit, adva neked egy (36, 36) értékű új eltolást, és biztosítva, hogy a sprite középpontja (100, 100)-nál rajzolódik ki, ahogy akartad.

Végül a sprite elküldődött a *Direct3D.Sprite* eszköznek:

```
        renderer.Draw(
                texture,
                rect,
                scaling,
                scaledanchor,
                angle,
                newoffset - camera.Offset,
                color );
}
```

Az egyetlen sor ebből, amivel illene törődnöd, az, amelyik tartalmazza, hogy *newoffset – camera.Offset*. Ez egyszerűen mozgatja a sprite-ot azzal az eltolással, ami a kamerának van. Ha egy sprite tételezzük fel (0, 0)-nál rajzolódik ki, és a kamera is (0, 0,)-nál van, akkor semmi sem történik. Ha a kamera (100,

0)-ra tekint, akkor a sprite elmozdul balra 100 képpontot, hogy a kamera jobbra mozgásának kinézetét adja. Elég ügyes, nem?

A kamera osztály

A *Camera* osztály, amit csináltam, kissé egyszerű. Becsületes mennyiségű kódot kellett hozzáadnom, hogy önműködően elvégezze a házimunkát, de a végén hihetetlenül könnyen kezelhetőkét jött ki.

Alapvetően egy kamera tudja, hogy milyen nagy a képernyő és hol mutat rá rajta, és van néhány vektora ezeket az információkat tárolni:

```
public class Camera
{
        DirectX.Vector2 position;
        DirectX.Vector2 screensize;
        DirectX.Vector2 finaloffset;
}
```

Ha a kamera (0, 0)-ra mutat, akkor az a *position*-ban tárolódik. Ha a képernyő 640x480-as, akkor a (640, 480) a *screensize*-ban tárolódik.

Szükséges elvégezned még egy kis munkát, amiért a harmadik vektor, a *finaloffset* van. Alapvetően ez a vektor egy számított eltolást tárol, amely a vektor helyzetén és a képernyőméreten alapul. Például, akarsz valamit rajzolni (0, 0)-nál, és a kamerát is (0, 0)-ra akarod mutatni. A *Direct3D.Sprite* mindent (0, 0)-nál fog rajzolni a képernyő bal felső sarkában, de nem ezt akarod. Ahogy akarod a (0, 0)-t rajzolva a képernyő közepénél, mozgatnod kell a képernyőt félig balra és le. Tehát ha a képernyő 640x480-as, akkor ki kell vonnod 320 képpontot az *X* összetevőből és 240-et az *Y* összetevőből. Ezek az értékek a két számító függvénnyel számítódnak:

```
void CalculateXOffset()
{
        finaloffset.X = position.X - (screensize.X / 2);
}
void CalculateYOffset()
{
        finaloffset.Y = position.Y - (screensize.Y / 2);
}
```

A pozíció adat és a képernyő szélesség adat tulajdonságokon keresztül érhető el, mint ez:

```
public float X
{
	get { return position.X; }
	set { position.X = value; CalculateXOffset(); }
}
```

Jegyezd meg, hogy akármikor beállítasz egy új értéket, az eltolások újraszámítódnak, úgyhogy később csak az *Offset* tulajdonságot tudod hívni (mely csak olvasható), hogy megkapd a végső eltolási vektort. Felhasználását előzőleg az én *AdvancedFramework.Sprite.Draw* függvényemben láthattad.

A kameráknak vannak kötelező konstruktorai is a képernyő méretének és helyzetének beállítására:

```
AdvancedFramework.Camera c;
c = new AdvancedFramework.Camera( 640, 480, 100, 100 );
c = new AdvancedFramework.Camera( 640, 480 );
```

Az első kamera (100, 100)-ra mutat, a második pedig (0, 0)-ra.

Demo 7.5

Most, hogy van egy működő sprite és kamera osztályod, felépíthetsz egy programot bemutatni ezeket a képességeket. A Demo 7.5-öt erre hoztam létre.

Az adat

Ennek a demónak szüksége lesz némi adatok tárolására, kifejezetten a sprite-okat, egy kamerát, a Direct3D változókat, és néhány egyéb segédváltozót:

```
Direct3D.Texture texture = null; // a textúra
Direct3D.Sprite spriterenderer = null; // a sprite renderelő
Sprite[] sprites = null; // sprite-ok tömbje
int numsprites = 4; // sprite-ok száma
Camera camera = null; // a kamera
float movingx = 0.0f; // milyen gyorsan mozog a kamera x irányba?
float movingy = 0.0f; // milyen gyorsan mozog a kamera y irányba?
```

Ez a demó animált lesz és még interaktívabb, mint az előzők bármelyike, ezért beépítettem két lebegőpontos változót a kamera x és y irányba való mozgásának meghatározására.

Az adat beállítása

Mint az előző demóknál, az összes adat az *InitializeResources* függvényen belül állítódik be.

Először a textúra:

```
public void InitializeResources()
{
        texture = Direct3D.TextureLoader.FromFile(
        graphics, “texture.bmp”, 0, 0, 0, 0, Direct3D.Format.Unknown,
        Direct3D.Pool.Managed, Direct3D.Filter.Linear,
        Direct3D.Filter.Linear, Color.FromArgb( 0, 0, 255 ).ToArgb() );
```

Ez nagyon hasonló ahhoz, amit ezelőtt láttál, egy változással: hozzáadtam egy színkulcsát a tiszta kéknek. A textúra ehhez a demóhoz egy tégla bittérkép egy nagy kék körrel a közepén, és mivel a kék kört teljesen átlátszónak akarom, ezért a színkulcsot kékre állítottam; mikor a textúra betöltődik, a kék figyelmen kívül hagyódik.

Megjegyzés: azért, hogy színkulcsokat használj, egy veszteségmentes grafikus formátumot kell használnod, mint BMP-k vagy TGA-k. Nem használhatsz veszteséges formátumokat színkulcsnak, mint JPG-k, mert nem tárolják az adatot egy tiszta módban. Ha csináltál egy JPG-t egy tiszta kék körrel a közepén, akkor a JPG tömörítési folyamat enyhén megváltoztatná a színt.
Megjegyzés: két különböző képformátum létezik: veszteséges és veszteségmentes. Egy veszteségmentes képformátum a képadatot pontosan tárolja; minden képpont a képben úgy van eltárolva, ahogyan lennie kell. A veszteséges képformátumok viszont „veszítenek” információt. A JPG-k különösen hírhedtek erről. Egy jobb tömörítési arány megkapásához a veszteséges képformátumok *megközelítik*, hogy milyen színek léteznek a képben, és így kapsz egy olyan képet, amely jól néz ki az emberi szemnek – a pontos képpont információ nem tárolódik, mivel veszteséges formátumok. Akármikor használsz színkulcsokat, *muszáj* veszteségmentes formátumokat használnod.

Ezután a sprite renderelődik és a sprite tömb létrejön:

```
spriterenderer = new Direct3D.Sprite( graphics );
sprites = new Sprite[numsprites];
```

Aztán a sprite-ok inicializálódnak:

```
for( int i = 0; i < numsprites; i++ )
{
	sprites[i] = new Sprite();
	sprites[i].Texture = texture;
	sprites[i].Scale = (i+1) * 0.2f;
	sprites[i].X = i * 75.0f;
	sprites[i].Y = i * 75.0f;
	sprites[i].Angle = i;
	sprites[i].XAnchor = 0.5f;
	sprites[i].YAnchor = 0.5f;
}
```

Minden sprite ugyanazt a textúrát fogja használni és minden sprite különbözőképpen méreteződik. Például az első sprite 0.2-n lesz méretezve, a következő 0.4-en, a következő 0.6-on, és így tovább.

Ugyanez érvényes a pozícióra; az első (0, 0)-n, a következő (75, 75)-ön, a következő (150, 150)-en, és így tovább.

Minden sprite különböző szöggel is indul ki: 0 radián, 1 radián, 2 radián, és így tovább.

Végül minden sprite a középpontjánál horgonyzódik le.

Úgy döntöttem, legyen még néhány móka, és eljátszogatok néhány sprite színezésével is:

```
sprites[0].Color = Color.FromArgb( 127, 127, 255, 0 );
sprites[1].Color = Color.FromArgb( 127, 255, 0, 127 );
sprites[2].Color = Color.FromArgb( 127, 0, 127, 255 );
```

Az első egy vöröses-zöld, a következő egy kékes-piros, és az utolsó egy zöldes-kék. A negyedik sprite (index 3) nem színezett. Jegyezd meg, hogy az első

három sprite 50 százalékban átlátszó, melynek néhány csinos hatást kellene eredményeznie.

Végül a kamera is létrejön:

```
        camera = new Camera( screenwidth, screenheight );
}
```

Az adat animálása

Ez az első demó animáció használatára. Az animálási információ a *ProcessFrame* függvényen belül tárolódik:

```
if( !paused )
{
        float t = gametimer.Elapsed();
        camera.X += (movingx * t);
        camera.Y += (movingy * t);
        for( int i = 0; i < numsprites; i++ )
        {
                sprites[i].Angle += 1.0f * t;
        }
}
```

Az első dolog, amit a függvény csinál, hogy megkapja az utolsó képkocka óta eltelt időmennyiséget, és a *t* változóban tárolja.

A következőben a kamera elmozdul a *movingx* és *movingy* változókat használva. Például ha a *movingx* 100.0f-et tartalmaz, akkor ez azt jelenti, hogy a kamera 100 képpont per másodperccel jobbra fog mozogni.

Utolsó lépésben a ciklus végigmegy mind a négy sprite-on és elforgatja őket. 1 radián per másodperc arányon fognak forogni – amely megközelítőleg 57 fok másodpercenként – óramutató járásával ellentétes irányba.

Renderelés

A sprite-ok renderelése egyszerű folyamat; csupán végig kell menned a tömbön és kirajzolnod mindegyiket:

```
graphics.Clear( Direct3D.ClearFlags.Target, Color.White , 1.0f, 0 );
graphics.BeginScene();
spriterenderer.Begin();
for( int i = numsprites - 1; i >= 0; i— )
{
        sprites[i].Draw( spriterenderer, camera );
}
spriterenderer.End();
graphics.EndScene();
graphics.Present();
```

A sprite-ok elsőtől utolsóig renderelése helyett (amely a legkisebbet helyezné legalulra) a másik irányba megyek, és elsőnek az utolsó sprite-ot rajzolom. Ez lesz olyan, amit szép keverési hatásként láthatsz, melyet a sprite-ok adnak.

Minden sprite-nak van hívott *Draw* függvénye, használva a sprite renderelőt és a kamerát, a kódot csinossá és könnyen használhatóvá téve.

Interakció

Az utolsó változtatás a demóhoz az interaktívvá tétele. Minthogy még nincs a programba beépítve DirectInput, ezért beraktam egy nagyon kezdetleges bemeneti rendszert, felhasználva a Windows *OnKeyDown* és *OnKeyUp* eseményeit, melyeket láthattál előzőleg használva a keretrendszerben.

Hozzáadtam a következő sorokat az *OnKeyDown*-hoz:

```
if( e.KeyCode == System.Windows.Forms.Keys.W ) { movingy = -100.0f; }
if( e.KeyCode == System.Windows.Forms.Keys.S ) { movingy = 100.0f; }
if( e.KeyCode == System.Windows.Forms.Keys.A ) { movingx = -100.0f; }
if( e.KeyCode == System.Windows.Forms.Keys.D ) { movingx = 100.0f; }
```

Amikor a felhasználó lenyomja a W gombot a billentyűzeten, akkor a kamera elkezd mozogni fölfelé 100 képpontot másodpercenként. Hasonlóképpen az S lefelé mozgatja a kamerát 100 képpont per másodperccel, az A balra 100 képpont per másodperccel és a D pedig jobbra 100 képpont per másodperccel.

Mikor a felhasználó már nem nyomja le azokat a gombokat, a kameramozgásnak meg kell állnia, így hozzáadtam a következő sorokat az *OnKeyUp*-hoz:

```
if( e.KeyCode == System.Windows.Forms.Keys.W ) { movingy = 0.0f; }
if( e.KeyCode == System.Windows.Forms.Keys.S ) { movingy = 0.0f; }
if( e.KeyCode == System.Windows.Forms.Keys.A ) { movingx = 0.0f; }
if( e.KeyCode == System.Windows.Forms.Keys.D ) { movingx = 0.0f; }
```

Bármikor felengedődnek a gombok, a mozgás megáll.

Futtasd a demót és lásd a csodát

Végül már csak fordítanod és futtatnod kell a demót. Ebben a bilentyűzet W, A, S és D gombjaival mozoghatsz, a P-vel pedig szüneteltetheted a futását vagy folytathatod azt. Jegyezd meg, hogy az animáció akkor is meg fog állni, amikor kilépsz az ablakból egy másikba (az ön-szüneteltetés miatt a keretrendszerben).

Betűkészletek

A D3DX könyvtár egy ügyes *Font* osztályt is nyújt neked, hogy használd. A múltban el kellett volna készítened a saját betűkészlet textúráidat vagy menned kellett volna összezavarni a GDI-t, hogy rajzoljon betűkészleteket. Nos, ez többé már nincs így!

Egy rendszer betűkészlet létrehozása

A .NET keretrendszernek vannak beépített betűkészletei, használva a System.Drawing.Font osztályt. Ez az osztály használható bármilyen fajta betűkészlet jelképezésére, melyek rendszerint true-type fájlok (.ttf). Azért, hogy betűkészleteket rajzolj a játékodban, neked először hozzá kell férni egy rendszer betűkészlethez. Szerencsére ez hihetetlenül könnyű:

```
System.Drawing.Font sysfont;
sysfont = new System.Drawing.Font(
        “Arial”, 16,
        System.Drawing.FontStyle.Bold |
        System.Drawing.FontStyle.Italic );
```

Ez a kód létrehoz egy új objektumot, ami jelképezi az Arial betűkészletet, 16 pontnyi mérettel, használva egy félkövér és dőlt stílust. Kombinálhatod a flag-eket, használva a bináris VAGY műveleti jelet, ahogy a kódban is látható.

A következő táblázatban láthatók az elérhető stílusok:

Stílus	Leírás
Bold	A szöveg félkövér, vastag betűkkel
Italic	A szöveg dőlt
Regular	Nincs hatás
Strikeout	A szöveg középen áthúzott egy vonallal
Underline	A szöveg aláhúzott egy vonallal

Egy Direct3D betűkészlet létrehozása

Most, hogy van egy rendszer betűkészleted, előremehetsz és készíthetsz egy Direct3D betűkészletet:

```
Direct3D.Font font;
font = new Direct3D.Font( graphics, font );
```

Minden, amit tenned kell, átadni egy hivatkozást a grafikus eszköznek és a Windows betűkészlet objektumnak.

Szöveg rajzolása

Most, hogy be vannak állítva a betűkészlet osztályaid, belevághatsz a közepébe és elkezdhetsz szövegeket rajzolni. Csodálatos!

A szövegrajzolás a *Direct3D.Font DrawText* függvényét használja. Itt egy példa:

```
Rectangle rect = new Rectangle( 0, 0, 640, 100 );
dxfont.DrawText(
        “Szia ott!”,
        rect,
        Direct3D.DrawTextFormat.NoClip,
        Color.CornflowerBlue );
```

Ez a kód létrehoz egy téglalapot, ami jelképez egy dobozt a renderelt ablakod tetején, 640 képpont szélességben és 100 képpont magasságban. Ez a renderelő terület, amelybe a betűkészlet rajzolódik.

A következő lépés a „Szia ott!” szöveg rajzolása a téglalap belsejében, használva a *NoClip* szövegformázási lehetőséget és búzavirág kék színt.

Neked igazán csak a formázó lehetőségekkel kell tisztában lenned, melyek lehetővé teszik számodra, hogy elmondd a renderelőnek, hogy pontosan hogyan akarsz szöveget rajzolni. A következő táblázat mutatja a leghasznosabb lehetőségeket:

Hasznos *DrawTextFormat* értékek

Érték	**Célja**
WordBreak	Ha a szöveg hosszabb, mint a megadott doboz, ez a lehetőség feltöri a szöveget a következő sorra vagy sorokra, amennyi kitölti az egész szöveget.
VerticalCenter	A szöveget függőlegesen igazítja
Top	A téglalap tetejére rajzolja a szöveget (alapértelmezett)
Bottom	A téglalap aljára rajzolja a szöveget
SingleLine	Egy vonalba rajzolja a szöveget, az új sor karakterek kihagyásával.
NoClip	Alapértelmezésben a Font osztály a megadott téglalapba kapcsolja a szöveget, így a dobozon kívülre semmi nem rajzolódik. Ez letiltja ezt a tulajdonságot és a rajzolást gyorsabbá teszi.
Center	A szöveget vízszintesen igazítja
Right	A jobb margóhoz rajzolja a szöveget
Left	A bal margóhoz igazítja a szöveget (alapértelmezett)

Demo 7.6

A demó 7.6 változatos betűkészlet lehetőségeket mutat be, amiket használhatsz. Összességében, ez egy rendkívül egyszerű demó.

Az első lépés megadni a betűkészlet erőforrásokat:

```
Font windowsfont = null;
Direct3D.Font dxfont = null;
```

És aztán inicializálni az erőforrásokat, amikor a program betöltődik:

```
public void InitializeResources()
{
        windowsfont = new System.Drawing.Font(
        “Arial”, 16, System.Drawing.FontStyle.Bold );
        dxfont = new Direct3D.Font( graphics, windowsfont );
}
```

Végül minden, amit csinálok, egy csomó szöveg rajzolása a képernyőn a *Render* függvényen belül:

```
graphics.Clear( Direct3D.ClearFlags.Target, Color.White , 1.0f, 0 );
graphics.BeginScene();
```

```
Rectangle rect = new Rectangle( 0, 0, 640, 100 );
dxfont.DrawText(
        “Üdvözöllek a Demo 7.6-ban”,
        rect,
        Direct3D.DrawTextFormat.NoClip,
        Color.CornflowerBlue );


rect = new Rectangle( 0, 50, 640, 100 );
dxfont.DrawText(
        “Van itt még néhány szöveg! Középen!”,
        rect,
        Direct3D.DrawTextFormat.Center |
        Direct3D.DrawTextFormat.NoClip,
        Color.PaleVioletRed );

rect = new Rectangle( 0, 100, 300, 300 );
dxfont.DrawText(
        “Ez egy beillesztett szöveg bekezdése egy 300x300 dobozban!”,
        rect,
        Direct3D.DrawTextFormat.WordBreak |
        Direct3D.DrawTextFormat.NoClip,
        Color.MediumOrchid );

rect = new Rectangle( 0, 150, 640, 300 );
dxfont.DrawText(
        “Ugye izgalmas szöveg?”,
        rect,
        Direct3D.DrawTextFormat.Right |
        Direct3D.DrawTextFormat.NoClip,
        Color.LawnGreen );

graphics.EndScene();
graphics.Present();
```

Összegzés

Ez eddig a legnagyobb fejezet a könyvben, és van némi rossz hírem számodra: éppenhogy karcoltad a Direct3D felszínét. Igazán nem vicceltem, amikor azt mondtam, hogy a Direct3D egyike a legnagyobb és legösszetettebb API-knak a világon. Hihetetlen mennyiségű dolog van az API-n belül, amelyek lehetővé teszik számodra, hogy készíts néhány nagyszerű játékot.

Tonnányi dolog van, amelyeket bemutatni egyszerűen nincs elegendő helyem, mint pl. csúcspont pufferek, index pufferek, fény, 3D átalakítás számítások,

mesh-ek, csúcspont shader-ek, pixel shader-ek, kötet textúrák, köd, mélység pufferek, kulcsképkocka animációk, pont sprite-ok, részecskék… De ne riadj el! Már elég jó fejed van az elinduláshoz és elkezdheted csinálni a saját 2D-s játékaidat, ha akarod. A könyv következő fejezete az adatbevitelen és a hang programozáson megy végig, melyek kisebb (de azért nagyon fontos) részei a játékprogramozásnak.

8. fejezet

DirectInput

A 6. és 7. fejezetben már láttál néhány kezdetleges adatbeviteli programozást. Azok a fejezetek Windows eseményeket használtak annak meghatározására, hogy a billentyűzeten le volt-e nyomva egy gomb. Míg az adatbevitel íly módon való kezelése nagyszerű az olyan dolgoknak, mint egy Windows alkalmazás, egy játék környezethez már nem igazán illik. A játékok jellemzően az adatbevitel gyors kezelését igénylik és további ellenőrzést a pillanatnyi eszközök felett – ellenőrzést, amit a szokásos Windows események egyszerűen nem adnak meg neked. Ez az amiért van a DirectInput, a DirectX API, amely a lehető legtöbb ellenőrzést adja egy adatbeviteli eszköz felett.

Billentyűzetek

A billentyűzet egyike volt a legkorábban elérhető számítógépes adatbeviteli eszközöknek. Manapság nehéz egy billentyűzet nélküli PC-t találni, és valószínűleg feltételezed, hogy bárki, aki egy PC játékkal játszik, annak van egy billentyűzete.

A Haladó Keretrendszer a demó 7.1-ből magában foglal olyan kódot, ami létrehoz egy billentyűzetet, de én nem mentem végig annak összes jelentésén.

Mint a Direct3D-ben, a DirectInput API-nak van egy *Device* osztálya, amely jelképezi a jelenlegi adatbeviteli eszközöket:

```
DirectInput.Device keyboard = null;
```

Egy eszköz létrehozása

Amint létrehoztál egy változót az eszközöd tárolására, szükséges létrehoznod az eszközt:

```
keyboard = new DirectInput.Device( DirectInput.SystemGuid.Keyboard );
```

Az eszköz (*Device*) létrehozója fog egy *globális egyéni azonosító*-t (*globally unique identifier* – GUID) a paramétereként. Egy GUID egyszerűen egy 128-bites szám, amely minden eszközhöz hozzá van rendelve az eszközödön. Mivel egy billentyűzet és egy egér feltételezhetően létezik egy PC rendszeren, a DirectInputnak van egy osztálya, amely visszatér a billentyűzet és egér eszközök GUID-jával neked; az osztálynak *DirectInput.SystemGuid* a neve. A *Keyboard* tag visszatér a rendszer billentyűzet GUID-jával, a *Mouse* tag pedig a rendszer egér GUID-jával.

A következő lépés az eszköz együttműködési szintjének beállítása:

```
keyboard.SetCooperativeLevel(
        this,
        DirectInput.CooperativeLevelFlags.Background |
        DirectInput.CooperativeLevelFlags.NonExclusive );
```

Egy közös téma, amit illene látnod a játékprogramozásban, hogy *nem te tulajdonlod a gépet, amin futtatod*. Az a billentyűzet a felhasználóhoz tartozik, nem a programodhoz. Ez azt jelenti, hogy el kell mondanod a DirectInput-nak, hogy hogyan akarsz hozzáférni az eszközhöz. Az alábbi táblázat listázza az összes használható flag-et.

Jó ötlet a *NonExclusive* flag-gel létrehozni az eszközöket, mint billentyűzet és egér, mert így minden más által használhatók. Ha nem akarod, hogy a programod kiragadja az adatbevitelt, amíg az ablakod kis méretre van minimalizálva, akkor a *Foreground* módot kellene használnod. Sajnos, ha ezt csinálod, akkor az eszközeid ismeretlenné válnak bármikor valaki kikapcsol a programodból, és ezért a programodnak vissza kell azokat kapcsolnia magától. Én a *Background* módot részesítem előnyben, és nem gyűjteni adatbevitelt amíg az alkalmazás kis méretben van.

A következő lépés az együttműködési szint beállítása után az eszköz megszerzése:

```
keyboard.Acquire();
```

És most használhatod a billentyűzetet a Windows esemény üzenetek helyett a DirectInput-tal.

Zászló	Jelentése
NoWindowsKey	A Windows gomb letiltva (csak billentyűzetek)
Background	Az eszköz hozzáférhető, még ha az alkalmazásod a háttérben is van
Foreground	Az eszköz csak akkor használható, ha az alkalmazásod aktív
Exclusive	A program kizárólagosan kéri használni az eszközt
NonExclusive	Az eszköz megosztható a többi program között

Összegyűjtő adatbevitel lekérdezéssel

Két módja van annak, hogy összegyűjtsünk adatbevitelt egy billentyűzetről. Az első módszer a billentyűzet lekérdezése által, hogy „Mely gombok vannak éppen most lenyomva?" A másik módszer egy esemény bejelentő használata; ez nagyon hasonló ahhoz, amit már láttál a Windows *OnKeyDown* eseményeivel. (Nem szándékozom belemélyülni a második módszerbe, mert ez magába foglalja a *többszálúság*ot, mely téma lefedéséhez nincs elég hely ebben a könyvben.)

Itt van, hogy hogyan kell lekérdezni a billentyűzetet:

```
DirectInput.Key[] keys = keyboard.GetPressedKeys();
```

A *GetPressedKeys* függvény csakis billentyűzetekkel működik, és visszatér a *DirectInput.Key*-ek tömbjével, mely csupán egy felsorolás, amely különböző gombokat jelképez a billentyűzeten. Néhány példa a *DirectInput.Key* értékeire: *DirectInput.Key.Return*, *DirectInput.Key.Q, DirectInput.Key.Numpad6*, és így tovább. Rengeteg van belőlük.

Egerek

Az egerek csodálatosan hasznos eszközök, és mint a billentyűzet esetében, úgy e nélkül sem igazán találsz már számítógépet. Az egereket csak egy kicsit nehezebb használni a billentyűzeteknél.

Az egerek különböznek a billentyűzetektől abban, hogy azt tárolják, milyen távolságban mozogtak egy 2D rácsban, tehát minden alkalommal, mikor lekérdezel egy egeret, tájékoztatást kapsz arról, hogy mely gombok nyomódtak le és mennyit mozdult el a legutóbbi lekérdezés óta.

Egy egér létrehozása

Egy egér létrehozása szintén hihetetlenül könnyű, és a folyamata majdnem azonos azzal, amit használtál egy billentyűzet eszköz létrehozásakor. A következő kód feltételezi, hogy *mouse* egy *DirectInput.Device*:

```
mouse = new DirectInput.Device( DirectInput.SystemGuid.Mouse );
mouse.SetCooperativeLevel(
        this,
        DirectInput.CooperativeLevelFlags.Background |
        DirectInput.CooperativeLevelFlags.NonExclusive );
mouse.SetDataFormat( DirectInput.DeviceDataFormat.Mouse );
mouse.Acquire();
```

Az egyetlen sor, amely jelentősen különbözik a billentyűzet kódtól, félkövér. Az a sor elmondja, hogy az adatokat fogadni szándékozó eszköz egy egér objektum számára van létrehozva.

Egy egér lekérdezése

Egy egér objektum lekérdezhető a *Device CurrentMouseState* tulajdonsága visszakeresése által, amely visszatér egy *DirectInput.MouseState* objektummal:

```
DirectInput.MouseState state = mouse.CurrentMouseState;
```

Az állapot szerkezetnek lesz néhány változója, amely érdekelhet:

```
int deltax = state.X; // mennyi egység egér mozdult vízszintesen
int deltay = state.Y; // mennyi egység egér mozdult függőlegesen
int deltaz = state.Z; // mennyi egység egér-kerék mozdult
byte[] buttons = state.GetMouseButtons();
if( buttons[0] == 128 )
```

```
        // 0 gombja lenyomva
if( buttons[1] == 128 )
        // 1-es gombja lenyomva
// és így tovább...
```

Az *X*, *Y* és *Z* változók visszatérési értéke integer típusú szám, amely lehet pozitív vagy negatív. Pillanatnyi jelentése függ a rendszertől, tehát változó; ez az, amiért rendszerint jó ötlet lehetővé tenni a játékosnak, hogy beállíthassa az egér érzékenységet (a delta értékek némi állandóval (konstans) való szorzásával).

Gomb állapot információ megkapásához egyszerűen csak hívd az állapotszerkezet *GetMouseButtons* függvényét, mely visszatér bájtok egy tömbjével. Minden bájt egy gombot jelképez, és a gomb két állapot egyikében lehet: 0 vagy 128. A 0 azt jelenti, hogy a gomb nincs lenyomva, a 128 pedig hogy le van. Ezzel minden megvan, amit tudni érdemes.

Játék eszközök

Az adatbeviteli eszközök másik fő típusát általánosan játék eszközöknek nevezik. Egy *játék eszköz* egyszerűen egy játékok számára tervezett eszköz, ami nem egy egér vagy egy billentyűzet. Ez lehet egy botkormány (joystick), egy kormánykerék, egy játék pad (játékvezérlő), pedálok, vagy bármi más, amire gondolhatsz.

A játék eszközök hasonlóak az egerekhez – van egy vagy több tengelyük és számos gombjuk. De nagyon összetettek is lehetnek, és magukba foglalhatnak olyan változatos vezérlést, ami az egérnek nincs, beleértve csúszkákat, forgató tengelyt, három alapértelmezett tengelyt, gombokat, és még a virtuális valóság szemüveget is. Ebben a fejezetben a *joystick* kifejezést fogom használni a játék eszközök bármely típusára.

Egy játék eszköz keresése

Mivel nem minden számítógépnek vannak játék eszközei, ezért nem tételezheted fel, hogy mindig lesz elérhető játék eszközöd. És az USB csatlakozók megjelenése óta nem tételezheted fel azt sem, hogy a rendszernek csak egy játék eszköze lesz; számos eszközt találhatsz egy gépen.

Szükséged van valami módszerre egy eszköz GUID visszanyerésére a számítógépből. Szerencsére a *DirectInput* nyújt egy ügyes *Manager* osztályt, amely összegyűjti neked az elérhető eszközök egy listáját. Itt egy példa:

```
foreach( DirectInput.DeviceInstance i in
        DirectInput.Manager.GetDevices(
        DirectInput.DeviceClass.GameControl,
        DirectInput.EnumDevicesFlags.AttachedOnly ) )
{
// csinálni itt valamit az “i” változóval
}
```

A *Manager.GetDevices* függvény visszaadja az eszközpéldányok (*DeviceInstance*) egy felsorolását, melyeket megkap az általad beadott paraméterektől függően. Az első paraméter meghatározza, hogy milyen eszközosztályt akarsz megtalálni; ez lehet egy *Pointer* (egy egér eszköz), egy *Keyboard* (billentyűzet), egy *GameControl* (botkormány-típusú eszköz), *Other* (rajzoló tábla vagy más csodabogarak) vagy *All* (összes).

A következő paraméter meghatározza, hogy mely flag-eknek kell lenniük az eszközöknek. Például én az *AttachedOnly* flag-et használtam, mely csak a csatlakoztatott eszközöket keresi. Ha nem használod ezt a flag-et, akkor a függvény visszatért volna egy mutatóval egy botkormányra, amit a felhasználó kihúzott – nem akarsz olyan eszközöket használni, amelyek jelenleg nincsenek csatlakoztatva a rendszerhez.

A másik két fontos flag az *AllDevices* és a *ForceFeedback*. A *ForceFeedback* flag csak azokkal az eszközökkel tér vissza, amelyeknek van erő visszacsatolás hatásuk; a visszacsatolás hatásokra később még visszatérek ebben a fejezetben.

Egy játék eszköz létrehozása

Amint találtál egy olyan eszközt, amit használni akarsz, létrehozhatod azt, ahogy egy egeret vagy billentyűzetet is létrehoznál:

```
gameinput = new DirectInput.Device( i.InstanceGuid );
gameinput.SetCooperativeLevel(
        this,
        DirectInput.CooperativeLevelFlags.Background |
        DirectInput.CooperativeLevelFlags.NonExclusive );
gameinput.SetDataFormat( DirectInput.DeviceDataFormat.Joystick );
```

```
gameinput.Acquire();
```

Az előző kódszelet a *foreach* blokkon belül elhelyezve azt jelentette, hogy találjon eszközöket, így az *i* egy *DirectInput.DeviceInstance* tárgy, amely egy csatlakoztatott játékvezérlőt jelképez. Ezen kívül semmi új nincs a kódban.

Botkormány tengely adat szerzése

Mivel olyan sokféle botkormány létezik, nem igazán tudod, hogy milyen tengelyek, csúszkák, gombok, stb. vannak egy egyéni eszköznél. A *JoystickState* szerkezetnek vannak változói minden elképzelhető dolog tárolására. Az alábbi táblázat listázza néhány tulajdonságát a botkormány állapot szerkezetnek:

Tulajdonság	**Leírás**
X	X tengely abszolút helye
Rx	X tengely forgási szöge
ARx	X tengely szögletes gyorsulása
AX	X tengely gyorsulása
FRx	X tengely forgatónyomatéka
FX	X tengely erő
VRx	X tengely szögletes sebessége
VX	X tengely sebessége

Felcserélheted X-et Y vagy Z ezen változóival, hogy megkapd ugyanezen információkat az Y és Z tengelyekre vonatkozóan.

Milyen sok információ! A lehetőségek megvannak, de nem mindegyikükre lehet szükséged. Legtöbbször csak az *X*, *Y* és *Z* változók kellenek majd, esetleg még az *Rx*, *Ry* és *Rz* változók. Mindez attól függ, hogy hogyan működik a joystickod.

Az én botkormányomnál (Logitech Wingman Force 3D) például az *X* és *Y* változók jelképezik a bal-jobb és fel-le elhelyezkedését a rúdnak, az *Rz* a bal-jobb forgását és a *Z* nem használt. De egy másik, azonos márkájú joystickom a *Z* változót a gáz csúszóka jelképezésére használja. Minden a saját, egyéni botkormánytól függ.

Tipp: jó ötlet lehetővé tenni a játékosaid számára, hogy beállíthassák saját botkormányaikat a programodban.

A DirectInput öt tengely adatig kezeli az eszközöket, de csak az első három tengelyt tudod megkapni az eddig ismert tulajdonságokkal. Azért, hogy

megkapd a további két tengelyt (rendszerint *u* és *v* tengelyeknek hívják őket), néhány egyéb függvény használatára van szükséged. A függvény, amellyel főleg érintett leszel, a *GetSlider*, amely két integer típusú szám tömbjével tér vissza. Az első ezek közül az *u* tengely abszolút helyzetét jelképezi, míg a második a *v* tengelyét.

Három további függvény, amelyre valószínűleg nem lesz szükséged, a *GetASlider*, a *GetFSlider* és a *GetVSlider*, melyek visszatérnek két integer szám tömbjével, amik a gyorsulását, az erőt és a sebességét jelképezik a két csúszókának.

Megjegyzés: különösnek tűnik a mód, ahogy az XYZ tengelyekhez való hozzáférés teljesen különbözik az UV tengelyekhez való hozzáféréstől, és nincs aki igazán biztos benne, hogy miért van ez. Valószínűleg valami elnézés a DirectX csapat részéről. Ők tisztában vannak ezzel az ellentmondással, és ez valószínűleg az API jövőbeni változatában lesz javítva.

Tengely attribútumok módosítása

A DirectInput lehetővé teszi, hogy módosítsd a tengely attribútumokat, és ez sok helyzetben hasznossá válik, így finom hangolhatod eszközeid használatát.

A hatótávolság módosítása

Ha beindítasz egy programot és elkezdesz tengelyadatokat nyerni egy joystickból, akkor valószínűleg 0-tól 65,535-ig terjedő értéktartományba eső adatot kapsz, de ez nem garantált. Továbbá, legtöbbször az adatoknak ebben a tartományban nincs teljesen értelmük, mert a tengelyek zéró-helye 32,768-ként fog feltűnni, ami nem igazán hasznos. Egy tengely zéró-helye rendszerint azt jelenti, hogy egyáltalán nem mozogsz, tehát azt gondolnád, hogy annak az értéknek 0-nak kellene lennie, ugye? Nos, nem. De megváltoztathatod az értékeket, hogy egy tengely visszatérjen hozzád.

Minden botkormány alkotóelemnek tulajdonképpen van egy *DirectInput.DeviceObjectInstance* szerkezete, őt jelképezve. Ez azt jelenti, hogy minden tengelynek, minden gombnak, minden POV kalapnak, és minden csúszkának van leírása ezen szerkezetek egyikében. Nem igazán leszel érintett ezzel az összes adattal, de ha valaha is akarod, ez mind a rendelkezésedre áll.

Használhatod ezen eszközobjektum példányokat minden tengely hatótávolságának módosítására. Például ha a joystick hatótávolságát -10,000-től 10,000-ig akarod 0-65,535 helyett, akkor beállíthatod, használva a *DirectInput.Device.Properties.SetRange* függvényt.

Az első lépés felsorolni az összes eszköz objektum példányt, hogy megtaláld az összes tengelyt:

```
foreach( DirectInput.DeviceObjectInstance inst in gameinput.Objects )
{
```

Ez felsorol minden példányt egy *foreach* cikluson belül, az *inst* változót használva minden példány jelképezésére. A *DirectInput.Device.Objects* tulajdonság visszatér ezen példányok egy listájával.

Minden példánynál szükséges egy ellenőrzés, hogy megbizonyosodj arról, hogy egy tengely:

```
    if( (inst.ObjectId & (int)DirectInput.DeviceObjectTypeFlags.Axis) != 0 )
    {
```

Használva az *ObjectId* adatot bináris *és* kapcsolattal a *DirectInput.DeviceObjectTypeFlags* értékkel, ellenőrizheted a többi tulajdonságot, mint hogy gomb-e (*Button*) vagy egy *Pov*, vagy más dolgok bármely száma, amelyekkel valószínűleg nem törődsz.

Amint megtudtad, hogy van egy tengelyed, beállíthatod annak hatótávolságát:

```
    gameinput.Properties.SetRange(
            DirectInput.ParameterHow.ById,
            inst.ObjectId,
            new DirectInput.InputRange( -10000, 10000 ) );
```

Ez valami zavaros, mert nincs megengedve az, hogy csakúgy megváltoztasd az objektum példányt magad. Helyette hozzá kell férned a *DirectInput.Device.Properties.SetRange* függvényhez, hogy beállíts egy új értéktartományt, használva az objektum példány *ObjectId* tulajdonságát. Láthatod, hogy a kód beállítja a bemeneti tartományt -10,000-től 10,000-ig, jelentvén hogy most minden tengelynek az lesz a hatótávolsága, 0

középponttal, melynek sokkal több értelme van az alapértelmezett tartományhoz képest.

Megjegyzés: az ilyen értéktartomány használata egy botkormány hagyományos tengelyeinek csinál értelmet, de van egy kivétele a szabálynak: a gáz csúszkák. Ezeket jellemzően 0-tól valami nagy számig (esetleg 10,000? Tőled függ) terjedő értékig akarod, mintsem valami nagy negatív számtól egy másik nagy számig, mert a való világban nincsen olyan dolog, hogy „negatív gáz". A választás, mint mindig, a tiéd.

A holt zóna

Illene tisztában lenned a *holt zóná*val is. Nem, én nem egy Stephen King könyvről beszélek; én egy tengely tényleges tulajdonságáról ejtek szót. A holt zóna egy övezet, melyben egy egyéni tengely zéró-értékkel tér vissza, még akkor is, ha tulajdonképpen nem zéró.

Ha tudod, hogyan működik egy joystick, akkor az eszköznek rendszerint van egy rúdja a középpontban; ha elmozdítod bármerre és hagyod elmozdulni, akkor egy rugó visszaállítja középre önműködően. Eszményien a pontosan 0 értéket akarod a középpontnak lenni, amikor a botkormány nem mozog, de a valóságban ez ritka eset. Tökéletlenségek az anyagban és más tényezők okozhatják a joysticknak, hogy majdnem mindig 0-tól eltérő értékei vannak, amikor nem érsz hozzá. Még akkor is, ha az értékek kicsik, attól még nem-nullák, és ettől a játék úgy érzékeli, hogy a felhasználó mozgatja a botkormányt. Ez ritkán egy jó dolog.

Ez az, amiért a *holt zóna* bejön. Beállíthatod bármire, amire akarod; egy kb. 15 százalékos érték általában jó. Állítsd be, ahogyan a tengelyeid hatótávolságát:

```
gameinput.Properties.SetDeadZone(
        DirectInput.ParameterHow.ById,
        inst.ObjectId,
        1500 );
```

A beadott érték 0-tól 10,000-ig terjedhet, ahol 10,000 képviseli a teljes értéktartományát a vezérlőnek. Az előző kóddarab egy 15 százalékos értéktartományt mutat.

További joystick adat

Eddig csak azt tudod, hogyan nyerj adatot egy botkormány különböző tengelyeiről, de vannak még további bemeneti objektumok egy joystickon, amelyekről kaphatsz információt, beleértve a gomb értékek és POV kalap értékek.

POV kalapok

A POV kalapok olyanok, mint a gombok, kivéve hogy különböző irányokba mozoghatnak; általában négy, nyolc vagy tizenhat különböző értékük van. Egy négy értékűn az irányok általában észak, kelet, dél és nyugat. Egy nyolc irányún továbbiak az északkelet, északnyugat, délkelet és délnyugat, míg egy tizenhat irányún megkapod az észak-északkelet, nyugat-délnyugat, stb. irányokat is.

A *DirectInput.JoystickState.GetPointOfView* függvény visszatér integer számok egy tömbjével, mindegyik egy kalap értéket jelképezve. A DirectInput 9.0 úgy tűnik, legfeljebb négy különböző kalapot tud beállítani, ami megváltozhat, tehát a legjobb, ha meghatározod, hogy mennyi kalap használható egy eszköz képességek megtekintése által. A következő demóban megmutatom neked, hogyan csináld.

Minden egyes POV kalapnak az érték jellemzően egy fok érték. Ha ez -1, akkor azt jelenti, hogy a kalap érintetlen (vagy nem létezik). Egy négy irányú kalapnak az értékek jellemzően 0 (észak), 9000 (kelet), 18000 (dél) és 27000 (nyugat). Jegyezd meg, hogy ezek durván viszonyítanak 100-zal szorzott fokszögekre, néhány kisebb különbséggel: óramutató járásával megegyezően mennek, ellentétben mint ahogy fokok szokták, és a 0 fok északot jelent, nem keletet.

Megjegyzés: néhány botkormány illesztő program a 65,535 értéket használja -1 helyett, hogy jelezze azt, hogy a kalap nincs lenyomva. Keress utána, ha kell.

Gombok

A gombok egy joystickon nagyon egyszerűek, és olyan a hozzáférés, akárcsak az egérgombokhoz. A *DirectInput.JoystickState.GetButtons* függvény visszatér bájtok egy tömbjével, amiben minden bájt 0 vagy 128 lehet, és 128 jelenti, hogy a gomb le van nyomva, a 0 pedig hogy nincs lenyomva.

Botkormány demó

Ez a legösszetettebb az összes demó közül eddig ebben a fejezetben, egyszerűen azért, mert a botkormányok a legösszetettebb eszközök. Be szándékozom mutatni neked a *ProcessInput* függvényt, amely kinyeri az összes elérhető joystick adatot, és kiírja szövegesen, hogy elmondja neked pontosan, mi folyik éppen.

A legelső dolog, hogy a függvény megkapja az eszköz állapotát és képességeit:

```
protected virtual void ProcessInput()
{
        // az eszköz állapotának megkapása
        DirectInput.JoystickState state = gameinput.CurrentJoystickState;
        DirectInput.DeviceCaps caps = gameinput.Caps;
        gameinputinfo = “Device Information:\n”;
        gameinputinfo += “ Type: “ + caps.DeviceType.ToString() + “\n”;
        gameinputinfo += “ Axes: “ + caps.NumberAxes.ToString() + “\n”;
        gameinputinfo += “ Buttons: “ + caps.NumberButtons.ToString() + “\n”;
        gameinputinfo += “ POVs: “ + caps.NumberPointOfViews.ToString() + “\n”;
        gameinputinfo += “ Force Feedback: “ + caps.ForceFeedback.ToString() + “\n\n”;
```

A *gameinput* változó egy *DirectInput.Device*, amely a botkormányodat jelképezi.

Egy eszköz állapot szerkezetet használva kinyerheted az eszköz típusát (*Keyboard*, *Mouse*, *Joystick*, *Gamepad*, *Driving* vagy más értékek bármely száma). Aztán visszanyerheted a tengelyek számát a rúdon. Legfeljebb öt tengely lehet egy eszközön; jellemzően az első három az X, Y és Z tengelyek, az utolsó kettő pedig az U és V tengelyek.

A következőben a demó lekérdezi a gombok és a POV kalapok számát, illetve hogy az eszköz támogatja-e vajon az erő visszacsatolást.

Miután megmutattad az összes lehetőséget, megkaphatod az eszközállapotot:

```
        gameinputinfo += “X: “ + state.X.ToString() + “ “;
        gameinputinfo += “Rx: “ + state.Rx.ToString() + “ “;
        gameinputinfo += “ARx: “ + state.ARx.ToString() + “ “;
        gameinputinfo += “AX: “ + state.AX.ToString() + “ “;
        gameinputinfo += “FRx: “ + state.FRx.ToString() + “ “;
        gameinputinfo += “FX: “ + state.FX.ToString() + “ “;
        gameinputinfo += “VRx: “ + state.VRx.ToString() + “ “;
        gameinputinfo += “VX: “ + state.VX.ToString() + “\n”;
```

Az idő nagy részében a legtöbb változó 0 lesz, ahogy említettem korábban. Az első változó, az *X*, a legfontosabb. A kód megkettőzött még kétszer, egyszer az Y és egyszer a Z tengelynek, úgyhogy ezeket nem szükséges neked megmutatni.

A kód következő része megkapja az U és V tengely információt:

```
int[] sliders = state.GetSlider();
int[] sa = state.GetASlider();
int[] sf = state.GetFSlider();
int[] sv = state.GetVSlider();
for( int i = 0; i < sliders.Length; i++ )
{
        gameinputinfo += “Slider “ + i.ToString() + “: “;
        gameinputinfo += “Position: “ + sliders[i].ToString();
        gameinputinfo += “ Acceleration: “ + sa[i].ToString();
        gameinputinfo += “ Force: “ + sf[i].ToString();
        gameinputinfo += “ Velocity: “ + sv[i].ToString() + “\n”;
}
```

Tapasztalatom szerint az első tömb (*sliders*) az egyetlen, amivel érintett leszel; jellemzően a többi üres. De bemutatási célokból lekérdeztem az összes tömböt, és végigmentem ciklusosan mindegyiken, mutatva mindegyik csuszka értéket.

Most lekérdezzük a POV kalap információt:

```
int[] POV = state.GetPointOfView();
for( int i = 0; i < POV.Length; i++ )
{
        gameinputinfo += “POV “ + i.ToString() + “: “ +
        POV[i].ToString() + “\n”;
}
```

Ez egyszerűen végigfut ciklusosan a POV értékek tömbjén, és kiírja mindegyiket.

Végül, mutatjuk a gomb információt:

```
byte[] buttons = state.GetButtons();
gameinputinfo += “Buttons: “;
for( int i = 0; i < buttons.Length; i++ )
{
        if( buttons[i] == 128 )
        gameinputinfo += “ Button “ + i.ToString();
}
```

Ha egy gomb le van nyomva, akkor a „Button X” íródik ki (ahol X a gomb száma), egyébként semmit nem ír ki a képernyőre.

Erő visszacsatolás

Mikor gyerek voltam, volt egy *After Burner* nevű játéktermi játék. Ez egy igazán menő gép volt, amibe beülhettél, és amikor körbemozgattad a vezérlőt, akkor az egész gép körbemozdult, azt az érzést adva, hogy valójában egy repülőgéppel repülsz. Lehet, hogy nem adta vissza pontosan azt a 6G-s érzést, amit egy igazi vadászrepülő nyújt, amikor egy óra alatt ezer mérföldet repülsz vele, de még ez a korlátozott mozgás is sokkal valósághűbb volt, mint amit azelőtt éreztem vagy láttam. Amikor rakéta- vagy lövedéktalálatot kaptál, az egész gép megremegett, és igazán úgy érezhetted, mintha eltalált volna valami. Ezért voltak a játéktermek sokkal jobbak, mint hazamenni és játszani a játékot Sega Genesis-en – jobban belemerültél az élménybe.

Az otthoni számítógépen és videójáték konzolokon játszott játékoknak nem volt ilyen teljes élményük az utóbbi évekig, mikor a Nintendo elkészítette a Rumble Pak-et, egy kis eszközt a N64 vezérlődben, ami vibrált bizonyos események megtörténtekor, mint pl. ütközéskor. A Nintendo képessé tette a N64-et, hogy adatot küldjön vissza a vezérlőnek, így az többé vált, mint egy egyszerű adatbeviteli eszköz.

Ez a rezgő hatás volt egy hatalmas forradalom kezdete a játék adatbeviteli technológiában, amit *erő visszacsatolás*nak hívtak. Az erővisszacsatolás ötlete az, hogy tegyük a játékvezérlőt interaktívvá – adat visszaküldése által a felhasználónak. Ha repülsz egy sugárhajtású repülővel és az oldalkormányaid megsérültek, akkor a botkormány adhat neked némi ellenállást, amikor megpróbálod felhúzni. Ha egy légturbulenciába repülsz, akkor a vezérlő rázkódni fog, vagy ha egy göröngyös úton vezetsz, akkor a kormánykereked föl és le mozog.

Az erővisszacsatolás sokkal valósághűbbé teszi a játékaidat, és a DirectInput hihetetlenül könnyen használhatóvá teszi ezt.

A hatás szerkesztő

A DirectX SDK egy fedit.exe nevű programmal jön, amely lehetővé teszi neked, hogy létrehozz változatos erővisszacsatolás hatásokat és elmentsd a lemezre ezeket. Ez igazából egy egyszerű program, és könnyebbé teszi az életedet; sok beépített hatással is jön. A hatások FFE (*force feedback effects*) állományokban tárolódnak.

A szerkesztő támogatja hatások számos különböző fajtáját; a legegyszerűbb az *állandó erő* (*constant force*) hatás, mely alkalmaz egy egyszerű állandó erőt a kiválasztott irányba.

Ha beilleszted ezen hatások egyikét, akkor az alapértelmezésben északra mutat és a maximum erőt használja. Ha megnyomod a Play (Lejátszás) gombot, akkor látni fogod, hogy az eszközöd botja azonnal visszafelé mozog és ott is marad a hatás végéig. Megváltoztathatod a hatás irányát, ha belemész a tulajdonságaiba és szórakozol a beállítási lehetőségeivel.

Egy csomó másféle hatástípus is van, mint például az emelkedő hatás, amely lineárisan mozgatni fogja a botot idő alapján egy irányból egy másikba, vagy változatos ismétlő minták (szinusz, fűrészfog, négyszög), amelyek egy egyéni algoritmusnak megfelelően mozgatják a botot előre-hátra.

Aztán vannak hatások, amelyek nehézzé teszik neked a fogantyú mozgatását, mint a súrlódás, a csappantyú, és így tovább. Csak játssz el velük és lásd, hogy mit tudsz csinálni.

Betöltési hatások

A DirectInput lehetővé teszi számodra, hogy betölts hatásokat egy állományból, és játssz velük, de a folyamat kissé nehéz, ezért létrehoztam egy segítő osztályt, ami megkönnyítheti a dolgodat. Az osztály *AdvancedFramework.ForceEffect* néven található a Demo 8.4 mappájában.

Lényegében egy FFE állomány hatások egy listáját tartalmazza, amiben minden hatás az előzőekben általam említett hatások akármelyike lehet, mint egy szinusz hullám hatás, vagy egy állandó erő hatás. Tehát a *ForceEffect* osztály tárolni fogja ezen hatások egy tömbjét, ahogy a hatás nevét is:

```
public class ForceEffect
{
	System.Collections.ArrayList effectlist;
	string name;
	public string Name
	{
		get { return name; }
	}
	<code snipped>
}
```

Amint megvan az összes, át tudod adni egy FFE állomány fájlnevét a konstruktorba, ahogy a hatás nevét is, és azt az eszközt, amelyet használni fogsz:

```
public ForceEffect( string filename, string name, Device device )
{
	// a név mentése
	this.name = name;
	effectlist = new System.Collections.ArrayList();
```

Ez csak inicializálja a nevet és a hatások listáját.

A következőben betöltöd a hatásfájlok egy listáját a lemezről:

```
	EffectList effects;
	effects = device.GetEffects( filename, FileEffectsFlags.ModifyIfNeeded );
```

Egy *EffectList* csupán hatásfájlok egy listája, melyek szerkezetek, amik leírnak egy hatást egy FFE állományon belül. Amint megvan az a lista, rá kell nézned minden egyesre, és betöltened egy *EffectObject* objektumba:

```
	foreach( FileEffect e in effects )
	{
		EffectObject effect = new EffectObject(
		e.EffectGuid, e.EffectStruct, device );
		effectlist.Add( effect );
	}
}
```

Most az *effectlist* magában tartalmaz egy csomó *EffectObject*-et, és az jelképezi az egész FFE állományt.

Lejátszó hatások

Amint van egy hatáslistád betöltve, kell valami mód, hogy tulajdonképpen lejátszd a hatásokat; ezt a *Play* függvénnyel teheted meg:

```
public void Start( int iterations, bool restart )
{
	foreach( EffectObject effect in effectlist )
	{
		if( !effect.EffectStatus.Playing || restart )
		{
			effect.Start( iterations, 0 );
		}
	}
}
```

A függvény fog két paramétert: az ismétlések (*iterations*) számát, és hogy újraindítottnak kell-e vagy sem lennie a hatásnak. A listán belül minden *EffectObject*-nek a függvény ellenőrzi, hogy a hatás lejátszódik-e már. Ha így van és nem akarod újra beállítani, akkor a hatás nem játszódik le; különben igen. A *Start* függvény hívása egy hatáson önműködően újra beállítja azt, tehát ezért kell azt ellenőrizned.

A *Start* függvény első paramétereként azt a számot kapja, ahányszor ismételni akarod a hatást, és egy flag-et másodiknak. Csak két érvényes flag van: *EffectStartFlags.NoDownload*, amely letiltja a hatások önletöltését az eszközbe (nem vagyok benne biztos, hogy miért akarná ezt bárki is), és *EffectStartFlags.Solo*, amely leállít mindent, amit az eszköz lejátszik, és elindítja az új hatást. Valószínűleg nem fogod használni ezek egyikét sem, de jó ha tudod, hogy léteznek.

Megállító hatások

A következő dolog amit az osztály kezel, az egy hatás lejátszás megállításának képessége; ez egy különösen egyszerű folyamat:

```
public void Stop()
{
	foreach( EffectObject effect in effectlist )
	{
		effect.Stop();
	}
}
```

Demó 8.4

A Demó 8.4 azért készült, hogy megmutassa, hogyan használj erővisszacsatolási hatásokat. Meglehetősen egyszerű, mivel a munka neheze már készen van a *ForceEffect* osztályon belül.

Az első dolog amit a demó csinál, hogy deklarál néhány új változót:

```
ForceEffect[] effects;
string instructions;
int currenteffect;
bool lastfiring = false;
bool lastchanging = false;
```

Az első a hatások egy tömbje, mely a hatásokat fogja tárolni. A következő az utasítások egy szövege, mely közli a felhasználóval, hogy hogyan használja a demót.

A *currenteffect* változó tárolja a jelenleg használt hatás indexét, és az utolsó két logikai változó használt annak meghatározására, hogy egy gombállapot vajon megváltozott-e az utolsó képkocka óta. De majd látni fogod, hogyan működnek ezek.

Az eszköz beállítása

Van két változtatás, amelyet szükséges megtenned a kódodban, hogy egy DirectInput eszközt beállíts: meg kell keresned az erő visszacsatolásos eszközöket, és meg kell szerezni azt kizárólagos módban. Ez valószínűleg nyilvánvaló számodra, de egyébként is elmondom: megpróbálni erővisszacsatolási hatásokat futtatni egy nem erővisszacsatolásos eszközön hibákat okozhat.

A DirectInput megköveteli, hogy az erővisszacsatolási hatások csak kizárólagos eszközökön legyenek használva – kettő program nem használhat egy időben egy eszközt, ha erővisszacsatolási hatásokat használsz. Tehát itt az új beállító kód:

```
foreach( DirectInput.DeviceInstance i in
        DirectInput.Manager.GetDevices(
        DirectInput.DeviceClass.GameControl,
        DirectInput.EnumDevicesFlags.AttachedOnly |
        DirectInput.EnumDevicesFlags.ForceFeeback ) )
{
        gameinput = new DirectInput.Device( i.InstanceGuid );
        gameinput.SetCooperativeLevel(
                this,
                DirectInput.CooperativeLevelFlags.Background |
                DirectInput.CooperativeLevelFlags.Exclusive );
        gameinput.SetDataFormat( DirectInput.DeviceDataFormat.Joystick );
<code snipped>
}
```

A két fontos változtatás a Demo 8.3-ból félkövér betűkkel van írva.

A hatások inicializálása

Az ügyes *ForceEffect* osztálynak köszönhetően az FFE állományok betöltése csupán egy csettintés!

```
effects = new ForceEffect[3];
effects[0] = new ForceEffect( “gatling.ffe”, “Gatling Gun”, gameinput );
effects[1] = new ForceEffect( “pistol.ffe”, “Pistol”, gameinput );
effects[2] = new ForceEffect( “shotgun3.ffe”, “Shotgun”, gameinput );
```

Ez betölt három FFE állományt, amelyek Gatling gépfegyver, pisztoly és sörétes puska tüzét utánozzák. Mindhárom a DXSDK-val jött, és még egy csomó van ott, ahonnan származnak.

A bemenet és lejátszás kimenet összegyűjtése

Az utolsó lépés a bemenet össszegyűjtése, és a visszacsatolás lejátszása az eszközre, melyek tökéletesen megvalósíthatók a *ProcessInput*-on belül:

```
DirectInput.JoystickState state = gameinput.CurrentJoystickState;
if( state.GetButtons()[0] != 0 )
{
	if( lastfiring == false )
		effects[currenteffect].Start( 1, true );
	lastfiring = true;
}
else
{
	if( lastfiring == true )
		effects[currenteffect].Stop();
	lastfiring = false;
}
```

Az első kóddarab visszatér a botkormány állapottal, és ellenőrzi, hogy vajon le van-e nyomva a 0-s gomb. Ha igen, akkor azt ellenőrzi, hogy a *lastfiring* változó hamis-e (*false*). Ha ez hamis, az azt jelenti, hogy a legutóbbi bevitel összegyűlt, a 0-s gomb fel volt engedve és ez idő szerint le van nyomva. Ezért a program közli, hogy a jelenlegi hatást el kell indítani lejátszani.

Az ok, amiért a *lastfiring* logikai változó, hogy pontosan közölje, hogy vajon egy gomb le van-e nyomva. A lehetősége meg van annak, hogy ha gyorsan lenyomsz egy gombot, körülbelül 10-20 képkocka lesz feldolgozva, mielőtt elengeded, és anélkül hogy ellenőrzés lenne, elindítanád egy helyett 10-20-szor a hatást. Hasonlóképpen, a függvény kitalálja, hogy mikor fejezted be a gombnyomást, és megállítja a hatást.

A második botkormány gomb hasonló módon használatos a jelenlegi hatás megváltoztatására:

```
if( state.GetButtons()[1] != 0 )
{
	if( lastchanging == false )
	{
		currenteffect++;
		if( currenteffect >= effects.Length )
			currenteffect = 0;
	}
	lastchanging = true;
}
else
{
	lastchanging = false;
```

}

És végül az utasítások szövege jön létre:

```
instructions = “Nyomd le a 0-s gombot a hatás lejátszásához\n”;
instructions += “Nyomd le az 1-es gombot hogy megváltoztasd a hatást\n”;
instructions += “Jelenlegi hatás: “ + effects[currenteffect].Name;
```

9. fejezet

DirectSound

Azt már tudod, hogy hogyan mutass grafikát és hogyan szerezd meg a felhasználói adatbevitelt, de még nem tanultál a játékprogramozás egy fő részéről: hogyan használj hangokat a játékaidban. Ebben a fejezetben orvosolni fogom ezt a helyzetet a DirectSound alapjainak megtanításával, ami a DirectX hang API-ja.

A hang eszköz

Mint a Direct3D-nél és a DirectInput-nál, az alap osztály a DirectSound-ban a *Device*, mely a hangkártyádat jelképezi.

Létrehozni igazán egyszerű egy hangeszközt:

```
DirectInput.Device sound;
sound = new DirectSound.Device();
sound.SetCooperativeLevel( this, DirectSound.CooperativeLevel.Priority );
```

Csak létrehoz egy hangeszközt és beállítja az együttműködési szintet. Emlékezz arra, hogy a hangeszköz megosztott az egész operációs rendszeren keresztül, így neked kell elmondanod, hogyan akarsz együttműködni a többi programmal. Egy játéknak az *Elsőbbségi* (*Priority*) szintet szándékozol használni, mert ez ad neked a hangkártyához egy tisztességes hozzáférési szintet.

Hang pufferek

DirectSound-ban a hang alapegysége egy puffer. Egy *hang puffer* az egy objektum, amely *wave*, azaz hullámforma alapú adatokat tartalmaz, melyeket a hangkártya használ lejátszani a hangszórókon. A hang puffereknek kettő fajtája van: elsődleges pufferek és másodlagos pufferek.

Minden programnak van egy elsődleges puffere, amit a DirectSound hoz létre és önműködően kezel számodra. Ez a puffer a *Buffer* osztály által van jelképezve, és végrehajtja a másodlagos pufferek keverését.

A másodlagos pufferek egyéni hangokat fognak tartalmazni, amelyeket le akarsz játszani; ezek a *SecondaryBuffer* osztály által vannak jelképezve. Legtöbbször a másodlagos pufferekkel szándékozol majd dolgozni.

Egy WAV állomány betöltése egy másodlagos pufferbe és annak lejátszása hihetetlenül egyszerű:

```
DirectSound.SecondaryBuffer wave;
wave = new DirectSound.SecondaryBuffer( “laser1.wav”, sound );
wave.Play( 0, DirectSound.BufferPlayFlags.Default );
```

A másodlagos puffer konstruktor fogja a betöltendő wave állomány nevét és egy hivatkozást a hang eszközre, amit én most *sound* néven adtam meg.

Lejátszó pufferek

A *Play* függvény fog két paramétert: egy elsődleges értéket (biztonságosan használhatsz 0-t, mely a legmagasabb elsőbbség) és egy flag-et, amely azt jelképezi, hogyan kellene lejátszani a puffert. A következő táblázat a *BufferPlayFlags* elérhető értékeit listázza:

Érték	**Jelentés**
Default	Alapértelmezett értékek használata
Looping	A hang ismétlődik, míg kézzel le nem állítják
LocateInHardware	Lejátszani ezt a hangot egy hardver pufferben
LocateInSoftware	Lejátszani ezt a hangot egy szoftver pufferben
TerminateByTime	Ha a hardver tele van, a legkevesebb idő múlva megáll a hang, és ez a hang elkezd lejátszódni

TerminateByDistance	Ha a hardver tele van, akkor a felhasználótól legtávolabbi hang leáll
TerminateByPriority	Ha a hardver tele van, akkor a legalacsonyabb elsőbbségű hang (legnagyobb szám) leáll

Valószínűleg csak az alapértelmezett értéket fogod használni mindenhez. Egy hardver eszköz egy adott időben csak bizonyos számú hangot tud mixelni. A legtöbb hangkártya körülbelül 32 különböző hangot tud kezelni. Habár, ha szükséges 32 puffernél több, az alapértelmezett flag önműködően elkezdi keverni a többi hangot szoftverben, mely extra számítási időt vesz igénybe. Ez nem nagy kár, mert a hangszámítás napjainkban viszonylag olcsó. De ritkán kell majd 32 hangnál többet lejátszanod egyszerre.

Puffer leírások

Egyszerűbb anyagok számára rendben van az a módszer, amit előzőleg mutattam egy puffer létrehozására, de ez nem tesz lehetővé sok testreszabást. Azért, hogy testreszabd a puffereidet, ki kell töltened egy *BufferDescription* objektumot. Ezeknek az objektumoknak sok változóik vannak; a legfontosabbakat a következő táblázat listázza:

Érték	**Jelentés**
BufferBytes	A puffer mérete bájtokban. Elmehet 0-ig, mikor betöltesz egy wave állományt.
Control3D	Manipulálható 3D-ben.
ControlEffects	Manipulálható hatás számításokkal.
ControlPan	Bal vagy jobb oldali hangzás lehet. Nem használható 3D számításnál.
ControlVolume	A puffer hangerejének vezérlése.
DeferLocation	Hardverbe vagy szoftverbe helyezi a puffert, mikor lejátszódik.
GlobalFocus	Ha igaz (*true*), még minimalizált ablak esetében is lejátszódik. (Alapértelmezésben a pufferek nem játszódnak le, mikor az ablakod nincs fókuszban.)
StickyFocus	Ha igaz (*true*), lejátszódik, mikor az ablakod nincs fókuszban, de csak ha a jelenleg fókuszban lévő ablak nem használ DirectSound-ot.

Sokkal több érték létezik, mint ami a táblázatban listázva van, de valószínűleg azokkal nem leszel érintett.

Itt van hogy hozz létre egy hang puffert, ami hanghatásokat támogat:

```
DirectSound.BufferDescription desc = new DirectSound.BufferDescription();
desc.ControlEffects = true;
```

Amint beállítottad a leírásodat, használhatod egy hang puffer létrehozására:

```
wave = new DirectSound.SecondaryBuffer( “laser1.wav”, desc, sound );
```

Hang hatások

A hatás számítás egy nagyon tiszta dolog. Korábban úgy volt, hogy csak betöltöttél wave állományokat és játszottál velük, nem számított, hogy milyen volt a játék környezete. De így gyakran egy valótlan hatást kaptál. Ha egy hang történik egy zárt szennyvízcsatornában, akkor az visszapattan a falakról és visszhangzik, míg ugyanaz a hang, amelyet a szabadban hoznak létre, nem visszhangzik.

Hatás számítással megváltoztathatod, hogyan hangozzék egy hang! Különböző nagy számú paramétert szerkeszthetsz, hogy testreszabd, hogyan akarod a dolgokat hallani. Ezek a hatás flag-ek a *DSoundHelper* osztályban vannak, és a következő táblázat listázza őket:

Érték	**Hatás**
StandardChorusGuid	Kórus hatás; a hang egy „megkettőzését” okozza
StandardCompressorGuid	Tömörítő hatás; bizonyos amplitúdókat vág le
StandardDistortionGuid	Torzítja a hangokat a hullámformák tetejének levágásával
StandardEchoGuid	Visszhang hatás; hang ismétlést okoz
StandardFlangerGuid	Hasonló egy visszhanghoz
StandardGargleGuid	Toroköblögetés-szerű hangot okoz
StandardParamEqGuid	Parametrikus equalizer; bizonyos frekvenciák módosítását teszi lehetővé
StandardWavesReverbGuid	A hang visszaverődését okozza

Nagyon könnyű beállítani egy hatást vagy hatások bármely számát:

```
DirectSound.EffectDescription[] e = new DirectSound.EffectDescription[1];
e[0].GuidEffectClass = DirectSound.DSoundHelper.StandardFlangerGuid;
wave.SetEffects( e );
```

Az első sor létrehozza a hatásleírások egy új tömbjét. Mivel csak egy hatást használok, ezért csak egy indexre van szükségem.

A következő sor beállítja a hatás osztályát egy „flanger”-nek, és az utolsó sor beállítja a hullámforma hatásait.

Megjegyzés: a hatásokat csak azon pufferekhez lehet alkalmazni, amelyek annál a pillanatnál nem játszódnak le, tehát bizonyosodj meg arról, hogy hívod a Stop függvényt egy pufferen, mielőtt megváltoztatod a hatásait.

Hang 3D-ben

Egyike a legújabb hangtechnológiáknak a 3D hang számítás, és a DirectSound ezt hihetetlenül könnyen kezelhetővé teszi.

3D pufferek

A 3D hang kulcs összetevője a *Buffer3D* osztály, mely egyszerűen körülvesz egy létező *SecondaryBuffer*-t:

```
// egy másodlagos puffer létrehozása először
DirectSound.BufferDescription desc = new DirectSound.BufferDescription();
desc.Control3D = true;
wave = new DirectSound.SecondaryBuffer( “explosion1.wav”, desc, sound );
// a 3D puffer létrehozása
wave3D = new DirectSound.Buffer3D( wave );
```

Először is létre kell hoznod a másodlagos puffert a *Control3D* flag beállításával. Ha ezt nem teszed meg, akkor a 3D puffered nem jön létre megfelelően. Amint ezzel megvagy, egyszerűen létre tudsz hozni egy *Buffer3D* objektumot, beleágyazva a hangodat, és minden készen áll.

Most minden amit tenned kell, a másodlagos puffer lejátszása, és az önműködően 3D-be lesz számítva.

Mozoghatsz a puffer körül a *Position* tulajdonságának megváltoztatásával:

```
wave3D.Position = new Vector3( 1, 0, 0 );
```

Ez beállítja a puffert egy egységgel jobbra, így az a jobb oldali hangszóróból fog lejátszódni.

További 3D témák

Ez csak egy nagyon alapszintű bemutatója a DirectSound 3D összetevőjének. Tulajdonképpen sokkal többminden van, mint amit én bemutattam, de ennyi elég az elinduláshoz.

Van néhány további pont, amelyeket észben kell tartanod, mikor 3D hang puffereket használsz. Az első, hogy a 3D hangoknak monóknak kell lenniük, nem sztereóknak, ezért meg kell bizonyosodnod, hogy a betöltött hullámformák monó formátumban vannak. A 3D hang egyedüli forrásként számítódik, és a sztereó hang egy mesterségs alkotás 2D hang szimulálására tervezve, nem 3D hang.

Másik dolog, amit észben kell tartanod, hogy minden másodlagos puffer csak egy 3D pufferhez lehet társítva; azért, hogy több 3D hangod legyen, minden hang megköveteli a saját másodlagos pufferét és 3D pufferét.

Végül, nagyon költséges mozgásban tartani az összes puffert, úgyhogy lehet, hogy bele akarsz nézni a *Listener3D* osztályba, ami megengedi neked, hogy ugyanazon időben mozgass egy listener-t („hallgató”) és a puffereidet, egy igazi 3D környezetet szimulálva. Egy listener beállítása megköveteli az elsődleges puffer körüli lejátszást.

Nézd meg a Doppler hatások használatát a *Listener3D* osztállyal kapcsolatban. Ezek azok a hatások, amiket akkor tapasztalsz, amikor egy versenypályán vagy és hallod az autó sebességét az irányodba aztán gyorsan el. Amint az autó közeledik, a hang magassága egyre nagyobb lesz, mert a hanghullámok tömörödnek, és ahogy az autó távolodik, úgy csökken a hangmagasság a hanghullámok kitömörödése miatt. A Doppler hatások használatával igazán csinos programokat alkothatsz.

10. fejezet

Összerakni egy játékká

Ennél a pontnál – feltételezve, hogy először elolvastad a könyvet, és nem valami különc vagy, aki az utolsó fejezetnél kezdi el az olvasást – eleget kell tudnod a C#-ról és a DirectX-ről, hogy elkezdd összerakni a saját játékodat. Szó szerint millió különféle játékfajta van, amit megcsinálhatsz, és én nem tudom neked megmutatni az összeset; helyette én csak azt mutatom meg, hogyan készíts el egy egyszerű kis játékot.

Egy tervezet összeállítása

Sok ember, mikor először elkezd írni játékokat, elköveti azt a hibát, hogy csak leül egy számítógép elé és elkezdi a kódírást olyan gyorsan, amennyire csak tudja. A játékprogramozás korai időszakában alkalmazhattad volna ezt a megközelítést, de többé már nem tehetsz így. Előbb vagy utóbb nekiütközöl egy problémának, és vissza kell menned a kód elejére, hogy kitaláld, hogyan oldd meg. Ez nem mókás, hidd el nekem.

Hogy élvezd a feszültségmentes játékalkotást, egy általános ötletet kell felállítanod arról, hogy milyen fajta játékot akarsz írni, mi kell hozzá, ki akar majd játszani vele, és így tovább.

A játék műfaj

Az első kérdés, amit fel kell tenned magadnak, amikor tervezed a játékodat, hogy „Milyen fajta játékot akarok készíteni?” Legyen egy első személyű lövöldözős? Egy játéktermi („arcade”) játék? Egy szerepjáték? Egy kirakós játék vagy inkább valami sport játék? Talán elég vállalkozó kedvű vagy, hogy olyan játékra gondolj, mint amilyenre ezelőtt senki sem; ha így van, jó neked!

Az egyik legnagyobb hiba, amit egy kezdő játékprogramozó elkövethet, hogy nagyobbat harap, mint amennyit rágni tud. Könnyű ránézni az új, élvonalbeli játékokra és azt mondani, hogy „Ó, istenem, én valami olyasmit akarok, mint ez!” Mindenki képes akar lenni megcsinálni a saját Doom III játékát. De egy Doom III-szerű játék több évnyi erőfeszítésbe, tucatnyi emberbe, és

dollármilliókba kerül, míg elkészül. Mint kezdő programozónak, valószínűleg nincs meg a hozzáférésed ezekhez az erőforrásokhoz (ha mégis, akkor küldj nekem egy e-mail-t – talán dolgozhatunk együtt valamin).

De a játékok lehetnek mókásak megamániás szörnyűségek nélkül is. Néhány évvel ezelőtt az embereknek az volt a hozzáállása, hogy ha egy játék nem véres 3D-s, akkor nem is érdemes játszani vele. Később az olyan társaságok, mint a Popcap, megmutatták, hogy még a kis, egyszerű játékok is lehetnek rendkívül mókásak.

A grafika önmagában még nem csinál egy játékot. A fenébe is, a játékoknak még csak nem is szükséges, hogy legyenek rajzaik. Egy játékot a hangulata határozza meg, nem az, hogy milyen csinosan néz ki, és ez az első dolog, amivel tisztában kell lenned.

Azt javaslom, hogy kezdetnek válassz valami egyszerűt. A legfontosabb tényező, hogy elég egyszerű játékot válassz, amit be is tudsz fejezni. Az első számú „lelohasztó” ok a játékprogramozóknak, hogy nem képesek befejezni egy játékot, mert az túl nagy. Gondold végig: megpróbálsz elkezdeni csinálni egy hatalmas RPG-t (szerepjáték), és néhány hét után rájössz, hogy nem végeztél semmi látványosat – hirtelen elveszíted az ösztönzést a project befejezésére, és az soha nem fog elkészülni.

Ha szükséged van egy ötletre, hogy hol kezdj hozzá, menj vissza a régi Atari, NES (Nintendo Entertainment Sytem) és SNES (Super Nintendo Entertainment Sytem) játékokhoz. Tonnányi mókás játék készült azokra a konzolokra, és azok legtöbbjét igazán egyszerű elkészíteni. Én mindig szerettem az űrhajós lövöldözős játékokat, így ez az a játéktípus is, amit tervezni szándékozom. A neve *Generic Space Shooter 3000*.

Eldönteni, hogyan működjön a játék

Mondhatod, hogy akarsz készíteni egy űrhajós lövöldözőst vagy RPG-t vagy bármit, de az nem csinál neked sok jót, míg ki nem találod pontosan, hogy mit szándékozol belerakni, és hogyan fog minden egymásra hatni.

Az univerzum

Amint eldöntötted a műfajodat, gondolkodnod kell az univerzumról, amibe a játékodat elhelyezed. Ha csinálsz egy játékot, ami modellez némi való-világ elgondolást (futás mindenfelé, lőni dolgokat, szálló repülőgépek, és így tovább), akkor ez az elgondolás meglehetősen egyszerű. Egy RPG vagy akciójátékban az univerzum mindenhol meghatározott, ahol a játékos tud menni.

De a dolgok egy kis eltérést kapnak, mikor egy elvont műfajba mész, mint a kirakós játékoké – nem tudsz könnyebben meghatározni egy univerzumot való világú feltételekben. Az univerzum ebben az esetben egyszerűen a tábla/képernyő, ahol a játékos játszani fog.

Az ötlet a *Generic Space Shooter 3000* számára meglehetősen egyszerű. A játékos egy űrhajó, amely röpköd az űrben és lövöldöz dolgokat. Az univerzum egyszerűen maga a világűr. Igazán semmi bonyolult nincsen, csupán egy nyitott univerzum, amiben a játékot csinálom.

A szereplők

A következő dolog, amivel tisztában kell lenned, azok a szereplők – bármi a játékban, ami létezik az univerzumon belül. A szereplőknek nem kell tényleges animált objektumoknak lenniük; egyszerűen bármik lehetnek, amikkel érdemes együttműködni. Ha tudsz mozgatni sziklákat a talaj fölött a játékodban, akkor egy szikla az egy szereplő.

A *Generic Space Shooter 3000* számára három fő szereplőtípus lesz:

- **Űrhajók**, amik az ide-oda repülő hajók, és irányíthatók a játékos vagy a számítógép által.
- **Lövedékek**, amik objektumokként vannak meghatározva, amelyek más objektumoknak okoznak kárt, mint a rakéták vagy lézersugarak.
- **Erősítők**, amik egyszerű felvehető objektumok és a játékos járművét fejlesztik.

Az adat

Amint megvan az ötleted a játékod általános tervéről, ki kell találnod, hogy pontosan milyen adatokkal kell a szereplőknek rendelkeznie. Keresztülviszlek azon a folyamaton, amely annak kitalálásához vezet, hogy a *Generic Space Shooter 3000* milyen adatokat igényel.

Űrhajók

Ez egy egyszerű lövöldözős játék akar lenni, így az űrhajóknak nem lesz szükségük olyan sok információra. Lesz *energiá*juk, amely azt jelzi, hogy milyen erős az űrhajó. Ha az energiaszint eléri a 0-t, akkor a játékos meghal.

Egy közönséges elgondolás a játékokban a rugalmasság – minél rugalmasabb minden, annál inkább tudsz változtatni a játék futása közben, és adhatsz a játékosaidnak további elérendő célokat.

Ahelyett, hogy azt mondanád, „Az összes űrhajó mozogjon 200 képpontnyit másodpercenként”, inkább mondhatod azt, hogy „Az űrhajók tudnak mozogni 200 képpontnyit másodpercenként, de ha kapnak egy sebesség erősítőt, akkor gyorsabban tudnak mozogni!” A rugalmasság érdekében az űrhajók a GSS3K-ban tartalmazni fognak egy sebesség változót.

Csak hogy fűszerezzem a dolgokat, hozzáadtam a *pajzs* fogalmát az űrhajókhoz. Egy pajzs a károkozást csökkenteni vagy teljesen törölni fogja – mondjuk, ha a hajó lézersugár találatot kap, a károkozás csökkenni fog, így az nem sérti a hajót annyira. A pajzs értékek 0.0 és 1.0 határértékek között fognak mozogni, ahol a 0 azt jelenti, hogy többé nem tereli el a károkozást, míg 1.0 el fogja hárítani az összes kárt. A dolgok további fűszerezése érdekében, mindig, amikor a játékos pajzsa találatot kap valamitől, csökkenni fog 0.02-vel (2 százalék), ami azt jelenti, hogy egy hajó 50-szer kaphat találatot, mielőtt a pajzsa teljesen lemerül. Ha a pajzsok 0.5 értéken vannak (50 százalék), akkor egy 10 károkozás értékű lézersugár le fog venni 5 energiapontot és 2 százalékot a pajzsaidból.

Az űrhajóknak lesz egy fegyvereket tartalmazó tömbje, amit a játékos használni tud, egy változó, ami a következő időt tartalmazza, amelynél a játékos tüzelhet a fegyvereivel, és egy pontszám változó.

Fegyverek

A fegyverek egyszerű objektumok, amelyek leírják, hogy a lövedékek hogyan jönnek létre. A fegyvereknek van egy hangjuk, ami lejátszódik, amikor tüzelnek, egy késleltetési idő, amely meghatározza, hogy a fegyver újratöltéséhez mennyi idő kell, és egy név. Amikor a fegyverek tüzelnek, visszatérnek lövedékek egy tömbjével.

Lövedékek

A lövedékek a tényleges objektumok a játékban, amelyek kárt okoznak; ezek különösen egyszerű objektumok. A lövedékek egyszerűen azt tárolják, hogy mekkora kárt okoznak és van egy hivatkozásuk az űrhajóra, amely kilőtte őket, így tudod pontokkal díjazni, ha megsemmisített bármit is.

Erősítők

Az erősítők a legelvontabb tárgyak a játékban; tulajdonképpen nem tárolnak semmilyen különleges adatot. Egyszerűen van egy elvont függvényük, amelyet alkalmaznak egy űrhajóra, amikor a játékos megkapja őket.

Közös jellegzetességek

A három megfogható objektumtípus a *GSS3K*-ban – űrhajók, lövedékek és erősítők – osztozik néhány közös tulajdonságon. Mindenekelőtt van képük, így kell, hogy legyen textúra információjuk, valamint helyzetük, méretük, szögek és így tovább.

Hol láttad ezeket a tulajdonságokat ezelőtt? A *Sprite* osztályban a 7. fejezetből, természetesen! Úgy van, az én egyéni *Sprite* osztályomnak megvan azon jellegzetességek mindegyike, így az összes játék objektum örökölni fog a sprite-ból.

Bár a játék objektumoknak többminden szükséges, mint amit a jellegzetességek nyújtanak a *Sprite* osztályból. A játék fizikája számára, az összes játék objektum tárolni fog további vektor adatokat, jelképezve a tárgyak jelenlegi x és y sebességét, valamint jelenlegi x és y gyorsítását.

További jellegzetességek az ütközési téglalap és *megsemmisült* néven egy logikai változó. Az ütközési téglalap lesz használva annak közlésére, hogy az

objektum vajon ütközött-e egy másik objektummal, a *megsemmisült* nevű logikai változó pedig elmondani a játéknak, hogy az objektumot megölték, és ezért nem kell további számításokban részt vennie. Később látni fogod, hogy miért vannak ezek.

Ez eléggé összefoglalja a *GSS3K* objektumait, így most tovább egy új keretrendszer felé!

Egy rövid fizika óra

A *sebesség* és *gyorsítás* nagyon egyszerű fogalmak. Ha van egy objektumod, aminek a sebessége meg van határozva, mint 10 méter másodpercenként az X tengelyen, akkor a tárgy mozogni fog előre 10 métert az X tengelyen minden másodpercben. Ez egy egyszerű számítás:

új hely x = régi hely x + (10 x eltelt idő)

Ha 0,3 másodperc telt el, akkor három egység adódik a tárgy x helyéhez.

A gyorsulás egy még haladóbb fogalom, de ezt is könnyű megérteni. A gyorsulás egyszerűen azt a sebességet határozza meg, amelyen egy tárgy gyorsasága megváltozik. Ha van egy egy méter per másodpercenkénti gyorsulásod, az azt jelenti, hogy egy objektum gyorsasága egy méter per másodperccel növekszik minden másodpercben. Ez szintén egy egyszerű számítás:

új gyorsulás x = régi gyorsulás x + (1 x eltelt idő)

Ha elindulsz egy 10-es értékű sebességgel, akkor egy másodperc után a sebesség 11 lesz, aztán 12 a következő másodpercben, és így tovább.

Egy új keretrendszer

Az egyszerű keretrendszerek, amiket a legtöbb fordító létrehoz, megvalósítják a céljaikat: nyújtani neked egy egyszerű helyet, amelyből felépítheted a programodat. Az ilyen keretrendszerek nem szörnyen összetettek, és nem is igazán bővíthetők, ők csak valami gyorsak-és-piszkosak.

De egy keretrendszer sokkal haladóbb és rugalmasabb lehet. Például majdnem minden játék használja a *játék állapot* fogalmát. Egy játék lehet „Bemutató” módban, „Játszás” módban, „Csúcs pontok mutatása” módban, és még millió más módban. Az egyszerű keretrendszerek, amiket eddig láttál, nem csinálták a *játék állapot* fogalom használatát.

Előrementem és terveztem egy vadonatúj keretrendszert neked, hogy használhasd. Nem olyan összetett, de magábafoglal tisztességes mennyiségű kódot, sokat, melyet nem mutatok meg. Az összes kód a könyv CD mellékletén megtalálható, a Demo 10.1-ben. A keretrendszer használja az állapotok új fogalmát.

Felállítás

Az előző demókban én csak feltételeztem, hogy az elérhető képernyőfelbontás 640x480. Amíg 99,999 százaléknyi esély van arra, hogy a felbontás elérhető, attól még az csak egy feltételezés, és így veszélyes. Garantálom, hogy néhány játékosnak nem lesz meg az a felbontás a rendszerén, és tudni akarja, hogy a játékod miért nem fut. Tehát mindig jó ötlet hagyni, hogy a játék ellenőrzési képességekkel rendelkezzen, és a felhasználónak legyenek választási lehetőségei.

A Setup.cs állományban a CD-n van egy form osztály, amely felhasználói preferenciákat tölt be. A form lehetővé teszi a felhasználónak, hogy kiválaszthassa a felbontást, amelyben futtatni akarja a játékot, és a használni kívánt adatbeviteli eszközt. A kód alapvetően belemegy a Direct3D-be és a DirectInput-ba, és segítségükkel kilistázza az elérhető eszközöket. A setup form fog egy *DeviceOptions* osztályt mikor létrejön, és kitölti az osztályt a megfelelő paraméterekkel, attól függően, hogy a játékos mit akar használni.

Eszköz lehetőségek

A *DeviceOptions* osztály egyszerű (a Device.cs-ben):

```
public class DeviceOptions
{
        // grafika
        public bool Windowed;
        public D3D.DisplayMode GraphicsMode;
```

```
        // joystick
        public bool UseJoystick = false;
        public Guid Joystick;
        public int JoystickDeadzone = 1500;
        public DI.InputRange JoystickRange = new DI.InputRange( -10000, 10000 );
}
```

Az opciók lehetővé teszik, hogy meghatározz néhány beállítható lehetőséget minden eszköznek a rendszeren. A grafikus rész például meghatározza, hogy a felhasználó ablakos módot akar-e, és milyen kijelzőmódot használ. A setup form megengedi mindkettő kitöltését.

A joystick résznek egy kissé több információja van; meghatározza, hogy vajon a felhsaználó akar-e használni botkormányt, és ha igen, akkor meghatározza annak GUID-ját. Emlékszel a demókra a 8. fejezetből? Az egyszerűen felszedte az első csatlakoztatott botkormányt; ez elég buta dolog, mert egy felhasználónak egynél több telepített joystickja is lehet. A *DeviceOptions* osztály (a setup form-mal kapcsolatban használva, amire később kitérek) megengedi a felhasználónak, hogy beállítsa, amelyiket akarja.

A holt zóna és határérték bizonyos értékekkel keményen-kódolt, és a setup form nem engedi meg, hogy megváltoztasd őket. Erre igazán nincs is ok. Ez az osztály egyszerűen egy kényelmes hely az eszközértékek elhelyezésére.

Eszköz blokkok

Úgy találtam, hogy hasznos az összes játék eszközt egy osztályba zárni. Így is tettem, és elneveztem *DeviceBlock* osztálynak.

```
public class DeviceBlock
{
        public D3D.Device Graphics = null;
        public DS.Device Sound = null;
        public DI.Device Keyboard = null;
        public DI.Device Mouse = null;
        public DI.Device Joystick = null;
        public D3D.Sprite Sprite = null;

        public void Initialize( DeviceOptions options, Game game )
        void InitGraphics( DeviceOptions options, Game game )
        void InitSound( Game game )
        void InitKeyboard( Game game )
```

```
        void InitMouse( Game game )
        void InitJoystick( DeviceOptions options, Game game )
}
```

Kivágtam az összes kódot a függvényeknek, mert 99 százalékát már láttad a 6, 7, 8 és 9-es fejezetekben.

Megjegyzés: szabadon hozzáadhatsz még eszköz opciókat a *DeviceOptions* osztályhoz, hogy még rugalmasabbá tedd. Ez az egyszerű változat az én szükségleteimnek megfelelő, így nem láttam mindent, amiért ultra konfigurálhatóvá tettem volna.

Bevitel ellenőrzők

Emlékszel, hogy a Demó 8.4 hogyan használt logikai változókat annak pontos meghatározására, hogy egy joystick gomb le van-e nyomva vagy felengedve, átalakítva a szelektív hívásos („polling”) adatbeveitel összegyűjtési folyamatot ál „esemény vezérelt” rendszerré? Fájdalom volt a fenéknek, ugye? Kellett minden gombhoz egy Boolean típusú változó, és azt is ellenőrizned kellett, hogy ezek közül melyik változott meg. Úgy döntöttem, beépítem azt a viselkedést a *KeyboardChecker*, *MouseChecker* és *JoystickChecker* osztályokba. Mindegyik osztály különböző, és mindegyik különböző fajta eszközt kezel. Fogadok, hogy kitalálod, melyik osztály melyik eszközt kezeli. Ezek az osztályok az *Input.cs*-ben vannak.

A kód az osztályoknak hosszú és unalmas, és nem igazán mutat neked semmit, amit nem láttál ezelőtt, ezért ki fogom hagyni és beleugrok annak kifejtésébe, hogy hogyan működnek az osztályok.

Billentyűzet ellenőrzés

A billentyűzet egy egyszerű eszköz, amely csak gombokat foglal magába. Amikor egy gomb lenyomódik vagy felengedődik, egy üzenetet kell küldeni egy eseménykezelőnek.

A billentyűzet ellenőrzőnek lekérdezőnek kell lennie; ezt a *KeyboardChecker.ProcessInput* hívásával végezhetjük el. Ez a függvény a lenyomott gombok egy tömbjét kapja, és összehasonlítja ezt a tömböt azokkal a gombokkal, amelyek utoljára le voltak nyomva, mikor a *ProcessInput* hívva

volt. Ha a gombok bármelyike megváltozott, a függvény kiküldi a gomb eseményt.

A gomb események felállítása delegáltakat használ, mint ez:

```
KeyboardChecker checker = new KeyboardChecker( keyboard );
checker.PressEvent = new KeyboardChecker.KeyFunction( KeyDown );
checker.ReleaseEvent = new KeyboardChecker.KeyFunction( KeyUp );
```

A *KeyboardChecker.KeyFunction* egy delegált egy függvénynek, amely egy *DirectInput.Key* szerkezetet fog adatbeviteleként. Az előző kód hívná aztán ezt a függvényt egy gomb bármikori lenyomásakor:

```
public void KeyDown( DirectInput.Key k )
{
	// gombnyomás kezelő kód
}
```

Egér ellenőrzés

Az egerek összetettebbek, mint a billentyűzetek, mert vannak gombjaik és tengelyeik, és kezelned kell mindkettőt. A gombok kezelése hasonló módon történik, mint a billentyűzet gomboké, kivéve hogy *DirectInput.Key* értékek helyett *int* típusú számokat használunk. Például:

```
MouseChecker checker = new MouseChecker( mouse )
checker.PressEvent = new MouseChecker.ButtonFunction( MouseDown );
// később:
public void MouseDown( int button )
{
	// gombnyomás kezelés
}
```

Az ellenőrző a tengelyek mozgását is ellenőrzi:

```
checker.MoveEventX = new MouseChecker.MovementFunction( MouseMoveX );
checker.MoveEventY = new MouseChecker.MovementFunction( MouseMoveY );
checker.MoveEventZ = new MouseChecker.MovementFunction( MouseMoveZ );
// később:
public void MouseMoveX( int delta )
{
	// x elmozdult “delta” egységekkel, kezeld azt itt
```

```
}
```

Bármikor egy egér elmozdul, az mozgás események változnak. Ha egy egér nem mozdul, a *delta* érték 0, és az esemény nem kap változást.

Mint a billentyűzetnél, az egér ellenőrzőknek lekérdezőknek kell lenniük, a *ProcessInput* függvényt használva:

```
checker.ProcessInput(); // bármilyen esemény hívása, ha megtörténik
```

Botkormányok

Az utolsó osztály, a *JoystickChecker*, nagyon hasonló az egér ellenőrzőhöz, egy különbséggel a viselkedésben: a Joystick tengely események minden időben hívódnak, mikor a *ProcessInput* függvény hívódik. Ez azért van, mert a joystick eszközök abszolút pozicionálást használnak – sokkal hasznosabb azt tudni, hogy *hová* van mindig a botkormány pozicionálva, mint egyszerűen azt, hogy *ha* elmozdult.

A botkormányoknak két gomb delegáltjuk és öt tengely delegáltjuk van:

```
public ButtonFunction PressEvent;
public ButtonFunction ReleaseEvent;
public MovementFunction MoveEventX;
public MovementFunction MoveEventY;
public MovementFunction MoveEventZ;
public MovementFunction MoveEventU;
public MovementFunction MoveEventV;
```

Ezek az egér függvényekhez hasonlatosan használtak.

Játék állapotok

A játékok *állapot készülékek*, ami azt jelenti, hogy állapotuk vagy módjuk mindig változik. Mikor elindítasz egy játékot, mutatja a címképernyőt, esetleg a társaság nevét, amely tervezte. Ez a bemutató („Introduction”) állapot. Amikor tulajdonképpen játszol a játékkal, akkor a Játék állapotban vagy. Amikor a beállítási lehetőségekkel játszol, akkor a Konfigurációs állapotban vagy. Néhány játéknak lehet még sok különböző fajta játékállapota, mint amikor a játékosnak

meg kell változtatnia a felhasználói felületet, hogy különböző játékmódokba léphessen.

Minden állapot eltérően kezeli a dolgokat. A bemutató állapotban az összes adatbevitelt a kilépéshez és a Menü állapothoz ugráshoz akarhatod. A Játék állapotban, amikor egy gombot lenyomnak, akkor egy fegyver tüzel, de magától értetődően nem akarod, hogy ez megtörténjen más egyéb állapotban. A Súgó állapot egy súgó menüt fog mutatni neked, kifejtéssel a játékról, és így nem szeretnéd, hogy a játék összes más objektuma ki legyen rajzolva; csak a Súgú menüt akarod látni.

Különböző állapotok megléte lehetővé teszi, hogy elkülönítsd az összes fajtáját az anyagodnak diszkrét darabokba. Egy állapot objektum kezelni fogja az adatbevitelt, a számítási eseményeket, és kirajzolja a játék képernyőt, lényegében magábafoglalva a *ProcessInput*, *ProcessFrame* és *Render* függvényeket a korábbi keretrendszerekből.

Hadd mutassam meg neked az alap *GameState* osztályt, kód nélkül:

```
{
        protected bool graphicslost = false;
        protected KeyboardChecker keychecker = null;
        protected MouseChecker mousechecker = null;
        protected JoystickChecker joystickchecker = null;

        public GameState()
        public virtual void ProcessInput()

        protected virtual void KeyboardDown( DI.Key key ) {}
        protected virtual void KeyboardUp( DI.Key key ) {}

        protected virtual void MouseButtonDown( int button ){}
        protected virtual void MouseButtonUp( int button ) {}
        protected virtual void MouseMoveX( int delta ) {}
        protected virtual void MouseMoveY( int delta ) {}
        protected virtual void MouseMoveZ( int delta ) {}

        protected virtual void JoyButtonDown( int button ) {}
        protected virtual void JoyButtonUp( int button ) {}
        protected virtual void JoyMoveX( int delta ) {}
        protected virtual void JoyMoveY( int delta ) {}
        protected virtual void JoyMoveZ( int delta ) {}
        protected virtual void JoyMoveU( int delta ) {}
```

```
        protected virtual void JoyMoveV( int delta ) {}

        public virtual void LostFocus() {}
        public virtual void GotFocus() {}

        public abstract GameStateChange ProcessFrame();
        protected abstract void CustomRender();
        public void Render()
}
```

Nyilván sokkal többminden tartozik az osztályhoz, mint az a három függvény. Az adatbevitel számítása egy fő összetevője ennek az osztálynak – meghatároz három adatbeviteli ellenőrzőt és a velük használt függvényeket. Jegyezd meg, hogy az összes adatbeviteli esemény függvény üres és virtuális, lehetővé téve neked, hogy felülírd azokat, amelyek később fontosak lesznek neked, a többit pedig nem kell kódolnod. Ha nem törődsz egy joystick V tengelyének olvasásával (igazán, ki törődik vele?), akkor nem kell deklarálni a saját egyéni *JoyMoveV* függvényedet, mert az alap *GameState* osztálynak már van egy, ami nem csinál semmit.

A renderelés egy kicsit áramvonalas; két renderelő függvény van: *Render* és *CustomRender*. A *Render* az végrehajtott ebben az osztályban; gondoskodik az eszköz elvesztéséről és minden más dologról, így arról neked nem kell. A *Render* hívja a *CustomRender* függvényt, aminek meghatározása tőled függ.

Állapot változások

Az állapot rendszer ehhez a keretrendszerhez verem-alapú, ami azt jelenti, hogy az összes állapot egy vermen tárolódik. Ez nagyon hasznos, és hadd fejtsem ki, hogy miért. Először is, képzelj el egy nem-verem alapú rendszert: kezded a játékot egy bemutató állapottal – mutatja a címképernyőt és talán egy intróvideót vagy valami hasonlót – aztán a játék átkapcsol a Menü állapotra, mely lehetővé teszi egy új játék indítását, és ezután a program Játék állapotra kapcsol.

Mikor a Menü állapot kilépett, a program megsemmisíti és létrehoz egy Játék állapotot.

Most mi történik, ha a felhasználó kilép a játékból? Egy nem-verem-alapú rendszerben a Játék állapot megsemmisül és a Menü állapot jön létre újra. Ez pazarlásnak tűnik; a Menü állapot már létrejött egyszer, tehát miért semmisül meg a Menü állapot, ha a játék később még vissza akar térni oda?

Ez az a pont, ahol a verem rendszer bejön: a Menü állapot megsemmisítése helyett megtarthatod azt, és rakhatsz egy új állapotot a tetejére – a Játék állapotot. Amikor a Játék állapot befejeződik, szedd le a veremről és menj vissza a Menü állapotra.

A verem lehetővé teszi neked, hogy felfüggessz egy állapotot, és aztán visszatérj hozzá egy későbbi időpontban, bármi megsemmisítése nélkül.

A *ProcessFrame* függvény visszatér egy *GameStateChange* osztállyal, mely elmondja a játéknak, hogyan kell változtatni az állapotot. Két változója van:

```
public class GameStateChange
{
        public bool Terminated = false;
        public GameState NextState = null;
}
```

A *Terminated* logikai változó elmondja a játéknak, hogy a jelenlegi állapot megszűnt-e vagy sem. Ha igen, az állapot lekerül a veremről. A *NextState* változó egy hivatkozást tart a következő állapotra; ha ez *null*, akkor a játéknak vissza kell térnie az előző állapotra a vermen. A következő táblázat listázza a viselkedéseket:

Terminated	**NextState**	**Viselkedés**
false	null	Lényegében nem csinál semmit, mert az állapot nincs megszakítva és nincs új állapot. Nincs ok ezeket használni.
true	null	Jelenlegi állapot megszakítva, így a játéknak vissza kell térnie az előző állapotra.
false	nem-null	A jelenlegi állapot felfüggesztése és egy új állapot adása a létező állapot tetején.
true	nem-null	A jelenlegi állapot megszakítása és menni egy újra.

Egy minta állapot

Elláttam egy minta állapottal a *State.cs* állományt neked, vizsgálat céljából. Itt van:

```
public class SampleState : GameState
{
	bool done = false;
	bool focused = true;
	public override GameStateChange ProcessFrame()
	{
		if( done == true )
			return new GameStateChange( true, null );
		if( focused )
		{
			// csinálni itt számítást
		}
		else
		{
			// különben altatni a szálat, hogy a CPU ne járjon fölöslegesen
			System.Threading.Thread.Sleep( 1 );
		}
		return null;
	}
	public override void LostFocus() { focused = false; }
	public override void GotFocus() { focused = true; }
	protected override void CustomRender()
	{
		Game.devices.Graphics.Clear(
			D3D.ClearFlags.Target, System.Drawing.Color.White,
			1.0f, 0 );
		Game.devices.Graphics.BeginScene();
		// csinálni rajzolást itt
		Game.devices.Graphics.EndScene();
		Game.devices.Graphics.Present();
	}
	protected override void KeyboardDown( DI.Key key )
	{
		if( key == DI.Key.Escape )
			done = true;
	}
}
```

Ez az állapot egyszerűen rajzol egy fehér képernyőt, és kilép, amikor a felhasználó lenyomja az Escape gombot. Ez eléggé hasonló viselkedés ahhoz,

amit a kezdeti keretrendszerben láttál, kivéve hogy most be van építve egy nagyon sajátos állapot objektumba ahelyett, hogy egy *Game* osztályon belül legyen.

A játék osztály

Végül itt van a régi *Game* osztály, amely a belépési pontot nyújtja a programodnak. Némely részei sokat változtak, míg mások egyáltalán nem. Először is itt van az adat:

```
public class Game : Form
{
        static string gametitle = “Demo 10.1 - Framework V3”;
        public static DeviceOptions options = new DeviceOptions();
        public static DeviceBlock devices = new DeviceBlock();
        static System.Collections.Stack states = new System.Collections.Stack();
        static GameState state = null;
< kivágás >
}
```

A játék címe van itt újra, de minden más különböző.

A játék opciók és egy eszköz blokk (mely tartalmazza az összes eszköz opciódat és eszközeidet) szintén itt vannak. Az eszközök és eszköz opciók public-static állapotúak, úgyhogy minden el tudja érni ezeket a játékodban; így ha programod bármely részének szükséges egy grafikus eszköz, meg tudja kapni a *Game.devices.Graphics* elérésével. Míg általában rossz gyakorlat az, ha globális objektumaid vannak, ez ezesetben nem nagy ügy, mert nincs sok való-világbeli helyzet, melyben egy játékosnak szükséges hozzáférnie eszközök különböző készleteihez ugyanazon gépen.

Vannak állapot objektumok is: állapotok egy verme és a *state*, mely tárolja a jelenlegi állapotot. A *state* csak egy segítő, amely megkönnyíti, hogy megkapd az érvényes állapotot. Ahelyett, hogy folyamatosan kódolnád, hogy *(GameState).states.Peek()*, elegendő azt írnod, hogy *state*.

Állapotok változtatása

Két függvény van, amelyek szabályozzák az állapotok változtatását: *StateChange* és *AddState*. A *StateChange* fog egy *GameStateChange*

objektumot, és megváltoztatja az állapotot azt az objektumot használva; az *AddState* pedig egyszerűen fog egy *GameState* objektumot és átkapcsol rá.

Itt a *StateChange*:

```
void StateChange( GameStateChange change )
{
	// a jelenlegi állapot megszakadt, le kell szedni
	if( change.Terminated )
		states.Pop();
	// előmozdítani az új állapotot, ha van ilyen
	if( change.NextState != null )
		states.Push( change.NextState );
	// ha van állapot a veremben, a tetejére állítani
	if( states.Count != 0 )
		state = (GameState)states.Peek();
	else
		state = null;
}
```

A kód eléggé magától értetődő. Az egyetlen része, ami miatt lehet, hogy aggódsz, az utolsó. Ha a veremben nincs több állapot, akkor a *state* értéke *null* lesz. Ez azt jelenti, hogy a játéknak vége, és aztán a játék ciklus ellenőrizni fogja, hogy lássa, a *state* értéke *null*-e. Látni fogod ezt a viselkedést később.

A másik függvény:

```
void AddState( GameState newstate )
{
	states.Push( newstate );
	state = newstate;
}
```

A játék ciklus

A játék ciklus nagyon hasonló az előzőleg látott ciklusra a régi keretrendszerből, kivéve hogy állapotokat használ a kemény-kódolású függvények helyett. Vess rá egy pillantást:

```
public void Run()
{
	// egy állapotváltozás meghatározása így nem hozunk létre folyamatosan
	// egyet a vermen
	GameStateChange result = null;
	// ciklus amíg az állapot érvényes
	while( state != null )
	{
		// jelenlegi színhely renderelése
		state.Render();
		// csak folyamat bemenet amíg fókuszban van
		if( this.Focused == true )
			state.ProcessInput();
		// animáció egy képkockájának feldolgozása
		result = state.ProcessFrame();
		// állapotváltás, ha kérésre került
		if( result != null )
			StateChange( result );
		// összes esemény kezelése
		Application.DoEvents();
	}
}
```

Összességében ez a játék ciklus nagyon hasonlít a régi keretrendszer *Run* függvényéhez. A fő eltérés az, hogy a globális *Render*, *ProcessInput* és *ProcessFrame* függvények hívása helyett a jelenlegi állapot objektumokon hívod őket.

Tipp: egy való-világbeli helyzetben az *Application.DoEvents*-et valószínűleg sokkal kisebb arányban akarod hívni egy külön szálban. Ennek oka, hogy a függvény sok memóriaelosztást csinál, és így jelentős lassulást okoz a játékodban.

A belépő pont

A keretrendszer utolsó része a belépő pont, amely betölti a játékot és futtatja:

```
static void Main()
{
	Game game;
	try
	{
		// a setup form mutatása eszköz lehetőségek begyűjtéséhez
		Setup setup = new Setup( options );
		setup.ShowDialog();
```

```
                // egy új game form létrehozása
                game = new Game();
                // az eszközök inicializálása
                devices.Initialize( options, game );
                // a játék objektumok inicializálása
                InitializeGlobalResources();
                // a játék állapot inicializálása
                game.AddState( new SampleState() );
                // mutatni a form-ot és futtatni
                game.Show();
                game.Run();
        }
        catch( Exception e )
        {
                MessageBox.Show( “Error: “ + e.Message + “\n” + e.ToString());
        }
}
```

Ez nagyon hasonló a keretrendszer régi változatához, néhány kulcsfontosságú változással. Az első változás, hogy a *Setup* form létrejön azért, hogy felhasználói preferenciákat gyűjtsön. Amint ez megvan, a játék létrejön, az eszközök inicializálódnak, a globális erőforrások inicializálódnak, és a demó elkezd futtatni egy *SampleState*-et.

Generic Space Shooter 3000

A *Generic Space Shooter 3000* azon az új keretrendszeren alapul, amelyről tanultál, és amely könnyen tervezhetővé tesz egy játékot.

Játék objektumok

A következő alfejezetekben részletezni fogom a változatos játékobjektumok mindegyikét, amelyek létrejönnek és használtak a *GSS3K* folyamán.

A betöltő

A legelső objektum, amit a *GSS3K* számára létrehoztam, az egy játék objektum betöltő, amely betölti az összes játék erőforrást. A *GSS3K*-ban a játék betöltő egésze keményen kódolt, amely valójában nem a legjobb módszer, de működik és megfelel ezen könyv céljainak.

Tipp: a játék erőforrásaid kemény-kódolása helyett jobb, ha vannak erőforrás fájl listáid, melyek egyszerű állományok, amik elmondják a játéknak, hogy mely erőforrásokat kell betöltenie. Még jobb, ha létrehozod a saját archív fájlodat, mely tárolna minden szükséges információt egy nagy állományban, és elmondja a játéknak, hogy hogyan kell ezeket betölteni. A legtöbb korszerű játék használja ezt a megközelítést.

Ez a betöltő segítő függvényeket és statikus változókat tartalmaz, amelyek sablonokként lesznek használhatók a játékban.

Itt van az osztály egy része az összes kód kivágásával:

```
public class GameObjectLoader
{
        // az összes textúra meghatározása
        public static D3D.Texture[] ShipTextures;
        public static D3D.Texture[] ProjectileTextures;
        public static D3D.Texture[] PowerupTextures;
        public static D3D.Texture[] PowerBars;
        public static DS.SecondaryBuffer[] Bloops;
        public static DS.SecondaryBuffer[] FiringSounds;
        public static DS.SecondaryBuffer[] Explosions;
```

Az osztálynak számos textúratömbjei és hang pufferei vannak. Ahogy elképzelnéd, ezek tartalmaznak textúrákat és hang puffereket az összes objektum számára a játékban. Az összes ilyen objektum a következő függvények használatával töltődik be:

```
        // az összes objektum betöltése
        public static void LoadObjects()
        static void LoadTextures()
        static void LoadSounds()
        // segítők:
        public static D3D.Texture LoadTexture( string filename )
        public static DS.SecondaryBuffer LoadSound( string filename )
```

Az utolsó két függvény általad ezelőtt már látott kódot tartalmaz; ezek csak a textúrák és hangok betöltésének egyszerűsítésére léteznek, így neked nem kell összezavarnod a kódodat milliónyi paraméterre való emlékezéssel. Szándékozom megmutatni a *LoadObjects* függvényt később egy kicsit; most hadd mutassam meg az osztály többi részét:

```
        public static Powerup RandomPowerup()
        // statikus sablonok:
        public static Spaceship Player = new Spaceship( 100, 200.0f, 0.5f, 0 );
        public static Spaceship Enemy0 = new Spaceship( 5, 0, 0.0f, 10 );
        public static Spaceship Enemy1 = new Spaceship( 10, 0, 0.0f, 15 );
        public static Spaceship Enemy2 = new Spaceship( 15, 0, 0.0f, 20 );
        public static Spaceship[] Enemies =
                new Spaceship[3] { Enemy0, Enemy1, Enemy2 };
        public static Projectile Laser = new Projectile();
        public static Projectile Plasma = new Projectile();
        public static Projectile Missile = new Projectile();
}
```

A *RandomPowerUp* visszatér egy új véletlen *PowerUp* objektummal, amit később kapok meg. A többi felsorolt objektum mind sablon, a statikus *SpaceShip*-ek és *Projectile*-ek azok, amik használva lesznek aktuális játék objektumok létrehozására.

Amint azt a kódban láthatod, négy különböző fajta űrhajó és három különböző fajta lövedék van. Az űrhajóknak használt értékek később nyilvánvalóvá válnak, mikor megmutatom a *Spaceship* osztályt.

Átmenetileg, megmutatom a *LoadObjects* függvény egy kis részét:

```
public static void LoadObjects()
{
        LoadTextures();
        LoadSounds();
        // játékos textúrák beállítása és méretezése
        Player.Texture = ShipTextures[1];
        Player.Scale = 0.25f; Player.Center();
        Player.AddWeapon( new LaserWeapon() );
        < kivágás >
}
```

A statikus *Player* objektum betöltéskor jön létre, amikor a programod fut. Mivel a *Player* statikus, létrejön és a konstruktora hívódik még azelőtt, hogy a programod futtatja a *Main* függvényt.

Tartsd észben, hogy egy *Player* az egy *Spaceship*, ami egy *GameObject*, amely viszont egy *Sprite*, ami egy *Texture* objektumot tartalmaz. Nem hozhatsz létre egy textúrát egy grafikus eszköz nélkül, és nem akarsz létrehozni egy grafikus

eszközt, mielőtt beállítási lehetőségeket kapsz a felhasználótól, ami azt jelenti, hogy a *Player* objektum létrejön mielőtt még a textúrák betöltődtek a játékba. Ez a sajnálatos eredménye annak igénylésének, hogy a játék menjen vissza és később állítsa be a *Player* textúrát, mely pontosan az, amit a *LoadObjects* függvény csinál.

Mikor létrehozod a *Player* objektumot a következő kóddal:

```
public static Spaceship Player = new Spaceship( 100, 200.0f, 0.5f, 0 );
```

akkor csak az energia, sebesség, pajzsok és pontszám változókat állíthatod be. A textúráját nem tudod beállítani, mert a textúrák még nincsenek betöltve. Ez az, ahol a *LoadObjects* függvény bejön. Ez betölti az összes textúrát és hangokat, aztán beállítja a *Player* textúráját és méretét, majd elmondja a *Player*-nek, hogy állítsa magát a középpontba, és hozzáad egy *LaserWeapon*-t.

Megjegyzés: az űrhajó textúrák 256x256-osak, melyek hatalmasak a sprite-oknak. A GSS3K számára 25 százalékkal méreteztem a kiterjedést, 64x64-essé téve őket. Ez az a fajta adat, amit lemezen kell tárolni, és nem keményen kódolni a játékba. Ez csak egy „gyors és piszkos" megoldás.

A függvény többi része hasonló; betölti a többi űrhajót és a lövedék sablonokat.

Az alap játék objektum osztály

Érintettem ezt korábban: a játék objektumok legtöbbje (az űrhajók, lövedékek és erősítők különösen) megoszt sok közös tulajdonságot, mint fizikai adatot és sprite adatot. Ezért logikus létrehozni egy alap osztályt az összes ilyen tárgynak. Itt is van, a kivágott kód nélkül:

```
public class GameObject : Sprite, System.ICloneable
{
        // adat:
        bool destroyed = false;
        float vx, vy, ax, ay;
        System.Drawing.Rectangle bounding;
         // tulajdonságok:
        public float VX
        public float VY
        public float AX
        public float AY
```

```
        public bool Destroyed
        // függvények:
        public bool Collide( GameObject other )
        public void CalculateBounding()
        public void Move( float timedelta )
        public object Clone()
}
```

Ahogy korábban említettem, az osztálynak van sebességi és gyorsítási adata, valamint egy Boolean típusú változó, amely jelöli, hogy meg van semmisítve vagy sem. Ezen értékek mindegyikének vannak tartozék tulajdonságai is.

Minden játék objektumnak van egy *System.Drawing.Rectangle*-je is, mely tartani fog egy téglalapot, amely a tárgyat jelképezi a világ térben. Az ok, amiért mindegyiküknek van egy téglalapja, hogy a Rectangle osztálynak van *foglalat vizsgálat*a – hívhatsz egy függvényt látni azt, hogy egy téglalap átfed-e egy másikat. Ha azt csinálja, akkor tudod, hogy az objektumok ütköztek; ez az ismeret különösen hasznossá válik később, amikor meg akarod határozni, hogy egy lézer eltalált egy hajót, vagy egy hajó felvett egy erősítőt, és így tovább.

A *Collide* függvény ellenőrzi a téglalapokat, hogy vajon ütköztek-e, de van egy dolog: a függvény nem számítja újra a téglalapokat neked. Ezt saját magadnak kell megtenned, a *CalculateBounding* függvény használatával. Majd később belemegyek a részletekbe, amikor az ütközési rendszert mutatom meg neked.

A *Move* függvény az egyszerű fizikai számításokat hajtja végre az objektumon, egy idő értéken alapulva:

```
public void Move( float timedelta )
{
        VX += AX * timedelta;
        VY += AY * timedelta;
        X += VX * timedelta;
        Y += VY * timedelta;
}
```

Kiszámítódik az új sebesség és aztán az új hely, mindkettő az átadott *time delta* értéken alapulva. Az idő érték a jelképezett időben (másodpercekben) adódik át, mióta az objektum utoljára elmozdult.

Az utolsó függvény a *Clone*, melyet az *ICloneable* határfelületből való örökléssel kapsz. Bár, neked magadnak kell végrehajtanod:

```
public object Clone( )
{
        return MemberwiseClone();
}
```

Ez egyszerűen hívja a *MemberwiseClone* függvényt, melyet az *Object* osztályból kapsz. A *MemberwiseClone* egyszerűen másolja az összes hivatkozását az objektumoknak, majd elhelyezi őket egy új objektumba és visszatér azzal.

Megjegyzés: a hivatkozások másolásának folyamata egy új objektumba és az azzal való visszatérés *sekély másolás*ként ismert, mert minden csak másolódik egy szinten. Hadd adjak egy példát: mivel a *GameObject* osztály örököl a *Sprite*-ból, van egy *Direct3D.Texture* hivatkozása abban. Ez a sekély másolás létre fog hozni egy vadonatúj *GameObject*-et, de az az új tárgy azonos textúra objektumra fog mutatni, mint amit az eredeti csinál. A sekély másolásai folyamat ellentéte a *mély másolás*. Egy játék objektum mély másolása tulajdonképpen létrehozna egy vadonatúj textúrát, ami megegyezik az eredeti textúrával, és az új objektum mutatni fog az új textúrára ahelyett, hogy lenne két objektum ugyanazt a textúrát mutatva. Neked magadnak kell végrehajtani a mély másolásokat, mert a C# nem nyújt egy önműködő módszert ehhez.

Az űrhajó osztály

Az űrhajók a játék objektumok legösszetettebbjei, és bennük van a legtöbb adat. Itt az adat listázása:

```
public class Spaceship : GameObject
{
        float energy;
        float speed;
        float shields;
        float nextfire = 0;
        System.Collections.ArrayList weapons = new System.Collections.ArrayList();
        int currentweapon = 0;
        int score;
< kivágás >
}
```

Az összes adat már részletezve lett ezelőtt, mikor végigmentem azon, hogy milyen adatai lesznek a játék objektumoknak. Néhány változó, amellyel a hajók

rendelkeznek az energia, a sebesség, a pajzsok, a következő tüzelési idő, a fegyverek egy tömb listája, a jelenlegi fegyver indexe, és a hajó pontszáma.

Ezen változók egyike sem publikus, és ennek jó oka van: az űrhajók használnak tulajdonságokat annak kényszerítésére, hogy bizonyos értékek ne menjenek adott értékek alá vagy fölé. Például egy hajó energiája nem mehet 100 fölé, és ha eléri 0-t vagy annál kisebbet, akkor megsemmisül. Továbbá, bármikor csökken egy hajó energiája, a kár mennyiségét méretezni kell a jelenlegi pajzs érték alapján.

Ez sok extra számítás; minderre az *Energy* tulajdonság használatával ügyelek:

```
public float Energy
{
	get { return energy; }
	set
	{
		// kitalálni, hogy milyen sok energia változik
		float delta = value - energy;
		// energia csökkenés, pajzs elnyelés számítása
		if( delta < 0 )
		{
			delta *= (1.0f - Shields);
			Shields -= 0.02f;
		}
		energy += delta;
		// megbizonyosodni, hogy az energia nem megy 0 alá vagy 100 fölé
		if( energy <= 0 )
		{
			Destroyed = true;
			energy = 0;
		}
		if( energy > 100.0f )
			energy = 100.0f;
	}
}
```

Ez sok kód, de ez biztosítja a szabályok betartását. A *get* kód meglehetősen egyszerű; csupán visszatér a nyers energia értékkel.

A *set* kód az, ahol az összes varázslat elkészül. Először is kitalálja a *delta* értéket, vagy hogy milyen sok energia változik. Ez különösen fontos, mert ha az energia csökken, akkor a pajzsoknak kellene kapniuk valamit arról a kárról. Ha a pajzsok

megkapják a kárt, akkor a *delta* érték szorzódik a pajzsok elnyelési arányával. Ha például a pajzsok 1.0-n vannak, akkor a *delta* 0-val szorzódik, ami pontosan 0 kárt okoz az energiának; akkor a pajzsok teljesítménye 2 százalékkal csökken. Azután a delta hozzáadódik a nyers energiaértékhez, és a függvény ellenőrzi, hogy lássa, az energiaszint vajon 0 alá ment vagy 100 fölé, amely esetekben további műveleteket kell végrehajtani. Ha az energia 0 alá megy, akkor a hajó megsemmisül, és az energia visszaáll 0-ra (a tisztaság kedvéért). Ha az energia 100 fölé ment, akkor visszaállítódik 100-ra.

Megjegyzés: találhattál volna módot ráadás energia használatára, inkább minthogy hagyd veszendőbe menni. Talán lehetne egy különleges „tartalék tartályod", ami felszerelhető a hajóra, melybe az összes extra energia belemehet. A lehetőségek igazán végtelenek.

A *Shields* tulajdonság egyszerűbb; az csak megbizonyosodik arról, hogy az energia nem mehet 0 alá vagy 1 fölé. A *CurrentWeapon*-nak csak *get* tulajdonsága van (nincs *set* függvény); egyszerűen csak belemegy a *weapons* tömbbe, és visszatér egy hivatkozással a jelenlegi fegyverre.

Itt a függvények többi része az osztályban; a kód eltávolítva:

```
public void NextWeapon()
public void PreviousWeapon()
public void AddWeapon( Weapon weapon )
public Projectile[] Fire( float time )
```

Az első két függvény átváltja a jelenlegi fegyvert az előző vagy a következő fegyverre. Az *AddWeapon* egyszerűen hozzáad egy új *Weapon* objektumot a fegyverekhez, a *Fire* pedig kéri a jelenlegi fegyvert egy lövedéktömb számára – de csak ha a hajó tud tüzelni. Ha éppen nem tud lőni a hajó, akkor a *null* érték tér vissza. Itt a kódja a függvénynek, amely ezt ellenőrzi:

```
public Projectile[] Fire( float time )
{
        // nincsenek fegyverek
        if( weapons.Count == 0 )
                return null;
        // az idő ellenőrzése
        if( time >= NextFire )
        {
                // megkapni a fegyvert
```

```
                Weapon w = CurrentWeapon;
                // tüzelésidőzítő alaphelyzetbe állítása
                NextFire = time + w.Delay;
                // tudatni a fegyverrel, hogy lőjön
                return w.Fire( this );
        }
        // nem tud még lőni
        return null;
}
```

A kód legfontosabb része az, ahol az idő ellenőrződik. A fegyverek csak bizonyos időközönként tudnak lőni, így ha a kódod azt mondja a hajónak, hogy tüzeljen, mielőtt az képes erre, akkor a hajónak azt kell mondania, hogy „Semmiképpen!”, miközben toppant a lábával. A fegyver osztálynak kell egy hivatkozás a *Spaceship*-re, amikor tüzel, mert bizonyos információk szükségesek neki, mint a jelenlegi sebesség és gyorsulás, azért, hogy létrehozhassa a lövedékeit. Ezenkívül a lövedékeknek tudniuk kell, hogy ki lőtte ki őket, hogy pontokat adhassanak ezeknek, amikor valami megsemmisül. Ez az, amiért egy *this* hivatkozás adódik a tüzelés függvénybe.

Lövedékek

A lövedékek igazán egyszerű objektumok:

```
public class Projectile : GameObject
{
        int damage;
        Spaceship owner = null;
<kivágás>
}
```

A *Damage* és *Owner* tulajdonságok egyszerűen megkapják és beállítják a károkozás és a tulajdonos értékeit. Minden más, ami egy lövedéknek kell, az a *GameObject*-ből öröklődik.

Erősítők

Az erősítők még egyszerűbbek, mint a lövedékek:

```
abstract public class Powerup : GameObject
{
	public Powerup()
	{
		this.VY = 50;
	}
	public abstract void DoPowerup( Spaceship s );
}
```

Az erősítők beállítják az alapértelmezett Y sebességüket 50-re, ami azt jelenti, hogy lefelé gördülnek 50 képpontot másodpercenként, elég lassan, hogy a játékosnak legyen lehetősége elkapni őket. Meghatároznak egy elvont *DoPowerup* függvényt is, mely véghezviszi egy űrhajó megerősítését. Nézd végig a következő két erősítés példát.

Energiaerősítők

Az első példa egy energiaerősítő, amely energiát ad egy hajónak:

```
public class EnergyPowerup : Powerup
{
	public EnergyPowerup()
	{
		this.Texture = GameObjectLoader.PowerupTextures[0];
		this.Scale = 0.5f;
		this.Center();
	}
	public override void DoPowerup( Spaceship s )
	{
		s.Energy += 20.0f;
	}
}
```

Egy energiaerősítő a 0-s erősítő textúrát használja, és a textúra mérete 50 százalékosra van méretezve (a textúrák 64x64-esek, így az erősítők 32x32-esek).

A *DoPowerup* függvény egyszerűen 20 energiát ad egy hajónak, és ennyi.

A pajzs és sebesség erősítők majdnem azonosak az energiaerősítőkkel; ezek csak pajzsot és sebességet adnak az űrhajónak.

Fegyver erősítők

A második példa – és a negyedik erősítő típus – a *WeaponPowerup*, amely egy új fegyvert ad az űrhajónak:

```
public class WeaponPowerup : Powerup
{
<kivágás>
	public override void DoPowerup( Spaceship s )
	{
		// kitalálni, mennyi fegyvere van az űrhajónak
		switch( s.Weapons )
		{
			case 1:
				s.AddWeapon( new DoubleLaserWeapon() );
				break;
			case 2:
				s.AddWeapon( new PlasmaWeapon() );
				break;
			case 3:
				s.AddWeapon( new DoublePlasmaWeapon() );
				break;
			case 4:
				s.AddWeapon( new MissileWeapon() );
				break;
			case 5:
				s.AddWeapon( new DoubleMissileWeapon() );
				break;
			case 6:
				s.AddWeapon( new LaserSpreadWeapon() );
				break;
			case 7:
				s.AddWeapon( new AnihilatorWeapon() );
				break;
		}
	}
}
```

Én kivágtam a konstruktort; az egyszerűen csak előhozakodott ugyanazzal a kóddal, amit ezelőtt már láttál. A fegyver erősítő ellenőrzi, hogy mennyi fegyvere van már az űrhajónak, és attól függően, hogy mennyi van, hozzáadja egy eltérő típus új fegyverét. Tehát ha egy hajónak csak egy fegyvere van, akkor a *WeaponPowerup* hozzáad egy kettős-lézer fegyvert, ha kettő fegyvere van, akkor hozzáad egy plazma fegyvert, és így tovább.

Megjegyzés: ez mind keményen kódolt, ami – mint ezelőtt már említettem – rugalmatlan. Egy jobb módszer ennek elvégzésére, ha volna egy fejlettebb listád mindenfajta űrhajóról, vagy lenne különböző erősítő minden fegyverfajtához. Például egy lézer erősítő tudott volna adni egy kettős-lézer fegyvert, aztán egy háromszoros lézer fegyvert, majd egy lézerszórót, és így tovább. Szabadon eldöntheted, hogy mit akarsz.

Fegyverek

A *GSS3K* fegyverrendszere meglehetősen rugalmas, mert minden fegyver elvonatkoztatott a saját osztályába. Programozhatod mindegyik osztályt, hogy csináljon bármit, amit akarsz.

Itt az alap *Weapon* osztály kód nélkül:

```
public abstract class Weapon
{
        // adat
        float delay = 0;
        DS.SecondaryBuffer sound;
        string name;
        // tulajdonságok
        public float Delay
        public DS.SecondaryBuffer Sound
        public string Name
        // konstruktor
        public Weapon( float delay, string name )
        public Projectile[] Fire( Spaceship owner )
        protected abstract Projectile[] CustomFire( Spaceship owner );
        protected void SetupProjectile(
                Projectile p,
                float vx, float vy,
                float ax, float ay,
                float offsetx,
                int damage,
                Spaceship owner )
}
```

Minden fegyver tudja, hogy meddig tart az újratöltése, köszönhetően a *delay* változónak. A fegyverek hangot is kiadnak tüzeléskor – ezért van nekik egy *DirectSound* másodlagos pufferük.

Az érdekes részek a *Fire*, *CustomFire* és *SetupProjectile* függvények. A *Fire* egy függvény, amely gondoskodik néhány „háztartási” részletről:

```
public Projectile[] Fire( Spaceship owner )
{
	// hang lejátszása
	Sound.Stop();
	Sound.Play( 0, DS.BufferPlayFlags.Default );
	// visszatérés egy egyéni tüzelési cselekvéssel
	return CustomFire( owner );
}
```

Ez lejátssza a hangot (abban az esetben, ha már lejátszódik, akkor megállítja), és aztán hívja a *CustomFire*-t tulajdonképpen visszatérni egy lövedéklistával. Ez úgy történik, hogy a *CustomFire* függvényednek (amely itt elvont) nem kell emlékeznie a tüzelési hang lejátszására. Meg tudod csinálni, hogy a saját fegyver objektumaid további hangokat játsszanak le, ha szeretnéd.

A *SetupProjectile* egy segítő függvény, amely önműködően beállít egy csomó információt, mint sebesség, gyorsulás, károkozás, és így tovább, egy lövedék objektum számára. Ez nem igazán olyan fontos, de ha akarsz, akkor belenézhetsz a CD-n; a Weapons.cs állományban található.

A saját fegyvereid

Meghatároztam egy csomó szokásos fegyvert neked használni, de csináld meg szabadon a sajátjaid. Itt egy példája a legegyszerűbbnek:

```
public class LaserWeapon : Weapon
{
	// tüzelés minden 1/2 másodpercben, a neve “Lasers”, és a 0. tüzelési hangja van
	public LaserWeapon()
		: base( 0.5f, “Lasers” )
	{
		Sound = GameObjectLoader.FiringSounds[0].Clone( Game.devices.Sound );
	}
	protected override Projectile[] CustomFire( Spaceship owner )
	{
		Projectile[] values = new Projectile[1];
		values[0] = (Projectile)GameObjectLoader.Laser.Clone();
		SetupProjectile( values[0], 0, 400, 0, 0, 0, 5, owner );
		return values;
	}
}
```

Ez a kód igényel egy kis magyarázatot. A konstruktor létrehoz egy új hang puffert a létező *FiringSound[0]* puffer objektum megkettőzésével; ez kész van, mert bármilyen adott hang puffer egy időben csak egy hangot tud csinálni. Ha két fegyvered van, amik azonos hang puffert használnak, és majdnem ugyanabban az időben lőnek, akkor csak az egyikük fog hangot kiadni.

Azért, hogy a hang megkettőződjön, mikor mindkét fegyver tüzel, a puffert is meg kell kettőzni.

A CustomFire függvény létrehozza lövedékek egy új tömbjét, amely csak egy lövedéket tartalmaz, ami a *Laser* objektumból van klónozva. Most, hogy van egy alap lézer lövedéked, minden amit tenned kell, hogy beállítsd a többi értéket rajta; ez a *SetupProjectile* által végezhető el. Ebben a konkrét esetben a *SetupProjectile* létrehoz egy lézert, amely másodpercenként 0 képpontnyit halad x irányba, és 400 képpontnyit másodpercenként y irányba, van 0 gyorsulása x és y iránynak, egy 0 x eltolása (majd látni fogod, hogy ez mire való), és 5 kárt csinál.

Végül a lövedék tömb visszaadódik.

Itt egy még összetettebb példa:

```
public class DoubleLaserWeapon : Weapon
{
<kivágás>
        protected override Projectile[] CustomFire( Spaceship owner )
        {
                Projectile[] values = new Projectile[2];
                values[0] = (Projectile)GameObjectLoader.Laser.Clone();
                values[1] = (Projectile)GameObjectLoader.Laser.Clone();
                SetupProjectile( values[0], 0, 400, 0, 0, 28, 5, owner );
                SetupProjectile( values[1], 0, 400, 0, 0, -28, 5, owner );
                return values;
        }
}
```

Ez létrehoz egy „kettős lézert”, amely két lövedék. A fő különbség az x eltolási paraméterek, mely 28-ra és -28-ra van állítva, ami azt jelenti, hogy az első lövedék elmozdult 28 képpontnyit jobbra, a második pedig 28 képpontnyit balra.

Vannak további fegyverek meghatározva, mint plazma, kettős plazma, rakéták, kettős rakéták, lézerszóró, és a megsemmisítő fegyver is. Játszhatsz a játékkal vagy belenézhetsz a kódba, hogy lásd, hogyan működnek ezek.

Az állapotok a GSS3K-nak

A játéknak négy különböző állapota lesz. Az első az indítási állapot, mely a címképernyőt mutatja. A következő állapot a kötelező „szirupos játéktermi lövöldözős történet”, amely megnyitja a játékot egy nagyon homályos háttér történettel. A legfontosabb állapot a tényleges játék állapot, mely kezeli az összes objektumot és fizikát és minden mást. Az utolsó állapot gondot fordít a súgó menüre, mutatva a felhasználónak, hogy mely gombok használatosak a különböző feladatok elvégzésére.

Az összes állapot a GSS3KStates.cs állományon belül tárolódik.

Az indító állapot

Az indító állapot nagyon egyszerű; minden, amit csinál, hogy betölt egy textúrát a lemezről, és néhány másodpercig mutatja. Ebből az állapotból úgy is illene lehetni kilépni, ha a felhasználó lenyom egy bizonyos gombot – nem akarhatod, hogy a felhasználó hosszabb ideig nézze a címképernyőt, mint ameddig szeretné. (Én legalábbis utálom, ha egy játék ezt csinálja velem!)

Az állapotnak szüksége lesz egy időzítőre, egy Boolean változóra annak eldöntéséhez, hogy a felhasználó vajon ki akar-e lépni, egy textúrára, és néhány csúcspontra a textúra mutatásának használatához:

```
public class GSS3KStartup : GameState
{
        Timer timer;
        bool done = false;
        D3D.Texture loadscreen;
        D3D.CustomVertex.TransformedTextured[] vertexes = null;
<kivágás>
}
```

Amint látod, ez megfelelően örököl a *GameState* osztályból, az összes csodálatos beépített hasznosságait megszerezve. Csupán a következők meghatározását követeli meg ez az osztály:

```
public GSS3KStartup()
public override GameStateChange ProcessFrame()
protected override void CustomRender()
protected override void KeyboardUp( DI.Key key )
protected override void MouseButtonUp( int button )
protected override void JoyButtonUp( int button )
```

Kivágtam a kódot a függvényeknek. A konstruktor, a *GSS3KStartup*, nem csinál semmi látványosat; egyszerűen csak betölt egy textúrát a lemezről, és beállítja a csúcspontokat mutatni azt a képernyőn.

A *ProcessFrame* egészen egyszerű:

```
public override GameStateChange ProcessFrame()
{
	// visszatérés 10 másodperc vagy gombnyomás után
	if( timer.Time() >= 10.0f || done )
		return new GameStateChange( true, new GSS3KIntro() );
	// a szál altatása, hogy megelőzzük a felesleges processzormunkát
	System.Threading.Thread.Sleep( 1 );
	return null;
}
```

Alapvetően a *ProcessFrame* csak vár az időzítőre, míg eltelik 10 másodperc, vagy amíg a játékos lenyom egy gombot (amely a *done* változót igazra állítja, amint azt mindjárt látni fogod). Ha ezen feltételek egyike teljesül, akkor egy új játék állapot tér vissza, közölve a játékkal, hogy semmisítse meg ezt az állapotot és kapcsoljon át a *GSS3KIntro* állapotra.

Nincs igazán szükséged arra, hogy lásd a *CustomRender* függvényt – minden, amit ez csinál, hogy kirajzolja a textúrát a képernyőre azon a módon, amit a Demo 7.4-ben láttál.

Az adatbeviteli függvények teljesen egyszerűek:

```
protected override void KeyboardUp( DI.Key key )
{
	done = true;
}
```

```
protected override void MouseButtonUp( int button )
 {
        done = true;
}
protected override void JoyButtonUp( int button )
{
        done = true;
}
```

Amikor egy billentyűzet gomb, egér gomb, vagy botkormány gomb felengedődik, a *done* változó *true* (igaz) értékű lesz, így az állapot tudja, hogy ki kell lépnie. Egyszerű, ugye?

Megjegyzés: amikor állapotokat változtatsz, általában jobb ötlet egy gomb felengedett mivoltát ellenőrizni, mint hogy le van-e nyomva. Bármikor kapcsolsz egy vadonatúj állapotra, új adatbeveitel kezelő objektum keletkezik. Ezek azt gondolják, hogy lenyomtál egy gombot, mivel nincsenek korábbi összehasonlítható állapotadataik, és az állapot sokkal gyorsabban megváltozik, mint hogy fel tudnád engedni a gombot. Az egyetlen mód ekörül az, hogy a bevitel ellenőrzők átadják a gomb állapotokat egyik állapotról a másikra; a keretrendszer nem kezeli ezt a jelenlegi állapotában.

A bevezetés állapot

Nem szándékozom sok időt eltölteni ezzel az állapottal, mivel ez igazán csak valami szirupos és összedobtam a játékot úgy, hogy az 1980-as évek játéktermi gépeinek hangulatát adja. Egyszerűen kiír némi szöveget a képernyőre, kérve a játékost, hogy segítsen megvédeni egy űrállomást az űrbéli támadók ellen.

Itt az adat:

```
public class GSS3KIntro : GameState
{
        Timer timer;
        float next;
        bool done = false;
        D3D.Font typefont = null;
        DS.SecondaryBuffer beep = null;
        string fullmessage = @"... BEJÖVŐ ADÁS ...\n
        ... TYRAZIEL KAPITÁNY... KÉREM JÖJJÖN...\n
        ... A GSK53-ALFA ÁLLOMÁST MEGTÁMADTÁK ... \n
        ... SZÜKSÉGÜNK VAN ÖNRE ...";
        string message = "";
        int stage = 0;
```

```
<kivágás>
}
```

A *fullmessage* változó tartalmazza az egész szöveget, a *message* pedig egy rész szöveget. Az ötlet az, hogy fogsz egy szöveges üzenetet egy mélyűri kommunikációból (vagy valami hasonló, ez nem számít); fogod az adást betűről betűre és az kiíródik a képernyőre, ahogy kapod az üzenetet. Ezért szükséges a *stage* – nyomon követi az üzenetet a fogadás szempontjából. Ha *stage* értéke 0, akkor nem kell még fognod egyik részét sem az üzenetnek, ha pedig 10, akkor azt fogtad eddig, hogy „… BEJÖV”, tehát ez kell, hogy legyen a *message* változón belül. A *beep* puffer egy sípoló hangot tartalmaz, amely lejátszódik minden írásjel kiírásakor.

Megjegyzés: a @ jel a kódban talán még új neked. Ez egyszerűen kijelenti, hogy mindent, ami a @ jelet követő idézőjelek között van, szó szerint ki kell olvasni a fájlból. Ez egy könnyű módszer, ami lehetővé teszi neked, hogy hosszú szövegeket több sorra bonts.

Itt a *ProcessFrame* függvény:

```
public override GameStateChange ProcessFrame()
{
	// kilépés a felhasználói gombnyomáskor, irány a játék
	if( done == true )
		return new GameStateChange( true, new GSS3KGame() );
	// egy új betű hozzáadása, ha eljött az ideje, és vannak még
	// hozzáadandó írásjelek hátra
	if( timer.Time() >= next && stage < fullmessage.Length )
	{
		stage++;
		message = fullmessage.Substring( 0, stage );
		next += 0.10f;
		beep.Stop();
		beep.Play( 0, DS.BufferPlayFlags.Default );
	}
	// a szál altatása, hogy megelőzzük a felesleges processzormunkát
	System.Threading.Thread.Sleep( 1 );
	return null;
}
```

A kód alapvetően hozzáad egy új írásjelet a *message* változóhoz minden 1/10. másodpercben, amíg már nincs több betű hátra. Aztán a kód itt várakozik, hogy

a játékos nyomjon egy billentyűt vagy gombot, és ezt követően átkapcsol Játék állapotba.

A Súgó állapot

A Súgó állapot majdnem azonos az Indító állapottal, kivéve, hogy ennek nincs időzítője – csak várakozik, mutatva a súgó képernyőt, amíg a játékos le nem nyom egy billentyűt vagy gombot, melynél megsemmisíti magát. A Súgó állapot a Játék állapot tetején ül, tehát mikor megsemmisül, nem indít egy új állapotot; helyette visszatér a Játék állapothoz.

A Játék állapot

Nyilvánvaló okokból messze ez a legösszetettebb állapot a játékban. Sok dolog van, amikre ügyelni kell; az adatok magyarázatával fogom majd kezdeni.

```
public class GSS3KGame : GameState
{
        Timer timer;
        bool done = false;
        bool help = false;
        bool paused = false;
        System.Random random = new System.Random();
        float nextwave = 0;
        Camera camera = new Camera( GSS3KConstants.Width,
                GSS3KConstants.Height );
        Camera UIcamera = new Camera( GSS3KConstants.Width,
                GSS3KConstants.Height,
                GSS3KConstants.Width / 2,
                GSS3KConstants.Height / 2 );
        Spaceship player;
        System.Collections.ArrayList ships =
                new System.Collections.ArrayList();
        System.Collections.ArrayList projectiles =
                new System.Collections.ArrayList();
        System.Collections.ArrayList powerups =
                new System.Collections.ArrayList();
        // bármely gombirány lenyomva? Joystickok mozogtak?
        // a játékos tüzel?
        bool kl, kr, ku, kd;
        float jx, jy;
        bool firing = false;
        Sprite EnergyBar;
        Sprite ShieldBar;
<kivágás>
```

}

A *timer* és a *done* logikai változók nem újak neked, de a *help* igen. A *help* egyszerűen egy jelzőzászló (flag), amely elmondja az állapotnak, hogy a játékos lekérte megnézni a súgó menüt. A *paused* logikai változó tudatja az állapottal, hogy szünetel vagy sem, a *random* változó pedig egyszerűen tartalmaz egy véletlen szám előállítót. A *nextwave* változó azt az időt tartalmazza, aminél a következő ellenséges hullámnak létre kell jönnie. (Később ebbe részletesebben bele fogok menni.)

Két kamera van létrehozva; egy a játéknak és egy másik a felhasználói felületnek (user interface=UI). Az UI kamera egyszerűen 0,0-ra helyezi a koordinátákat, felbukkanva ezáltal a képernyő bal felső sarkában a középpont helyett.

Megjegyzés: lehetne megjeegyzésed a *GSS3KConstants* osztály használatára. Ez egy egyszerű osztály, amely egy csomó állandót tartalmaz, mint például a játék mező szélessége és magassága, többek között. Ez világosabbá teszi a programodat ahelyett, mintha mindenhová beírnád a 640 és 480 értékeket.

Van egy *Spaceship*, ami jelképezi a játékost, és három *ArrayList*, melyben tárolódnak az összes űrhajók, lövedékek és erősítők.

A kód következő blokkja logikai változókat tartalmaz, amelyek meghatározzák, hogy mely nyíl gombok vannak lenyomva, az utolsó joystick értékeket, és hogy vajon a játékos űrhajója tüzel-e. Látni fogod, hogy mindez hogy működik, mikor megmutatom az adatbeviteli és a számítási szakaszt.

Végül van két kép, egyik az energiacsíkot, másik a pajzs csíkot jelképezi, melyek az UI kirajzolására lesznek használva.

Vegyes függvények

Az osztály tartalmaz sok függvényt és tulajdonságot, amelyek nem nagyon bonyolultak; én csak röviden elmagyarázom ezeket a kód megmutatása nélkül. A *Paused* tulajdonság kapcsolgatja a *paused* logikai változót, és elindítja vagy megállítja az időzítőt, attól függően, hogy milyen értéket változtattál a

tulajdonságon. Ez nagyon hasonló ahhoz, amit már láttál a korábbi keretrendszerekben.

A konstruktor, *GSS3KGame*, létrehoz egy játékos objektumot, és hozzáadja a *ships* listához, aztán létrehozza a két GUI csík képet.

A *LostFocus* függvény (felülírva a *GameState* osztályból) egyszerűen szüneteltetni a játékot. A játékot szüneteltetve akarod, mikor a játékos átkapcsolja az ablakokat.

A *NextWeapon* és a *PreviousWeapon* akkor hívódik, mikor a játékos a fegyverek között akar váltani. Fegyverváltáshoz a játék egyszerűen lejátszik egy hangot, és aztán közli a *player* objektummal, hogy megváltoztassa a jelenlegi fegyverét.

A játék feldolgozás

A *ProcessFrame* függvény sok munkát végez a játékállapot számára. Ez nem is meglepő, mivel itt történik a játék feldolgozása.

Végigvezetlek a kódon lépésről-lépésre, kezdve az állapotváltozásokkal:

```
public override GameStateChange ProcessFrame()
{
	if( done )
		return new GameStateChange( true, null );
	if( help )
	{
		Paused = true;
		help = false;
		return new GameStateChange( false, new GSS3KHelp() );
	}
```

Ha a *done* logikai változó beállítódik, akkor az állapotnak egyszerűen ki kellene lépnie, anélkül, hogy új állapot állítódik be. Ebben az egyszerű demóban az egész program egyszerűen ki fog lépni. Egy még összetettebb rendszerben akarhatnál egy menürendszert, ami a kilépést vezérli.

Ha a *help* logikai változó beállítódik, akkor a játék szünetel, a változó visszaáll hamisra (false), és az állapot megváltozik *GSS3KHelp* állapotra.

Folytatva, a játék ellenőrzi, hogy szüneteltetve van-e, és aztán végrehajtja a feldolgozást:

```
if( !Paused )
{
	// megkapni az eltelt idő mennyiségét
	float time = timer.Elapsed();
	// új ellenséges hullám létrehozása, ha itt az ideje
	if( timer.Time() >= nextwave )
		GenerateEnemyWave();
	// játék feldolgozás
	foreach( GameObject o in ships )
		o.Move( time );
	foreach( GameObject o in projectiles )
		o.Move( time );
	foreach( GameObject o in powerups )
		o.Move( time );
```

Az utolsó képkockaszámítás óta eltelt időmennyiség kinyerődik és tárolódik a *time* változóban. A kód következő része ellenőrzi, hogy az ellenségek egy új hullámát létre kell-e hozni, és ha igen, akkor létrehozza.

Aztán minden objektum feldolgozódik a *GameObject* osztály *Move* függvényét felhasználva, mely végrehajtja a fizikai számításokat.

A következő lépésben a játék ellenőrzi, hogy a játékos nem bitangolt el a képernyőn kívülre, és ha igen, akkor kijavítja a pozícióját:

```
	// most javítjuk a játékos pozícióját
	if( player.X < GSS3KConstants.LeftBound )
		player.X = GSS3KConstants.LeftBound;
	if( player.X > GSS3KConstants.RightBound )
		player.X = GSS3KConstants.RightBound;
	if( player.Y > GSS3KConstants.BottomBound )
		player.Y = GSS3KConstants.BottomBound;
	if( player.Y < GSS3KConstants.TopBound )
		player.Y = GSS3KConstants.TopBound;
```

Megjegyzés: egy még hatékonyabb módszer arról megbizonyosodni, hogy a játékos nem bitangolt el a képernyőn kívülre az lenne, ha csak akkor ellenőriznéd ezeket az értékeket, amikor megváltoztak, ahelyett, hogy ezt a játék ciklus minden ismétlésében megtennénk. Ennél az egyszerű játéknál kisebb teljesítményvesztést tudsz tartani, de tartsd észben, hogy ha elkezdesz csinálni ütközési foglalatokat sok játékobjektum ellenőrzésére csak a játékos helyett, akkor egy jobb módszert kellene találnod.

És végül a számítás utolsó része:

```
                // tüzelés végrehajtása
                DoFire();
                // ütközés ellenőrzés végrehajtása
                DoCollisions();
                // rendben van-e az összes objektum érvényessége
                CheckValidity( ships );
                CheckValidity( projectiles );
                CheckValidity( powerups );
        }
        else
                System.Threading.Thread.Sleep( 1 );
        return null;
}
```

A *DoFire* végrehajtja az összes lövési számítást, a *DoCollisions* elvégzi az ütközésellenőrzést, és a *CheckValidity* végigmegy az összes objektumon, hogy lássa, vajon megsemmisültek-e. Ha igen, a játék előremegy és megsemmisíti őket.

Ha a játék szünetel, akkor a szál „alszik”, így nem kell felhasználnod az összes processzor ciklust. Végül a *null* tér vissza, jelezve, hogy az állapot nem változott.

Tüzelési számítások végrehajtása

Ahogy korábban láttad, a játék becsomagolta egyetlen függvénybe az összes tüzelési számítást: a *DoFire*-be. A játék nagyon egyszerű és nem beszélhetünk semmilyen mesterséges intelligenciáról; a tény az, hogy az ellenséges hajók csak jönnek lefelé és folyamatosan lőnek. Meglehetősen bután néz ki, de működik.

Ez a függvény végig fog menni minden hajón, és közli velük, hogy lőjenek:

```
void DoFire()
{
        Projectile[] plist;
        float time = timer.Time();
        foreach( Spaceship s in ships )
        {
```

Ennél a pontnál a kód végigmegy az összes űrhajón a játékban és végrehajt egy egyszerű ellenőrzést:

```
// csak akkor lőjön, ha nem-játékos
// vagy a “firing” logikai változó igaz értékű
if( firing || (s != player) )
{
```

Alapvetően az egyetlen idő, amikor egy hajó nem fog lőni, mikor a hajó a játékos és a *firing* logikai változó hamis (false). Az összes többi hajó tüzelni fog, mivel ezek a számítógép hajói. Aztán a függvény közli a hajóval, hogy térjen vissza a lövedékei egy listájával, és ha a lista nem *null*, akkor hozzáadja az összes lövedéket a *projectiles* listához:

```
                plist = s.Fire( time );
                if( plist != null )
                {
                        foreach( Projectile p in plist )
                        projectiles.Add( p );
                }
        }
    }
}
```

Ez minden!

Tipp: egy még összetettebb rendszer meghatározná, hogy vajon a hajók eseti elbírálás alapon tüzeltek-e, és nem ellenőrizné minden egyes képkockában, és nem gyakori, hogy egy hajó képkockánként egyszer bárhol tüzeljen. Ennek az egyszerű játéknak ez a rendszer nem olyan hatástalan, de egy nagyobb játéknál akarhatnál felfedezni egy eseményrendszert, amiben a játék várna, amíg egy hajó tud tüzelni, hogy lássa, akar-e vagy sem lőni, ahelyett, hogy folyamatosan kérdeznénk, hogy „Tudsz-e már lőni?”

Objektum érvényesség ellenőrzés

Mikor egy objektum „megsemmisült” ebben a játékban – lézertalálatot kapott és meghalt, fölvette egy űrhajó vagy valami hasonló -, akkor tulajdonképpen nem semmisült meg azonnal. Az csak egy hatalmas, csúnya zavart okozna. Ehelyett az objektum *destroyed* logikai változója *true* lesz, és a játék eltávolítja a megsemmisült objektumot egy későbbi időpontban. A függvény, amely ezt

véghezviszi, a *CheckValidity* függvény, mely fogja a *GameObject*-ek egy *ArrayList* tömblistáját, és eltávolítja belőle mindet, ami megsemmisült:

```
void CheckValidity( System.Collections.ArrayList list )
{
	for( int i = 0; i < list.Count; i++ )
	{
		GameObject o = (GameObject)list[i];
```

Itt van egy ciklus, amely végigmegy minden objektumon a listában, ellenőrizve, hogy vajon megsemmisült-e:

```
		// csak azokon az objektumokon hajtódnak végre ezek az akciók,
		// amelyek kifejezetten megsemmisültek a játék által
		if( o.Destroyed )
			PerformDestroyActions( o );
```

Két módon semmisülhet meg egy objetkum ebben a játékban. Az első mód, amikor kifejezetten megsemmisül (a hajó károsodik a lézerek által, egy erősítőt felszed egy hajó, stb.), mely esetben a tárgy *Destroyed* flag-e beállítódik. Ha ez az eset történik, akkor a játék hívja a *PerformDestroyActions* függvényt azon az objektumon. A másik mód, amikor egyszerűen kimegy a játékmezőről, mely esetben a tárgy nem igazán semmisül meg – csak eltávolítod a játékból.

A kód következő része egyszerűen eltávolítja az összes objektumot, amelyek megfelelnek a paramétereknek:

```
		if( o.Destroyed ||
			o.Y < GSS3KConstants.KillZoneBottom ||
			o.Y > GSS3KConstants.KillZoneTop )
		{
			list.RemoveAt( i );
			i—; // visszamozog egy indexet, minden lefelé mozdul
		}
	}
}
```

Objektumok megsemmisítése

Ha egy tárgy fizikailag megsemmisült, akkor a *PerformDestroyActions* függvény hívódik rajta. Ez csinál még az objektumon némi ráadásszámítást, mielőtt végül

eldobja. Az űrhajóknál a játék kitalálja, hogy létre kell-e hozni egy véletlen erősítőt, és lejátszik egy véletlen robbanás hangot:

```
void PerformDestroyActions( GameObject o )
{
        if( o is Spaceship )
        {
                GameObjectLoader.Explosions[random.Next(3)].Play(
                        0, DS.BufferPlayFlags.Default );
                // ½ esély egy erősítőre
                if( random.Next( 2 ) != 0 )
                {
                        Powerup p = GameObjectLoader.RandomPowerup();
                        p.X = o.X;
                        p.Y = o.Y;
                        powerups.Add( p );
                }
        }
```

Erősítőknél szintén lejátszódik egy hang:

```
        if( o is Powerup )
        {
                GameObjectLoader.Bloops[0].Play(
                0, DS.BufferPlayFlags.Default );
        }
}
```

Egy lövedék megsemmisülésénél nem szükséges semmilyen cselekvés; bár gondolkodhatsz azon, hogy hozzáadsz a jövőben erővisszacsatolásos hatást. Ha egy lövedék megsemmisült, és a játékos hajója volt az, amit eltalált, akkor azt megfontolhatod, hogy a játékos botkormányára csinálsz egy erővisszacsatolás hatást.

Hullámok létrehozása

Az ellenséges űrhajók „hullámait” a *GenerateEnemyWave* függvény állítja elő. Nem szándékozom ebbe túl mélyen belemenni, mert ez nem igazán fontos, ameddig a fogalmakról tanulnod kellene.

Alapvetően a *GenerateEnemyWave* fog egy véletlen egész számot, és aztán létrehozza az új hajók egy listáját, ami ezen a számon alapul:

```
Spaceship[] newships = null;
// egy véletlenszám választása
int pattern = random.Next( 0, 5 );
switch( pattern )
{
        case 0: // egy véletlen hajó
                newships = new Spaceship[1];
                newships[0] =
                        (Spaceship)GameObjectLoader.Enemies[random.Next(3)].Clone();
                newships[0].X =
                        random.Next( GSS3KConstants.LeftBound,
                                GSS3KConstants.RightBound );
                newships[0].Y = GSS3KConstants.SpawnZone;
                newships[0].VY = 200;
                break;
<kivágás>
// hozzáadni a hullámot a hajókhoz
foreach( Spaceship s in newships )
        ships.Add( s );
}
```

Csak egy véletlenszámot mutattam, mert általában hasonló a kód minden számhoz. A kód kifog egy véletlenszámot, és aztán a switch kifejezést használja, hogy a megfelelő kódra ugorjon. A szám, amit itt mutattam, egy véletlen hajót állít elő, aminek 200 képpont másodpercenként a sebessége.

Tipp: akarhatod fontolgatni egy fájlformátum készítését, amely részletezi a véletlenmintákat, és aztán létrehozza a hajókat, felhasználva azt az állomány formátumot ahelyett, hogy mindent keményen kódolnál. Ezen a módon a lehetőségek végtelenek, és könnyedén adhatsz új mintákat. Mindegyik minta jelölhetné, hogy mennyi hajót kell létrehozni, azok relatív helyzetét és így tovább.

Ütközésellenőrzés

Az ütközésellenőrzés egyike e játék legösszetettebb részeinek. Általában véve csak néhány eset van, amiben tisztában szándékozol lenni az ütközéssel, melyek mindegyike egyetlen függvényen belül kezelődik. Először is, szükséges ellenőrizned a hajó-hajó ütközéseket, hogy lásd, a hajód nekiment-e egy másiknak. Aztán a hajó-lövedék ütközéseket is ellenőrizni akarod, hogy lásd, becsapódott-e bármely lövedék bármely hajóba. Végül a játékos-erősítő ütközéseket akarod ellenőrizni.

Jegyezd meg, hogy nem beszéltem hajó-erősítő ütközésekről – a játék nem törődik azzal, ha egy ellenséges hajó ütközik egy erősítővel; az ellenséges hajóknak nem kellene erősítőket kapniuk. Az erősítők jutalmak a játékosnak és csakis a játékosnak, ezért nem kell ellenőrizned minden hajóval az erősítő ütközéseket – csak a játékos hajójával.

Szintén nincsenek erősítő-lövedék ütközések; ez csak az én személyes kedvezményem. Ha fondorlatos akarsz lenni, megadhatod ezt a lehetőséget, hogy találat esetén megsemmisüljenek az erősítők.

Itt a kód:

```
void DoCollisions()
{
	// először kiszámítjuk az összes ütközési téglalapot
	foreach( GameObject o in ships )
		o.CalculateBounding();
	foreach( GameObject o in projectiles )
		o.CalculateBounding();
	foreach( GameObject o in powerups )
		o.CalculateBounding();
```

Első lépésben kiszámítjuk minden objektum ütközési téglalapját. Minthogy a *DoCollisions* függvény egyszer hívódik képkockánként, miután minden objektum elmozdult, frissíteni szükséges az ütközési téglalapokat minden játék objektumnak, mivel a régi téglalapjaik többé nem érvényesek.

Most tudod ellenőrizni a hajó-lövedék ütközéseket:

```
	// lássuk, bármely lövedék  becsapódott-e bármely hajóba
	foreach( Spaceship s in ships )
	{
		// csak nem-megsemmisült  hajókat ellenőrzünk
		if( !s.Destroyed )
		{
			// az összes lövedék ellenőrzése
			foreach( Projectile p in projectiles )
			{
				// csak nem-megsemmisült lövedékeket ellenőrzünk
				// és megbizonyosodunk, hogy a lövedéket egyik hajó
				// sem tulajdonolja
				if( !p.Destroyed && p.Owner != s )
				{
```

```
                        if( s.Collide( p ) )
                        {
                                // egy ütközésünk van!
                                Collide( s, p );
                        }
                }
        } // lövedék ellenőrzés vége
```

Nem akarod, hogy a függvény ránézzen bármely megsemmisült hajóra vagy megsemmisült lövedékre, mert azok az objektumok többé már nem a játék részei (bár még benne lehetnek a tömbökbe). Bármi, aminek a *Destroyed* tulajdonsága be van állítva, az figyelmen kívül lesz hagyva, és egy későbbi időpontban eltávolítódik a játékból (rossz ötlet eltávolítani objektumokat tömbökből, amik jelenleg még használtak).

Azt is ellenőrizni akarod, hogy megbizonyosodj arról, egy lövedék nem tudja eltalálni azt a hajót, amely kilőtte. Ha csinálsz egy ütközést, akkor a *Collide* függvényt kellene hívnod (amit egy kicsit elmagyarázok majd).

Következik a hajó-hajó ütközések:

```
        foreach( Spaceship s2 in ships )
        {
                // csak nem-megsemmisült hajókat ellenőrzünk
                // és hogy a hajó nem önmaga
                if( !s2.Destroyed && s != s2 )
                {
                        if( s.Collide( s2 ) )
                        {
                                // van egy ütközésünk!
                                Collide( s, s2 );
                        }
                }
        } // hajó ütközések vége
} // hajó/lövedék, hajó/hajó ütközések vége
```

Az előző kóddarab leginkább ugyanaz, mint a hajó-lövedék ütközés ellenőrző kód, kivéve hogy lövedékek helyett a kód leellenőriz minden hajót a többivel.

Végül az utolsó kóddarab ellenőrzi, hogy a játékos ütközött-e bármelyik erősítővel:

```
        // erősítők ellenőrzése
        if( s == player )
        {
                foreach( Powerup p in powerups )
                {
                        // csak nem-megsemmisült erősítőket ellenőrzünk
                        if( !p.Destroyed )
                        {
                                if( s.Collide( p ) )
                                {
                                        // van egy ütközésünk!
                                        Collide( s, p );
                                }
                        }
                }
        } // erősítő ellenőrzés vége
    }
}
```

Ez tulajdonképpen rengeteg kód; ennek oka az összes különleges eset ellenőrzés. Egy igazán rugalmas ütközési rendszerben nem léteznének különleges esetek, és a dolgok tudnák, hogyan ütköznének egymással, így tudtál volna egyszerűen egyetlen ütközési ciklust csinálni, amely ellenőriz minden objektumot minden más objektummal. Ehelyett most három ciklusod van minden különleges esetnek. A rendszerválasztás igazán tőled függ; én úgy gondoltam ennek a játéknak, hogy a különleges-eset rendszer jobb, mint az ütközéseket kezelni azon tárgyak között, amelyek nem igénylik az ütközést.

Hajók és lövedékek ütközése

Ha egy ütközés történik, a három *Collision* függvény egyike hívódik. Az első a hajó-lövedék ütközési függvény:

```
void Collide( Spaceship s, Projectile p )
{
        // hit the ship
        s.Energy -= p.Damage;
        // lövedék megsemmisítése; csak egy hajót sebezhet
        p.Destroyed = true;
        // ha a hajó megsemmisült, pontokat adunk a lövedék kibocsátójának
        if( s.Destroyed )
                p.Owner.Score += s.Score;
}
```

A hajó energiája csökken azzal a kármennyiséggel, amennyit a lövedék okoz, és a lövedék megsemmisül.

Ha a hajó elpusztul, akkor a lövedéket kilövő hajó (*p.Owner*) pontszáma növekszik az elpusztított hajó pontjával.

Hajók ütközése

A hajó-hajó ütközések egy stratégia elemet adnak a játékhoz. Ahelyett, hogy hátul maradsz és úgy semmisíted meg a hajókat, hogy lézereket lövöldözöl rájuk, kamikaze módjára nekik is ronthatsz, ha van elég energiád.

Ez az a függvény, amely két hajó ütközésekor hívódik:

```
void Collide( Spaceship s1, Spaceship s2 )
{
	// a kész kármennyiség ugyanaz, mint amennyi energiája
	// van egy hajónak kétszer.
	float e1 = s1.Energy * 2;
	float e2 = s2.Energy * 2;
	s2.Energy -= e1;
	s1.Energy -= e2;
	// pontok módosítása, ha egyik hajó megsemmisül.
	int score1 = s1.Score;
	int score2 = s2.Score;
	if( s1.Destroyed )
		s2.Score += score1;
	if( s2.Destroyed )
		s1.Score += score2;
}
```

Az általános szabály ütköző hajóknak, hogy az egyiknek okozott kár a másik jelenlegi energiszintjének kétszerese. Tehát ha van 100 energiám és belémjön egy 10 energiájú hajó, akkor én okozok neki 200 energiapontnyi kárt, ő pedig nekem 20 energiapontot. Ha valamelyik hajó elpusztul, akkor a túlélő hajó pontszáma megnövekszik a megsemmisült hajó pontszámával.

Hajók és erősítők ütközése

Ez a legkönnyebb ütközési függvény, amely lehetővé teszi hajók ütközését erősítőkkel:

```
void Collide( Spaceship s, Powerup p )
{
	// az űrhajó megerősítése
	p.DoPowerup( s );
	// az erősítő megsemmisítése
	p.Destroyed = true;
}
```

Az erősítő megerősíti az űrhajót és aztán megsemmisül.

Kell adatbevitel!

A kövezkező része a játékállapot kezelésnek az adatbevitel gyűjtése. Szerencsére neked, ez nem mind olyan nehéz.

A billentyűzet

A billentyűzet az alapértelmezett vezérlője a játéknak. Tulajdonképpen én előnyben részesítem a botkormány használatát ehhez a játékhoz, de jaj, nem remélheted, hogy minden játékosodnak lesz egy joystickja, ezért támogatnod kell a billentyűzetet is. Itt a *KeyboardDown* függvény:

```
protected override void KeyboardDown( DI.Key key )
{
	switch( key )
	{
		case DI.Key.Left:
			kl = true; break;
		case DI.Key.Right:
			kr = true; break;
		case DI.Key.Up:
			ku = true; break;
		case DI.Key.Down:
			kd = true; break;
		case DI.Key.Space:
			firing = true; break;
		case DI.Key.LeftControl:
			PreviousWeapon(); break;
		case DI.Key.RightControl:
			NextWeapon(); break;
	}
}
```

A nyíl gombok számára a *kl*, *kr*, *ku* és *kd* változók beállítódnak *true* értékűre, ha azon gombok lenyomódnak. Egy kicsit később látni fogod, hogy működnek ezek; most minden, amit tudnod kell, hogy azok a logikai változók *true* értékűek, ha egy gomb le van nyomva, és *false* értékűek, ha fel vannak engedve.

A bal Ctrl vagy jobb Ctrl gombok nyomogatásával változtatható a játékos jelenlegi fegyvere.

A következő a *KeyboardDown* kiegészítője, a *KeyboardUp*:

```
protected override void KeyboardUp( DI.Key key )
{
	switch( key )
	{
		case DI.Key.Escape:
			done = true; break;
		case DI.Key.P:
			Paused = !Paused; break;
		case DI.Key.F1:
			help = true; break;
		case DI.Key.Left:
			kl = false; break;
		case DI.Key.Right:
			kr = false; break;
		case DI.Key.Up:
			ku = false; break;
		case DI.Key.Down:
			kd = false; break;
		case DI.Key.Space:
			firing = false; break;
	}
}
```

Ez azonkívül, hogy beállítja *false* értékűre a nyíl gombok logikai változóit, ha azok fel vannak engedve, gondoskodik az Escape, a P és az F1 gombokról is. Az Escape okozza a kilépést a játékból a *done* változó *true* értékűre állításával, a P a szünet állapot között kapcsol, és az F1 vált súgó módba.

A botkormány

A joystick adatbevitel is egyszerű:

```
protected override void JoyMoveX( int delta )
{
	jx = delta;
}
protected override void JoyMoveY( int delta )
{
	jy = delta;
}
```

A tengely mozgás függvény egyszerűen elmondja a *jx* és *jy* változóknak a jelenlegi értékét a joystick pozíciónak.

A következő kóddarab kezeli a gombnyomásokat:

```
protected override void JoyButtonDown( int button )
{
	switch( button )
	{
		case 0:
			firing = true; break;
		case 1:
			NextWeapon(); break;
		case 2:
			PreviousWeapon(); break;
	}
}
protected override void JoyButtonUp( int button )
{
	if( button == 0 )
		firing = false;
}
```

A botkormány gombok kezelik a tüzelést (0. gomb) és a fegyverek közötti váltást (1. és 2. gomb).

Adatbevitel számítása

Az előző játékállapot osztályokban nem volt szükség a *GameState.ProcessInput* függvény változtatására. A függvény egyszerűen hívta az adatbevitel ellenőrzőket, amelyek viszont közvetlenül kezelték a billentyűzet/egér/joystick fel/le/tengely függvényeket. Habár nem tudod azt csinálni ezesetben, mert az adatbevitel nincs igazán kezelve. Például amikor a felhasználó lenyomja a Bal Nyíl gombot, a hajó feltételezhetően balra mozdul, de ez nem történik meg.

Helyette a *kl* logikai változó *true* értékű lesz, és semmi más nem történik. A játék feltételezi, hogy a játékos mozgatási számítások később kezelődnek.

A *ProcessInput* függvény, ha egyszer összegyűjtötte az összes adatbevitelt, ténylegesen fel is kell dolgoznia, amit én felülírok:

```
public override void ProcessInput()
{
        // csinálni az alapértelmezett adatbevitel számítást
        base.ProcessInput();
```

Az első dolog, amit csinálok, hogy hívom az alap *ProcessInput* függvényt, melyek egyszerűen elmondják az egér, billentyűzet és botkormány adatbevitel kezelőknek, hogy gyűjtsenek adatokat. Ennél a pontnál a *kl*, *kr*, *ku*, *kd*, *jx* és *jy* változók tartalmazni fogják a nyíl gombok és a joystick tengelyek állapotát. Most itt az idő tulajdonképpen valamit csinálni ezekkel az értékekkel, például kiszámítani a játékos hajójának X és Y gyorsaságát:

```
        float vx = 0;
        float vy = 0;
        if( kl )
                vx -= player.Speed;
        if( kr )
                vx += player.Speed;
        if( ku )
                vy -= player.Speed;
        if( kd )
                vy += player.Speed;
        // most hozzáadjuk a joystick értékeket
        vx += (jx/10000.0f) * player.Speed;
        vy += (jy/10000.0f) * player.Speed;
```

Ha a Bal Nyíl gomb lenyomódik, akkor a játékos sebessége kivonódik a játékos X gyorsaságából. Ez azt jelenti, hogy ha csak a Bal Nyíl gomb van lenyomva, és a hajó 200 képpontot tud mozogni másodpercenként, akkor a *vx* -200 lesz. Hasonló dolog lesz a másik három nyíl gomb esetében is.

A következő lépés a joystick adatbevitel számítása. A függvény fogja a *jx* és *jy* változókat, és elosztja ezeket 10,000-rel (a botkormány hatótávolsága). Ha *jx* 5000-nél van, akkor 0,5-tel fogsz végezni, ami azt jelenti, hogy a rúd félúton van jobbra. Ezesetben szorzod azt az értéket a játékos sebességével, és ha a fenti

példát használod, kapni fogod a *vx* összetevő +100 képpont másodpercenkénti értékét. Hasonló van az Y tengelynél is.

Így most kiszámítottad a hajó sebességét az X és Y tengelyekben, de van egy probléma: ahogy most áll, a játékos tud „csalni”, és meg tudja csinálni, hogy a hajója kétszer olyan gyorsan menjen felfelé, mint amennyire lehet! Ha a játékos lenyomva tartja a Bal Nyíl gombot és a botkormány a -10,000 *x* pozíciónál van, ez számítani fog egy -400 képpont másodpercenkénti sebességet, ami kétszer olyan gyors, mint a 200-as sebesség. Hoppá! (Igen, tulajdonképpen bonyolult a nyíl gombokkal és a botkormánnyal játszani egyszerre, de valaki csinálhatja.)

Tehát az utolsó lépésben megbizonyosodunk, hogy az értékek soha nem haladják meg a hajó sebességét, és aztán beállítjuk a sebességet megfelelően:

```
        // megbizonyosodni, hogy a játékos nem kap sebességnövelést
        // a billentyűzet és a botkormány egyidejű használatával
        if( vx > player.Speed )
                vx = player.Speed;
        if( vx < -player.Speed )
                vx = -player.Speed;
        if( vy > player.Speed )
                vy = player.Speed;
        if( vy < -player.Speed )
                vy = -player.Speed;
        player.VX = vx;
        player.VY = vy;
}
```

Most a játékosaid nem tudnak csalni!

Renderelés

A renderelés meglehetősen egyszerű. Először hadd mutassam meg a *CustomRender* függvényt:

```
protected override void CustomRender()
{
        Game.devices.Graphics.Clear( D3D.ClearFlags.Target,
                System.Drawing.Color.Black, 1.0f, 0 );
        Game.devices.Graphics.BeginScene();
        Game.devices.Sprite.Begin();
        foreach( Sprite s in ships )
                s.Draw( Game.devices.Sprite, camera );
```

```
        foreach( Sprite s in projectiles )
                s.Draw( Game.devices.Sprite, camera );
        foreach( Sprite s in powerups )
                s.Draw( Game.devices.Sprite, camera );
```

Az azonos régi D3D inicializációs kód alkotja a függvény első három sorát, és aztán a *sprite*-ok rajzolása jön be. A játék egyszerűen ismétlődve átmegy a három objektum tömbön, és kirajzolja azokat. Emlékezz, hogy mivel a *GameObject*-ek *Sprite*-ok, egészen könnyen ki tudod őket rajzolni.

A következő lépés a felhasználói felület (UI) kirajzolása és némi vegyes szöveg:

```
        DrawUI();
        if( paused )
        {
                Game.bigfont.DrawText(
                “SZÜNET”,
                new System.Drawing.Rectangle(
                0, 0,
                GSS3KConstants.Width, GSS3KConstants.Height ),
                D3D.DrawTextFormat.Center |
                D3D.DrawTextFormat.VerticalCenter,
                System.Drawing.Color.White );
        }
        Game.smallfont.DrawText(
                “Nyomj F1-et a súgóhoz”,
                new System.Drawing.Rectangle(
                        0, 0,
                        GSS3KConstants.Width, GSS3KConstants.Height ),
                0, System.Drawing.Color.FromArgb( 127, 255, 255, 255 ) );
        Game.devices.Sprite.End();
        Game.devices.Graphics.EndScene();
        Game.devices.Graphics.Present();
}
```

Ha a játék futása szünetel, a SZÜNET szó íródik ki a képernyőre, és egy „Nyomj F1-et a súgóhoz” szöveg íródik ki a képernyő bal felső sarkában. A szöveget félig áttetszővé tettem.

A felhasználói felület a DrawUI függvényen belül rajzolódik ki, de nem szándékozom azt megmutatni neked. Valószínűleg már rosszul vagy ettől a sok grafikus kódtól. Ha érdekel, nézd meg a könyv CD mellékletén.

Játék a GSS3K-val

A játékkal játszani meglehetősen egyszerű, ahogy már láttad a vezérlő kódban. A nyíl gombok mozgatják a hajódat, a szóközzel tüzelsz, a Control gombok a fegyvereidet váltják, a P szünetelteti a játékot, az F1 a súgóképernyőre visz, és az Escape-pel kiléphetsz. Továbbá választhatod a botkormány használatát (bizonyosodj meg arról, hogy kiválasztottad a beállító képernyőn), mely esetben a 0. gombbal lőhetsz és az 1-es és 2-es gombok használatosak a fegyverek váltására.

A játék nagyon egyszerű, ahogy azt mondjuk 10 másodpercnyi játék után látni fogod. De ez a kezdet! Azonkívül ez az egyszerű játék 10-szer összetettebb, mint a legtöbb eredeti űrhajós lövöldözős.

Tehetsz a játékra néhány csípős megjegyzést – dolgokra, amiket nem volt időm megcsinálni. A játék csak folymatosan megy, miután meghaltál. Ez valószínűleg egy rossz ötlet, de a kód eléggé struktúrált, és folyamatosan fut gond nélkül. A jövőben valószínűleg fontolgathatod egy „VESZTETTÉL!” feliratú képernyő hozzáadását a játékhoz.

Ahogy most állunk, a játék végtelen; nincs végső főgonosz, nincsenek szintek. Hajók egy végtelen tömbje van, melyek körülötted száguldoznak. Ez egy másik helyzet, amit meg akarhatsz változtatni a játék egy valódi verziójában.

A másik fő probléma a részletek hiánya – ez egy űrjáték, mégsincsenek benne űrobjektumok, mint pl. csillagok, aszteroidák, stb. Ilyen részleteket a játékhoz adva az még elragadóbb lehet. Ahogy most áll, a játék elég unalmasan néz ki – csak egy fekete képernyő, rajta űrhajókkal.

Akárhogy is, de a Demo 10.2 egy megfelelő bemutatója valaminek, amely tovább folytatódhat, hogy egy nagyszerű játéktermi-stílusú játékká váljon. Érezd szabadon módosíthatóvá a saját céljaid szerint.

A jövő

Könyvet írni általános játékprogramozásról nagyon bonyolult dolog. A gond az, hogy szó szerint több ezer játékműfaj létezik, és egy könyv nem képes lefedni ezt a jelentős mennyiségű információt.

Ez a könyv sok témát lefed, amely látható majdnem minden játékban, tehát ez egy jó kiindulási alapot adhat neked. A legtöbb játék használ grafikát, hangokat, adatbevitelt, sprite-okat, ütközéskezelést, és így tovább. És még tonnányi más elgondolás van, amit a játékok használnak, de nincs elég helyem, hogy mindet bemutassam.

3D világok

Napjainkban a legtöbb játék 3D-s, és jó okkal! A 3D-s játékok sokkal valósághűbben néznek ki, mint a 2D-sek, és sokkal több rugalmasságuk van. Nem csak ez, de a 3D-s eszközök már fillérekbe kerülnek. Illene találnod kevesebb, mint 20$-ért olyan videókártyát, aminek van színező hardvere, és olyan videókártyát, amely támogatja a hardveres csúcspont transzfromációkat és megvilágítást, kevesebb, mint 50$-ért.

Haladó ütközésérzékelés

Magától értetődően, az ütközésérzékelés a *GSS3K*-ban egyszerű; alapvetően elhelyez egy elméleti négyszöget az objektumaid körül, és ellenőrzi, hogy azok átfednek-e. De a való világban nem minden négyszög-alakú. A probléma 3D-ben még rosszabb, mert nem minden kocka-alakú; tehát tonnányi különböző módja van annak, hogy ellenőrizd, ütközik-e egy objektum a játékban.

Mesterséges intelligencia

A *GSS3K*-ban semmilyen fajtája nincs a mesterséges intelligenciának (AI). Az ellenséges hajók egyszerűen csak mennek előre és lőnek rád. Elég hülyeség, ha engem kérdezel. A játékkészítő társaságok sok kutatást fektetnek az AI-ba, mert ez minél haladóbb, a játék annál valósághűbb. Semmi sem rombolja jobban a valóságosság elemeit, mint ha van egy számítógép-irányította objektum, ami valami igazán nagy hülyeséget csinál, például beszorul egy falba vagy hasonló. Egész könyvek íródtak az AI témájáról. Ez összetett dolog.

Hálózat

A hálózat használata egy másik hatalmas témája a játékprogramozásnak. Mikor játszottál utoljára egy olyan nagy játékkal, amely nem támogatta a többjátékos módú online játékot. Valószínűleg nagyon régen, ugye?

Haladó tárolás

A tárolás a *GSS3K* számára csodálatosan egyszerű: objektumok három tömbje. Ez olyan egyszerű, mivel könnyebb nem is lehet. Nagyobb játékokban ki kell találnod, hogy hogyan tárolod az adatok minden fajtáját – néha az egész világ adatainak értékét! Hogyan fogod ezt csinálni? Nos, ez az, ahol az adatszerkezetek elméletben a számításba jöhetnek. Több adatszerkezet van, mint csak a tömbök, és neked kell kifejlesztened, hogy milyen fajta szerkezet illik legjobban a virtuális világod tárolásához.

Talán megnézheted, hogy van-e adatbázisod az adatok tárolásához, de azt rendszerint csak a hatalmas, masszívan többjátékos módú online játékok végzik.

Összegzés

Nos, most, hogy össze van állítva neked egy hatalmas játék, mit fogsz tenni? Elmész a Disney World-be? El kell gondolkodnod azon, hogy beépítsd az első játékodba, vagy létrehozz valami jobbat. Mostanra eleget kell tudnod ahhoz, hogy elindulj a saját programjaiddal.

Záró szavak

Most már tudsz mindent, ami tudni való van a játékprogramozásról C#-ban! Oké, nem, én csak viccelek. Igazán utálok az lenni, aki összetör téged, de csak alig karcoltad a felszínt. De egy nagyszerű elindulásnál vagy! A játék programozás az egyik legösszetettebb tanulmányi terület az egész világon, és büszke lehetsz arra, hogy tagja vagy ennek az elit közösségnek.

Te mindig tanulni fogsz. Senki nem tudhat mindent, amit valaha tudni kell a játékprogramozásról. Ez egyszerűen lehetetlen. De ez egy jó dolog. Úgy találom, hogy a legtöbb ember bekerül a játékprogramozásba első helyben,

mert ők a tanulás egy végtelen keresésében vannak. Ha ez jellemez téged, akkor fiú, megtaláltad a megfelelő helyet!

Az 5. és 10. fejezetek adtak neked néhány ötletet arról, hogy hol bővítsd az ismereteidet a C# területén és a játékprogramozásban általában. Szerencsére neked, rengeteg könyv létezik, amelyek segíteni tudnak neked ebben a keresésben. Különösen, valószínűleg akarod folytatni az olvasást a DirectX-ről és a Direct3D-ről. Wendy Jones: Kezdő DirectX 9 (Beginning DirectX 9) és Wolfgang Engel: Kezdő Direct3D játékprogramozás (Beginning Direct3D Game Programming, 2nd edition) című könyvei különösen hasznosak lesznek számodra.

És most, tanulók, attól félek, itt az idő, hogy menjek és itt az ideje, hogy kitárd a szárnyaidat és felfedezd a játékprogramozás hatalmas világát.

Köszönöm, hogy elolvastad ezt a könyvet – öröm volt téged játékprogramozásra tanítani C#-ban.

FÜGGELÉK

A DirectX és a .NET beállítása

A .NET platform nagyon menő több okból is, de az én kedvenc indokom hogy az SDK és a fordító egésze teljesen ingyenes. Neked nem kell fizetned senkinek semennyit azért, hogy használd a .NET-et és a DirectX-et C# játékok programozásához.

A .NET keretrendszer

A legelső dolog amit tenned kell, hogy telepítsd a .NET keretrendszert, mely tartalmaz mindent, ami kell neked, hogy futtass .NET programokat. Ha Windows XP-t futtatsz, akkor már van egy .NET keretrendszered telepítve, de valószínűleg egy régebbi változat, mint pl. az 1.0. E könyv írásakor a jelenlegi változat az 1.1; ezt a verziót kellene telepítened. Ha a Microsoft a jövőben kiadja egy új változatát, akkor megtalálod azt a http://microsoft.com –on, ingyen letölthetőként, ahogy a http://windowsupdate.microsoft.com – on is. Elhelyeztem az 1.1 keretrendszer telepítőfájlját a CD mellékleten, az \extras\ DOTNet1.1Framework\ mappában.

A .NET SDK

A keretrendszer csak .NET programok futtatását teszi lehetővé, nem létrehozni őket. Azért, hogy létrehozhass .NET programokat, szükséged van a .NET 1.1 SDK-ra. Ez ingyen letölthető a Microsofttól, de megtalálhatod a CD mellékleten is a \extras\DOTNet1.1SDK\ mappában. Az SDK telepít mindent, ami szükséges neked, hogy megalkosd a saját C# programjaidat, beleértve – és legfontosabbként – a *csc* parancssori C# fordítót, amit láttál a 2. fejezetben.

Beépített fejlesztő környezetek (IDE)

Parancssoron keresztül fordítani nehéz. Senki sem csinálja már, mert senki sem helyezi a kódját egyetlen állományba többé, és többfájlos projektek fordítása parancssorral fájdalom a fenéknek.

Ez itt a jövő! IDE-ket használunk most. Egy IDE egy grafikus program, ami nyilvántartja a projektjeidet neked, és önműködően fordítja azokat. Ha valaha használtad már a Visual Studio-t, akkor tudod, hogy mi egy IDE.

Sajnos az új játékprogramozóknak az IDE-k drágák. Ezek 200-500$-os értékben vannak, és néhányan sok lóvét költenek rá. Szerencsére van egy jobb megoldás: ne fizess semennyit!

Van egy kiváló C# IDE, amit SharpDevelop-nak hívnak. Ebben a könyvben az összes példa ezzel fordítódott és futott, ahogy szintén Visual C#-pal (a Visual Studio egy összetevője). Hogy megkapd a SharpDevelop-ot, csak menj a http://icsharpcode.net honlapra, és töltsd le az IDE legújabb változatát.

Kezelt DirectX

A Microsoft a DirectX SDK-t is ingyen nyújtja. Milyen szép tőlük! Letöltheted a legújabb változatát a http://microsoft.com -ról, vagy telepítheted a DX9.0b SDK-t a CD-ről, az \extras\DirectX9.0bSDK mappából.

E könyv írásakor a DirectX 9 SDK legújabb változata a 9.0c, mely túl későn érkezett nekem, hogy végigpróbáljak mindent azon. A fő változások az SDK azzal a változatával mind a D3DX könyvtáron belül vannak, mely a textúra betöltést és a sprite-okat érinti a 7. fejezetből, de semmi mást.

Hivatkozások beállítása

A C# projektek hivatkozásoknak (referenciák) nevezett elgondolásra támaszkodnak. Egy hivatkozás az olyan, mint egy könyvtárfájl a C++-ban; meghatároz egy könyvtárat, amit a programod használni fog. A .NET-nek ezek a hivatkozások rendszerint DLL állományokban tárolódnak.

Az összes DLL-t megtalálod a .NET keretrendszernek és DirectX-nek a merevlemezeden, jellemzően a C:\Windows\Microsoft.Net könyvtáron belül.

Hivatkozásokat kell adnod a projektedhez, amikor használni akarsz különféle névtereket. Például, ha a *System* névteret akarod használni, akkor hozzá kell adnod egy hivatkozást a *System.dll*-hez a projektodba, és ha a *System.Windows.Forms*-ot akarod használni, akkor a *System.Windows.Forms.dll*-t kell a projektedhez adnod.

Visual C#-ban jobb kattintással adhatsz hivatkozásokat a *References*-en, a *Solution Explorer* ablakban, aztán az *Add Reference*-t választva. Ekkor az *Add Reference* párbeszédablak bukkan fel.

Tudsz duplán kattintani a hivatkozások bármelyikén a párbeszédablakban, vagy kattints a Browse gombra, hogy a merevlemezedről kiválaszthass bármilyen hivatkozást, ami nincs listázva.

A folyamat hasonló a SharpDevelop-nál: menj a *Project* fülre, jobb kattintás a *References*-eken, és válaszd az *Add Reference*-t. Egy párbeszédablak fog felbukkanni.

Kiválaszthatod a telepített hivatkozások bármelyikét az ablakban, vagy kattinthatsz a *.NET Assembly Browser* fülön, hogy kikeress bármilyen hivatkozást a merevlemezedről, amit hozzá akarsz adni.