CodeX
Nemzetközi
Számítástechnikai-Programozói Szaklap

FŐSZERKESZTŐ:
Hevesi Zsolt
h.zsolt@codexonline.hu

SZERKESZTŐ
Dr. Medzihradszky Dénes

FELELŐS KIADÓ
CODEX INTERSOFT Bt.

GRAFIKUS
Kostean Norbert

TÖRDELŐ:
Kostean Norbert

FŐ TÁMOGATÓINK:
Oktatási Minisztérium
Márton Áron Szakkollégium

FŐ MÉDIATÁMOGATÓNK:
IDG Magyarország

SZAKMAI PARTNEREINK:
BioDigit Kft.
Next IT Hungary Kft.
Számalk Rt.

A **CODEX** INTERNETES ELÉRHETŐSÉGE:
www.codexonline.hu

NYOMDA:
Révai Nyomda Kft.
1037 Bp., Kunigunda útja 68

PÉLDÁNYSZÁM:
20.000

TERJESZTÉS:
Magyarországon és 7 határmenti országban

Kedves Kolléga!

Ha érdekli Önt az informatikán belül a **programozás**, a számítástechnikai rendszerek működése, akkor ez a folyóirat Önnek szól!

Szeretnénk megismertetni egy új lehetőséget, melynek segítségével továbbfejlesztheti tudását, de ugyanakkor át is adhatja ismereteit másoknak. Egy kiterjedt oktatási projektről van szó, mely **"hetedhét országra szól".**

Elvünk, hogy a tudás terjesztésében nincs korhatárra vagy tudásszintre vonatkozó megkötés! A projekt keretében a hangsúlyt egy nemzetközi szemléletű, mind Internetes, mind nyomtatott formában megjelenő, az Olvasó **által is írható** folyóiratra helyezzük.

A **CodeX folyóirat** egy éven át folyamatosan megjelent az Interneten. Az elmúlt időszakban 10 szám került ki a világhálóra, melyekben összesen több mint 220 informatikai tárgyú cikk látott napvilágot. A népszerű témáktól kezdve - sokszor középiskolások, és felsőoktatási intézmények hallgatóinak tollából - egészen az új technológiákat bemutató szakcikkekig sok minden megtalálható.

Cikkeink **főként programozás**sal kapcsolatos témákat tárgyalnak, az **egyszerű szakmai fogásoktól a grafikai programok készítéséig, karakteres képernyőtől a WAP oldalakig**, különböző nyelvek és eszközök használatával. Megtalálhatók közöttük hardveres problémák leírásai és adatbázis kezelési ismeretek is.

Profiljának megfelelően a lap lehetőséget ad különböző témák publikálására; legyen az akár programozási feladat, vagy oktató, ismeretterjesztő jellegű cikk leközlése. Ez nem csak felajánlás, hanem kérés is, hiszen **mindenki ezreket taníthat a saját munkájával**. Aki pedig egy-egy programozási nyelvnek szakembere, sokat segíthet tudásával mint szaklektor a publikálni vágyó, a szakmában fiatal tehetségeknek.

E mostani nyomtatott folyóirat a sorozat 11. száma, hiszen az elektronikus lap folytatása. Így az egyes cikksorozatok itt kerülnek folytatásra. De természetesen új cikkekkel is jelentkezünk szokásunkhoz híven.

Hevesi Zsolt

KÖSZÖNETNYILVÁNÍTÁS
Köszönetet szeretnék mondani mind a 20 000 leendő Olvasónk nevében az Oktatási Minisztériumnak és a Márton Áron Szakkollégiumnak! Köszönetet a lehetőségért, mellyel a CodeX projektet kirobbanásszerűen nemzetközi viszonylatban is megismerhetik. A nyomtatott folyóirat alapul szolgál az informatikai szakma tanításában, a tehetségesek felkutatásában és nem utolsósorban a kapcsolatépítésben. És hálás vagyok, hogy nem hagyták veszni kemény hároméves munkám, és figyeltek az eddigi több, mint 12 000 egyedi online olvasónk érdeklődésére.

CODEX
Nemzetközi
Számítástechnikai-Programozói Szaklap

FŐSZERKESZTŐ:
Hevesi Zsolt
h.zsolt@codexonline.hu

SZERKESZTŐ
Dr. Medzihradszky Dénes

FELELŐS KIADÓ
CODEX INTERSOFT Bt.

GRAFIKUS
Mura István Zoltán

TÖRDELŐ:
Mura István Zoltán

FŐ TÁMOGATÓINK:
Oktatási Minisztérium
Márton Áron Szakkollégium

SZAKMAI PARTNEREINK:
BioDigit Kft.
Next IT Hungary Kft.
Számalk Rt.

A **CODEX** INTERNETES ELÉRHETŐSÉGE:
www.codexonline.hu

NYOMDA:

PÉLDÁNYSZÁM:

TERJESZTÉS:
Magyarországon és 7 határmenti országban

Kedves Olvasó!

Az első nyomtatott lapunk megjelenése sokakban kérdéssorokat vetett fel, hogy ”Miért egy újabb nyomtatott számítástechnikai újság? - főleg hogy az Internet óriási léptekben terjed. És miért egy programozási szaklap? amikor egy oly rohamosan fejlõdõ szakágról van szó. És mégis mivel nyújthat többet egy nyomtatott lap, mint az elektronikus anyag.”

A szerkesztõséggel, nem egy ”újabb” számítástechnikai lapot szeretnénk vinni, és nem is a szó szerinti egy „merev” programozási szaklapot, ahol a forráskód ismertetés legyen a célunk.

Sokan úgy gondolják, hogy minden megtalálható az Interneten, ami egy nyomtatott újságban is szerepel, és mindent megtalálnak, amire csak szükségük van. Ez sok esetben elõfordulhat. Azonban mi ezen meggyõzõdésen változtatni szeretnénk. Célunk, hogy újszerû gondolatokat közöljük; hogy szinte nyelvfüggetlenül is frappáns megoldásokat, és elgondolkodtató, tanulságos gondolatokat közöljünk. Nem a kész forrásprogramok elkészítésén, és bemutatásán dolgozunk, hanem kulcsgondolatokat gyártunk. Olyan cikkekre törekszünk, melyek felébresztik az emberben a cselekvés vágyát. Azt, hogy élvezzék a továbbgondolás minden egyes percét, Azt, hogy akár több év múlva is használni tudja a megszerzett ismereteket. Nem egy ”szélkergette” információzáport szeretnénk az Olvasókra zúdítani, hanem mély, szakmai gondolatokat, mely a maga nemében megállja a helyét, akár három éve született meg, akár két év múlva szeretnék felhasználni.

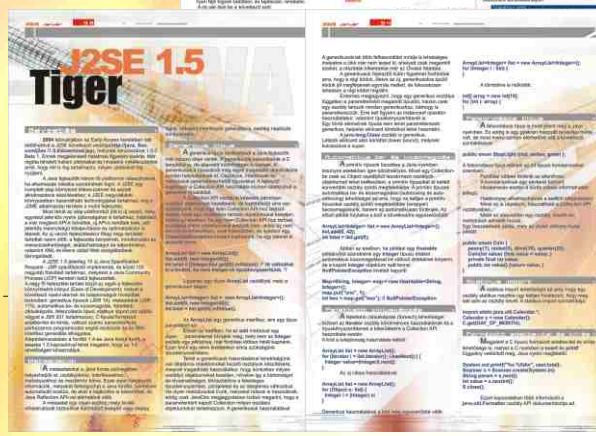
Törekszünk továbbá, hogy a cikkek minél olvasmányosabbak, és érthetõek legyenek, hogy a munkahely után fotelben kellemes idõtöltést nyújtson, vagy a lefekvés elõtt tanulságos, vagy épp szórakoztató olvasnivalót nyújtson.

Az oldalszámot az elõzõ nyomtatott számhoz képest a duplájára növeltük. Reméljük, hogy a cikkek ezúttal is elnyerik olvasóink tetszését. Folytatódnak a már jól ismert cikksorozataink az ”LPT portok vezérlése”, az ”Egy Rendszergazda hétköznapjai” és az ”OpenGL.hu” is. És sok új cikksorozatot is indítunk, melyek közül ki szeretném emelni az ”Eseménykezelés Flash MX-ben” sorozatunkat.

Kellemes olvasást kívánva, tisztelettel Hevesi Zsolt.

Alapító–főszerkesztő
Hevesi Zsolt

Tartalom



ITT AZ ÖN SZOFTVERE BESZÉL...

Beszélő programok fejlesztése egyszerűen

Ki ne hallotta volna a mobiltelefonjában azt a „kedves, szőke, kék szemű női hangot”, aki közli velünk számlaegyenlegünket, esetleg útbaigazít a szolgáltató menürendszerében.Beszédét általában kellemesnek ítéljük meg, néha azonban kicsit gépiessé válik, esetleg kattan, vagy „ugrik” egyet… Miért van ez? Aki végigolvassa a sorozatunkat, meg fogja érteni. Sőt, arra is kap útmutatást, hogy a fenti hibábat miként kerülheti el. Amikor a gondolkodó ház elterjedése napi kérdéssé érett, a háztartási berendezések egyre intelligensebbek lesznek, a háztartási robotok pedig egyre többször tünnek fel -elsősorban a Japán újságcikkekben, de lassan a boltok polcain is-, akkor egyre égetőbbé válik, hogy megtanuljunk néhány apró, szakmai fogást, melyekkel mi is tudunk beszélő programokat készíteni. Ha ehhez hozzátesszük az LPT-s cikksorozatban megismerteket, illetve a webprogramozással foglalkozó cikkeket, akkor határtalan lehetőségek nyílnak meg előttünk; Számítógépünkkel a kommunikációnk sokkal emberibbé, közvetlenebbé válik, ugyanakkor a gépet beköltöztethetjük lakásunk, autónk vezérlőegységébe, stb. S így egy meglepően kellemes, intelligens személyiség benyomását keltő segítőtársunk lehet az a PC, melyet korábban sokan az íróasztalon tudtak csak elképzelni. Ebben a cikksorozatban röviden összefoglalom a beszédelőállítási módszereket, majd ennek egy fajtáját részletesebben ismertetem.

Kis történelem:

Talán meglepőnek hangzik, azonban a beszédelőállítás korántsem mai találmány, sok forrásban lehet olvasni éneklő, beszélő automaták leírásairól már a középkorban is. Vélhetőleg korábban szintén kísérletezhettek ezzel. Ellenben először, -építési leírással dokumentáltan- Kempelen Farkas készített beszélő gépet, mely mechanikus úton képes volt az emberi hangokat előállítani. Természetesen nem volt egyszerű dolog „beszéltetni” ezt a berendezést! Szinte zongoraművész tehetség kellett ahhoz, hogy valami érthető, emberi hangot kicsiholjon belőle. Azonban a gép ennek ellenére mégiscsak beszélt! Bizonyítva ezzel, nem kizárólag az élőlények privilégiuma a beszéd előállítása, hanem technikai úton is lehetséges ez. Fontos megmlíteni, hogy nem hangrögzítőt alkotott, hanem -mai szóval- „hangszintetizátort”, ami az egyszerű rögzítésnél sokkal előremutatóbb dolog!

A következő lépést Edison találmánya, a fonográf, majd ennek tökéletesített változata, a gramofon jelentette. Edison évekig táviratozással kereste a kenyerét, s megfigyelte, hogy a távíró tűje leérkezéskor a papíron sercegő hangot ad, mert a folyamatosan elhaladó papírszalag barázdáit követve rezgésbe jön. Arra gondolt, mi lenne, ha zenével, emberi hanggal próbálná meg a szalagra felvinni ezeket a barázdákat. Ugyanis ezt visszahallgatva, talán a távírótű megszólalna… Sztaniollal bevont fahenger lett a hordozó, ami egy kézzel tekert menetes orsón forgott, s a tű egy fémlemezkét rezgetett, illetve annak a rezgéseit karcolta a lemezbe. Jellemző, hogy az első szöveg, amit visszajátszottak a bemutatón a szerkezettel egy gyerekdal pár sora: „Mérinek kicsiny barikája volt…” Természetesen a hangrögzítés forradalma újabb távlatokat nyitott a beszélő gépek számára is. A telefon elterjedésével híreket, fontos eseményeket, gyerekeknek szóló esti meséket mondott a telefonközpont. Azonban számunkra igazán érdekes a pontos időt bemondó automata volt!

Eleinte fonográfhengerekre, majd gramofonlemezekre, később pedig egy optikai tárcsára, -fénnyel-, (mint a filmeknél a „fényhang”) vették fel a szavakat. Egy mechanikus óra tengelye forgatta a tárcsákat, az időnek megfelelelően. Amikor be kellett mondani az időt, akkor a tárcsák állásának megfelelően egyik tárcsát a másik után bejátszották. Ez az első olyan gépi beszéd volt, amikor külső eseménytől (pontosidőtől) függően kellett változó szöveget bemondani.

A 30-as években ezek a berendezések gombamódon elszaporodtak, divattá vált a „házirobot”, illetve a kiállításokon feltűntek a mágneses hangrögzítőkkel felszerelt beszélő robotok, idegenvezetők. Ezekben egy, vagy több magnetofon állította elő a beszédet. Bár hatalmas szerkezetek voltak, de itt már külső eseménytől (hőmérséklet, idő, szenzorok jelei, stb…) függő, emberi beszéd szólalt meg. A technika fejlődésével ezek a gépek is egyre fejlettebbé váltak. Azonban a fenti alaptípusok, vagyis a szintetizált szöveg, illetve a tárolt beszéd variálásával működő módszer lényegében a két, ma is alapvetően elterjedt beszédgenerálási eljárás. Nézzük meg tehát ezek alapvető sajátosságait:

Szintézisen alapuló módszer

Az emberi beszéd egy nagyon összetett függvénnyel leírható jelsorozat. Mivel egyénenként változó az egyes hangok ejtése, -magassága, a felharmonikusok összetétele, a beszédtempó, a remegés (vagyis „intonáció”), stb,- szintetikus úton valakinek a hangját pontosan leutánozni szinte lehetetlen. Az is hatalmas erőfeszítést igénylő feladat, ha egy semleges, ellenben már nem gépies hangzású, monoton beszédet szeretnénk így kapni. A szintetizált beszédnél egy bemondóval felolvastatnak egy előre meghatározott, kellően változatos szöveget. Ezután a szövegen Fourier transzformációt hajtanak végre. Ekkor megkapják a frekvencia-idő diagramját. (Aki nem tudja miről van szó, gondoljon a népszerű WINAMP program „spectrum bar” kijelzőjére, csak sokkal több oszloppal.) Ezután elkezdik elemezni, hogy a kimondott szöveg ismeretében az egyes hangok milyen frekvencia és amplitúdóösszetevőkkel rendelkeznek. Összegyűjtik az egyformákat, különválasztják az eltérőket. A hangok azonban nem egyszerűen hangonként bírnak jellemzőkkel. Másképp szól, ha másik hanghoz kapcsolva ejtünk ki egy mássalhangzót. (pl. „ED és ÖD kapcsolat) Vannak felfutó, lefutó ívek, ívdarabok, réshangokhoz, zörejhangokhoz kapcsolódó tagok, stb. Ezeket mind, mind el kell különíteni. Végül létrejön egy ún. „hangszelettár”, melyből építkezve már tetszőleges szövegek építhetők fel. A hangszelettárnak jobb esetben is többszáz, de gyakran több ezer önálló eleme lesz. Ezeket az elemeket részletesen megvizsgálják. Feljegyzik az erősség változásaikat, vagyis az „amplitúdó- burkológörbét”, illetve az egyes frekvenciaösszetevők arányait. A kapott adatokat táblázatos formában eltárolják. Így jön létre a „hangszelettár-paramétertáblázat”. A további eljárás már a technológiától függ. Régebben célhardvert építettek programozható frekvenciájú oszcillátorból, (zönge generátor) zajgenerátorból, (zörejgenerátor), illetve programozható frekvenciájú szűrőkből. /Általában a 4. harmonikusig szintetizálják a hangot, (ekkora pontosság a gyakorlatban elegendőnek bizonyult) ehhez tehát 4 szűrő kell. / A zajt, illetve az alapfrekvenciát, s a szűrőket egy keverőre vezették, melyet szintén programozni lehetett. Ennek kimenetén áll elő a kívánt beszédhang. A megoldás hátránya, hogy célhardvert igényel, előnye, hogy a számítógép teljesítménye viszonylag alacsony lehet. Gyakran a beszédszintetizátort egyetlen integrált áramkörben valósították meg. Az IC néha tartalmazta a hangszelettárszintéziséhez szükséges adattáblázatot, néha pedig helyette a gép adta a paramétereket is. Előző esetben egyszerűbb volt a szoftver, utóbbiban tetszőlegesen változtatható a hang minden paramétere. Az idő múlásával azonban megjelentek előbb a nagyobb teljesítményű asztali számítógépek, majd a gyorsabb mikrovezérlők, illetve a DSP-k (digitális jelfeldolgozó processzorok). Ezeknél már szoftverből a teljes szintetizátor hardvere helyettesíthető, méghozzá valós időben! Vagyis a szintetizált beszéd előállítása teljesen folyamatosan, szoftverből történik. A számítógép a digitális jeleket végül egy D/A átalakítóra küldi, s ezzel előállt a hang. Az ilyen szoftver már jóval bonyolultabb természetesen, hiszen jelalak szinten mindent a programnak kell kiszámolni. Azonban a programot csak egyszer kell megírni, így végül a termék nagyon olcsó lehet. Fontos előnye a rendszernek, hogy tetszőleges szavak kimondására alkalmas, vagyis ma még nem létező, új szavakat is képes lesz a szoftver a jövőben kimondani, ha szöveges formában (pl. TXT) elé tesszük. Vannak direkt olyan rendszerek, melyek úgynevezetett „text to speech”, (vagyis szöveg-beszéd) átalakítók. Szokás ezeket a rendszereket emiatt „kötetlen szótáras beszélő rendszernek” is nevezni. Egyébként a programok a nyelvtani szabályok, hasonulások, kiejtési kivételek, stb. ismeretében meglepően kevés kiejtési hibával tudnak beszélni, folyamatosan. Vannak vakok, illetve gyengénlátók számára felolvasó szoftverek, melyek jó ideje kifogástalanul üzemelnek. Számtalan előnye mellett meg kell azonban említeni a módszer hátrányát is: sajnos a kapott beszéd ma még általában monoton, gépies hangzású. Habár egyre jobb és jobb szoftverek látnak napvilágot… Bátran mondhatjuk tehát, hogy Kempelen Farkas egykori találmánya a mai napig is kihat ránk, mitöbb megjósolható, hogy a távoli jövőben szinte kizárólag ez fog megmaradni.

Mintavételezésen alapuló módszer (sampler)

Ennél a módszernél a feladatot alapvetően más oldalról közelítjük meg. Egy beszélővel a létező összes mondatot, vagy szót felmondatjuk, amit szükségesnek tartunk s a megfelelő minőségben rögzítjük. Ezután az egyes szavakat, mondatokat, mondattöredékeket variálva, a megfelelő sorrendben történő szakaszok lejátszásával állítjuk elő a beszédet. Fontos észerevenni, hogy ennél a módszernél csak azt tudjuk kimondatni a géppel, amit a beszélő is kimondott egyszer. Vagyis ezek szókészlete kötött. Ezért ezeket a rendszereket szokás „kötött szótáras beszélő rendszernek” is nevezni. A kapott hangminőség általában kitűnő, hiszen a beszélő saját hangján szólal meg a rendszer. Olyan helyeken, ahol viszonylag keveset kell beszélni, azonban igen fontos az érthetőség, szinte kizárólag ezeket használják. Sok program alternatív szókészletet tartalmaz, s kiválaszthatjuk a számunkra legkellemesebben beszélő hangját. A mobiltelefonok, az ébresztőórák, a felvonók, ipari berendezések beszélő rendszerei ma ilyen módon szólnak hozzánk. Amellett, hogy a jövőt vélhetően a szintetizált beszédelőállítás fogja jelenteni, a jelen egyértelműen a mintavételezésé.
Mi is az ilyen programok készítését fogjuk áttekinteni, mert megdöbbentően kevés munkával lehet a mai eszközökkel igen látványos eredményeket elérni.

Hibrid beszélő rendszerek

Ezeknél a fenti két módszer kombinációját alkalmazzák. Igen sokfajta megoldás elképzelhető, így csak címszavakban említünk meg néhányat. A tárolt beszédből kivett darabok adják a hangszelettárat, melyből építkezünk. Ekkor a beszélő hangján fog megszólalni a rendszer. Azon változtatni nem tudunk, csakis másik hangminta alkalmazásával. Ellenben ugyanolyan kötetlen szótáras rendszert tudunk felépíteni belőle. Természetesen a beszélő eredeti hangját nem fogja megközelíteni a kapott hangzás, annál jóval „robotosabb” lesz.

A másik érdekes kombináció, amikor szintetikus úton állítunk elő beszédet, de csak pár szót. Ilyenkor lehetőség van az igen finom kidolgozásra, ami szinte csilingelően szép hangzást adhat. Mintha egy földöntúli lény beszélne kristálytiszta hangon az emberhez. Ilyenkor -bár szintézist végzünk-, mégis kötöttszótáras rendszerünk van. A módszert csak nagyon ritkán alkalmazzák, (pl. filmekben) mert meglehetősen gazdaságtalan eljárás. Gyakorlatban olcsóbb egy jóhangú narrátort megfizetni és a stúdióban elektronikusan „kiszinezni” hangját.
Ezzel a bevezetőnk végére értünk. A sorozat következő részeiben megismerkedünk egy módszerrel, melynek segítségével házilagosan is elfogadható minőségű hangokat leszünk képesek előállítani. Annyit már most is elárulhatunk, hogy számtalan szakmai fogást kell bevetni a kívánt eredmény eléréséhez. Megtanuljuk, hogyan lehet megtervezni egy beszélő szoftver szókincsét, összeállítani a szavakat, helyesen beolvasni, normalizálni, majd megalkotni a hang-adatbankot.
Végül pedig egy gyakorlati példát mutatunk be Visual Basic nyelven írt beszélő órára. A program messze nem tökéletes, mert viszonylag gyorsan kellett összedobnom VB vizsgára. Ellenben nagyon jól lehet belőle tanulni, mert szépen látszanak, hallatszanak a helyes, illetve helytelen megoldások hatásai is.

Kis Norbert norbimagan@freemail.hu

KOMPONENS fejlesztése Delphi -ben

A Delphi alatt végzett fejlesztések azért olyan hatékonyak, mert programjainkat komponensekből építhetjük fel, amelyek igen sok feladatot megoldanak külön programozás nélkül. A Delphiben azonban nem csak a vele adott komponenseket használhatjuk, hanem igen sok le is tölthető az Internetről, sőt mi magunk is készíthetünk ilyeneket, amelyeket utána ugyan úgy használhatunk, mint a beépítetteket.

Nagyon érdekes, hogy a Delphi alatt fejlesztők nagy része még soha nem írt komponenst, pedig ezzel igen megkönnyítené és meggyorsítaná saját munkáját. Ebben a sorozatban azt szeretném megmutatni, hogy hogyan lehet komponenst készíteni.

Én a komponensírást sokkal izgalmasabbnak, szórakoztatóbbnak tartom, mint magát a program írását. Sokan azért nem írtak eddig komponenst, mert igazából soha nem jártak utána annak, hogy hogyan is kell ezeket elkészíteni.

Talán kezdjük egy igen egyszerű, de ennek ellenére hasznos komponens megírásával! A Delphivel adott StatusBar komponensre nem tudunk más komponenseket ráhelyezni, pedig időnként igen kellemes lenne egy-egy kép, vagy nyomógombot elhelyezése a felületén. Írjunk egy olyan komponenst, ami ezt a hiányosságot kiküszöböli.

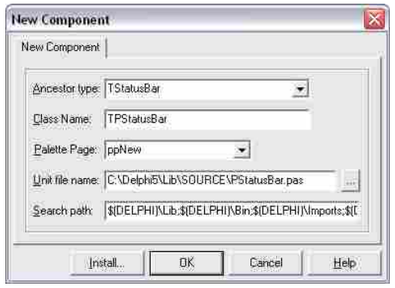
A feladat megoldása igen egyszerű, a StatusBar stílusának rendelkeznie kell a csAcceptsControls tulajdonsággal. Ez a tulajdonság teszi lehetővé, hogy más elemeket ráhelyezzünk az adott komponensre.

Akkor kezdjünk is hozzá az új komponensünk elkészítésének. Válasszuk ki a Componenent->New Component menüpontot. Itt hozhatjuk létre a legegyszerűbben az új komponenseinket. Az új TPStatusBar komponensünket a TStatusBar komponensből származtatjuk, így tulajdonságai meg fognak egyezni az ős tulajdonságaival, csak a stílusát fogjuk most kiegészíteni egy új elemmel!

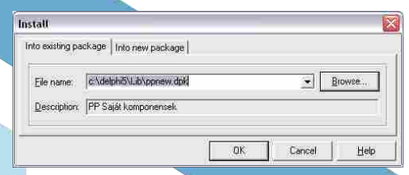
Az „Ancestor type:”-nál az ős komponens nevét kell megadnunk, vagyis azt, amiből származtatni kívánjuk az új komponensünket. Jelen esetben ez a TStatusBar. (A származtatatás osztályairól majd később bővebben is szó lesz!)

A „Class name:” mezőbe az új komponensünk nevét kell megadni, jelen esetben például írjuk be a TPStatusBar-t! A „Palette page:” mezőben annak a palettának a nevét adjuk meg, vagy válasszuk ki, amelyikre az új komponenst helyezni szeretnénk. Itt természetesen új nevet isy megadhatunk, vagy a már meglévőkből is választhatunk!

A „Unit file name” mezőben az új unit neve szerepel. Ezt a Delphi automatikusan generálja a komponens nevéből, de természetesen ez is átírható! A „Search path:”-ban a keresési útvonalat kell megadni, ebben szerepelni kell a komponensünk alkönyvtárának!



Amennyiben ezzel készen vagyunk, létre is hozhatjuk új komponensünket. Az „OK” gomb lenyomására elkészül az új unit, az „Install” megnyomására az új komponens hozzáadódik a megadott csomaghoz, és felkerül a kiválasztott palettára. Most nyomjuk meg az „Install” gombot, hogy a komponensünk a palettán megjelenjen.
Ekkor megnyílik egy ablak, ahol megadhatjuk azt a csomagot, amibe be szeretnénk rakni az új komponensünk.



Az első fülön egy már meglévő csomagot választhatunk ki, az „Into new package” fülnél pedig egy új csomagot hozhatunk létre. A „File name” mezőben meg kell adni a csomag nevét, (ez mindig DPK kiterjesztésű), a „Description” sorban pedig szövegesen megadhatjuk a csomag rövid leírását.

Az OK gomb megnyomásával létre is jön az új komponensünk forrása, és már bele is fordíthatjuk a kiválasztott csomagba! Amennyiben ezt tettük, az új komponensünk meg fog jelenni a kiválasztott palettán, és már használható is. Az új komponensünk ebben az állapotában teljesen megegyezik a TStatusBar komponenssel, de már létrejött, és használható!

Talán nagyon röviden nézzük meg a komponensünk forrását:

```
TPStatusBar = class(TStatusBar)
private
        {Private
declarations }
    protected
        {Protected
declarations }
    public
        {Public
declarations }
    published
        {Published
declarations }
    end;

procedure Register;

implementation

procedure Register;
begin
    RegisterComponents('ppNew'
, [TPStatusBar]);
End;
```

Az első sorban jön létre az új osztály a TStatusBar komponensből. Ebben az állapotban az osztályunk teljesen megegyezik az ős osztállyal. (Származtattuk!)
Van egy érdekes „Register” függvény is a forrásban, tulajdonképpen ez a sor gondoskodik arról, hogy a komponensünk megjelenjen a palettán. Ezt a „RegisterComponents„ függvény végzi, ahol először meg kell adni a paletta nevét, majd fel kell sorolni a palettára kerülő komponenseket. (Többet is megadhatunk itt!)

Azért azt már láthatjuk, hogy egy új komponens, legalábbis az alap osztály létrehozása nem is olyan nehéz dolog.

Akkor most folytassuk a munkát, és változtassuk meg a komponensünk stílusát úgy, hogy elemeket tudjon fogadni. A Create konstruktort kell ehhez átírnunk. Először is az objektum „public” részében írjuk felül az ős Create függvényt:

```
public
        constructor Create(AOwner:
TComponent); override;
```

Az „override” az ős osztály felülírása miatt kell! Nagyon fontos, származtatáskor soha nem szabad elhagyni! Ezzel biztosítjuk azt, hogy az ős osztály azonos nevű függvénye is lefusson!

```
constructor
TPStatusBar.Create(AOwner:
TCOmponent);
begin
        inherited;
        ControlStyle:=ControlStyle+[cs
AcceptsControls];
end;
```

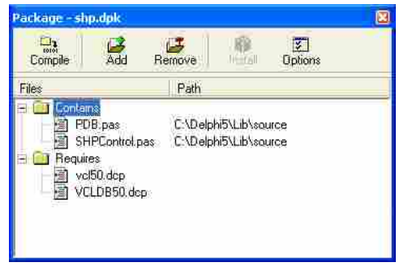
A Create függvényben az „inherited” sorral gondoskodunk arról, hogy az ős osztály konstruktora fusson le először, ezzel létrejöjjön az osztály, majd átállítjuk stílust.

Ezzel már készen is vagyunk az első saját készítésű komponensünkkel! Ha megnézzük a palettát, ahová az új komponensünk került, láthatjuk, hogy a képe megegyezik a StatusBar képével, (amiből származtattuk) de ez könnyen megváltoztatható az Image Editor programmal, és egyedi képet készíthetünk komponenseinknek.

Most már láttuk, hogy hogyan tudunk gyorsan új komponenst létrehozni, de nézzük meg azt is, hogy hogyan tudunk egy már kész komponenst, vagy komponens csomagot beilleszteni a Delphibe.

Komponensekhez két formában juthatunk. Egyrészt kaphatunk egy „Delphi Package”-et, vagyis egy komponens csomagot, ami tartalmazza azokat az állományokat, amire a csomaghoz szükség van.

A csomagot a kiterjesztéséről ismerhetjük meg, ez a „.DPK”.Amennyiben egy letöltött állományban „.DPK” kiterjesztésű fájlt is találunk, igen könnyű dolgunk van, ezt kell csak beilleszteni a Delphibe. Másoljuk be az állományokat egy alkönyvtárba, majd az Fájl -> Open menüpontot választva válasszuk ki a „.DPK” kiterjesztésű állományt, és nyissuk meg. Ekkor a megnyíló „Package” ablakban láthatjuk a csomagban található komponenseket, amikből természetesen több is lehet!



Az első dolgunk a csomag lefordítása a „Compile” gomb megnyomásával. A hibátlan fordítás után, az „Install”-al tudjuk beilleszteni a csomagban lévő komponenseket a Delphi alá. Amennyiben a fordítás alatt olyan hibaüzenetet kapunk, hogy nem talál a fordító valamilyen meglévő állományt, akkor a Tools->Environment ablak Library fülén adjuk hozzá a „Library Path”-hoz a komponensünk alkönyvtárát is! (Ez elég gyakori probléma szokott lenni!)

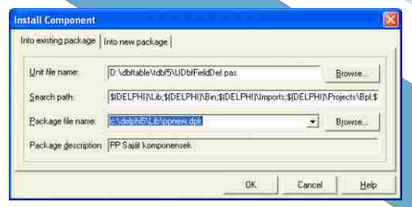
Amennyiben itt minden rendben volt, akkor az új komponensek már meg is jelennek a palettán, és használhatjuk is őket!

A „Package” ablakban lehetőség van komponensek hozzáadására, és törlésére is, a megfelelő gomb megnyomásával.

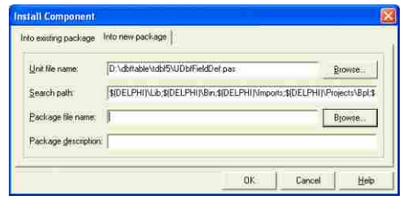
A másik esetben nincs „.DPK” állomány, ilyenkor nekünk kell létrehozni, vagy beletenni a komponenst egy csomagba. Azért ez sem egy bonyolult dolog, nézzünk erre is egy példát.

Készítsünk egy olyan csomagot, amiben ezeket a letöltött komponenseket fogjuk gyűjteni. A komponens állományait először másoljuk be egy alkönyvtárba.

Ezek után hozzunk létre egy új csomagot, amibe bele fogjuk tenni az új komponensünket. Ezt a „Component”->”Install component” menüpontban tehetjük meg:



A megnyíló ablakban egyrészt már meglévő csomagba („Into existing package”), vagy egy új csomagba tehetünk bele egy új komponenst („Into new package”). Először is adjuk meg a komponens forrását, amit a csomagba kívánunk tenni (Unit file name). A „Search path”-hoz adjuk hozzá a komponensünk alkönyvtárát, amennyiben az még nem szerepel benne. (Ezekben az alkönyvtárakban keresi az állományokat a fordító!) Már létező csomag esetén a lenyíló ablakból válasszuk ki a csomagot, amibe a komponenst kívánjuk helyezni. Új csomag létrehozása esetén válasszuk az „Into new package” fület:



Itt a „Package file name” mezőben adjuk meg az új csomag pontos nevét „.DPK” kiterjesztéssel, a „Package description” sorban pedig a csomag szöveges megnevezését, ami utal a csomag tartalmára. Ide például írhatjuk az „Internetről letöltött komponensek” szöveget! Az „OK” gombra kattintva a komponens bekerül a már meglévő, vagy az új csomagba. Ezután megnyílik a csomag, benne a komponensekkel. Itt már az előbb bemutatott eljárással le kell fordítani a csomagot, és installálni kell a Delphi alá. Ha minden rendben volt, akkor ne felejtsük elmenteni a kész csomagot a benne lévő komponensekkel!

Pammer Pál
pp.freeweb.hu
pp@freeweb.hu

C# Bevezetés

Üdvözlök minden kedves Olvasót! Ez most egy cikksorozat első része, és a sorozat középpontjában a C# (ejtsd: C-Sharp), ill. a .Net Framework fog állni. Ajánlom mindenkinek, aki már használt objektum-orientált programozási nyelvet, de nem túl régen programozik, vagy csak ki szeretne próbálni valami újat. A célom, hogy a rendszeres olvasóval egy-egy konkrét problémával foglalkozó programot megír(at)va bevezessem őt a Framework összetevőibe, valamint a 2d/3d grafika, internetprogramozás, adatbázisok vagy a digitális képfeldolgozás alapjaiba. Mindezek közben persze egyre többet és többet tárok majd fel a .Net Framework-ről és a C#-ról. Ez az első rész egy kicsit ki fog lógni a sorból, mert a többi cikkel ellentétben sokkal általánosabb lesz: A .Net Framework-öt nem ismerőknek szól, és megpróbálja majd felkelteni az említett technológia vagy a C# nyelv iránti érdeklődésüket. A sorozat követését annak is ajánlom, aki a C#-tól eltérő, de szintén .Net nyelvvel foglalkozik!

A mi nyelvünk, a C#

A C#… van benne egy C betű. Nem meglepő, hiszen szintaktikailag sok közös vonása van a C nyelvvel. Aki mélyebbről is ismeri, az még a Java-val és a Visual Basic-kel is talál benne hasonlóságokat. A C# az említett három nyelv összeolvadásából született, és megpróbálta mindegyik előnyeit magával hozni: A C-től a tiszta, tömör, bár egy laikus számára néha átláthatatlan szintaktikát örökölte, a VB-től a programozás és a fejlesztői környezet használatának egyszerűségét, míg a Javától a futás közbeni fordítást, a cross-platform tulajdonságot, valamint az alapkönyvtár ötletét hozta magával. Ez a „cross-platform” annyit tesz, hogy több számítógép-architektúrán és több operációs rendszeren is elfutnak a C#-ban írt programok (bizonyos feltételek mellett). Már az elejétől fogva mennek ezek a programok mind PC-n, mind pedig Windows-alapú, hordozható számítógépeken is (pl. Pocket PC), és mivel a Microsoft (ez az egész az ő szüleménye) néhány dolgot közzé tett, megszületett a Project Mono, ami lehetővé teszi ezen alkalmazások Linuxon való futtatását is. Persze nem ugyanaz a bináris állomány fut el mindenütt, de a forráskódon csak minimális változtatásokat kell alkalmazni.

A .NET Framework újabb technológia az MS-tol

De mi teszi mindezt lehetővé? Ezen előnyök nagy része tulajdonképpen nem is magában a nyelvben rejlik, hanem abban a .Net Framework-nek elkeresztelt (.Net Keretrendszer) futtatási alrétegben, amit még a C#-on kívül sok-sok nyelv használ. Egy kicsit eltúlozva a Framework a Windows elavult, nehezen használható elsődleges programozási felületét, a Windows API-t hívatott leváltani, amit majd a következő windows-verzióban, a Longhornban meg is tesz. Bár egyes funkciókat megkövetel az őt használó nyelvektől, cserébe viszont annál többet kínál számukra. Ha egy program valamilyen operációs rendszertől függő eljárást szeretne elérni, azt a kérést nem a Linuxnak vagy a Windowsnak küldi el, hanem a Framework-nek, ami aztán megfelelő átalakítások után továbbküldi azt a „célszemélynek”. Így a programunk alatt csak egy a megfelelő operációs rendszerhez írt Framework-re van szükségünk (Linuxnál ez lenne a Mono), és onnantól kezdve kész van a „port”. Ennek hátránya viszont, hogy mivel a .Net Framework jelenleg egyik operációs rendszerben sincs benne gyárilag, ránk, programozókra hárul a feladat, hogy eljuttassuk a keretrendszert a felhasználóhoz.

Előnyök boncolgatása

Vegyünk sorra néhány a Keretrendszer által nyújtott dolgot:

Általános, szabványosított programozási felület. Ezt részben a minden nyelv számára elérhető, óriási BCL (base class library, alapvető osztályok könyvtára) teszi lehetővé, így tehát számítógéptől, operációs rendszertől és programozási nyelvtől függetlenül mindenki ugyanúgy hozhat létre / olvashat fájlokat, rajzolhat képeket, jeleníthet meg grafikus ablakokat, kommunikálhat az interneten stb… Persze a különböző nyelveken íródott programok/könyvtárak gond nélkül hívogathatják egymás eljárásait, mintha csak egy nyelven íródtak volna.

Futás közbeni fordítás. Ez a program futása elején eléggé lelassíthatja a dolgokat, de utána nem lesz sokkal lassabb, még mindig többször olyan gyors marad, mint egy VB-ben írt alkalmazás. Sokkal ügyesebben oldották meg ezt, mint a Java esetében, és így nem egy sima interpretert kaptunk, hanem egy másfajta JIT (Just-In-Time) fordítót, ami nagyságrendekkel hatékonyabb. Ennek az az oka, hogy egyrészt nem parancsonként, hanem egész eljárásonként fordít (tehát az eljárás meghívása elején egy picit várni kell, de utána natív kód sebességével fut a programunk), másrészt pedig ez lehetőséget ad a felhasználó hardver- és szoftverkörnyezetéhez optimalizált program létrehozására (magyarul pl. a forráskód megváltoztatása nélkül kaphatunk programot, ami szuperül megy régi gépeken is, de egy P4-en kihasználja a legújabb utasításkészleteket bár erre még éppen nem képes, de ténylegesen várható ilyen funkció a jövőben). A lefordított eljárások natív „képe” eltárolódik, így azt a következő meghíváskor már nem kell lefordítani.

Garbage Collector („szemétgyűjtő”). A C guruk tudják mennyi fejfájás származhat a memóriakezelésből. A VB-seknek és Java-soknak gőzük sincs róla, mert nekik már volt valami hasonló, ami levette a vállukról a memóriabajlódások többségét. Egyelőre csak azt kell megjegyeznünk, hogy még ha van is egy-két hátránya, képes hihetetlenül megkönnyíteni az életünket.

Persze az imént említett dolgok szorosan összefüggnek, hiszen például a „cross-platform”-ot a BCL és a JIT fordító közös erővel érik el. Eddig a .NET Framework tulajdonságairól a teljesség igénye nélkül írtam, annyit lehetne boncolgatni a témát, hogy azt egyelőre felesleges lenne az olvasóra zúdítani. Ha letöltjük az SDK-t, és elkezdjük tanulgatni a részleteket, ott úgy is mindent bőségesen kifejtenek.

Mire lesz szükségem?

Meglepő módon semmire. Vegyük elő a Jegyzettömböt és kódoljunk! Na jó, csak vicceltem. Bár tényleg megtehetnénk, hiszen a szükséges fordítók részei a Framework-nek (ami ingyenes), de azért könnyebb dolgunk lesz egy IDE-vel. Ám nem kell aggódni, most e téren is jó hírt hozok: A Visual C# 2005, Visual Basic .NET 2005, Visual Web Developer 2005 és SQL Server 2005 Express (butított, de minden meg van benne, amire csak szükségünk lehet) verzióit tanulási célokra ingyen a diákok kezébe adja a Microsoft! Így hát nem kell mást tenni, mint elbarangolni a http://lab.msdn.microsoft.com/vs2005/ címre, és egy regisztráció után már tölthetjük is a fejlesztőkörnyezetet. Minden, amit ott van az csak béta még, de 2005 eleje körül már várhatóak a végleges változatok is. Várunk mi addig? NEM!

És ami még várható…

A későbbi részekben, mint azt fentebb is említettem, konkrét feladatokat fogunk megoldani C#-ban. Ez a rész csak egy ismertető volt, de a cikksorozat későbbi részei sem fogják megtanítani, mi is a különbség egy for-ciklus és egy if-elágaztatás között. Ha viszont az olvasó érdekesnek találja a .Net-et, letöltheti az említett URL címről a szükséges programokat, és legalább minimális gyakorlatot szerez a következő számig (ha már ismer egy objektum-orientált nyelvet, ez úgysem lesz gond). A C# valóban egy olyan nyelv, aminél nem kell az általa állított akadályokra figyelni, és ténylegesen teljes erődből az aktuális problémára/programra lehet koncentrálni. Mindenkinek ajánlom, nem csak kezdőknek!

Pados Károly

Rekurzivitás 2. Rész

A rekurzivitásban rejlő veszélyek. Óvintézkedések

Rögtön az elején tisztázni szeretnénk egy apró félreértést. A cikksorozat első részében (lásd CodeX 2004.szeptemberi szám „Rekurzivitás elegancia veszélyekkel") említett könyvtárbejáró program forrásának bemutatására a cikksorozat utolsó részében kerül sor. Elnézést a félreérthető fogalmazásért mindazoktól, akik az előző számban keresték a forrást.

Az előző részben ismertettük a rekurzivitás fogalmát, valamint a függvény hívásakor zajló háttértevékenységeket. Itt az ideje, hogy megvizsgáljuk a rekurzivitás gyakorlati hasznát továbbá a benne rejlő veszélyeket, illetve a hibák elkerülésének egy lehetséges megvalósítását.

A rekurzivitás mélysége, más szóval a rekurzív hívások száma, egyenesen arányos az „elfogyasztott" veremterülettel. A verem mindaddig nem szabadul fel, míg a legbelső függvényből vissza nem térünk. Gyakorlatilag arról van szó, hogy a függvény lokális veremterületének visszafejtése nem történik meg. Emlékeztetőül: a cikksorozat előző részében felsoroltuk azokat a háttértevékenységeket, amelyek egy függvény hívásakor végrehajtódnak. Ezek közül a legutolsó, a 7. pont a verem visszafejtése, amely kizárólag akkor hajtódik végre, ha a legbelső függvényből visszatérünk. Az ezt megelőző tevékenységek azonban minden egyes függvényhíváskor végrehajtódnak, ami a veremmemória szabad kapacitásának folyamatos csökkenésével jár.

A veremmemória a program számára allokált memória része, amely véges, így előbb-utóbb betelik. A jelenséget veremtúlcsordulásnak (angolul „stack overflow") nevezik, és kellemetlen hibajelenségek forrása. A verem alulcsordulása is bekövetkezhet (angolul „stack underflow"), ha a veremről többet emelünk le, mint amennyit ténylegesen ráhelyeztünk. Utóbbi ritka jelenség, mivel a verem karbantartása amint azt már említettük nem a programozó feladata. Azonban némi assembly tudással lehetőségünk van „belepiszkálni" a verembe, a megfelelő regiszterek módosítgatásával. Másrészt a verem tartalma a PUSH / POP utasítások segítségével is módosítható. Ilyesmire viszonylag ritkán van szükség, leginkább elhivatott bitfaragók vagy kísérletező kedvű programozók kezelik ilyen alacsony szinten a vermet. Amennyiben valóban elkerülhetetlen a verem tartalmának illetve regisztereinek közvetlen módosítása, legyünk nagyon körültekintőek, hiszen igen érzékeny területen „garázdálkodunk".

Most, hogy a bemutatott problémák kellőképpen elvették a kedvünket a rekurzivitás alkalmazásától, vizsgáljuk meg mégis miért nem száműzték ezt a módszert. A rekurzivitás nagyon elegáns megoldásokhoz vezethet, hiszen általában sokkal rövidebb kódot eredményez, mint a klasszikus megoldás. Idézet a napokban megjelent Gyakorlati C++ című könyvem, a „Rekurzív függvények. A veremtúlcsordulás" című szövegrészből:

Ha egy szám faktoriálisát szeretnénk kiszámítani, a legegyszerűbben egy rekurzív függvénnyel tudjuk ezt megvalósítani. Egy pozitív egész szám faktoriálisát a következőképpen számítjuk ki:

n! = n * (n-1) * (n-2) * (n-3) ... * 1
Azaz például 7! = 7*6*5*4*3*2*1

Megfigyelhető, hogy egy ismétlődő műveletsorról van szó dekrementálás és szorzás -, és ezt a megfigyelést ültetjük át a gyakorlatba rekurzív függvényünkön keresztül:

```
unsigned long fakt ( unsigned long long_in )
{
      if ( long_in > 1 )
      {
            return long_in * fakt( long_in - 1
);;
      }
      else // kisebb, mint 1? -> 0! = 1
      {
            return 1;
      }
}
```

Megjegyzés:

A fenti függvény gyengéjét viszonylag gyorsan észre fogjuk venni, ugyanis a unsigned long int felső határát (4.294.967.295) elég gyorsan elérjük: 12! = 5.748.019.200.

A rekurzív függvények veszélyessé válhatnak, különösképpen, ha működésük bonyolult, nehezen átlátható, és a kilépés nincs minden esetben biztosítva. Ezenfelül, a rekurzív függvények végrehajtását lassítja a függvényverem felállítása, a függvényparaméterek és lokális változók lemásolása ami minden egyes függvényhíváskor megtörténik (nehogy egy újabb függvényhívás felülírja az előző lokális változóit, paramétereit).

Ezáltal minden egyes függvényhíváskor csökken a veremmemória szabadon felhasználható területe, és amikor végképp elfogy, az ún. veremtúlcsordulással (stack overflow) van dolgunk. Ennek az ellenkezője a verem alulcsordulása (stack underflow), ez olyankor szokott bekövetkezni, amikor több elemet emelünk le a veremről, mint amennyit ténylegesen elhelyeztünk benne.
A verem túlcsordulása gyakoribb az alulcsordulásnál, mivel nagy számú rekurzív függvényhívással aránylag gyorsan be lehet telíteni a veremmemóriát főleg ha az adott függvénynek méretes lokális objektumai vagy érték szerint átadott paraméterei vannak.

Ha egy rekurzív függvény kilépési pontja nem teljesen egyértelmű, biztonsági intézkedésként deklarálhatunk egy statikus számlálót, amit minden függvényhívás előtt növelünk, illetve utána csökkentünk. Ha elértünk egy megszabott felső határt, kilépünk a függvényből:

```
#define MAX_CALL_DEPTH 10

void recursive()
{
      static int cnt;
      if ( cnt >= MAX_CALL_DEPTH )
      {
            cout << endl << "Kritikus hiba!
Maximális függvényhívás- mélység! " << endl;
            return;
      }

      ++cnt; // rekurzív hívás előtt
inkrementáljuk
      cout << cnt << " ";
      recursive();
      --cnt; // rekurzív hívás után
dekrementáljuk
      cout << cnt << " ";
}
// főprogramint main()
{
      recursive();
      return 0;
}
```

A függvényhívások örvénylését megállítjuk a 10. hívás után:

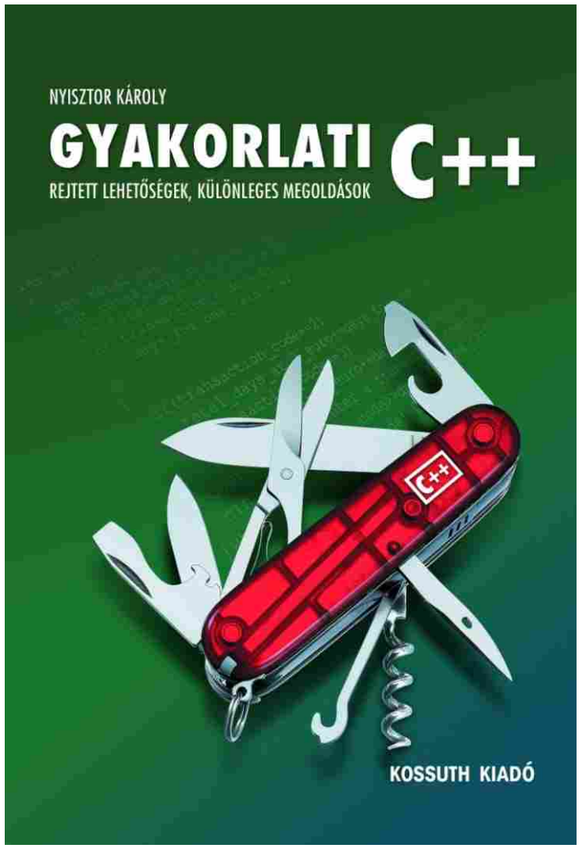
```
1 2 3 4 5 6 7 8 9 10
Kritikus hiba! Maximális függvényhívás-mélység!
9 8 7 6 5 4 3 2 1 0
```

MAX_CALL_DEPTH értéke elég nagy kell legyen ahhoz, hogy a függvény elvégezze feladatát, ugyanakkor meg kell akadályozza az eltúlzott rekurzióból származó veremtúlcsordulást - ha például függvényünk rekurzív módszerrel állományokat keres, értelmetlen az adott operációs rendszer által biztosított maximális könyvtár-mélységnél mélyebbre ásni)."

A rekurzivitásról szóló cikksorozat következő számában a függvényhívási konvenciókról lesz szó. A szabványos __cdecl C függvényhívási konvención kívül bemutatásra kerül még a Microsoft-specifikus __stdcall (WINAPI), a __fastcall illetve a thiscall is.

Addig is tiszta vermet és kevés „elszállást"! ;)

Nyisztor Károly
nyisztor.karoly@evosoft.hu



Nyisztor Károly számos cikkét olvashattuk már a CodeXben. Szerzőnk egy 400 oldalas könyvvel lepett meg minket, amely átfogó képet szeretne nyújtani a C++-ról, elsősorban gyakorlati szempontokból elemezve a nyelv elemeit. A Gyakorlati C++ című könyv októberben jelent meg a Kossuth kiadó gondozásában.

Következzen egy rövid összefoglaló a könyvről: „Könyvünk a C++ nyelv alapjait és működésének elvontabb, finomabb aspektusait egyaránt feldolgozza az előfeldolgozástól a kivételkezelésig.

Olyan gyakorlati problémákra kapunk választ, mint például az állományok befordításakor fellépő végtelen rekurzió, a számok gépi ábrázolása és az ebből eredő alul-, illetve felülcsordulási gondok, valamint a véges pontosság jelentése.

Konkrét megoldásokkal találkozhatunk a lebegőpontos számok bináris ábrázolására, egy dinamikus tömbosztály megvalósítására. Fény derül a nyelv egyik alapeleme, a pointerek belső működésére és a velük kapcsolatos veszélyekre. Betekintést nyerhetünk a függvényhívások rejtelmeibe, világossá válnak a rekurzív függvényhívásokban rejlő kockázatok, nyilvánvalóvá az okok. A könyv külön figyelmet fordít az objektumorientáltsággal felmerülő fogalmak tisztázására, az osztályok, struktúrák, virtuális függvények helyes használatára és optimális kiaknázására. Fontosságának megfelelő helyet kap a kivételkezelés. Természetesen nem maradhat el a nyelv legújabb vívmánya, az STL - a konténerek, iterátorok és algoritmusok - bemutatása sem."

A könyvről további részleteket a szerző weboldalán Olvashatunk: http://nkari.uw.hu

Cold Fusion MX 2.Rész

Kinek mi jut eszébe e szó hallatán? Nekem egy programozási nyelv, melyről eddig nem sokan hallottak kicsiny hazánkban. A Cold Fusion ahogy a cikk első részében is mondtam - egy webre szánt programozási nyelv, mellyel nagyon egyszerűen készíthetünk alkalmazásokat az internetre (vagy intranetre). Alkalmazás alatt értendő mondjuk egy portál, melyeket nap, mint nap is látogatunk.

Áttekintés

A cikk első részében megtárgyaltuk a Cold Fusion szintaktikáját, és azt is láttuk, hogy mennyire egyszerű, könnyen tanulható maga a nyelv, gondoljunk csak vissza az SQL lekérdezésekre. Mint ígértem, ez alkalommal megpróbálom bemutatni a CF-ben használt fontosabb függvényeket. Arra is ki fogunk térni, hogy hogyan hozzunk létre saját függvényeket, és hogy miként használjuk fel ezeket az alkalmazásainkban.

UDF Az meg mi?

Az UDF a User-Defined Function-nek a rövidítése, magyarul felhasználó által definiált függvény. Ez a szolgáltatás a Cold Fusion 5-ös verziójában jelent meg először, és természetesen a CF MX is támogatja. Az un. CFScript típusú kódokat a <cfscript> és a </cfscript> elemek közé kell elhelyeznünk.

```
<cfscript>
function HelloVilag() {
   var nev = "Hello Világ!";
   return nev;
}
</cfscript>

<cfoutput>#HelloVilag()#</cfoutput>
```

A kód első sorában a <cfscript>-tel jeleztük a fordító felé, hogy CFScript-et szeretnénk használni. A function után megadtuk a függvény nevét (HelloVilag), majd a var kulcsszóval deklaráltunk egy változót (nev). Ezek után visszatértünk a nev értékével, és a korábban megismert <cfoutput>-al megjelenítettük a képernyőn a változó tartalmát. A példából jól látható, hogy függvényeket a #FuggvenyNeve()# formában hívhatunk meg.

1 nyelv 2 szintaktika?

Ha azonban alaposabban szemügyre vesszük a forráskódot, két dolgot is tapasztalhatunk:

Először, a CFScript szintaktikája eléggé eltér az eddig megszokott CFML-étől (Cold Fusion Markup Language): az előbbi inkább a C-hez, míg az utóbbi a HTML-hez hasonlít. Néhány fejlesztőnek ez a kevert módszer kényelmetlen lehet.

Másodszor, a CFScript-ben nem helyezhettünk el Cold Fusion tagokat (pl. egy <cfoutput>), csak kifejezéseket, feltételes szerkezeteket, függvényeket stb. Például a képernyőre való íratás így festett a CF 5-ben függvényen belül CFScript nyelven, a writeoutput-on keresztül:

```
<cfscript>
function HelloVilag() {
         writeoutput("Hello Világ!");
}
</cfscript>

<cfoutput>#HelloVilag()#</cfoutput>
```

A Macromedia fejlesztői is rájöttek erre, hogy ez így nem nagyon lesz jó, éppen ezért megvalósították a <cffunction>-ös szerkezetet. Természetesen a Cold Fusion MX-ben, az új <cffunction> mellett továbbra is használhatjuk a <cfscript>-et. Visszatérve a Hello Világos példára, Cold Fusion MX-ben ez így is megvalósítható:

```
<cffunction name = "HelloVilag" hint = "Hello
Világ példaprogram" returntype = "string" output
= "true">
   <cfset nev = "Hello Világ!">
   <cfreturn nev>
</cffunction>

<cfoutput>#HelloVilag()#</cfoutput>
```

Tehát egy új függvény létrehozásakor meg kell adnunk a függvény nevét, a <cffunction> elem name attribútumával. Opcionális megjegyzést is fűzhetünk a függvényhez, erre a hint kulcsszó áll rendelkezésünkre. A hint után találunk egy returntype-ra keresztelt lehetőséget, melyből 13 típus áll rendelkezésünkre:

- Any (bármilyen)
- Array (tömb)
- Binary (bináris)
- Boolen (logikai)
- Numeric (szám)
- Date (dátum)
- Guid (globális egyedi azonosító)
- Query (lekérdezés)
- String (karaktersorozat)
- Struct (több változós struktúra)
- Uuid (univerzális egyedi azonosító)
- Variablename (változó)
- Void (nincs visszatérési érték)

Így pontosan meghatározhatjuk, hogy milyen típusú adatot fog visszaadni az adott függvényünk, és nem érhet minket váratlan meglepetés. Ez biztonsági okokból is jó megoldás, ugyanis ha nem olyan értéket kapunk vissza, amilyet előre meghatároztunk, akkor a fordító hibát jelez, és megszakad a program futása. Mivel a nev változó a példánkban egy karaktersorozatot (string), ezért a függvény visszatérési értéke is string lesz.

A <cfset> segítségével deklarálhatunk változókat, mint ahogy a CFScript-ben a var-ral. Az output 2 értéket vehet fel: true és false. Ezzel tulajdonképpen azt jelezzük a szerver felé, hogy ez a függvény a <cfoutput> között (true), vagy a <cfsilent> elemek között (false) fog megjelenni a későbbiekben.

CFLib.org Frissen, ropogósan

Ha fejlesztéseink során szükségünk lenne egy fontos funkcióra, és nem szeretnénk rajta órákat ülni, akkor csak látogassunk el a www.cflib.org weboldalra, ahonnan több mint 800 előre elkészített függvényt tölthetünk le! Az oldal Common Function Library Project keretében indult, s szinte mindent megtalálunk az e-mail cím ellenőrzéstől kezdve, a különböző titkosító algoritmusokig. Az oldalon lévő UDF-ket tetszés szerint módosíthatjuk, és felhasználhatjuk programjainkban.

Dátum

Jelenítsük meg a dátumot az alapértelmezett formázással!

```
<cfoutput>
   <p>
      Formázás nélkül: #Now()#
   </p>
   <p>
      Formázva: #DateFormat(Now())#,
#TimeFormat(Now())#
   </p>
</cfoutput>
```

Az aktuális dátumot és időt a Now() függvény adja vissza, melyet a Cold Fusion szervertől kér le. Lehetőségünk van ezt kedvünk szerint formázni; ebben a DateFormat(Datum()) és a TimeFormat(Ido()) segít.

A függvény-referenciát mindenki elérheti a Cold Fusion adminisztrátor (127.0.0.1/CFIDE/administrator/index.cfm) Documentation menüpontja alatt (CFML Reference).

Ciklusok

Nem létezik programozási nyelv ciklusok nélkül így van ez a Cold Fusion esetében is.
Írassuk ki a számokat 1-től 10-ig!

```
<cfloop index = "i" from = "1" to = "10">
   <cfoutput>
     #i#<br>
   </cfoutput>
</cfloop>
```

A <cfloop> elemmel hozhatunk létre ciklusokat, a ciklus nevét pedig az index attribútumával adhatjuk meg. A kezdőértéket a from, az utolsó számot a to jelenti. A ciklus minden lefutásakor kiíratjuk az i értékét és beszúrunk egy soremelést, hogy minden szám egymás alatt jelenjen meg. Végeredményül megkapjuk a számokat 1-től 10-ig.

Most annyit módosítsunk a kódon, hogy visszafelé (azaz csökkenő sorrendben) jelenjenek meg a számok:

```
<cfloop index = "i" from = "10" to = "1" step =
"-1">
   <cfoutput>
      #i#<br>
   </cfoutput>
</cfloop>
```

Itt a kezdő és az utolsó számot felcseréltük, és egy step tulajdonságot is beállítottunk. Ennek hatására a szám értéke mindig 1-el csökken, amíg el nem éri az 1-et.

Feltételes szerkezetek

Végrehajthatunk programrészeket egy feltétel teljesülése esetén is (pl. ha i = 8 akkor a ciklus álljon le):

```
<cfloop index = "i" from = "1" to = "10">
   <cfif i is "8">
      <cfbreak>
   </cfif>
   <cfoutput>
     #i#<br>
   </cfoutput>
</cfloop>
```

Ha az i értéke eléri a 8-at, akkor a <cfbreak> utasítással a ciklust megszakítjuk. Ennek hatására a számok kiírása 7-nél megszakad. Ezt feltételes ciklussal is megtehetjük:

```
<cfset i = 0>
<cfloop condition = "i LESS THAN OR EQUAL TO 7">
   <cfset i = i + 1>
   <cfoutput>
      #i#<br>
   </cfoutput>
</cfloop>
```

A második sorban egy feltételes ciklust hoztunk létre: ha i értéke kevesebb vagy egyenlő, mint 7, akkor álljon le. Az eredmény ugyanaz, mint az első példánál.

Tömbök

Nézzünk egy példát a tömbök használatára:

```
<cfset HelloVilagTomb = ArrayNew(1)>
<cfset temp = ArraySet(HelloVilagTomb, 1, 5,
"Hello Világ!")>
<cfset HelloVilagTomb[1] = "Hello Világ! 1.">
<cfset HelloVilagTomb[2] = "Hello Világ! 2.">
<cfset HelloVilagTomb[3] = "Hello Világ! 3.">
<cfset HelloVilagTomb[5] = "Hello Világ! 5.">

<cfloop index = "i" from = "1" to =
"#ArrayLen(HelloVilagTomb)# ">
   <cfoutput>
     #HelloVilagTomb[i]#<br>
   </cfoutput>
</cfloop>
```

Az első sorban létrehoztunk egy 1 dimenziós tömböt (több is lehet, maximum 3): ArrayNew(1), a másodikban megadtuk a tömb kezdő (1), utolsó elemét (5) és a tömb alapértelmezett elemének az értékét (Hello Világ). Ezek után beállítottuk az első, második, harmadik és utolsó elemek értékét, a negyedik elemet direkt nem deklaráltuk hogy lássuk, mi történik ilyenkor. Ezt követően egy index-es ciklussal kiírattuk a tömb elemeit. A 4. elemet nem deklaráltuk, így az alapértelmezett értéket veszi fel, azaz a Hello Világ-ot.

Hogyan tovább?

Rengetek lehetőség áll rendelkezésünkre, amikről még nem is beszéltünk. Például az UDF-ek formázása (megadhatjuk a sémát külső fájlként), külső függvények meghívása fájlokból stb. A cikk 3. részében egy adatbázis alapú vendégkönyvet fogunk elkészíteni.

Kiss-Tóth Marcell marcell@kiss-toth.hu

Egy rendszergazda hétköznapjai. XII

Az előző részben a nehezebben kezelhető, körülményes programok, vagy unalmas rutinfeladatok makrókkal való automatizálásáról volt szó. A számtalan érdeklődő levél mutatja, hogy olvasóink között sok olyan van, akiket az egyszerű, de praktikus, kezelési megoldások foglalkoztatnak. Ezúttal szintén egy rendszergazdáknak és felhasználóknak egyaránt hasznos, -talán sokak által már ismert- kis segédeszközzel fogunk megismerkedni…

A CD-RŐL MŰKÖDŐ OPERÁCIÓS RENDSZER

Miért hasznos?

Feltehetően számtalan olvasónk szembesült azzal a kellemetlen problémával, hogy a számítógépén valamilyen levakarhatatlan program élősködik. Hiába a csökkentett mód, a rendszer-visszaállítási pont, a parancssorban való bűvészkedés, illetve az egyéb, valóban sokszor igen hasznos eszköz: mégsem tudjuk letakarítani a betolakodót. Az is előfordul néha, hogy valaki elfelejti a jelszavát, és nem tud az adataihoz hozzáférni.

Azonban pl. egy selejtes merevlemezen keletkező, az operációs rendszer kulcsfontosságú állományaiban lévő szektorhiba is csúf dolgokat tud művelni. Ezekben az esetekben gyakran jó szolgálatot tehet egy olyan, külső adathordozóról BOOT-olni képes eszköz, mely képes írni/olvasni az NTFS, valamint a FAT32-es partíciókat a merevlemezen. FAT32-re van szép számban megoldás, de az NTFS gyakran feladja a leckét a felhasználóknak.

Szintén hasznos lenne, ha a WINDOWS alatt megszokott, grafikus környezetben tudnánk dolgozni, s esetleg az ott üzemelő segédeszközeinket futtatni. Egyik bevált gyakorlat, hogy kivesszük a merevlemezt a gépből, és egy másik, XP operációsrendszert tartalmazó számítógép merevlemeze mellé tesszük. Ekkor a gép saját rendszere látni fogja az adatokat. Azonban ez szereléssel járó, körülményes dolog, ami nem is mindig veszélytelen.

Végül előfordult, hogy valamilyen okból kifolyólag olyan weboldalakra kellene navigálnunk, ami arról híres, hogy mindenféle kémprogramot igyekszik mind újabb és újabb trükkökkel telepíteni gépünkre, valamint részben átkonfigurálja a rendszerünket. Egy ilyen akció után kemény munka, illetve ráfordított idő árán szabadulhatunk meg a hívatlan betolakodóktól, amik instabillá, lassúvá teszik a rendszerünket. Túl azon, hogy esetleg bizalmas adatainkhoz, szokásainkhoz is segíthetnek kívülállóknak hozzáférni, a gép rendes használatát is megnehezíthetik. Ekkor megint jó lenne egy olyan operációs rendszer, amit garantáltan nem tudnak ezek a rosszindulatú oldalak akaratunk ellenére módosítani.

Az igazság kedvéért megjegyzendő, hogy a védekezésnek nem feltétlenül az a legcélszerűbb módja, (cikksorozatunk egy későbbi részében visszatérünk rá) de vitathatatlanul hatékony!

Szintén hasznos, pl. az oktatásban, hiszen a CD-t behelyezve, illetve onnan BOOT-olva a gépet ideiglenesen, -remekül oktatható az adott operációsrendszer anélkül, hogy azt ténylegesen fel kellene telepíteni a merevlemezre. Az is körülbelül látható, hogy a hardveren mennyire lesz gyors, hatékony a futás, így telepítés előtt már sejtésünk lehet, mire számíthatunk.

Gyakorlati megvalósítások:

Először csak DOS-os kivitelben alkalmaztak BOOTCD-ket, mert a grafikus, multitaszkos környezetben ennek megvalósítása körülményes. Ezután a LINUX tábor lépett először, és egy oktatásra kiválóan alkalmas, grafikus rendszert hozott össze. A LINUX-os olvasók számára minden bizonnyal ismerős a KNOPPIX szó. Ez egy olyan grafikus felülettel működő DEBIAN LINUX disztribúció, ami CD-lemezről fut, és nem szükséges a helyi merevlemezre telepíteni. Az NTFS-t ez a rendszer is tudja olvasni. (Írni nem!) Bár eddigi tapasztalataim szerint nem egészen tökéletes még az NTFS kezelése.

A WINDOWS-használók számára sokáig nem volt hasonló megoldás, pedig egyre égetőbbé vált ennek szükségessége. Jó hír, hogy ezt felismerve lépett Bart Lagerweij, és készített egy ilyen rendszert, BART-PE néven. A rendszert egyfelől Bart-nak, másfelől pedig a Microsoftnak köszönhetjük. A Windows XP és WinPE (PE=preinstaller, azaz a rendszer felállásakor lefutó mini install program, ami az adott hardverhez igazítja a rendszert.) E nélkül aligha lehetett volna elkészíteni a CD-s utilt! Ugyanis ezek a rendszerek erősen építenek a hardverre, és a normál telepítés alatt optimalizálják magukat, következésképpen nem feltétlenül képesek futni ismeretlen környezetben. Az XP meglehetősen rugalmasan oldja meg ezt a problémát az automata hardverfelismeréssel, de ehhez hatalmas háttértár kapacitásra, valamint írható tárolóra lenne szükség. A CD pedig sem nem hatalmas kapacitású, sem nem „írható”. (Természetesen itt a szabad, realtime írhatóságról-olvashatóságról van szó, nem pedig a CD-R, illetve CD-RW eseti, lassú (újra)írásáról.) Ezt az ellentmondást oldotta fel Bart, Pl. a merevlemez helyett egy „RAMDRIVE” alkalmazásával.

Szerzői-jogi kérdések:

Azt kell mondani, hogy míg Bart munkája módosítás nélkül szabadon felhasználható, ez egyáltalán nem mondható el az XP-ről! Ez a MICROSOFT tulajdonát képezi, és fizetős. Ezért természetesen nem is lehet az Internetről letölteni készen a teljes lemezt, mint pl. a teljesen ingyenes KNOPPIX esetében. Erre a célra egy utilt kell leszedni, bekonfigurálni, majd elindítani. Ez a program kérni fogja a jogtiszta XP-telepítő lemezünket, és legenerál belőle egy ISO állományt. Ezt az állományt CD-re kiírva már előáll a kész rendszer.

A szerző felhívja minden olvasónk figyelmét, hogy az egyes országokban a helyi jogrendszer eltérhet, így a MICROSOFT licenszelési gyakorlata (EULA) is változhat. Mivel lapunk több országban megjelenik, ezért általános recepteket nem tudunk adni. Fontos tehát a Bart Lagerweij weblapján lévő tájékoztató információkat megszívlelni, illetve kikérni a hivatalos, helyi állásfoglalást, mielőtt a BOOT-CD előállítását elvégezzük. E nélkül igencsak kellemetlen meglepetés (jogi szankció) érheti a gyanútlan használót, amiért sem a lap, sem a cikk szerzője nem vállalja a felelősséget!

Fontos továbbá megemlíteni, hogy a CD-t generáló utilt eredeti angol nyelvű XP-re dolgozták ki. Egyéb, nemzeti verzióknál a util helyes működése nem garantálható…

Mit tartalmaz a telepítő csomag?

A telepítő csomag letölthető a szerző weblapjáról: http://www.nu2.nu/pebuilder/ További hasznos információkat számos helyen találhatunk a neten, mert a csomag igen népszerű. Pl. http://www.runtime.org/peb.htm. Az 1., ábracsoporton munka közben figyelhető meg a rendszer: 1., képcsoport, a BARTPE munka közben… (A képek forrása: http://www.nu2.nu/pebuilder/)

Fontos megemlíteni, hogy a konfiguráció testre szabható. A komponensek három részre oszlanak. Az első részt az XP-hez adott programok alkotják, (CMD, külső parancsok, kernel, és egyéb rendszerállományok) a másodikat az a kilenc FREWARE util, amit Bart hasznosnak látott a rendszerhez hozzáadni.(pl. Egy kézreálló fájlkezelő, DRIVE-klónozó, hálózati konfigurációs modul, Internet-böngésző, stb.…) Végül a harmadik részt olyan programok, amelyek bár nem ingyenesek, (így az alap rendszerben sem szerepelhetnek,) de megvásárlásuk esetén utólag beépíthetők a rendszerbe, számukra előkészítették a beillesztést. Ilyen pl. a víruskereső program, vagy az egyik legnépszerűbb klónozó program 32-biten futni képes változata, a GHOST32.

A rendszer a Windowsban is megszokott .CAB (cabinet fájl) kiterjesztésű, tömörített állományokban tartja a járulékos, illetve utólag beilleszthető programokat is, melyeket induláskor tömörít ki a ramdrive-ra.

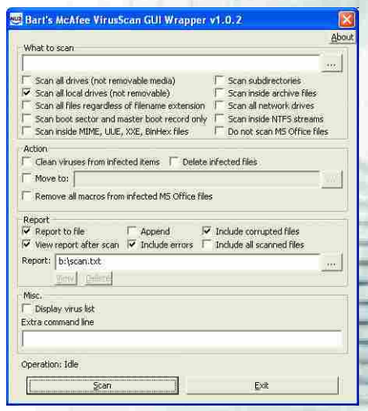
Ettől függetlenül lehetőség van teljesen más programok hozzáadására, a startmenü, vagy az indítópult bővítésére, stb. Egyszóval nagyon rugalmasan lehet a rendszert összerakni. Igazán dicséretes munkát végzett a készítője, akit a neten sokfele egyszerűen csak „megváltónak” becéznek a BARTPE-rajongók… .

Kis Norbert norbimagan@freemail.hu

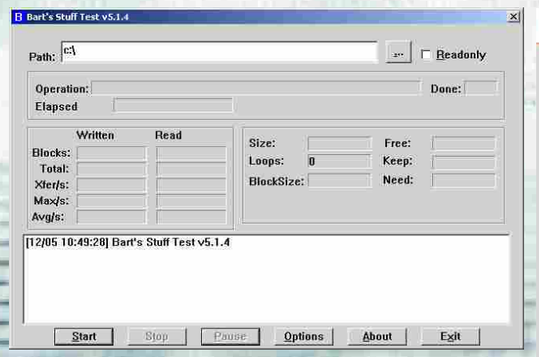
A fájlkezelő nem a Windows megszokott intézője, azonban igen kézreálló.

Bár az asztal a hagyományos XP-ben megszokottnál egyszerűbb, de a startmenü itt is megtalálható

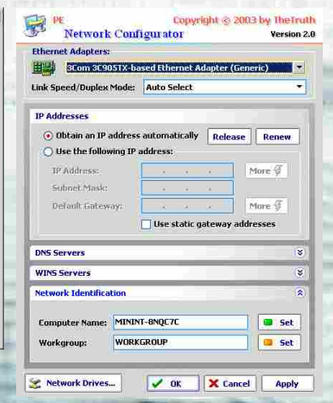
A profilok alkalmazása újabb hasznos apróság.



A víruskereső elmaradhatatlan tartozéka egy BOOT-lemeznek. Baj esetén nagyon hasznos!



A CD-n több util kapott helyet. Pl. található itt ingyenes HDD-image szoftver, melynek segítségével a merevlemezek tartalma klónozható. Természetesen tömöríteni is tudja a kapott állományt.



A rendszer az automatikus konfiguráció után a hálózati beállításokra is rákérdez. Megadható DHCP, vagy fix IP, ahogy a helyzet megkívánja.

Játékfejlesztés: percek alatt (2. Rész)

Az előző részben elkezdett játékunkat fogjuk most folytatni. Most az irányítást fejlesztjük tovább, valamint zenét is le fogunk játszani, mialatt fut a játék, valamint megismerkedhetünk egy egyszerű és hasznos pályaszerkesztővel, mellyel percek alatt elkészíthetünk játékunkhoz egy-egy pályát.

Az előző részben csak négy gombunk volt, melyek segítségével mozgathattuk a kamerát. Most beállítjuk az egeret is, hisz ez ma már alapfeltétel egy 3D-s FPS játéknál. Közvetlenül a do után gépeljük be a következőket:

```
camerax#=wrapvalue(camerax#+mousemovey())
cameray#=wrapvalue(cameray#+mousemovex())
cameraz#=wrapvalue(cameraz#+mousemovez())
rotate camera camerax#,cameray#,cameraz#
```

Ezzel azt értük el, hogy az egérrel a kamerát mindig arra fordítjuk, amerre az egeret mozgatjuk. Ha kipróbáljuk, meglátjuk, hogy ez a kód még mindig finomításra szorul, hisz ha a plafont vesszük célba, és megnyomjuk az előre gombot, akkor „felrepülünk”, majd kirepülünk a szobából. Ezt a későbbiekben természetesen orvosolni fogjuk.
Most pedig töltsünk be egy zeneszámot, hogy élvezetesebbé tegyük a játékot!

A DBPro már tud mp3-mat kezelni, így most egy ilyen fájlt fogunk betölteni, és lejátszani, ismételni.
A do elé írjuk be a következő sort:

```
load music "rithm_world.mp3", 501
```

majd a billentyűkezelő parancsok után írjuk be ezt:

```
play music 501
loop music 501
```

Néhányakban felvetődhet a kérdés, hogy a fájlnév után mi az az 501? A DBPro-ban mindig, amikor betöltünk valamit, vagy létrehozunk valamit, meg kell adni egy számot, lényegében egy azonosítót. Jól látható, hogy most betöltjük a „rithm_world.mp3” nevű fájlt az 501-es helyre, és később csak erre a számra kell hivatkoznunk, nem kell megadnunk újból a szám címét, és hogy honnan töltjük be.

Egy kis kitérő: a forrásban érdemes megjegyzéseket elhelyezni. Erre minden nyelvben van valamiféle lehetőség, valami parancs. A DarkBasicPro-ban a „rem” szócska mögé helyezhetjük a kommentárjainkat. Ha hosszabban szeretnénk valamit elmagyarázni, akkor érdemesebb a „remstart” és a „remend” parancsokat használni. Próbáljuk is ki! A „load music "rithm_world.mp3", 501” sor elé írjuk be:

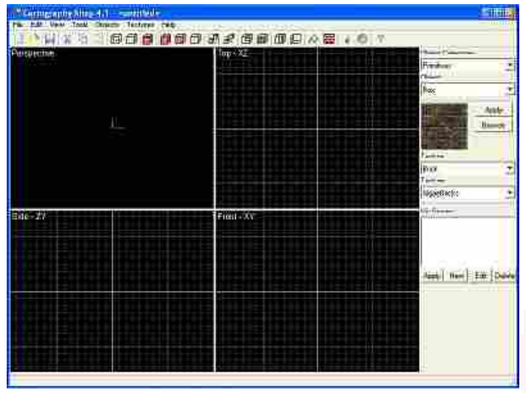
```
Rem köszönet Laguel Redouane-nak (avagy
Rodgonak), hogy rendelkezésünkre
bocsátotta a szerzeményét!
```

Amint látjuk, ezt a sort a szerkesztő más színnel is írja ki, így téve egyértelművé, hogy ami ebben sorban van, az nem hajtódik végre a program futása során.
Most próbáljuk ki a másik lehetőséget! A kód legelejére írjuk be:

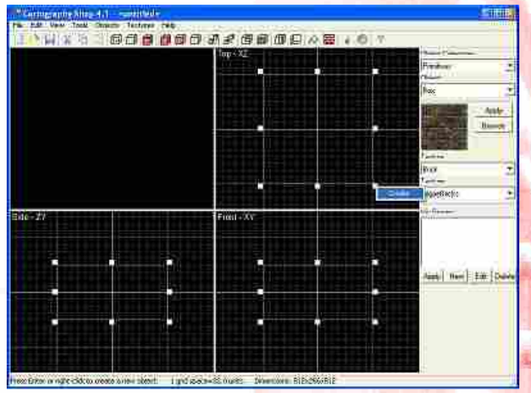
```
Remstart
Pályák betöltése  1-től 200-ig
Zenék betöltése  501-től 600-ig
Egységek betöltése  1000-től 2000-ig
Képek betöltése  4000-től 5000-ig
Remend
```

Itt is észrevesszük, hogy megint csak más színnel írta ki a fenti pár sort (alapbeállításokkal szürke). És hogy a fenti pár sor miért is lehet fontos? Mit kell magyarázni egy „rem remstart remend” szócskán? Amint látható, itt fent leírtuk magunknak, hogy milyen médiákat milyen sorszámú helyekre fogunk betölteni. Ez nagy segítség lesz a későbbiekben! Természetesen itt mindenki olyan számokat ad meg, amilyenek tetszenek neki, a lényeg, hogy amikor a programunk elkezd növekedni, akkor csak a kód elejére ugrunk, megnézzük, hogy amit be akarunk tölteni, azt hova is töltsük be (500-as kategória, 1000-res kategória, stb.).

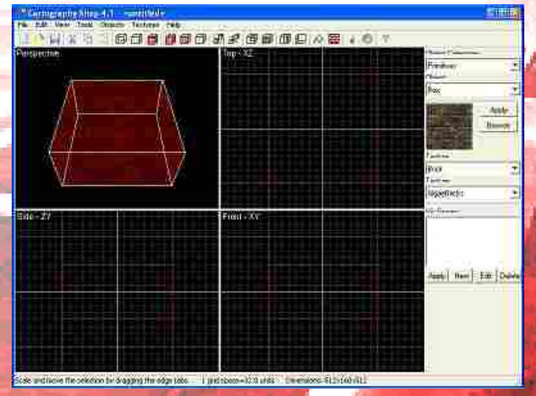
Most pedig jöjjön egy kis pályaszerkesztés játékunkhoz! A DBPro-hoz kapható a Carography Shop nevű szoftver. Ezzel igen könnyen és gyorsan tudunk pályákat készíteni születendő FPS játékainkhoz. Az angol honlapról letölthető a demo verzió (nem tudunk menteni link lentebb). Indítsuk el! Ha minden rendben van, ezt kell látnunk:



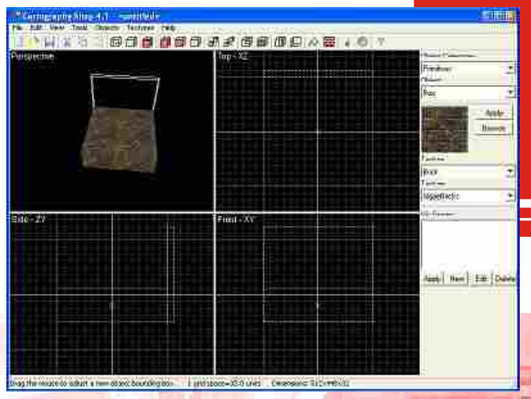
Majd az Objects menüben válaszzuk ki a Primiteves / Box pontot. A Top ablakban kattintsunk bal egérgombbal egy helyre, majd húzzuk el jobbra lefelé az egeret. Ha elengedtük az egér gombját, kattintsunk az objektumra jobb egérgombbal, és válasszuk a „Create” funkciót. Ha ezt nem tesszük meg, akkor a program nem hozza létre az adott testet! Ha nem végezzük el ezt a „létrehozás” funkciót minden objektumnál, akkor a következő kattintásnál (ha az nem az objektumon van), az objektum eltűnik.



Ha sikeresen létrehoztuk az objektumot, akkor kedvünkre átméretezhetjük!

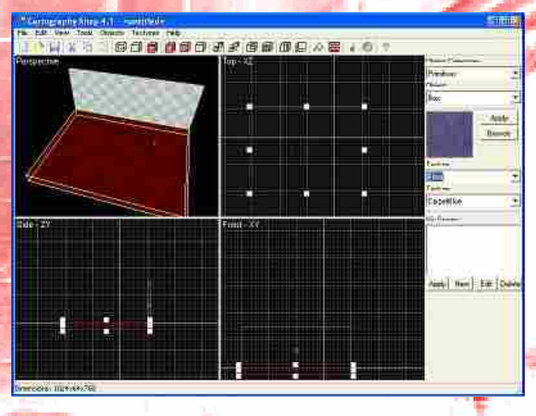


Ha készen vagyunk a „padlóval”, létrehozhatunk falakat, oszlopokat, tetőt, stb.



Megjegyzés: a bal felső sarokban „készen” textúrázva látjuk a pályánkat. Kattintsunk a jobb egérgombbal, majd mozdítsuk el! Ha görgős az egerünk, tudunk közelíteni és távolítani is a pályánkhoz! A másik három szerkesztőablakban a le-föl-balra-jobbra billentyűkkel egyszerűen mozoghatunk, illetve egerünk görgőjével itt is közelíthetünk, távolodhatunk. A görgőt helyettesítendő, megtehetjük ezt a + , - jelekkel is.

A textúrázás igen egyszerűen működik: kijelöljük az objektumot, amit textúrázni kívánunk, és a jobb oldali panelen kiválasztjuk a listából azt a képet, amit rá szeretnénk húzni.



A következő részben nem csak a kódot írjuk tovább, és a pályaszerkesztést folytatjuk, de még a modellezéshez is hozzálátunk, hogy legyen majd mivel lövöldöznünk!

Kapcsolódó oldalak:
www.thegamecreators.com
www.darkbasic.hu
www.jatekfejlesztes.hu

Keleti István jatekfejlesztes@jatekfejlesztes.hu

Java 3D avagy 3D grafika bárhol

Java. Ez a szó mindenkinek ugyanazt kell az eszébe juttassa: objektumorientáltság, platformfüggetlenség, könnyen elsajátítható szintaxis. 3D. Ezt olvasván pedig mindenki pörgő cselekményre gondol, háromdimenziós világba helyezett agyonoptimizált kódok, fölösleges utasítások nélkül. Most pedig joggal jöhet a kérdés: Miként lehetséges ezt a két fogalmat kibékíteni egymással. Márpedig lehetséges. Hogy miként? Hát erről szól a cikk.

Mi is ez a Java3D API?

A Java3D egy Java standard kiterjesztés, mely segítségével 3D grafikát megjelenítő, illetve ezzel a grafikával interakciót végző Java programokat készíthetünk. Az API a rendelkezésünkre bocsát egy osztálygyűjteményt, mely minden 3D grafikához kapcsolódó művelet elvégzését biztosítja. Ezen osztályok ú.n. magas-szintű hozzáférést biztosítanak a grafikai elemekhez, ez alatt azt kell érteni, hogy nekünk nem kell olyasmivel bajlódni, mint koordináta transzformációk, meg „polygon fillező” algoritmusok gyártása.

A Java3D ideális a kezdő 3D programozók számára, mert könnyen elsajátítható szintaxisa révén betekintést nyújt a 3D világ felépítésébe, anélkül, hogy a programozónak bajlódnia kellene mélyremenő szintekig a koordinátákkal, meg hasonló „kis” részletkérdésekkel. Ideális eszköz szemléltető alkalmazások, 3D bemutatók készítésére, hisz aránylag kis erőfeszítéssel nagyon látványos eredményre juthatunk.

Platform, függetlenség és másegyebek

A Java3D csomag mostmár szinte minden operációs rendszerre elkészült, bővebb információ a **www.j3d.org** címen található. Szintén ezen a címen (angolul nem tudók részére: sajnos angolul) található rengeteg tutorial, kód, meg link, ahol sok Java3D hez kapcsolódó hasznos információ található.

Manapság mindent le lehet tölteni az internetről, így a Java3D t is, a következő címről: **http://java.sun.com/products/java-media/3D/**. Miután kiválasztottuk a nekünk megfelelő változatot, és le is töltöttük, egyszerűen végrehajtva a letöltött programot már fel is installálódott a rendszer. Vigyázzunk azonban egy dologra: a Java3D nek minimum egy Direct3D (Windows) kompatibilis videokártya az alapigénye, jobb esetben pedig megpróbálkozhatunk az OpenGL változattal is.

Sebességgondok a java programok esetében bizony előfordulnak, viszont ez a Java3D esetében a minimálisra lett csökkentve. A Java3D egyenes kapcsolatban van az alatta elterülő réteggel, ami nem más mint a Direct3D, vagy az OpenGL, és aránylag jó eredményeket ér el. Sajnos nincs mód ezt a két réteget „direkt” programozni Java ból, csak közvetett módon, az API segítségével érhetjük el a grafikát.

Nézzünk bele a csomagba

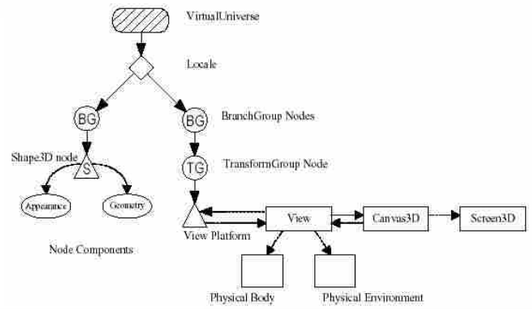
Minden Java3D program Java3D objektumokból van összerakva, szigorúan követve az objektumorientált programozás szabályait. Egy Java3D program mindig egy virtuális világot tartalmaz (VirtualUniverse). Ez a virtuális világ tartalmazza a maga során a 3D objektumokat, és az ezeken az objektumokon végrehajtott transzformációkat (forgatás, eltolás... igen, ezek is objektumok).

Egy Java3D virtuális univerzum egy gráf alapján van felépítve, mely sokatmondóan a „Scene graph” nevet viseli. Ez a Scene graph tartalmazza az összes Java3D objektumot, amelyek a világ geometriáját definiálják. Ez a gráf csomópontokat tartalmaz, amelyek nem másak, mint a világban előforduló objektumok, forgatások, fények, satöbbi. Ezek közt a csomópontok közt a legtöbb esetben apa fiu kapcsolat van (pld, egy transzformációs csoport tartalmaz egy kockát), viszont pár esetben referencia típusú a kapcsolat (egy 3D objektum tartalmaz egy referenciát a pontokra, amelyek őt alkotják). A csomópontok olyan tulajdonságokkal bírnak, mint például futás közbeni módosítás (a csomópont tulajdonságainak írása, olvasása, például a koordináták változása, amit igen jól lehet használni a forgatáskor, meg morpholáskor, stb...) és másegyebek, melyekre nem térek most ki, hisz megtalálhatóak a Java3D dokumentációban, melyet a fent említett címről lehet letölteni.

A Scene graphnak két fő ága van: az egyik, mely tartalmazza az objektumokat, ez a „content branch”, vagyis a tartalom ág nevet viseli. A másik ág a világban előforduló transzformációkat tartalmazza. Ennek a neve: „view branch”.

Egy másik fontos művelet a Scene graph lefordítása, vagyis „kompilálása”. Ezzel operációs rendszer baráttá tesszük az osztályunkat, és rengeteg számítást elvégezünk, úgymond előregenerálunk egy pár változót. A következő kép a Java3D világ Scene graphjának bemutatását tűzte ki céljául.

Forrás: Sun Microsystems



A világ receptje

Hogy egy Java3D programot logikailag helyesen fel tudjunk építeni, a következő lépéseket kell végrehajtani:

1. Hozzunk létre egy Canvas3D objektumot. Ez az objektum lesz felelős a felhasználóval való kommunikációért, illetve ez a Java objektum, amely „rajzolni tud” a képernyőre.

2. Hozzunk létre egy VirtualUniverse objektumot. Ez lesz a világunk alapja.

3. Hozzunk létre egy Locale objektumot. Ez fogja tartalmazni a virtuális világ összes 3D objektumát.

4. Hozzuk létre a view branchot:
   a. Hozzunk létre egy View objektumot
   b. Hozzunk létre egy ViewPlatform objektumot
   c. Hozzunk létre egy PhysicalBody objektumot
   d. Hozzunk létre egy PhysicalEnvironment objektumot.
   e. Kapcsoljuk össze a ViewPlatform, PhyisicalBody, PhyisicalEnvironment, és a Canvas3D objektumokat a View objektummal.

5. Hozzuk létre a ContentBranch-ot (azaz a 3D világ objektumait, mint pl. kocka, satöbbi)

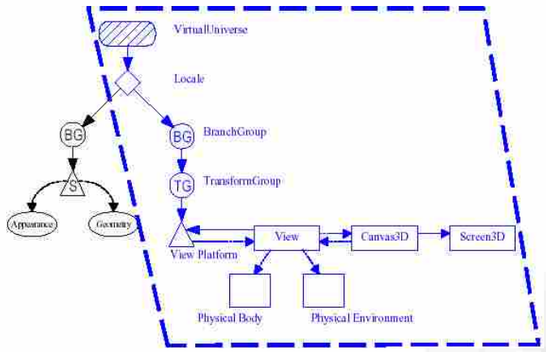
6. Fordítsuk le a gráfot

7. Szúrjuk be a gráfot a Locale objektumba

Egy egyszerű világ

Azoknak, akiket megijesztett az előbbi lista, megnyugtatásként közlöm: van egyszerűbb módja is, hogy Java3D programot írjunk. Mégpedig használván egy SimpleUniverse objektumot, amelyet természetesen a SimpleUniverse osztály alapján hozunk létre. Ez a SimpleUniverse osztály teljes egészében kiveszi kezünkből a 4. pontot, és létrehozza az összes szükséges objektumot, mint ahogy a következő kép mutatja

Forrás: Sun Microsystems



A kékkel bekeretezett rész kezelését egyszerűen el lehet felejteni, nekünk nem marad más, mint az objektumok, és transzformációk létrehozása, illetve beillesztése a virtuális világba.

És végül egy program...

...mely a szokásos alapműveletet végzi el: forgat egy kockát. Figyeljük meg, milyen elegáns módon van „elintézve” a kocka forgatása: Egyszerűen hozzáadtunk egy Rotation Interpolator objektumot a gráfhoz. És ennyi. Apropó, ahhoz, hogy leforduljon, szükség van a java3d jar file-ok berakására a CLASSPATH ba, mint pl: j3dcore.jar, satöbbi...

```
import java.applet.Applet;
import java.awt.BorderLayout;
import java.awt.event.*;
import java.awt.GraphicsConfiguration;
import com.sun.j3d.utils.applet.MainFrame;
import com.sun.j3d.utils.geometry.ColorCube;
import com.sun.j3d.utils.universe.*;
import javax.media.j3d.*;
import javax.vecmath.*;

public class HelloUniverse extends Applet {
        /**
         * Ez a fuggveny letrehozza a scene graph objektumot.
         **/
        public BranchGroup createSceneGraph() {

                // Ez lesz a graf "gyokere"
                BranchGroup objRoot = new BranchGroup();

                // Hozzuk letre a transzformacios csoportot,
                // melynek beallitjuk azt a tulajdonsagat, hogy
                // futas kozben modisithato legyen.
                TransformGroup objTrans = new
                TransformGroup();
                        objTrans.setCapability
                (TransformGroup.ALLOW_TRANSFORM_WRITE);

                //Adjuk hozza ezt az obejktumot a grafhoz.
                objRoot.addChild(objTrans);

                // Hozzunk letre egy egyszeru objektumot, pld egy
szines kockat
                objTrans.addChild(new ColorCube(0.4));

                // Hozzunk letre egy viselkedes objektumot, amely
szerep
                // az lesz, hogy elvegzi a szukseges
transzformaciot a
                // megfelelo transfrom groupon.
                Transform3D yAxis = new Transform3D();
                Alpha rotationAlpha = new Alpha(-1, 4000);

                RotationInterpolator rotator =
                        new RotationInterpolator(rotationAlpha,
objTrans, yAxis, 0.0f,
                                        (float) Math.PI * 2.0f);
                BoundingSphere bounds =
                        new BoundingSphere(new Point3d(0.0, 0.0,
0.0), 100.0);
                rotator.setSchedulingBounds(bounds);
                objRoot.addChild(rotator);

                // Forditsuk le a grafot
                objRoot.compile();

                return objRoot;
        }
        /**
         * Ez egy uj obketkumot hoz letre.
         **/
        public HelloUniverse() {
                setLayout(new BorderLayout());
                // hasznaljuk a SimpleUniverse osztalyt, hogy
megkimeljuk
                // magunkat sok munkatol
                GraphicsConfiguration config =

SimpleUniverse.getPreferredConfiguration();

                Canvas3D c = new Canvas3D(config);
                add("Center", c);

                // Hozzuk letre a scene graphot.
                BranchGroup scene = createSceneGraph();
                SimpleUniverse u = new SimpleUniverse(c);
                // Kis modositas, hogy a "legjobb" ralatast
kapjuk a vilagra.
u.getViewingPlatform().setNominalViewingTransform();

                u.addBranchGraph(scene);
        }
        /**
         * Ezzel futtatjuk a programot
         **/
        public static void main(String[] args) {
                new MainFrame(new HelloUniverse(), 256, 256);
        }
}
```

Akiknek a fantáziáját megragadtam ezzel a kis bemutatóval, annak javallom, hogy mélyebben tanulmányozza ezt a java kiterjesztést, hisz nagyon „handy tool”-t kap a kezébe 3D világok tervezéséhez.

Deák Ferenc f.deak@freemail.hu

Vezérlés az LPT-porton

Alfától Omegáig. XII.

Az előző cikkünkben elkezdtünk ismerkedni a léptetőmotorokkal. Nagy vonalakban áttekintettük az elvi felépítést, működést, a z olcsó beszerzési lehetőségeket, (a szaküzleteken kívül milyen kidobott számítástechnikai eszközökben fordul elő gyakrabban). Most újabb hasznos információkat sajátíthatunk el…

Kis gyakorlati feladat: kivezetések beazonosítása:

Az előző számban arra biztattam minden olvasót, hogy keressen 5/6 vezetékes léptetőmotorokat. Akinek sikerült találnia, most elvégezheti a kivezetések beazonosítását… Ha visszaemlékszünk, akkor mindig egy tekercs van áram alatt, és a tekercseket egymás után, sorban ki / bekapcsolgatjuk. Ennek hatására a kis mágneses „fogacskák” szépen lépkednek a következő pozícióba.

Ezt az ismeretünket fogjuk felhasználni a munkánk során. A tekercsek beazonosítását értelemszerűen úgy lehet megtenni, hogy előbb ellenállásmérővel kikeressük a közös vége(ke)t. Ez, mint már az előző részben szó volt róla, 5-kivezetéses motoroknál egy, 6-kivezetéses motoroknál, pedig két szál drótot jelent. A mérésnél úgy tekintjük, hogy a tekercsek Y-alakba kötött ellenállások, tehát azok a végek, amik egymáshoz képest nagyobb (dupla) ellenállást mutatnak, a fáziskivezetések. A fennmaradó egy, illetve két vég pedig, - ami a kisebbik ellenállást mutatja a többivel -, lesz a közös kivezetés. Az előző számban látható volt a motor villamos jelölése. Ott hat kivezetéses kivitel van feltüntetve és a közös kivezetéseknek a tekercsek középső „megcsapolásai” felelnek meg. Értelemszerűen az öt kivezetéses motornál ezek a fémházon belül már eleve közösítve vannak. Hat kivezetésnél, pedig nekünk kell ezt megtenni. Remélhetőleg, a rajz, illetve a leírás alapján egyszerű lesz a közös szálak kikeresése.

Miután ezzel megvagyunk, kapcsoljuk a közös véget fixen a PC-nk tápegységének +12V-os kivezetésére. (Emlékezzünk: sárga színű vezeték, pl. a HDD tápkábelen!) Most négy, szabad vég marad a kezünkben. Ez lesz a négy fáziskivezetés. Egyelőre mindegyik a levegőben lóg, és nem ér sem egymáshoz, sem a GND-hez (Fekete vezeték, vagy a számítógép fémburkolata.) A közös kivezetésre ragasszunk egy „+” címkét, hogy pontosan tudjuk, hova való.

FONTOS FIGYELMEZTETÉS! A SZÁMÍTÓGÉP ÁRAMKÖREIT, A PORTOK KIVEZETÉSEIT VÉLETLENÜL SE ÉRINTSÜK MEG SEM A 12V-AL, SEM A LÉPTETŐMOTOR SZABAD VÉGEIVEL, (AZAZ ÜGYELJÜNK, NEHOGY A LEVEGŐBEN LÓGÓ SZÁLAK BAJT OKOZHASSANAK) MERT AZ A RENDSZER AZONNALI, ÉS TELJES MEGHIBÁSODÁSÁT OKOZZA! (A PIROS +5V ÉS A SÁRGA, +12V ÖSSZEÉRINTÉSE IS HASONLÓAN TRAGIKUS LEHET A GÉPÜNK ÉLETÉRE NÉZVE.) A GYAKORLOTTABB OLVASÓINK SZÁMÁRA TALÁN TRIVIÁLIS DOLGOK EZEK, AZONBAN GONDOLNUNK KELL A KEZDŐKRE IS… MINT ANNYISZOR MÁR, A SZERZŐ JAVASOLJA, HA BIZONYTALANOK VAGYUNK AZ ELEKTRONIKÁBAN, AKKOR KÉRJÜK INKÁBB EGY KÖZELI SZAKEMBER TANÁCSÁT, VAGY SEGÍTSÉGÉT.

Most jön az izgalmas rész: megkeressük, és megszámozzuk a fáziskivezetéseket! Ehhez találomra kiválasztunk egyetlen szál vezetéket, és ráragasztunk egy címkét, 1-es felirattal. Most keressük a hozzá való kettest. A kettes az lesz, amelyik a motort jobb kéz felé, (óramutató járásának irányában) a következő, legkisebb elmozdulást mutató pozícióba fordítja..

Azaz: alaphelyzetbe hozzuk a motort, tehát az egyes kivezetést hozzáérintjük pillanatra a GND-re. Motorkánk a pozíciójától függően vagy nem modul sehova, (mert eleve jó helyen állt a tengelye) vagy beugrik a helyére. Most elengedjük az 1-es számmal jelzett szálat, és találomra megfogunk egy másikat, majd azt érintjük GND-re.

Ekkor egy ugrást látunk előre, vagy hátra. Megjegyezzük, hogy merre, illetve mekkorát lépett a motor, s a szálat félrehajlítjuk. Most ismét az egyes számú kivezetést kötjük GND-re, hogy a motor visszaálljon a kezdeti pozícióba, majd a maradék két szál közül egyet GND-re érintünk. Megint megfigyeljük, merre és mekkora az elmozdulás. Ezután megint jön az alaphelyzet, vagyis az egyes szállal való visszaállítás. Ebben a sorozatban végezetül a harmadik, utolsó jelöletlen drót GND-re érintése. Így már mindhárom lépési lehetőséget látva tudni fogjuk, melyik jelentette a helyes variációt. Erre a 2-es feliratot ragasszuk rá!

Ha megvan az egyes, illetve a kettesszámú fázisunk is, már csupán el kell dönteni: A maradék kettő közül vajon melyik a 3-as, illetve 4-es? Mi sem egyszerűbb… Most a kettes pozícióba léptessük a motorunkat! A fentihez hasonló módon érintsük GND-re pillanatra a kettesszámú, ezután a két ismeretlen szál egyikét. Most megint a kettest, majd a maradék ismeretlent. Amelyik a helyes irányú és mértékű elmozdulást adta, arra tegyük a 3-as címkét, a másikra, pedig a 4-eset. Ennyi az egész!!! Elsőre talán körülményes, de ha megszokjuk a módszert, egy motort kb. egy perc alatt lehet így kimérni.

Illendő ilyenkor egy utolsó tesztet végezni: 1,2,3,4 sorrendben érintgessük felváltva a vezetékeket GND-re, és lássunk csodát, a motorkánk szépen, egyenletes lépésekkel bicegve fordulni fog. Megpróbálhatjuk a 4,3,2,1 sorrendet is, akkor meg visszafele fog fordulni.

A teszt során hasznos hosszabb mutatót készíteni a tengelyre, hogy a csillogó, vékony fém ellenére is jól látható legyen az elmozdulás. Ez célszerűen lehet egy kidobott rádió forgatógombja, amit kis csavarral rögzíthető, s erre már pillanatragasztóval tudunk szívószálat, hurkapálcikát erősíteni. Nagyon szépen felnagyítja a picike elfordulást, hatalmas segítséget jelentve ezzel a tesztelés során. Azonban a fantázia határtalan: próbálkozhatunk öntapadós árazó címkével, a tengelyre ráhúzott, vastag, pipa alakban meghajlított szívószállal, stb…

A fáziskivezetések és a GND közé betehetünk nyomógombokat, mikrokapcsolókat, s ekkor a számítógép, illetve az elektronika működését kényelmesebben tudjuk szimulálni, illetve próbálgatni a téves kombinációk hatásait a motorra.

Teljes lépéses vezérlési mód:

Miután az egyszerűbb, de durvább lépést adó egészlépéses üzemmódot megismertük, most áttekintjük, hogyan lehet az előbbi felbontásra ráduplázni, és a fordulat század részét jelentő lépés helyett most 200 részre osztjuk el a kört!

Ezt úgy tudjuk megtenni, hogy nem egyszerűen a következő tekercsre átkapcsoljuk az áramot, hanem bevezetünk egy középső fázist, a „fél lépést”, amikor egy időben a két szomszédos tekercs van bekapcsolva. Értelemszerűen a motor áramfelvétele ekkor dupla nagyságú, viszont a mágneses fogacskák a forgórésszel úgy fordulnak be, hogy az állórész két fogacskája közé mutatnak. Ennek oka egyszerű: a két lépéshez tartozó fogak egyforma erővel húzzák a forgórészt, így az a kettő között lesz stabil állapotban.

Fontos megemlíteni, hogy a fél lépés állapot csak addig marad fent, míg a motor áram alatt van. Ha az áramot kikapcsoljuk, akkor előre/hátra a legközelebbi egészlépésbe fog azonnal beállni a motor. Erre a programozás során érdemes ügyelni majd! Ugyanis az áramot a motorunkra ismét rákapcsolva egy adott ideig a fél lépés kombinációját kint kell tartani, majd csak utána lehet a következő állapot kombinációját beállítanunk, ha biztosan finom indítást szeretnénk elérni. A féllépéses vezérlés diagrammját az **1., ábra** mutatja. Láthatóan ez is pofonegyszerű, akárcsak a későbbiekben ismertetett vezérlőprogram.

Vigyázat, leállhat a motorunk!

Eddig nem beszéltem a motor néhány, fontos sajátságáról: a melegedésről, az indulási, valamint benntartási nyomatékról, illetve a maximális üzemidőről; a bekapcsolt tekercs melegedni fog. Minél tovább van áram alatt, illetve magasabb az áramerősség, természetesen annál jobban! Ezért az egyszerűbb elektronikák csak a léptetés ideje alatt adnak feszültséget a tekercsre, majd amikor az már biztosan elfordította a következő pozícióba a motort, akkor azonnal lekapcsolják. A léptető motort gyakran használják „szakaszos” üzemben is. Ekkor a léptetés alatt folyamatosan kap áramot valamelyik tekercs mindig, de ez a művelet 1..2 percnél nem tart tovább, ezután pedig lekapcsolják az áramot a tekercsekről, hagyják hűlni motorunkat. Ez elsőre meglepő, de tegyük hozzá: a motor nyomatéka is meglepő tud lenni! Ha a kézi léptetéskor meg akarjuk forgatni a tengelyt, akkor azt tapasztaljuk, hogy árammentes állapotban is kell egy érezhető erő az elforduláshoz, azonban ha bármely tekercsen áram van, a motor rendkívül erősen „ragaszkodik” a pillanatnyi pozíciójához. Látható tehát, hogy más az árammal átjárt, illetve az árammentes motor tartónyomatéka. Ha pl. egy robotba építünk steppert, akkor érdemes figyelni erre, hiszen a kikapcsolt motor esetén elforduló tengely miatt elveszíthetjük a pontos pozíciónkat! A problémára több megoldást találtak ki a szakemberek:

Az első, hogy egy féket építenek a tengelyre, ami akkor old, ha a motor áram alatt van, és akkor szorít, ha árammentes. Ez remek megoldás, csak a mechanika és a plusz elektromágnes miatt drága, körülményes, illetve karbantartást igényel..

A következő megoldás abból a megfigyelésből adódik, hogy az elektromágneses tér erőssége a távolság négyzetével arányosan csökken. Azaz egy relét, elektromágnest, vagy éppen a motorunkat mindig megindítani a nehezebb. Amikor már kezd befordulni a vasmag a helyére, akkor csökken a távolság, négyzetesen növekszik a vonzóerő is. A pozícióban tartáshoz tehát kisebb mágneses tér is elegendő lenne, mint a léptetéshez.
A kisebb mágneses teret úgy érik el, hogy a 12V-os tápfeszültséget 5V-ra csökkentik le, amikor a motor mozdulatlan, és 12V-ra emelik fel, amikor lépnie kell. Ez nagyon ügyes megoldás, mert egyszerű, olcsó, és megbízható. 5V-on a tekercsek nem melegszenek számottevően, de a benntartási nyomatékuk így is nagyobb, mint a 12V-on mért nyomaték megindításkor, azaz nem a benntartás a gyengébb láncszem, s ennyi már nekünk elegendő. Szintén egy okos kis „trükk”, hogy a motort induláskor egy 12V-ra feltöltött kondenzátor közbeiktatásával kötik a tápfeszültségre. Ekkor 12V tápfeszültség, plusz a kondenzátorban tárolt 12V feszültség miatt kezdetben 24V éri a tekercset. Ez kétszeresére megnöveli a kezdeti nyomatékot, vagyis a leggyengébb részét az üzemnek.
Amikor a kondenzátor töltése „elfogy”, egy dióda segítségével a feszültség beáll 12V-ra, s a motor így mozog tovább. Amikor pedig a forgatóüzem szünetel, a táp visszakapcsol 5V-ra.

Olyat is szokás csinálni, hogy a szögsebességtől (fordulatszámtól) teszik függővé a tekercseken áthaladó áramot. Belátható ugyanis, hogy a fordulatszám emelésével (minél gyorsabban szeretnénk lépegetni) egyre inkább csökken a nyomaték, azaz mindinkább hajlamossá válik „megcsúszni”, téveszteni a motorunk. Azonban nem mindig kell gyorsan menni, tehát átmenetileg kissé túlterhelhetjük a motorunkat, ha az átlagos terhelési határt nem lépjük át. Azután szokás áramgenerátorral vezérelni a motort, ami kezdetben akár 60V-os feszültséget is „rádurranthat” a tekercsekre, majd az elfordulás közben ezt veszi vissza a névleges értékre. Számos ilyen, illetve hasonló trükköcske létezik, ami még több erőt képes motorunkból kifacsarni. S mindez még semmi! Van egy igen érdekes, ún. -finomléptetéses -, precíziós üzemmód. Ez azt jelenti, hogy nem ki/bekapcsolt tekercsekkel dolgozunk, hanem folyamatosan, szinuszosan változó értékű feszültségekkel. Ekkor a feszültségek tetszőlegesen finom felbontásban állíthatók, azaz a motor tetszőleges, a fogacskák közötti pozíciót felvehet. Hiszen a fix lépések közei tetszőleges számra felbonthatóvá válnak! Félelmetes, amit így a léptetőmotorunkból kihozhatunk, de sajnos a vezérlő elektronika ára és bonyolultsága is gondolkodásra inti az embert: ilyet csak különlegesen indokolt esetben szoktunk alkalmazni.

E megoldások azonban már az ipari kategóriákba tartoznak, és nem valók kezdőnek. Ezért mi csupán egy rendkívül egyszerű, de ennek ellenére remekül működő, 12V-os vezérlőáramkörrel ismerkedünk meg a következő, folytatásban. Hasonlóan egyszerű kis tesztprogramunk is erre fog épülni. Azonban addig gyakoroljuk a motorok bemérését, illetve a beszerzést, hogy a következőkben csupán néhány apró alkatrészt kelljen összekötni, és máris futtatható legyen a motorvezérlő programunk.

1., ábra: a féllépéses üzemmód vezérlési módja

1.lépés	2.lépés	3.lépés	4.lépés	5.lépés	6.lépés	7.lépés	8.lépés	1.lépésfázis	2.lépésfázis	3.lépésfázis	További lépések ...
Első tekercs	Első tekercs és második tekercs	Második tekercs	Második tekercs és harmadik tekercs	Harmadik tekercs	Harmadik tekercs és negyedik tekercs	Negyedik tekercs	Negyedik tekercs és első tekercs	Első tekercs	Első tekercs és második tekercs	Második tekercs	S így tovább...

Az ábrát készítette: a szerző.

Kis Norbert
norbimagan@freemail.hu

SOA másként

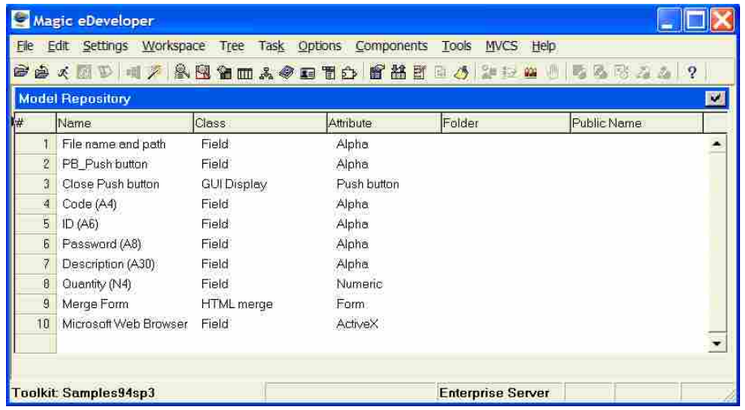
Bevezetés

Az alkalmazás fejlesztés technológiája nemcsak megújult, de jelentős változáson és fejlődésen is keresztülment. Az így kialakult új helyzetnek a fejlesztőeszköz gyártók nem igazán tudtak megfelelni, az egyébként is tapasztalható technológiai koncentrációt (nagyhalak megeszik a kishalakat) meghaladó mértékben egyszerűsödött le a termékek palettája.

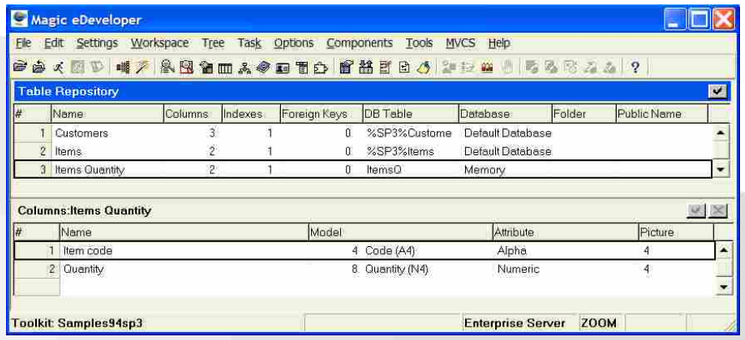
A technológia a kitüntetett szerepben megjelenő résztvevők (kliens/szerver) helyett a szolgáltatás alapú megközelítés felé mozdult el, melyben bárki lehet szolgáltató, és bárki lehet felhasználó. A lényeg a szabványos felületek kialakítása és elterjedése, ami a 'nagyok' által erőltetett saját technológia elfogadtatására irányult (Java, .NET). Ma a probléma egy független szabvány kialakulásával (web services) látszik megoldódni. Míg mindez a napi gyakorlat szintjén eluralkodik (értsd kizárólagossá válik) együtt kell élnünk, esetleg megfelelő technológiai háttérrel előnyt kell kovácsolnunk a hárompólusú világ együtt-működtetéséből.

Megoldásként kínálkozik a konkrét feladatok egyedi megoldása adhoc jelleggel, vagy olyan eszköz választása esetleg kiterjesztő, kiegészítő jelleggel ami képes az átjárhatóságot megteremteni, rutinszerű általános ha úgy tetszik 'iparszerű' módon. Egy ilyen eszköz az eDeveloper melynek bemutatására vállalkoztunk a cikksorozatban, nem elfelejtve vagy elhanyagolva az általánosan levonható következtetéseket eszköztől független tanulságokat. Mindezt gyakorlati módon megközelítve próbáljuk bemutatni, az eszköz iránti érdeklődés felkeltésének szándékával. Ugyanakkor gondolva az eDev használóinak népes hazai táborára is akik napi rutinként az adatbázis kezelő kliens/szerver alkalmazások mellett talán segítséget kaphatnak az architektúrális lehetőségek megismerésében.

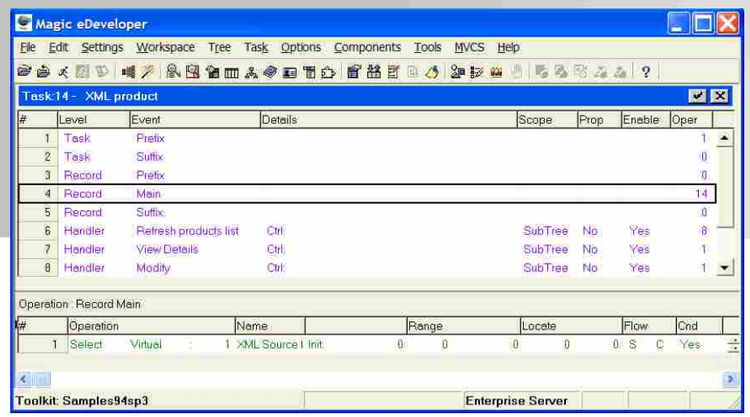
Az eDev technológia magában hordozza a környezet megváltozásához alkalmazkodás képességét ezért elkerülhetetlen néhány szót ejtenünk az eszköz alap tulajdonságairól is. Ez egy 1989 óta létező termék mely motor alapon (lásd Java, Framework) működik. Ezzel téve lehetővé a platform független alkalmazások fejlesztését (lásd Java). Ehhez párosult egy speciális irányultság az adatbázisa kezelő alkalmazásokra történő optimalizálással, ami a gateway technológia alapján adatbázis függetlenséggel párosult. Nem létezik és soha nem létezett Magic nevű adatbázis-kezelő, (az eDev korábbi verzióinak és magának a termék gyártójának is a neve) hanem egy adatbázis gyártó motorjával (Oracle, MsSql, Pervasive, Db2, stb) együtt alkotott működőképes rendszert. Az alkalmazás és a fejlesztő eszköz a folyamat szemlélet helyett vagy mellett kiemelten támaszkodik az adatkörnyezet vagy adatnézet kitüntetett szerepére. Fontos momentum bár a szolgáltatás alapú logika szempontjából csak közvetetten fontos, hogy mindehhez egy táblázatos programozási felület és egy ebből következő deklaratív jellegű fejlesztés társul. Mindez robosztus és skálázható megoldást lehetővé tevő termékekkel, többrétegű alkalmazás logikával párosult a kezdetektől.



1. ábra Minták vagy modellek



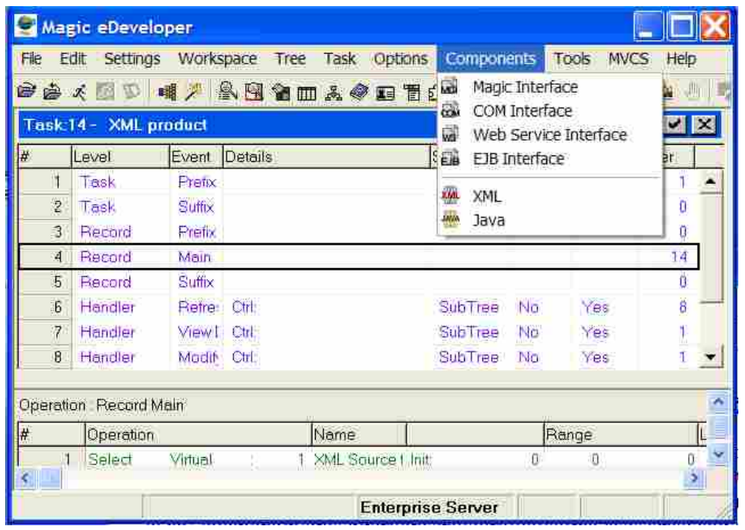
2. ábra Táblák



3. ábra Program szerkezet

A minta, tábla és program logika ábrája próbálja meg érzékeltetni az alkalmazás alap összetevőit illetve a táblázatos forma, deklaratív programozás felületét bemutatni. A harmadik ábra annak leszögezésére is alkalmat ad, hogy az eDev-et nagyon szigorúan struktúrált alkalmazás szerkezetként is felfoghassuk, kikényszerítve a jól átlátható, könnyen karbantartható, továbbfejleszthető belső felépítést. (A strukturáltság alkalmazása, program, taszk, műveleti csoportok, ezek eseményei és általában a műveletek sorrendjéből az utóbbi hármat megjelenítve.)

Visszatérve cikkünk tárgyát képező szolgáltatás alapú alkalmazásokhoz a 3. ábra Components legördülő menüjére kellene figyelmünket irányítani.



4. ábra Komponensek

Nem derül ki a menü pontjai alapján a mögöttük rejlő lehetőség mely szerint az eDev alkalmazás Java, .NET és web services környezetben is képes szolgáltató és felhasználó is lenni, de a felsorolás a technológiák találkozását talán sejteti. Első pontja (Magic interface) az önmagában működő, csak eDev alapon elkészített alkalmazások ilyen módon történő összekapcsolásának helye. Azon túl, hogy egy szolgáltatás-logika ismerhető meg ezen keresztül az alkalmazás struktúra sokfélesége is megismerhető ezen keresztül, alkalmazás szerver, köztes réteg technológia, üzenettovábbító alkalmazása, logika direkt összefűzése stb. A COM ként megjelenő funkció gyűjtemény a .NET világ kapcsolatát jelenti. Mindez persze nem az utolsó néhány verzió újdonsága, hosszú fejlesztés többfázisú érlelődésének az összegzése. Tetszőleges harmadik generációs nyelven írt modulok elérhetősége (CALL UDF), szabadon integrálható DLL modulok aktiválásának lehetősége, DDE és OLE technológiákkal is támogatva akár. A Web Service Interface az a belépési pont amin keresztül SOAP alapon működő szolgáltatások tehetők alkalmazásunk szolgáltatás vagy felhasználás szintű részévé. Ennek részletes megismerése a sorozat további részeiben kerül tárgyalásra. EJB (enterpise java bean) a Java-s kapcsolat. Amit a .NET-re előbb elmondtunk, mindaz talán kevesebb előzménnyel a Java-ra is igaz. Részleteket lásd később. A vonal mint egy logikai váltás kicsit más területekre kalauzol a menüpontok révén. Az XML mint az általánosan elfogadott adatcsere formátum kitüntetett szerepet játszik a web szolgáltatásokban is (szokták egyszerűsítve a SOAP=HTTP+XML definíciót használni), de az adatbázis kezelő alkalmazások egyéb kapcsolatainál is újat és többet tud nyújtani a szokásos formáknál. Ennek mindenre kiterjedő generátorát (xcg=xml component generator) találjuk ebben a pontban. Táblákká alakításon át a sémák (xsd file) kezelésén keresztül egészen a karbantartó programok elkészítéséig teljes körű funkcionalitás gyűjtemény kelthető életre ebben a pontban. A záró Java menüpont hasonló funkció gazdagságot kínál fel a fejlesztőnek Java-s környezetben illetve a már felvillantott EJB belépési pontot egészíti ki további lehetőségekkel.

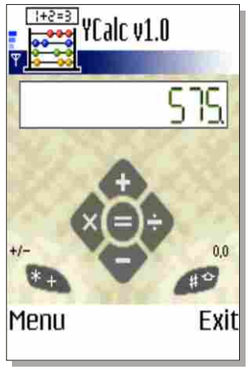
Ezeket fogjuk a sorozat további részeiben ismertetni vizsgálva a szolgáltatás és azok használatának lehetőségeit bemutatva a technológiák találkozását az eDev keretein belül.

Nádasy Gábor -
gabor_nadasy@magicsoftware.com
Magic (Onyx) Magyarország Kft.
Technológiai igazgató

A cikksorozat oktatási változata megtalálható a Széchenyi Egyetem és a BMF anyagai között, letölthető a http://winnie.nik.bmf.hu hallgatói szerver Magic tantárgyi webhelyekről.

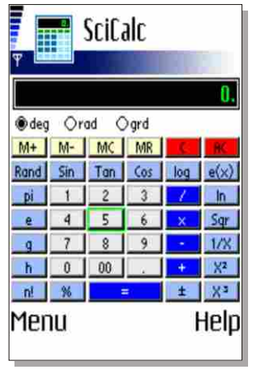
Használjuk a telefonunkat

Számológép helyett!



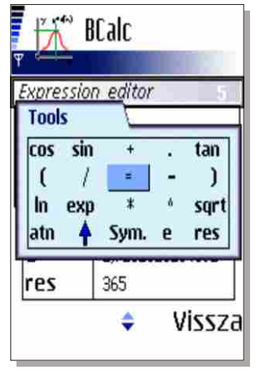
Mobiltelefonunk sokkal komolyabb hardverrel rendelkezik, mint egy számológép, így joggal vetődhet fel a kérdés, hogy találhatunk-e megfelelő szoftvert ilyen célra. Amennyiben symbian operációs rendszerrel felszerelt telefonunk van, akkor több program is a segítségünkre siet, én most ezek közül hármat fogok bemutatni.

Először is szeretném megemlíteni az YCalc nevű programot, amely, bár tudása igen szerény, rendkívül gyorsan és könnyen használható. Ezzel a programmal csupán a négy alapművelet végezhető el, ám mindenképpen ajánlott a beépített számológép helyett ezt használnunk praktikussága végett.

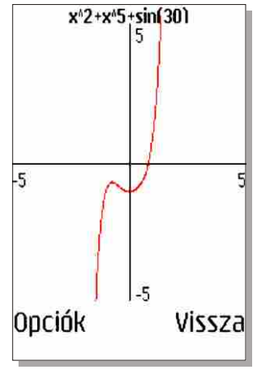


Amennyiben egy komolyabb tudású, ám egyszerűen kezelhető számológépre lenne szükségünk, akkor rendelkezésre áll a SciCalc, amely már trigonometrikus- és exponenciális függvényeket is kezel, elvégzi a gyökvonást, valamint a négyzetre és köbre emelést is. Kezelőfelülete jól átlátható, de sajnos használata lassú, mivel a navigációs gombbal kell elérnünk minden funkciót, csupán a számokat üthetjük be közvetlenül, a telefon billentyűzetének segítségével. Sajnálatos módon a „c” billentyűvel sem törölhetünk, el kell lépkednünk az ötirányú navigomb segítségével a c gombhoz a kijelzőn.

A tesztelt programok közül minden kétséget kizáróan a BCalc a legkomolyabb program, amely már valóban kiválthatja akár tudományos számológépeinket, sőt, egyes funkciókban meg is előzi azokat. Bár kezelőfelülete első látásra talán egy kissé barátságtalannak tűnhet, pár perc használat után megszeretjük hatalmas tudása miatt. A program alapból ismeri a pi és e számokat, de további (pl. fizikai) konstansokat is megtaníthatunk neki. A számológéppel bonyolultabb számolások is végezhetők, hiszen a zárójeleket is kezeli. A hatványozás, exponenciális- és trigonometria függvények illetve a gyökvonás sem okoznak természetesen problémát. Az opciók -> settings menüpontban beállíthatjuk a bevitel módját (numerikus, vagy alfanumerikus). A BCalc



függvényeket is tud ábrázolni, amire bizony csak kevés számológép képes, ráadásul itt a rendelkezésünkre áll egy nagy felbontású, színes kiejlző. A függvény beírásakor célszerű az alfanumerikus bevitelt választani, különben nehéz lesz beütni a paramétert. Miután függvényünket begépeltük az Opciók -> Calculator -> Plot graph menüpont alatt tekinthetjük meg az ábrázolt függvényt, amelyet természetesen nagyíthatunk és kicsinyíthetünk is.



Összeségében elmondható, hogy az átlagos felhasználóknak elegendő az első két program, programozók, fizikusok és matematikusok számára pedig a BCalc és az YCalc párosítás lehet a nyerő.

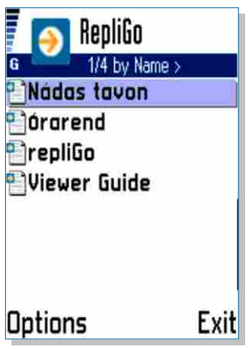
Dózsa Martin
e-mail: mdozsa@vnet.hu

RepliGO Dokumentumok az S60on

Mai világunkban egyre elterjedtebbek a személyi számítógépek. Ki ne dolgozott volna már valamilyen szövegszerkesztő, táblázatkezelő, vagy egyéb Microsoft Office programon. Elképzelhető, de nem hiszem, hogy csak az én fejemben fordult volna meg, hogy kedvenc telefonkészülékem képernyőjén belenézhessek még az elkövetkező óra, vagy épp tárgyalás anyagába, anélkül, hogy laptopra és az ezzel együtt járó procedúrára legyek kényszerülve. Ehhez kényelmes és kellően apró megoldást nyújt a Cerience Co. alkalmazása, mellyel a számítógépünkön átkonvertált file-okat tekinthetjük meg Series 60 telefonunkon. De hogyan is működik?

A programcsomag 2 részből épül fel, egy, a PCn futó konvertáló, illetve olvasó elemből, mellyel a kellő kiterjesztésű file-ok elkészíthetőek és utólag ellenőrizhetőek is, és egy, a maroktelefonunk számára készült modulból. Fontos tudnivaló, hogy a programok nem képesek a dokumentumok tartalmát módosítani, csak azokat megjeleníteni.

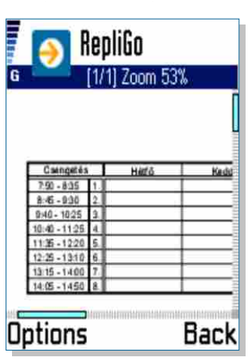
Nézzük előbb a számítógépre írt részt: Installláskor beépülő elemek sokaságából kell kiválasztanunk a számunkra fontosakat: A Microsoft Office és Explorer Intergration-t nagyon ajánlom, mert így egyszerűbb módon alakíthatóak majd az adatok, de érdemes minden elemet alkalmazni, már csak a kényelem érdekében is. PCs installáció közben amennyiben aktív és párosított készülékünk is van, a telefonra szánt applikáció is telepíthető innen, de ezt akár későbbre is halaszthatjuk. A program mintegy 6-8 MB helyet foglal el merevlemezünkből, de ezért cserébe bárhonnan könnyedén alakíthatjuk hordozhatóvá virtuális iratainkat. A repliGO desktop egy egyszerű help-nek is nevezhető. Angolul megtaláljuk benne, hogyan is használjuk a programot. Még linkeket és egy demo-t is tartalmaz a kezelésről. A repliGo viewer a már átkonvertált és feltöltésre kész adatok visszaellenőrzésére szolgál. Az igazi értékeket azonban abban találjuk, hogy beépül programjainkba. A menü-sorban RepliGO pont jelzi az új jövevény jelenlétét. Nincs más dolgunk, mint belépni a Word, Excel, Internet Explorer illetve Power Point-unkba, megnyitni a konvertálandó file-t és vagy a bal oldali aprócska nyílra, vagy a menüben a „Convert File” parancsra kattintani. (Explorer-ben csak az eszköztárban elhelyezkedő jelzés alkalmas erre) Az ekkor felugró ablakban állíthatjuk be az új nevet, a mentés helyét, és pár extrát, mint például esetleges biztonsági másolatot, amennyiben egyből e-mailben továbbítjuk adatainkat a program segítségével. Ebben azt hiszem semmi ördöngősség nincs, viszont mit tegyünk abban az esetben, ha hőn áhított adatunk nem ezen programok valamelyikével készült? Nos erre is megtaláljuk a megoldást, csak az adott file saját szerkesztőjébe lépve a nyomtatás parancs után aktuális nyomtatónk helyett a repliGO-t kell választanunk. Az alkalmazás ugyanis beépülve a rendszerbe, mintegy „virtuális-nyomtatóként” funkcionál, így alakítva át az adatokat. Gyakorlatilag aprócska fényképet készít, amit később könnyedén visszaolvas. Mivel erősen tömörített adatokat szolgáltat végeredményként, a helytakarékosság jegyében a módosított képek nem igazán élvezhetőek. Ez html oldalak átalakításánál okozhat gondot, ami esetében az 1.2es verzió még küszködik apróbb nehézségekkel. Ha ugyanis az oldal háttere egy nagy kép, azt furcsa mód darabolja, és a képek minősége is gyatra, de nézzük el, hogy mindez az apró méretek miatt olyan amilyen. (Egy 33kb-os Word dokumentum, tömörítés után 6kb-ra zsugorodott.) Úgy érzem az elmúlt pár sor elég világosan jellemzi a PCs alkalmazás feladatát, már csak a feltöltésről ejtenék pár szót. Adataink Bluetooth kapcsolat, Infraport, vagy MMC kártyaolvasó által felmásolva



érhetőek majd el ezután készülékünkön. Alapértelmezésként C illetve E meghajtó Documents mappájából olvassa be telefonunk a *.rgo kiterjesztésű dokumentumokat. Célszerű tehát átküldéskor ide menteni. Ha a feltöltéssel elkészültünk már csak a telefonon futó alkalmazás megismerése vár ránk, lépjünk hát be.

Először is a telepítés. Összesen mintegy 203 kb-ot emészt el a tárhelyből a kis nézegető modul. Elindítva puritán üdvözlés, semmi csicsa. Lényegre törően, komolyan fogadnak minket a régebbről már jól ismert file-ok. Mit is kezdhetünk velük? Lássuk. Előbb az alapképernyő menüpontjait zongorázzuk végig. Innen ugyanis egyből továbbíthatjuk is e-mail, Bluetooth, vagy Infra segítségével, törölhetjük értelemszerűen a „delete” paranccsal és különféle (név, dátum, méret) sorrendbe is rendezhetjük az adatokat a „sort”-ra kattintva. Ha valamelyik iratot megnyitjuk, maximum apró pixeleket láthatunk a kijelzőn, de itt jön a trükk, mely miatt kellőképp elnyerte tetszésem az alkalmazás. A számbillentyűk segítségével ugyanis pofonegyszerűen navigálhatunk a dokumentumban. Az 1-es gomb lenyomásával az egész oldal válik újra láthatóvá, míg a 2-es egy nagyon érdekes funkciót takar. A szöveget folyamatosan olvashatóvá teszi, 5 féle méret megadása mellett. A 3-as gomb a zoom-ot aktiválja, aminek erősségét a 4-es és 5-ös gombokkal módosíthatjuk. A 6-os gomb ismét küldésre szolgál a már megismert módokon. A 7-es és 8-as billentyűk a későbbi verziókban szöveg, vagy egyéb elem keresésére szolgálnak, de itt ez még nem használható funkció,

nem úgy, mint a 0-ás gomb, amely az oldalak közti navigációért felelős. A *-gomb a teljes képernyős nézet, míg a # az elforgatás aktiválására szolgál. Ezzel a dokumentum-nézegetés végére is értünk, még az alapképernyőn van pár be nem mutatott apróság. Ha a file-ok listáját nézzük az options menüben a settings paranccsal az adatok keresésének helyét (alapértelmezésként \Documents), vagy épp az üdvözlő képernyőt „szabhatjuk testre”. A listában bárhol is járjunk, betűnként, vagy egész névre is rákereshetünk. Ha pedig az első helyre szeretnénk lépni, nyomjuk meg a joy-t balra, a végére lépéshez pedig jobbra. Nos be is fejeztük a RepliGO v1.2 bemutatását, a végére csak az újonnan megjelenő 2.0-s verzió legfőbb újítására szeretném felhívni a figyelmet. Itt már akár szövegre is rákereshetünk, ami megkönnyíti a tájékozódást, amennyiben a sokadik oldal valamelyikén tanyázik a nélkülözhetetlen információnk. Egyéb készülékekre is megtalálhatjuk a RepliGO-t. Például: PPC-re, PalmOS-ra, Microsoft Smartphone-ra és SonyEricsson okostelefonokra szánt verzió is létezik. Az ezekre írt változatok több extra funkciót is tartalmaznak, mint például a könyvjelzők, vagy a megjegyzések hozzáfűzése. A **www.cerience.com** oldalról mindenki letöltheti a számára legmegfelelőbb próbaverziót és amennyiben az beváltja a hozzáfűzött reményeket akár meg is vásárolhatjuk a teljes programot.



Seres Gábor
e-mail: seres.gabor@vodafone.hu

Szoftvervédelem

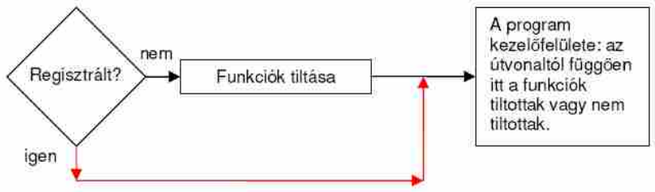
A kreatív programozó

Hol van az a határ, melyről már állítható, hogy egy szoftver védett? Tényezők, amik fölöttébb relatívvá teszik ezt a kérdést: az idő és a viszonyítási alap. Az idő teltével és a kutatások felhasználásával fejlődnek a védelmi mechanizmusok. Mivel egy lefordított programkódban, érdemben turkálni, már nem alapszintű tudást igénylő feladat, ezért egy nem védett szoftver is „védett”. Lássuk be, hogy kizárólag erre támaszkodni bőven nem elegendő.

A programvédelem alapmechanizmusa

Adott egy kereskedelmi szoftver, ahol n fizikailag beépített funkció a felhasználó számára közvetlenül nem elérhető. Ahhoz, hogy ezen *n* funkció közül, legalább egy aktivizálódni tudjon, meg kell vásárolni a terméket. A vásárlás után a felhasználó kap egy ún. regisztrációs kulcsot, amivel aktiválhatja a terméket. Sikeres aktiválás után elérhető lesz az összes olyan tiltott funkció, amiért a vásárló fizetett. Az, hogy az aktiválás milyen módon történik, (pl. regisztrációs szám, kulcsfájl, hardverkulcs…) az a szoftverfejlesztő cég döntése.

Leggyakoribb a regisztrációs szám, amely teljesen egyedi, bonyolult algoritmussal előállított kódsor és általában tartalmaz érvényességi időt, engedélyezett funkciókat és egyéb opcionális adatokat. Ez a **vásárlás-aktiválás** folyamat sok esetben nem így működik, azzal a különbséggel, hogy a vásárlás elmarad. Ez természetesen anyagi mínusz a vállalatnak, amit egyik cégtulajdonos sem szeret. Tény, hogy a vásárlás mindig elkerülhető úgy, hogy az aktiválás sikeres legyen. Tehát **bármely program kódja módosítható úgy, hogy azzal a program funkcionálisan teljes értékűvé váljon. Feltétel, hogy tartalmazzon n fizikailag beépített, de alapesetben nem elérhető funkciót, továbbá tartalmazza valamennyi funkcióhoz vezető feltételes útvonalat.**



A fenti sematikus ábra az útvonalakat illusztrálja. A programot a processzor **alacsony szintű utasítások sorozata**ként hajtja végre. Ezek az **utasítások szabadon módosíthatók** (jogilag nem feltétlen) mielőtt a processzor végrehajtana. Tehát a program útvonali (konstansai, de bármely bit-e) megváltoztathatók. Amit a program feltörők ellen lehet tenni védekezésképp, az a kreatív ötletek: több ellenőrzéspont, ciklikus ellenőrzések, integritás ellenőrzések (…). A védelem kialakítása sokszor a strukturált programozást felborítja. Természetesen a szoftverfejlesztők számára minden ugyanolyan áttekinthető maradhat. A cél mindig: a cracker-ek összezavarása. Igen, még egy fontos közölnivaló: a „cracker” szóról megoszlanak a vélemények; tulajdonképpen én mindig program feltörőt értek alatta. Olyan személyt, aki saját tudását alkalmazva képes egy **szoftver terméket funkcionálisan befolyásolni (kód átírása, kód hozzácsatolása, kulcs generálása) úgy, hogy azzal saját magának vagy/és környezetének kedvezzen.**

Licenc menedzselők lustaság, pénz, tudás és időhiány

Mint szinte minden problémára, a programvédelemre is léteznek **univerzális módszerek**. Gondolom senkinek nem kell bemutatni az univerzális festéket, ragasztót. Noha a tendencia görbéje szigorúan emelkedik; „ezek” mindig elmaradnak a speciális megoldásoktól. De legyen **más nézőpontja** a programozónak! Mikor egy licenc generátor alá kerül a program valójában a következőt teszi a programozó, mint felhasználó:

rábízza egy programra saját alkalmazása védelmi rendszerét,

valójában **fogalma sincs, hogy mi történik** a háttérben, milyen mechanizmussal,

a program feltörők szeretnek „nagy halakra” vadászni, ezért a **cracker-eket fokozottan ingerli** ez a fajta univerzális rendszer,

onnan kezdve, hogy egy alkalmazást sikerült feltörni az adott verziójú, licenc menedzselt programok halmazából elméletileg az **összes többi program feltörhető** ugyanazon (de max. paraméterekben különböző) módszerrel,

az univerzális védelmi mechanizmusok leírásának (cracking tutorials) **megjelenése több olvasótábort** tud magáénak, ezért több ember ismeri meg a védelmi rendszer működését rendszerközeli szinten.

Minden licenc menedzselő, a szoftvert hivatott védeni. Beépítésük a programba változó. Vannak, amikhez abszolút nem szükséges **semmilyen programozói tudás**, csupán az adott PE (Portable Executable) fájlt kell megadni és beállítani az esetleges paramétereket. Shareware programok egy részhalmaza e technológiával védett. **Programozói tudást igénylő** csoportba tartoznak azok a menedzserek, amelyekhez fájlokat és dokumentációt kap a programozó. A dokumentáció tartalmazza a helyes beépítést a forráskódba, és a menedzselést. A védelmi rendszer motorja előre megírt; a programozó feladata a komponensek beépítése és a szükséges környezet beállítása. Ez a technológia általában előnyösebb az előbbinél, mert több beleszólása van a szoftverfejlesztőnek a védelem kialakításában.

Korlátok, fizikai sebezhetőségek

Ahhoz, hogy egy szoftverfejlesztő cég el tudja adni programját, nem elég beszélni róla a felhasználóknak, hanem be is kell azt mutatni nekik. Tegyük fel, hogy az egyik felhasználó letöltött egy kereskedelmi programot, tetszik neki, és használja is. Miért vásárolja meg, ha így is tudja használni? Azért, mert ő egy shareware (demo…) verziót töltött le, ami korlátozottan használható. Például, **30 napra korlátozódik a használat, vagy nem lehet benne menteni**, de a fejlesztőtől függ bármely limitáció meghatározása. Tegyük fel, hogy az egyik programot öt programindítás választja el, hogy funkcionálisan teljes legyen. Tehát ha a felhasználó már hatodik alkalommal szeretne indítani a programját, nem indul. Ekkor a cracker megkeresi azt a részt a program kódjában, ahol ez a feltételes elágazás van. Ez kb. a következőképp néz ki:

```
cmp     eax,00000005    összehasonlítja az eax regisztert 5-tel

ja      KILEP           ha nagyobb, ugrik a KILEP címkére

jmp     TOVABB          a program tovább fut
```

A cracker általában kétféleképpen közelítheti meg a törést. Visszafordítja a teljes programot assembly nyelvre, vagy nyomkövetővel (hibakeresővel) debuggolja. Az assembly nyelv, API hívások és a rendszerközeli ismeretek elengedhetetlenül szükségesek, mind a töréshez, mind a hatékony programvédelem kialakításához. Visszatérve a fenti utasításssorozathoz, a programkód átírásával megszűnik ez a limit. Ettől a program funkcionálisan nem lehet regisztrált, viszont funkcionálisan teljes értékűvé válik (amennyiben ez volt az egyetlen limitáció és nincs speciális védelem: pl. integritás ellenőrzés).

Regisztráció, aktiválás inaktív korlátok

Adott kereskedelmi szoftver aktiválása történhet regisztrációs szám megadásával, kulcsfájl tallózásával, de ennek módját a szoftverfejlesztő határozza meg. Biztonsági szempontból egy nagyon kritikus pontról van szó. A cracker innen kiindulva tudja megtalálni a regisztrációs szám előállításának algoritmusát, vagy megtalálni azt a pontot, amit átírva regisztráltként indul a program. Miért pont innen indul ki? Mert általában ezen a ponton van a regisztráltság ellenőrzése. A programok több mint 50%-ában ez a hasonlóképp néz ki:

```
call    REG_ELLENORZES      szubrutin, a regisztráció ellenorzése,reg. szám

cmp     eax,00000001        ha eax-ba 1 kerül, akkor regisztrált

je      REGISZTRALT         és regisztráltként folytatódik

jmp     NEM_REGISZTRALT     nem regisztráltként folytatódik.
```

Látható, hogy a REG_ELLENORZES szubrutin-hívásban ellenőrzi a program a regisztrációs számot. Feltételezhető, hogy ez a szubrutin induláskor is meghívódik, hogy ellenőrizzen. Ezen ellenőrzések utáni ugrások átírásával, a program funkcionálisan regisztrált lehet.

***

Remélem, hogy Kedves Olvasó elé tudtam tárni, valójában mi a szoftvervédelem. Ez kinek mennyire fontos, azt mindenki saját maga dönti el. Viszont, ami a cikk folytatására késztet, az a szakirodalom hiánya a hazai és nemzetközi piacon egyaránt. A folytatásban ötleteket írok le, melyek alkalmazásával a szoftver biztonsága növelhető (esetleg tippem újabb tippet szül) az internetes kalózokkal szemben.

Suszter Attila **suszter@freemail.hu**

OpenGL.HU XI. Rész

glInformation() Bevezető

A cikksorozatnak immáron a tizenegyedik részét tarthatják a kezükben az olvasók, akiktől egyre több levelet kapok, bennük kérdéseket, ötleteket, felvetéseket és nem utolsósorban témákat, amelyek a cikksorozat lényegét adják. A továbbiakban is buzdítok mindenkit, hogy jelezze, ha egy adott dologra kíváncsi. Ettől a számtól kezdve az OpenGL.Hu szerkezete egy kicsit át fog alakulni úgy, hogy öt részre bontom: lesz egy bevezető, aktuális információkkal, majd egy olyan rész, ahol leírom az aktuálisan használt szoftvereszközök listáját, megelőzve ezzel néhány kérdést. Ezek után következik egy, az olvasók által kért téma, s rögtön utána az, amit én választottam erre az alkalomra. Végül a zárszó és előretekintés fejezi be a cikket. A részek címei a már itt látható glInformation(), glToolBox, glRequest(), glArticle() és glEnd() lettek, nem kis szakmai ártalommal átitatott névválasztási rituálé után

glToolBox() Ami kell...

Fordító:

- BloodShed DevC++ v4.9.8.0

Library:

- GLAux,
- GLUT v3.7.6,
- ExtGLGen

glRequest() Mindent az olvasóért!

Erre a hónapra a kért témák közül olyat választottam, ami a legutóbbi szám „ködös” részére épült. Meglepően sok levelet kaptam, melyben azt kérdezték, hogy ezen az egyszerű köd-technikán kívül van-e más megoldás is, hiszen vannak esetek, amikor nem biztos, hogy a ködnek minden irányban ugyanúgy kell látszódnia. Nos, igen, van ilyen technika, s nem is egy bonyolult dologról van szó: ez a volumetrikus köd, eredeti, hangzatos nevén: volumetric fog. A dolog lényege, meglepő módon pont az, ami nekünk itt kell: egy olyan köd, amelyet sokkal jobban lehet kontrollálni, mint a korábban látott distance fog-ot. Egészen pontosan, a volumetric fog vertexekhez köthető, ugyanúgy, ahogyan egy textúra. Tehát egy vertexek által határolt teret tudunk vele meglepően egyszerűen kitölteni, mint például egy sötét szakadék vagy liftakna, aminek nem látszik az alja. A szemfüleseknek egyből beugorhat, milyen jól megoldható ez egy fekete ködde, amelyik csak azon a részen létezik, s a szakadék alján áthatolhatatlanul sűrű, feljebb haladva viszont ritkul.

Tehát, akkor most erre fogunk egy példát adni! Hogy mire is lesz szükségünk? A szokásos függvényeken és egyebeken kívül használni fogunk egy EXT_fog_coord nevű estension-t, amit remélhetőleg minden gond nélkül be fogunk tudni tölteni, ugyanis nem egy vadonatúj darab. Korábban már volt szó az extension-ök elegáns betöltéséről a OglExt library segítségével, így most is ezt kellene segítségül hívnunk. Igen, kellene. De sajnos ez esetben úgy tűnt, ez a lib csődöt mondott, így a továbbiakban mellőzni is fogjuk a használatát. A helyére természetesen kellett valami, hiszen valljuk be, senkinek sincs kedve pointereket gyártogatni az egyes extension-ökhöz. Az új „játékszer” glExtGen névre hallgat. Egy kis programról van szó, amely az SGI oldaláról letölthető glext.h fileból a videokártyánk tulajdonságait figyelembe véve egy headert és egy cpp filet generál, amelyet a programunkba illesztve gond nélkül érjük el az extension-öket.

Na, akkor lássuk most a lényeget: a kódot. Egyetlen vertex definiálását emelném csak ki, hiszen a dolog egyszerű elven alapul: a volumetrikus ködöt vertexekhez köti, pontosan, mint a textúrákat. Itt azonban koordináta helyett azt kell megadnunk, hogy az adott pontban milyen sűrű a köd. Természetesen a nem megadott pontokban a sűrűséget az OpenGL interpolálja, minél simább átmenetet képezve az egyes pontok között. Egyetlen pontra hasonló sorokat kell látnunk annak megadásakor:

```
glFogCoordf( 0.0f);
glTexCoord2f(0.0f, 0.0f);
glVertex3f(-.5f, -.5f,  -1.0f);
```

A glFogCoordf(float) függvény adja meg a sűrűséget az adott pontban, 0.0f és 1.0f közötti értékkel. Természetesen lehet más (pl. Normal) koordinátákat beállító függvényeket is használni, a FogCoord nem interferál velük. Sajnos, mint később látni fogjuk, vannak helyzetek, amikor nem tudjuk használni, de ettől függetlenül igen hasznos és kellemes látványt nyujtó kis technikáról van szó. Ha a koordinátákat beállítottuk, akkor azt kell megadni, hogy milyen messzire látszik el a köd és hol kezdődik, hasonlóan, mint a 'rendes' ködnél. Nos, itt a dolog annyival egyszerűbb, hogy a ködöt vertexek fogják körbe, így ezzel a távolsággal nem sok tennivalónk van, a legtöbb esetben a következő beállítás bőven megteszi:

```
glFogf(GL_FOG_START, 1.0f);
glFogf(GL_FOG_END, 0.0f);
```

Mióta használtam a volumetric fog-ot, ez a beállítás még mindig megtette, de persze nyugodtan lehet kísérletezni. A legrosszabb esetben nem jelenik meg köd...

Még egyetlen lépés van hátra, hogy teljesen működőképes volumetric fog boldog gazdái legyünk: meg kell mondani az OpenGL-nek, hogy a megadott koordináták alapján számolja a ködöt, effektíve hozzákötni ezzel azt az objektumhoz. Erre a célra az alábbi utasítást használhatjuk:

```
GlFogi(GL_FOG_COORDINATE_SOURCE_EXT,GL_FOG_COORDINATE_EXT);
```

Maga a függvény már talán nem ismeretlen: integer értéket állítunk be vele egy OpenGL rendszerparaméternek. Mint a példakódból is látszik, természetesen az összes eddigi ködbeállítást (típus, stb.) most is lehet, sőt kell alkalmazni.

glArticle() És lőn világosság!

Ez a téma egy kicsit több szempontból is érdekes lesz. Először is azért, mert igen fontos része az OpenGL-nek, másrészről azért, mert amennyire csak lehet, kerülik a fejlesztők a használatát, mert igen erőforrásigényes és mindemellett igen behatárolt feltételek között működő dologról van szó. Ez a dolog nem más, mint a dinamikus világítás (dynamic lighting, vagy esetünkben konkrétabban, a per-face dynamic lighting). A dinamikus világítás a már korábban tárgyalt shadow-mapping (vagy light light-mapping, ahogy tetszik) technikával ellentétben nem stacionárius fényeket és árnyékokat hoz léte, hanem nagyon is élőket. Itt olyasmire gondolok, mint a kiszáradt kútba dobott égő fáklya vibráló, ugráló fénye és az általa vetett árnyékok. Ez az ideális, mármár közhelyszámba menő példa egyenlőre még igen messze van tőlünk, de elég lesz egyelőre megismerni a dinamikus világítás alapjait. Az „alapok” szó apropóján megjegyezném, hogy a megvilágítás matematikáját és kevésbé a programozáshoz tartozó szabályait most nem szándékozom tárgyalni, mert ebben a témában bőven áll rendelkezésre egyetemi jegyzet és könyv, vagy cikk a világhálón és a könyvtárakban, boltokban.

A lehető legjobban a lényegre térve elmondhatjuk, hogy a legegyszerűbb világítási módszer, a per-face dynamic lighting gyakorlatilag az általános iskolai fizikaóra tananyagában a témáról fellelhető információk egy-az-egyben törénő számítógépes adaptációja. A lényege, hogy egy fényforrást hozunk létre valahol, amely megvilágítja az n db face-ből álló objektumunkat. Ha egy face egy kicsit meg van világítva, az jelen esetben azt jelenti, hogy az egész face meg van világítva: vagy világos a face vagy nem. Ennyire egyszerű. Nos, a fény definiálásához meg kell adnunk pár paramétert, mint például a fény színe:

```
GLfloat LightAmbient[]= {1.0f, 1.0f, 1.0f, 1.0f };
GLfloat LightDiffuse[]= {0.0f, 1.0f, 0.0f, 1.0f };
```

Már látom, hogy sokan vakargatják a fejüket, mi ez a két tömb. Nos, mindkettő standard RGBA formátumú adat, a fény színét adják meg. Hogy miért is itt van két darab szín definiálva? A válasz egyszerű: minden fény világítja a környezetét és a kiszemelt tárgyat is. Az első szín azt mutatja, milyen színnel lesz megvilágítva a környezet (úgy kell elképzelni, mint egy nagy fénycső „mindenhová eljutó”, egyenletes fényét!). Ez a szórt fény. A második az a direkt módon a tárgyakra vetülő fény, igazából ez látszik rajtuk meg jobban. Ezzel kapcsolatban megint csak kísérletezésre tudok majd bátorítani mindenkit, illetve arra, hogy írjon e-mailt, vagy kérdezzen bátran.

Ha a fény színével megvolnánk, jöjjön a megvilágítást elősegítő fény-objektum pozíciója, mint szükséges beállítás:

```
GLfloat LightPosition[]= {0.0f, 0.0f, -1.0f, 1.0f };
```

A pozíció itt az ún., homogén-koordinátás megadásban szerepel, aminek az első három tagja a szokásos X,Y,Z érték, a negyediket pedig hagyjuk most 1.0f-re állítva, hacsak nem akarunk egy hosszas „teológiai” elmélkedésbe belekezdeni a homogén-koordináták mibenlétével kapcsolatban.

Ha már ezeket az értékeket ilyen lelkesen definiáltuk, a következő kis blokk megmutatja, hogyan hozzuk őket az OpenGL tudomására.

```
GlLightfv(GL_LIGHT1, GL_AMBIENT,LightAmbient);
glLightfv(GL_LIGHT1, GL_DIFFUSE,LightDiffuse);
glLightfv(GL_LIGHT1,GL_POSITION,LightPosition);
```

Amint láthatjuk, az első paraméter egy definiált konstans, amely megjelöli, melyik fény számára lépnek életbe a beállítások. (A fények számára még visszatérünk!) Az OpenGL alapból 8 fényt támogat, de a legtöbb esetben ez még soknak is bizonyult, tekintve, hogy a dinamikus fény mindig hatásosan emészti fel a futtató rendszer erőforrásait. A második paraméter azt jelenti, hogy az adott számú fény melyik paraméterét szeretnénk beállítani, a harmadikról pedig már beszéltünk korábban is.

Ez a kódrészlet a legtöbb leírásban valamilyen egyszer lefutó függvényben van elhelyezve, de ezzel egy jó néhány dinamikus megoldástól fosszuk meg magunkat: pl falon vibráló fáklya fény. Ha megbékéltünk a fenti pár utasítással és ezeknek a felhasználásával már elkezdtünk egy kis programot csinálni, csak egy dolog van hátra: engedélyezni kell az OpenGL-nek a fény használatát.

```
glEnable(GL_LIGHT1);
glEnable(GL_LIGHTING);
```

Tehát, összegezzük: kiválasztottuk, beállítottuk, aktiváltuk a fényt és..... semmi?! Na, igen, egy dolog még kimaradt. Nos, a fizikában amikor azt számoljuk, hogy egy felület hol van megvilágítva, használni szoktunk egy ún. beesési merőlegest, amelyhez a fénysugarak szögét viszonyítjuk. Ugyanaz a viselkedés, ugyanazok a szabályok megvannak az OpenGL-ben is, de itt a beesési merőlegest ún. felületnormálnak hívjuk. Legegyszerűbb úgy elképzelni, mint a felületre merőleges 1.0f egységnyi hosszú szakaszt vagy félegyenest, amelyiknek azt a végpontját adjuk meg, amelyik nem érintkezik a felülettel. (Tehát egy pontot, amely pont egy egységnyire van a felülettől, s a kettőt össze tudjuk kötni úgy, hogy a képzeletbeli összekötő vonal merőleges legyen a felületre.) Ennek a pontnak a megadása a szokásos XYZ koordinátákkal történik az alábbi módon:

```
glNormal3f( 0.0f, 0.0f, 1.0f);
```

Ezt minden felületre csupán egyetlen egyszer kell megadni, s a többi számítást majd az OpenGL végzi már el. Ha vertexenként adjuk meg a testen az összes face-t, akkor elég kiábrándító lehet ezeknek a pontoknak a kiszámítása, de szerencsére könnyen lehet olyan függvényt írni, amely a koordinátákból ezt kiszámolja nekünk. Ha minden igaz, most már látnunk kell a megvilágítás hatását feltéve, hogy nem számoltunk el valamit.

`glEnd()` Zárszó, előretekintés

Remélem mindenkinek elnyeri tetszését ez az új cikkszerkezet! Ezek után főleg várok, mindenféle kommentárt és jobbító tanácsot, valamint glRequest() témát. Szóval, kérdezzenek bátran és én megpróbálok válaszolni. A következő számig kellemes kódolást és hasznos időtöltést kívánok mindenkinek.

Merczel „MaNiAc” László laszlo.merczel@mailbox.hu

J2SE 1.5 Tiger JAVA

Bevezetés

2004 februárjában az Early Access keretében vált letölthetővé a J2SE következő verziója(**http://java. Sun. com/j2se /1.5.0/download.jsp**), melynek verziószáma 1.5.0 Beta 1. Ennek megjelenését hatalmas figyelem kísérte. Már régóta lehetett hallani pletykákat és hivatalos nyilatkozatokat arról, hogy mit is fog tartalmazni, milyen újításokat fog nyújtani.

A Java fejlesztők három fő platformot választhatnak, ha alkalmazás írásába szeretnének fogni. A J2SE egy komplett alap környezet kliens-szerver és asztali alkalmazások készítéséhez; a J2EE nagyvállalati környezetben használható technológiákat tartalmaz; míg a J2ME alkalmazási területe a mobil fejlesztés.

Most tehát az alap platformból jött ki új verzió, mely egyrészt jelentős nyelvi újdonságokat is tartalmaz, másrészt a már meglévő API-k bővültek, új API-k kerültek bele, sőt jelentős mennyiségű hibajavításon és optimalizáción is átesett. Az új verzió fejlesztésekor főleg négy területet tartottak szem előtt: a fejlesztés kényelmét, monitorozást és menedzselhetőséget, skálázhatóságot és teljesítményt, valamint XML és kliens oldali Web szolgáltatások támogatását.

A J2SE 1.5 jelenleg 15 új Java Specification Request - JSR specifikációt implementál, és közel 100 nagyobb frissítést tartalmaz, melyeket a Java Community Process (JCP) keretein belül fejlesztettek.
A négy fő fejlesztési terület közül az egyik a fejlesztés könnyítésére irányul (Ease of Development), melyet a következő nyelvi elemek és tulajdonságok hivatottak biztosítani: generikus típusok (JSR 14), metaadatok (JSR 175), automatikus be- és kicsomagolás, fejlettebb ciklusképzés, felsorolásos típus, statikus import (ez utóbbi négyet a JSR 201 tartalmazza), C típusú formázott adatbevitel és kiírás, változó számú paraméterlista, párhuzamos programozást segítő eszközök és az RMI interfész generálás elhagyása.
Alapértelmezésben a fordító 1.4-es Java kódot fordít, a **source** 1.5 kapcsolóval lehet megadni, hogy az 1.5 lehetőségeit kihasználjuk.

Metaadatok

A metaadatokat a Java forrás szövegében helyezhetjük el, osztályokhoz, interfészekhez, metódusokhoz és mezőkhöz kötve. Ezek olyan kiegészítő információk, melyeket feldolgozhat a Java fordító, bármilyen automatizált eszköz, de akár a bájtkódba is bekerülhet, és Java Reflection API-val elérhetővé válik.

A metaadat egy olyan eszköz, mely kiváló infrastruktúrát biztosíthat különböző kisegítő vagy deploy fájlok, kötelező interfészek generálásra, esetleg naplózás könnyítésére.

Generikus típus

A generikus típus bevezetését a Java fejlesztők már hosszú ideje várták. A generikusok hasonlítanak a C templatekre, de alapvető különbségek is vannak. A generikusok a típusoknál még egyel magasabb absztrakciós szinten helyezkednek el. Osztályok, interfészek és metódusok paraméterezhetők típusokkal. A fejlesztő legelőször a Collection API használata közben találkozhat a generikus típusokkal.

A Collection API osztályai képesek bármilyen osztályú objektumok kezelésére, de legtöbbször arra van szükségünk, hogy egy ilyen Collection API-hoz tartozó osztály csak egy osztályhoz tartozó objektumokat kezeljen. Abban az esetben, ha egy ilyen Collection API-hoz tartozó osztályba eltérő objektumokat teszünk bele, akkor az nem derül ki fordításidőben, csak futásidőben, és ilyenkor egy **ClassCastException** kivételt kaphatunk, ha egy elemet ki akarunk venni.

```
ArrayList list = new ArrayList();
list.add(0, new Integer(42));
int total = ((Integer)list.get(0)).intValue(); /* Itt váltódhat
ki a kivétel, ha nem Integer-ré típuskényszerítünk. */
```

Ugyanez egy olyan **ArrayList** osztállyal, mely a generikuson alapul:

```
ArrayList<Integer> list =  new ArrayList<Integer>();
list.add(0, new Integer(42));
int total = list.get(0).intValue();
```

Az **ArrayList** egy generikus interfész, ami egy típus paramétert vár.

Ebben az esetben, ha az **add** metódust egy olyan paraméterrel hívnánk meg, mely nem az **Integer** osztály egy példánya, már fordítási időben hibát kapnánk. Ezen kívül egy elem kivételekor sincs szükségünk típuskényszerítésre.

Tehát a generikusok használatával lehetőségünk van általános objektumokat kezelő osztályok készítésére, melynél megadható használatkor, hogy konkrétan milyen osztályú objektumokat kezeljen, növelve így a biztonságot és olvashatóságot, kiküszöbölve a felesleges típuskényszerítést, zárójeleket és az ideiglenes változókat. Ha olyan metódusokat írunk, melyeket mások is használnak, eddig csak JavaDoc megjegyzésben tudtuk megadni, hogy a paraméterként kapott Collection milyen osztályú objektumokat tartalmazzon. A generikusok használatával csak olyan Collectiont lehet átadni, melyre a megírt metódus számít. Ez a metódus szignatúrájából látszik, és a hiba már fordítási időben kiderül.

Az előző példa egy már implementált generikus típusú osztályt mutatott be, de lehetőség van arra, hogy saját magunk is készítsünk ilyent, alapul véve például a Collection API forráskódját.
Részlet:

```
interface List<E> {
  void add(E x);
  Iterator<E> iterator();
}

interface Iterator<E> {
  E next();
  boolean hasNext();
}
```

A relációs jelek között a formális paraméter típust adjuk meg. A típus paramétereket legtöbbször ott használjuk, ahol a szokványos típusokat is.

Amikor meghívjuk a generikus deklarációját, akkor a formális paraméter típust a futtató környezet kicseréli a hívásnál megadott típussal. Ez a csere a valóságban azonban nem jelenik meg, nem lesz új entitás sem forrás, sem bájtkód szinten, sem a lemezen vagy memóriában, csak egyszer lesz lefordítva, egyetlen egy class fájllá.
A formális paraméter típus neve legyen minél rövidebb, lehetőleg egy karakterből álló, és első karaktere legyen nagybetű. Ahogy a példa is mutatja a Collection API formális paraméter típusának neve E.

Fontos tulajdonsága a generikusoknak, hogy ha egy generikusnak egy paramétert átadunk, és ugyanazon generikusnak ezen paraméter egy altípusát (alosztály vagy alinterfész), akkor az altípus reláció nem marad meg a generikusoknál. Például:

```
List<String> ls = new ArrayList<String>();
List<Object> lo = ls;
```

Az ls nem altípusa az lo-nak, így a fordító a második sorra hibát fog jelezni. Hiszen nézzük a következő példát:

```
lo.add(new Object());
String s = ls.get(0);
```

Ebben az esetben a második sor egy **ClassCastException** hibát dobna, így megszűnne a generikusok típus biztonsága.

Ilyen módon tehát az sem lehetséges, hogy olyan metódust írjunk, mely egy **Collection<Object>** típust kap paraméterként, és kezeli az összes más paraméterrel szereplő generikust, hiszen a **Collection<Object>** nem „őse” például a **Collection<String>** generikusnak. E problémára találták ki a wildcard típust, így lehet például egy „ismeretlen típusú Collectiont” paraméterként kapó metódust deklarálni:

```
void printCollection(Collection<?> c) {
  for (Object e : c) {
    System.out.println(e);
  }
}
```

Az előzőekből kifolyólag a következő kódrészlet második sora szintén fordítási hibát adna, és ez a bizonyíték arra, hogy a típus biztonság garantált:

```
Collection<?> c = new ArrayList<String>;
c.add(new Object());
```

Az **add** paraméterként egy E típust várna, de mivel a c deklarációja ezt a típust nem határozza meg, ezért a fordító nem tudja eldönteni, hogy az **Object** osztályú példány megfelelő-e a biztonság szempontjából.

A **get** metódust viszont tudjuk használni, hiszen az mindenképpen egy Object osztály vagy leszármazottjának egy példányát fogja visszaadni, hiszen minden Java osztálynak az **Object** osztály az őse.

Az osztályok öröklődését azonban mégis felhasználhatjuk a generikusoknál, méghozzá a következőképpen:

```
public void drawAll(List<? extends Shape> shapes) {... }
```

Ebben az esetben viszont a **drawAll** metódusnak paraméterként adhatunk olyan **List**-et is, melynek aktuális paraméter típusa a **Shape** osztály valamely leszármazottja. Ennek a deklarációnak a neve korlátozott wildcard (bounded wildcard), ahol a paraméter típusa szintén ismeretlen, csak annyit tudunk róla, hogy a **Shape** osztály vagy annak valamely leszármazottja. Ezt úgy mondjuk, hogy a **Shape** a wildcard felső korlátja (upper bound). A következő sor hibás lenne az előzőleg definiált metódusban:

```
shapes.add(0, new Rectangle());
```

A **Map** is genrikus típus, melynek két paraméter típusa van, név konvenció szerint K és V, azaz „Key” és „Value”.
Az előbbiek miatt például a következő metódus sem megfelelő:

```
static void fromArrayToCollection(Object[] a,
Collection<?> c) {
  for (Object o : a) {
    c.add(o);
  }
}
```

A második sor fordítási hibát eredményez. A kívánt funkció generikus metódusokkal azonban mégis megoldható, ekkor a metódust is paraméterezhetjük különböző típusokkal:

```
<T> static void fromArrayToCollection(T[] a,
Collection<T> c) {
  for (T o : a) {
    c.add(o);
  }
}
```

A metódus hívásakor nem kell paraméter típust átadni, ehelyett a fordító automatikusan meghatározza az aktuális paraméter típusát véve alapul.

```
Object[] oa = new Object[100];
Collection<Object> co = new ArrayList<Object>;
fromArrayToCollection(oa, co);
```

A generikusoknak több felhasználási módja is lehetséges, melyekre a cikk már nem terjed ki, ehelyett csak megemlíti ezeket, a részletek kikeresése már az Olvasó feladata.

A generikusok fejlesztői külön figyelmet fordítottak arra, hogy a régi kódok, illetve az új, generikusokra épülő kódok jól megférjenek egymás mellett, és fokozatosan lehessen a régi kódot migrálni.

Érdemes megjegyezni, hogy egy generikus osztálya független a paraméterként megadott típustól, hiszen csak egy osztály tartozik minden generikushoz, bárhogy is paraméterezzük. Erre kell figyelni az instanceof operátor használatakor, valamint típuskényszerítésnél is.
Egy tömb elemeinek típusa nem lehet paraméterezett generikus, helyette wildcard tömböket lehet használni.

A **java.lang.Class** osztály is generikus.
Létezik wildcard alsó korláttal (lower bound), melynek kulcsszava a super.

Automatikus be- és kicsomagolás

A primitív típusok kezelése a Java nyelvben bizonyos esetekben igen körülményes. Mivel egy Collection-be csak az Object osztályból leszármazó osztályok objektumait lehet beilleszteni, a primitív típusokat át kellett konvertálni osztály szintű megfelelőikbe. A primitív típusok automatikus be- és kicsomagolása (autoboxing és auto-unboxing) lehetőséget ad arra, hogy ne kelljen a primitív típusokat osztály szintű megfelelőikbe (wrapper) becsomagolnunk, hanem ez automatikusan történjen. Az előző példát folytatva a kód a következőre egyszerűsödik:

```
ArrayList<Integer> list = new ArrayList<Integer>();
list.add(0, 42);
int total = list.get(0);
```

Abban az esetben, ha például egy **Hastable** példányból szeretnénk egy **Integer** típusú értéket automatikus kicsomagolással int változó értékének kinyerni, de a kapott **Integer** objektum **null** lenne, **NullPointerException** kivételt kapunk:

```
Map<String, Integer> map = new Hashtable<String,
Integer>();
map.put("one", 1);
int two = map.get("two"); // NullPointerException
```

Fejlettebb ciklusképzés

A fejlettebb ciklusképzés (foreach) lehetőséget biztosít az **Iterator** osztály körülményes használatának és a típuskényszerítésnek a kikerülésére a Collection API használata esetén.
A kód e tulajdonság használata nélkül:

```
ArrayList list = new ArrayList();
for (Iterator i = list.iterator(); i.hasNext();) {
  Integer value=(Integer)i.next();
}
```

Az új ciklus használatával:

```
ArrayList list = new ArrayList();
for (Object o: list) {
  Integer i = (Integer) o;
}
```

Generikus használatával a kód még egyszerűbbé válik:

```
ArrayList<Integer> list = new ArrayList<Integer>();
for (Integer i : list) {
}
```

A tömbökre is működik:

```
int[] array = new int[10];
for (int i: array) {
}
```

Felsorolásos típus

A felsorolásos típus is most jelent meg a Java nyelvben. Ez eddig is egy gyakran használt tervezési minta volt, de most nyelvi szinten elérhetővé vált a következő szintaktikával:

```
public enum StopLight {red, amber, green };
```

A felsorolásos típus előnyei az int típusú konstansokkal szemben:

- Fordítási időben történik az ellenőrzés;
- A konstansoknak egy névteret biztosít;
- Hibakeresés esetén a kiírás sokkal informatívabb jellegű;
- Hatékonyan alkalmazhatóak a **switch** kifejezésben;
- Mivel ez is objektum, használható a Collection API osztályaiban;
- Mivel ez alapvetően egy osztály, mezők és metódusok adhatók hozzá.

Egy összetettebb példa, mely az utolsó előnyre mutat példát:

```
public enum Coin {
   penny(1), nickel(5), dime(10), quarter(25);
   Coin(int value) {this.value = value; }
   private final int value;
   public int value() {return value; }
}
```

Statikus import

A statikus import lehetőséget ad arra, hogy egy osztály statikus mezőire úgy kelljen hivatkozni, hogy meg kell adni az osztály nevét. A statikus import szintaktikája:

```
import static java.util.Calendar.*;
Calendar c = new Calendar();
C.get(DAY_OF_MONTH);
```

Formázott adatbevitel és kiírás

Megjelent a C típusú formázott adatbevitel és kiírás lehetősége is, melyet a C nyelvben a **scanf** és **printf** függvény valósított meg. Java nyelvi megfelelői:

```
System.out.printf("%s %5d\n", user,total);
Scanner s = Scanner.create(System.in);
String param = s.next();
int value = s.nextInt();
S.close();
```

Ezzel kapcsolatban több információt a **java.util.Formatter** osztály API dokumentációja ad.

Változó számú paraméterlista

A változó számú paraméterlistát is a C nyelvből vették át a következő szintaxissal:

```
void argtest(Object ... args) {
 for (int i=0;i <args.length; i++) {
   System.out.println(args[i]);
 }
}
Argtest("test", "data");
```

A példa **argtest** metódusát bármilyen számú és típusú paraméterrel meg lehet hívni, melyeket a metódus kiír.

Párhuzamos programok

A párhuzamos programok fejlesztésére is jelentős számú új funkcionalitást vezettek be, melyek a JSR-166-on alapulnak.

Dinamikus proxyk

Nincs többé szükség az rmic (RMI compiler) eszközre, ami a távoli interfész vázakat generálta, ehelyett a dinamikus proxy-k bevezetésével a régebben a vázak által biztosított adatok futásidőben felderíthetők.

Konklúzió

A Java 1.5-ben megjelenő újdonságokra a Java programozók már régóta vártak. A tervezők maximálisan figyelembe vették a visszafelé kompatibilitást, a régi programok migrációjának lehetőségét. Az új funkciók segítik a programozást, használatukkal átláthatóbb, tisztább, és biztonságosabb programokat lehet írni.

Az átállásra még van idejük a fejlesztőknek, hiszen a közelmúltban jelent meg az Early Access program keretében a beta verzió, és alig van olyan fejlesztőeszköz, mely támogatná az új lehetőségeket. De addig is érdemes velük megismerkedni, hogy könnyen váltani tudjunk.
Ez a cikk csak néhány új nyelvi elemet mutat be nagy vonalakban, de ezeken kívül rengeteg új funkció, újítás van az új verzióban, melyek jelentősen könnyíthetik a programozó munkáját, így megismerésük feltétlenül ajánlott.

Hivatkozások

A szerzőről

Viczián István a Debreceni Egyetem program-tervező matematikus szakán végzett 2001 nyarán, most ugyanott levelező PhD hallgató az elosztott rendszerek, middleware-ek témakörében. Jelenleg vezető fejlesztőként dolgozik Budapesten, melynek keretében csoportmunkát segítő eszközökkel, Java nyelvvel, webes technológiákkal, middleware szoftverekkel és alkalmazásintegrációval foglalkozik. Szabadidejében optimalizációs alkalmazást fejleszt, új Java technológiákkal ismerkedik, valamint Java blog-ot ír (JTechLog).

e-mail: viczus@freemail.hu
Honlap: http://dragon.unideb.hu/~vicziani

Java Naming and Directory Interface

A JNDI távoli objektumok elérésre szolgál, függetlenül attól, hogy azok Java nyelven íródtak vagy sem.

Context és InitialContext

A Context interfész létfontosságú szerepet játszik a JNDI-ben. Egy context egy bindings(kötés)-sorozatot képez egy konkrét névszolgáltatáson belül. Egy Context objektum metódusokat szolgáltat az objektum binding nevének, az objektum elérését biztosítja amikor hivatkozás történik annak nevére, kilistázza a használt binding-eket és biztosítja a bindin-gek esetleges átnevezését.

A JNDI a context-re nézve relatívan hajtja végre az összes névszolgáltatást. Induláskor a JNDI specifikációk egy InitialContext osztályt értelmeznek. Ezen osztály instanciálva van azon tulajdonságokkal kezdődően, amelyek értelmezik a névszolgáltatás típusát , és ahol szükséges a kezdeti kontextushoz való kapcsolódás pillanatában, a felhasználóinév és jelszó, biztonsági elemeket is használ.

Az InitialContext értelmezési tulajdonságai

A kezdeti kontextus létrehozásának pillanatában, ennek konstruktora egy sor inicializálási információt kell átvegyen. Ezen információk szolgáltatása TulajdonsagNev=TulajdonsagErtek formában történik. A TulajdonsagNev két legfontosabb értéke:

```
java.naming.provider.url
java.naming.factory.initial
```

Például:

```
java.naming.provider.url=rmi://localhost:1099
java.naming.factory.initial=com.sun.jndi.rmi.regi
stry.RegistryContextFactory
```

A factory osztály különböző szolgáltatásokhoz leggyakrabban használt TulajdonsagErtek-eket tartalmazza az alábbi táblázat:Ezen tulajdonságokon kívül vannak mások is, de azokat ritkán használjuk.

Service	Factory
Filesystem	com.sun.jndi.fscontext.FSContextFactory sau com.sun.Jndi.fscontext.RefFSContextFactory
RMI registry	com.sun.jndi.rmi.registry.RegistryContextFactory
COS	com.sun.jndi.cosnaming.CNCtxFactory
DNS	com.sun.jndi.dns.DnsContextFactory
LDAPv3	com.sun.jndi.ldap.LdapCtxFactory
NIS	com.sun.jndi.nis.NISCtxFactory
NDS	com.novell.naming.service.nds.NdsInitialContextFactory

Ezen tulajdonságokon kívül vannak mások is, de azokat ritkán használjuk.

Hogyan adjuk át a különböző tulajdonságokat a JNDI-nek?

A kezdeti kontextus felépítéséhez szükséges tulajdonságok négy módon adhatók át:

1. egy Hashtable objektum létrehozásával, amely tartalmazza a tulajdonságokat, és amelyet bemenő paraméterként adunk át az InitialContext-nek
Például:

```
Hashtable ht = new Hashtable();
ht.put("java.naming.factory.initial","com.sun.jnd
i.rmi.registry.RegistryContextFactory");
ht.put("java.naming.provider.url",
"rmi://localhost:1099");
Context ctx = new InitialContext(ht);
```

vagy

```
Hashtable ht = new Hashtable();
ht.put(Context.
INITIAL_CONTEXT_FACTORY,"com.sun.jndi.rmi.registr
y.RegistryContextFactory");
ht.put(Context.PROVIDER_URL,
"rmi://localhost:1099");
Context ctx = new InitialContext(ht);
```

2. ezen tulajdonságok hozzáadásával a rendszertulajdonságokhoz.Ez kétféleképpen lehetséges:
Parancssorban-System.setProperty metódus használatával
Példa:

```
System.setProperty("java.naming.factory.initial",
"com.sun.jndi.rmi.registry.RegistryContextFactory
");
System.setProperty("java.naming.provider.url","rm
i://localhost:1099");
```

vagy

```
System.setProperty(Context.
INITIAL_CONTEXT_FACTORY,"com.sun.jndi.rmi.registr
y.RegistryContextFactory");
System.setProperty(Context.PROVIDER_URL,"rmi://lo
calhost:1099");
```

És csak ez(ek) után:

```
Context ctx = new InitialContext();
```

3. a paraméterek egy ?APPLET tag általi értelmezésével, abban az esetben, ha a kliens egy JNDI appletet használ
Példa:

```
<applet code = "Prog.class" width = 600 height =
100 >
<param name="java.naming.factory.initial"
value="com.sun.jndi.rmi.registry.RegistryContextF
actory">
<param name="java.naming.provider.url"
value="rmi://localhost:1099">
</applet>
```

4. a tulajdonságok egy lokális erőforrásfileban való specifikálásával
Egy JNDI-t használó alkalmazás futtatásakor, ez a classpath által látható könyvtárakban keresi a jndi.properties nevu filet. Ezen file minden sorát, mint tulajdonságot értelmezi. Tehát a jndi.properties file tartalma lehet például:

```
java.naming.factory.initial=com.sun.jndi.rmi.regi
stry.RegistryContextFactory
Java.naming.provider.url=rmi://localhost:1099
```

Műveletek attributumokkal

Úgy a DirContext, mint az InitialContext az attributumokkal való műveletekhez specifikus metódusokat ajánl. A legfontosabb metódusok:

```
void bind String nev, Object objektum, Attributes
attributumok)
void rebind String nev, Object objektum,
Attributes attributumok)
DirContext createSubcontext(String nev,
Attributes attributumok)
Attributes getAttributes(String nev)
Attributes getAttributes(String nev, String
[]AttributumNevek)
void modifyAttributes(String nev, int muvelet,
Attributes attributumok)
```

A konstans muveletek:

```
ADD_ATTRIBUTE,
REPLACE_ATTRIBUTE,
REMOVE_ATTRIBUTE.
```

JDBC használata JNDI segítségével

A JDBC 2,0-s verziójától kezdődően egy sor új dolog jelent meg a javax.sql csomagban. Ezek közül említésre méltó: JDBC-ben való JNDI használat, kapcsolatokkal és osztott tranzakciókkal kapcsolatos műveletek. Egy driver, amelyik támogatja a JDBC 2.0-t, kell tartalmazza legalább a DataSource interfész implementálását. Napjaink drivereinek többsége implementálja a JDBC-nek ezt a bővítését.
Na, és miért fontoz ez számunkra? Mert a DataSource által egyszerübben tudunk kapcsolódni az adatbázishoz, mint a DriverManager használata által. DataSource használata tipikus JNDI:

```
javax.naming.Context cx = new InitialContext();
javax.sql.DataSource ds
=(DataSource)cx.lookup("JNDIAdatBazisNev");
javax.sql.Connection co = ds.getConnection();
```

A kapcsolat létrejötte után minden ugyanúgy zajlik, mint ahogy az Adatbázisok Java-ban címü fejezetben ecseteltem. Természetesen ezután a kapcsolattal kapcsolatos egyéb konfigurációk mind a JNDI feladatkörévé válnak.

Az aktuális törekvés a DriverManagertol való fokozatos megválás és a DataSource előtérbe helyezése. Napjainkig nagyon sok modern adatbáziskezelő: **ORACLE, MySQL, MSSQLServer**,sőt az **Access** is lehetővé tette az adatbázishoz való kapcsolódást JNDI által. Úgyszintén, jópár technológia, mint a Tomcat és EJB implementálták a JDBC-hez szükséges JNDI interfészeket.

Finta Anna Maria

ESEMÉNYKEZELÉS FLASH MX

Cikkünkben a Flash MX eseménykezelő-modelljével ismerkedünk meg részletesebben. Nagy fejlődést mutat ez a Flash 5 lehetőségeihez képest, szépen felépített és jól áttekinthető, ugyanakkor moduláris és könnyen variálható scripteket írhatunk. Ezekkel egyszerűbben és sokrétűbben hozhatunk létre animációkat, ötletes és hasznos interaktív elemeket. Az alapvető meghatározások mellett néhány trükköt is bemutatunk, amelyek elsőre nem feltétlenül egyértelműek, de a későbbiek folyamán igen hasznosnak bizonyulhatnak.

Alapvető ismeretek

Elsőként az **esemény (event)** fogalmát kell tisztáznunk. Eseménynek nevezzük az a dolgot, ami a mozi futása közben történik. Ez lehet **a felhasználó által generált esemény**, mint pl. egy egérkattintás vagy egy billentyű lenyomása, ill. lehet **rendszeresemény**, amire jó példa az, amikor a mozi egy képkockáról a következőre lép. A mozinkban elhelyezett objektumok figyelhetik ezeket az eseményeket, és reagálhatnak is rájuk. Ezek az **eseményfigyelők (listener)**. Nem mindenféle objektum figyel minden eseményre, valamelyikhez nekünk kell hozzárendelni azt. Ilyen pl. a moziklip objektum: figyel az egéreseményekre, de a billentyűzetfigyelésre már utólag kell rábírnunk.

Az **eseménykezelő** valójában egy függvény, ami akkor hajtódik végre, amikor az adott esemény bekövetkezik. Eseménykezelő pl. az onEnterFrame vagy az onMouseMove, ez azonban nem csinál semmit addig, amíg nem definiálunk egy függvényt, amit végrehajthat (addig undefined az értéke). Itt fontos megjegyeznünk, hogy valójában az enterFrame esemény következik be, és az onEnterFrame, ill. az ahhoz kapcsolt függvény az, amivel megmondjuk, hogy mi történjen. Azt, hogy enterFrame, Flash MX-ben nem írjuk a scriptbe soha, csak azért használjuk így, hogy tudjunk beszélni róla.

Flash 5 stílusban, ha egy moziklippet mozgatni akarunk, a legkézenfekvőbb megoldásnak az bizonyul, ha kihúzunk egy példányt a színpadra, és arra ráírjuk pl. a következőt:

```
onClipEvent(enterFrame) {
_x+=Math.round(Math.random()*5);
}
```

A fenti kód az enterFrame esemény érzékelésének köszönhetően másodpercenként annyiszor lefut, amennyi az fps-nél beállított érték. Ez alapértelmezés szerint 12, de a 24-25 az ideális, mert már a folyamatosság látszatát kelti, de még nem gazdálkodik túl pazarlóan egy átlagos gép erőforrásaival (most állítsuk 25-re). Tehát az eredmény az, hogy a moziklipünk 0 és 5 közötti véletlenszerű egész egységekkel jobbra mozdul, egyfajta egyenetlen sebességű, egyirányú mozgást produkálva. Tegyük fel, hogy 15 golyóbist szeretnénk ily módon mozgatni. Így mindegyikre rá kell ezt a kódot másolni. Ez egyrészt felesleges időpocsékolás, másrészt ha egy apró kiegészítést szeretnénk a viselkedésükben (pl. az _y tulajdonságot is növelni akarjuk), mindenhol át kell írni a sorokat, ami az időt és az átláthatóságot tekintve sem optimális megoldás. Ehelyett egy más megközelítést alkalmazunk. Hozzunk létre egy új mozit, ebben egy script és egy golyok nevű réteget a fő idősíkon (_root), nevezzük el a golyobis moziklip egy példányát golyobis_mc-nek a golyok rétegen, és írjuk a script réteg egyetlen frame-jére a következőket:

```
function mozgas () {
this._x+= Math.round(Math.random()*5);
}

stop();
golyobis_mc.onEnterFrame=mozgas;
```

Az első, amire rá kell mutatnunk, a this szerepe. Ha egy változót, tulajdonságot, szimbólum példánynevet elérési út nélkül adunk meg (pl. _x vagy golyobis_mc), az az adott idősíkra fog vonatkozni, esetünkben ez a _root-ot jelentené. A this használatával viszont a hatókört a meghívó objektumra, itt éppen a golyobis_mc-re helyezzük át. Direkt meghívással (mozgas() v. _root.mozgas()) a teljes fő idősík mozdulna, de így, hogy a golyobis_mc onEnterFrame eseménykezelőjéhez rendelve hívódik meg, csak a golyobis_mc mozdul.

A második dolog, hogy attól még, hogy a stop()-pal leállítottuk a fő idősíkot, és valójában nem ugrunk a következő frame-re, (főleg, ha csak egy frame-ből áll a mozi), az onEnterFrame-hez csatolt függvény még továbbra is folyamatosan lefut a fps-nek megfelelő gyakorisággal.
A harmadik, és egyben legfontosabb említendő momentum az eseménykezelő metódus definiálása. Amikor egy névvel ellátott függvényt hozunk létre, maga a név mutat a memóriában létrehozott utasításhalmazra. Ezt a hivatkozást egy másik változónak - pl. masikMozgas=mozgas -, vagy akár a fent látott módon egy eseménykezelőnek is átadhatjuk. Vigyáznunk kell viszont arra, hogy ekkor zárójelek nélkül kell a függvénynevet használnunk, mert a zárójelpár mindenképpen meghívást jelent.

```
// az onEnterFrame a mozgas fv. visszatérési
értékét veszi fel - helytelen módszer
golyobis_mc.onEnterFrame=mozgas();

// az onEnterFrame magára a mozgas fv.-re
mutat
golyobis_mc.onEnterFrame=mozgas;
```

Tegyünk egy próbát! Az alábbi esetben a mozgas függvény mindössze egyszer hívódik meg, a visszatérési értéke a golyóbis _x tulajdonsága, amit a trace paranccsal kiíratunk az Output ablakba, mint a golyobis_mc.onEnterFrame értékét. Ennek pedig nem sok értelme van, azon felül, hogy nem is működik a mozgás, amit látni szeretnénk. Ha viszont az utolsó előtti sorból a zárójelpárt levesszük, rendesen működik az animáció, a trace pedig immár helyesen type Function-t ír ki, mivel az onEnterFrame-hez függvényt rendeltünk.

```
function mozgas () {
this._x+= Math.round(Math.random()*5);
return this._x;
}

stop();
golyobis_mc.onEnterFrame=mozgas();
trace(golyobis_mc.onEnterFrame);
```

Ezzel azonban még nem vagyunk készen, ugyanis csak egy golyóbist mozgattunk. 15 példányt a színpadra kihúzva, és azokat a golyobis0_mc, golyobis1_mc stb. sorozatként golyobis14_mc-ig elnevezve a következő egyszerű, rövid ciklussal az összes moziklipet irányíthatjuk.

```
function mozgas () {
this._x+= Math.round(Math.random()*5);
}

stop();
for (i=0; i<15; i++) {
_root["golyobis"+i+"_mc"].onEnterFrame=mozgas;
}
```

Itt azt használjuk ki, hogy különböző objektumokat, szimbólumok példányait, azok tulajdonságait, metódusait el lehet érni a tömbjelölés segítségével is, ahol a nevet sztringként alkalmazzuk. _root["golyobis"+6+"_mc"] ugyanazt jelenti, mint _root.golyobis6_mc.
Ezen kívül még létezik egy másik lehetőség is, ami adott esetben még hasznosabb lehet, mivel a for ciklusra nincs szükség, és még a sorozatos elnevezéssel sem kell bíbelődni, bár a nevek használatának nélkülözése a későbbi, ill. futásidejű változtatásokat nem könnyíti meg. Töröljünk le minden moziklip-példányt a Stage-ről, a _root-on pedig csak a mozgas függvénydeklarációt és a stop()-ot hagyjuk. Magában a golyobis Library-elemben a scriptnek hozzunk létre egy - a szokásos módon script nevű - külön réteget, ennek egyetlen frame-jére pedig a következőt írjuk:

```
stop();
onEnterFrame=_root.mozgas;
```

Ettől kezdve az összes színpadra kihúzott példány lassan áthalad a képernyőn, mindegyik a saját véletlenszerű sebességváltozásainak megfelelő módon.
Ha azt szeretnénk, hogy mondjuk az x tengelyen 300 pixelnél megálljanak a golyóbisok, lehet a mozgas függvénybe egy olyan feltételt tenni, ami ha 300-nál nagyobb lenne az _x, akkor az maradjon 300-on. Ekkor azonban az onEnterFrame még azután is fut, hogy már nincs rá szükség, mert a golyó áll. Ha a mozi futása során még sok onEnterFrame-et használunk, és egyiket sem állítjuk le, miután elvégezte a dolgát, előbb-utóbb irgalmatlanul belassul a lejátszás. A feleslegesen futó metódust érdemes ilyenkor lekapcsolni az eseménykezelőről. Tegyük a golyóbisokat 300-nál kisebb x-pozícióba, és írjuk át a mozgas függvényt a következők szerint:

```
function mozgas () {
this._x+= Math.round(Math.random()*5);
if (this._x>=300) {
    this._x=300;
    this.onEnterFrame=undefined;
}
}
```

A this.onEnterFrame=undefined ugyanazt eredményezi, mint a delete this.onEnterFrame, jóformán ízlés kérdése, hogy melyiket használjuk. Ezen utasítások nem szüntetik meg a mozgas függvényt, csupán innentől kezdve az nincs hozzárendelve az eseménykezelőhöz. Amennyiben a későbbiek során újra mozgatni akarjuk valamelyik golyóbisunkat, elég újra a golyobis1_mc.onEnterFrame=mozgas formát használni, miután az egyik moziklipünket golyobis1_mc-nek neveztük el. Máris halad tovább a moziklip, amennyiben valahogy visszatettük előtte 300-nál kisebb x-pozícióba. Erre megoldásként kínálkozik a következő példa, mely rögvest az eseménykezelők hozzárendelésének egy másik módját is

bemutatja. A _root-on lévő script végére tegyük a következőket:

```
onMouseDown=function() {
    golyobis1_mc._x=150;
    golyobis1_mc.onEnterFrame=mozgas;
};
```

Előbb visszahelyeztük a moziklipet, majd mozgásra bírtuk, és mindezek megtörténtét egy egérkattintáshoz kötöttük. Ami újszerű, az az, hogy itt közvetlenül az eseménykezelőhöz (onMouseDown) rendeltünk egy névtelen függvényt. Ezt a rövidebb módszert akkor érdemes használni, amikor tudjuk, hogy a függvényt csak egyszer akarjuk az adott kezelőhöz kapcsolni. Ha ugyanis egyszer - pl. az undefined alkalmazásával - leválaszottuk az eseménykezelőről, később már - név hiányában - nem tudunk rá sehogy sem hivatkozni.

Ezt a formát kell akkor is követnünk, amikor az eseménykezelőhöz tartozó függvénynek paramétereket kívánunk átadni. Az eddigi példánál maradva, immár paraméterekkel szeretnénk megmondani, hogy meddig menjen a golyóbis. Az onMouseDown-t és a hozzá tartozó névtelen függvényt töröljük ki a _root-ról, a mozgas függvényt pedig írjuk át az alábbiaknak megfelelően:

```
function mozgas (klip, hatar) {
klip._x+= Math.round(Math.random()*5);
if (klip._x>=hatar) {
    klip._x=hatar;
    klip.onEnterFrame=undefined;
}
```

A golyobis Library-elemben található onEnterFrame-et a következőképpen fogalmazzuk át:

```
onEnterFrame = function () {
    _root.mozgas(this, 200);
};
```

A mozgas függvényt kibővítettük két paraméterrel, a klipp az adott moziklippre fog vonatkozni, a hatar pedig azt hiszem, nem szorul magyarázatra. Az onEnterFrame-hez közvetlenül nem használhatjuk az onEnterFrame=_root.mozgas(this, 200) formát, mivel megbeszéltük, hogy a zárójelek miatt a mozgas rögtön meghívásra kerülne és csak egyszer, helytelen működést eredményezve. Így tehát egy névtelen függvénybe kell csomagolnunk a mozgas meghívását. Mivel már nem közvetlenül az eseménykezelőhöz kapcsoltuk azt, hanem direkt módon meghívtuk, ha a mozgas-ban nem lenne klip paraméter, és a this-eket meghagytuk volna, - a cikk elején bemutatottaknak megfelelően - a teljes fő idősík mozdulna el, mivel a this ez esetben arra mutatna. Így szükséges bevezetnünk egy paramétert, hogy megmondjuk, mire vonatkozzon a mozdítás. A golyobis-ban a _root.mozgas(this, 200)-ban szereplő this mindegyik golyóbis-példány saját idősíkját jelenti, így megoldottuk a feladatot.

Ugyanúgy, ahogy először hozzákapcsoltunk valamilyen függvényt egy eseménykezelőhöz, majd leválaszottuk róla, természetesen nem csak ugyanazt rendelhetjük hozzá újra, hanem akármilyen másik függvényt is. Ki is próbálhatjuk, például írjunk egy újabb függvényt a mozgas elé, ami a mozgás végeztével triplájára növeli a golyóbisunkat:

```
function noveles () {
    this._xscale=this._yscale+=10;
if (this._xscale>=300) {
        this._xscale=this._yscale=300;
this.onEnterFrame=undefined;
}
}
```

A mozgas függvényben csak a következőt kell változtatni: klip.onEnterFrame=noveles.

További trükkök, fontos apróságok

Az eseménykezelők közül érdemes még kiemelni az onMouseMove-ot, mivel segítségével könnyedén készíthetünk saját egérmutatót, és egy fontos elvi dologra világítunk rá. Egy teljesen új, egy frame-ből álló moziban hozzunk létre egy klip és egy script réteget. A klip rétegre egy tetszőleges, egeret helyettesítő moziklipet tegyünk, aminek legyen eger_mc a neve, a script rétegre pedig írjuk a következőket:

```
stop();
Mouse.hide();
eger_mc.onMouseMove=function() {
    this._x=_xmouse;
this._y=_ymouse;
};
```

A Mouse.hide()-dal eltüntetjük az eredeti egeret, és ezzel készen is volnánk, de van egy kis bökkenő. Itt ugyanis csak onnantól kezdi az egérmutató helyét követni a moziklipünk, miután megmozdítjuk egerünket. Próbáljuk ki, a Flash MX-ben Ctrl+Enter-rel teszteljük a mozit, de ne mozdítsuk meg az egeret, csak később. Addig a moziklip a színpadon van a kezdeti pozícióban. Kézenfekvőnek látszik a megoldás, hogy meg kell hívni az onMouseMove-hoz tartozó függvényt. Csakhogy az egy névtelen függvény, nem tudunk rá hivatkozni a megszokott függvényhívást alkalmazva. Beírni mégegyszer az onMouseMove-ban szereplő utasításokat, hogy ezek egyszer a mozi kezdetén mindenképp lefussanak, nem valami professzionális megoldás. Szerencsére az eseménykezelőhöz kapcsolt függvényt meg tudjuk hívni a következőképpen is: eger_mc.onMouseMove(). Írjuk is be a kódunk végére. Ez természetesen itt nem takarít meg sok helyet és munkát, de egy bonyolult, hosszú eseménykezelő metódus esetében ez a módszer nagyon hasznos lehet. Gondoljunk egy sok gombból álló menüre, mindegyik onRelease-éhez különböző függvények, utasítások tartoznak: ha valahonnan szeretnénk az egyik menüpontra történő kattintást utánozni tényleges kattintás nélkül, elég csupán valami hasonlót csinálni:

```
tesztKlip_mc.onRollOver=function() {
trace("mintha a menupont3_mc-re kattintottunk
volna");
    menupont3_mc.onRelease();
}
```

Ily módon paramétereket is átadhatunk az eseménykezelő metódusnak, ritkán van rá szükség, de jól jöhet. Hogy mikor érdemes alkalmazni, azt már az Olvasóra bízzuk:

```
proba_mc.onRelease=function(par1, par2) {
    // fuggvenytorzs
}
proba_mc.onRelease("valami", pl_egy_valtozo);
```

Egy pár mondat erejéig visszatérve az egérkövető moziklipphez, a függvénytörzs utolsó soraként érdemes beszúrni az updateAfterEvent() utasítást. Ez annyiszor frissíti a képernyőt, ahányszor csak az adott esemény bekövetkezik. onMouseMove-nál használva gyakorlatilag nagyon szép folyamatos mozgást produkálhatunk, akár 1 fps mellett is. Állítsuk ilyen alacsonyra, próbáljuk ki updateAfterEvent()-tel, aztán nélküle is - észrevehető a különbség :)

Végezetül nem szabad elmennünk a függvények deklarálásának két módja mellett:

```
function probaFuggveny() {
}

probaFuggveny=function() {
}
```

Meghíváskor ugyanúgy a probaFuggveny()-t használjuk, de az eseménykezelőkhöz rendelésnél vigyázni kell a sorrendre. Ha az első formát alkalmazzuk, mindegy, hogy a kódban az eseménykezelőhöz a függvénydeklaráció előtt vagy után rendeljük a függvényt, mert a Flash legelőször a függvényeket nézi meg. A második formánál azonban annak ellenére, hogy itt is függvényről van szó (még ha névtelenről is), egy értékadás jobb oldalán találjuk a function-t, így ezt már a szekvenciális megközelítésnek köszönhetően mindenképpen az eseménykezelőhöz kapcsolás elé kell írni, ugyanis addig nem létezik a függvény, amíg a Flash „fentről lefelé” el nem jut odáig az értelmezésben.

Összefoglalás

Áttekintettük tehát az eseménykezelés csínját-bínját, az alapokat, illetve az azokhoz kapcsolódó, nem túl kézenfekvő, de lényeges ismereteket. Mindezeket egyszerű példákon keresztül végigkövetve az Olvasó már bátran építkezhet. E módszerek ismeretében és kreatív alkalmazásával megvan a lehetősége olyan Flash-mozik készítésére, melyek a programozó és a felhasználó számára egyaránt az elégedettség élményét nyújtják.

Molnár Ákos - flessmx@yahoo.com

A kibicnek semmi sem drága!

Sokaknak ismerős lehet az a csalódottsággal teli, nyomasztó érzés, ami akkor tölti el az embert, amikor ez a találó mondás kívánkozik a nyelvére. Az igazságérzet és bizonyítási vágy fűtötte indulatot, mégis gyakran lehűti a kiábrándultság és a kiszolgáltatottság. A mondat talán már el sem hangzik. Minden alkotó ember tudja: érvnek ez bizony elég kevés! A szerződés kötelez, a megrendelő diktál hiszen ő fizet a vásárlónak pedig mindig igaza van!

Mielőtt azonban az önsajnálat tagadhatatlanul kényelmes, de zsákutcát jelentő állapotába lovallnánk magunkat, be kell látnunk: alkotni nem kötelező! Ha az embert büszkeséggel tölti el, hogy viszontlátja ötleteit a képernyőn, megjelenik a neve egy CD borítón, esetleg egy újság vagy könyv lapjain, akkor ennek bizony ára van. Jó érzés végigtekinteni az elkészült programon, íráson, grafikán és belegondolni, hogy ez részévé válik sok általunk nem ismert ember munkájának, szórakozásának. De botorság lenne azt hinni, hogy ez megtörténhet kompromisszumok nélkül. Sőt gyakran éppen a nehézségek és akadályok kényszerítik ki azokat a bizonyos „nagy ötleteket”! Bár erre mindig csak utólag jövünk rá.

Jól ismerjük ezeket a „legendás” kompromisszumokat a nagy művészekről szóló anekdotákból. Szinte minden híres festőről, szobrászról, íróról, költőről keringenek olyan történek, amelyek az alkotó elme győzelmét hirdetik a körülmények, az ellenlábasok vagy a középszerűség felett. A híressé vált tudósoknak szinte „védjegyévé” vált ez a képesség, hiszen a szürke hétköznapokon végzett „szellemi favágás” a kívülálló számára érdektelen, az eredményre pedig nincsen 100%-os garancia.

De hova soroljuk be a számítástechnika szellemi alkotásait? Kézenfekvőnek tűnik, hogy a tudományok családjában keressük a „rokonságot”. Nem is tévedhetünk nagyot, hiszen magának az eszköznek a számítógépnek a létrejötte és fejlődése tudós koponyák egész hadát igényelte és igényli ma is. Ezzel azonban látszólag már el is zárkózunk a többi nagy „családtól”, mint a művészet, a kultúra vagy éppen a filozófia. A mindennapi példák viszont éppen azt mutatják, hogy a számítástechnika legújabb nemzedéke már előszeretettel „házasodik be” hol jól, hol rosszul ezekbe a „családokba”. Nem is olyan régen még heves, érzelmektől sem mentes vitákat váltott ki, ha egy hagyományos művészet neve előtt megjelent a „számítógépes” jelző. (Na nem mintha ez új lenne a nap alatt! Jó száz éve még minden „épeszű” ember számára nyilvánvaló volt, hogy a benzinmotoros autó soha nem veheti fel a versenyt egy megbízható hátaslóval, a fényképezés pedig, mint művészet nem említhető egy lapon a festészettel!) A történelem ugyan nem azt mutatja, hogy az emberiség túl sokat tanulna a korábbi hibáiból, azért a számítástechnika (ahogy például az elektromosság vagy vegyészet) lassan mégis csak „beilleszkedett”. Ma már a többség nem kételkedik abban, hogy egy számítógéppel készített grafikának is lehet (persze nem automatikusan) művészi értéke és egy zenész sem válik művészből tudóssá csak azért, mert számítógép segítségével komponál.

Belátták ezt a software ipar mértékadó képviselői is, így mára szinte természetes, hogy a művészi kreativitást igénylő részfeladatokat valóban művészek végzik el. (Ezt tapasztalhattuk már korábban az autóiparban is, ahol a „nyers” technikát már jó ideje igyekeznek igényes esztétikai kivitelben megjeleníteni.) Továbbra is fennáll azonban a kérdés? Minek is minősül egy számítógép program elkészítése? A közvélekedés szerint leginkább szellemi szakmunka, amely alapvetően a (matematikai) logikából építkezik. El kell ismerni, hogy ez az esetek elsöprő többségében igaz is. Néha amikor valóban valami új születik a programozás tudománnyá válhat. Ne feledjük: a ma mindennapos, megszokott technika születésekor bizony tudományos vívmányként látta meg a napvilágot! De vajon rejlik-e művészet a számítógép programozásban? És itt most nem az informatikába „ágyazott” művészetről van szó! Szó sincs design-ról, welcome screen-ekről, intro-król vagy aláfestő zenéről. A kérdés az, hogy van-e a (számítás)technikának esztétikája? Lehet szép egy algoritmus? Létezik csúnya és szép megoldás egy problémára? Lehet egy kevesebbre képes program szebb fejlettebb riválisánál? Nevezhetjük-e Peter Norton legendás Commander-ét például szépnek függetlenül a külső megjelenéstől? Mi lehet az oka, hogy oly sokan használták sőt használják még évekkel az után is, hogy technikai értelemben elszállt felette az idő? Talán ugyanazért, amiért még mindig gurulnak az utakon Volkswagen „bogarak” és Citroën „kacsák” és a gőzmozdony vontatta vonatnak is még van utasa. Persze lehet, hogy mindez csak nosztalgia, de vajon mitől függ, hogy mi iránt érzünk nosztalgiát? Emlékei között kutatva biztosan minden számítógép kedvelő talál néhány olyan berendezést, programot vagy trükköt, ami maradandó nyomot hagyott a gondolataiban.

Mi lehet mégis az oka annak, hogy a számítástechnika művelőit még mindig külön „kasztként” kezeli az átlagember? Miért hisszük el azt a Hollywood által agyonkoptatott képet, ami szerint a számítástechnikus egy sötét, bűzös lukban ücsörgő, szemüveges különc zseni? Talán azért mert zseninek látszani nem is olyan rossz dolog. Nem is kell hozzá sokat tenni, hogy ez a látszat megmaradjon. Néhány elhadart szakkifejezés, egy-két „csípőből” megoldott (ál)probléma és persze a megfelelő pillanatban alkalmazott rejtélyes hallgatás. A gond csak az, hogy ezzel a képpel semmivé foszlik a háttérben húzódó valódi teljesítmény! A sok megoldott probléma, a kitartó szorgalmas munka, a rengeteg tanulás és már-már megszállott alaposság. Elhatározás kérdése személyek és cégek esetén is hogy felépítjük-e a tévedhetetlenség mítoszát. Aki viszont megteszi, ne csodálkozzon a kárörvendő kibicek hadán, amikor a mítosz mégis „megroggyan”! Senki sem szeret kérkedni a hibáival, tévedéseivel. Érthető, ha valaki nem veri nagydobra a gondokat, csak „csendben” tanul belőle. A problémák létezésének reflexszerű tagadása viszont nem vall nagy bölcsességre. A laikusokat talán meg lehet téveszteni ezzel, ám a kicsit jártasabb embernek ez mindig is „sántítani fog”.

Különösen igaz ez a számítástechnika egy olyan szerteágazó területére, mint a multimédia. Aki már kicsit is belekóstolt alkotóként ebbe a műfajba, hajlamos úgy érezni, hogy itt nem csak a „média” „multi” tehát sokszoros hanem a gond, a probléma, a buktató no meg persze a lehetőség is! Valóban lenyűgöző távlatokat nyit meg a technika fejlődése! Nem tévedünk sokat, ha azt érezzük: csak rajtunk áll merre és meddig jutunk el ezzel a lehetőséggel! A gondok és nehézségek rejtegetése itt viszont már valóban reménytelen. Nem is gondolhatjuk komolyan, hogy óramű pontossággal készülhet el egy olyan mű, amelyben igényes grafikai arculat jeleníti meg a szakmailag, nyelvileg magas szintű tartalmat, mindezt a legújabb technika által megteremtett legjobb minőségű hang és mozgókép élménnyel „dúsítva” és persze olcsón, gyorsan, egyszerűen! A követelmények hacsak el nem hagyunk belőle néhányat már önmagukban hordozzák a kompromisszum kényszerét, nem is beszélve arról, hogy mindez csak csapatmunkában elképzelhető, ami szintén nehéz bár sokszor építő jellegű vitákkal jár. De talán mindez nem olyan nagy gond! Talán a problémák, tévedések, nehézségek is vannak olyan érdekesek, izgalmasak ha nem izgalmasabbak mint maga a megoldás! Ezt persze csak az tudja eldönteni, aki mindkettővel szembesül. Ezért határozta el e cikk szerzője, hogy egy szellemi kalandtúrára invitálja meg a kedves olvasót! Hónapról-hónapra fogjuk végigkövetni egy multimédia CD-ROM „életútját” itt a Codex hasábjain, az ötlet megszületésétől a reménykedve várt megjelenésig. És, hogy mi a garancia a kalandra? Nem „vegytiszta” tanpéldáról van szó. E sorok írásakor a CD-ROM még közel sincsen készen. Hátra van még számos gond és akadály, amelyek gerincét fogják képezni a tervezett cikksorozatnak. Bizonyára meglátogatunk utunk során néhány zsákutcát, teszünk néhány vargabetűt. Talán éppen ettől lesz majd örömteli a most még csak homályosan derengő végcél elérése! A CD alkotóinak kis csapata aggódó, de bizakodó szülőkként követik az első lépéseket. Az ezzel járó felfedezés örömét szeretnénk ezúton megosztani a Codex olvasóival. Így erre a hónapra nem is maradt más mondandó. Kalandra fel!

Szűcs Zoltán **szucs.zoltan@intermail.hu**

PHP kezdőknek I.

Tudom a cím nem túl eredeti, de ez tükrözi talán legjobban ennek a cikksorozatnak a lényegét, és ha valaki most akar megismerkedni a PHP rejtelmeivel, akkor úgyis az ilyen című cikkekre veti rá magát. Ebben a sorozatban tehát a PHP-val fogunk megismerkedni.

Ha manapság valaki elkezd foglalkozni a weblapfejlesztéssel, annak szinte kötelező, hogy ismerje a PHP script nyelvet. Ennek oka, hogy ez a script nyelv legalább olyan egyszerű, mint a Java script, de annál sokkal biztonságosabb, és hatékonyabb. A php csak a kiszolgálón futtatható script nyelv, ami azt jelenti, hogy a script a szerveren fut le, és nem pedig a felhasználó számítógépén. Ez azt is jelenti, hogy a felhasználó nem látja a scriptet, ebből fakad a biztonság. Ilyet a Java script is tud, csak sokkal kevesebb funkcióval, és bonyolultabban. Emiatt hatékonyabb a PHP.

Nos akkor nem is húzom tovább az időt mindenféle összehasonlítgatással, hiszen ez nem is igazán lehetséges, és nem is annyira érdekes. Inkább vágjunk rögtön a közepébe, vagy ha nem is pont a közepébe, de legalábbis a lényeges részébe a dolgoknak. Vagyis, hogy hogyan is épül fel a PHP script.

Gondolom, aki már olvasott PHP-vel foglalkozó könyvet, az lehet, hogy felhördül: miért nem a PHP fordító telepítésével kezdem. Nos a válaszom erre az, hogy amikor elkezdtem foglalkozni a PHP-val akkor engem meglepett az, hogy minden könyvben az van leírva először, hogy hogyan telepítsük ezt a fordítót, szerintem ezzel egy kezdőt nem kellene traktálni, hiszen manapság annyi ingyenes és nem ingyenes szolgáltatót találunk, aki nyújt PHP szolgáltatást is (sőt már egyre többen elkezdték a MySQL szolgáltatást is nyújtani, az ingyenes szolgáltatók közül is). És egy kezdőnek éppen megfelel egy ilyen szolgáltatónál regisztrálva próbálgatni a szárnyait, mintsem az otthoni gépére kelljen felrakni különböző webszerver programokat, és azokkal vesződnie, illetve azok beállításával.

Szóval a PHP scriptet legáltalánosabban a HTML forrásban a következőképpen szoktuk beilleszteni:

```
<?php
...
?>
```

Én is ezt a formát fogom a továbbiakban használni, van más jelzésmód is, de ezek bemutatásával most nem foglalom a helyet, majd egy későbbi cikkben leírom azokat is.

Mielőtt még mélyebbre ásnánk magunkat magában a script készítésben, ejtenék pár szót a kiterjesztésről, vagyis hogy milyen kiterjesztésű lehet egy ilyen scriptet tartalmazó HTML oldal. Ezt a kiszolgáló szabja meg, illetve ha valaki magának állít be szervert, akkor az egy config fájlban állíthatja be, hogy mely kiterjesztéseknél fusson a php fordító. Ennek pontos beállítását a php readme-je tartalmazza. Általában a kiterjesztés *.php illetve egyes szolgáltatók szokták még használni a *.php3 illetve a *.html kiterjesztéseket is.

A script lényegében bárhova elhelyezhető a HTML forrásban. Ennyi bevezető után készítsünk egy nagyon egyszerű kis php kódot, amit a <body> </body> közzé illesztve máris kipróbálhatunk. A példa egyszerű: írassuk ki a weblapra a php segítségével, hogy Hello Világ! :

```
<?php
   echo('Hello Világ!');
?>
```

Ebből a pár sorból is kitűnik, hogy a php sok mindenben hasonlít más programozási nyelvekre. Azaz a sorvégi ; illetve a zárójelek használata, vagy a szöveget (string) jelölő ' jel, ami helyett lehet használni az ” jelet is, de ezt azért nem nagyon ajánlatos használni, mivel ha forrásrészletet szeretnénk beszúratni a php-val, akkor összekeveredhetnek a dolgok. Ha már úgyis itt tartok, akkor legyen a következő példa amelyben egy HTML forrásrészletet szúrunk be a php segítségével a lapra.

```
<?php
   echo('<br><font color=”FFFFFF”><b>Hello
Világ!</b></font>');
?>
```

Így most a Hello Világ! szöveg új sorba fehéren, és félkövéren fog kerülni. Ezzel bemutattam az echo parancs lényegét, persze lehet vele változókat is kiíratni, de ez majd később jön. Most előbb bemutatom, hogyan lehet még beilleszteni a forrásba egy másik fájl tartalmát:

```
<?php
   include('szoveg.txt');
?>
```

Az include al is lehet olyan fájlt beilleszteni, amiben van html forrás, sőt még php kódot is tartalmazhat. Ezzel nagyban meg lehet gyorsítani a weblapok készítését, hiszen ha egy szépen felépített weblapban csak a szöveg változik, akkor nem kell az egészet újra minden szöveghez megépíteni. Persze - mondhatják azok, akik már eléggé gyakorlottak a HTML készítésben de erre valók a frame-ek. Én viszont úgy gondolom, hogy a frame-ek kezelése bonyolultabb, hosszabb, és persze kevésbé elegánsabb, mint a php, de aki ennek ellenére a frame-eket szereti inkább használni, annak attól még nem kell figyelmen kívül hagynia az include eljárást, mivel ezt sok egyéb másra is lehet használni, ahol a frame-eket nem.

Ebbe a cikkbe nagyjából ennyi fért, tudom hogy eléggé kevés, és az ember nem szereti az elején abbahagyni, viszont most következne a változók témaköre, ami túl hosszú, és félbehagyni nem igazán jó. Ezért a következő rész nagyjából csak a változókkal fog foglalkozni, abban már sokkal több példa lesz ismertetve.

Csubák Péter Chuby@chello.hu
http://www.csu-ga.hu/chubyprograms

FOLYAMAT MONITORIZÁLÁS

Mikrovezérlő és számítógép segítségével. Első rész: Bevezető

Körülbelül tizenkét évvel korábban, amikor elõször láttam egy PLC-t vagy magyar nevén mikrovezérlõt, el nem tudtam képzelni, mi lehet az a “fekete doboz” mit tud és fõként a ***hogyan csinálja*** izgatott. Ugyanekkor kerültem érintés közelbe számítógépekkel, amirõl ugyan többet hallottam azelõtt és kíváncsiságot ébresztettek bennem de akkoriban ez még eléggé elérhetetlennek tûnt egy földi halandó számára.

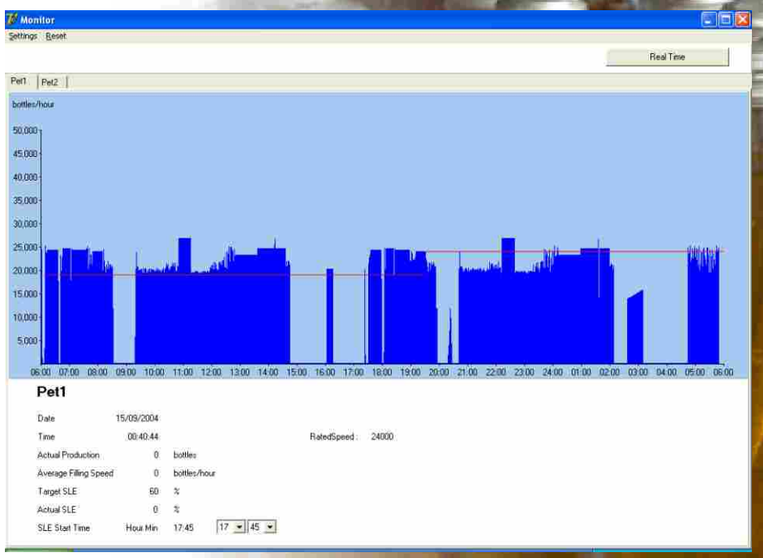
Az itt bemutatott rendszerben PLC figyeli a folyamatot a tárolt program szerint, a PC pedig vizualizálja az idõbeli történést, online információt képes nyújtani a felhasználónak a folyamat állapotáról, és szükség esetén a felhasználó beleszólhat a folyamatba.

Jelen munka egy csepp a tengerben, e témakörben, de hatalmas jelentõséggel bír számomra, ugyanis megtanultam, hogy ha az ember valóban akar valamit azt el is tudja érni, a számítástechnika alkalmazásában pedig legyen az PLC vagy PC alkalmazás csak a képzelet a határ. Már most rengeteg ötlet alakul ki hogyan lehet tovább fejleszteni “kis” alkalmazásomat hogy minél többre lehessen használni, minél többet nyújtson, de az új témájú ötletek sora is végtelen.

Egy palackozó üzemben dolgozom, ahol nagyon fontos a termelési hatékonyság. Ennek a mérése elméletileg nagyon egyszerû, meg van adva a névleges sebesség, amivel a töltõsornak termelnie kell, palack/órában, ismerjük a termelési idõ tartamát, ezt megszorozzuk egy cél hatékonyság értékkel, és az így megkapott elméleti számot összehasonlítjuk a valóban letermelt mennyiséggel. Ez mind szép, de amikor a célérték alatt teljesít a gyártósor, akkor megjelennek a kérdések, hogy mi ment rosszul, mik voltak az okok, mennyi idõ ment el erre, meg arra, de nem mindig sikerül megválaszolni érdemben ezeket a kéréseket mivel a folyamatról kapott információk nem elég részletesek, nem elég pontosak. Jelenleg az állásidõket a töltõgépkezelõ rögzíti „kézileg”. Na most képzeljük el, hogy van egy komoly meghibásodás, amit õ kell elhárítson, ez esetben minden gondja nagyobb, annál hogy megnézze az órát, hogy mikor ált le a sor, és mikor sikerült visszaindítani, megbecsüli az idõt, és ezt az értéket beírja a jelentésébe. Más estben vannak technológiai állások, íz váltás vagy méret átszerelés, ezekre léteznek úgymond standard értékek, amit õk automatikusan beírnak a jelentésbe, de mivel emberek, van aki jobban csinálja, van aki kevésbé jól. Egy másik jellegzetes hibaforrás soha nem jelenik meg a jelentésben, hogy a gépkezelõ hibája miatt nem dolgozott az elvártak szerint a termelõ sor. Az ilyen jellegû hibákat mindig ráírják más, általában könnyebben elfogadott okokra. Ez csak egy kis ízelítõ, abból hogy milyen torzulásokat szenved az információ, aminek alapján komoly döntések születhetnek.

A megfelelõ információ hiányában, nem tudjuk mit kell tennünk, ahhoz hogy javítani tudjuk munkánkat, nem tudjuk hol pazarolódnak el erõforrásaink. A gyenge hatékonyságú folyamat pedig jelen piaci körülmények között elmúlásra van ítélve. A versenyképesség megszerzése és megtartása érdekében ki kell használni a folyamatokban rejlõ lehetõségeket, ezt pedig csak úgy tudjuk elérni, ha tudjuk, hogy mi történik a folyamat lefolyása alatt.

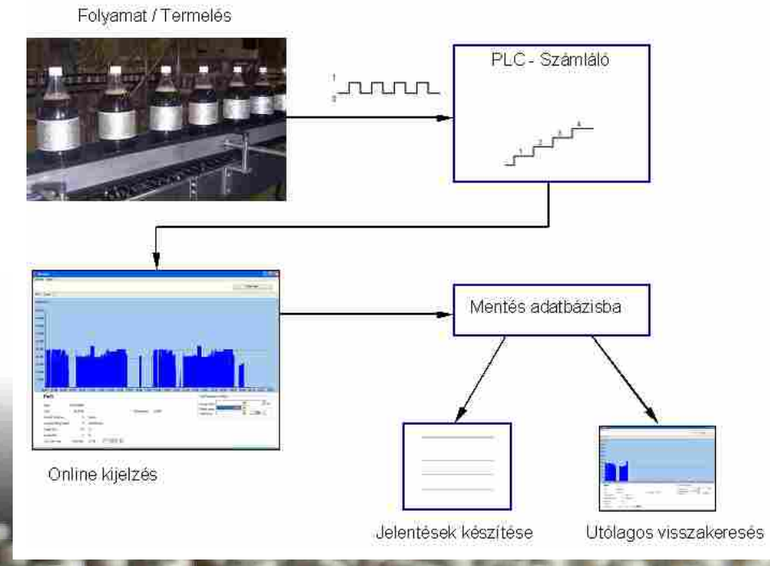
Egy elsõ megcélzott lépés az állásidõk pontos mérése. Ha egy egyszerû érzékelõvel, például egy fénysorompóval (fotocella) megszámoljuk a töltõgépbõl kijött megtöltött palackokat, idõegység alatt, mondjuk minden percben, ebbõl látjuk hogy elõször is hogy mûködik-e a sor vagy áll. A megszámolt palackok számából megtudjuk az idõpontig legyártott mennyiséget, töltési sebességet, átlag sebességet tudunk számolni, és már van is egy nagy mennyiségû értékes információnk és egy valós képünk a lezajlott vagy folyamatban levõ folyamatról. Ha ezt még ki is rajzoljuk grafikusan egy PC képernyõjére, például egy 24 órás idõtartamra akkor kapunk egyetlen pillanat alatt egy teljes képet a töltõ sor mûködésérõl, teljesítményérõl. A folyamatos grafikus kirajzolás (1. ábra) pedig valós idejû helyzetjelenést nyújt minden az érintetteknek a teljesítményrõl, megteremtve a lehetõséget a mielõbbi korrigálásra.



Jogosan merül fel a kérdés miért nem használunk egy a piacon már jelenlévõ erre a célra kifejlesztett szoftvert. Erre azt tudom válaszolni, hogy összekötöm a kellemest a hasznossal. Mint már leírtam, eltökélt szándékom hogy megismerjem mi van a „fekete dobozban” és megtudjam, hogyan mûködik. Erre ez legjobb módszer, ha magam csinálom. Ugyanakkor jelen van a saját fejlesztésû szoftver minden elõnye, mint testre szabottság, alacsony ár, bármikor lehet alakítani, fejleszteni. Természetesen az ezzel járó hátrányok is jelen vannak, de ezekbõl lehet sokat tanulni.

A 2. ábrán láthatjuk a rendszer elvi felépítését. A fotocella elõtt elhaladó palackok megszakítják a fénynyalábot, ezzel egy elektromos jelet generálnak, amely négyszögjel formájában eljut a PLC-be. A PLC-be be van programozva egy (vagy több) számláló, amelynek a tartalma a bemeneti jeltõl függõen növekszik. PC-s alkalmazásunk minden percben lekérdezi a számláló tartalmát és az így kapott adatot feldolgozza. A feldolgozás abból áll, hogy a mellékelt ábra szerint kirajzolja grafikusan a képernyõre a töltõsor (folyamat) állapotát, számítja és a kirajzolással egy idõben kiírja a számolt mérõszámokat és elmenti adatbázisba azokat az adatokat amelyek fontosak a késõbbi elemzés szempontjából. Késõbbi feldolgozás, elemzés lehet például egy hosszabb idõszak lekérdezése, hét, hónap év, vagy lehet mûszakokra mûszakvezetõkre lebontott jelentéseket elõállítani, stb. A következõ részekben kitérünk a felvázolt rendszer részeinek bemutatására, a PLC program megvalósítására, a kommunikáció megvalósítására a PLC és a PC között, a PC-s alkalmazás bemutatására. A PLC program a gyakorlatban használt Mitsubishi mikrovezérlõ saját programjában van megírva, a PC alkalmazáshoz pedig a Delphi fejlesztõkörnyezetet használtam amely lehetõséget nyújt a grafikus megjelenítésre és az ide tartozó adatbázis kezelésére is.

Elvi felépítés



Soós Csaba csoos@from.ro

Pythonra

Programozzunk Python nyelven III

Ebben a részben néhány egyszerűbb python scriptet mutatok be, majd egy-két gyakran használt algoritmus python megvalósítását fogjuk tárgyalni.

Biztosan sokan hallottak arról hogy az ember olvasáskor elsősorban az utolsó néhány karakterét figyeli meg a szavaknak, és az ezek között található karakterek sorrendje nem olyan fontos az olvashatóság szempontjából.

„Valuózínsleg ezt a szöevget is el tjdua osavlni mndneiki.”

Készítsünk egy olyan python scriptet ami a megadott file szavait beolvassa, és összekeveri bennük a betűket, majd kiírja az eredményt egy másik file-ba.

A python filekezelése:

Pythonban, csakúgy mint más nyelvekben, ahhoz hogy egy file-ból olvasni tudjunk előbb meg kell nyitnunk. Erre az open() függvény szolgál.
Az open() nek 2 string paramétere van, az első a file neve, a használt operációs rendszernek megfelelő elérési úttal. A második pedig a file hozzáférési mód,
hogy írásra, olvasásra vagy hozzáfűzésre akarjuk használni, a szöveg vagy bináris file-t.

Hozzáférési mód	Leírás
r	Csak olvasásra nyitja, meg. Ilyenkor írni nem tudunk a file-ba
w	Csak írásra nyitja meg.
a	Hozzáfűzésre nyitja meg, vagy a file végéhez pozicionál, és onnan kezdve írhatunk a file-ba.
rb	Binárisként kezeli a file-t, és olvasásra nyitja meg.
wb	Írásra nyitja meg a bináris file-t.
ab	Hozzáfűzésre nyitja meg, és binárisként kezeli.
r+	Olvasásra és írásra nyitja meg.
w+	Írásra és olvasásra nyitja meg.
a+	Hozzáfűzésre és olvasásra.
rb+	Olvasásra és írásra nyitja meg a bináris file-t.
wb+	Írásra és olvasásra nyitja meg a bináris file-t.
ab+	Hozzáfűzésre és olvasásra nyitja meg bináris file-t.

Ha egy már létezo file-t írásra nyitunk meg akkor a file tartalma elvész. Olvasásra csak létezo file-t nyithatunk meg.

Példa:

```
>>> f = open("c:\\boot.ini")
>>> f.readline()
'[boot loader]\n'
>>> f.close()
```

Vegyük észre hogy az open() egy objektumot ad vissza (OOP-rol bovebben a következo részben), aminek különféle metódusai vannak, amiket az objektum nevétol ponttal elválasztva érhetünk el. Ilyen a close() ami a file bezárására szolgál. Ha nem akarunk több muveletet végezni a file-lal akkor ne felejtsük el mindig bezárni a file-t.
Olvasásra szolgál a readline() metódus ami egy sort olvas, és a read() ami az egész file-t. Van egy másik hasznos olvasásra használható függvény, a readlines(), ami szintén az egész file-t olvassa be, de a sorokat egy tömbbe helyezi. Írni pedig a write() és writelines() metódusokkal tudunk.

Most pedig jöjjön a program.

A felhasználó parancssoros paraméterként tudja megadni a bemeneti szövegfile nevét. A kimeneti file pedig a bemeneti + „.out” lesz. Ezután megnyitjuk a file-t, ha valami hiba lépne fel, akkor azt lekezeljük, majd soronként beolvassuk, a sorok szavait összekeverjük, úgyhogy az elso néhány karaktert a helyén hagyjuk, és kiírjuk a kimeneti file-ba.

Karakterek felcseréléséhez készítsünk egy kis függvényt, ami a megadott indexu karaktert felcseréli.

```
def sXchg(s, a, b) :
 ca = s[a]
 s = s[:a] + s[b] + s[a+1:]
 s = s[:b] + ca + s[b+1:]
 return s
```

Majd visszatér az új stringgel.

Két trim-elo rutinra is szükségünk lehet, ami levágja a szó végérol vagy elejérol a szóköz, illetve sorvége karaktereket.

```
def leftTrim(s) :
 L = len(s)
 i = 0
 while i < L and (s[i] == ' ' or s[i] == '\n'
or s[i] == '\r') : i+=1
 s = s[i:]
 return s
```

Most pedig jöjjön a teljes forrás.

import random, sys # a sys a paraméterek kezeléséhez kell, a random pedig a random függvényhez

```
def sXchg(s, a, b) :
 ca = s[a]
 s = s[:a] + s[b] + s[a+1:]
 s = s[:b] + ca + s[b+1:]
 return s

def leftTrim(s) :
 L = len(s)
 i = 0
 while i < L and (s[i] == ' ' or s[i] == '\n' or
s[i] == '\r') : i+=1
 s = s[i:]
 return s

def rightTrim(s) :
 L = len(s)
 i = L - 1
 while i > 0 and (s[i] == ' ' or s[i] == '\n' or
s[i] == '\r') : i-=1
 s = s[:i+1]
 return s

# Ez keveri össze a karaktereket. Ha kisebb a szó
hossza 3-nál akkor persze nem #csinál semmit.
def mixWord(s) :
 if len (s) <= 3 : return s
 m = s
 i = 0
# ha a szó végén pont, vesszo vagy más a szóhoz
tartozó karakter lenne, azt #békén hagyjuk.
 while (not s[i].isalnum()) and i<len(s) : i+=1
 bi = i + 1
 i = len(s) - 1
 ei = 1
 while (not s[i].isalnum()) and i>0 :
  ei+=1
  i -=1
 if len(s) - ei - bi < 2 : return s
#ha túl sok ilyen karakter van és a cserélheto
rész hossza túl kicsi akkor #visszatérünk

#Ha a cserélheto rész túl hosszú akkor több
karaktert hagyunk meg az elején és a #végén
 if len(s)-ei-bi > 11 :
  bi += 2
  ei += 2
 elif len(s)-ei-bi > 7 :
  bi += 2
  ei += 2
#Ez pedig a kavarás, ami össze cseréli a
karaktereket.
 for i in range(bi, len(s)-ei) :
  r = random.randint(bi, len(s)-(ei+1))
  while r == i : r = random.randint(bi, len(s)-
(ei+1))
  if m[r] == s [r] :
   s = sXchg(s, i, r)
 return s

#Innen kezdodik a foprogram
# a sys.argv egy tömb amiben a parancssoros
paraméterek vannak.
if len(sys.argv)<2 :
 print "Usage: wordmix filename"
else :
 fname = sys.argv[1] #elso paraméter a filename
 print 'Opening',fname
 try:
  fi = open(fname, 'r')
  fo = open(fname+'.out', 'w')
 except Exception :
 #kivételkezelés ami a file megnyitás közben
fellépo hibákat kezeli le, errol #bovebben késobb
  print 'File not found'
  exit
 print 'Reading',fname
 print 'Writing',fname+'.out'
 line = fi.readline()
# a file beolvasása most ciklussal történik,
lehetett volna itt is a readlines() #metódust
használni de nagy fileoknál talán így elonyösebb.
 while line :
  words = line.split(" ")
  for i in words :
   w = i
   ends = ' '
   if w.find('\n') != -1 : ends = '\n'
   w = leftTrim(w)
   w = rightTrim(w)
   fo.write(mixWord(w)+ends)
  line = fi.readline()
 fo.close()
 fi.close()
 print 'Done.'
```

A továbbiakban egy-két gyakran használt algoritmust fogok bemutatni.

Gyakran használt algoritmusok:

Ilyen például a rendezés vagy a keresés. Habár a legtöbb nyelvnél van arra is lehetoség hogy beépített függvényeket használjunk erre, de azért illik ismerni ezeket az eljárásokat.

Keresés:

Ha egy tömbben vagy más lineáris adatstruktúrában meg szeretnénk keresni egy elemet, akkor egyszeruen végig kell gyalogolnunk az adatfolyamon, és összehasonlítani az aktuális elemet a keresettel. Ez az ún. lineáris keresés, ami nagy tömböknél elég lassú. Ha rendezett a tömbünk, vagyis növekvo, vagy csökkeno számokat tartalmaz, akkor használhatunk bináris keresést is. Ezzel a tömb rendezettségét kihasználva, gyorsabb eredményre jutunk mint a lineáris kereséssel. Ilyenkor megfelezzük a tömböt, és eldöntjük hogy a keresett elem, a tömb melyik felében található (ha egyáltalán benne van). Majd azt a felét megint megfelezzük addig amig már csak egy elem maradt.

Ennek python megvalósítása:

```
t = [5, 8, 10 ,20 ,36 ,48 ,157 ,651]
def binfind(array, n) :
 height = len(array)-1
 low   = 0
 done = 0
 while not done and low<=height:
  mid = (height + low) / 2
  if n < array[mid] :
   height = mid -1
  elif n > array[mid] :
   low = mid + 1
  else : done = 1
 if array[mid] == n :
  return mid
 else : return -1

print binfind(t, 157)
raw_input("ok")
```

A kimenete:
6
ok

Nem szabad elfelejtenünk hogy ez a keresés csak rendezett tömbökre használható, és az sem mindegy hogy növekvo vagy csökkeno a tömb.

Rendezések:

Ilyen algoritmusokból is létezik párfajta, vannak lassabbak és gyorsabbak. Az egyik legegyszerubb, bár elég lassú rendezés, az úgynevezett buborék rendezés. Ennél a rendezésnél végiggyalogolunk a tömbön, és ha az aktuális elem kisebb (vagy nagyobb, attól függ hogyan rendezzük), az akuális + 1-edik elemnél (tehát a szomszédos elemeket hasonlítgatjuk) akkor megcséréljük a kettot. Az egészet addig folytatjuk amig egyetlen egy elemet sem kell megcserélni.

Megvalósítása pythonban:

Fontos hogy ne tuple tömböt használjunk hanem listát, hogy meg lehessen változtatni a tömbelemket. Régebbi python verziók még nem támogatták a boolean típust, ezért használok inkább int-et logikai típusként.

```
def bublesort(array) :
 done = 0
 end = 1
 while not done :
  done = 1
  for i in range(len(array)-end) :
   if array[i]>array[i+1] :
    done = 0
    array[i],array[i+1] = array[i+1], array[i]
  end += 1
 return array
t = [24 ,45, 5 ,11 ,4 , 7, 13,]
bublesort(t)
print t
raw_input("ok")
```

Kimenete:
[4, 5, 7, 11, 13, 24, 45]
ok

Az end változó azért van hogy feleslegesen ne mennyünk végig az egész tömbön ha a vége már úgyis rendezett, ugyanis minden ciklusban a legnagyobb elemet vegigcipeljük magunkkal a tömbvégére.

Nézzünk egy más elven muködo algoritmust, a beszúrós rendezést. Ennél a rendezésnél, a kiválasztott elemet, a tömb megfelelo helyére szúrjuk be, közben a tömböt mindig “elscrollozzuk” eggyel.

Python implementációja:

```
t = [24 ,45, 5 ,11 ,4 , 7, 13,]
def insertsort(array):
 for rIndex in range(1, len(array)):
  rValue = array[rIndex]
  iIndex = rIndex
  while iIndex > 0 and array[iIndex - 1] >
rValue:
   array[iIndex] = array[iIndex - 1]
   iIndex = iIndex - 1
  array[iIndex] = rValue

insertsort(t)
print str(t)
raw_input("done")

Eredmény:[4, 5, 7, 11, 13, 24, 45]
done
```

Most pedig jöjjön az egyik leggyorsabb a quicksort, vagyis a találóan elnevezett gyorsrendezés.

Ez egy úgynevezett rekurzív rendezés, a rekurzió tulajdonképpen azt jelenti hogy a függvény saját magát hívja meg.

Pl.

```
def fakt(n) :
   if n>1 : return n * fakt(n-1)
   return n
```

Ez egy faktoriális számoló rutin.

A rekurziónak van némi veszélye, de ez védett módú operációs rendszeren nemigen jön elo, ugyanis a rekurziónál használt vermet az oprendszer képes futásidoben megnövelni. Ennek ellenére a nagyon mély rekúrzió használata kerülendo. A valós módú operációsrendszerek idejében egy végtelen, vagy túl mély rekúrzió sokkal komolyabb gondokat tudott okozni mint egy sima végtelen ciklus. Ezért most is érdemes jobban odafigyelni az ilyen rutinokra.

Egy másik rekurzív rutin, a fibonacci sorozat eloállítására:

```
def fib(n):
  if n < 2:
    return n
  else:
    return fib(n - 2) + fib(n - 1)
print fib(20)
```

Gyorsrendezésnél kiválasztunk egy elemet a tömbbol, és minden egyes maradék elemere megnézzük hogy a kiválasztottnál kisebb vagy nagyobb. A nagyobbakat a kiválasztott elem egyik oldalára, a kisebbeket a másik oldalára helyezzük, majd mindkét oldalon végrehajtjuk ezt a rendezést rekurzívan amig rendezett tömböt nem kapunk. Ez pedig a gyorsrendezés pythonban:

```
def partition(array, start, end):
 while start < end:
  while start < end:
   if array[start] > array[end]:
    (array[start], array[end]) = (array[end],
array[start])
    break
   end = end - 1
  while start < end:
   if array[start] > array[end]:
    (array[start], array[end]) = (array[end],
array[start])
    break
   start = start + 1
 return start

def quicksort(array, start=None, end=None):
 if start is None: start = 0
 if end is None: end = len(array)
 if start < end:
  pivot = partition(array, start, end-1)
  quicksort(array, start, pivot)
  quicksort(array, pivot+1, end)


t = [24 ,45, 5, 54 ,11 ,4 ,32, 7, 13,]
quicksort(t)
print t
input()
```

Eredmény: [4, 5, 7, 11, 13, 24, 32, 45, 54]

A következo részben az objektum orientált programozással fogunk foglalkozni.

Magyar Attila zeroflag@elte.hu

Vigyázat! Botok!

Napjainkban tömegesen bukkannak fel a vírusok a neten, ha egy nap nem fedeznek fel újat, akkor mindenki meglepődik.

Az IRC-én van egy olyan kifejezés, amibe lépten-nyomon belebotlunk. Ez a szó a következő: bot. A robot rövidítéséből származik. Egyszerűen egy automatizált IRC klienst jelent, ami bizonyos eseményekre reagál.

Hogyan kapcsolható össze a két dolog? Nagyon könnyen. Gondoljuk csak el, hogy valaki egy sereg gépet szeretne irányítani, amelyeket valamire fel szeretne használni. Persze azt akarja, hogy egyszerre cselekedjenek. Itt jön képbe az IRC, mint ideális felület. Ha egy csapat botról van szó, amik egy meghatározott csatornán tömörülnek, akkor egyetlen szó beírásával, mintegy parancsszóra nekiindul a sereg, és teszi a dolgát.

Sokan alábecsülik ezeknek a jelentőségét, mondván, hogy ezek csupán egyszerű vírusok vagy hátsó kapuk. Viszont ha jobban a dolgok mögé nézünk, akkor megdöbbentő tényeket fedhetünk fel. A vírusos gépek számáról készülnek felmérések, mindenki tisztában van a veszéllyel. Az ilyen „alvó bestiák” számáról egészen sokáig semmiféle adat nem volt. Napjainkban viszont, amikor számuk hihetetlenül megnőtt, már kellő figyelmet fordítanak erre a problémára is.

A CERT egyszer, különös késztetést érezve utánanézett a dolognak. Pár napos munkával találtak két olyan bot-hálózatot, amik több ezer gépet foglaltak magukban. Erről írtak is, közzétették, mit tapasztaltak, és kértek mindenkit, hogy ha lehet, tegyen ez ellen. Ez volt talán az első ilyen hivatalos felmérés.

Az év elején egy amerikai IRC szolgáltatót zártak be, mert a tulajdonos egy DDoS-bérlő rendszert működtetett. Több embernek is fizetett azért, hogy azok a saját 5-10 ezer gépből álló hálózatukkal ellehetetlenítsék a konkurencia Internet elérését és megjelenését.

Nemrégiben egy norvég Internet szolgáltató (a Telenor) emberei találtak és hatástalanítottak egy tízezer botból álló hálózatot. A hálózatot arra használták, hogy DDoS támadást indítsanak weblapok ellen.

A Symantec az év elején napi 2000 botot futtató gépet fedezett fel. Júniusban ez a szám már 30 ezer volt naponta! Ez természetesen nem azt jelenti, hogy ezek egy kézben vannak. De mindenképpen elgondolkodtató a számuk növekedése.

Egyre több hasonló botnet van, amelyek „zombi” gépekből állnak össze. Ezeket a gépeket valamilyen módon ellátják egy bot programmal, ami a felhasználó számára szinte észrevétlenül lapul, de a megfelelő parancsokkal aktivizálni lehet. Ezeknek a programoknak a terjedését nagyon elősegíti a sok trójai vírus és biztonsági rés, amiknek a segítségével bejutnak az adott gépre.

A rosszindulatú botok általában a rendszer ismert sebezhetőségeit használják fel arra, hogy bejussanak a gépbe. A legismertebb botok szinte napról napra változnak, mindig újabb támadási lehetőségekkel látják el „valakik”.

Egy az ilyen ismertebb botok közül például a Gaobot-család, amik több ismert Windows sebezhetőséget kihasználva terjednek. Megjegyzem ezeknek a sebezhetőségeknek nagy részére már létezik javítócsomag, csak nem mindenki használja ki a lehetőséget, hogy feltelepítse.

Ha bekerült a gépbe, akkor szintén több fronton „jeleskedik”. Mondhatni teljesen szokásos, hogy DoS támadást lehet vele véghezvinni. Ezen kívül a változatai nagyon sokrétűek: némelyik játékok kulcsait próbálja ellopni az áldozat gépéről, némelyik FTP vagy email szervert tartalmaz, némelyik elmenti a felhasználó billentyűleütéseit, más pedig azzal kezdi tevékenységét, hogy az ismertebb antivírus-programok működését lehetetlenné teszi. Ez csak pár kiragadott példa volt sajnos, ennél sokkal veszélyesebbek.

A védekezés ellenük nem kíván sok erőfeszítést. Ha minden nap csak öt percet törődünk ezzel, máris védettnek, vagy legalábbis jobban védettnek mondhatjuk magukat. Ha csak annyit teszünk, hogy az operációs rendszerünket rendszeresen frissítjük, máris sokat tettünk. Ha ezután felteszünk még valamilyen antivírus programot is (megéri rászánni azt a kis összeget) esetleg a gépünkre valamilyen PFW (Personal FireWall) programot is telepítünk, akkor nyugodtabban aludhatunk, mert a mi gépünkön kevesebb támadási felület van, mint egy átlagos felhasználói gépen.

Ez a helyzet ma. Hogy mi lesz holnap, senki sem tudja. Újabb operációs rendszer sebezhetőségeket fedeznek majd fel, a botok ezeket kihasználják, egyre több dologra lesznek képesek… ez mind csak feltevés, de nagyon reális. Egy azonban teljesen biztos: a számuk növekedni fog.

Többször említettem a cikkben a **DoS**-t. Szerintem manapság már mindenki tisztában van vele, mit jelent. Ha mégsem, akkor röviden: a Denial of Service rövidítése. Olyan támadások gyűjtőneve, amiknek az a célja, hogy egy szolgáltatás vagy egy gép ne tudja elvégezni azt a munkát, amire szánták. Webszerverek esetén ne tudja a honlapot megmutatni, FTP esetén ne tudja a felhasználó elérni, „mezei” felhasználó esetén ne tudjon az internetre kapcsolódni. Ezt általában úgy érik el, hogy a megtámadott gépet olyan mennyiségű vagy minőségű adattal árasztják el, amitől az vagy teljesen válaszképtelen lesz, vagy a kapott adatok feldolgozása tölti ki minden idejét. A DDoS támadás a DoS egy speciális fajtája, amikor a támadás nem egy gépről érkezik, hanem több gépről egyszerre, megosztva (Distibuted Denial of Service). Az ilyen támadások sokkal hatékonyabbak és sokkal gyorsabban célt érnek a támadók.

Balla Imre - ircmania@talpak.org

Tippek & Trükkök I.

A programozók általában szeretik maguk megoldani a problémákat, de vannak bizonyos dolgok, amiket nem túlságosan jó mindig újra megírni, ezeket általában már mások megoldották. Ezek közül fogom bemutatni először a hardware-rel kapcsolatos ilyen trükköket.

CD meghajtó ajtó kinyitása, és becsukása

Az első fontos dolog az, hogy a uses-hez adjuk hozzá a MMSystem-et.Majd a meghívó objektum valamelyik eseményéhez a következőket kell hozzáadni: (kinyitás)

```
mciSendString('Set cdaudio door open wait', nil, 0, handle);
```

A becsukáshoz pedig:

```
mciSendString('Set cdaudio door closed wait', nil, 0,
handle);
```

A parancsok a windows-hoz jutnak el, és mivel a windows multimédia vezérlőjét nem fordították le „a velejéig”, ezért a parancsokat magyar windows-on is angolul kell elküldeni.

A processzor sebességének a megállapítása

A processzor aktuális sebességét egy függvény fogja megmondani nekünk, ami a következő:

```
function TForm1.CpuSpeed: Extended;
var
  t: DWORD;
  mhi, mlo, nhi, nlo: DWORD;
  t0, t1, chi, clo, shr32: Comp;
begin
  shr32 := 65536;
  shr32 := shr32 * 65536;
  t := GetTickCount;
  while t = GetTickCount do begin end;
  asm
    DB 0FH
    DB 031H
    mov mhi,edx
    mov mlo,eax
  end;

  while GetTickCount < (t + 1000) do begin end;
  asm
    DB 0FH
    DB 031H
    mov nhi,edx
    mov nlo,eax
  end;
  chi := mhi; if mhi < 0 then chi := chi + shr32;
  clo := mlo; if mlo < 0 then clo := clo + shr32;
  t0 := chi * shr32 + clo;
  chi := nhi; if nhi < 0 then chi := chi + shr32;
  clo := nlo; if nlo < 0 then clo := clo + shr32;
  t1 := chi * shr32 + clo;
  Result := (t1 - t0) / 1E6;
end;
```

Amit mondjuk egy label-be a következőképpen tudunk átadni:

```
label1.Caption := FloatToStr(CpuSpeed) + ' MHz';
```

A meghajtók fajtáinak meghatározása

A meghajtók fajtái a következőképpen határozható meg:

```
var
  x : char;
  DrvType   : Integer;
  DrvLetter,
  DrvString : String;
begin
  ListBox1.Clear;
  for x := 'A' to 'Z' do
  begin
    DrvLetter := x +':\';
    DrvType := GetDriveType(pChar(DrvLetter));
    case DrvType of
      0,1             : DrvString := '';
      DRIVE_REMOVABLE          : DrvString := 'Eltávolítható
lemez';
      DRIVE_FIXED      : DrvString := 'Fix lemez';
      DRIVE_REMOTE     : DrvString := 'Hálózati meghajtó';
      DRIVE_CDROM      : DrvString := 'CD-ROM meghajtó';
      DRIVE_RAMDISK    : DrvString := 'RAM meghajtó';
    end;
    if DrvString <> '' then
      Listbox1.Items.Add(DrvLetter +
                  ' = ' + DrvString);
    end;
  end;
```

A meghajtók neveikkel és fajtáikkal egy listbox-ba kerülnek.

A következő cikk:

A következő cikkben a grafikával kapcsolatos trükkökből és tippekből fogok szemezgetni.

Csubák Péter Chuby@chello.hu

Internetszerver

SZABAD SZOFTVEREKKEL

Sok cég és intézet azért nem valósítja meg belső számítógépes hálózatának az Internetre való biztonságos kapcsolódását, mert drágának és komplikáltnak tartják, vagy nagy összegeket költenek arra, hogy ennek megoldását egy külső cégre bízzák. Nincs annál olcsóbb és karbantarthatóbb megoldás, mint hogy saját magunk építünk szervert. Ahol a kiadásokra az átlagosnál sokkal jobban oda kell figyelni, mint például az egészségügyi intézményekben, ott kimondottan ajánlott szabad szoftvereket alkalmazni Internet-szerver építésére.

A cikksorozat célja, hogy bemutassa, amennyire lehet viszonylag egyszeruen hogyan lehet olcsón és biztonságosan egy cég vagy intézet belső számítógépes hálózatának minden munkaállomásáról az Internetre kapcsolódni, és a külvilág számára csak a kívánt információkat megosztani. Ehhez a biztonságáról, karbantarthatóságáról és szabadságáról híres Linuxot használjuk. A cikksorozatot ajánljuk mindazoknak, akik már ismerik ezt a nagyszeru operációs rendszert, de konkrét megoldásra még nem alkalmazták.

A szerver követelményei, jellemzői:

A cikksorozat végére egy olyan szervert kapunk, amely minden átlagos igényt kielégít, a belső számítógépes hálózatnak Internet-megosztást, e-mail-szolgáltatást, valamint a meghatározott karbantartók számára titkosított parancssori elérést biztosít, illetve a külvilág számára is webkiszolgálást és FTP-szolgáltatást tesz lehetővé, s amely a belső számítógépes hálózattól leválasztott, csomagszurő tűzfallal, lemezkvóta-rendszerrel, vírusirtóval és levélszemétszurővel rendelkezik.

A szerverhez bármilyen mai PC megfelelő, ajánlott RAID-1-es alaplap vagy kártya használata két merevlemez mellett, és az egyik merevlemezt tükrözni a másikra, így bármilyen merevlemez-meghibásodás esetén az adatokat a tükör-merevlemezről rövid idő alatt, viszonylag kevés leállási idővel visszaállíthatjuk. E mellett egy CD- vagy DVD-író használata is ajánlatos, aminek segítségével meghatározott időközönként lementhető a teljes rendszer.

A szerver két hálózati kártyát tartalmaz, az egyik a belső számítógépes hálózattal, a másik a külvilággal (az Internettel) tartja a kapcsolatot. A belső kártyának belső IP-címmel, a külső kártyának az Internet-szolgáltatónktól igényelt külső IP-címmel kell rendelkeznie. Szükséges egy tartománynév, amelyet az Internet-szolgáltatónktól kérhetünk.

A szerver szolgáltatásai:

A tűzfal a Linux biztonsági szemléletének köszönhetően magában a Linuxban, az operációs rendszer kerneljében (rendszermagjában) helyezkedik el. Ez a lehető legbiztonságosabb megoldás. A 2.2-es kernel ipchains nevu stabil csomagszurője látja el az alapvédelmet. A csomagtípusokat a megadott ki- és bemenő forgalomra (IP-címekre vagy címtartományokra) szabályozza, szükség szerint beengedi, továbbítja, naplózza, visszadobja vagy elutasítja. A csomagszurő tuzfal egy alap védelmi rendszer, további védelmet a kiszolgáló-programok saját beállított védelmi rendszerei nyújtanak.

A tűzfal szemlélete: ***"Mindent tilos, amit nem szabad!"*** Ez a biztonság első lépcsője. Ezáltal sokkal könnyebben módosítható a tuzfalszabályzat, nem kell a tiltásokkal külön foglalkozni.

Az átjárás (forward) tiltott, csak a belső hálózatról kifelé engedélyezett, vagyis az Internetkapcsolódás csak és kizárólag a belső hálózatról történik, a belső hálózaton lévő munkaállomások csak a szerveren keresztül kommunikálhatnak a külvilággal. A ki- és bemenő csomagok csak szabványosak lehetnek, üzenetszórást, címhamisítást, csomageláraszást, pásztázást nem engedélyez, ezzel alapjaiban meggátolva az illetéktelen hozzáférést és leterhelést (többek között a DoS-támadást). A névkiszolgálást a belső hálózatról kifelé engedélyezi, és befelé csak a szerverre. A normál és az SSL-titkosítású webkiszolgálást a belső hálózatról kifelé engedélyezi, és befelé csak a szerverre engedélyezi.

Az FTP-kiszolgálást a belső hálózatról kifelé engedélyezi, és befelé csak a szerverre engedélyezi. Az SSH-kiszolgálást a belső hálózatról kifelé engedélyezi, és befelé csak a megadott karbantartói gépekről a szerverre engedélyezi. Az e-mail-küldést és -fogadást a belső hálózatról kifelé engedélyezi, befelé csak a szerverre engedélyezi. Minden más szolgáltatás tiltott, minden más szolgáltatást, külön meg kell adni.

A fájlrendszer-szintu **lemez-quota** amely a rendszermag része megakadályozza azt, hogy a szerverre lépett bármely felhasználó illetéktelenül lefoglaljon tárterületet. Minden, a szerverre belépési jogosultsággal rendelkező felhasználónak megadott tárterület áll rendelkezésére, amelyet túllépni nem lehet.

A **DNS-kiszolgáló** (névkiszolgáló) az Internet-szolgáltatónk szerverével együttműködve lehetővé teszi, hogy az Internet-szerverre, valamint minden egyes Internetre kapcsolt, névvel bejegyzett számítógépet a nevükre és másodneveikre hivatkozva, s ne csak az IP-címeikkel lehessen elérni. Az Internetről érkező kéréseket csak a saját tartományunkra engedélyezi, ezáltal meggátolva azt, hogy más hálózatokról illetéktelenül igénybe vegyék és leterheljék az Internet-szerver névkiszolgálását.

A **DHCP**-kiszolgáló (automatikus címkiosztó) a belső hálózat munkaállomásai számára oszt ki csak belülről látható IP-címeket, ezzel biztosítva, hogy a külvilág felol közvetlenül senki se férjen a belső hálózat munkaállomásaihoz és egyéb esetleges belső szerveréhez. Az Ipcímek esetleges illetéktelen lefoglalása és a szerver leterhelése ellen egy-egy IP-címet megadott ideig ad ki. A belső hálózat esetleges többi belső szerverének fix, csak és kizárólag belülről látható IP-címeket adhatunk ki, így azokat mindig ugyanazon a címeken érhetjük el.

A **webkiszolgáló** a honlapunk elérését teszi lehetővé mind a belső hálózat, mind a külvilág számára. A biztonság alapja, hogy minden weblap-látogató a lehető legminimálisabb joggal rendelkezik a webfelület elérésére.

A **levelező-kiszolgáló** a belső számítógépes hálózat felhasználóinak nyújt belső és külső e-mailszolgáltatást (küldést és fogadást egyaránt). A levélfiókokat beépített szurő, és külön csatlakoztatott levélszemét- és vírusszurő védi.

A **levélszemét-szurő** a levelező-kiszolgálóval közösen szuri a levélszemeteket különböző szempontok szerint (saját és az Interneten fellelhető adatbázisok segítségével).

A **vírusszurő** a levelező rendszerrel együttmuködve ellenőrzi, és vírustalálat esetén törli a leveleket és csatolt állományait. A tevékenységről értesíti a feladót, jelezve azt, hogy tanácsos leellenőriznie a számítógépét. A vírusirtó adatbázisa napi több alkalommal is automatikusan frissül az Internetről, ezáltal biztosítva azt, hogy a legújabb különböző operációs rendszerekre készített vírusokat is felismerje.

Az **FTP-kiszolgáló** azoknak, akiknek a munkájához szükséges, lehetővé teszi adatállományok Interneten keresztüli cseréjét megadott könyvtáron keresztül. A biztonságos kapcsolatot az nyújtja, hogy a felhasználók csak saját könyvtárukba zártak, onnan nem tudnak kilépni, s azt is a lehető legminimálisabb jogú közös felhasználóként tudják elérni (írni és olvasni).

Az **SSH-kiszolgáló** munkaállomásról történő belépést biztosít a szerverre, a lehető legnagyobb biztonsággal, SSL titkosított csatornán keresztül. A hozzáférés korlátozott, a belépés csak és kizárólag a megadott IP-címu karbantartói gépekről lehetséges, csak korlátozott jogosultságú, jelszóval védett felhasználóval.
A szerverről elég ismeretekkel rendelkezünk ahhoz, hogy a következőkben a beállításokat és szolgáltatásokat konkrétan megvalósítsuk. A következő cikkben bemutatásra kerülnek az operációs rendszer alapbeállításai, a számítógép hálózati kiszolgálásra való felkészítése.

Szucs Tamás **szucs_t@freestart.hu**