Felhasználói felület fénysebességgel

Nem is olyan régen még a programozók rémálma volt a felhasználói felület elkészítése. A program sokszor már régóta működött, amikorra a programozó rászánta magát a kezelőfelület elkészítésére.

Ez mára némiképp megváltozott, ugyanis a fenti feladat egyszerűsítése érdekében gomba módra szaporodtak és ma is szaporodnak a különböző eszközkészletek (tool kits). Ebben a cikkben egy olyan eszközkészletet mutatok be röviden, ami méltánytalanul kevéssé ismert, holott sok előnyös tulajdonsága van, sok mindenben talán jobb, de biztosan más, mint a többiek.

A türelmetlenek számára előrebocsátva: az FLTK-ról (Fast Light Tool Kit) lesz szó. Előbb azonban ejtsünk néhány szót általában az eszközkészletekről. Ami közös bennük: mindegyik biztosít a programozó számára olyan elemeket, amelyek a felhasználói felületek összeállítására általánosan használatosak. A teljesség igénye nélkül a legelterjedtebbek: gombok, menük, ablakok, beviteli mezők, gördítősávok, eszközsávok. Ezeket használhatjuk a készlet által megadott formában, vagy ha az illesztőfelülete objektumalapú, akkor saját típust képezhetünk belőlük, és ezt alkalmazhatjuk a későbbiekben.

Nézzük tehát, milyen lehetősége van a programozónak, ha a „lényegi" munkát végző programjához egy kellemes, könnyen kezelhető felületet akar biztosítani. Mi az, amiben a fent említett elemek jelentős része benne van, mindenütt hozzáférhető, könnyen használható? Igen, a HTML alapú, böngészőben futó felületre gondolok! Alkalmazzák is a programozók elég széles körűen, mivel nagyon kevés munkával lehet működő felületet biztosítani. Előnyei: egyszerűség, rendszerfüggetlenség, hiszen bármely naprakész böngészőben működik, ily módon hordozható, végül: hálózati megoldásra is alkalmas. Szép számmal akadnak hátrányai is: a hatékony használathoz szükség van egy webkiszolgálóra, lassú, a programozó keze erősen meg van kötve a böngészők behatárolt és egymástól eltérő képességei miatt, pont akkor válik egyre kevésbé hordozhatóvá, ha olyan eszközöket használunk, amelyek bővítenék a felület képességeit, elsősorban a JavaScript-és a Java-betétekre gondolok.

Aki ennél hatékonyabb eszközkészletet szeretne, de mégis tart a „komoly" programozási nyelvektől, az feltehetőleg valamely parancsnyelvet és a hozzá illesztett eszközkészletet választja. A leggyakoribb a Perl/Tk kapcsolat. Itt, mivel ez a Perl nyelven keresztül egy teljes eszközkészletet tesz elérhetővé, szinte minden olyan lehetőségünk megvan, ami más készletek esetén létezik. Fő hátrány ebben az esetben a lassúság, valamint hogy a Perlnek és mellette a Tk-nak, valamint a hozzá kapcsolódó moduloknak mind a fejlesztői, mind a futtató gépen fenn kell lenniük, valamint hogy a forrás a parancs sajátságaként kényszerűen „nyílt". Ugyanez a Tk különben a Tcl nyelvhez illesztve is megtalálható, számomra kissé érthetetlenebbül, de biztosan akad, aki tud Tcl-ben dolgozni…

Más parancsnyelveket (PHP stb.) nagyvonalúan átugorva, rátérek az „igazi" programozási nyelvekre. Ez a legtöbb Linuxban dolgozó programozó számára a C, újabban a C++. Természetesen ezekhez a nyelvekhez is vannak eszközkészletek. Sok esetben az eszközkészletek valamely feladat megalkotására, valamely program megírásának a céljával, azt segítendő jönnek létre. Ilyen például a Gimp–Gtk. Ehhez közeli a KDE–Qt páros, ahol az ablakkezelő a Qt könyvtáraira épül, és ezt az eszközkészletet használva fejlesztik. Más irányból indulva ugyan, de windowsos felületen ilyen az MFC C++ készlet is. Azért más egy kicsit, mert itt előbb volt meg a rendszer alapja Pascal, később C nyelven, az eszközkészletet csak utóbb írták meg, immár elméleti szempontokat is figyelembe véve. Ez látszik is rajta. Helyben is vagyunk, ugyanis a két utóbbi példát tekintve két C++-eszközkészletet láthatunk, és ezek legfőbb hibáját a túlzott bonyolultságban, a túl magas szintű (elvonatkoztatási szintről van szó, nem minőségiről) elemkezelésben látom. Ezeket az eszközkészleteket használva egy-egy párbeszédablak elkészítéséhez valóban grafikus fejlesztőeszközre van szükség. Emellett eseménykezelésük is elég bonyolult és erőforrás-igényes.

A fentieket ***Bill Spitzak***, az FLTK eredeti fejlesztője már régen felismerte. Célul egy egyszerű, gyors, könnyen kezelhető, logikus felépítésű, szerény erőforrás-igényű eszközkészlet kifejlesztését tűzte ki. Ebben munkaadója, a Digital Domain egy ideig támogatta, így született meg az FLTK első változata. Azóta a hivatalos támogatás megszűnt, a programcsomag LGPL felhasználási szerződés alatt került kibocsátásra. Jelenleg egy önkéntes csapat fejleszti, akik Interneten keresztül tartják egymással a kapcsolatot. Nézzük meg a legfontosabb tulajdonságait:

- Az LGPL felhasználási szerződés hatálya alá tartozik, azzal a záradékkal, hogy a program statikusan is összeépíthető, anélkül, hogy a forrásszöveget nyilvánosságra kellene hozni. Így üzleti célú programokban is használható.
- C++-osztályok, könnyű memóriakezelés.
- A jelenleg megszokotthoz képest rendkívül kis méret, egy minden elemet tartalmazó program mérete is mindössze 600 kilobájt körül van, statikusan szerkesztve is!
- Moduláris felépítés, a könyvtárból csak azok a részek kerülnek a programba, amelyek szükségesek; nincsenek elemközti kapcsolatok.
- Linux- (Unix, X11), Windows-, Mac-változatban is elérhető, azaz ugyanaz a forrásállomány bármelyik operációs rendszerre lefordítható.
- GNU, Borland, Microsoft stb. fordítóprogramokkal is lefordítható.
- Egyszerű és hatékony eseménykezelés: az egyes elemek eseménykezelő függvényükben „nyilatkozhatnak", hogy szükséges-e számukra az esemény, és amennyiben nem, a rendszer továbbküldi azt a következőnek.
- Beépített OpenGL-támogatást tartalmaz.
- Interaktív, grafikus felhasználói felületszerkesztő, C++-kódot állít elő, ami utólag szerkeszthető.
- Gyors és hordozható 2D-s grafikus függvények.
- Kiváló, pdf formátumú leírással bír (sajnos csak angol nyelven).

```
include ../makeinclude
clean:
      -@ rm -f $(ALL) *.o core *~


# a lØnyeg...
nezoclass.o: nezoclass.H
nezo$(EXEEXT): nezo.o nezoclass.o
../lib/$(IMGLIBNAME)
      echo Linking $@...
      $(CXX) -I.. $(CPPFLAGS) $(CXXFLAGS)
      ➥nezo.o nezoclass.o -o $@
      ➥$(LINKFLTKIMG) $(LDLIBS)
```

```
#include <stdlib.h>
#include <stdio.h>
#include <FL/Fl.H>
#include <FL/Fl_File_Chooser.H>

#include "nezoclass.H"

Kep_Nez_Wnd *kep_nez_wnd=(Kep_Nez_Wnd*)0;
Fl_File_Chooser
*kep_opener=(Fl_File_Chooser*)0;


void init()
{
  fl_register_images();
  Fl_File_Chooser::add_favorites_label=
  ➥"HozzÆad a kedvencekhez";
...
...
}

int main(int argc, char ** argv) {
 init();
 ::kep_nez_wnd=new Kep_Nez_Wnd();
 ::kep_nez_wnd->loadkep("logo.png");
 ::kep_nez_wnd->show();
 Fl::run();
 }
```

Hogyan szerezhetjük be és bírhatjuk munkára? Ha a csomag nincs Linux-terjesztésünk lemezein, akkor megtalálhatjuk a 44. CD magazin/fltk könyvtárában, vagy töltsük le a ➲ http://www.fltk.org címről. Célszerű a legfrissebb üzembiztos változatot beszerezni. Ez a változat a cikk írásakor az 1.1.2-es volt. CVS-en keresztül a fejlesztői változat is elérhető, ez jelenleg a 2.0-s. Számos ígéretes újdonságot mutat fel, de egyelőre még befejezetlen. Az egész forrás másfél megabájtot tesz ki, tehát modemes kapcsolattal is kényelmesen letölthető. Ha megvan a csomag, bontsuk ki a

```
# tar xvzf fltk-1.1.2-source.tar.gz
```

paranccsal. Ekkor az ***fltk-1.1.2*** könyvtárba kerülnek az állományok. Ha fejlesztőeszközeinket megfelelően telepítettük, ebben a könyvtárban a

```
# ./configure
# make
```

parancsok kiadása után máris indul a fordítás. Ha a munka hibajelzés nélkül elkészült, a ***test*** könyvtárban találhatók a próba- és bemutatóprogramok, amelyeket ki is lehet próbálni. Ezeket az `xterm` ablakból indítsuk, ugyanis X alatt futnak, karakteres konzolról nem indulnak el.
Ha úgy gondoljuk, hogy később is használni akarjuk az FLTK-t, a

```
# make install
```

parancs segítségével telepíthetjük. Ez a lépés egyébként egyelőre szükségtelen.
Példaként egy olyan programot készítünk el, amely már ebben az egyszerű formájában is használható valamire. Nem szövegszerkesztőre gondolok, több okból sem. Először is: szövegszerkesztőkkel Dunát lehet rekeszteni, tehát nem hiányzik egy újabb; másodszor pedig az FLTK tartalmaz egy kész, egyszerű szövegszerkesztő elemet, amihez csak a kívánt menüket, gyorsbillentyűket, ikonokat kell hozzáadni, és már munkára is fogható. Ehelyett az alábbiakban egy egyszerű képnézőt írunk meg. Tapasztalatom szerint Linuxon futó jó képnézegető nem igazán van, talán az xv-t kivéve, így szükség is van rá.
Célkitűzésünk első megközelítésben a következő lesz:

- egyszerűség;
- könnyű kezelhetőség;
- bővíthetőség a további fejlesztések irányában (ki tudja, talán ez lesz „a” képnézegető Linuxon…);
- a legfontosabb: a megnyitott kép méretét igazítsa a programablakhoz. Roppant bosszantó, amikor állandóan tologatni kell, vagy menükben szükséges keresgélni a kicsinyítést;
- a legfontosabb szolgáltatások (nagyítás, kicsinyítés, ablakhoz illesztés) legyenek elérhetőek billentyűzetről is, mivel így sokkal könnyebb használni őket.

Az egyszerűség kedvéért, és mert így nem kell rendszerünkön véglegesen telepíteni az FLTK-t, akár egyszerű felhasználóként is megtehetünk minden lépést (ajánlott is). Az FLTK forrásfában hozzunk létre egy könyvtárat, párhuzamosan az src és test könyvtárakkal (azaz közvetlenül az ***fltk-1.1.2*** könyvtárban). Legyen a neve mondjuk ***nezo***. Mivel most „hagyományos” módon fogunk dolgozni, nem pedig grafikus, összevont fejlesztői környezetben, a kívánt fájlokat nekünk kell létrehoznunk. Előbb készítsük el őket üresen (`touch` vagy `> fÆjlnØv`):

- ***Makefile***
- ***nezo.cxx***
- ***nezoclass.cxx***
- ***nezoclass.H***

A fájlok elnevezésénél az FLTK megszokott elnevezési mintáját alkalmaztam, azaz a C++-fájlok ***.cxx***, a C++-fejállományok (include)***.H*** kiterjesztést kapnak.
(Az összes fájl, beleértve a kész bináris programot is, letölthető a cikk végén megadott címről, illetve megtalálható a 44. CD Magazin/fltk könyvtárában).
A `Makefile` fogja a GNU `make` programnak megmondani, mit is szeretnénk. Kedvenc szövegszerkesztőnkbe írjuk be az *1. listán* láthatókat (figyelem, ahol beljebb kezdődik a sor, ott TAB karakter van! Ha a Midnight Commander beépített szerkesztőjét használjuk, az ezt piros színnel jelzi is; a `make` érzékeny erre…).

Az utolsó rész egy sorba írandó. Az FLTK beállító parancsfájlja által fáradságosan elkészített `makeinclude` fájlt használjuk, mivel ez tartalmazza a rendszerünkre vonatkozó adatokat. Most pedig térjük rá a lényegre! Mivel ravasz módon a saját osztályokat külön fájlba helyezzük, a főprogram rendkívül egyszerű lesz (lásd *2. listán*).

Tekintsük át röviden a 2. listán látottakat! Minden FLTK programba be kell illeszteni a ***FL.H*** fejállományt. Ezenkívül pedig még azokat is, amelyek a használni kívánt elemeket tartalmazzák. Ebben az esetben ez az `Fl_File_Chooser`, azaz a fájlválasztó elem. Ez, mint működése során látni fogjuk, a szokásos (Windows és KDE alatt is) fájlkiválasztóhoz hasonlít, csak annyiban jobb azoknál, hogy lehetőséget biztosít kedvencek rögzítésére és szűrők megadására, amelyek a program következő indításakor is megtalálhatók lesznek.

Ezután beillesztjük a saját fejállományunkat (`#include "nezoclass.H"`). Ebben vannak az általunk használt osztályok, valamint az egyelőre egyetlen menü meghatározásai. Ekkor két teljes hatókörű változó, ez esetben két mutató megadása következik. Erre azért van szükség, hogy ezek a program minden részéből elérhetőek legyenek. Az egyik maga a főablak, a másik a fájlválasztó ablaka. Figyeljük meg a mutatók kezdeti `NULL` értékkel való feltöltésének módját: típuskényszerítéssel azt is megadjuk, hogy ezek milyen mutatók. Ez jó szokás, ugyanis olvashatóbbá teszi a programot. Soha ne hagyjunk mutatót érték nélkül! – ez ugyanis a legbiztosabb módja a `SIGSEGV` hiba (Windows alatt Segmentation fault) elérésének.

Egy függvény, az `init` következik. Ezt a `main` függvénybe is írhattuk volna, de így áttekinthetőbb. Az `fl_register_images` meghívása azért szükséges, mert az FLTK a különböző képformátumok kezelésekor úgynevezett kezelő (handler) módszert alkalmaz. Ez a függvény tölti be az összes, a programkönyvtárban fellelhető formátum kezelőjét. (Ehhez a gépünkön ***jpeglib***-nek és ***pnglib***-nek kell lennie a megfelelő formátumok kezeléséhez, amelyeket minden Linux-terjesztés tartalmazza). Az `init` további részében a fájlválasztót „vesszük rá” a magyar beszédre. Ha ezt kihagyjuk, akkor az alapértelmezett angol szöveggel működik.

Végül a főprogram a maga egyszerűségében: meghívja az `init`-et, létrehoz egy képnéző ablakot (`Kep_Nez_Wnd`), meghívja ennek képbetöltő tagfüggvényét, majd megjeleníti azt. Az utolsó sor jellegzetes `FLTK: Fl::run();`. Ezzel lép be a program az eseménykezelő ciklusba.

Miként tapasztaljuk, ettől még nem lettünk túl okosak. A lényegi rész nyilván a ***nezoclass*** fájlokban rejlik. Valóban így van. Nézzük meg a *3. listát*, és ott a ***nezoclass.H*** fájlt. A sorokat megszámoztam, hogy hivatkozni tudjak rájuk. A sorszám fordításkor ne legyen benne!

Az 1–5. sorban ismét a fejlécbeillesztések vannak. Itt már több elemet használunk. A ***Window*** maga az ablak, a ***Box*** egy olyan elem, amelyhez képet és szöveget is lehet rendelni, most ezt alkalmazzuk a kép megjelenítésére. A ***Scroll*** tetszőleges, más elemek görgetését teszi lehetővé, a Menu_Button alkalmazható lenyíló, és mint ebben az esetben tesszük, felbukkanó menük aktiválására; végül a legfontosabb a ***Shared_Image***. Ez teszi lehetővé, hogy tetszőleges, az FLTK által támogatott képet betöltsünk. Kiválasztja az adott típushoz tartozó kezelőt, és egy minden esetben azonos ***Image*** típust ad vissza, amit az FLTK programban már bárhol, ahol képet lehet beilleszteni, használni tudunk.

Ezután a 7–11. sorban található a `Kep_Box`, a képet majdan megjelenítő elemünk megadása. Két ok miatt hozzuk létre: egyrészt saját létrehozó függvényt (konstruktor) készítünk

```
1.  #include <FL/Fl_Window.H>
2.  #include <FL/Fl_Box.H>
3.  #include <FL/Fl_Scroll.H>
4.  #include <FL/Fl_Menu_Button.H>
5.  #include <FL/Fl_Shared_Image.H>
6.
7.  class Kep_Box : public Fl_Box{
8.      int handle(int event);
9.  public:
10.      Kep_Box(int,int,int,int);
11. };
12.
13. class Kep_Nez_Wnd : public Fl_Window{
14.     Fl_Window *wnd;
15.     int origw;
16.     int origh;
17.     float fact;
18.     float afact;
19.     Fl_Shared_Image* imag;
20.     Fl_Menu_Button* mb;
21. public:
22.     Kep_Nez_Wnd();
23.     void loadkep(const char *fname);
24.     void reloadkep();
25.     void zoomin();
26.     void zoomout();
27.     void zoomfit();
28.     Kep_Box *kephely;
29.     Fl_Scroll *oll;
30. };
```

neki, így a Box esetében a létrehozáskor nem kell nem alapértelmezett tulajdonságokat módosítani, másrészt saját eseménykezelő függvénye lesz (a handle), mivel azt szeretnénk, ha bizonyos billentyűeseményekre válaszolna. Azért nem maga az ablak fog eseménykezelőt kapni, mert akkor a `Scroll`-nak szóló eseményeket is nekünk kellene kezelnünk, ezt pedig nem akarjuk, mert azt jól tudja nélkülünk is.

A 12. sortól a fájl végéig a főablak megadása található. Mint látható, az FLTK ***Window*** eleméből származtatjuk, így csak az eltérő adat- és függvénytagokat kell meghatároznunk. Először néhány adattag: ezek fogják tárolni az eredeti képméreteket, a nagyítási tényezőt, magát a pillanatnyi képet, valamint a menügombelemet, illetve a végén nyilvánosként (public) a `Kep_Box` és a `Scroll` elem mutatói.

Végül az eljárások szerepelnek: a létrehozókon kívül a kép betöltését, nagyítás utáni újratöltését, és a nagyítást, illetve kicsinyítést lehetővé tevő függvények.

Nincs más hátra, mint hogy a *4. listán* megnézzük, hogyan is működnek ezek a függvények (ismét sorszámozva).

Az 1–8. sorban újra a már megszokott fejlécfájl-beillesztéseket láthatjuk, az `fl_ask` az egyszerű üzenet és döntés ablakok meghatározását tartalmazza. A 10–11. sorban adjuk meg, hogy a főprogramban meghatározott két mutató külső, azaz másik fájlban keresendő (extern). Majd egészen a 65. sorig az úgynevezett visszahívandó (callback) függvények következnek. Az FLTK-ban a különböző elemekhez hozzárendelhetők ilyen függvények, amiket akkor hív meg a rendszer, ha az adott elemet aktiváltuk, például menüpontnál kattintunk rajta. A különböző elemek esetében ez a függvény különböző események során kerül meghívásra. Egyelőre elég tudni, hogy a menüpontoknál

kattintáskor, illetve, ha gyorsbillentyű van hozzárendelve, akkor annak megnyomásakor történik ez. A mi meghívandó függvényeink mind a különböző menüpontokhoz tartoznak.
A következő éppen ez: a `menu_popup` tömb, amely `Fl_Menu_Item` (azaz menüpont) elemekből áll. Egy ilyen elem felépítése a következő:

```
(sz veg, gyorsbillentyß, megh vand  f ggvØny,
➥felhasznÆl adat, beÆll t  zÆszl k (flag))
```

valamint a végén még hozzá lehetne fűzni a megjelenítés betűtípusára vonatkozó adatokat. Már a beállító jelzőket sem kötelező megadni, ennek alapértelmezett értéke 0. Néhány menüpontnál nincs is, ahol van, ott most az `FL_MENU_DIVIDER` jelzőt használjuk, ennek hatására az FLTK az adott menüpont után egy elválasztó vonalat jelenít meg.
A másik gyakran használt jelző az `FL_MENU_INACTIVE`, amelynek esetén a menüpont nem lesz kiválasztható, ezért halványabban kerül megjelenítésre.
Következik a `Kep_Box` létrehozója, amelynél csak annyiban írjuk felül az `Fl_Box`-ot, hogy mindenféle keret nélküli megjelenítést írunk elő (`FL_NO_BOX`).
Érdekesebb viszont a `Kep_Box` eseménykezelő tagfüggvénye, amiben jelenleg a billentyűparancsokat kezeljük le. A számbillentyűk melletti + (összeadásjel) nagyít, a – (kivonásjel) kicsinyít, a * (csillag) pedig az ablakmérethez illeszti a képet. Csak az `FL_KEYUP` típusú esemény érdekel bennünket ebben az esetben. Azért nem az `FL_KEYDOWN` (lenyomás), mert azt – előttem még ismeretlen ok miatt – nem olyan megbízhatóan kapjuk meg. Figyeljük meg, hogy ha az adott elem felhasználja az eseményt, akkor igaz értéket (1-et) ad vissza, ha viszont nem, például ebben az esetben az összes többi gomb lenyomásakor, akkor hamisat (0), így a többi elem még felhasználhatja azokat. Ezzel a módszerrel oldja meg az FLTK az események láncolását, így nincs szükség a KDE esetén alkalmazott „érdekeltségi” táblákra, amelyek erősen növelik a program méretét, lassúságát és olvashatatlanságát.
Ha olyan eseményt kaptunk, ami egyáltalán nem érdekel bennünket, ebben az esetben a billentyűfelengedésen kívül mindent, akkor meghívjuk az alapértelmezett eseménykezelőt (`Fl_Widget::handle`), átadva neki az eseményt, hogy csináljon vele, amit akar (többnyire eldobja, de lehetnek bizonyos programszintű gyorsbillentyűk vagy nem ki és bevitellel kapcsolatos események, amiket kezel). Mint látható, itt ugyanazokat csinálja, mint a megfelelő menüpont által meghívott visszahívó függvények. Igen, valóban meg lehetett volna tenni, hogy ezeket a billentyűket az adott menüpontokhoz gyorsbillentyűként rendeljük hozzá, de akkor nem lenne eseménykezelőnk, és a további bővítéshez ez még hasznos lesz.
A figyelmes szemlélőnek feltűnhet, hogy nincs egérgombkattintáshoz rendelt eseménykezelés, holott a helyi menünek a jobb egérgombra kellene megjelennie. Ez azért van, mert ezt a kérdést elintézi az `Fl_Menu_Button` saját eseménykezelője. Mint később látni fogjuk, az `Fl_Menu_Button` elemet az ablak teljes méretére nagyítjuk, így a jobb gombbal bárhol kattintva a menü működik.
Itt van végre a főablak létrehozója! Kicsit egyedi, mivel most csak egy példány lesz belőle, így előbb ellenőrzi, hogy ez megvan-e (a teljes hatókörű mutatón keresztül, ha még nincs, akkor ez `NULL` – az előrelátásunknak hála. Ha nincs, létrehozza. Később, ha több ablakot akarunk, ezt kell kivennünk, hogy másikat készíthessen.)
Mivel ez az `Fl_Window` elemből származtatott, így oda kell gondolnunk a következő sort:

```
this=new Fl_Window(600,450,"KØpnØzı");
```

A szülő létrehozóját a megadott értékekkel hívja meg. Az FLTK elemelrendezésének jellegzetessége, hogy a `new` után addig, amíg az adott elem `end()` függvényét meg nem hívjuk, az összes új elem ennek a gyermeke lesz. (Természetesen csak akkor, ha ez lehetséges.) Csak a `Group` (csoport, tároló) típusból származtatott elemeknek van `end()` tagfüggvénye, így csak ezek tartalmazhatnak más elemeket. A pdf-leírásban megnézhetjük, hogy mely elemek származnak az `Fl_Group`-ból. Tehát nálunk az ablak tartalmaz egy `Fl_Scroll` elemet (görgető), és ez tartalmazza a képdobozt. Az ablakhoz rendelt még a `Menu_Button` is. Az `Fl_Scroll` természetesen szintén `Group` típusú, így az `end()` zárja le.
Még egy FLTK-érdekesség. Az elemek helyzetét és méretét pontértékben írjuk be. Nincs külön elrendezés- (layout handler) kezelő, ám mivel megadhatjuk, mely elemek méretezhetőek át, az ablak átméretezésekor történő változást rugalmasan irányíthatjuk. A mi esetünkben például ez oldja meg, hogy a `Menu_Button` mérete mindig megegyezzen az ablakéval, azaz mindenhol működjön.
A továbbiakban a többi tagfüggvényt határozzuk meg.
Ez elég egyértelmű. Ami érdekes, az talán a képbetöltés: a képfájlt az `Fl_Shared_Image` `get()` tagfüggvénye segítségével töltjük be. Ez a fájl tartalma alapján (nem csak a kiterjesztést nézi, tehát egy JPG fájlnak például ***.dat*** kiterjesztése is lehet, és ilyenkor is helyesen tölti be) kiválasztja a megfelelő kezelőt, majd egy képmutatót ad vissza. Figyelemre tarthat számot ez is: az új betöltés előtt a régi képet töröljük, hogy ne foglalja a memóriát. Mivel alapként mindig az `Fl_Shared_Image` szolgál, és ennek `copy()` tagfüggvényét használva másoljuk át szükség szerinti méretben a `Kep_Box`-ba, újabb kép betöltésekor ezt is törölni kell. Ezt nem tehetjük meg úgy, mint az `Fl_Image`-dzsel, azaz a delete paranccsal. Ezeket a képeket ugyanis az FLTK egy tömbben tárolja, mert ennek az elemtípusnak az is célja, hogy bizonyos képeket több helyen, újra lehessen használni, újabb betöltés nélkül (ezt mutatja a neve is). A `get()` csak akkor olvassa be újra a fájlból, ha még nincs a memóriában képként, így a `release()` tagfüggvényével kell törölni. Utána a megfelelő `NULL` értéket adjuk a mutatónak. Igaz, hogy pár sorral

```
1.   #include <FL/Fl.H>
2.   #include <FL/Fl_Box.H>
3.   #include <FL/Fl_Window.H>
4.   #include <FL/Fl_Shared_Image.H>
5.   #include <FL/Fl_Menu_Button.H>
6.   #include <FL/Fl_File_Chooser.H>
7.   #include <FL/fl_ask.H>
8.   #include "nezoclass.H"
9.
10.  extern Kep_Nez_Wnd* kep_nez_wnd;
11.  extern Fl_File_Chooser* kep_opener;
12.
13.  void exit_cb(Fl_Widget *, void *) {
14.    exit(0);
15.  }
16.
17.  void nevjegy_cb(Fl_Widget*w,void *d)
18.  {
19.   fl_message("FLTK KØpnØzı ver. 0.1\n
      ➥KØpnØzı dem  FLTK-hoz\n
      ➥(C)2002 HavrÆnek Ferenc\n
20.  Ez a program alkalmaz az Independent
     ➥JPEG Group Æltal kØsz tett
     ➥rØszeket.");
21.  }
22.
23.  void fit_cb(Fl_Widget *w,void *d)
24.  {
25.  Kep_Nez_Wnd *p;
26.  p=::kep_nez_wnd;
27.  p->zoomfit();
28.  p->reloadkep();
29.  p->redraw();
30.  }
31.
32.  void nagyit_cb(Fl_Widget *w,void *d)
33.  {
34.  Kep_Nez_Wnd *p;
35.  p=::kep_nez_wnd;
36.  p->zoomin();
37.  p->reloadkep();
38.  p->redraw();
39.  }
40.
41.  void kicsit_cb(Fl_Widget *w,void *d)
42.  {
43.  Kep_Nez_Wnd *p;
44.  p=::kep_nez_wnd;
45.  p->zoomout();
46.  p->reloadkep();
47.  p->redraw();
48.  }
49.
50.  void kep_open_cb(Fl_Widget *w, void *d)
51.  {
52.  if(!::kep_opener){
53.   Fl_File_Chooser *fc=new Fl_File_Chooser
      ➥(".","KØp fÆjlok (*.{bmp,gif,jpg,png,
      ➥pnm,pbm,BMP,GIF,JPG,PNG})",
      ➥Fl_File_Chooser::SINGLE,
      ➥"kØp bet ltØs");
54.  fc->preview(1);
55.  kep_opener=fc;
56.  kep_opener->newButton->hide();
57.  }
58.  kep_opener->rescan();
59.  kep_opener->show();
60.  while(kep_opener->visible()) Fl::wait();
61.  if(kep_opener->count()){
62.   ::kep_nez_wnd->loadkep
      ➥(kep_opener->value());
63.   ::kep_nez_wnd->redraw();
64.   }
65.  }
66.
67.  Fl_Menu_Item menu_popup[] = {
68.      {"KØp kivÆlasztÆsa", 0,
      ➥kep_open_cb,0,FL_MENU_DIVIDER},
69.      {"Passz t",0,fit_cb,0},
70.      {"Nagy t", 0, nagyit_cb, 0},
71.      {"Kicsiny t",0,kicsit_cb,
      ➥0,FL_MENU_DIVIDER},
72.      {"NØvjegy",0,nevjegy_cb,0,
      ➥FL_MENU_DIVIDER},
73.      {"KilØp",0,exit_cb,0},
75.      {0}
76.  };
77.
78.  Kep_Box::Kep_Box(int x,int y,int w,int h)
     ➥: Fl_Box(x,y,w,h)
79.  {
80.  this->box(FL_NO_BOX);
81.  }
82.
83.  int Kep_Box::handle(int event)
84.  {
85.  Kep_Nez_Wnd *p;
86.  p=::kep_nez_wnd;
87.    switch(event){
88.   case FL_KEYUP:
89.   switch(Fl::event_key()){
90.           case FL_KP+'+': p->zoomin();
91.          p->reloadkep();
92.                redraw();
93.                return 1;
94.           case FL_KP+'-': p->zoomout();
95.                p->reloadkep();
96.          p->redraw();
97.          redraw();
98.          return 1;
99.           case FL_KP+'*':
             ➥p->zoomfit();
100.               p->reloadkep();
101.               p->redraw();
102.               redraw();
103.               return 1;
104.  default: return 0;
105.  }
106.  default:  return
      ➥Fl_Widget::handle(event);
107.  }
```

folytatás a következő oldalon

folytatás az előző oldalról

```
108. }
109.
110.
111. Kep_Nez_Wnd::Kep_Nez_Wnd() :
     ➥Fl_Window(600,450,"KØp nØzı")
112. {
113.  if(!::kep_nez_wnd){
114.   mb=new Fl_Menu_Button(0,0,600,450,
       ➥"KØp men ");
115.   mb->type(Fl_Menu_Button::POPUP3);
116.   mb->menu(menu_popup);
117.
118.   oll=new Fl_Scroll(0,0,600,450);
119.   oll->type(Fl_Scroll::BOTH_ALWAYS);
120.   kephely=new Kep_Box(0,0,584,434);
121.   oll->end();
122.   this->end();
123.   afact=1.0;
124.   fact=1.2;
125.   imag=(Fl_Shared_Image*)0;
126.   ::kep_nez_wnd=this;
127.   this->resizable(mb);
128.   this->resizable(this);
129.   wnd=this;
130.  }
131. }
132.
134.
135. void Kep_Nez_Wnd::zoomin()
136. {
137.  afact=afact * fact;
138. }
139.
140. void Kep_Nez_Wnd::zoomout()
141. {
142.  afact=afact / fact;
143. }
144.
145. void Kep_Nez_Wnd::zoomfit()
146. {
147. float wd,hd;
148.  if(origw< w() && origh< h())
      ➥{ afact=1.0; return; }
149.  wd=(float)(w()-16) / (float)origw;
150.  hd=(float)(h()-16) / (float)origh;
152.  afact=(wd <= hd) ? wd : hd;
153.  if(afact>1.0)afact=1.0;
154.  this->oll->position(0,0);
155. }
156.
157.
158. void Kep_Nez_Wnd::loadkep
     ➥(const char *fname)
159. {
160.  Fl_Image *img;
161.  Fl_Image *oldimg;
162.  int nw,nh;
163.  afact=1.0;
164.  fact=1.2;
165.  if(imag) { imag->release();
      ➥imag=(Fl_Shared_Image*)0; }
166.  oldimg=this->kephely->image();
167.  if(oldimg){ delete oldimg;
      ➥this->kephely->image((Fl_Image*)0); }
168.  imag=Fl_Shared_Image::get(fname);
169.  origw=imag->w();
170.  origh=imag->h();
171.  zoomfit();
172.  nw=(int)(afact*origw);
173.
174.  nh=(int)(afact*origh);
175.  img=imag->copy(nw-1,nh-1);
176.  this->kephely->size(img->w(),img->h());
177.  this->kephely->image(img);
178. }
179.
180. void Kep_Nez_Wnd::reloadkep()
181. {
182.  int nw,nh;
183.  Fl_Image *img;
184.  Fl_Image *oldimg;
185.  oldimg=this->kephely->image();
186.  if(oldimg){ delete oldimg;
      this->kephely->image((Fl_Image*)0); }
187.  nw=(int)(afact*origw);
188.  nh=(int)(afact*origh);
189.  if(nw>=origw || nh>=origh){ nw=origw-1;
      ➥nh=origh-1; afact=1.0; }
190.  img=imag->copy(nw,nh);
191.  this->kephely->size(img->w(),img->h());
192.  this->kephely->image(img);
193. }
```

lejjebb már be is töltünk egy fájlt, és új értéke lesz, szóval értelmetlennek tűnik. De mi történik akkor, ha sikertelen a betöltés? Akkor a későbbiek folyamán egy érvénytelen mutató lenne a változóban, ami sok bajt okozhatna.
A `reload()` tagfüggvény a megfelelő méretszámítások után másolatot készít a tárolt `Fl_Shared_Image`-ről, és ezt hozzárendeli a `Kep_Box`-hoz, ami azután megjeleníti.
Röviden ennyit az FLTK-ról. Igyekeztem rámutatni, hogy érdemes megnézni. Akit mélyebben érdekel, javaslom, olvassa el a 44. CD Magazin/fltk/ könyvtárában lévő példaprogramok forrását, valamint az FLTK forrását. Rendkívül logikus és áttekinthető a felépítése. Aki félt a C++-tól, érdemes ezzel kezdenie, mert itt hamar sikerélményhez juthat!

Sorozatunk következő részben elkezdjük bővíteni a programot. Az első számú cél, hogy több fájlt lehessen betölteni egyszerre, akár több könyvtárból is, és a billentyűzet segítségével mozogni lehessen közöttük. Esetleg menteni lehessen az adott listát hogy a betöltés után önműködően le tudjuk játszani őket.

Havránek Ferenc
Automatikamérnökként dolgozik. Kedvtelései közé tartozik mindenféle kétkerekű járművön (kerékpár és motor) való közlekedés. Ezenkívül szívesen tölti idejét programozással, nemcsak PC-s, hanem egyéb környezetben is, például mikrovezérlő programokat ír.