© Kiskapu Kft. Minden jog fenntartva

Az eFltk bemutatása (3. rész)

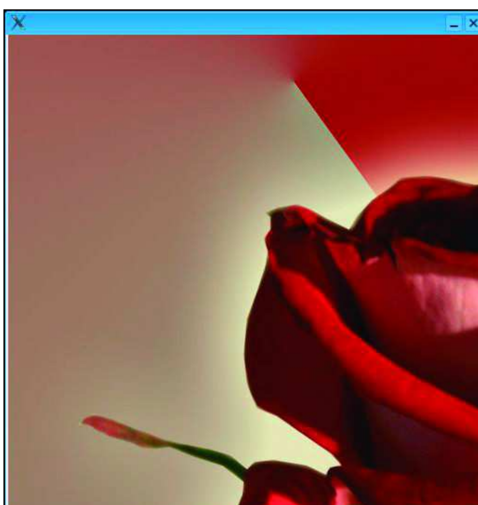
Saját vezérlőelemek készítése.

A sorozat harmadik részében bemutatom, hogyan készíthetünk saját vezérlőelemeket az ***eFltk*** megfelelő osztályainak felhasználásával. Sok esetben szükségünk lehet olyan grafikus elemekre, melyek nem találhatók meg a rutinkönyvtárban, gondoljunk csak egy folyamatosan mintavételezett adatokat megjelenítő grafikonra vagy akár egy kép nagyítását és eltolását lehetővé tévő elemre. Ezen oldalakon a későbbiekben az utóbbi példával fogom megvilágítani az ismeretek gyakorlati alkalmazását.

Mielőtt nekifognánk a megjelenítő osztály elkészítéséhez, hasznos lesz megismerkednünk a rendelkezésre álló rajzoló függvényekkel. Kezdjük az egyszerűbbekkel, mindenekelőtt ne felejtsük el beszúrni az ***efltk/fl_draw.h*** feljécállományt a forráskódba.

Lássuk tehát a színek beállítására szolgáló `fl_color()` függvényt, melyhez szükségünk lesz az ***efltk/Fl_Color.h*** állomány beszúrására is. A függvény első változatában megadhatunk három byte méretű összetevőt, melyből a függvény előállítja a szükséges `Fl_Color` típusú értéket. Használhatjuk a tizenhatos számrendszerben megadott színmeghatározás átalakítására is, amennyiben paraméterként csak egy `#RRGGBB` formájú (***HTML*** dokumentumokban szokásos forma) karakterláncot adunk át a függvénynek. Amennyiben előre megadott színt szeretnénk használni, tekintsük át az ***Fl_Color.h*** állományt, melyben megtalálhatók az alapszínek (például `FL_RED`, `FL_GREEN`, `FL_BLUE`, `FL_GRAY`, `FL_YELLOW`, `FL_MAGENTA`, `FL_BLACK`, `FL_WHITE`) és használhatunk további színeket módosító függvényeket is.

Ilyen módosító függvény lehet például az `fl_lighter()` és az `fl_darker()`, melyek a megadott szín világosabb és sötétebb változatát adják vissza. A másik hasznos függvény ebben az állományban, az `fl_color_average()`, mely két szín súlyozott átlagát számítja ki. Az első két paraméterben a színeket adjuk meg, míg a harmadik paraméter a súlyozási tényező. Ez az érték 0.0-tól 1.0-ig vehet fel értékeket, minél közelebb áll az 1-es értékhez, a visszaadott szín annál inkább az első paraméterben megadott színhez fog hasonlítani.



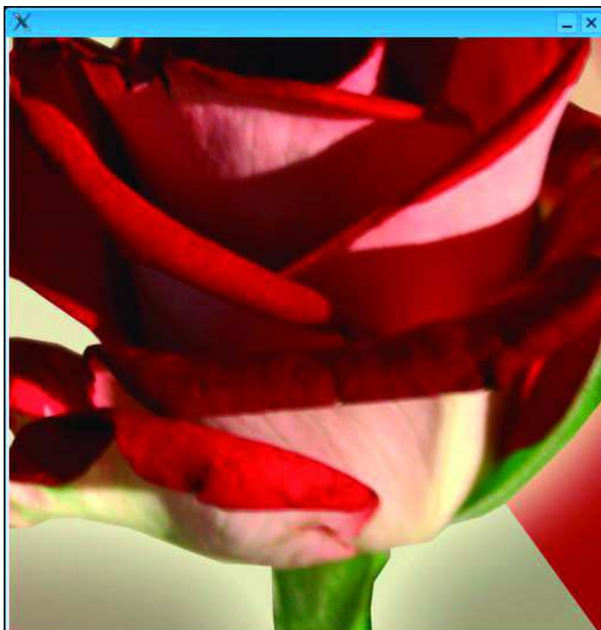
1. ábra A kiinduló kép

Miután a színekkel megismerkedtünk következzenek a rajzoló eljárások. Ezeket az eljárásokat általában a vezérlőelemek rajzolására használjuk, ahogyan azt majd a későbbiekben látni fogjuk. Az `fl_color()` eljárás alkalmas az aktuális rajzoló szín beállítására is, ilyenkor nem függvényként használjuk, hanem csak meghívjuk a megfelelő paraméterrel. Az `fl_line_style()` függvény alkalmas a rajzoláshoz használt vonal stílusának beállítására. Első paramétere a vonal stílusa, második a vonalvastagság, a harmadik pedig a szaggatott vonalak formáját meghatározó tömböt címző mutató. Ez utóbbi paraméternek általában `NULL` értéket adhatunk, hiszen rengeteg előre beállított vonalstílus közül válogathatunk. Ilyenek például az `FL_SOLID`, `FL_DASH`, `FL_DOT`, `FL_DASHDOT` és az `FL_DASHDOTDOT`. A vonalstílusokat kombinálhatjuk a vonalak végződését meghatározó `FL_FLAT`, `FL_ROUND` és `FL_SQUARE` értékekkel, amennyiben ezt nem adjuk meg, az ***eFltk*** rendszer az adott rendszerben leggyorsabban megvalósítható megoldást használja.

Egyszerű téglalapokat rajzolhatunk az `fl_rect()` függvénnyel, míg kitöltött téglalapokat az `fl_rectf()` fügvény használatával. Mindkét esetben a bal felső és a jobb alsó koordinátákat kell paraméterként átadni.

Az `fl_line()` függvény alkalmas vonalak rajzolására, a szokásos megoldás szerint a kezdő- és végpontok meghatározásával. Az `fl_point()` függvény segítségével pontokat rajzolhatunk a megadott koordinátán.

Körív rajzolására használható az `fl_arc()` függvény. Első négy paramétere meghatározza a körívhez tartozó téglalapot, melyben a kör vagy ellipszis elhelyezkedik, míg az ötödik és a hatodik paraméter a kezdő- és végszöget adja meg. Szintén használható körív rajzolására az `fl_pie()` függvény is, mégpedig a hetedik paraméterben megadott típus segítségével. Az első hat paraméter megegyezik az előző függvénynél használtakkal, azonban a hetedik egy konstans



2. ábra A képmegjelenítő a kép eltolása után

érték lehet. Ez az érték lehet FL_ARC, FL_PIE és FL_CHORD. Amennyiben nem adunk meg semmit, az alapértelmezés az FL_PIE és kitöltött körcikket kapunk eredményül.
Az fl_circle() függvénnyel kört rajzolhatunk. Első két paramétere a kör középpontja, míg a harmadik a kör sugara. Az fl_ellipse() négy paraméterével egy téglalapba foglalható ellipszist tudunk rajzolni.
Tömörítetlen képeket, melyek a memóriában találhatók, az fl_draw_image() függvény segítségével jeleníthetünk meg. A függvény első paramétere a képpontokat címző mutató, a következő négy határozza meg a megjelenítendő kép bal felső sarkának pozícióját és a kép méreteit. Az ötödik paraméter a képpontonkénti byte-ok száma, míg a hatodik egy konstans érték, mely a rajzoláskor minden sorban egy eltolást ad a forrásadatokhoz. Így valósítható meg például a függőleges irányú átméretezés vagy tükrözés. Amennyiben ez utóbbi értéket nem adjuk meg, az ***eFltk*** rutinok alapértelmezésben a kép szélességét és a képpontok mérete (byte-ban) alapján számítja ki a sorok eltolási értékét.
A rajzoló eljárások használata közben beállíthatunk vágási területet is. A terület beállítása után a függvények csak ezen a területen belül produkálnak látható eredményt. A vágási terület beállítására szolgáló eljárás az fl_push_clip(), melynek négy paramétere határozza meg a vágáshoz használandó téglalapot. A rajzoló műveletek befejezése után ne felejtsük el az fl_pop_clip() függvénnyel visszavonni a beállítást.
Az alábbiakban látható egy összefoglaló a rajzoláshoz használható függvényekről és paraméterezésükről.

```
fl_color(Fl_Color c)
fl_color(r, g, b)
Fl_Color fl_color()

fl_push_clip(x, y, w, h)
fl_pop_clip()
```

```
fl_line_style(style, width, *dashes)

fl_arc(x, y, w, h, a1, a2)
fl_circle(x, y, r)
fl_ellipse(x, y, w, h)
fl_pie(x, y, w, h, a1, a2, what)

fl_point(x, y)
fl_line(x1, y1, x2, y2)
fl_rect(x, y, w, h)
fl_rectf(x, y, w, h)
fl_rectf(x, y, w, h, Fl_Color c)
fl_rectf(x, y, w, h, r, g, b)

void fl_draw_image(*im, x, y, w, h, D, ld)
void fl_draw_image_mono(*im, x, y, w, h, D, ld)
```

Lehetőségeink megismerése után ideje hozzáfogni a saját vezérlőelemünk elkészítéséhez. Az objektum-orientált programozás szemléletének köszönhetően az ***eFltk***-ban már megvalósított elemeket felhasználhatjuk saját vezérlőelemek készítéséhez, amit most meg is teszünk. Az első példában az Fl_Double_Window lesz a kiindulási alap, és a célunk legyen egy olyan elem készítése, mely egérkattintáskor egy kört rajzol az ablakba. Amikor az egérgombot felengedjük, a kör már ne legyen látható.
Elsőként tehát határozzuk meg az osztályt. Az ősosztály az Fl_Double_Window, és célszerű bevezetni egy logikai változót a lenyomott állapot tárolására. Erre szolgál a _pushed osztálytag.
Tulajdonképpen az ***eFltk***-ban az új vezérlőelemek létrehozásakor két dologra kell figyelnünk. Gondoskodnunk kell az elemek kirajzolásáról és az események kezeléséről. Az alábbi listán látható a leszármazott osztály definíciója.

```
class myWindow:
 public Fl_Double_Window
{
private:
 bool _pushed;
public:
 myWindow (int w, int h);
 ~myWindow ();

protected:
 void draw ();
 int handle (int event);
};
```

A kirajzolásról gondoskodik a draw() osztályfüggvény, tehát itt kezeljük majd a lenyomott állapot és a felengedett állapot közötti megjelenésbeli eltérést. Ne felejtsünk el gondoskodni a változók kezdőértékének beállításáról, például az osztály konstruktorában. Szintén a konstruktorban kell gondoskodnunk az alaposztály életre keltéséről, tehát ne feledkezzünk meg az Fl_Double_Window konstruktorának meghívásáról. A rajzolást megvalósító függvény nagyon egyszerű, az alábbi néhány sor segítségével jól érthetővé válik.

© Kiskapu Kft. Minden jog fenntartva

© Kiskapu Kft. Minden jog fenntartva

```
myWindow::draw()
{
 // Ha az egészet újra kell rajzolni
 if (damage() & FL_DAMAGE_ALL) {
  fl_color(FL_WHITE);
  // alap téglalap kirajzolása
  fl_rectf(0, 0, w(), h());
  fl_color(FL_BLACK);
  // Csak ha lenyomva, akkor kell kör
  if (_pushed)
   fl_circle(w()/2, h()/2, h()/2-5);
 }
}
```

A másik fontos függvény, amit meg kell valósítanunk az eseménykezelő handle() függvény. Paraméterként kapja az esemény típusát. Az alábbi táblázatban látható az eFltk által kezelt események listája.

- `FL_PUSH`, `FL_RELEASE` egérgomb lenyomása, felengedése
- `FL_DRAG` egér mozgatása + lenyomott gomb
- `FL_MOVE` egér mozgatása
- `FL_MOUSEWHEEL` egérgörgő
- `FL_ENTER`, `FL_LEAVE` egérmutató az elem területén, elhagyja
- `FL_FOCUS`, `FL_UNFOCUS` billentyűzetfókusz, annak megszűnése
- `FL_KEYUP`, `FL_KEYDOWN` billentyű lenyomás-, felengedés
- `FL_SHOW`, `FL_HIDE` elem megjelenik, eltűnik

Ebben az egyszerű példában csak az `FL_PUSH` és az `FL_RELEASE` eseményre kell figyelmet fordítani, tehát az eseménykezelő függvény igen egyszerű lehet. Az ***eFltk*** mindaddig próbálja továbbadni az eseményeket az elemek között, amíg valamelyik le nem kezeli azokat. A `handle()` függvénynek kell tehát értesíteni a rendszert arról, hogy kezelte-e az eseményt vagy további feldolgozásra van szükség. Amennyiben az eseménykezelő 0 értékkel tér vissza, akkor szükség van további feldolgozásra, vagyis az esemény nem érdekes az aktuális elem szempontjából, míg ha nullától különböző a visszatérési érték, azzal jelezhetjük, hogy sikeresen feldolgoztuk az adott eseményt. Lássuk tehát a példabeli eseménykezelő függvényt.

```
int myWindow::handle(int event)
{
  switch (event) {
    case FL_PUSH:
      _pushed = true;
      redraw();
      return 1;
    case FL_RELEASE:
      _pushed = false;
      redraw();
      return 1;
    default:
      return Fl_Double_Window::handle(event);
  }
}
```

A fenti eseménykezelő eljárásban csak az egérgombra vonatkozó eseményeket kezeljük, mégpedig egyszerűen átállítjuk a lenyomást jelző `_pushed` változó értékét és újrarajzoljuk az ablakot. Amennyiben számunkra érdektelen eseményről van szó, meghívjuk az ős osztály eseménykezelő eljárását és visszatérünk az általa visszaadott értékkel.
Miután ezt az egyszerű példát megértettük, következzen egy összetett feladat megoldása. A feladat az, hogy olyan képmegjelenítő elemet készítsünk, melyben az egérgomb lenyomásával és az egér húzásával a képet mozgatni tudjuk. Néhány kiegészítő változót kell bevezetnünk, ahogyan az az osztálydefinícióban is látható. Szükségünk van egy `Fl_Image` objektumra, melyben tároljuk a megjelenítendő képet és egy kiegészítő képre, amire minden egérmozgatás után átmásoljuk a kép látható részét. Itt a sebességgel lehetnek apróbb gondok, de most az egyszerű megoldásra törekszünk. Szükségünk van továbbá egy eljárásra (`copy_image_region()`), ami a tényleges másolást elvégzi. Az eseménykezelő eljárásban az `FL_DRAG` eseményre kell figyelni, azonban ez az esemény csak akkor jut el az elemhez, ha az reagál az `FL_PUSH` és az `FL_RELEASE` eseményekre is.
Az eseménykezelő eljárásban kiszámítjuk az egér elmozdulását, átmásoljuk a kisebb képre az eredeti kép megfelelő területét és újrarajzoljuk a képet. Itt figyelnünk kell arra is, hogy a képnek csak a ténylegesen látható területével foglalkozzunk, vagyis ne csökkenhessen 0 alá az eltolást meghatározó érték és ne is növekedhessen olyan érték fölé, ami a kép túlcímzését eredményezné. Az eseménykezelő eljárás az alábbi listán látható.

```
int myWindow::handle (int event)
{
 int _dx, _dy;
 bool _changed = false;

 switch (event) {
 case FL_PUSH:
 case FL_RELEASE:
 return 1;
    case FL_DRAG:
      // koordináták kezdeti értéke
      if (_ox == 0 || _oy == 0) {
         _ox = Fl::event_x ();
         _oy = Fl::event_y ();
         return 1;
       }
      _dx = (_ox - Fl::event_x ());
      _dy = (_oy - Fl::event_y ());
      // ellenőrzés vízszintes irányban
      if (_sx+_dx>0 && _sx+_dx+this->w () <
              _image->width ()) {
         _sx += _dx; _changed = true;
       }
      // ellenőrzés függőleges irányban
      if (_sy+_dy>0 && _sy+_dy+this->h () <
              _image->height ()) {
         _sy += _dy; _changed = true;
       }
```

```
        // Csak akkor rajzolunk, ha változás volt
        if (_changed) {
           copy_image_region (_sx, _sy, w (),
           ➥h ());
           redraw ();
         }
         // koordináták frissítése
        _ox = Fl::event_x ();
        _oy = Fl::event_y ();
        return 1;
      default:
        return Fl_Double_Window::handle
        ➥(event);
      }
}
```

A rajzolást végző eljárásban egészen egyszerűen csak kirajzoljuk a már átmásolt képrészletet a grafikus elem megjelenítendő részére.

```
void myWindow::draw ()
{
 if (_cimage && _image)
 _cimage->draw (0, 0, w (), h ());
}
```

Amiről még szót kell ejtenem, a kép betöltését végző `set_image()` függvény. Az *eFltk*-ban egyszerűen betölthetünk *JPG*, *PNG* és *GIF* formátumú képeket, mégpedig az `Fl_Image::read(filename)` statikus metódus segítségével. A függvény egy *Fl_Image* típusú objektumot ad vissza, amit akár beállíthatunk valamilyen vezérlőelem képeként (az `Fl_Widget::image()`) függvény használatával) vagy további feldolgozás után tetszőleges módon feldolgozhatunk.

Összefoglalásképpen tekintsük át az új grafikus elem készítéséhez szükséges ismereteket. Először is kiválasztjuk a megfelelő ős osztályt. Kiegészítjük a szükséges osztálytagokkal a saját osztályunkat, majd megírjuk a kirajzolást végző `draw()` függvényt. Miután eldöntöttük, hogy milyen eseményekre kell reagálnia az elemnek, következhet a `handle()` eljárás elkészítése. Ebben az eljárásban a 0 visszatérési értékkel jelezzük az *eFtlk* rendszer számára, hogy az elem nem dolgozza fel a kapott eseményt, így azt továbbítani kell. A 0-tól eltérő érték jelzi, hogy az eseményt feldolgoztuk. Itt végül célszerű meghívni az ős osztály eseménykezelő eljárását és ha nem kezeltük az eseményt, akkor ennek visszatérési értékét felhasználni.

Végezetül láthatunk néhány képet, mely a cikk alapját képező programot próbálja szemléltetni működés közben. A program forráskódja szabadon hozzáférhető a *http://dzooli.uw.hu/efltk_draw.tgz* hivatkozáson keresztül. Mindenkinek kellemes időtöltést kívánva búcsúzom, elektronikus levél útján továbbra is szívesen válaszolok a felmerülő kérdésekre.

Fábián Zoltán

© Kiskapu Kft. Minden jog fenntartva