SZAKDOLGOZAT

Forray András Pál

Debrecen

2007

Debreceni Egyetem

Informatika Kar

Multimédiás adatformátumok feldolgozása Java nyelvben (animációk)

Témavezető:
Dr. Boda István
egyetemi adjunktus

Készítette:
Forray András Pál
Programozó matematikus

Debrecen

2007

Tartalomjegyzék

1. Bevezetés

A szakdolgozatom célja az FLC fájlformátum bemutatása és egy olyan lejátszóprogram készítése, mely alkalmas FLC formátumú állományok web böngészőben történő lejátszására. A dolgozat elkészítéséhez az alábbi irodalmakat vettem alapul:

1. ***Animator Pro File Formats***
   http://mediasrv.ns.ac.yu/extra/fileformat/animation/flc/flc.rtf
2. ***ITB CompuPhase: The FLIC file format***
   http://www.compuphase.com/flic.htm
3. ***FileFormat.Info : FLI File Format Summary***
   http://www.fileformat.info/format/fli/
4. ***Kuba Attila: FLI lejátszás***
   http://www.inf.u-szeged.hu/oktatas/jegyzetek/KubaAttila/animacio/9_fli.htm

A lejátszóprogram alapját J. Anders Java nyelven elkészített programja szolgáltatta:
http://rnvs.informatik.tu-chemnitz.de/%7Ejan/FLI/Flic_Player.java

Egykor (az 1990-es évek közepén) a FLIC fájlformátum volt az egyik legnépszerűbb animációs formátum MS-DOS, illetve Windows környezetben. Széles körben használták animációs programok, számítógépes játékok, valamint CAD alkalmazások területén. Kezdetben -más animációs formátumokkal együtt- a hang és a kép adatok együttes kezelését nem támogatta, helyette csupán állókép adatok sorozatait tárolta. Később az EGI[1] (2.0-tól) egy wave audio tömbbel terjesztette ki, s így alkalmassá vált hang és képadatok tárolására is. A FLIC népszerűségét növelte a mögötte álló ötlet egyszerűsége, valamint a hozzá tartozó programok könnyű megvalósíthatósága is. Lehetővé teszi az animációk gyors lejátszását, és nem kíván speciális hardver eszközt az adatok kódolásához és dekódolásához sem. A számítógéppel létrehozott vagy kézzel rajzolt animációs képsorozatok számára az egyik legalkalmasabb formátum, és ezzel a formátummal nagyon jó tömörítés valósítható meg. Természetesen a valós világnak azokat a képeit, melyekből animáció készíthető szintén tárolhatjuk ebben a formátumban. De az ilyenfajta képek rendszerint nagyon sok olyan adatot tartalmazhatnak, amelyek csökkentik a FLIC kódoló algoritmusok adattömörítő képességét, és

[1] FLIC fordító- és lejátszóprogram (*http://www.compuphase.com/software_egi.htm*)

ez hatással lehet az animáció lejátszási sebességére. A kevesebb színt tartalmazó animációk jobban tömöríthetőek.

A FLIC animációknak általában két típusát szokták megkülönböztetni. Az egyik a kezdeti FLIC formátum, amely „.FLI" (fájlnév) kiterjesztéssel rendelkezik. Ezen típusú fájlok maximális képfelbontása 320∗200 képpont. Az újabb, jobb képfelbontást lehetővé tevő típust az „.FLC" kiterjesztéssel látták el. Az FLC fájlok által alkalmazott adattömörítési eljárás szintén hatékonyabb, mint az FLI fájlok esetén használt. Az olyan alkalmazások, mint például az IBM Multimedia Tool sorozat, a Microsoft Video for Windows, és az Autodesk Animator Pro mind támogatják az FLC típusú fájlokat.

Minden olyan alkalmazásnak, amely képes az újabb FLC formátumú fájlok beolvasására, képesnek kellene lennie a régebbi FLI fájlok beolvasására és lejátszására is. Azonban a legtöbb újabb FLIC fájlíró egyedül FLC fájlokat tud létrehozni.

A fájl szerkezetet tekintve a FLIC animációk állóképek egymást követő sorozata. Az állóképeket más néven képkockáknak (*frame*-eknek) szokták nevezni. Minden képkocka az animáció egy darabjának adatait foglalja magában. Az animáció lejátszásának a sebességét az egyes képkockák között előforduló késleltetés mértékének megadásával szabályozhatjuk. Minden egyes képkockában az adatokhoz minden esetben egy színtérkép tartozik. Valamennyi képpont egy index értéket tartalmaz a -képkockához definiált- színtérképhez. Ha szükséges, képkockáról képkockára meg lehet változtatni a térképen található színeket. Jóllehet a FLIC fájlok megjelenítése maximálisan képkockánként 256 színre korlátozódik, de minden képpont 24 bites színmélységgel rendelkezik, így végeredményben több mint 16 millió színű palettából választhatunk.

A FLIC formátum különféle adattömörítési módokat támogat. A FLIC animáció minden képkockája az „*interframe delta*"[2] kódolási sémát használó sajátos tömörítéssel rendelkezik. Ez a tömörítési séma nem egyedül a képkockákat kódolja, hanem az egymás mellett lévő képkockák képeinek a különbségét. Ezzel a stratégiával jelentősen kisebb fájlokat kapunk, mintha minden egyes képkockát egymástól függetlenül kódolnánk (*intraframe encoding*),

[2]

valamint az „interframe” kódolt adatok kitömörítése és megjelenítése is nagyon gyors. Az RLE kódolási algoritmus alkalmazva a FLIC animációk első képkockája teljes egészében tömörített.

A felépítést tekintve a FLIC fájlok egy 128 bájtos fejléccel kezdődnek. Az „.FLI” és az „.FLC” fájlok esetén az első 9 mező (22 bájt) megegyezik. Az „.FLC” fájlokban egyedül az utolsó 10 mező (106 bájt) tartalmaz érvényes adatokat; az „.FLI” fájlok esetén ezen mezők értéke 00h (hexa) értékű. A fájlok (fejlécet követő) további részeit a későbbiekben tekintjük át részletesebben.

——— ♦ ———

2. Az FLI és FLC animációk lejátszása, készítése

Ezek az animációs formátumok ma már meglehetősen réginek számítanak, ezért a létrehozásuk óta eltelt időben igen sok lejátszó- és szerkesztőprogramot készítettek a különböző számítógépes architektúrákon. De az sem ritka, hogy megfelelő bővítménnyel kiegészítve, és a szükséges beállításokat elvégezve a különböző web böngésző programok is képesek lejátszani ezeket az állományokat. Ilyen lejátszóprogramok többek között:

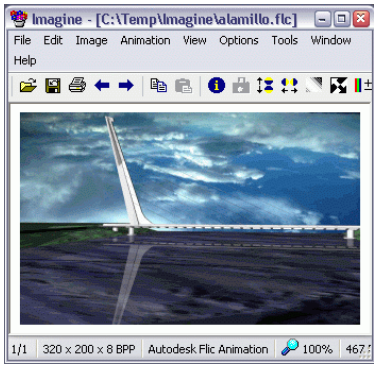
- a QuickTime[3] (Mac, Windows 3.1, Windows 95/98, Windows 2000/XP);

Mely egy ingyenes multimédia lejátszóprogram. Segítségével le tudunk játszani számos olyan fájlt, amely videót, audiot, állóképet, virtuális valóság („Virtual Reality") filmeket, valamint grafikát tartalmaz.

- a Waaplay[4] (Windows 3.1, Windows 95/98);

A program segítségével lejátszhatjuk az Autodesk Animator, az Autodesk 3D Studio, és az Autodesk Animator Pro programok által készített animációkat.

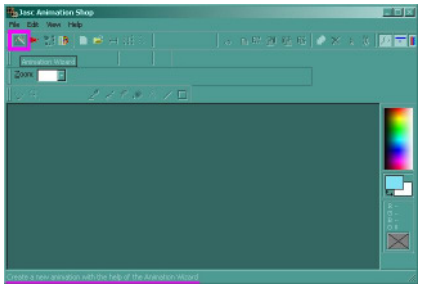
- az Imagine[5] (Windows 2000/XP);

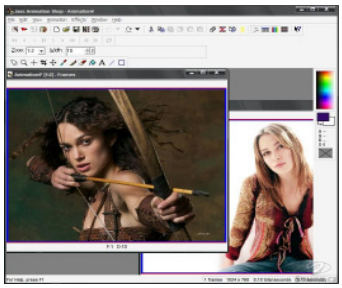


A program az animációs fájlformátumok mellett sok képformátumot támogat. A képekről és animációkról különböző információkat tudhatunk meg. Segítségével képeket és animációkat nézhetünk meg, egyszerűbb képszerkesztési műveleteket végezhetünk.

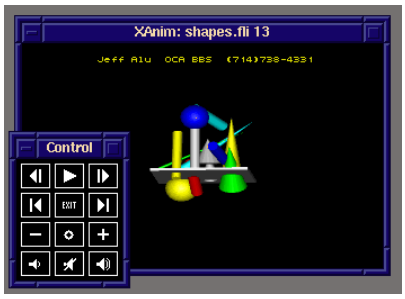
[3] Forrás: http://www.apple.com/quicktime/win.html
[4] Forrás: http://www.dolmendigital.es/asistencia/waaplay.zip
[5] Forrás: http://www.nyam.pe.kr/dev/imagine/download/Imagine_0.9.7.0.zip

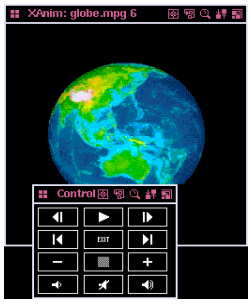
- az Animation Shop[6] (Windows 2000/XP);





- az Xanim[7] (X11 workstations);

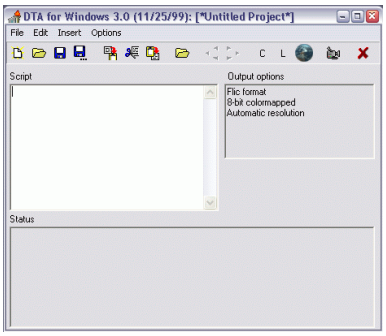




- az Imagen[8] (Windows 2000/XP).

Általában az FLI/FLC animációkészítő programok egy „köteg” képfájlból (melyek mindegyike a készítendő animáció egyetlen képkockáját tartalmazza) állítják össze az FLI/FLC animációt. Amióta az FLI/FLC állományok 8 bites színmélységgel rendelkeznek, általában van néhány olyan beállítási lehetőség, amely segítségével a 8 bitnél nagyobb színmélységgel rendelkező képekből az optimális színtérkép kiválasztható. Ilyen animáció-készítő program például:

- Dave’s Targa Animator[9] (DTA) (DOS, Windows)



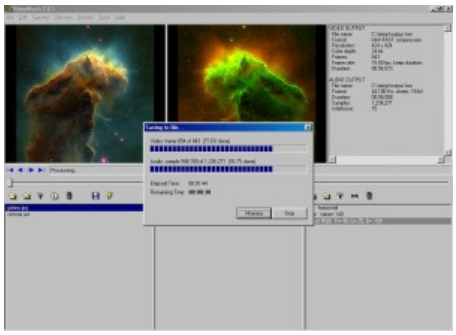
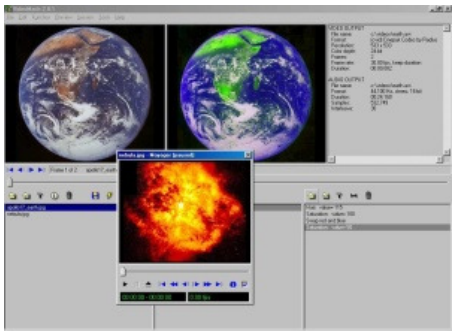
A program számos fájlformátumot támogat, többek közt TGA, PNG, PCX, AVI, GIF, JPG formátumokat, és meglehetősen sok beállítási lehetőséget kínál az animációkkal végzett munkához.

[6] Forrás: *http://www.jasc.com/products/animationshop?*
[7] Forrás: *http://xanim.polter.net/*
[8] Forrás: *http://www.gromada.com/imagen.html*
[9] Forrás: *http://www.povray.org/ftp/pub/povray/utilities/dta*

- DISPLAY[10] (DOS)

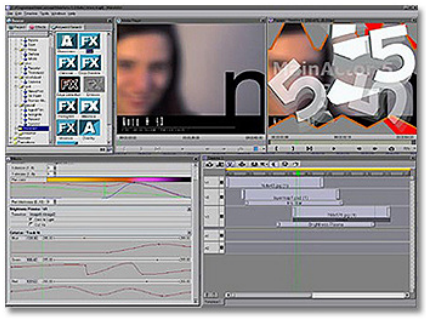
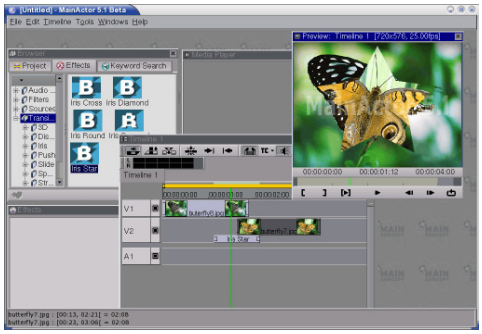
  A program a kép- és filmnéző funkciók mellett daraboló és átalalkító funkciókkal is rendelkezik.

- VideoMach[11] (régebbi neve: Fast Movie Processor) (Windows 98/2000/XP)

  Egyszerűen kezelhető konvertáló program, amely képkockákat tud FLC vagy más formátumokba (mint például AVI, MPEG) konvertálni.

- MainActor[12] (Linux, Windows 95/98/NT)

  Segítségével FLI/FLC, AVI, MPEG állományokat tudunk készíteni, valamint konvertálást is végezhetünk a különböző típusok között. Nagyon sok kódoló-dekódolót (*codec*) támogat.

- PPM2FLI[13] (UNIX)

  Bemenetként PPM, PGM, PBM vagy FBM fájlokat tud feldolgozni, és FLC animációkat tud készíteni belőlük. Valamint egy FLC dekólolót is tartalmaz, melynek segítségével az FLC animációkat különálló képekké tudja alakítani.

[10] Forrás: http://woodshole.er.usgs.gov/operations/modeling/download/disp183.zip
[11] Forrás: http://www.gromada.com/videomach.html
[12] Forrás: http://www.mainconcept.com/site/index.php?id=15
[13] Forrás: http://vento.pi.tu-berlin.de/STROEMUNGSAKUSTIK/SOFTWARE/ppm2fli/main.html

3. A FLIC fájlformátum

Az FLI és az FLC fájlformátumok az animációs fájlok osztályába sorolható FLIC fájlformátum kategóriába tartoznak. A FLIC fájlformátum kategóriába sorolható például még:

- az FLH és FLT fájlformátum (DTA);
- a CEL fájlformátum (Autodesk Animator);
- az FLX fájlformátum (Tempra Pro, Mathematica Inc. / U-Lead, 3DStudio MAX);
- az EGI által támogatott fájlformátumok (ITB CompuPhase).

Az eredeti FLIC fájlformátum leírását a fájlformátum megalkotója Jim Kent 1993 márciusában a Dr. Dobb's Journal[14] folyóiratban tette közzé. A formátum megjelenése óta eltelt időben az EGI több módosítást és bővítményt is hozzátett a formátumhoz, továbbá más kiterjesztésekkel és változtatásokkal is növekedett. A FLIC fájlformátumnak két elfogadott típusa van, az FLI és az FLC fájlformátum[15].

A régebbi az FLI formátum, mely esetén az animációk felbontásának maximális határa 320∗200 képpontra, a megjeleníthető színek száma 256 színre korlátozódik. Az FLC formátum tetszőleges felbontású animációt képes kezelni, de a legnagyobb alkalmazható színmélység továbbra is 8 bit (256 szín), bár a színpaletta sorról-sorra szabadon megváltoztatható, s így több szín biztosítható egy kép számára. Az FLC formátum támogatja a változtatható képkocka gyakoriságot, képkockákban megadva annak értékét. Rögzített képkocka gyakoriság (*frame rate*) esetén a képváltás értéke képkocka/millimásodpercben van megadva. Elméletileg minden FLC formátumú animációnak folytonosan ismételhetőnek kell lennie, de gyakorlatilag ezt a követelményt sokszor figyelmen kívül hagyják.

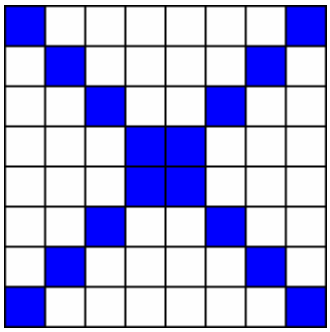
Az FLI állományok jellemzője, hogy felépítésük viszonylag összetett, de a mögöttük álló ötlet meglehetősen egyszerű: a következő képkockánál (*frame*) csak azokat a képpontokat (*pixel*) rajzoljuk ki újra, amelyek megváltoznak. Ezzel helyet és időt is meg lehet takarítani, mivel gyorsabb egy képpontot békén hagyni, mint egy újat megjeleníteni a helyén[16]. Az FLC formátum a régebbi FLI formátumhoz jobb tömörítést is adott. Mindkét fájlformátum

[14] Jim Kent: "The FLIC File format"; Dr. Dobb's Journal, March 1992 (Volume 18, issue 3)
[15] Ezen típusok neveivel ismerhetők fel az ugyanilyen kiterjesztésű fájlok is (pl.: earth.flc).
[16] FLI lejátszás: *http://www.prog.hu/cikkek/38/FLI+lejatszas.html*

veszteségmentes adattárolást tesz lehetővé[17]. A (kép)sorozat első képkockájára RLE tömörítést alkalmaznak, majd ezt követően a képkockáknak csak azon területeit tárolják és jelenítik meg, amelyek megváltoznak a rákövetkező képkockákban. Ezért jól alkalmazhatók rajzfilm és CGI[18] animációk esetén.

3.1. Betekintés az RLE tömörítési mód működésébe

Az RLE[19] tömörítési mód hatékony eljárás arra, hogy egyszerűen csökkentsük az olyan bitmap (bittérkép) fájlok méretét, melyek nagy kiterjedésű, azonos színű területeket tartalmaznak. Nem célravezető azonban olyan fényképek és más képek esetén, amelyek nem tartalmaznak nagyméretű egyszínű területeket. Működését a következő egyszerű példán keresztül tekintsük át. Az alábbi képet szeretnénk RLE tömörítési móddal tárolni:



Egyszerűen megoldható (tömörítés nékül) a tárolás, ha a képpontokat (vízszintes) soronként (ún. *scanline*) dolgozzuk fel, és minden egyes képpont színét a soron belüli előfordulás sorrendjében tároljuk. Amikor elérjük egy sor végét a következő sorral folytatjuk, egészen addig, amíg a sorok el nem fogynak. Az előbb leírtakat figyelembe véve a fenti képet eképpen tudnánk tárolni (még tömörítetlenül):

kék	fehér	fehér	fehér	fehér	fehér	fehér	kék
fehér	kék	fehér	fehér	fehér	fehér	kék	fehér
fehér	fehér	kék	fehér	fehér	kék	fehér	fehér

és így tovább, egészen az utolsó sor utolsó képpontjáig.

[17] EGI tömörítési séma: *http://www.compuphase.com/compress.htm*
[18] Számítógép generálta ábrázolás: *http://hu.wikipedia.org/wiki/CGI_(film)*
[19] Run-length compression: *http://www.graphicsacademy.com/what_rle.php*

Az RLE tömörítési módot használva a képet a következőképpen tudnánk tárolni:

1*kék	6*fehér	1*kék		
1*fehér	1*kék	4*fehér	1*kék	1*fehér
2*fehér	1*kék	2*fehér	1*kék	2*fehér

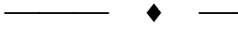
Az RLE tömörítési mód különösen népszerű formája a PackBits[20] tömörítés.

Az FLX formátum a hagyományos FLC formátumon történt csekély változtatás eredménye. Tulajdonképpen jelentéktelen eltéréssel két fajta FLX fájlformátum létezik. Mindkettő 32768 színt használ tömörített RGB[21] formátumban. Az egyik FLX formátumot a Mathematica Inc. definiálta a "Tempra Pro"[22] számára. A 3DStudio MAX (Discreet Inc.) által használt szintén FLX kiterjesztést kapott, de maga a formátum inkább a 15 bpp-s FLH fájlformátumhoz hasonló. Mindkét FLX fájlformátum az optimális megoldást adó RLE tömörítést használja, amit eredetileg a 8-bpp FLC fájlok számára terveztek.

Dave K. Mason DTA[23] programjához új tömböket definiált a tömörítés javítása, valamint a fájlformátum színmélységének rugalassabbá tétele érdekében. A megkülönböztethetőség kedvéért új formátumát egy új „típus” azonosítóval és új kiterjesztésekkel jelölte meg (FLH és FLT). A DTA tömböket és típusazonosítókat néhány más progam, nevezetesen az EGI szintén támogatja.

Végezetül -eltérő kiterjesztéssel és eggyel több tömbbel a „prefix adatok”[24] (*prefix data*) között- a CEL fájlok tulajdonképpen egyike az FLI vagy az FLC fájloknak.

A FLIC formátum tömbökből (*chunks*) áll, amelyek lehetővé teszik nagyméretű animációk lejátszását anélkül, hogy a teljes fájlt be kellene tölteni a memóriába.



[20] PackBits tömörítési mód: http://www.graphicsacademy.com/what_packbits.php
[21] Minden képpont színinformációja 2 bájton tárolódik (15 bit/pixel).
[22] Ez a formátum úgy is ismeretes, mint „Tempra FLX”.
[23] Dave's .TGA Animation Program (Copyright (c) 1991, 1992, 1993 by David K. Mason)
[24] Egyfajta opcionális feljéc.

4. A FLIC fájlformátum általános ismertetése

Egy FLIC fájl meghatározott számú képkockát tárol. Minden képkocka egy képet (*image*) (és esetleg egy színpalettát), egy címkét (*label*), illetve egyéb adatokat tartalmaz. Rendszerint egy FLIC fájlban a végén egy gyűrű képkocka (*ring frame*) is van, azért hogy az animáció újra meg újra lejátszható legyen anélkül, hogy érzékelhető lenne a megszakítás az utolsó képkocka és az első között. A szegmenseket tartalmazó FLIC fájl szegmensenként (EGI kiterjesztés) tartalmazhat egy gyűrű képkockát is. A FLIC fájlok tömb-hierarchia szerkezettel rendelkeznek. Egy tömb (*chunk*) egy fix (rögzített) és egy változó (módosítható) részt tartalmaz. Minden tömb rögzített része tartalmazza a típust és a tömb méretét. A tömb fennmaradó része nem rögzített formátummal rendelkezik, és függ a tömb típusától. A tömbös szerkezet célja az, hogy lehetővé tegye újabb tömbök hozzáadását anélkül, hogy a már meglevő lejátszóknak ez problémát okozna. Az olvasóprogram, amely nem tudja értelmezni valamely tömb típusát, képes egyszerűen átugrani azt (felhasználva a tömb fix részében található információt). A tömb fejlécében található méret mező lehetővé teszi, hogy a teljes, fel nem ismert tömböt egyszerűen figyelmen kívül hagyja az olvasóprogram.

Az alábbi ábra a FLIC fájl tömb-hierarchiaját mutatja be. Minden „szó” (*word*) (2 bájt) vagy „dupla szó” (*double word*) (4 bájt) érték tárolása Little Endian (ez az a bájt sorrend, amit az Intel 80x86 és a Pentium processzor sorozat használ) formátumú.

———— ♦ ————

A következő táblázat egy FLIC állomány szerkezetét mutatja be.

Pszeudokód	*Leírás*
`File header`	Általános állomány információ
`Prefix chunk`	Csak FLC fájlok esetén.
`Cel data chunk`	CEL fájloknál (megegyező az FLC fájlokéval)
`Frame chunk`	„Overlay frame”, csak EGI fájlok esetén.
`<image data>`	„Overlay” képpont adatok.
`Cel data chunk`	„Overlay” kezdeti és átlátszó szín.
`<label data>`	„Overlay” címke.
`Path map`	Csak EGI fájlok esetén.
`Segment table chunk`	Csak EGI fájlok esetén.
`Segment chunk`	Az összes szegmens információja, csak EGI fájlok esetén.
`<label data>`	A szegmensek szimbólikus nevei, csak EGI fájlok esetén.
`Huffman table chunk`	Csak EGI fájlok esetén.
`Script chunk`	Csak EGI fájlok esetén.
`Frame chunk`	Szabványos képkocka
`Postage stamp`	Ikon, csak FLC fájlok esetén.
`<image data>`	Tömörített vagy tömörítetlen.
`Cel data chunk`	Csak EGI fájlok esetén.
`<palette data>`	Szín adatok.
`<image data>`	Különféle módokon tömörítve.
`<mask data>`	Átlátszósági információk, csak EGI fájlok esetén.
`<label data>`	Szimbólikus vagy numerikus címkék, csak EGI fájloknál.
`Region chunk`	Csak EGI fájlok esetén.
`Wave audio chunk`	Digitalizált hang, csak EGI fájlok esetén.
`User string chunk`	Általános célú adatok, csak EGI fájlok esetén.
`Key palette`	Kulcs képkocka adatok, csak EGI fájlok esetén.
`Key image`	Kulcs képkocka adatok, csak EGI fájlok esetén.
`Ring frame chunk`	Az a képkocka, amely visszahurkol.
`<palette data>`	Lásd fentebb.
`<image data>`	Lásd fentebb.
`<mask data>`	Lásd fentebb.
`<label data>`	Lásd fentebb.
`Region chunk`	Lásd fentebb.
`Wave audio chunk`	Lásd fentebb.
`User string chunk`	Lásd fentebb.

A „<palette data>” valamelyik a kettő közül:	
"256" colour palette	Színpaletta 8-bpp RGB bejegyzésekkel.
"64" colour palette	Színpaletta 6-bpp RGB bejegyzésekkel.

Az „<image data>” a következők közül valamelyik:	
Black frame	Teljesen fekete képkocka.
Uncompressed full frame	Tömörítetlen képpont blokk.
Full frame	RLE tömörített, az EGI ezenfelül támogatja a Huffman/BWT tömörítést.
Delta frame (old style)	RLE tömörített.
Delta frame (new style)	RLE tömörített, az EGI ezenfelül támogatja a Huffman/BWT tömörítést.
HiColor/True Color frame	Csak DTA és EGI fájlok esetén

A „<mask data>” az alábbiak közül az egyik:	
Bitmap mask chunk	Csak EGI fájlok esetén.
Multilevel mask chunk	Csak EGI fájlok esetén.
Region mask chunk	Csak EGI fájlok esetén.

A „<label data>” valamelyik a kettő közül:	
Label chunk	Csak EGI fájlok esetén.
Extended label chunk	Csak EGI fájlok esetén.

4.1. Az állományfejrész

A FLIC fájlok elején 128 bájtos fejléc található. A típus mező a következőket tartalmazza:

0xAF11	FLI fájlok esetén.
0xAF12	Szabványos FLC fájlok esetén (8 bit színmélységgel rendelkeznek). Az FLX fájlok is ezt a fejléc típust használják.
0xAF44	Azon FLIC fájlok esetén, amelyeknek a színmélysége más (több) mint 8 bit. A (Dave K. Mason által készített) DTA program olyan FLIC fájlokat készít, amelyeknek a színmélysége 1, 15, 16 vagy 24 bit képpontonként. Az EGI 3.0 (és a későbbi) verziók a 16-bpp FLIC fájlokat támogatják.
0xAF30	Azon FLIC fájlok esetén, amelyek Huffman vagy BWT tömörítést használnak. Ezek a FLIC fájlok olyan színfelbontást használhatnak, amely eltér a 8-bpp értéktől.
0xAF31	Azon FLIC fájlok esetén, melyek „frame shift” tömörítést használnak. Ezek a FLIC fájlok használhatnak kiegészítő tömörítési módokat is, mint a Huffman és a BWT. Ezek a fájlok is rendelkezhetnek olyan színfelbontással, amely eltér a 8-bpp értéktől.
???	A jelszóval védett FLIC fájlok titkosítást leíró mezővel rendelkeznek, ezért a korábbi verziójú EGI és más FLIC olvasóprogramok nem megfelelő (vagy ismeretlen) FLIC fájlokként fogják felismerni őket.

Az FLI fájlok esetén két képkocka között a késleltetés 1/70 másodperc vagy annak egész számú többszöröse; a sebesség értékét az FLC fájloknál millimásodpercekben adják meg. Az első és a második képkockák eltolás értékét az FLI fájl fejléce nem tartalmazza; az első képkocka kezdete közvetlenül az állományfejrész után található (nincs "*prefix*" (előtag), "*segment table*" (szegmens tábla) vagy "*Huffman table*" (Huffman tábla) tömbök) és az első képkocka fejlécének beolvasása után a második képkockához az eltolás egyszerűen számítható. A második képkockához tartozó eltolás tárolásának az a célja, hogy a gyűrű képkocka lejátszása után megjelenítendő következő képkocka az animáció második képkockája legyen. Az Autodesk Animator Pro a FLIC fájl létrehozásának és utolsó módosításának dátumát és időbélyegét MS-DOS formátumban is tárolja. A FLIC állományfejrész „*frames*” mezője nem tartalmazza a gyűrű képkockát, amely segítségével vissza lehetne jutni az animáció elejére. Ha a FLIC fájl le van mentve lemezre, és helyesen van lezárva, akkor az Autodesk Animator Pro program a „*flags*” mező értékét 3-ra állítja. E mező értékét néhány más segédprogram nullára állítja.

Pszeudokód	*Hossz*	*Leírás*
`typedef struct {`		
`DWORD size;`	4	A teljes FLIC fájl mérete a fejlécet is beleszámítva.
`WORD type;`	2	Fájl típusa 0xAF11, 0xAF12, 0xAF30, 0xAF44, ...
`WORD frames;`	2	Az első szegmensben lévő képkockák száma.
`WORD width;`	2	A FLIC szélessége képpontokban.
`WORD height;`	2	A FLIC magassága képpontokban.
`WORD depth;`	2	Képpontonkénti bitek száma (általában 8).
`WORD flags;`	2	Az értéke 0-ra vagy 3-ra állítva.
`DWORD speed;`	4	A képkockák közötti késleltetés (msec-ban).
`WORD reserved1;`	2	Nem használt, az értéke mindig 0.
`DWORD created;`	4	A FLIC fájl létrehozásának dátuma (FLC esetén).
`DWORD creator;`	4	Sorozatszám vagy fordítóprogram ID (FLC-nél).
`DWORD updated;`	4	A FLIC fájl módosításának dátuma (FLC esetén).
`DWORD updater;`	4	Sorozatszám (FLC esetén, lásd a „creator” mezőt).
`WORD aspect_dx;`	2	A téglalap szélessége (FLC esetén).
`WORD aspect_dy;`	2	A téglalap magassága (FLC esetén).
`WORD ext_flags;`	2	EGI: jelzők EGI kiterjesztések esetén.
`WORD keyframes;`	2	EGI: kulcskép gyakorisága.
`WORD totalframes;`	2	EGI: az összes képkockák száma (szegmensek).
`DWORD req_memory;`	4	EGI: maximális tömb méret (tömörítetlen).
`WORD max_regions;`	2	EGI: max. régiók száma a CHK_REGION tömbben
`WORD transp_num;`	2	EGI: az átlátszósági szintek száma.
`BYTE reserved2[24];`	24	Az értéke mindig 0.
`DWORD oframe1;`	4	Az eltolás értéke a „frame1”-hez (FLC esetén).
`DWORD oframe2;`	4	Az eltolás értéke a „frame2”-höz (FLC esetén).
`BYTE reserved3[40];`	40	Nem használt, az értéke mindig 0.
`} FLIC_HEADER;`		

A FLIC állományfejrész (a teljes mérete 128 bájt)

4.2. A hierarchikus tömb fájl szerkezet

Az Animator Pro által ismert fájlok egy vagy több információt leíró tömböt tartalmaznak. Elméleti megközelítésben a tömb vezérlő információk és adatok együttese. Az ismert fájlformátumok többsége esetén a fejlécszerkezetet hierarchikus adattömbök követik. Minden tömb legalább 6 bájt hosszúságú fejléccel kezdődik. Az első 4 bájt a tömb hosszát tartalmazza (beleszámítva magát a fejlécet is, és ha léteznek, akkor tartalmazza az összes alsóbb szintű tömböt is). A következő 2 bájt egy azonosító „*szó*", amely a tömbben lévő adatok típusát adja meg. Néhány tömb fejléce hosszabb, mint 6 bájt, de a méret és az azonosító mezők mindig a fejléc első 6 bájtján foglalnak helyet.

Az Animator Pro által létrehozott animáció fájlban (mely hierarchikus tömb fájl szerkezetű) az adatok a következő sorrendben helyezkednek el:
animáció fájl:

⇒ elhagyható prefix (előtag) tömb:

- beállítások tömb (*settings chunk*)
- "cel" elhelyezés tömb (*cel placement chunk*)

⇒ 1. képkocka tömb (*frame1 chunk*):

- bélyeg tömb: bélyeg adatok (postage *stamp chunk: postage stamp data*)
- színpaletta tömb (*color palette chunk*)
- képpont adatok tömb (*pixel data chunk*)

⇒ 2. képkocka tömb (*frame 2 chunk*):

- képpont adatok tömbje (*pixel data chunk*)

⇒ 3. képkocka tömb (*frame 3 chunk*):

- színpaletta tömb (*color palette chunk*)
- képpont adatok tömb (*pixel data chunk*)

⇒ 4. képkocka tömb (*frame 4 chunk*):

- színpaletta tömb (*color palette chunk*)

⇒ "gyűrű" képkocka tömb (*ring frame chunk*):

- színpaletta tömb (*color palette chunk*)
- képpont adatok tömb (*pixel data chunk*)

A fájl 128 bájt hosszú fejlécet tartalmaz az elhagyható „*prefix*” tömböt követően, amit egy vagy több képkocka tömb követ (lásd az előbbi példában). A „*prefix*” tömb - ha létezik -, tartalmazza az Animator Pro beállítási információt, a „CEL” elhelyezési információt, és egyéb kiegészítő adatokat. Az animációban minden egyes képkocka részére létezik egy képkocka tömb (*frame chunk*). Az animáció képkockáinak a végén kiegészítésként egy „gyűrű” képkocka (*ring frame*) van. Minden képkocka tömb tartalmaz színpaletta információt és/vagy képpont adatokat. A „gyűrű” képkocka tartalmazza azt a delta-tömörített információt, amely ahhoz szükséges, hogy az utolsó képkockát az első képkockához vissza lehessen „hurkolni” (*loop back*). Ezt úgy lehet elképzelni, mintha eltérő módon tömörítve a „gyűrű” képkocka az első képkocka másolata lenne. Még az egyetlen képkockát tartalmazó FLIC fájlnak is tartalmaznia kell egy „gyűrű” képkockát.

——— ♦ ———

4.2.1. Az FLC fájl állományfejléce

Az FLC fájl egy 128 bájtos fejléccel kezdődik, melynek a leírása lentebb látható. Minden hosszúság és eltolás értéke bájtban van megadva. A fejléc mezőkben tárolt összes érték előjel nélküli.

Eltolás	*Hossz*	*Név*	*Leírás*
0	4	méret (size)	A teljes fájl mérete állományfejléccel együtt.
4	2	magic	Fájlformátum azonosító. Mindig AF12 (hexa) értékű.
6	2	képkockák (frames)	Az állományban lévő képkockák száma. Ebbe nincs beleszámítva a „gyűrű” képkocka. Az FLC fájlok maximum 4000 képkockát tartalmazhatnak.
8	2	szélesség (width)	A képernyő szélessége képpontokban.
10	2	magasság (height)	A képernyő magassága képpontokban.
12	2	színmélység (depth)	BPP (bit / pixel) –ben megadva. Mindig 8 az értéke.
14	2	jelzők (flags)	0003 (hexa) értékűre van beállítva a „gyűrű” képkocka lemezre írását követően. Ez azt mutatja, hogy a fájl helyesen van befejezve és lezárva.
16	4	sebesség (speed)	Lejátszás alatt az egyes képkockák közötti késleltetés értéke millimásodpercben.
20	2	fenntartott (reserved)	Nem használt „szó”, értéke mindig 0.
22	4	létrehozás (created)	A fájl létrehozásának dátuma és ideje MS-DOS formátumban.
26	4	létrehozó (creator)	Azon Animator Pro program sorozatszáma, mellyel a fájl létrehozása történt. Ha a fájl más programmal készült, akkor ez a mező 464C4942 (hexa) értékű.
30	4	frissítve (updated)	A fájl legfrisseb módosításának dátuma és időpontja MS-DOS formátumban.
34	4	frissítő (updater)	Azt mutatja meg, hogy ki frissítette legutoljára a fájl. Lásd a leírást a létrehozónál!
38	2	aspectx	Az x tengely képméretaránya, mellyel a fájl létrehozásra került.
40	2	aspecty	Az y tengely képméretaránya, mellyel a fájl létrehozása megtörtént. Leggyakrabban az x:y képméretarány 1:1. De ettől az értéktől eltérő képméretarány is megadható. Például a 320∗200 képfelbontás esetén ez az arány 6:5.
42	38	fenntartott (reserved)	Nem használt terület, végig 0 értékkel feltöltve.
80	4	oframe1	Az eltolás értéke a fájl kezdetétől számítva az első animáció képkocka tömbig.
80	4	oframe2	Az eltolás értéke a fájl kezdetétől számítva a második animáció képkocka tömbig. Ez az érték kerül alkalmazásra akkor, amikor a „gyűrű” képkockát a lejátszás alatt visszahurkoljuk a második képkockához.
88	40	fenntartott	Nem használt terület, végig 0 értékkel feltöltve.

4.2.2. Az FLC „prefix” tömb

Az elhagyható „*prefix*” tömb közvetlenül követheti az animáció állományfejrészét. A „*prefix*” tömb arra használható, hogy olyan kiegészítő adatokat tároljunk, amelyek nem játszanak közvetlen szerepet az animáció lejátszásában. A „*prefix*” tömb 16 bájt hosszúságú fejléccel kezdődik (amely szerkezetét tekintve azonos a képkocka fejlécével).

Eltolás	*Hossz*	*Név*	*Leírás*
0	4	méret (size)	A „prefix” tömb mérete, beleértve a fejlécet és az összes alsóbb szintű tömböt, ami követi.
4	2	típus (type)	A „prefix” tömb azonosítója. Az értéke mindig F100 (hexa).
6	2	tömbök (chunks)	A „prefix” tömbben lévő alsóbb szintű tömbök száma.
8	8	fenntartott (reserved)	Nem használt terület, végig 0 értékkel feltöltve.

A „prefix” tömb fejléce

Ahhoz, hogy meg lehessen határozni, hogy a „*prefix*” tömb létezik-e, be kell olvasni az állományfejrészt követő 16 bájt hosszúságú fejlécet. Ha a típus mező értéke F100 (hexa) értékű, akkor az a „*prefix*” tömb. Amennyiben ez az érték F1FA (hexa) érték, akkor az az első képkocka tömb, és nem létezik „*prefix*” tömb.

Megjegyzés: Az „Animator Pro”-tól eltérő progamok esetén egyáltalán nem szükséges olyan FLIC fájlok létrehozására, amelyek tartalmazzák a „*prefix*” tömböt. A programok a FLIC fájl beolvasásakor a „*prefix*” fejlécben szereplő méret értéket felhasználva figyelmen kívül tudják hagyni a „*prefix*” további részét, vagy az állományfejrészben lévő „*oframe1*” mező értékét felhasználva közvetlenül az első képkocka tömbre tudnak pozícionálni.

4.2.3. Az FLC képkocka tömbök

Az animáció számára a képpont és szín adatokat a képkocka tömbök tartalmazzák. A képkocka tömb több alsóbb szintű tömböt foglalhat magában, melyek közül minden egyes tömb az aktuális képkocka számára különböző típusú adatokat tartalmaz. Minden képkocka tömb a képkocka tartalmáról leírást adó 16 bájt hosszúságú fejléccel kezdődik.

Eltolás	*Hossz*	*Név*	*Leírás*
0	4	méret (size)	A képkocka tömb mérete, beleértve a fejlécet és az összes alsóbb szintű tömböt, ami követi.
4	2	típus (type)	A képkocka tömb azonosítója. Az értéke mindig F1FA (hexa) érték.
6	2	tömbök (chunks)	A képkocka tömbben lévő alsóbb szintű tömbök száma.
8	8	fenntartott (reserved)	Nem használt terület, végig 0 értékkel feltöltve.

A képkocka fejlécét közvetlenül a képkocka alsóbb szintű adat tömbjei követik. Amikor a képkocka fejlécében lévő tömbök mező értéke nulla, az azt mutatja, hogy ez a képkocka megegyezik a megelőző képkockával. Ez azt jelenti, hogy nem kell változtatni a képernyőn vagy a színpalettán, de a lejátszás közben a helyes késleltetés miatt mégis be kell illeszteni. A képkocka tömb belsejében minden adattömb a következő formátummal rendelkezik:

Eltolás	*Hossz*	*Név*	*Leírás*
0	4	méret (size)	A tömb mérete, beleszámítva a fejlécet is.
4	2	típus (type)	Adattípus azonosító.
6	(méret-6)	adatok (data)	Szín vagy képpont adatok.

A tömb fejlécében lévő típus mező értéke megmutatja, milyen típusú grafikus adatokat tartalmaz a tömb, valamint az adatok milyen tömörítési eljárással kódoltak. A következő

értékek (és a hozzájuk kapcsolódó mnemonikus nevek) találhatók meg jelenleg a képkocka adatok tömbjeiben:

Érték	*Név*	*Leírás*
4	FLI_COLOR256	256 szintű színpaletta információ.
7	FLI_SS2	„szó”-orientált delta tömörítés.
11	FLI_COLOR	64 szintű színpaletta információ.
12	FLI_LC	„bájt”-orientált delta tömörítés.
13	FLI_BLACK	Teljes képkocka szín indexe nulla.
15	FLI_BRUN	„byte run length” tömörítés
16	FLI_COPY	Nincs tömörítve.
18	FLI_PSTAMP	„Postage stamp” méretű kép.

——— ♦ ———

4.2.3.1. FLI_COLOR (11-es típusú) tömb

A régebbi FLI formátummal maximum 64 színt lehetett kezelni, ezeket a színinformációkat találjuk meg ebben a tömbben[25]. Ez a tömb, szerkezetét tekintve megegyezik az FLI_COLOR256 tömbbel, kivéve azt, hogy a színösszetevők értéktartománya a vörös, a zöld, és a kék színek számára nem 0-tól 255-ig terjed, hanem 0-tól 63-ig.

——— ♦ ———

4.2.3.2. FLI_COLOR256 (4-es típusú) tömb

A FLIC fájlformátum egy színtérképet (*color map*) használ arra, hogy az animáció számára színeket definiáljon. Az FLI formátumhoz képest az FLC formátummal kezelhető színek száma már maximum 256 lehet, és ebben a tömbben[26] ezeket a színinformációkat tárolja. Mind az FLI_COLOR, mind az FLI_COLOR256 tömb azonos felépítéssel rendelkezik.

[25] Más forrás ezt a tömböt „COLOR_64” néven említi: http://www.compuphase.com/flic.htm
[26] Egyes forrás ezt a tömböt „COLOR_256” néven említi: http://www.compuphase.com/flic.htm

Pszeudokód	*Leírás*
`typedef struct _ColormapChunk {`	
`  CHUNKHEADER Header;`	A tömb fejléce (6 bájt).
`  WORD NumberOfElements;`	Színelemek[27] száma a színtérképen.
`  Struct _ColorElement {`	
`    BYTE SkipCount;`	Színindex kihagyásszámláló.
`    BYTE ColorCount;`	A színek száma ebben az egységben.[28]
`    Struct _ColorComponent {`	
`      BYTE Red;`	Vörös színkomponens.
`      BYTE Green;`	Zöld színkomponens.
`      BYTE Blue;`	Kék színkomponens.
`    } ColorComponents[ColorCount];`	
`  } ColorElements[NumberOfElements];`	
`} COLORMAPCHUNK;`	

Az FLI_COLOR és az FLI_COLOR256 tömbök szerkezete

Az ilyen típusú tömbben lévő adatok csomagokba rendezettek. A tömb fejlécét követő első szó (*word*) (2 bájt) a tömbben található adatcsomagok számát tartalmazza. Ezt közvetlenül az adatcsomagok követik. Minden csomag egy 1 bájtos színindex kihagyásszámálóból[29] (*color index skip count*), egy 1 bájtos színszámlálóból[30] (*color count*), és minden definiált szín számára egy 3 bájtos színinformációból áll. A színindex kihagyásszámáló (*color index skip count*) azt mutatja meg, hogy mennyi színt kell átugrani.

A színindex 0 értéket vesz fel a tömb kezdetén. Egy csomagban található bármely szín feldolgozása előtt, az aktuális szín színindexéhez hozzáadódik a színindex kihagyásszámláló értéke. Ha a színszámláló bájt 0 értékű, azt jelzi, hogy az adatcsomagban 256 szín[31] fog következni. Minden szín számára 3 bájt definiálja a vörös, a zöld és a kék színösszetevőket ebben a sorrendben. Minden színösszetevő értéke 0-tól (kikapcsolva) 255-ig (teljes) változhat.

[27] Tulajdonképpen a tömb által tartalmazott adatcsomagok számát jelenti.
[28] Lényegében megmutatja, hogy mennyi színt kell kicserélni.
[29] Más forrás erre „skip count” néven hivatkozik: *http://www.compuphase.com/flic.htm*
[30] Egy másik forrás erre „copy count” néven hivatkozik: *http://www.compuphase.com/flic.htm*
[31] RGB színhármas (triplets)

A következő példa azt mutatja be, hogyan lehet 10 színből megváltoztatni a 2., 7., 8., 9. színeket:

Színindexek	0	1	2	3	4	5	6	7	8	9
Megváltozik			↑					↑	↑	↑

2	Két adatcsomag következik.
2, 1, r, g, b	[1.] Kettőt átlép (kihagy), egy megváltozik. [skip count] [copy count] [RGB triplets]
4, 3, r, g, b, r, g, b, r, g, b	[2.] Négyet átlép (kihagy), három megváltozik.

———— ♦ ————

4.2.3.3. FLI_BLACK (13-as típusú) tömb

A fejlécet követően ez a tömb semmilyen adatot nem tartalmaz. Az FLI_BLACK tömb az egyedüli, amely azon adatok képkockáját reprezentálja, amelyben az összes képpont a képkockához tartozó színtérképen 0 (rendszerint fekete) színindexre van beállítva. Egyedül a következő szerkezetű fejlécet tartalmazza:

Pszeudokód	*Leírás*
`typedef struct _BlackChunk {`	
`  CHUNKHEADER Header;`	Az FLI_BLACK tömb fejléce (6 bájt).
`} BLACKCHUNK;`	

Az FLI_BLACK tömb szerkezete

———— ♦ ————

4.2.3.4. FLI_COPY (16-os típusú) tömb

Ez a tömb tartalmazza a képkocka tömörítetlen képét. A tömb fejlécét pontosan annyi képpont követi, mint amennyi az animáció szélességének és az animáció magasságának a szorzata. Az adatok megadása a bal felső sarokban kezdődik, majd egy soron belül balról-jobbra halad, és a sorok megadásának sorrendje felülről-lefelé történik. Ilyen típusú tömb akkor keletkezik, amikor az előnyben részesített tömörítési mód (SS2 vagy BRUN) több adatot hoz létre, mint a tömörítetlen képkocka képe; ez viszonylag ritkán fordul elő.[32]

Pszeudokód	*Leírás*
`typedef struct _CopyChunk {`	
`  CHUNKHEADER Header;`	Az FLI_COPY tömb fejléce (6 bájt).
`  BYTE PixelData[];`	Strukturálatlan képpont adatok.
`} COPYCHUNK;`	

Az FLI_COPY tömb szerkezete

4.2.3.5. FLI_BRUN (15-ös típusú) tömb

A tömb fejlécét követő adatok a teljes képet bájthoz igazodó RLE[33] tömörített formában tartalmazzák. Rendszerint ezt a tömböt találjuk meg az animáció első képkockájában, vagy a „*postage stamp image*” tömbön belül.

Pszeudokód	*Leírás*
`typedef struct _ByteRunChunk {`	
`  CHUNKHEADER Header;`	Az FLI_BRUN tömb fejléce (6 bájt).
`  BYTE PixelData[];`	RLE tömörített képpont adatok
`} BYTERUNCHUNK;`	

Az FLI_BRUN tömb szerkezete

[32] Érdemes figyelembe venni, hogy az „FLX” fájlok 2 bájtot (15-BPP) használnak egy képpont tárolására.
[33] Byte Run Length tömörítés - byte oriented RLE

A kép minden sora külön-külön van tömörítve a kép legfelső részétől kezdődően. A tömbben az adatok sorokba vannak rendezve. Minden sor tömörített képpontok csomagjait foglalja magában. Az első sor a kép legfelső részén található, amit lefelé haladva a rákövetkező sorok követnek. A kép magasságát a tömbben lévő sorok száma adja meg. Minden sor első bájtja az adott sorban lévő adatcsomagok számát (*packet count*) adja meg. Valójában ez az FLI formátumból (kezdeti Animator programból) maradt meg, és figyelmen kívül szokták (kell) hagyni, mert már egy soron belül több, mint 255 adatcsomagot is el lehet helyezni (például FLC fájlok esetén). Helyette az animáció szélességét használják arra, hogy egy soron belül vezérelje az adatcsomagok dekódolását addig folytatva a beolvasást és feldolgozást, amíg a kitömörített képpontok száma meg nem egyezik a kép szélességével. Ezt követően haladnak tovább a következő sorra. Néhány lejátszóprogram azonban még a „*packet count*”-ra hagyatkozik, ezért a FLIC fordítóprogramoknak még mindig generálniuk kellene őket, és csak abban az esetben kellene nulla értékűre beállítaniuk, amikor a „*packet count*” ténylegesen meghaladja a 255 értéket. A „Byte Run” adatcsomagok esetén alkalmazott RLE tömörítési mód meglehetősen egyszerű. Minden adatcsomagnak az első bájtja a típus bájt, amely megmutatja, hogyan kell értelmezni az adatcsomag többi bájtját.

Minden RLE adatcsomag típus/méret (type/size)[34] bájtból áll, amit egy vagy több képpont követ. Ha a „*count*” bájt negatív értékű, akkor az abszolút értéke azon (a „*count*” bájtot követő) képpontok (vagy adatbájtok) száma, amelyeket az adatcsomagból az animációs képbe kell másolni. Ha a „*count*” bájt pozitív értékű, akkor annyiszor kell ismételni a „*count*” bájtot követő egyetlen képpontot (adatbájtot), amennyi a „*count*” bájt (abszolút) értéke.

4.2.3.6. FLI_LC (12-es típusú) tömb – „Byte Aligned Delta” tömörítés

Ezt a tömböt[35] az FLI típusú fájlok esetén használják; az újabb FLC formátum a „DELTA_FLC” tömbbel cserélte fel. A megelőző és az aktuális képkocka közötti különbségeket (*delta*) tartalmazza. A kezdeti Animator program ezt a tömörítési módot

[34] Más forrás ezt „count” bájtnak nevezi: http://www.compuphase.com/flic.htm
[35] Más forrásmű ezt a tömböt „DELTA_FLI” tömbnek nevezi: http://www.compuphase.com/flic.htm

használta, de nem az Animator Pro programnál jött létre. Ilyen típusú tömb megjelenhet olyan Animator Pro fájlban, amely eredetileg az Animator programmal készült, majd ezt követően néhány képkocka (de nem az összes) az Animator Pro programmal került módosításra.

A tömbben lévő első sor pozícióját a tömb fejlécét követő első 16 bites „*szó*” tartalmazza. Ez azon sorok darabszáma[36] (a kép tetejétől lefelé), amelyek változatlanok az előző képkockához képest. Például 320∗200-as felbontású képkocka esetén, ha csak a legalsó sorban (200.) történik változás, akkor ennek a 16 bites „*szó*”-nak az értéke 199 lesz. A tömbben lévő („megváltoztatandó”) sorok számát a második 16 bites „*szó*” tartalmazza. A sorok számára az adatok ezt a két „*szó*”-t követik.

P s z e u d o k ó d	***L e í r á s***
`typedef struct _DeltaFliChunk {`	
`  CHUNKHEADER Header;`	Az „FLI_LC” tömb fejléce.
`  WORD LinesToSkip;`	Kiinduló sorok száma, melyeket ki kell hagyni.
`  WORD NumberOfLines;`	Kódolt sorok száma.
Kódolt sor; annyi, mint amennyi a kódolt sorok száma.	
`  struct _Line {`	
`    BYTE NumberOfPackets;`	Adatcsomagok száma ebben a sorban.
`    BYTE LineSkipCount;`	Azon sorok száma, melyeket ki kell hagyni.
Kódolt adatcsomag; annyi, mint amennyi a csomagok száma a sorban.	
`    struct _Packet {`	
`      BYTE SkipCount;`	Azon képpontok száma, melyeket ki kell hagyni.
`      BYTE PacketType;`	Az adatcsomagon alkalmazott kódolás típusa.
`      BYTE PixelData[];`	Képpont adatok e tömb részére.
`    } Packet[NumberOfPackets];`	
`  } Lines[NumberOfLines];`	
`} DELTAFLICHUNK;`	

A „DELTA_FLI” tömb szerkezete

[36] Másszóval ez egy sor kihagyásszámláló („line skip count”).

Egy bájt hosszúságú adatcsomag számlálóval (*packet count*) kezdődik minden egyes sor. A BRUN tömörítéstől eltérően az adatcsomag számláló (*packet count*) nélkülözhetetlen, mert ezt a tömörítési módot csak 320∗200 felbontású fájlok esetén használják. Minden adatcsomag a csomag típus/méret bájtját követő egyetlen bájtból, az oszlop kihagyásból (*column skip*) áll. Ha az adatcsomag típus bájtja pozitív értékű, akkor ez az érték azon képpontok számát jelenti, melyeket a csomagból az animációs képbe kell másolni. Ha az adatcsomag típus bájtja negatív értékű, akkor egyetlen képpontot tartalmaz, melyet ismételni kell; az adatcsomag típus bájtjának abszolút értéke adja meg a képpont ismétléseinek a számát.

Megjegyzés: A BRUN tömörítéshez képest az LC tömörítésben az adatcsomag típus bájtoknak negatív/pozitív jelentése ellentétes. Ez lejátszás alatt jobb teljesítményt nyújt.

——— ♦ ———

4.2.3.7. FLI_SS2 (7-es típusú) tömb – „Word Aligned Delta” tömörítés

Ez a tömb[37] a „*DELTA_FLI*” tömb újabb vátozata, és minden „.FLC” fájlban megtalálható. Alapjában véve megegyezik a „*DELTA_FLI*” tömbbel.

Ez a tömb az egymást követő képkockák közötti különbségeket tartalmazza. Ez az a formátum, amit az Animator Pro program a leggyakrabban használ az animáció első képkockájától eltérő képkockák esetén. Hasonló a „*line coded delta*” (LC) tömörítéshez, de ez „*szó*” irányultságú, szemben a másikkal, ami „*bájt*” irányultságú.

[37] Más forrás ezt a tömböt „DELTA_FLC” tömbnek nevezi: http://www.compuphase.com/flic.htm

Pszeudokód	*Leírás*
`Typedef struct _DeltaFlcChunk {`	
`  CHUNKHEADER Header;`	Az „FLI_SS2” tömb fejléce.
`  WORD NumberOfLines;`	Kódolt sorok száma a tömbben.
Kódolt sor; annyi, mint amennyi a kódolt sorok száma.	
`  struct _Line {`	
`    WORD PacketCount;`	Azon sorok száma, melyeket ki kell hagyni.
`    Struct _Packet {`	
`      BYTE SkipCount;`	Azon képpontok száma, melyeket ki kell hagyni.
`      BYTE PacketType;`	Az adatcsomagon alkalmazott kódolás típusa.
`      WORD PixelData[];`	Képpont adatok e csomag részére.
`    } Packet[NumberOfPackets];`	
`  } Lines[NumberOfLines];`	
`} DELTAFLCCHUNK;`	

A „DELTA_FLC” tömb szerkezete

Az adatok sorokba, és minden sor adatcsomagokba van rendezve. Az első „*szó*” a tömbben lévő sorok számát tartalmazza a tömb fejlécet követő adatokban. Minden sor kezdődhet néhány elhagyható „*szó*”-val, amit abban az esetben használunk, ha sorokat akarunk kihagyni, és azért, hogy páratlan szélességű animációk esetén beállítsuk a sor utolsó bájtját. A sorban ezeket az elhagyható „*szavakat*” az adatcsomagok számlálója követi. A sorszámláló nem foglalja magában a kihagyott sorokat. A „*szó*” két legmagasabb értékű bitje arra használható, hogy segítségükkel meghatározzuk a „*szó*” tartalmát.

Bit 15	*Bit 14*	*Jelentése*
0	0	A „szó” a csomagszámlálót tartalmazza. Ezt a „szót” a csomagok követik. A csomagszámláló 0 értékű is lehet; ez abban az esetben történhet, ha csak az utolsó képpontja változik meg a sornak.
1	0	A legkisebb helyiértékű bájt tárolva van az aktuális sor utolsó bájtban. Ezt a „szót” mindig a csomagszámláló követi.
1	1	A „szó” a sor kihagyásszámlálót tartalmazza. A „szó” abszolút értéke megadja a kihagyott sorok számát.

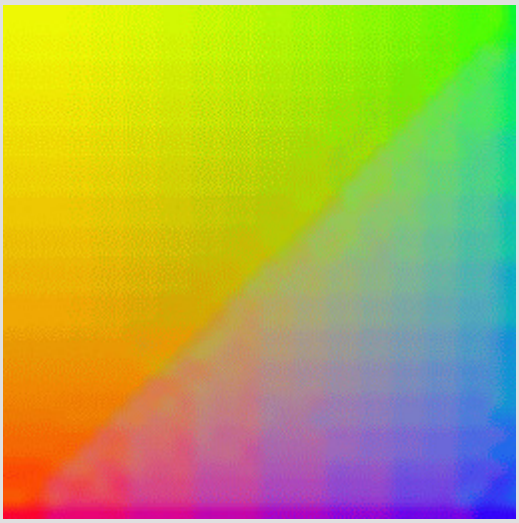
A sorokban lévő adatcsomagok hasonlóak a „*line coded*” (LC) tömbök esetén megtalálható adatcsomagokhoz. Minden adatcsomag első bájtja az oszlop kihagyásszámláló (*column skip count*) -, a második bájtja pedig a csomag típusát adja meg. Ha az adatcsomag típusát jelentő bájt értéke pozitív, akkor ez az érték azon „*szavak*” számát jelenti, amelyeket az adatcsomagból az animációs képbe kell másolni. Ha az adatcsomag típusát jelentő bájt negatív értékű, akkor a csomag még egy „*szó*”-t tartalmaz, melyet ismételni kell. A típust jelentő bájt negatív értékének az abszolút értéke adja meg, hogy hányszor kell a „*szó*”-t megismételni. Az ismételt „*szó*”-ban a legnagyobb és a legkisebb helyiértékű bájt nem szükségszerűen egyforma értékű.

——— ♦ ———

4.2.3.8. FLI_PSTAMP (18-as típusú) tömb

Az FLI_STAMP tömb tartalmazza a képkocka „*postage stamp*”-jét (a csökkentett méretű képet). A FLIC állományon belül általában csak az első képkocka tömbben jelenik meg. A „*postage stamp*” készítésekor az Animator Pro figyelembe veszi, azt hogy az optimális méret 100∗63 képpont. Azért, hogy az eredeti képméretarány megmaradjon, ha szükséges a tényleges méretet meg kell változtatni. Függetlenül a teljes képkocka képének színpaletta beállításaitól a „*postage stamp image*”-ben a képpontok egy „*six-cube*” színtérre vannak leképezve.

Pszeudokód	*Mintapélda*

start at palette entry 0
for red = 0 thru 5
  for green = 0 thru 5
  for blue = 0 thru 5
  palette_red = (red *256)/6
  palette_green = (green*256)/6
  palette_blue = (blue *256)/6
  move to next palette entry
  end for blue
  end for green
end for red
	

A „six-cube” színtér felépítése

Bármely tetszőleges RGB érték (amelyben minden komponens a 0-255 értéktartományban található) leképezhető a „*six-cube*” színtérre a következő összefüggés segítségével:

```
((6*red)/256)*36 + ((6*green)/256)*6 + ((6*blue)/256)
```

Amikor kiderül egy képkocka tömbről, hogy az FLI_PSTAMP tömb, akkor tudjuk, hogy a fejlécben a méret és a típus mezőkün kívül más mezők is találhatók.

Eltolás	*Hossz*	*N é v*	*L e í r á s*
0	4	méret (size)	A „postage stamp” tömb mérete, beleértve a fejlécet is.
4	2	típus (type)	A „postage stamp” azonosítója, az értéke mindig 18.
6	2	magasság (height)	A „postage stamp image” magassága képpontokban.
8	2	szélesség (width)	A „postage stamp image” szélessége képpontokban.
10	2	xlate	Szín transzláció típusa; az értéke mindig 1. A „six-cube” színteret jelzi.

A teljes „postage stamp” tömb fejléce

A „*postage stamp*” adatok közvetlenül ezen fejléc után következnek. Az adatok olyan formátummal rendelkeznek, mint egy tömb, amelynek szabályos méretet/típust tartalmazó fejléce van. A típus az alábbiak közül az egyik:

Érték	*N é v*	*L e í r á s*
15	FPS_BRUN	„Byte run length” tömörítés.
16	FPS_COPY	Nincs tömörítve.
18	FPS_XLAT256	„Six-cube” xlate színtábla.

Az FPS_BRUN és az FPS_COPY típusok megegyeznek az FLI_BRUN és az FLI_COPY kódolási módszerekkel. Az FPS_XLAT256 típus azt jelenti, hogy a tömb a képpont adatok helyett a 256 bájtos szín „*transaltion table*”-t tartalmazza. Az ilyen típusú „*postage stamp*” feldolgozásához be kell olvasni a képkocka teljes méretű képéhez tartozó képpont adatokat, és

a 256 bájtos színmegfeleltetési táblázat[38] segítségével megfelelteni őket a „six-cube” színtérben. Ilyen típusú „*postage stamp*” akkor jelenik meg, amikor az animáció képkockájának a mérete kisebb, mint standard 100∗63 „*postage stamp*” méret.

———— ♦ ————

4.2.3.9. SCRIPT_CHUNK tömb

Miután bepillantást kaptunk az FLC tömbökbe, érdekességképpen érdemes megemlíteni, hogy az EGI támogatja a PAWN[39] nyelven írt, a FLIC fájlokba beágyazott futásidejű „script”-eket. A lefordított P-kód hozzá van adva a FLIC fájlhoz. A FLIC animációban a futásidejű „script”-ekben megadott függvények meghatározott események bekövetkezése esetén hajtódnak végre, például a szegmens utolsó képkockájának elérésekor (ez az a képkocka, ami a gyűrű képkocka előtt található). Jelenleg az alábbi függvények definiáltak:

Függvény	*Leírás*
@FlicClose()	Akkor kerül meghívásra, amikor az animáció bezárult.
@FlicFrame(frame_number)	Akkor kerül meghívásra, amikor az összes képkocka dekódolt.
@FlicLabel(label)	Akkor kerül meghívásra, amikor elértük a megadott címkét.
@FlicLastFrame()	Akkor kerül meghívásra, amikor elértük az animáció aktuális szegmensének utolsó képkockáját.
@FlicLButtonDown(x, y)	Akkor kerül meghívásra, amikor az animált képkockán a bal egérgomb megnyomása megtörtént.
@FlicLButtonUp(x, y)	Akkor kerül meghívásra, amikor az animált képkockán a bal egérgomb felengedése megtörtént.
@FlicOpen()	Akkor kerül meghívásra, amikor az animáció megnyitásra került.
@FlicPlay()	Akkor kerül meghívásra, amikor az aimáció lejátszása elkezdődött.
@FlicRButtonDown(x, y)	Akkor kerül meghívásra, amikor az animált képkockán a jobb egérgomb megnyomása megtörtént.
@FlicRButtonUp(x, y)	Akkor kerül meghívásra, amikor az animált képkockán a jobb egérgomb felengedése megtörtént.

[38] Color translation table
[39] Forrás: http://www.compuphase.com/pawn/pawn.htm

@FlicSegment(segment_number)	Akkor kerül meghívásra, amikor éppen egy új szegmens kezdődött el.
@FlicStop()	Akkor kerül meghívásra, amikor az animációt leállítottuk (explicitly).

——— ♦ ———

5. Egy FLC állomány fejlécének szerkezete a gyakorlatban

A különböző tömbök áttekintését követően most egy rövid példán keresztül nézzük meg egy minta FLC állomány fejlécének felépítését. Ne felejtsük el, hogy a bájtok Little-endian módon vannak tárolva! A táblázatban a számok hexadecimális formában kerültek ábrázolásra, valamint a bájtok dupla szavanként (4 bájtonként) csoportosítottak.

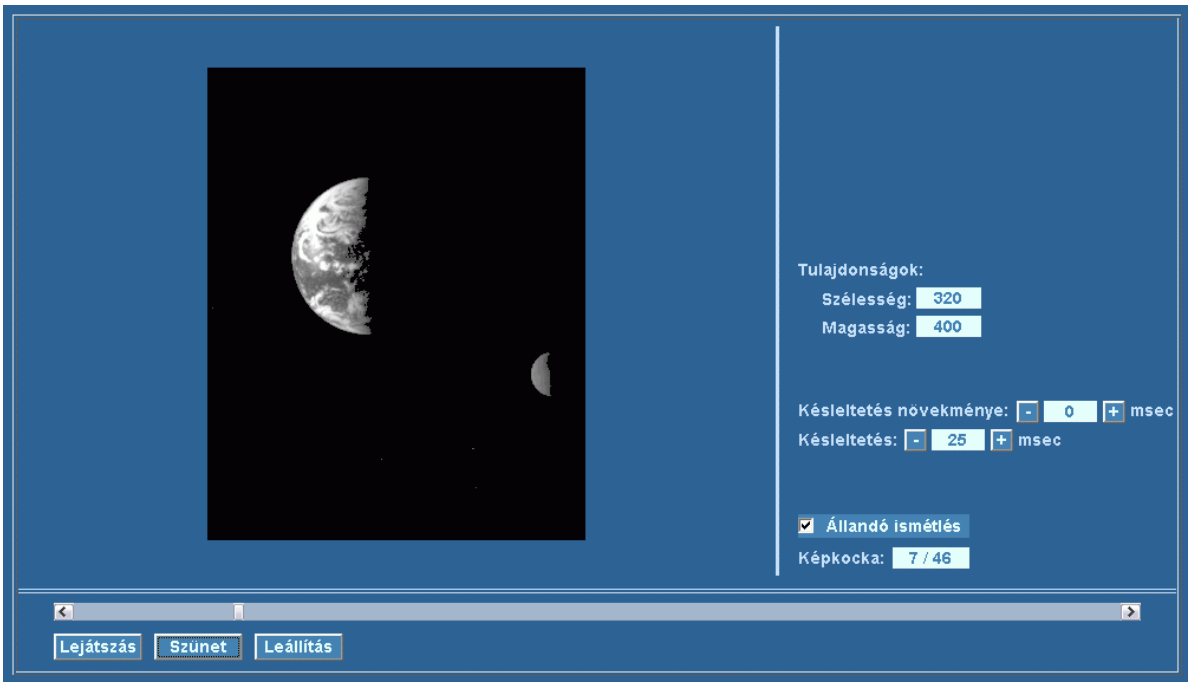
Bájtok	*FLC állomány fejléce (128 bájt)*			
0 - 15	BC D2 05 00	12 AF 2E 00	40 01 90 01	08 00 00 00
16 - 31	00 00 00 00	00 00 00 00	00 00 00 00	00 00 00 00
32 - 47	00 00 00 00	00 00 00 00	00 00 00 00	00 00 00 00
48 - 63	00 00 00 00	00 00 00 00	00 00 00 00	00 00 00 00
64 - 79	00 00 00 00	00 00 00 00	00 00 00 00	00 00 00 00
80 - 95	80 00 00 00	F6 27 00 00	00 00 00 00	00 00 00 00
96 - 111	00 00 00 00	00 00 00 00	00 00 00 00	00 00 00 00
112 - 128	00 00 00 00	00 00 00 00	00 00 00 00	00 00 00 00

Az előzőekben leírtak fényében most vizsgáljuk meg, hogy mit is jelentenek pontosan az egyes bájtcsoportok. A szimbólikus nevek jelentését már korábban, az FLC állomány fejlécének ismertetésekor áttekintettük, ezért erre most külön nem térünk ki. A minta FLC állomány legyen az „emconj.flc”[40] fájl, mely a dolgozat CD mellékletén is megtalálható.

[40] Forrás: *http://www.fileformat.info/format/fli/sample/1b8bf91714764f1fa729ce53d9bd3be6/EMCONJ.FLC*
http://netghost.narod.ru/gff/sample/images/fli/emconj.flc

Méret	*Hossz [bájt]*	*Szimbólikus név*	*Little-endian bájt sorend*	*Big-endian bájt sorrend*	*Jelentés*
DWORD	4	SIZE	BC D2 05 00	00 05 D2 BC	381 628 bájt
WORD	2	TYPE	12 AF	AF 12	FLC fájl
WORD	2	FRAMES	2E 00	00 2E	46 képkocka
WORD	2	WIDTH	40 01	01 40	320 pixel
WORD	2	HEIGHT	90 01	01 90	400 pixel
WORD	2	DEPTH	08 00	00 08	8 bit
DWORD	4	oframe1	80 00 00 00	00 00 00 80	128 bájt
DWORD	4	oframe2	F6 27 00 00	00 00 27 F6	10230 bájt

A fentiek alátámasztásaként ugyanezeket az értékeket találjuk meg a dolgozathoz elkészített lejátszóprogramban megnyitva az állományt:



6. Gyakori hibák

Számos olyan segédprogram, mellyel FLIC fájlokat lehet létrehozni, sajnos nem tesz eleget teljes mértékbenn a szabványnak. A legtöbb FLIC lejátszóprogram megpróbálja ezeket a hibákat felismerni és hibajelzés nélkül kijavítani. A FLIC fájlokban szereplő néhány leggyakoribb hiba a következő:

- ***A színmélység értéke nulla***

  A színmélység értékét rendszerint 8-ra szokták beállítani. A DTA és az EGI (a 3.0-ás verziótól kezdve) támogatja az olyan FLIC fájlok létrehozását is, melyeknek színmélysége eltér az általában megszokott értéktől, és egy másik „*típus*” értéket használnak a FLIC állományfejrészben a színmélység a jelzésére.

- ***A FLIC fájlban az eltolás értéke az első és a második képkockához nulla***

  Ez a hiba az FLC fejléccel rendelkező FLI fájlok esetén jelenik meg.

- ***Az eltolás értéke a második képkockához hibásan van beállítva***

  Vannak olyan FLIC fájlok, melyekben a második képkockához tartozó eltolás értéke nullára, az első képkockához tartozó eltolás értékére, a gyűrű képkockához tartozó eltolás értékére, vagy egy látszólag véletlen értékre van beállítva.

- ***A fájlban a DELTA_FLC tömbök FLI fejléccel szerepelnek***

  A legtöbb lejátszóprogam egyben FLI és FLC fájlok lejátszására is alkalmas, ez általában nem okozna problémát, azonban FLC fejléccel kellene rendelkezniük a DELTA_FLC tömbökkel rendelkező FLIC fájloknak.

- ***A tömbök mérete nincs páros számú bájtra felkerekítve***

  Csak nagyon kevés lejátszóprogramnak okoz problémát a páratlan számú bájttal rendelkező képkocka, de ez mégis ellentmond a szabványnak.

- ***A tömbben, illetve a fejlécben szereplő „fenntartott” mezőkben „szemét” van***

  A tömbben lévő „fenntartott” mezőket nulla értékűre kell beállítani. Ellenkező esetben a tömböket nehezen lehet feldolgozni.

- ***A DELTA_FLI tömb nincs helyesen dokumentálva***

  Az Autodesk-től származó dokumentum, - melyet kis módosítással a Dr. Dobb’s Journal folyóiratban jelentettek meg - azt állítja, hogy: „Minden sor két bájttal kezdődik. Az első bájt tartalmazza a soron belül az adatok kezdő x pozícióját, és a

második bájt a sor részére az adatcsomagok számát". Ezzel szemben minden sor egyetlen bájttal kezdődik, amely a sor részére az adatcsomagok számát tartalmazza.

——— ♦ ———

7. A lejátszóprogram

Olyan programozási nyelvet kellett választani a lejátszóprogram megírásához, amely biztosítja többek között:

- a forráskód újrafelhasználhatóságát, s így a már meglévő kódok, kódrészletek felhasználásával a gyorsabb, megbízhatóbb programírást;
- a forráskód és a már meglévő kódok szabadon terjeszthetővé tételét, hogy másoknak is lehetősége legyen a forráskód módosítására, bővítésére;
- böngésző program segítségével lefuttatni a lefordított forráskódot, és így az internet adta lehetőségeket kihasználva távoli gépeken lévő fájlokat lejátszani.

A következőkben tekintsük át a Java Applet-ként megírt FLI/FLC fájlok lejátszására alkalmas lejátszóprogramot. A lejátszóprogram a következőképpen működik:

1. az FLI/FLC fájl betöltése és szintaktikai elemzése;
2. az animáció minden egyes képkockájának rekonstruálása;
3. a képkockák lejátszása animációként egy szálban.

Számos előnnyel jár, ha az animációt képkockák sorozatára bontjuk, s ezt követően játszuk le. Mint például:

⇒ a kiszolgáló (szerver) oldalon kisebb memóriaigény;
⇒ jobb kezelhetőség (a kisebb részek jobban kezelhetők);
⇒ kevesebb az átvivendő adatok mennyisége;
⇒ nem szükséges kapcsolatot kiépíteni minden egyes képkocka esetén, ami csökkentené a betöltés sebességét.

De ez a megoldás hátrányokkal is jár:

⇒ a kliens oldalon külön munkát igényel a dekódolás.

Képek sorozatából FLI vagy FLC fájlokat a következő programok segítségével hozhatunk létre:

⇒ DTA és DTAX (MS-DOS környezetben),

⇒ fbm2fli (UNIX környezetben).

Ahhoz, hogy az FLI/FLC lejátszóprogramot használni lehessen a következő szöveget kell a web oldalon elhelyezni:

```
<APPLET CODE="Flic_Player.class" WIDTH=980 HEIGHT=550>
<PARAM NAME=WIDTH  VALUE=980>
<PARAM NAME=HEIGHT VALUE=550>
<PARAM NAME=FILENAME VALUE="emconj.flc">
<PARAM NAME=DELAY VALUE=25>
</APPLET>
```

Az animáció lejátszása közben két képkocka közötti késleltetést a(z) (elhagyható) "DELAY" paraméter adja meg millimásodpecben (1/1000 másodperc). Alapértelmezett értéke: 25 millimásodperc.

Ismert problémák:

⇒ problémát jelenthet, ha a „DELAY” paraméter értéke túl kicsi;

⇒ az, hogy a nagyméretű FLI/FLC fájlok lejátszása szaggatott lesz-e az alkalmazott operációs rendszertől, a számítógép hardverétől és/vagy a web böngészőtől függ.

Most röviden tekintsük át a lejátszóprogram kezelőfelületét.

A program kezelőfelülete, s így a program kezelése is meglehetősen egyszerű. Megjeleníti az animáció lejátszását vezérlő gombokon és görgetősávon kívül a lejátszandó állomány fontosabb információit is: az animáció szélességét, magasságát és az animációban található képkockák számát, valamint a képkockák közötti késleltetés mértékét. A késleltetés idejét, valamint a késleltetés változtatásának a mértékét külön lehet szabályozni. Az előbbit 0-999-ig, az utóbbit 0-100-ig terjedő értéktartományban. Külön állítható, hogy az animáció lejátszása folyamatosan törénjen vagy sem.

A következőkben tekintsük át a lejátszóprogram forráskódjának a felépítését. Az alábbi Java osztályokból épül fel a forráskód:

Java osztály	*L e í r á s*
Err	A lehetséges hibaüzeneteket tartalmazza.
Pixel_List	A képkocka adatok listájának felépítéséhez szükséges.
Flic_Player	Ez az osztály valósítja meg az Applet-et. Létrehozza a Scanner osztály egy példányát, amely segítségével feldolgozza az animációt tartalmazó fájlt.
Scanner	Ez az osztály szolgáltatja azokat a metódusokat, amelyek alkalmasak az FLI/FLC fájlokban kódolt adatok elemzésére (feldolgozására).
myProducer	A képek létrehozásához szükséges metódusokat tartalmazza.

Most pedig tekintsünk bele egy kicsit részletesebben néhány Java osztály, és azok metódusainak működésébe. A vizsgálódás tárgyának két Java osztályt célszerű kiemelni, az egyik a Flic_Player osztály, a másik pedig a Scanner osztály. Azért esett ezen osztályokra a választás, mert a Flic_Player osztály tartalmazza mindazon metódusokat, amelyek segítségével megvalósításra került maga az Applet, illetve azokat a részeket, amelyek az Applet működéséért felelősek. A Scanner osztály pedig azokat az adattagokat és metódusokat foglalja magában, amelyek segítségével az FLI/FLC fájlokban talalható kódolt adatokat fel lehet dolgozni, valamint azokhoz az információkhoz lehet hozzájutni, amelyek szükségesek a lejátszáshoz. Ilyen információk például a következők:

⇒ az animáció képkockáinak a száma;

⇒ a képkocka szélessége képpontokban;

⇒ a képkocka magassága képpontokban;

⇒ egyéb információk, amelyek egy kép előállításához szükségesek.

7.1. Betekintés néhány osztályba és azok működésébe

Mint arról, már az előzőekben szó esett, ez az osztály felelős az Applet megvalósításáért, ami többek között magában foglalja:

- ⇒ az Applet-hez tartozó felület komponenseinek a létrehozását, és ezen komponensek kezdeti paramétereinek a beállítását;
- ⇒ az Applet-hez tartozó komponensek működtetéséért felelős osztályokat, illetve metódusokat;
- ⇒ a Scanner osztály és más osztályok segítségével létrehozott képsorozat mozgóképként történő lejátszásához szükséges kódrészleteket.

7.1.1. Kiemelt kódrészletek

Először vessünk egy pillantást az Applet működéséért felelős ***Flic_Player*** osztályban lévő metódusokra. Az osztályban található adattagok nagy része az Applet megjelenítéséért, és az Applet komponenseinek működtetéséért felelős, ezért ezekre nem térnék ki külön. A következő táblázat az osztályban található metódusokat foglalja össze:

Metódus	*Leírás*
init	Az Applet osztályban lévő „init” metódust definiálja felül. Az Applet komponenseinek és a Flic_Player osztály adattagjainak a kezdeti paramétereit állítja be, valamint a komponensek működéséért felelős metódusok definícióit tartalmazza. A legfontosabb feladata a lejátszandó állomány adatainak a feldolgozása és a lejátszás előkészítése.
run	Az animáció lejátszását vezérli. Mivel az Applet szálként van megvalósítva, ezért ebben a metódusban kell elhelyezni a szál működését biztosító utasításokat.
start	Nincs más szerepe, csak létrehozza a szálat.
update	Az Applet osztályban lévő „update” metódust definiálja felül.
stop	A szálat szünteti meg.
paint	Az Applet osztály „paint” metódusát definiálja felül. Ez a metódus felelős az Applet működtetéséért.

A következőkben tekintsük át a ***Scanner*** osztályban található fontosabb adattagokat és metódusokat. A könnyebb áttekinthetőség miatt az ismertetésben egy-egy metódusnak csak azon része szerepel, amely a magyarázószövegben is megtalálható. Az osztályok és a metódusok teljes forráskódját a függelék tartalmazza.

Az alábbi kódrészletben azok a konstansok ismerhetők fel, melyek a korábban már említett, és részletesebben ismertetett tömb típusoknak felelnek meg:

```
final private int HEADER_LENGTH   = 26 + 102;
final private int FRAME_HEADER_LENGTH = 16;
final private short FLI_COLOR_256 = (short) 4;
final private short FLI_COLOR     = (short) 11;
final private short FLI_LC        = (short) 12;
final private short FLI_WORD_LC   = (short) 7;
final private short FLI_BLACK     = (short) 13;
final private short FLI_BRUN      = (short) 15;
final private short FLI_COPY      = (short) 16;
```

A Scanner osztály ***scan_flic*** metósusa végzi az FLI/FLC állományfejrészének beolvasását, valamint a szélesség (*width*) és a magasság (*height*) adattagok beállítását. A ***ReadContent*** metódus segítségével az állományfejrészt egy pufferbe olvassa be. Majd beállítja a megfelelő adattagok értékeit a pufferből a ***to_int*** metódus felhasználásával.

```
public boolean scan_flic(){
  int length;
  short magic, size, speed;

  buffer = new byte[HEADER_LENGTH];
  try {
    ReadContent(In_str, buffer, HEADER_LENGTH);
  }
  . . .
  length =         to_int(buffer, 4,  0);
  magic  = (short) to_int(buffer, 2,  4);
  size   = (short) to_int(buffer, 2,  6);
  width  = (short) to_int(buffer, 2,  8);
  height = (short) to_int(buffer, 2, 10);
  speed  = (short) to_int(buffer, 2, 16);
  . . .
  return io;
}
```

A ***ReadContent*** metódus segítségével a ***scan_chunk()*** metódus egy tömböt olvas be (hasonlóan az előzőleg látott módhoz), és a beolvasott tömb típusától függően (a már korábban említett konstansokat felhasználva) a tömben található adatok feldolgozásához különböző metódusokat fog meghívni.

```
private void scan_chunk(){
  int ch_size;
  short ch_type;

  buffer = new byte[6];
  try {
    ReadContent( In_str, buffer, 6 );
  }
  . . .
  ch_size =         to_int(buffer, 4, 0);
  ch_type = (short) to_int(buffer, 2, 4);
  . . .
  switch (ch_type) { // dectect chunk type
    case FLI_COLOR:       scan_FLI_COLOR(2);  break;
    case FLI_COLOR_256:   scan_FLI_COLOR(0);  break;
    case FLI_LC:          scan_FLI_LC();      break;
    case FLI_WORD_LC:     scan_FLI_WORD_LC(); break;
    case FLI_BRUN:        scan_FLI_BRUN();    break;
    case FLI_BLACK:   break;
    case FLI_COPY:    break;
    default: Err.Msg = "chunk type " + ch_type + " unknown";
    io = false;
    return;
  }
}
```

Az előzőekben látott különböző típusú tömbök esetén az adott típusú tömbnek megfelelő metódus kerül meghívásra. Most ezek közül vizsgáljunk meg néhányat. Az első legyen a ***scan_FLI_BRUN()*** metódus, amely az FLI_BRUN adatokat olvassa be. Ezek az adatok egy képet írnak le képpontról képpontra. Az adatokat tovább lehet osztani adatcsomagokba, ez látható az alábbi kódrészletben.

```
private void scan_FLI_BRUN(){
  int idx = 0;  // az eltolás értéke az adat pufferben
  int ppx;      // az eltolás értéke a képkockában
  int pkts;     // adatcsomagok száma a soron belül
  int xchpx;    // aktuális képpont érték
  int pixel;    // képpont számláló
  int line;     // aktuális sor
  byte f;

  short line_count = (short) height;
  . . .
  // a képkockákat tartalmazó lista első elemének az előállítása:
  last = anchor = new Pixel_List( null, myModel, width, height );
  frame_counter++;
  idx = 0;

  for( line = 0; line < line_count; line++ ){
    pkts = ( 0xff & buffer[idx++] );
    ppx  = line * width;
    for( int pkt_nr = 0; pkt_nr < pkts; pkt_nr++ ){
      xchpx = (int) buffer[idx++];
      if ( xchpx >= 0 ){
        f = (byte) (0xff & buffer[idx++]);
        for( int k = 0; k < xchpx; k++ ){
          anchor.color_values[ppx++] = f;
        }
      }
      else {
        xchpx = -xchpx;
        for( pixel = 0; pixel < xchpx; pixel++ ){
          anchor.color_values[ppx++] = (byte) ( 0xff & buffer[idx++] );
        }
      }
    }
  }
}
```

A következő metódus legyen a ***scan_FLI_LC()***, amely a FLIC fájlokban található FLI_LC adatokat „ismeri fel”. Ezek az adatok az aktuális képkocka és a megelőző képkocka közötti különbségeket írják le. Minden egyes sor adatait (az előbbihez hasonlóan módon) adatcsomagokba lehet rendezni. Az alábbi kódrészlet ezt szemlélteti:

```
private void scan_FLI_LC(){
  int idx = 0;  // az eltolás értéke a pufferben
  int ppx;      // az eltolás értéke a képkockában
  int pkts;     // adatcsomagok száma a soron belül
  int xchpx;    // jelenlegi érték
  int pixel;    // képpont számláló
  int line;     // aktuális sor
  short line_count;
  byte f;
  idx = 0;
  . . .
  // a képkockákat tartalmazó lista következő elemének előállítása
  last.next = new Pixel_List(last, myModel, width, height);
  frame_counter++;
  last = last.next;
  line       =         to_int(buffer, 2, idx);  idx += 2;
  line_count = (short) to_int(buffer, 2, idx);  idx += 2;

  for( ; line < line_count; line++ ){
    pkts = (0xff & buffer[idx++]);
    ppx = line * width;
    for(int pkt_nr = 0; pkt_nr < pkts; pkt_nr++){
      ppx += (0xff & buffer[idx++]);  // képpontok átlépése
      xchpx = (int) buffer[idx++];
      if (xchpx < 0) {
        xchpx = -xchpx;
        f = (byte) (0xff & buffer[idx++]);
        for (int k = 0; k < xchpx; k++) {
          last.color_values[ppx++] = f;
        }
      }
      else{
        for( pixel = 0; pixel < xchpx; pixel++ ){
          last.color_values[ppx++] = (byte) (0xff & buffer[idx++]);
        }
      }
    }
  }
}
```

Összefoglalás

Bár a FLIC fájlformátumok ma már meglehetősen réginek tűnhetnek, mégis számos területen lehet még most is alkalmazni őket. Főleg olyan területeken, ahol nem szükséges, hogy nagyon részletgazdag animációkkal dolgozzunk. Ilyen lehet például egy meteorológiai térkép, vagy fizikai jelenségek (például hullámformák) szemléltetése, esetleg költöző madarak vagy bálnák vándorlását nyomonkövető térképek, vagy egy adott terület éves hőmérséklet-ingadozása. Az interneten számos példát találhatunk arra, ahol ténylegesen ilyen célra használják a FLIC formátumú animációkat. További előnye még a FLIC formátumnak, hogy az előbb felsorolt animációkat igen kis méretben tudja tárolni.

Bízom benne, hogy a dolgozatomban a fájlformátumot sikerült könnyen érthetően ismertetnem, és felkeltettem az érdeklődését mindazoknak, akik még nem ismerték ezt a formátumot. Estleg más animációs formátumot már ismernek ugyan de látólörüket tudtam hasznos ismeretekkel bővíteni. A lejátszóprogramot a későbbiekben szeretném még kibővíteni, hogy lehessen:

- a lejátszóprogramban megadni a forrásállományt;
- lejátszási listában kezelni a megnézni kívánt állományokat;
- más fájlformátumok kezelésére felkészíteni a lejátszóprogramot (pl. MPEG);
- az animáció képkockáit kimenteni;
- az EGI által készített kiterjesztéseket részben vagy egészben beépíteni a lejátszó-programba.

Köszönetnyilvánítás

Ezúton szeretném megköszönni témavezetőmnek, Dr. Boda Istvánnak a szakdolgozat elkészítésében nyújtott segítségét, útmutatásait, értékes tanácsait, és türelmét.

Nagyon sok köszönettel tartozom Édesanyámnak és Keresztanyámnak a rengeteg türelemért és építő támogatásukért. Segítségük, bíztatásuk nélkül a dolgozat biztosan nem készült volna el. Itt szeretnék köszönetet mondani Édesanyámnak a tanulmányaim során nyújtott szerető támogatásáért és türelméért.

S végül, de nem utolsó sorban köszönettel tartozom AtomAnt barátomnak és ismerőseimnek, akik rengeteget segítettek az angol nyelvű dokumentációk fordításában. Mindazoknak, akik segítettek szakdolgozatom elkészítésében köszönöm szépen a támogatásukat, megértésüket, s türelmüket.

Irodalomjegyzék

Internetes források, elektronikus média-hivatkozások:

[1] ai_lab

Animator Pro File Formats, 2006. nov. 8

http://mediasrv.ns.ac.yu/extra/fileformat/animation/flc/flc.rtf

[2] ITB CompuPhase

The FLIC file format, 2006. 08. 14

http://www.compuphase.com/flic.htm

[3] FileFormat.Info

FLI File Format Summary, 2007

http://www.fileformat.info/format/fli/

[4] Kuba Attila

FLI lejátszás, 2003. nov. 28

http://www.inf.u-szeged.hu/oktatas/jegyzetek/KubaAttila/animacio/9_fli.htm

[5] Kumm

FLI lejátszás, 1997. okt. 4

http://www.prog.hu/cikkek/38/FLI+lejatszas.html

[6] J. Anders

Inline FLI/FLC- player in JAVA, 2004. okt. 25

http://vsr.informatik.tu-chemnitz.de/~jan/FLI/FlicFile.html

http://rnvs.informatik.tu-chemnitz.de/%7Ejan/FLI/Flic_Player.java

Függelék

A lejátszóprogram teljes forráskódja:

```
import java.io.*;
import java.net.*;
import java.applet.Applet;
import java.awt.event.*;
import java.awt.*;
import java.awt.image.IndexColorModel;
import java.awt.image.ColorModel;
import java.awt.image.ImageProducer;
import java.awt.image.ImageConsumer;

class Err { /* Ez az osztály a lehetséges hibaüzeneteket tartalmazza. */
  public static String Msg = null;
}

class Pixel_List {                /* képkockaadatok listájához */
  public Pixel_List next = null;  /* egy elem képkockánként    */
  public byte color_values[];
  public IndexColorModel curr_Model;    /* aktuális színmodell */

  Pixel_List ( Pixel_List predecessor, IndexColorModel m, int width, int
               height ) {
    color_values = new byte[ width * height ];
    if ( m != null ) { curr_Model = m; }
    if ( predecessor != null ) {
      System.arraycopy( predecessor.color_values, 0, color_values, 0,
                        color_values.length );
      /* Ha nincs színmodell definiálva, akkor az aktuális színmodell a
         megelőző képkocka színmodellje lesz.*/
      if ( m == null ) curr_Model = predecessor.curr_Model;
      predecessor.next = this;
    }
  }
}

public class Flic_Player extends Applet implements Runnable {
  private int curr_nr = 0;      /* a lejátszás alatti aktuális képkocka */
  private Scanner scan;
  private Thread AnimatorThread;

  boolean painted = false;    /* az értéke "true", ha a "curr_nr"
                                 (aktuális) képkocka ki lett rajzolva */
  private int delay = 200;    /* két képkocka közötti késleltetés ms-ban */
  private Image Images[];     /* az animáció képkockáinak sorozata */
  private boolean io = true;  /* hiba esetén az értéke "false" */

  private Color playerbackground = new Color( 46, 96, 146 );
  private Color foregroundcolor  = new Color( 198, 226, 255 );
  private Color buttonBackground = new Color( 70, 130, 180 );
  private Color buttonForeground = new Color( 224, 255, 255 );
  private Color seekBarBackground = new Color( 162, 181, 205 );
  private Font fontType1 = new Font("Courier", Font.BOLD+Font.ITALIC, 17);
  private Font fontType2 = new Font("Courier", Font.BOLD, 14);
```

```
private Cursor kurzor = new Cursor( 12 );

private int xmax = 640;
private int ymax = 480;
private int szelesseg = xmax;
private int magassag = ymax;
private int dx = 0;
private int dy = 0;

private Scrollbar seekBar = new Scrollbar(Scrollbar.HORIZONTAL, 0, 10, 0,
                                          1010);
private int frame_nr  = 0;
private int frame_cnt = 0;
private int increment = 0;

private Button playButton  = new Button( "Lejátszás" );
private Button pauseButton = new Button( "Szünet" );
private Button stopButton  = new Button( "Leállítás" );

private boolean isPlayed = false;
private boolean isPaused = false;
private boolean isStoped = false;

private Label kepKockak    = new Label();
private Label kepSzelesseg = new Label();
private Label kepMagassag  = new Label();

private Checkbox loop   = new Checkbox("  Állandó ismétlés");

private boolean theEnd      = false;  /* A lejátszás véget ért-e? */
private boolean neverend    = true;   /* Állandóan ismételjen-e?  */
private boolean neverendnew = true;

private Label kesleltetes           = new Label();
private Label kesleltetesNovekmenye = new Label();
private int delayIncrement = 0;
private Button plusDelayButton     = new Button( "+" );
private Button minusDelayButton    = new Button( "-" );
private Button plusDelayIncButton  = new Button( "+" );
private Button minusDelayIncButton = new Button( "-" );

public void init () {
  setLayout( null );
  setBackground( playerbackground );
  setFont( fontType1 );

  seekBar.setBackground( seekBarBackground );
  seekBar.setCursor( kurzor ); seekBar.setValue( 0 );
  seekBar.setBounds(30, 493, 920, 15);

  seekBar.addAdjustmentListener( new AdjustmentListener(){
    public void adjustmentValueChanged(AdjustmentEvent ev){
      curr_nr = (int) Math.round( (frame_nr * ev.getValue()) / 1000.0 );
      frame_cnt = curr_nr;
      repaint();
    }
  });
  add( seekBar );
```

```
playButton.setFont( fontType2 );
playButton.setBackground( buttonBackground );
playButton.setForeground( buttonForeground );
playButton.setCursor( kurzor );
playButton.setBounds( 30, 519, 75, 23 );
playButton.setActionCommand("play");
playButton.addActionListener( new ActionListener(){
  public void actionPerformed(ActionEvent ev){
    String ac = ev.getActionCommand();
    if( ac.equals("play") ){
      if( !isPlayed ){
        isPlayed = true; isPaused = false; isStoped = false;
      }
      if( !isPlayed && isPaused ){
        isPlayed = true; isPaused = false;
      }
    }
  }
});

pauseButton.setFont( fontType2 );
pauseButton.setBackground( buttonBackground );
pauseButton.setForeground( buttonForeground );
pauseButton.setCursor( kurzor );
pauseButton.setBounds( 115, 519, 75, 23 );
pauseButton.setActionCommand("pause");
pauseButton.addActionListener( new ActionListener(){
  public void actionPerformed(ActionEvent ev){
    String ac = ev.getActionCommand();
    if( ac.equals("pause") ){
      if( isPlayed && !isPaused ){
        isPaused = true; isPlayed = false;
      }
    }
  }
});

stopButton.setFont( fontType2 );
stopButton.setBackground( buttonBackground );
stopButton.setForeground( buttonForeground );
stopButton.setCursor( kurzor );
stopButton.setBounds( 200, 519, 75, 23 );
stopButton.setActionCommand("stop");
stopButton.addActionListener( new ActionListener(){
  public void actionPerformed(ActionEvent ev){
    String ac = ev.getActionCommand();
    if( ac.equals("stop") ){
      if( isPlayed && !isStoped ){
        isStoped = true;
        isPlayed = false;
        curr_nr = 0;
        frame_cnt = 0;
        seekBar.setValue( 0 );
        if( neverend == true ){ neverendnew = false; }
      }
    }
  }
});
```

```
add( playButton ); add( pauseButton ); add( stopButton );

kepKockak.setBackground( buttonForeground );
kepKockak.setForeground( buttonBackground );
kepKockak.setFont( fontType2 ); kepKockak.setAlignment( Label.CENTER );
kepKockak.setBounds( xmax+100, ymax-34, 65, 18 );
add( kepKockak );

kepSzelesseg.setBackground( buttonForeground );
kepSzelesseg.setForeground( buttonBackground );
kepSzelesseg.setFont( fontType2 );
kepSzelesseg.setAlignment( Label.CENTER );
kepSzelesseg.setBounds( xmax+120, ymax-254, 55, 18 );
add( kepSzelesseg );

kepMagassag.setBackground( buttonForeground );
kepMagassag.setForeground( buttonBackground );
kepMagassag.setFont( fontType2 );
kepMagassag.setAlignment( Label.CENTER );
kepMagassag.setBounds( xmax+120, ymax-230, 55, 18 );
add( kepMagassag );

loop.setState( true ); loop.setFont( fontType2 );
loop.setBackground( buttonBackground );
loop.setForeground( buttonForeground );
loop.setBounds( xmax+20, ymax-62, 145, 20 );
loop.addItemListener( new ItemListener(){
  public void itemStateChanged(ItemEvent ie){
    if( ie.getStateChange() == ItemEvent.DESELECTED ){
      neverend = false; repaint();
    }
    if( ie.getStateChange() == ItemEvent.SELECTED ){
      neverend = true; repaint();
    }
  }
});
add( loop );

kesleltetes.setBackground( buttonForeground );
kesleltetes.setForeground( buttonBackground );
kesleltetes.setFont( fontType2 );
kesleltetes.setAlignment( Label.CENTER );
kesleltetes.setBounds( xmax+133, ymax-134, 45, 18 );
add( kesleltetes );

kesleltetesNovekmenye.setBackground( buttonForeground );
kesleltetesNovekmenye.setForeground( buttonBackground );
kesleltetesNovekmenye.setFont( fontType2 );
kesleltetesNovekmenye.setAlignment( Label.CENTER );
kesleltetesNovekmenye.setBounds( xmax+228, ymax-158, 45, 18 );
add( kesleltetesNovekmenye );

minusDelayButton.setFont( fontType2 );
minusDelayButton.setCursor( kurzor );
minusDelayButton.setBackground( buttonBackground );
minusDelayButton.setForeground( buttonForeground );
minusDelayButton.setBounds( xmax+110, ymax-134, 19, 19 );
minusDelayButton.setActionCommand("minusDelay");
minusDelayButton.addActionListener( new ActionListener(){
```

```
  public void actionPerformed(ActionEvent ev){
    String ac = ev.getActionCommand();
    if( ac.equals("minusDelay") ){
      if( (delay-delayIncrement) > 0 ) {
        delay -= delayIncrement; repaint();
      }
    }
  }
});

plusDelayButton.setFont( fontType2 );
plusDelayButton.setCursor( kurzor );
plusDelayButton.setBackground( buttonBackground );
plusDelayButton.setForeground( buttonForeground );
plusDelayButton.setBounds( xmax+183, ymax-134, 19, 19 );
plusDelayButton.setActionCommand("plusDelay");
plusDelayButton.addActionListener( new ActionListener(){
  public void actionPerformed(ActionEvent ev){
    String ac = ev.getActionCommand();
    if( ac.equals("plusDelay") ){
      if( (delay+delayIncrement) < 1000 ){
        delay += delayIncrement; repaint();
      }
    }
  }
});

minusDelayIncButton.setFont( fontType2 );
minusDelayIncButton.setCursor( kurzor );
minusDelayIncButton.setBackground( buttonBackground );
minusDelayIncButton.setForeground( buttonForeground );
minusDelayIncButton.setBounds( xmax+205, ymax-158, 19, 19 );
minusDelayIncButton.setActionCommand("minusDelayInc");
minusDelayIncButton.addActionListener( new ActionListener(){
  public void actionPerformed(ActionEvent ev){
    String ac = ev.getActionCommand();
    if( ac.equals("minusDelayInc") ){
      if( delayIncrement > 0 ){ delayIncrement--; repaint(); }
    }
  }
});

plusDelayIncButton.setFont( fontType2 );
plusDelayIncButton.setCursor( kurzor );
plusDelayIncButton.setBackground( buttonBackground );
plusDelayIncButton.setForeground( buttonForeground );
plusDelayIncButton.setBounds( xmax+278, ymax-158, 19, 19 );
plusDelayIncButton.setActionCommand("plusDelayInc");
plusDelayIncButton.addActionListener( new ActionListener(){
  public void actionPerformed(ActionEvent ev){
    String ac = ev.getActionCommand();
    if( ac.equals("plusDelayInc") ){
      if( delayIncrement < 100 ){ delayIncrement++; repaint(); }
    }
  }
});

add( plusDelayButton ); add( minusDelayButton );
add( plusDelayIncButton ); add( minusDelayIncButton );
```

```
String s_szelesseg = null; /* szélesség, mint String */
String s_magassag  = null; /* magasság, mint String  */

String filename = null;  /* az FLI/FLC fájl neve*/
String s_delay  = null;  /* késleltétés, mint String */
URL source_url;
Pixel_List pl;           /* segédváltozó a képek listájához */
DataInputStream in_stream = null;

try {
  s_szelesseg = getParameter("WIDTH");
  s_magassag  = getParameter("HEIGHT");

  filename = getParameter("FILENAME");
  s_delay = getParameter("DELAY");
  if (s_delay != null) {
    try {
      delay = Integer.parseInt(s_delay);
      szelesseg = Integer.parseInt(s_szelesseg);
      magassag  = Integer.parseInt(s_magassag);
    }
    catch ( NumberFormatException e ) {
      Err.Msg = s_delay + " is not a number";
      io = false;
    }
  }
}
catch ( NullPointerException e ) {
  Err.Msg = "no parameter \"FILENAME\"";
  io = false;
}

if ( filename == null ) {
  Err.Msg = "no parameter \"FILENAME\"";
  io = false;
}

if (io) {
  try {
    source_url = new URL(getCodeBase(), filename);
    in_stream = new DataInputStream(source_url.openStream());
  }
  catch ( MalformedURLException e ) {
    Err.Msg = "MalformedURLException";
    io = false;
  }
  catch ( IOException e ) {
    Err.Msg = "Cannot open: " + getCodeBase() + "/" + filename;
    io = false;
  }
}

if (io) {
  scan = new Scanner(in_stream);
  io = scan.scan_flic();

  if (io) {
    kepSzelesseg.setText( scan.width +"" );
    kepMagassag.setText ( scan.height+"" );
```

```
      dx = (int) Math.round( (xmax - scan.width)  / 2.0);
      dy = (int) Math.round( (ymax - scan.height) / 2.0 );

      resize(scan.width, scan.height);
      Images = new Image[scan.frame_counter];
      pl = scan.anchor;

      frame_nr = scan.frame_counter;
      increment = (int) Math.round( 1001.0 / scan.frame_counter );

      for (int i = 0; i < scan.frame_counter && io; i++) {
        if ( pl == null ) {
          Err.Msg = "error in tracing the list";
          io = false;
        }
        Images[i] = createImage( new myProducer(pl.color_values,
                                 scan.width, scan.height, pl.curr_Model));
        pl = pl.next;
      }
    }
  }
}

public void run() {
  long starttime = System.currentTimeMillis();
  if( neverend == true ){ neverendnew = false; }
  while (Thread.currentThread() == AnimatorThread) {
    if( isPlayed && !neverendnew ){
      repaint();
      painted = false;
      try { /* a "sleep()" "InterruptedException" kivételt dobhat */
        starttime += delay;
        Thread.sleep(Math.max(0,starttime - System.currentTimeMillis()));
      }
      catch (InterruptedException e) { break; }
      if( frame_cnt == 0 ){
        if( neverend == true ){ neverendnew = false; }
        if( neverend == false ){ neverendnew = true; }
      }
      if( painted ){
        if( seekBar.getValue() < 1000 ){
          frame_cnt = curr_nr % scan.frame_counter;
          seekBar.setValue( frame_cnt * increment );
          if( frame_cnt > frame_nr ){ frame_cnt = frame_nr; }
        }
        else{ seekBar.setValue( 0 ); frame_cnt = 0; }
        curr_nr++;
      }
    }
  }
}

public void start() {
  if ( !io ) return;
  if ( AnimatorThread == null ) {
    AnimatorThread = new Thread(this);
    AnimatorThread.start();
  }
}
```

```
  public void update( Graphics g ) {
    g.setColor( foregroundcolor );

    if( isPlayed ){
      g.drawImage( Images[curr_nr % scan.frame_counter], dx, dy, this);
    }

    g.fillRect( 641, 5, 3, 465 );
    g.drawLine(0, 481, szelesseg, 481 );
    g.drawLine(0, 484, szelesseg, 484 );

    g.setFont( fontType2 );
    g.drawString("Tulajdonságok: ", xmax+20, ymax-264);
    g.drawString("Szélesség: ", xmax+40, ymax-239);
    g.drawString("Magasság: ", xmax+40, ymax-215);
    g.drawString("Képkocka: ", xmax+20, ymax-20);
    kepKockak.setText(frame_cnt+" / "+frame_nr);
    g.drawString("Késleltetés növekménye: ", xmax+20, ymax-145);
    g.drawString("msec", xmax+301, ymax-145);
    kesleltetesNovekmenye.setText(delayIncrement+"");
    g.drawString("Késleltetés: ", xmax+20, ymax-120);
    g.drawString("msec", xmax+206, ymax-120);
    kesleltetes.setText(delay+"");
    painted = true;
  }

  public void stop() {
    AnimatorThread = null; /* Különben az animáció lejátszása folytatódna a
                              háttérben. */
  }

  public void paint( Graphics g ) {
    g.setColor( foregroundcolor );
    if ( Err.Msg != null) {
      g.setFont( new Font(g.getFont().getName(), Font.BOLD, 10) );
      g.drawString( Err.Msg, 10, 100 );
      return;
    }
    if( isPlayed ){
      g.drawImage( Images[curr_nr % scan.frame_counter], dx, dy, this );
    }
    g.fillRect( 641, 5, 3, 465 );
    g.drawLine(0, 481, szelesseg, 481 );
    g.drawLine(0, 484, szelesseg, 484 );
    g.setFont( fontType2 );
    g.drawString("Tulajdonságok: ", xmax+20, ymax-264);
    g.drawString("Szélesség: ", xmax+40, ymax-239);
    g.drawString("Magasság: ", xmax+40, ymax-215);
    g.drawString("Képkocka: ", xmax+20, ymax-20);
    kepKockak.setText(frame_cnt+" / "+frame_nr);
    g.drawString("Késleltetés növekménye: ", xmax+20, ymax-145);
    g.drawString("msec", xmax+301, ymax-145);
    kesleltetesNovekmenye.setText(delayIncrement+"");
    g.drawString("Késleltetés: ", xmax+20, ymax-120);
    g.drawString("msec", xmax+206, ymax-120);
    kesleltetes.setText(delay+"");
  }
}
```

```
class Scanner {
  /* Ezek az adattagok nagyrészt tömb típusokat jelölnek. */
  final private int FLI_MAGIC = 0xAF11;
  final private int HEADER_LENGTH = 26 + 102;
  final private int FRAME_HEADER_LENGTH = 16;
  final private short M_FLI = (short) 0xAF11;
  final private short M_FLC = (short) 0xAF12;
  final private short FLI_COLOR_256 = (short) 4;
  final private short FLI_COLOR = (short) 11;
  final private short FLI_LC    = (short) 12;
  final private short FLI_WORD_LC = (short) 7;
  final private short FLI_BLACK = (short) 13;
  final private short FLI_BRUN  = (short) 15;
  final private short FLI_COPY  = (short) 16;
  final private short FRAME_ID  = (short) 0xf1fa;

  public Pixel_List anchor = null;
  public int width, height, frame_counter = 0;

  private boolean io = true;
  private byte buffer[];
  private DataInputStream In_str;
  private Pixel_List last = null;   /* a képadatok listájának vége */
  private IndexColorModel myModel;
  private byte r[] = new byte[256];
  private byte g[] = new byte[256];
  private byte b[] = new byte[256];

  Scanner(DataInputStream f_In_str) {
    In_str = f_In_str;
  }

  public boolean scan_flic() {
    int length;  /* a fájl mérete */
    short magic, size, speed;

    buffer = new byte[HEADER_LENGTH];
    try {
      ReadContent(In_str, buffer, HEADER_LENGTH);
    }
    catch (IOException e) {
      Err.Msg = "read error 1";
      return false;
    }

    length =         to_int(buffer, 4,  0);
    magic  = (short) to_int(buffer, 2,  4);
    size   = (short) to_int(buffer, 2,  6);
    width  = (short) to_int(buffer, 2,  8);
    height = (short) to_int(buffer, 2, 10);
    speed  = (short) to_int(buffer, 2, 16);

    switch (magic) {
      case M_FLI: break;
      case M_FLC: break;
      default: Err.Msg = "Number " + magic + "unknown";
      return false;
    }
```

```
  length -= HEADER_LENGTH; /* hátralevő hossz */

  while ( length > 0 && io ) {
    length -= scan_frame();
  }
  frame_counter--;
  return io;
}

private int scan_frame() {
  int frame_size;
  short fh_id, chunks;

  buffer = new byte[FRAME_HEADER_LENGTH];

  try {
    ReadContent( In_str, buffer, FRAME_HEADER_LENGTH );
  }
  catch ( IOException e ) {
    Err.Msg = "read error 2";
    io = false;
    return 0;
  }

  frame_size =         to_int(buffer, 4, 0);
  fh_id      = (short) to_int(buffer, 2, 4);
  chunks     = (short) to_int(buffer, 2, 6);

  myModel = null;

  if ( fh_id != FRAME_ID ) {
    Err.Msg = "FLI/FLC synchronization error";
    io = false;
    return 0;
  }
  for ( int ch = 0; ch < chunks; ch++) {
    scan_chunk();
  }
  return frame_size;
}

private void scan_chunk() {
  int ch_size;
  short ch_type;

  buffer = new byte[6];
  try {
    ReadContent( In_str, buffer, 6 );
  }
  catch ( IOException e ) {
    Err.Msg = "read error 3";
    io = false;
    return;
  }

  ch_size =         to_int(buffer, 4, 0);
  ch_type = (short) to_int(buffer, 2, 4);

  buffer = new byte[ch_size - 6];
```

```
  try {
    ReadContent( In_str, buffer, ch_size - 6 );
  }
  catch ( IOException e ) {
    Err.Msg = "read error 4";
    io = false;
    return;
  }
  switch (ch_type) {
    case FLI_COLOR:     scan_FLI_COLOR(2);  break;
    case FLI_COLOR_256: scan_FLI_COLOR(0);  break;
    case FLI_LC:        scan_FLI_LC();      break;
    case FLI_WORD_LC:   scan_FLI_WORD_LC(); break;
    case FLI_BRUN:      scan_FLI_BRUN();    break;
    case FLI_BLACK: break;
    case FLI_COPY:  break;
    default: Err.Msg = "chunk type " + ch_type + " unknown";
    io = false;
    return;
  }
}

private void scan_FLI_BRUN() {
  int idx = 0;  /* eltolás az adatpufferben     */
  int ppx;
  int pkts;     /* adatcsomag számláló a sorban */
  int xchpx;    /* aktuális képpont érték       */
  int pixel;    /* képpont számláló             */
  int line;     /* aktuális sor                 */
  byte f;

  short line_count = (short) height;

  if ( frame_counter > 0 ) {
    Err.Msg = "FLI_BRUN although frame_counter > 0";
    io = false;
    return;
  }

  if ( myModel == null) {
    Err.Msg = "No color model for the first frame ?";
    io = false;
    return;
  }

  last = anchor = new Pixel_List(null, myModel, width, height);
  frame_counter++;
  idx = 0;

  for ( line = 0; line < line_count; line++ ) {
    pkts = ( 0xff & buffer[idx++] );
    ppx  = line * width;
    for ( int pkt_nr = 0; pkt_nr < pkts; pkt_nr++ ) {
      xchpx = (int) buffer[idx++];
      if ( xchpx >= 0 ) {
        f = (byte) (0xff & buffer[idx++]);
        for (int k = 0; k < xchpx; k++) {
          anchor.color_values[ppx++] = f;
        }
```

```
      }
      else {
        xchpx = -xchpx;
        for (pixel = 0; pixel < xchpx; pixel++) {
          anchor.color_values[ppx++] = (byte) (0xff & buffer[idx++]);
        }
      }
    }
  }
}

private void scan_FLI_LC() {
  int idx = 0;  /* eltolás a pufferben */
  int ppx;
  int pkts;     /* a sorban levő adatcsomagok számlálója */
  int xchpx;
  int pixel;    /* képpont számláló    */
  int line;     /* aktuális sor        */
  short line_count;  /* a megváltoztatott sorok száma */
  byte f;

  idx = 0;
  if ( frame_counter == 0 ) {
    Err.Msg = "FLI_LC although frame_counter == 0";
    io = false;
    return;
  }

  last.next = new Pixel_List(last, myModel, width, height);
  frame_counter++;

  last = last.next;

  line       =         to_int(buffer, 2, idx);  idx += 2;
  line_count = (short) to_int(buffer, 2, idx);  idx += 2;

  for ( ; line < line_count; line++) {
    pkts = (0xff & buffer[idx++]);
    ppx = line * width;
    for (int pkt_nr = 0; pkt_nr < pkts; pkt_nr++) {
      ppx += (0xff & buffer[idx++]);
      xchpx = (int) buffer[idx++];
      if (xchpx < 0) {
        xchpx = -xchpx;
        f = (byte) (0xff & buffer[idx++]);
        for (int k = 0; k < xchpx; k++) {
          last.color_values[ppx++] = f;
        }
      }
      else {
        for (pixel = 0; pixel < xchpx; pixel++) {
          last.color_values[ppx++] = (byte) (0xff & buffer[idx++]);
        }
      }
    }
  }
}
```

```
private void scan_FLI_WORD_LC() {
  int idx = 0;      /* eltolás a pufferben */
  int ppx;
  int xchpx;
  int pixel;        /* képpont számláló    */
  int line;         /* sorszámláló         */
  int yoff;         /* a valódi aktuális sor */
  short line_count;
  short pkts;       /* az adatcsomagok száma a sorban */
  byte f1, f2;
  boolean last_pix_flag = false;
  byte last_pixel = 0;

  idx = 0;
  if ( frame_counter == 0 ) {
    Err.Msg = "FLI_LC_WORD although frame_counter == 0";
    io = false;
    return;
  }

  last.next = new Pixel_List(last, myModel, width, height);
  frame_counter++;

  last = last.next;
  line_count = (short) to_int(buffer, 2, idx);  idx += 2;
  yoff = 0;
  for ( line = 0; line < line_count; line++ ) {
    pkts     = (short) to_int(buffer, 2, idx);  idx += 2;
    while ((pkts & 0x8000) != 0) {
      if ((pkts & 0x4000) != 0) {
        yoff -= (int) pkts;
      }
      else {
        last_pix_flag = true;
        last_pixel = (byte) (pkts & 0xff);
      }
      pkts   = (short) to_int(buffer, 2, idx);  idx += 2;
    }

    ppx = yoff * width;
    for ( int pkt_nr = 0; pkt_nr < pkts; pkt_nr++ ) {
      ppx += (0xff & buffer[idx++]);
      xchpx = (int) buffer[idx++];
      if ( xchpx < 0 ) {
        xchpx = -xchpx;
        f1 = (byte) (0xff & buffer[idx++]);
        f2 = (byte) (0xff & buffer[idx++]);
        for ( int k = 0; k < xchpx; k++ ) {
          last.color_values[ppx++] = f1;
          last.color_values[ppx++] = f2;
        }
      }
      else {
        for ( pixel = 0; pixel < xchpx; pixel++ ) {
          last.color_values[ppx++] = (byte) (0xff & buffer[idx++]);
          last.color_values[ppx++] = (byte) (0xff & buffer[idx++]);
        }
      }
    }
```

```
      if ( last_pix_flag ) {
        last.color_values[(yoff + 1) * width - 1] = last_pixel;
        last_pix_flag = false;
      }
      yoff++;
    }
  }

  private void scan_FLI_COLOR(int shift) {
    short pkts;        /* adatcsomagok száma   */
    int skip;
    int count;
    int idx = 0;
    int table_idx = 0; /* index a színtérképen */

    pkts = (short) to_int(buffer, 2, 0);  idx += 2;
    for ( int pkt_nr = 0; pkt_nr < pkts; pkt_nr++) {
      skip  = 0xff & (buffer[idx++]);
      count = 0xff & (buffer[idx++]);
      table_idx += skip;
      if ( count == 0 ) count = 256;
      for ( int j = 0; j < count; j++) {
        r[table_idx  ] = (byte) ((0xff & buffer[idx++]) << shift);
        g[table_idx  ] = (byte) ((0xff & buffer[idx++]) << shift);
        b[table_idx++] = (byte) ((0xff & buffer[idx++]) << shift);
      }
    }
    myModel = new IndexColorModel(8, table_idx, r, g, b);
  }

  private void ReadContent(DataInputStream Stream, byte field[], int size)
                           throws IOException {
    int bytes_read = 0;
    while ( bytes_read < size) {
      bytes_read += Stream.read( field, bytes_read, size - bytes_read );
    }
  }

  private int to_int(byte b[], int length, int off) {
    int r_val = 0;
    for ( int i = 0; i < length; i++ ) {
      r_val <<= 8;
      r_val |= b[length - 1 - i + off] & 0xff;
    }
    return r_val;
  }
}

class myProducer implements ImageProducer {
  private byte Pix_Data[];
  private int width, height;
  ColorModel Model;

  myProducer(byte pixels[], int w, int h, ColorModel cm) {
    Pix_Data = pixels;
    width = w; height = h; Model = cm;
  }
```

```
  public void addConsumer(ImageConsumer ic) {
    ic.setDimensions(width, height);
    ic.setColorModel(Model);
  }

  public boolean isConsumer(ImageConsumer ic) {
    return true;
  }

  public void removeConsumer(ImageConsumer ic) { }

  public void requestTopDownLeftRightResend(ImageConsumer ic) {
    ic.setHints(ImageConsumer.TOPDOWNLEFTRIGHT);
    ic.setPixels(0, 0, width, height, Model, Pix_Data, 0, width);
    ic.imageComplete(ImageConsumer.STATICIMAGEDONE);
  }

  public void startProduction(ImageConsumer ic) {
    ic.setDimensions(width, height);
    ic.setColorModel(Model);
    requestTopDownLeftRightResend(ic);
  }

}
```