Pildioperatsioonid

Raster, RGB, baidid, filter, joonistuskiirus

Pildifaili loomine

Soovides joonistatud pildi andmeid talletada või mujale üle kanda, tuleb need paigutada edasiseks lugemiseks arusaadavale kujule. Loodud Image-tüüpi pildist võib andmed PixelGrabberi abil küsida täisarvumassiivi, kus iga element vastab ühele punktile pildil ning neid arve edaspidi talletada või töödelda ning hiljem MemoryImageSource abil uuesti pildiks muundada. Tahtes aga loodud kujutist mõne teise programmi abil edaspidi kasutada, tuleb see salvestada üldtunnustatud formaati. Õnnetuseks aga veel JDK1.3 puhul standardvahenditesse sisse ehitatud kujul üldlevinud failiformaatidesse salvestamist polnud, seega tuli kasutada muid teid. Lisavahendid Java Media Framework ning Java Advanced Imaging võimaldavad mitmeid lisaoperatsioone piltidega, ka salvestamist. Salvestamise tarvis on loonud koodilõike mitmed firmad ja programmeerijad, küllalt levinud on ACME GIF-ide salvestamise vahend. Sun-i Javaga kaasa tulev pakett com.sun.image.codec.jpeg võimaldab pildi paari käsuga salvestada JPEG formaati.

```
JPEGCodec.createJPEGEncoder(
      new FileOutputStream("pilt1.jpeg")
    ).encode(pilt);
```

loob väljundvoo faili nimega pilt1.jpeg ning saadab sinna muutujas pilt oleva pildi. Nõnda tekib kettale eespool loodud pilt. Kuna voogude sihtpunkte saame vabalt valida, siis võib samade vahendite abil pildi ka teise masinasse vaatamiseks saata kasutades küllalt hästi optimeeritud formaati.

```
import com.sun.image.codec.jpeg.*;
                      //kuulub SUNi JDK-sse
import java.awt.image.*;
import java.awt.*;
import java.io.*;

public class JPEGKodeerija1{
  public static void main(String argumendid[])
                              throws IOException{
    BufferedImage pilt = new BufferedImage(50, 50,
                         BufferedImage.TYPE_INT_RGB);
    Graphics piltg=pilt.createGraphics();
    piltg.setColor(Color.green);
    piltg.drawOval(5, 5, 40, 40);
    JPEGCodec.createJPEGEncoder(
      new FileOutputStream("pilt1.jpeg")
    ).encode(pilt);
  }
}
```

Alates Java versioonist 1.4 saab kasutada standardpaketti javax.imageio, kuju on koondatud korralik kogumik klasse ja käsklusi piltide lugemiseks ja kirjutamiseks. Järgnevalt näide ImageIO klassi abil pildifaili loomisest.

```
import javax.imageio.*;
import java.awt.image.*;
import java.awt.*;
import java.io.*;
public class JPEGKodeerija2{
  public static void main(String argumendid[])
                              throws IOException{
```

```
    BufferedImage pilt = new BufferedImage(50, 50,
                         BufferedImage.TYPE_INT_RGB);
    Graphics piltg=pilt.createGraphics();
    piltg.setColor(Color.green);
    piltg.drawOval(5, 5, 40, 40);
    ImageIO.write(pilt, "jpg", new File("pilt2.jpeg"));
  }
}
```

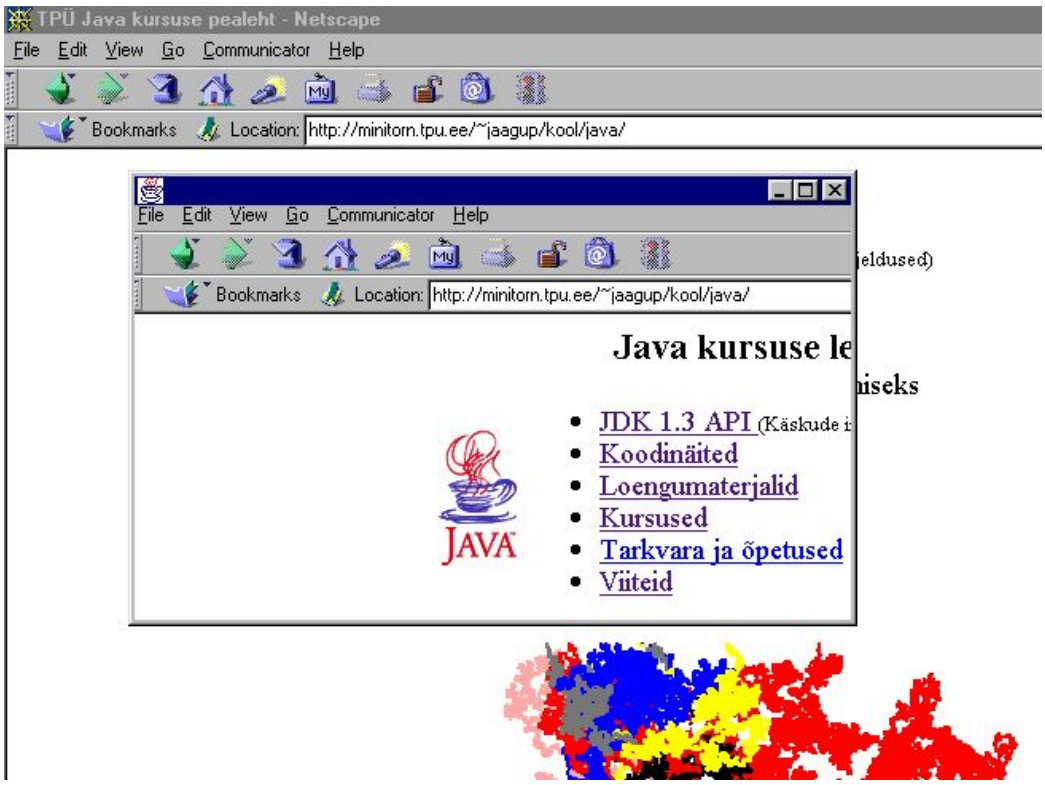
Ekraanipildi kopeerimine

Ekraanipilti kopeerida ning klaviatuuri ja hiire teateid luua aitab klassi java.awt.Robot, seda vaid juhul kui programmi vastav tegevus on lubatud. Näiteks rakendid ei tohi turvanõudeid arvestades vastavate ettevõtmistega tegelda. Ekraanipildi saab tavaliseks pildiks

```
pilt=r.createScreenCapture(
            new Rectangle(0, 0, 400, 300));
```

abil. Mis programm sellega edaspidi peale hakkab on juba programmeerija otsustada. Siin näidatakse salvestis lihtsalt ekraanile, kuid selle saab ka faili kirjutada või hoopis pildi pealt kujundeid otsides kasutaja tegevust analüüsima hakata.

```
import java.awt.*;
public class Robot2 extends Frame{
  Image pilt;
  public Robot2(){
   try{
    Robot r=new Robot();
    pilt=r.createScreenCapture(
            new Rectangle(0, 0, 400, 300));
    setSize(300, 200);
    setLocation(200, 100);
    setVisible(true);
   }catch(Exception e){}
  }
  public void paint(Graphics g){
    g.drawImage(pilt, 0, 0, this);
  }
  public static void main(String argumendid[]){
    new Robot2();
  }
}
```



Pildi muutmine

Paketi java.awt.image mitmed operatsioonid lubavad olemasoleva pildi väljanägemist muuta. Järgnevates näidetes luuakse BufferedImage ning sellele joonistatakse ringe. Edasi koostatakse esimesest pildist operatsiooni teel teine pilt ning näidatakse need kõrvuti ekraanile

Värvide tugevus

RescaleOp võimaldab muuta pildi värve nii kõiki üheskoos kui punast, rohelist ning sinist eraldi. Siin näites luuakse

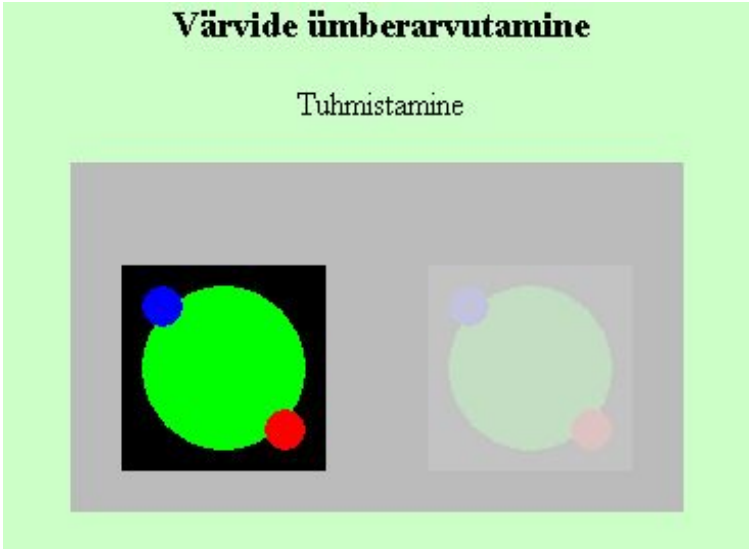
```
RescaleOp rop = new RescaleOp(0.1f, 200.0f, null);
```

, mis kõikjal jätab algsetest värvidest alles vaid 0,1 ehk 10% ning igale poole kuhu võimalik lisab iga värvi väärtusele 200. Niisuguse toimimise puhul saadakse tulemuseks valkjastuhm pilt, kuna esialgsetest väärtustest on alles vaid tühine osa ning pildi kõikide punktide kõik värvid on peaaegu maksimumväärtuse (255) lähedal. Kolmas parameeter on jäetud tühjaks (null), selle kaudu saaks soovi korral värvimuutjale edasi anda viimistlusvihjeid (RenderingHints) muutmise ressursinõuldlikkuse ja kvaliteedi kohta.

```
import java.awt.image.*;
import java.awt.geom.*;
import java.awt.*;
import java.applet.*;
public class Pildiskaleerimine1 extends Applet{
  BufferedImage pilt1=new BufferedImage(100, 100,
                      BufferedImage.TYPE_INT_RGB);
  BufferedImage pilt2=new BufferedImage(100, 100,
                      BufferedImage.TYPE_INT_RGB);
  public Pildiskaleerimine1(){
   Graphics g=pilt1.createGraphics();
   g.setColor(Color.green);
   g.fillOval(10, 10, 80, 80);
   g.setColor(Color.blue);
   g.fillOval(10, 10, 20, 20);
   g.setColor(Color.red);
   g.fillOval(70, 70, 20, 20);
   RescaleOp rop = new RescaleOp(0.1f, 200.0f, null);
```

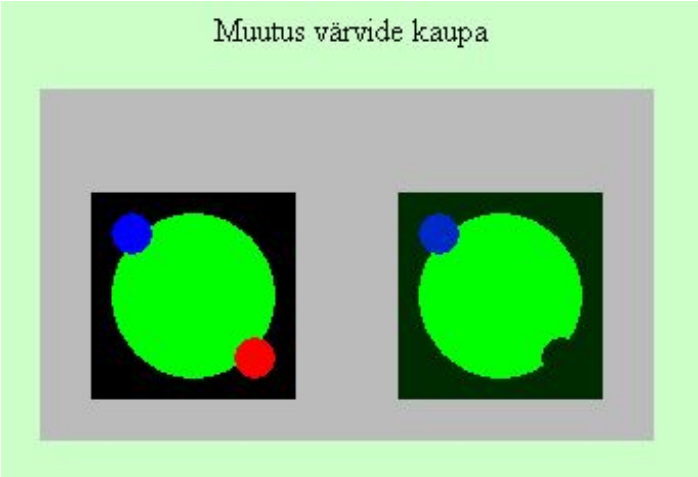
```
                                    //väärtus*0.1 + 200
   rop.filter(pilt1,pilt2);
  }
  public void paint(Graphics g){
    g.drawImage(pilt1, 25, 50, this);
    g.drawImage(pilt2, 175, 50, this);
  }

  public static void main(String argumendid[]){
    Frame f=new Frame("Pildi värvide muutmine");
    f.add(new Pildiskaleerimine1());
    f.setSize(300, 200);
    f.setVisible(true);
  }
}
```



Väärtusi võib anda ka igale värvile eraldi. Kui ennist oli RescaleOp'i konstruktori parameetriteks kaks arvu ning null, siis siin on värvide muutmiseks ette antud kaks kolmeelemendilist massiivi ning viimistlusvihjete kohale endiselt null. Esimese massiivi iga element tähistab vasstavale värvile omistatud kordajat, teise massiiv liikmed näitavad, palju igale värvile tugevust juurde liita. Kui mõne värvi kordajaks on 0, siis seda uuele pildile ei jõuagi. Kordaja 1 puhul jääb endine väärtus alles ning vahepealse väärtuse puhul jääb endisest alles osa.

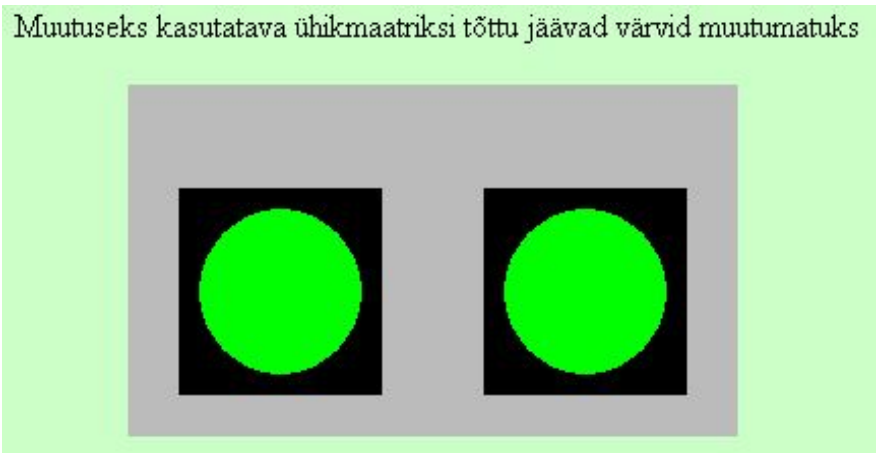
```
RescaleOp rop = new RescaleOp(
     new float[]{0, 1, 0.8f},
     new float[]{0, 50, 0}, null
);
//punane kaob, roheline jääb alles,
//sinisest 80%. Rohelisele lisatakse 50 ühikut.
rop.filter(pilt1,pilt2);
```



Värviarvutus maatriksiga

Kui punkti uus värv ei sõltu mitte ainult sama värvi tugevusest eelmisel pildil, vaid tahetakse uue punkti rohelise tugevuse arvutamisel arvestada ka eelmise punkti sinise väärtust, siis aitab BandCombineOp ning sõltuvused tuleb ette anda kahemõõtmelise massivi ehk maatriksina. Iga rea peal on kirjas, palju punkti vastava uue värvi arvutamisel tuleb arvestada algse punkti punast, rohelist ning sinist värvi.

```
float andmed[][]=new float[][]{
     {1,  0,   0},
     {0,  1,   0},
     {0,  0,   1}
};
// uus roheline = 0*vana punane+
// 1* vana roheline + 0*vana sinine
BandCombineOp bco=new BandCombineOp(
      andmed, null
);
bco.filter(pilt1.getData(), pilt2.getRaster());
```

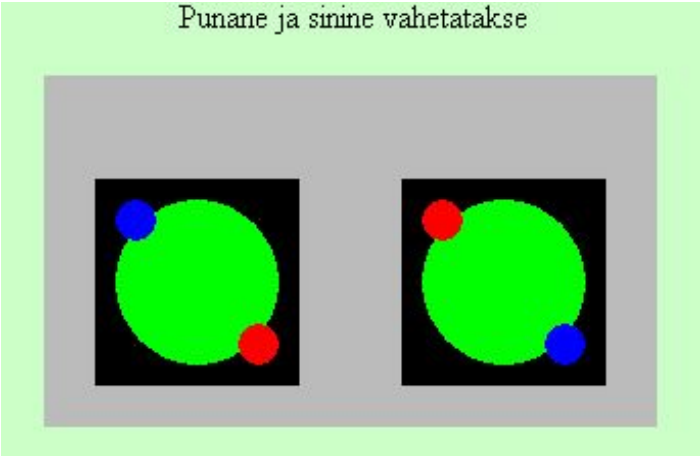


BandCombineOp'ile ei piisa piltide eneste etteandmisest. Filtrisse andmete lugemiseks tuleb sellele ette anda pildipunktide raster, mille väljastab BufferedImage meetod getData. Sihtpildist on tarvis ette anda WritableRaster ehk isend, kus pilt oma andmeid hoiab ning mille elementide muutmisel ka pildi punktid omale uue värvi saavad.

Põhivärvide vahetamine

Uue punkti punase osa arvutamisel võib endise punase sootuks arvestamata jätta ning võtta väärtus näiteks algse punkti sinise oma. Nii õnnestub värve ühendada või vahetada.

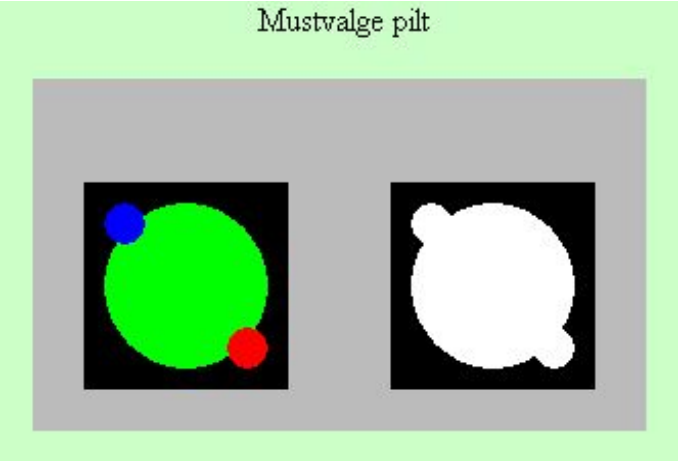
```
float andmed[][]=new float[][]{
     {0,  0,   1},   //punane ja sinine vahetatakse
     {0,  1,   0},
     {1,  0,   0}
};
```



Pilt mustvalgeks

Kui igale poole, kus ennist mingigi värv leidus panna juurde ka kõik teised värvid, siis on tulemuseks mustvalge pilt, sest värvide puudumine on must ning kõikide värvide kompott valge. Kui mõnes kohas oli ennist värve vaid osaliselt, siis nüüd oleks tulemuseks halltoon.

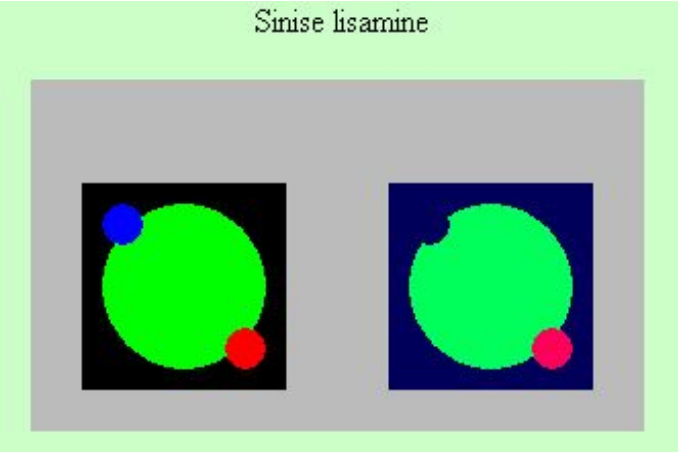
```
float andmed[][]=new float[][]{
     {1, 1, 1},
     {1, 1, 1},
     {1, 1, 1}
};
```



Põhivärvi lisamine

Tahtes üle kogu pildi põhivärvile väärtust lisada, võib massiivile juurde luua neljanda veeru, kus öeldakse, palju seda värvi juurde pannakse.

```
float andmed[][]=new float[][]{
     {1,  0,   0,  0},
     {0,  1,   0,  0},
     {0,  0,   0,  100}
};
//kõikjale lisatakse 100 ühiku ulatuses sinist (max 255)
```

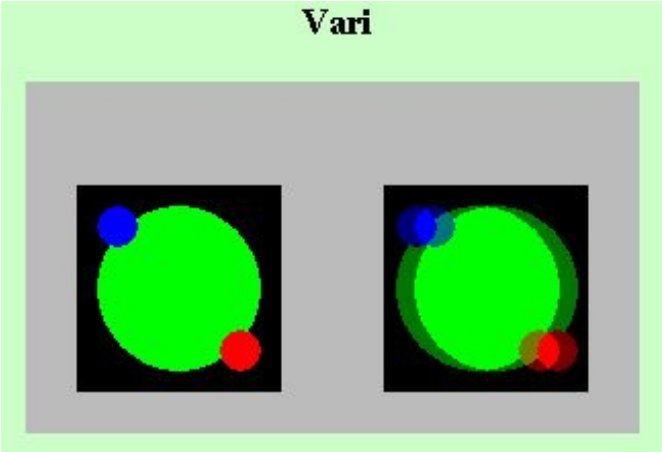


Varju loomine

ConvolveOp võtab uue pildi loomisel arvesse vana pildi vastava punkti naaberpunktide väärtused. Siin antakse operatsioonile ette kümneelemendiline massiiv, mille esimene ning viimane element on väärtusega 0,5 , keskmised aga tühjad. Nii segatakse uue pildi loomisel kokku algsest punktist viis kohta vasakul ning viis kohta paremal asunud punkti värvid ningu tulemusena saadakse eelmine pilt nõnda, nagu oleks seda pildistamise ajal väristatud. Massiivi elementidega mängides võib ka varju tugevamaks või nõrgemaks muuta, arvestada ka vahepealsete punktide väärtusi või sootuks piirduda vaid ühel pool kaugel asuva punktiga. Siis tunduks, nagu oleks uus pilt tervikuna vanaga võrreldes etteantud suunas liikunud.

```
float andmed[]=new float[]
       {0.5f, 0, 0, 0, 0, 0, 0, 0, 0, 0.5f};
ConvolveOp co=new ConvolveOp(new Kernel(10, 1, andmed));
```

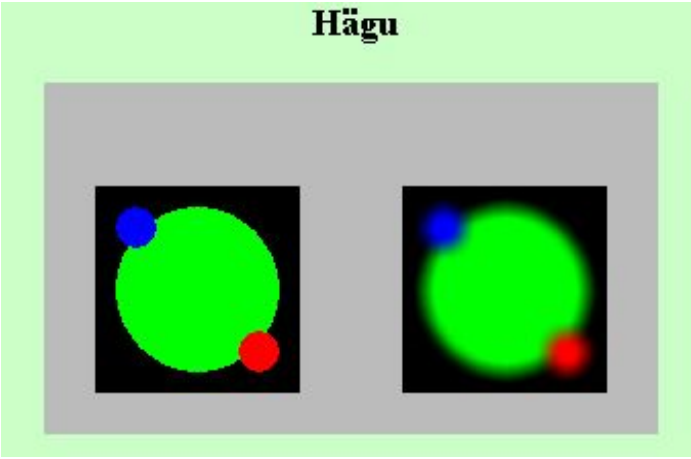
```
                                //10 veergu(x), 1 rida(y)
co.filter(pilt1, pilt2);
```



Piirjoonte hägustamine

Kui ConvolveOp'ile anda ette kahemõõtmeline massiiv, siis arvestatakse uue punkti arvutamisel endisest nii vasakul, paremal, üleval kui all asuvaid punkte. Kui võtta uue punkti aluseks ümbritsevate 10*10 punktide keskmised väärtused nagu siin, siis on tulemuseks udustatud piirjoontega pilt, sest kui terava piiri juures arvutatakse uuel pildil punkti leidmiseks mitme ümbritseva punkti keskmine, siis ei saa ju kõrvutiasetsevate punktide värvid enam nii tugevalt eristuda ning tulemuseks ongi pehme üleminek.

```
float andmed[]=new float[100];
for(int i=0; i<100;i++)andmed[i]=0.01f;
ConvolveOp co=new ConvolveOp(
                              new Kernel(10, 10, andmed));
```

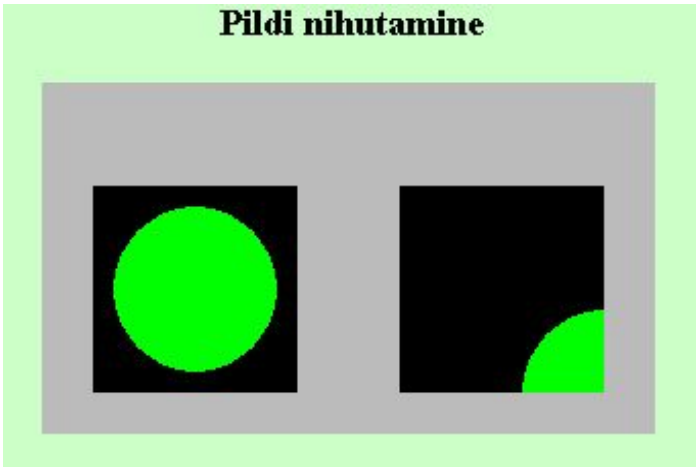


Nihutamine

Lihtne nihutusoperatsioon sarnaneb Graphics2D poolt pakutavaga. AffineTransformOp'ile saab anda ette AffineTransformi kõikide oma võimaluste ja nende kombinatsioonidega. Võib algset pilti nihutada, suurendada, keerata ning kiiva/kaldu ajada.

```
AffineTransformOp ato=new AffineTransformOp(
```

```
        AffineTransform.getTranslateInstance(50, 50),
        AffineTransformOp.TYPE_BILINEAR
);
ato.filter(pilt1, pilt2);
```



Pildi koostamine

Kui harilikud joonistusvahendid tunduvad aeglaseks või kohmakaks jääma, siis on võimalik pilt ise üksikute punktide kaupa täisarvumassiivis välja arvutada ning siis ühe käsuga joonistatavaks pildiks muuta. All näites koostatakse värviüleminekuga pilt, kus ühtlasel õrnal rohelisel foonil suureneb kõrguse kasvades punase osa ning laiuse kasvades sinise osa. Andmete paigutamiseks tuleb luua nii pikk täisarvumassiiv, kui palju on pildi peal punkte kokku. Siin näites on kõrguseks ja laiuseks 200 punkti, nii et massiivi pikkuseks tuleb 200*200 ehk 40000. Arvestades, et iga täisarv võtab neli baiti, on tegemist 160 kilobaidiga ehk täiesti arvestatava mälumahuga. Pildipunktidele vastavad massiivi elemendid nõnda, et kõigepealt on esimese rea punktid, edasi teise rea omad jne. Iga täisarv kannab eneses pildipunkti nelja omadust Alpha-Red-Green-Blue värvimudeli järgi. Esimene määrab paistvuse, ehk kui palju koostatud värvist üldse näha on. Ülejäänud kolm osa tähistavad vastavalt iga värvi tugevust ning nagu arvutimaailmas tavaks, segatakse muud värvid nende kolme pealt kokku, sest ka inimsilm pole võimeline palju enam värve nägema kui nendest kolmest kokku segada annab. Igale komponendile eraldatakse neljabaidilisest int'ist üks bait. See määrab, et iga suuruse vähim väärtus on 0 ning suurim 255. Täisarvu baitide ühekaupa kasutamine võib olla harjumatu, kuid see võimaldab kiirust kaotamata anda edasi lihtsalt ülekantava lihttüübi väärtusega kogu ühe punkti värvi määramiseks vajaliku teabe. Kasutataks samaks otstarbeks objektide omadusi, kuluks hulk masina jõudu sealt andmete ühest kohast teise liigutamiseks ning kätte saamiseks. Lähemal vaatamisel ei peakski baitide kaupa andmete määramine väga hirmus olema, liiatigi, kui selle tarvis on soovi korral võimalik kirjutada paarirealine alamprogramm. Siin aga püüame koodi lühiduse huvides ilma selleta hakkama saada. Kui soovin kogu loodud värvi nähtavaks saada, tuleks vasakusse baiti kirjutada 255. Otse käsklust millega täisarvu baidi peale kirjutada saaks loodud ei ole, kuna see on lühidalt lahendatav. Soovides luua täisarvu, mille vasak bait on 255 ning ülejäänud nullid, võin kõigepealt luua täisarvu, mille väärtus on 255 (st. parempoolse baidi väärtus 255, ehk kõik bitid ühed ning kõik ülejäänud baidid nullid).

```
int a=255;
```

Kui edasi soovin väärtuse edasi kanda parempoolsest (esimesest) baidist vasakpoolsesse (neljandasse), siis tuleks mul algset väärtust kolme baidi ehk 24 biti jagu vasakule nihutada.

```
int b=a<<24;
```

Nii ongi loodud int, mille vasakpoolne bait on väärtusega 255 ning ülejäänud nullid. Kui sooviksin, et rohelise tugevust tähistava baidi (paremalt teise) väärtus oleks 100, siis tuleks mul numbrit 100 ühe baidi

ehk kaheksa biti jagu vasakule nihutada.

```
int c=100<<24;
```

Kui soovin, et loodavas arvus oleks ühendatud kahe arvu mittenullilise väärtusega baidid (bitid), siis võin nende väärtused ühendada operaatori | abil.

```
int d=b|c;
```

Korraga võin ühendada (liita) ka rohkemate numbrite bittide väärtusi. Kui ennist on välja arvutatud punasele, rohelisele ning sinisele vastav number, massiiv punktid tähistab loodava pildi punktide jaoks leitavaid täisarve ning nr arvutatava punkti järjekorranumbrit, siis

```
punktid[nr++] = (255<<24)|(punane << 16) | roheline << 8 | sinine;
```

annab kokku ilusti täisarvu, mille esimene bait ütleb, et värvi tuleb näidata täies ulatuses, ülejäänud aga asetavad oma kohtadele igale värvile vastava tugevuse. Meetodi lõpus väljastatakse `createImage(new MemoryImageSource(laius, korgus, punktid, 0, laius));` , mis loob etteantud punktimassiivist pildi. Esimesed kolm parameetrit peaksid nimede järgi olema arusaadavad, 0 näitab, et andmeid hakatakse lugema massiivi algusest ning viimane näitab, mitme punkti kaupa rea tarvis massiivist andmeid võetakse. Harilikult on see võrdne pildi laiusega punktides. Joonistusmeetodis paint pole muud, kui pildiloomismeetodi käest pilt küsida ning see ekraanile joonistada.

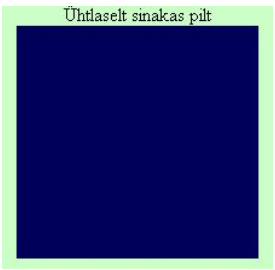
```
import java.awt.image.*;
import java.awt.*;
import java.applet.Applet;
public class Pildiloomine1 extends Applet{
   public Image looPilt(){
        int laius=200;
        int korgus=200;
        int punktid[] = new int[laius*korgus];
        int nr=0;
        for (int y = 0; y < korgus; y++){
            int punane = y;
            for (int x = 0; x < laius; x++) {
                int sinine =  x;
                int roheline = 50;
                punktid[nr++] = (255<<24)|(punane << 16) | roheline << 8 | sinine;
            }
        }
        return createImage(new MemoryImageSource(laius, korgus, punktid, 0, laius));
   }

   public void paint(Graphics g){
        Image pilt=looPilt();
        g.drawImage(pilt, 0, 0, null);
   }

  public static void main(String argumendid[]){
    Frame f=new Frame();
    f.add(new Pildiloomine1());
    f.setSize(200, 220);
    f.setVisible(true);
  }
}
```

Kui kõikidele punktidele anda ühesugune sinist värvi tähistav väärtus, siis on tulemuseks ka ühtlaselt sinine pilt. Abivahend neile, kes eelmist näidet lihtsustada tahavad.

```
public Image looPilt(){
       int[] punktid=new int[200*200];
       for(int i=0; i<punktid.length; i++){
         punktid[i]=(255<<24)+100;
       }
       return createImage(new MemoryImageSource(
                       200, 200, punktid, 0, 200));
}
```



Pildi arvutamisel saab arvestada kõiki parameetreid, mis programmeerijale ette jäävad ning kasutada on. Siin tähistavad x ning y kohta, kus kasutaja viimati hiirega ekraanile vajutanud on. Punase tugevus arvutatkse vastavalt leitava punkti kaugusest hiirevajutuskohast.

```
public Image looPilt(){
        int laius=200;
        int korgus=200;
        int punktid[] = new int[laius*korgus];
        int nr=0;
        for (int y = 0; y < korgus; y++){
            for (int x = 0; x < laius; x++) {
                int punane=(int)(255*Math.sqrt(
                    (x-hx)*(x-hx)+(y-hy)*(y-hy))/laius);
                int sinine =  100;
                punktid[nr++] =
                  (255<<24)|(punane << 16) | sinine;
            }
        }
        return createImage(new MemoryImageSource(
                     laius, korgus, punktid, 0, laius));
   }
```

Lainetus

Kui punase tugevus ei sõltu mitte lihtsalt arvutatava punkti ning hiirevajutuse kaugusest vaid kauguse siinusest, mis nähtavuse suurendamise eesmärgil mingi kordajaga läbi korrutatud, siis saab tulemuseks lainetuse, sest siinus on perioodiline funktsioon ning kauguse suurenedes lainepikkuse jagu jõutakse arvutustega jälle sama kaugele tagasi. Update on üle kirjutatud seetõttu, et vajutusejärgsel pildi taasloomisel ei käiks taustavärvi vilksatus üle vaid oleks kohe uus pilt paigas.

```
import java.awt.image.*;
import java.awt.*;
import java.awt.event.*;
import java.applet.Applet;
public class Pildiloomine3 extends Applet implements MouseListener{
   int hx=100, hy=100;
   public Pildiloomine3(){
     addMouseListener(this);
   }
   public Image looLainetus(int laius, int korgus, int kx, int ky,
                                double lainepikkus){
        int punktid[] = new int[laius*korgus];
        int nr=0;
        for (int y = 0; y < korgus; y++){
            for (int x = 0; x < laius; x++) {
                int punane=125+(int)(125*Math.sin(kaugus(x, y, kx, ky)*2*Math.PI/lainepikkus));
//                int punane=Math.max(255-(int)(255*Math.sqrt((x-hx)*(x-hx)+(y-hy)*(y-hy))/(laius)), 0);
                int sinine =  200;
                punktid[nr++] = (255<<24)|(punane << 16) | sinine;
            }
        }
        return createImage(new MemoryImageSource(laius, korgus, punktid, 0, laius));
   }

   static double kaugus(double x1, double y1, double x2, double y2){
     double vahex=x2-x1;
     double vahey=y2-y1;
     return Math.sqrt(vahex*vahex+vahey*vahey);
   }

   public void paint(Graphics g){
        Image pilt=looLainetus(getSize().width, getSize().height, hx, hy, 20);
        g.drawImage(pilt, 0, 0, null);
   }

   public void update(Graphics g){
     paint(g);
   }


   public void mousePressed(MouseEvent e){
     hx=e.getX();
     hy=e.getY();
     repaint();
   }
   public void mouseReleased(MouseEvent e){}
```

```
   public void mouseClicked(MouseEvent e){}
   public void mouseEntered(MouseEvent e){}
   public void mouseExited(MouseEvent e){}

  public static void main(String argumendid[]){
    Frame f=new Frame();
    f.add(new Pildiloomine3());
    f.setSize(200, 220);
    f.setVisible(true);
  }
}
```

Liikuv lainetus

Kui joonistamise parameetrid kasutajale muuta anda, siis õnnestub lainetus tema tarvis küllalt paindlikuks teha. Parameetrite muutmisel luuakse uus pildiseeria, mis ühtejärge kordudes loob illusiooni lainete liikumisest. Kaadrite arvu vähendades võib liigutamist sujuvuse hinnaga arvutile kergemaks teha. Heledusega saab määrata punase üldist tugevust, kontrastsusega et kui palju laine harjale ning lohule vastavad väärtused erinevad. Üle 255 ega 0 ei lasta väärtustel minna, muidu ei vastaks pilt enam oodatule. Pildiarvutuslõime prioriteeti on alandatud, et eelmine joonis suudaks senikaua edasi joosta kuni uue lõigu tarvis pilte arvutatakse. Uued pildid luuakse nii kerimisribaka antavate parameetrite muutmisel kui ka hiirega kuhugi vajutamisel, mis puhul leitakse lainete suubumisele uus keskkoht. Samuti tulevad uued pildid, kui joonistuskomponendi suurust muudetakse. Nii võib kasutaja valida vastavalt oma arvuti võimsusele piisavalt väikese akna, kus joonis veel mõistliku kiirusega töötada suudab.

```
import java.awt.image.*;
import java.awt.*;
import java.awt.event.*;
import java.applet.Applet;
public class Pildiloomine5 extends Applet implements MouseListener,
          Runnable, AdjustmentListener{
   int hx=75, hy=75;
   int pildilaius, pildikorgus;
   int pildinr=0;
   Scrollbar ootesb=new Scrollbar(Scrollbar.HORIZONTAL, 100, 10, 0, 500);
   Scrollbar lainepikkusesb=new Scrollbar(Scrollbar.HORIZONTAL, 20, 10, 1, 300);
   Scrollbar kaadriarvusb=new Scrollbar(Scrollbar.HORIZONTAL, 20, 3, 1, 50);
   Scrollbar heledusesb=new Scrollbar(Scrollbar.HORIZONTAL, 125, 10, 0, 255);
   Scrollbar kontrastsusesb=new Scrollbar(Scrollbar.HORIZONTAL, 100, 10, 0, 150);
   Canvas louend=new Canvas(){
     public void paint(Graphics g){
       joonista(g);
     }
     public void update(Graphics g){
       paint(g);
     }
   };

   Image pildid[];
   double lainepikkus=20;
   boolean veel;
   int kaadritearv=15;
   public Pildiloomine5(){
     louend.addMouseListener(this);
```

```
  new Thread(this).start();
  Panel p1=new Panel(new GridLayout(5, 2));
  p1.add(new Label("Aeglus:")); p1.add(ootesb);
  p1.add(new Label("Lainepikkus:")); p1.add(lainepikkusesb);
  p1.add(new Label("Kaadrite arv:")); p1.add(kaadriarvusb);
  p1.add(new Label("Heledus:")); p1.add(heledusesb);
  p1.add(new Label("Kontrastsus:")); p1.add(kontrastsusesb);
  setLayout(new BorderLayout());
  add(p1, BorderLayout.SOUTH);
  add(louend, BorderLayout.CENTER);
  lainepikkusesb.addAdjustmentListener(this);
  kaadriarvusb.addAdjustmentListener(this);
  heledusesb.addAdjustmentListener(this);
  kontrastsusesb.addAdjustmentListener(this);
}
public Image looLainetus(int laius, int korgus, int kx, int ky,
                              double lainepikkus, double faas){
     int punktid[] = new int[laius*korgus];
     int nr=0;
     for (int y = 0; y < korgus; y++){
         for (int x = 0; x < laius; x++) {
             int punane=heledusesb.getValue()+(int)(kontrastsusesb.getValue()*
                    Math.sin(kaugus(x, y, kx, ky)*2*Math.PI/lainepikkus+faas));
             if(punane>255)punane=255;
             if(punane<0)punane=0;
             int sinine =  heledusesb.getValue();
             punktid[nr++] = (255<<24)|(punane << 16) | sinine;
         }
     }
     return createImage(new MemoryImageSource(laius, korgus, punktid, 0, laius));
}


public Image[] looPildiseeria(int laius, int korgus, int kx, int ky,
               double lainepikkus, int kaadritearv){
   Image[] pildikaadrid=new Image[kaadritearv];
   for(int i=0; i<kaadritearv; i++){
     pildikaadrid[i]=looLainetus(
         laius, korgus, kx, ky, lainepikkus, 2*Math.PI*i/kaadritearv);
   }
   pildilaius=laius; pildikorgus=korgus;
   return pildikaadrid;
}

public void arvutaPildid(){
 new Thread(){
   public void run(){
    Thread.currentThread().setPriority(Thread.MIN_PRIORITY);
    pildid=looPildiseeria(louend.getSize().width, louend.getSize().height, hx, hy,
      lainepikkusesb.getValue(),    kaadriarvusb.getValue());
    if(pildinr>=pildid.length)pildinr=0;
   }
 }.start();
}

void kontrolliSuurust(){
  if(pildilaius!=louend.getSize().width || pildikorgus!=louend.getSize().height){
     hx=louend.getSize().width/2;
     hy=louend.getSize().height/2;
     arvutaPildid();
  }
}

public void adjustmentValueChanged(AdjustmentEvent e){
  arvutaPildid();
}

public void run(){
 veel=true;
 while(veel){
  if(pildid!=null && ++pildinr>=pildid.length)pildinr=0;
  louend.paint(louend.getGraphics());
  try{Thread.sleep(ootesb.getValue());}catch(Exception e){}
 }
}


static double kaugus(double x1, double y1, double x2, double y2){
  double vahex=x2-x1;
  double vahey=y2-y1;
  return Math.sqrt(vahex*vahex+vahey*vahey);
}

public void joonista(Graphics g){
     if(g==null)return;
     if(pildid!=null && pildinr<pildid.length){
```

```
            g.drawImage(pildid[pildinr], 0, 0, null);
            kontrolliSuurust();
         } else {
            g.drawString("arvutatakse", 10, 50);
            arvutaPildid();
         }
    }

    public void mousePressed(MouseEvent e){
      hx=e.getX();
      hy=e.getY();
      arvutaPildid();
      repaint();
    }
    public void mouseReleased(MouseEvent e){}
    public void mouseClicked(MouseEvent e){}
    public void mouseEntered(MouseEvent e){}
    public void mouseExited(MouseEvent e){}

   public static void main(String argumendid[]){
     Frame f=new Frame();
     f.add(new Pildiloomine5());
     f.setSize(150, 250);
     f.setVisible(true);
   }
}
```

XOR tehe joonistamisel.

Kui 1980ndatel loodi küllalt sujuva liikumisega arvutianimatsioone, siis tuli läbi ajada tunduvalt väiksema jõudlusega arvutitega kui kakskümmend aastat hiljem. Sellegipoolest õnnestus saavutada küllaltki tõetruu elamus. Üheks võlusõnaks sellise liikumise juures olid spraidid ning XOR. Mobiilirakenduste ekraanidel on nimetatud tehnikad nüüdki esirinnas, mujal aga sageli põhjendamatult soiku jäänud. XOR tehte abil aga on praegugi vahel mugavam väikest joonist suurel pinnal liigutada kui graafikalehekülgi või mälupuhvreid kasutada.

Matemaatiline taust.

Tõeväärtuse puhul saab kasutada kaht võimalikku väärtust. Olgu nendeks siis arvud 0 ja 1, sõnad false ja true, värvid must ja valge, perfolindil terve koht / auk või veel mõni teineteist välistav väärtus. Levinumaid tehteid selliste väärtustega on neli ning igaüks sellistest tehetest annab vastuseks samuti väärtuse kahest võimalikust väärtusest. Ja ehk and ehk loogiline korrutamine annab tõese tulemuse vaid siis, kui mõlemad tehtes osalevad väärtused on tõesed. Igal muul juhul on tulemuseks väär väärtus, tähistatagu seda siis arvuga 0, sõnaga false või mõnel muul moel. Java keeles tähiseks üldjuhul kaks ampersandi &&. Või ehk or ehk loogiline liitmine annab tõese tulemuse juhul, kui vähemalt üks osapool on tõene. Java keeles tähiseks üldjuhul kaks püstkriipsu ||. Eitus ehk not eeldab vaid üht väärtust ning tehe keerab talle etteantud väärtuse vastupidiseks. Java keeles tähiseks hüüumärk !. Neljas tehe nimega "välistav või" ehk eXclusive OR (XOR) annab tõese tulemuse juhul, kui tehtes osaleva paari väärtused on erinevad sõltumata elementide järjekorrast. Tehte tähiseks Javas karree ^. Seda salapärast tehet annab joonistamisel enese tarbeks kasutada.

Mustvalge katsetus.

Alljärgnevalt paistab, et osalt üksteise peale on joonistatud kaks musta ruutu. Koht, kuhu musta värvi kahel korral joonistatud paistab valgena. Kui nüüd kõrvale võtta matemaatiline võrdlus, siis võiks tähistada, nagu valge oleks tehtes osalev 0 ja must 1 ning joonistamine oleks XOR-tehte tulemus. Esimese ruudu joonistamisel tekib ilusti must ruut, sest 0 ja 1 on erinevad ning igal pool

on tulemuseks 1. Teise ruudu puhul aga on osalt aluspinnal nullid, osalt ühed. Kus ennist oli valge taust all, sinna tekib must värv. Kus aga enne olid musta värvi punktid siis teise musta värvi pealejoonistamisel on tehteks 1^1 ning tulemuseks 0 ehk valge. Siitkaudu paistavad välja XORi abil joonistamise kaks tähtsamat väljundit:

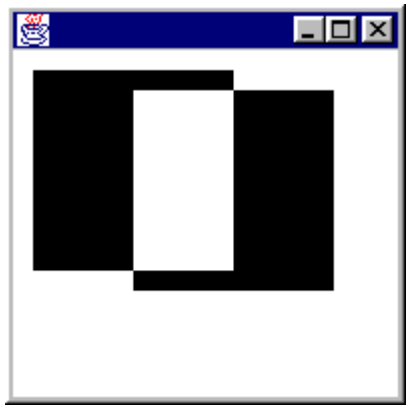
- Eelmine pilt paistab alt välja
- Teist korda kujundit samale kohale joonistades joonistatav kujund kaob ning taastub esialgne pilt.

Nende aga eeskätt viimase omaduse tõttu võimaldab selline joonistusvõte liikumise juures märgatavalt arvuti ressursse kokku hoida.

Kood näeb välja nagu iga muu tavalise joonistamisega seotud rakenduse juures. Vaid paint-meetodis on enne joonistuskäskude käivitamist antud käsuks g.setXORMode(Color.white), mis teatab, et sellest hetkest alates tuleb joonistamisel rakendada välistava või tehet. Ning et kui joonistatakse peale sama värvi punkt kui juhtub all olema, siis olgu uue punkti värviks valge. Teist korda joonistamisel, ehk siin näites siis valge peale joonistamisel saadakse tulemuseks joonistatav värv.

```
import java.applet.Applet;
import java.awt.*;

public class XORJoonis1 extends Applet{
  public void paint(Graphics g){
    g.setColor(Color.black);
    g.setXORMode(Color.white);
    g.fillRect(10, 10, 100, 100);
    g.fillRect(60, 20, 100, 100);
  }
  public static void main(String[] argumendid){
     Frame f=new Frame();
     f.add(new XORJoonis1());
     f.setSize(200, 200);
     f.setVisible(true);
  }
}
```

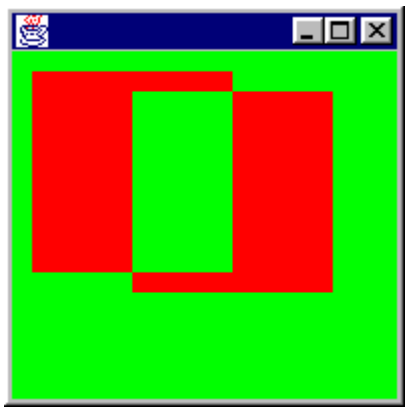


Värviline XOR

Sama lähenemine toimub ka värviliste kujundite juures. Olles kahel korral joonistanud rohelisele taustale sinise punkti, on tulemuseks taas roheline.

```
import java.applet.Applet;
import java.awt.*;

public class XORJoonis2 extends Applet{
  public void paint(Graphics g){
    setBackground(Color.green);
    g.setColor(Color.blue);
    g.setXORMode(Color.white);
    g.fillRect(10, 10, 100, 100);
    g.fillRect(60, 20, 100, 100);
  }
  public static void main(String[] argumendid){
     Frame f=new Frame();
     f.add(new XORJoonis2());
     f.setSize(200, 200);
     f.setVisible(true);
  }
}
```

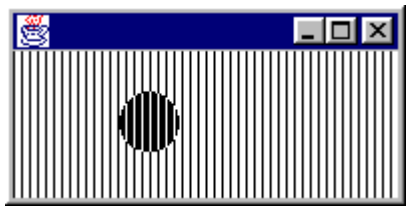


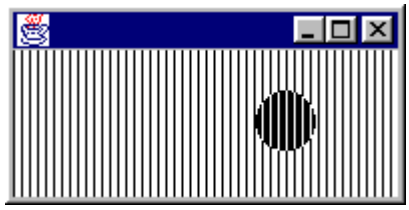
Liikumine

Enam on XORi kasu tunda liigutamise puhul. Triibulisel taustal ringi ilma vilkumata liikuma panek eeldaks vähemasti topeltpuhverdust. Nõnda aga pääseb palju väiksema joonistuspinnaga. Igal korral tuleb üle käia vaid punktid kus ring enne paiknes või kuhu ta uue sammu juures jõuab. Mõeldav oleks ka vaid ühe sammu jooksul muudetava ala puhvrisse võtmine, kuid XORi lähenemine on tuntavalt lihtsam. Eriti, kui oleks tahtmist korraga liikuma panna mitu kujundit.

```
import java.applet.Applet;
import java.awt.*;

public class XORJoonis3 extends Applet{
  public void paint(Graphics g){
    g.setColor(Color.black);
    for(int i=0; i<getSize().width; i++){
          //triibuline taust
      if(i%4==0){
        g.drawLine(i, 0, i, getSize().height);
      }
    }
    g.setXORMode(Color.white);
    for(int i=0; i<getSize().width; i++){
     g.fillOval(i, 20, 30, 30);
     try{Thread.sleep(50);} catch (Exception e){}
     g.fillOval(i, 20, 30, 30);
    }
  }
  public static void main(String[] argumendid){
     Frame f=new Frame();
     f.add(new XORJoonis3());
     f.setSize(200, 100);
     f.setVisible(true);
  }
}
```



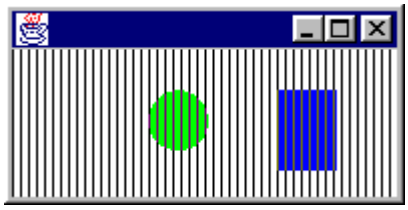


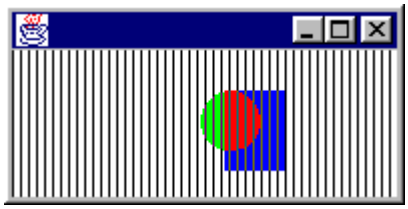
Kujundite kattumine liikumisel

Järgnevalt ongi näha mitu liikuvat kujundit. Ning taust paistab läbi nii kummastki kujundist eraldi kui üksteise peale jõudnud kujundite puhul.

```
import java.applet.Applet;
import java.awt.*;

public class XORJoonis4 extends Applet{
  public void paint(Graphics g){
    g.setColor(Color.black);
    for(int i=0; i<getSize().width; i++){
       //triibuline taust
      if(i%4==0){
        g.drawLine(i, 0, i,
                   getSize().height);
      }
    }
    g.setXORMode(Color.white);
    for(int i=0; i<getSize().width; i++){
     g.setColor(Color.green);
     g.fillOval(i, 20, 30, 30);
     g.setColor(Color.blue);
     g.fillRect(200-i, 20, 30, 40);
     try{Thread.sleep(50);}
     catch (Exception e){}
     g.setColor(Color.green);
     g.fillOval(i, 20, 30, 30);
     g.setColor(Color.blue);
     g.fillRect(200-i, 20, 30, 40);
    }
  }
  public static void main(
      String[] argumendid){
     Frame f=new Frame();
     f.add(new XORJoonis4());
     f.setSize(200, 100);
     f.setVisible(true);
  }
}
```





Liikumine omaette lõimes

Võimaluse korral pole liikumise tarvis mõeldud viivitust siiski viisakas paint-käskluse sisse kirjutada. Meetod paint on ette nähtud pildi staatiliseks kuvamiseks, viivitused selles kipuvad häirima akna suuruse muutmise sujuvust. Hoopis viisakam on paigutada joonistamine omaette lõime ning lasta sellel siis ringi joonistamiste ja kustutamiste eest hoolt kanda. Kuna joonistusalgoritmiks on XOR, siis kustutamiseks piisab samast käsust kui joonistamise puhulgi. Teistkordsel sama kujundi samasse kohta joonistamisel taastatakse algne olukord.

Lõime käivitamine on toodud meetodisse paint, sest varem ei pruugi komponendi peale veel võimalik olla joonistada. Ka run-meetodisse oleks võimalik luua kontroll, et joonistataks alles siis, kui getGraphics on tagastanud joonistamisvõimelise objekti, kuid praegune näide peaks ressursse vähem kulutama.

```
import java.applet.Applet;
import java.awt.*;

public class XORJoonis5 extends Applet implements Runnable{
  boolean algus=true;
  public void paint(Graphics g){
    g.setColor(Color.black);
    for(int i=0; i<getSize().width; i++){       //triibuline taust
      if(i%4==0){
        g.drawLine(i, 0, i, getSize().height);
      }
    }
    if(algus){
      new Thread(this).start();
      //kui paint käivitub, siis selleks ajaks on juba võimalik joonistada.
      algus=false;
    }
  }
  public void run(){
   Graphics g=getGraphics();
```

```
   g.setXORMode(Color.white);
   while(true){
    for(int i=0; i<getSize().width; i++){
     g.setColor(Color.green);
     g.fillOval(i, 20, 30, 30);
     g.setColor(Color.blue);
     g.fillRect(200-i, 20, 30, 40);
     try{Thread.sleep(50);} catch (Exception e){}
     g.setColor(Color.green);
     g.fillOval(i, 20, 30, 30);
     g.setColor(Color.blue);
     g.fillRect(200-i, 20, 30, 40);
    }
   }
  }
  public static void main(String[] argumendid){
     Frame f=new Frame();
     f.add(new XORJoonis5());
     f.setSize(200, 100);
     f.setVisible(true);
  }
}
```

Jällegi võib edaspidi siinseid näiteid oma koodis aluseks võtta ning soovi korral midagi märgatavalt ilusamat ja põhjalikumat kokku panna.

Ülesandeid

Ekraanipilt

- Kopeeritud ekraanipildi parem alumine nurgatükk näidatakse suurendatuna üle ekraani.
- Kopeeritud ekraanipildi nurgatükk liigub alaservas edasi-tagasi.
- Kopeeritud ekraanipildi serv jagatakse mõnekümne punkti pikkuse küljega ruutudeks. Need ruudud liiguvad ekraani servas ringiratast ümber ekraani.

Kopeeritud ekraanipilt

- Programmi käivitamisel kopeeritakse parasjagu nähtav ekraanipilt mällu ning näidatakse kasutajale.
- Pilt kopeeritakse mällu kogu ekraani ulatuses ning näidatakse ekraanile klassi Window abil, seega ilma välimise raamita ning peaks näima tõetruu.
- Loodud ekraanipildile hakkavad juhuslikesse kohtadesse tekkima täpikesed, kuni lõpuks on kogu pilt kirju.
- Ekraanipilt salvestatakse vähendatuna jpeg-faili.
- Väljastatakse, kas ja kus leidub ekraanil musta värvi täpp
- Ekraanipildilt otsitakse üles ja tehakse topeltklõps akende ülanurgas olevatele sulgemisristikestele.

Pildi koostamine.

- Loo MemoryImageSource abil rohekas pilt.
- Pilt on ülalt servast tumedam ning alt heledam.
- Kopeeri ekraanipilt ning keera see tagurpidi.
- Saada loodud pilt võrgu kaudu teise arvutisse vaatamiseks.

Joon pildil

- Loo MemoryImageSource abil pilt, kus mustal taustal oleks valge joon.
- Loo joon keskelt valge, servad lähevad aga sujuvalt taustaks üle.
- Pane eelmises punktis loodud joon horisontaalsuunas liikuma.

Värviüleminek

- Loo MemoryImageSource abil värviüleminek punaselt sinisele.
- Kasutaja märgib pildil kolm punkti. Ühes neist on maksimumis punane, teises roheline ning kolmandas sinine värv. Värviüleminekud pildil on sujuvad.
- Märgitud haripunktide asukohti saab kasutaja hiirega hiljem liigutada.