Filling polygons

How to determine the interior of a polygon:

Odd parity rule: A point lies inside a polygon if a line starting at the point intersects with an odd number of edges of the polygon.

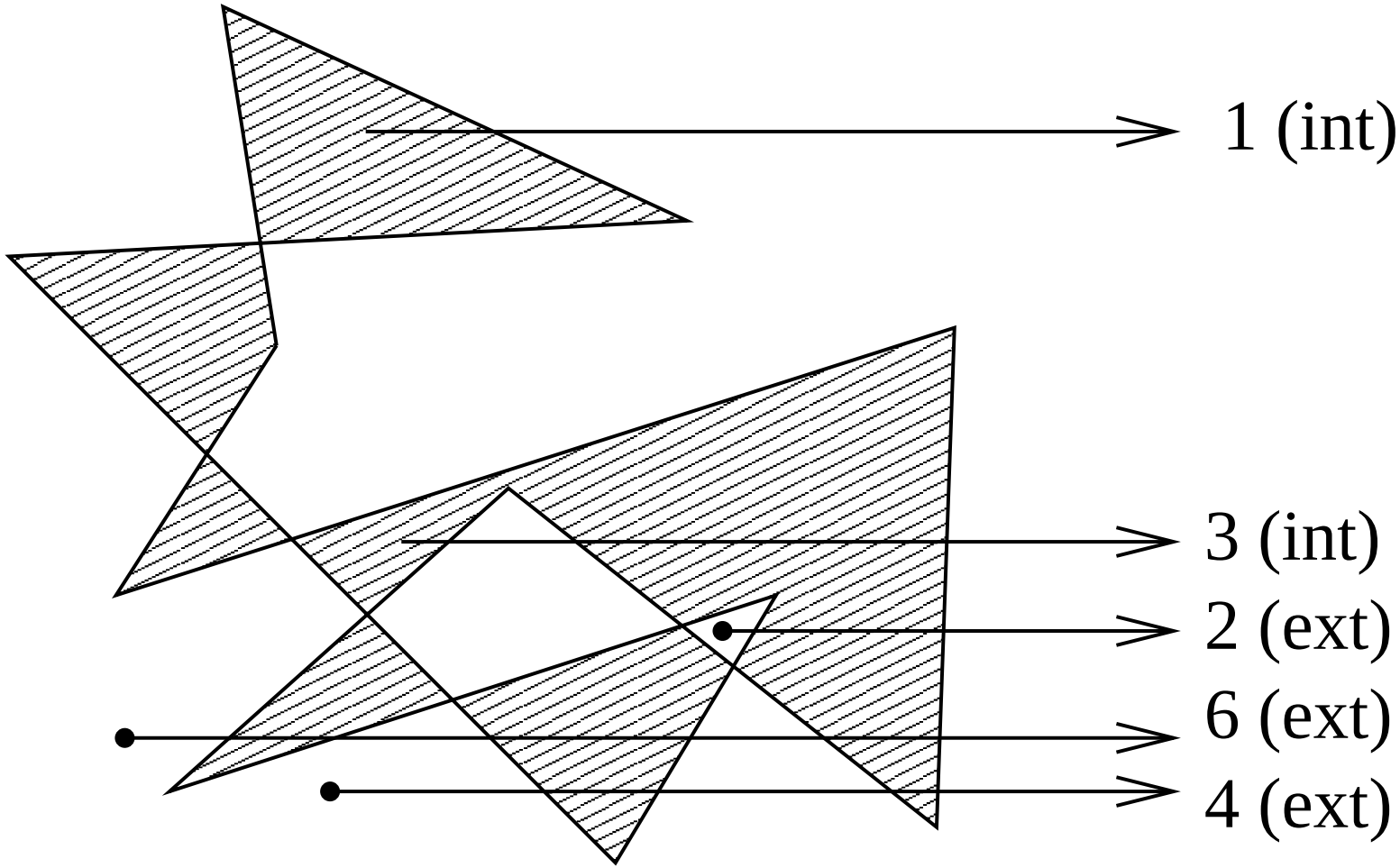
Java 2D provides the method `g2d.fill(s);` for filling a `Shape s`.

In order to apply the odd parity rule for filling, f.e. a `GeneralPath gp`,

```
gp.setWindingRule(GeneralPath.WIND_EVEN_ODD)
```

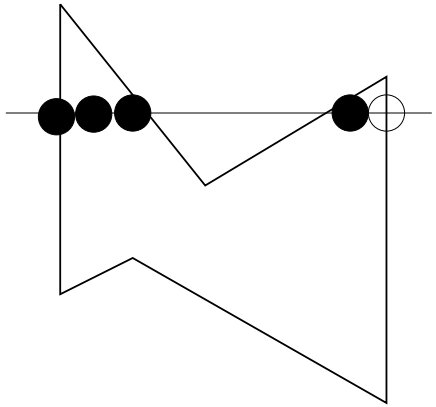
is needed.

Odd parity rule



Filling polygons

In order to fill a polygon with a given colour, a scan line is used for each y-value and the points to be drawn along the scan line are determined.



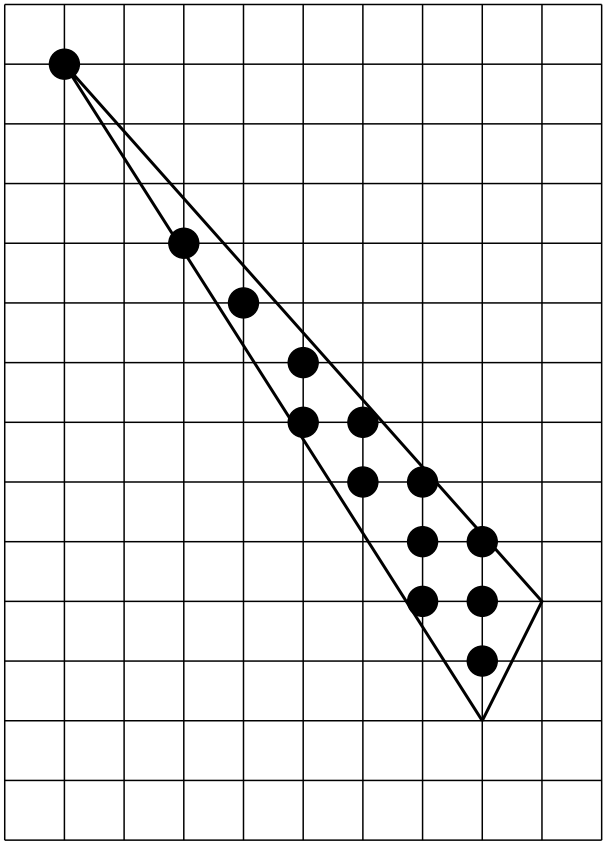
1. Determine all intersection points of the scan line with all polygon edges.
2. Sort the intersection points in increasing order w.r.t. their x-coordinates.

Filling polygons

3. Apply the odd parity rule to determine which points lie inside the polygon.
   - Start from the left with even parity.
   - Jump to the first intersection point of the scan line with the polygon, (change the parity to odd and) draw all pixel until the next intersection point is reached.
   - Then skip the pixels until the next intersection point is reached. Draw all pixel until the next intersection point is reached. . . .

Filling polygons

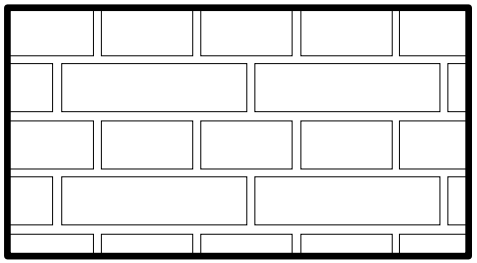
Aliasing effect occurring when thin polygons are filled:

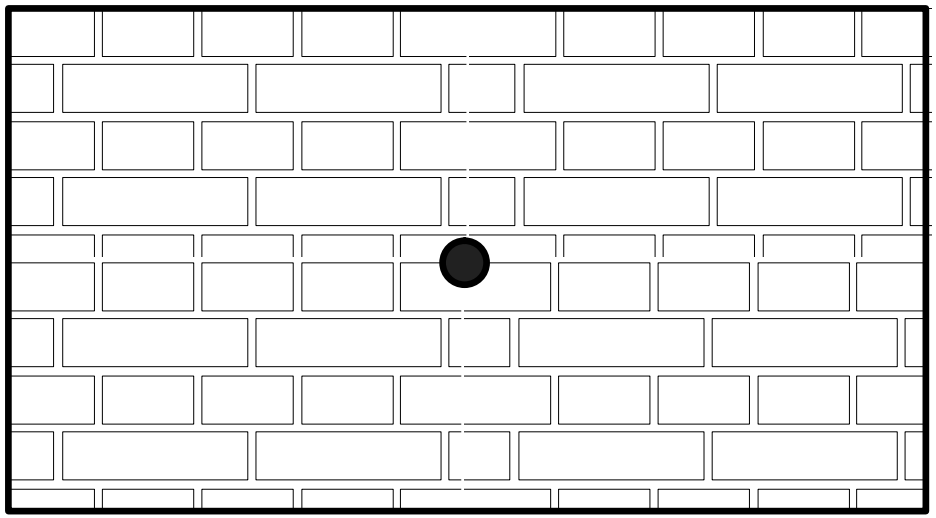


Filling closed curves

- Filling circles, ellipses, arcs of circles or ellipses is done in the same way.
- Symmetry properties should be exploited to reduce the calculations.
- Filling of arbitrary closed curves is also done in the same way.
- When the boundary of an area is drawn using antialiasing, the pixels belonging to the edge are no longer uniquely defined.

Textures





Using a texture once or more than once for filling an area.

An **anchor** must be specified. The anchor defines a starting point from which the texture is laid like tiles.

Buffered Images

A `BufferedImage` is an image in the memory

- which can be drawn on and
- which can also be drawn on an `Image`.

```
BufferedImage bi =
    new BufferedImage(width,height,
            BufferedImage.TYPE_INT_RGB);

Graphics2D g2dbi = bi.createGraphics();
```

Buffered Images

```
g2dbi.draw(...);

g2dbi.fill(...)

g2d.drawImage(bi,xpos,ypos,null);
```

`drawImage` can be used to draw on a window or on a `BufferedImage`.

Double Buffering

Principle of double buffering: Use

- a `BufferedImage theBackground` containing the static background that will not change,
- a `BufferedImage theImage` on which after each step of animation first the `theBackground` is drawn and then the animated objects, and
- a Graphics2D object (as usual `g2d`) which is used for drawing `theImage` on the window.

Double Buffering

Window to be drawn upon (`BufferedImageDrawer`):

Attributes:

```
public BufferedImage bi;
public Graphics2D g2dbi;
```

Methods:

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

public void update(Graphics g)
{
  g2d = (Graphics2D) g;
  g2d.drawImage(bi,0,0,null);
}
```

Double Buffering

Implementation of these methods in the class `BufferedImageDrawer`. Necessary steps:

- Computations take place in a class extending the Java class `TimerTask`.
- The `run` method in this class contains the computations that were previously done in the FOR-loop of the `paint`-Method.
- Drawing is not done with the `Graphics2D` object of the `paint` method of the window, but with the `Graphics2D` object of the corresponding `BufferedImage bi`.

Double Buffering

- Instead of clearing the drawing area (painting the `BufferedImage bi` white) in each step, another `BufferedImage` containing the background is drawn on `bi` each step.
- The `repaint` method of the `BufferedImageDrawer` must be called at the end of the `run` method.

• *Double Buffering*

Initialisations take place in the `main` method and in the constructor of the specific class.

The `run` method, in which the sequence of images is computed and which initiates drawing, is called repeatedly by

```
Timer t = new Timer();
t.scheduleAtFixedRate(dbce,0,delay);
```

(see `DoubleBufferingClockExample.java`)

Loading images

Using

```
Image theImage;
theImage = new javax.swing.ImageIcon(
                "filename.jpg").getImage();
```

an image can be loaded. This image can be drawn at the position (x, y) using

```
g2d.drawImage(theImage,x,y,null);
```

`g2d` is the Graphics2D object of the window or of a `BufferedImages`.

Saving images

- Generate a `BufferedImage theImage`.
- Generate the corresponding Graphics2D object: `Graphics2D g2dImage = theImage.createGraphics();`
- Draw the image (using `g2dImage`).
- Generate a `FileOutputStream fos` for the file in which the image should be stored.
- Store the image using the `JPEGImageEncoder`.

(see `ImageSavingExample.java`)

Saving images

```
try
{
  FileOutputStream fos = new
        FileOutputStream("test.jpg");
  JPEGImageEncoder jie =
       JPEGCodec.createJPEGEncoder(fos);
  jie.encode(theImage);
}
catch (Exception e)
{
  System.out.println(e);
}
```

Textures in Java 2D

Using a texture `theImage` (for instance, obtained from JPEG image) once to fill a `Shape s`:

```
Shape clipShape = g2d.getClip();
g2d.setClip(s);
g2d.drawImage(theImage,50,50,null);
g2d.setClip(clipShape);
```

Textures in Java 2D

Filling a `Shape s` repeatedly with a texture `theImage` given as a JPEG image:

- The texture must be loaded and drawn onto a `BufferedImage buffImage`.
- Generate a

```
TexturePaint tp =
new TexturePaint(buffImage,
  new Rectangle(0,0,
                buffImage.getWidth(),
                buffImage.getHeight()));
g2d.setPaint(tp);
g2d.fill(s);
```

Text

Letters and symbols can be saved in two forms:

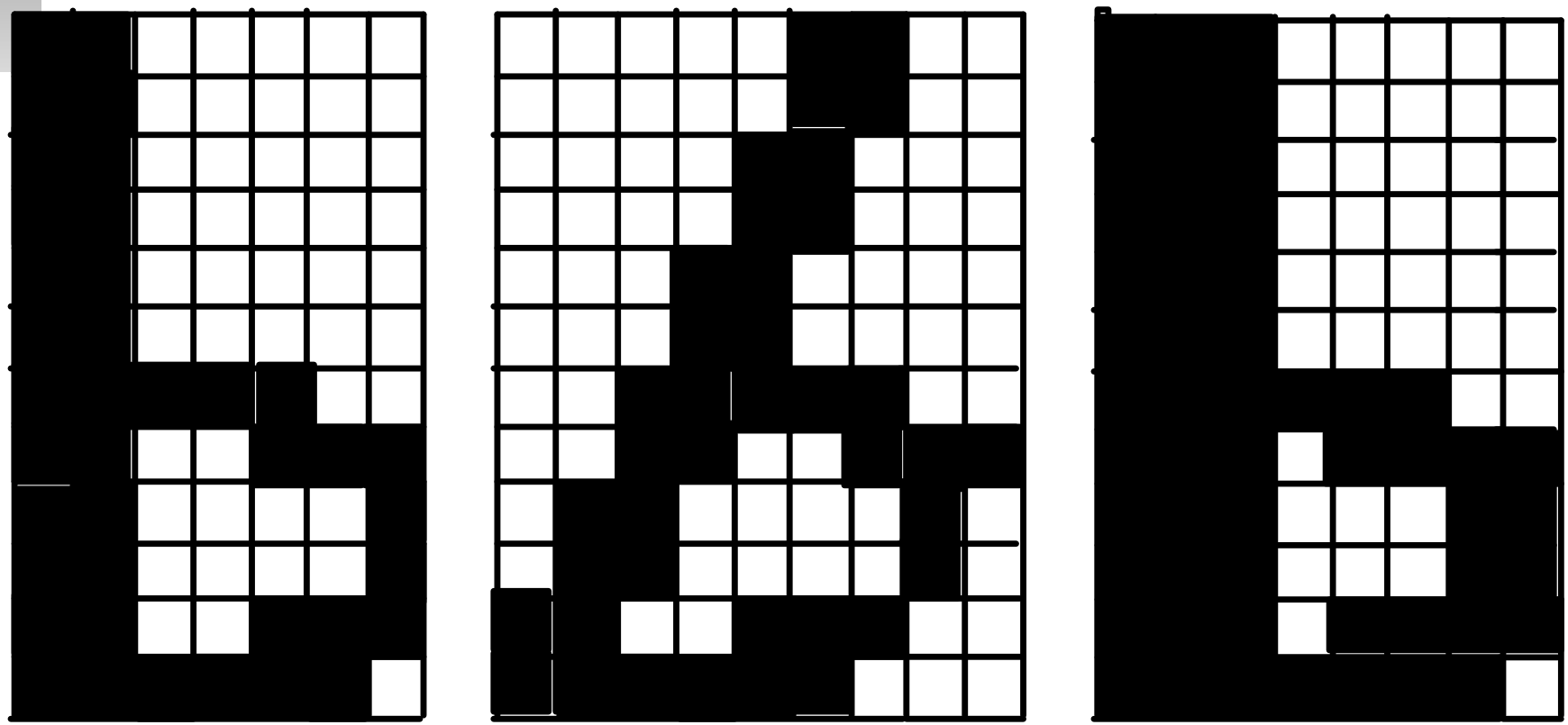
- As a bitmap (pixel matrix) or
- in the form of polygons or curves.

When bitmaps are used, each symbol must be stored in all sizes and all styles (normal, boldface, italic, boldface italic).

A representation as polygons or curves is scalable. Different sizes can be computed from one representation of the letter.

Text

An old, simple, but not very nice technique for generating italic and boldface symbols from bitmaps:



normal *italic* **boldface**

Text in Java 2D

Drawing/writing text:

```
g2d.drawString("text",posx,posy);
```

Choosing the font:

```
Font f = new Font("type",Font.STYLE,size);
```

Available fonts (for type):

```
Font[] fl =
GraphicsEnvironment.
  getLocalGraphicsEnvironment(
                        ).getAllFonts();
  for (int i=0; i<fl.length; i++)
  {
    System.out.println(fl[i].getName());
  }
```

Values for `STYLE`:

- `PLAIN` (normal)
- `ITALIC` (italic/slanted)
- `BOLD` (boldface)
- `ITALIC | BOLD` (italic and boldface)

`size` specifies the size of the font (in the unit pt, not in pixels).

After calling the method `g2d.setFont(f)`, `drawString` is using the font `f`.

Modifying fonts

Applying a transformation `affTrans` to the whole font:

```
Font transformedFont =
                f.deriveFont(affTrans);
```

Transformation of single symbols of a string `s`:

```
FontRenderContext frc =
                g2d.getFontRenderContext();
GlyphVector gv =
                f.createGlyphVector(frc,s);
```

Modifying symbols

```
Point2D p = gv.getGlyphPosition(i);

Shape glyph = gv.getGlyphOutline(i);

Shape transGlyph =
        at.createTransformedShape(glyph);

g2d.fill(transfGlyph);
```

(see `TextExample.java`)

Grey & colour images

Intensity levels for grey or colour values:

Pixels cannot be only black or white. Instead a finite number of intensity levels (grey-levels or intensity levels for colours) is available.

The human perception of light intensities is mainly a relative one.

A 60-watt light bulb is much brighter in comparison to a 20-watt light bulb than a 100-watt light bulb in comparison to a 60-watt light bulb.

Grey & colour images

The intensity levels should not increase linearly, but exponentially.

Starting from the lowest intensity ((almost) black for grey values) I_0, the levels should be chosen according to

$$I_0 = I_0,\ I_1 = rI_0,\ I_2 = rI_1 = r^2 I_0, \ldots$$

up to a maximum intensity $I_n = r^n I_0$.

Grey & colour images

The average human vision system is able to distinguish between grey-levels when they differ by at least 1%, i.e., if $r > 1.01$ holds.

Assuming a maximum intensity $I_n = 1$ and a minimum intensity I_0, depending on the output device, from

$$1.01^n I_0 \leq 1,$$

follows that more than

$$n \leq \frac{\ln\left(\frac{1}{I_0}\right)}{\ln(1.01)}$$

grey-levels will not contribute to better image quality.

Grey & colour images

Medium	I_0 (ca.)	Max. no. of grey-levels
monitor	0.005-0.025	372-533
newspaper	0.1	232
photo	0.01	464
slide	0.001	695

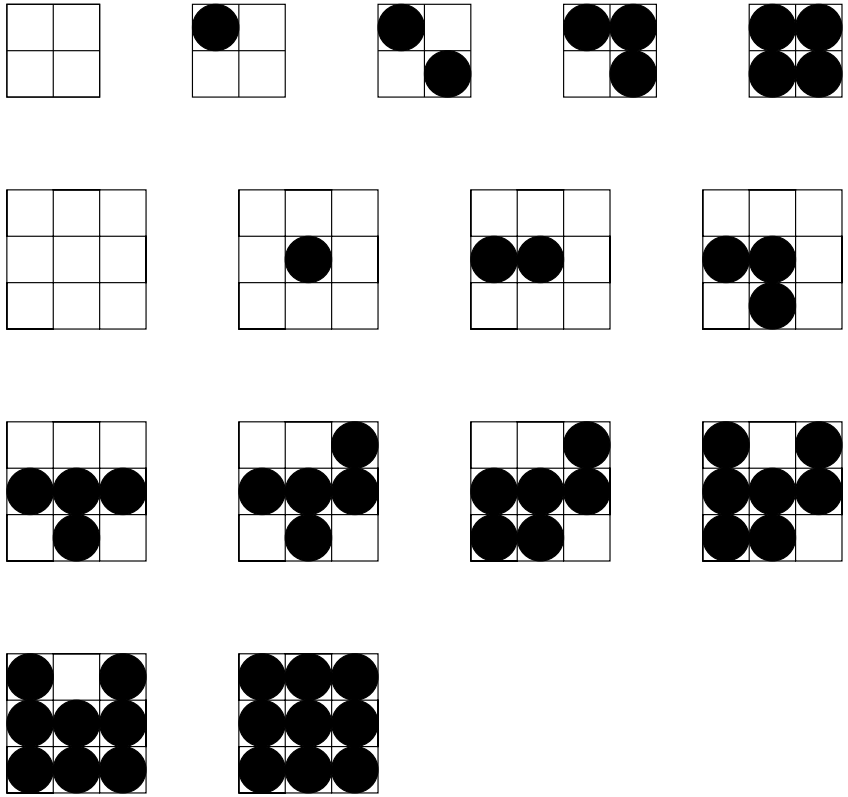
If an output device allows only binary pixels, for instance a black-and-white laser printer, different intensity levels can be represented for the price of a lower resolution.

Halftoning

Combining 2×2-Pixel to a larger pixel, 5 intensity levels can be represented. For 3×3 pixels 10, for $n \times n$ pixels $n^2 + 1$.

- The coarsened resolution must still be high enough so that the pixel raster is not immediately visible.
- The k pixels for the grey-level k should be chosen in such a way that they are neighbouring pixels and they do not form a regular pattern like a line. Otherwise, this might introduce new visual artifacts like stripes in an area of identical intensity values.

Halftoning



Representation by dither matrices:

$$D_2 = \begin{pmatrix} 0 & 2 \\ 3 & 1 \end{pmatrix}, \qquad D_3 = \begin{pmatrix} 6 & 8 & 4 \\ 1 & 0 & 3 \\ 5 & 2 & 7 \end{pmatrix}$$

Halftoning

A similar refinement technique can be applied when nonbinary intensity levels are available in the first place. For instance, using 2×2 pixel matrices where each pixel can have four different intensity levels yields 13 possible intensity levels:

$$\begin{pmatrix} 0 & 0 \\ 0 & 0 \end{pmatrix}, \quad \begin{pmatrix} 1 & 0 \\ 0 & 0 \end{pmatrix}, \quad \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}, \quad \begin{pmatrix} 1 & 1 \\ 0 & 1 \end{pmatrix}, \quad \begin{pmatrix} 1 & 1 \\ 1 & 1 \end{pmatrix},$$

$$\begin{pmatrix} 2 & 1 \\ 1 & 1 \end{pmatrix}, \quad \begin{pmatrix} 2 & 1 \\ 1 & 2 \end{pmatrix}, \quad \begin{pmatrix} 2 & 2 \\ 1 & 2 \end{pmatrix}, \quad \begin{pmatrix} 2 & 2 \\ 2 & 2 \end{pmatrix}, \quad \begin{pmatrix} 3 & 2 \\ 2 & 2 \end{pmatrix},$$

$$\begin{pmatrix} 3 & 2 \\ 2 & 3 \end{pmatrix}, \quad \begin{pmatrix} 3 & 3 \\ 2 & 3 \end{pmatrix}, \quad \begin{pmatrix} 3 & 3 \\ 3 & 3 \end{pmatrix}$$

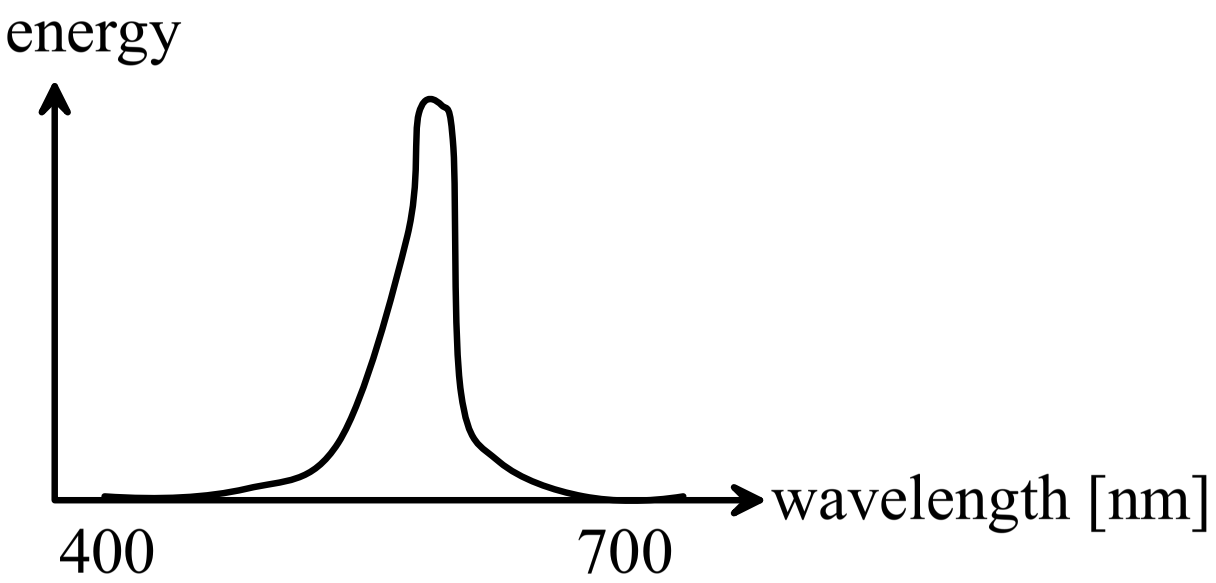
Colour models

- Theoretical description of a colour: Intensities in the colour spectrum.
- Main characteristics of a colour:
  - **Hue** corresponding to the dominant wavelength in the spectrum of the colour,
  - **Saturation** or purity which is high when the spectrum consists of a narrow peak at the dominant wavelength, and which is low for a flatter spectrum, and
  - Intensity or **lightness** depending on the energy of the spectrum. The higher the energy of the single frequencies, the higher is the intensity.

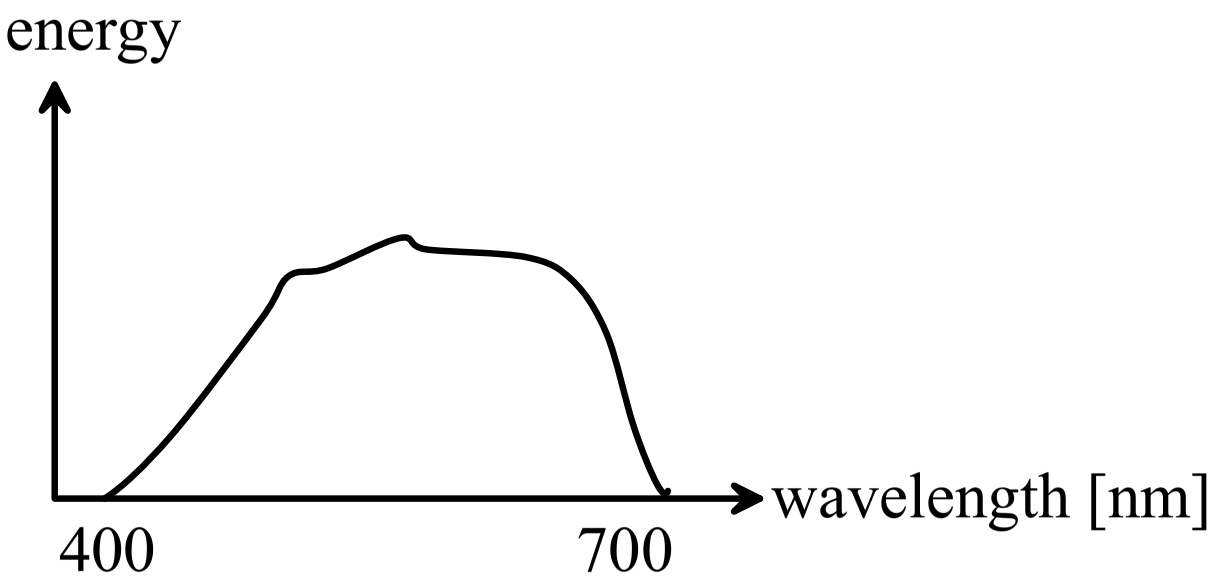
Colour models

visible range: ca. 400nm (violet) to 700nm (red)

High saturation: High concentration in the spectrum.



Low saturation: Wide-spread spectrum.



Additive/subtractive colour models

- Additive colour models for light:
  - Black (dark) background
  - red+green+blue yields white.
  - Example: monitor
- subtractive colour model for colours:
  - White (light) background
  - red+green+blue yields black/dark brown.
  - Example: printer

The RGB model

Standard colour model for monitors: **RGB model**.

Each colour is composed of the three primary colours red, green and blue.

$$\text{colour} = r \cdot R + g \cdot G + b \cdot B$$

where $r, g, b \in [0, 1]$.

$(0, 0, 0)$ corresponds to black, $(1, 1, 1)$ is white, (x, x, x) defines a lighter or darker grey, depending on the choice of x, $(1, 0, 0)$ encodes red, $(0, 1, 0)$ green and $(0, 0, 1)$ blue.

The RGB model

Usually, for the coding of the intensity of a colour, one byte is used, so that each of the primary colours has 256 different levels of intensity.
In Java 2D, a colour can be defined using the constructors

- `new Color(float r, float g, float b)` (`r,g,b`$\in [0,1]$) or
- `new Color(int r, int g, int b)` (`r,g,b`$\in \{0,1,\ldots,255\}$).

Not every colour can be represented within the RGB model.

The CIEXYZ model

For this reason, the Commission Internationale de l'Éclairage (CIE) introduced a model with three artificial colours X, Y and Z which can represent any other colour.

$$\text{colour} = x \cdot X + y \cdot Y + z \cdot Z$$

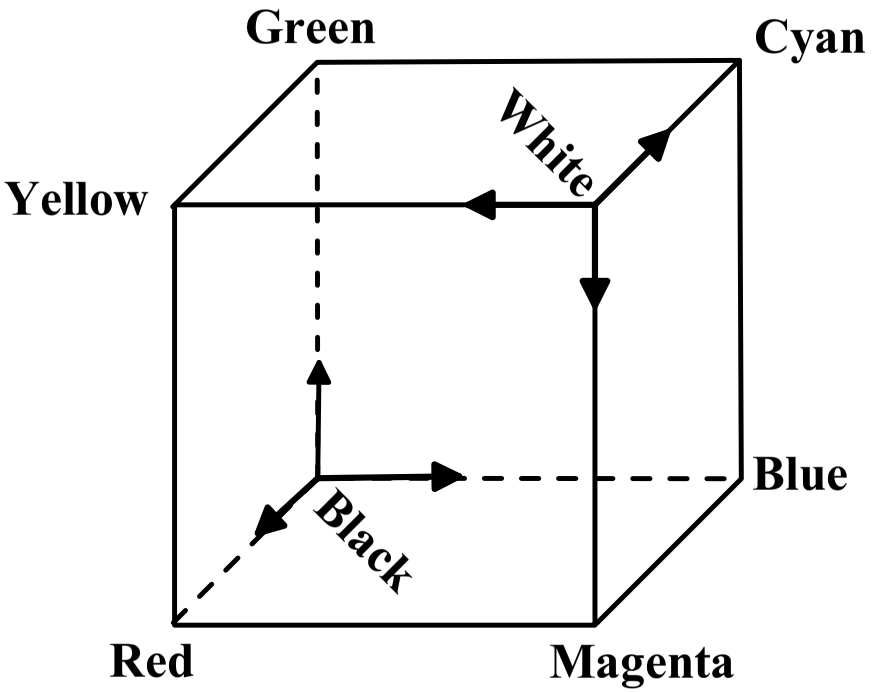
where $x, y, z \in [0, 1]$.

However, finding suitable combinations of the three artificial colours to model a desired colour is not very intuitive, so that the *CIEXYZ model* is seldom used.

The CMY model

The subtractive *CMY model* is the dual to the RGB model and is used for printers and plotters

The primary colours are cyan, magenta and yellow.



$$\begin{pmatrix} C \\ M \\ Y \end{pmatrix} = 1 - \begin{pmatrix} R \\ G \\ B \end{pmatrix}$$

The CMYK model

The CMYK model uses black (K: key) as a fourth colour.

The CMYK model is the standard colour model for printers yielding better colours than the CMY model.

CMY $\rightarrow$ CMYK

$$\begin{aligned} K &:= \min\{C, M, Y\} \\ C &:= C - K \\ M &:= M - K \\ Y &:= Y - K \end{aligned}$$

$\rightarrow$ At least one of the four values C, Y, M, K is 0.

The YIQ model

The YIQ model is not based on three primary colours as in the RGB and the CMY model, but it uses the three components

luminance Y and

chromaticity I, Q

to define a colour.

Used in the American NTSC television norm.

Colour $\rightarrow$ grey image: Use only the Y-value.

The YIQ model

$$\begin{pmatrix} Y \\ I \\ Q \end{pmatrix} = \begin{pmatrix} 0.299 & 0.587 & 0.114 \\ 0.596 & -0.275 & -0.321 \\ 0.212 & -0.523 & 0.311 \end{pmatrix} \cdot \begin{pmatrix} R \\ G \\ B \end{pmatrix}$$

Advantages of the YIQ model:

- Easy transformation from colours to grey-levels, determined only by the Y-value.
- Helpful when computer monitors with different brightness should be adjusted, so that the colours they show are more or less the same for identical RGB values. Much easier to adjust only the Y-value than to adjust the three values R, G and B at the same time.

The HSV model

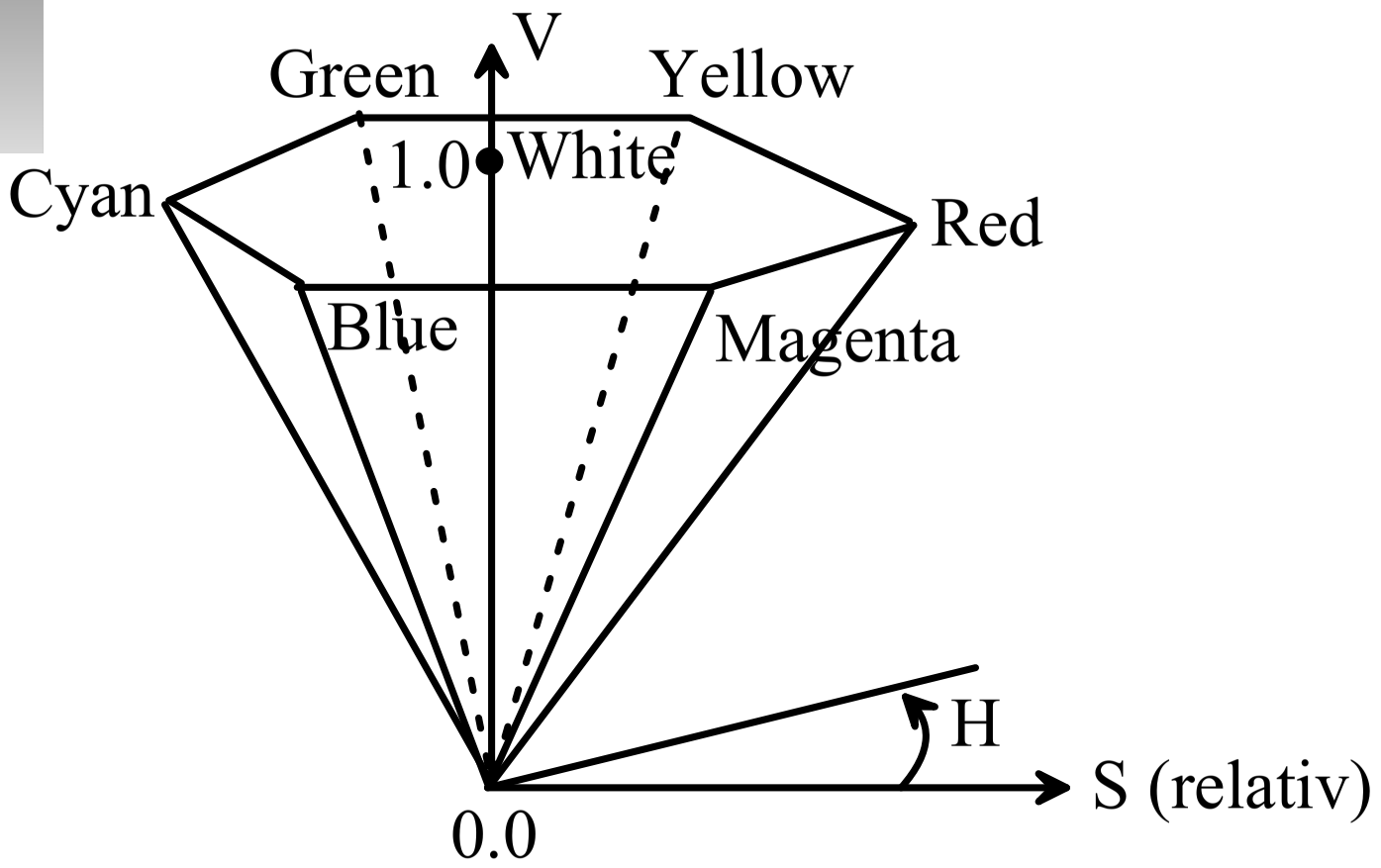
Parameters:

- hue
- saturation
- value

Model in the form of an upside down pyramid:

- Tip of the pyramid: black.
- The hue H is given by the angle.
- Saturation (S): 0 in the middle, 1 at the surface.
- The middle axis determines the value V.

The HSV model



The HLS model

Similar principle as in the HSV model.

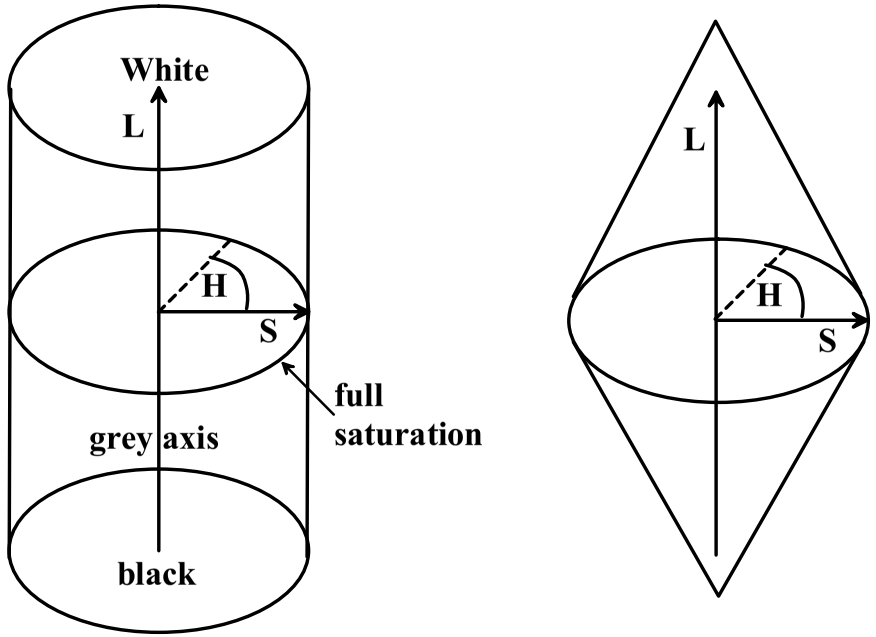
Hue: Angle between $0°$ and $360°$.

Lightness: Value between 0 and 1.

Saturation: Distance of a colour to the centre (representing grey values) between 0 and 1.

For hue: Colour circle with red at $0°$, yellow at $60°$, green at $120°$, blue at $240°$ and violet at $300°$.

The HLS model



Sometimes a double cone is preferred for the HLS model. The double cone reflects the fact that it does not make sense to speak of saturation for the colours black and white, when grey-values are already characterised by the luminance value.

Transformation from RGB to HLS

```
max = Max(R,G,B);
min = Min(R,G,B);
L = (max+min)/2;
if (min==max)
{
  S = 0;  //H arbitrary
  return;
}
mm = max-min;
if (L<=0.5) S = mm/(max+min); else S = mm/(2-max-min);
r = (R-min)/mm; g = (G-min)/mm; b = (B-min)/mm;
if       (R==max) H = g-b;
else {if (G==max) H = 2+b-r;
else               H = 4+r-g;}
if (H<0) H = H+6;
H = H*60;
```

Perception-oriented colour models

The HSV and the HLS model belong to the class of **perception-oriented colour models** since they reflect the intuitive perception of colours better.

Specifying the parameters of a desired colour is easier with perception-oriented models than with models like RGB or CMY.

CNS is another perception-oriented colour model.

The CNS model

As in HSV and HLS: Colours are defined by hue (colour), saturation and lightness.

Instead of numbers, words are used for the values:

Colour: purple, red, orange, brown, yellow, green, blue (with further refinements like yellowish green, green-yellow, greenish yellow)

Lightness: very dark, dark, medium, light, very light

Saturation: greyish, moderate, strong, vivid

Restriction to 560 combinations, but very intuitive.

Java 2D: Colour models

Java 2D supports the following colour models (`ColorSpace` types):

CIEXYZ: `TYPE_XYZ`

RGB: `TYPE_RGB`

Grey-value images: `TYPE_GRAY`

HSV: `TYPE_HSV`

CMY: `TYPE_CMY`

CMYK: `TYPE_CMYK`

Java 2D: RGB colours

RGB colours can be defined with the constructor `new Color(...)`.

Possible arguments of the constructor:

- Three `float`-values between 0 and 1, specifying the red, green and blue intensity, respectively.
- Three `int`-values between 0 and 255, specifying the red, green and blue intensity, respectively.
- A single `int`-value interpreted as a byte vector of four values. Three bytes are used for RGB, the fourth one as a so-called alpha-value which defines how transparent the colour is.

Java 2D: RGB colours

The method `g2d.setPaint(col)` sets the drawing colour to `col`.

The class

```
GradientPaint gradPaint =
    new GradientPaint(x0,y0,colour0,
                      x1,y1,colour1, repeat);
```

allows colour interpolation in terms of colour gradients.

- Colour interpolation takes place along the vector from `(x0,y0)` to `(x1,y1)`.
- The `colour0` is used in point `(x0,y0)`, the `colour1` in point `(x1,y1)`.

Java 2D: RGB colours

- RGB-values are interpolated along this vector by corresponding convex combinations of `colour0` and `colour1`. (The same applied to vectors to the specified vector.)
- `repeat` is a value of type `boolean`.
  - In case of `true`, the colour gradient is repeated again and again, reversing its direction each time.
  - In case of `false`, `colour0` is used before the point `(x0,y0)` and after `(x1,y1)`.

Use: `g2d.setPaint(gradPaint);`

Colour interpolation

`GradientPaint` interpolates colours using convex combinations:

$$(r, g, b) = (1 - \alpha) \cdot (r_0, g_0, b_0) + \alpha \cdot (r_1, g_1, b_1) \qquad (\alpha \in [0, 1])$$

Colour interpolation for textures to avoid the tiling effect: Interpolate at the edges, for instance with a filter matrix

0.1	0.1	0.1
0.1	0.2	0.1
0.1	0.1	0.1

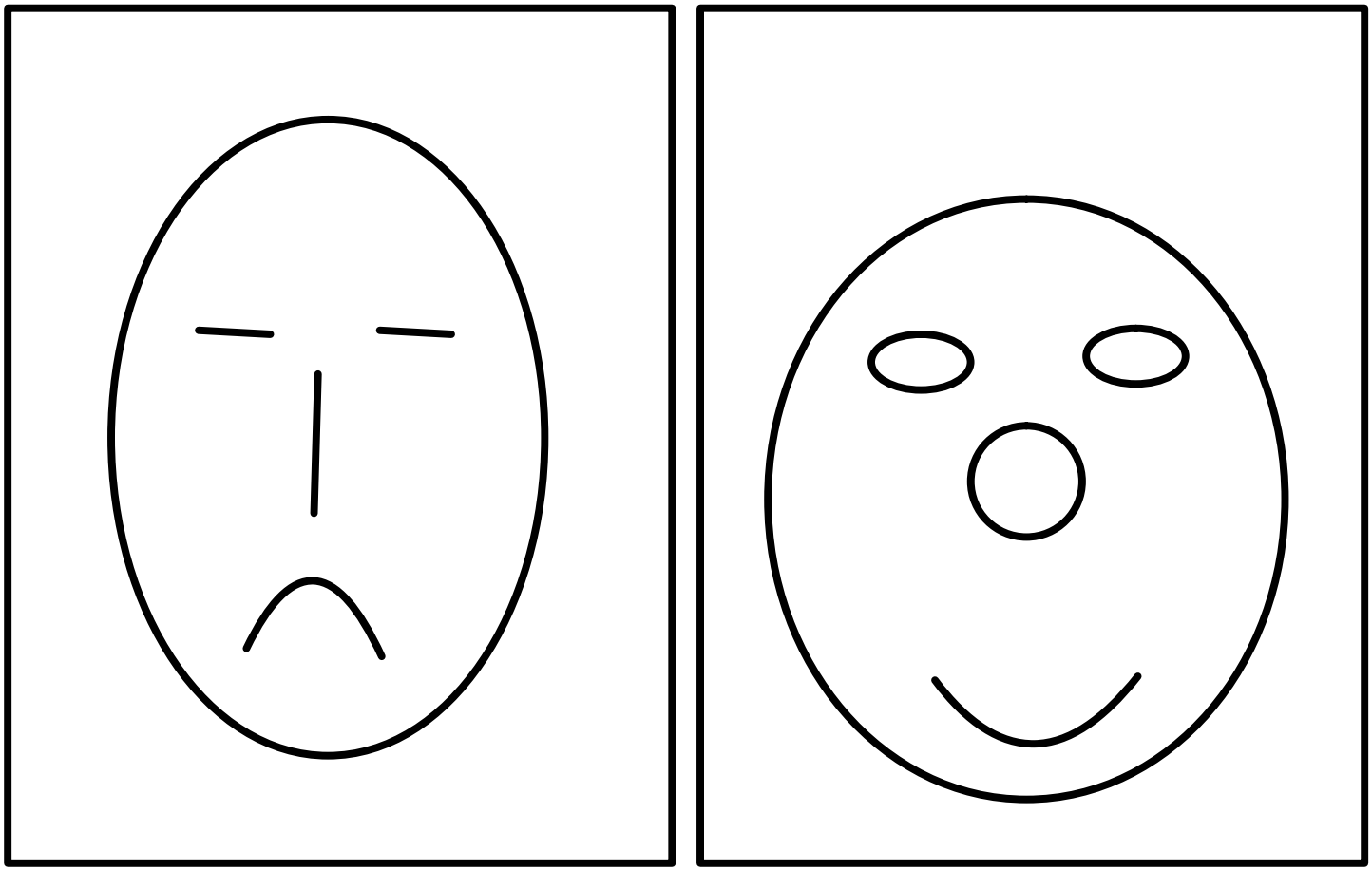
Colour interpolation

Interpolated colour for pixel p_0:

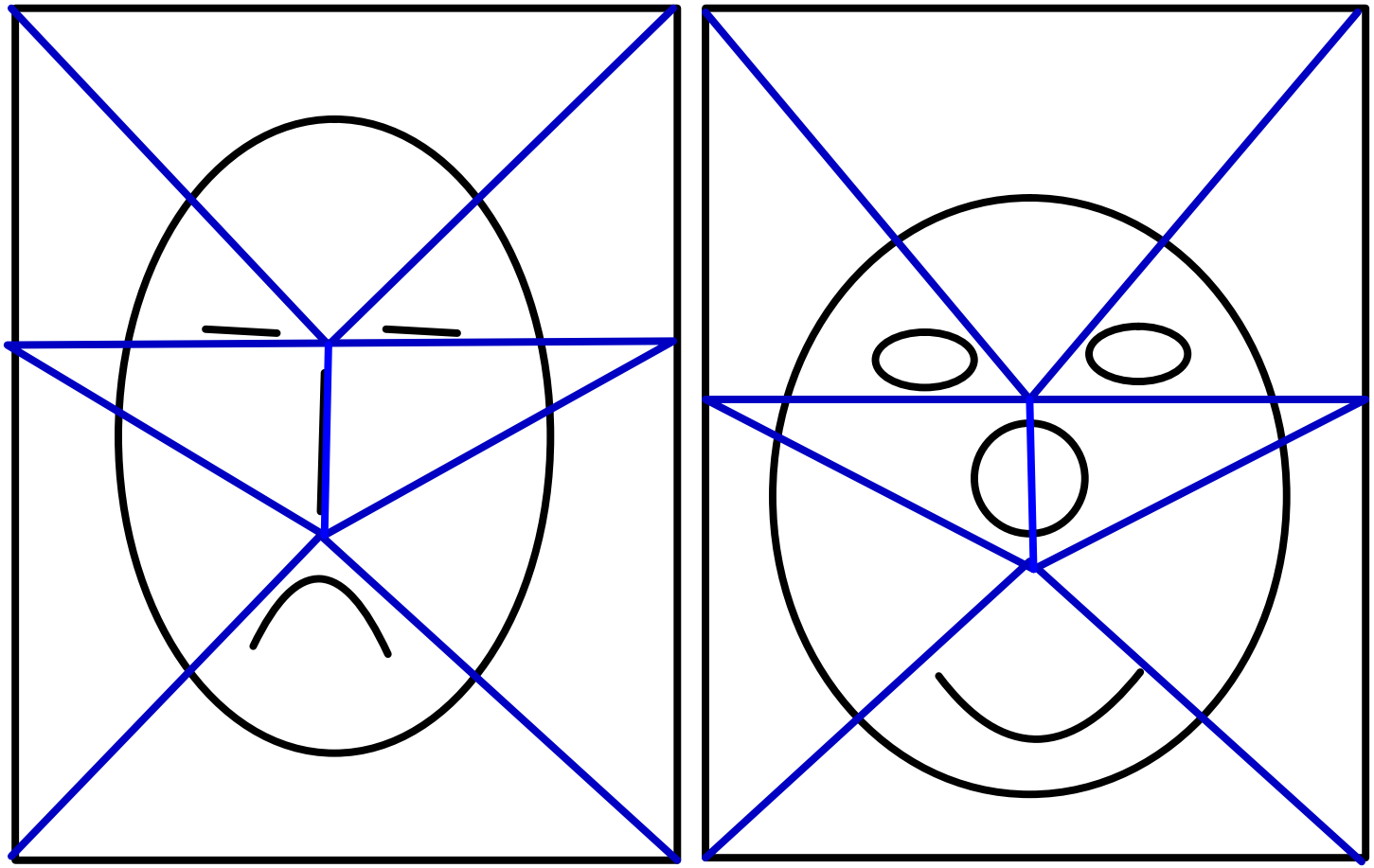
$$(r, g, b) = 0.2 \cdot (r_{p_0}, g_{p_0}, b_{p_0}) + \sum_{p\text{: neighbour of } p_0} 0.1 \cdot (r_p, g_p, b_p)$$

- Instead of a 3×3 filter matrix larger matrices can be used.
- Dynamic filter matrix: At the edge, the above filter matrix is used.
  With increasing distance to the edge the value in the centre of the matrix tends to 1,all other values to 0.

Shape and colour interpolation



Shape and colour interpolation



Shape and colour interpolation

- Convex combinations of the vertices of corresponding triangles are computed to define triangulations for interpolated images.
- Every pixel $\mathbf{q}$ has a unique representation for each triangle in the form

$$\mathbf{q} = \alpha_1 \cdot \mathbf{p}_1 + \alpha_2 \cdot \mathbf{p}_2 + \alpha_3 \cdot \mathbf{p}_3$$

where

$$\alpha_1 + \alpha_2 + \alpha_3 = 1.$$

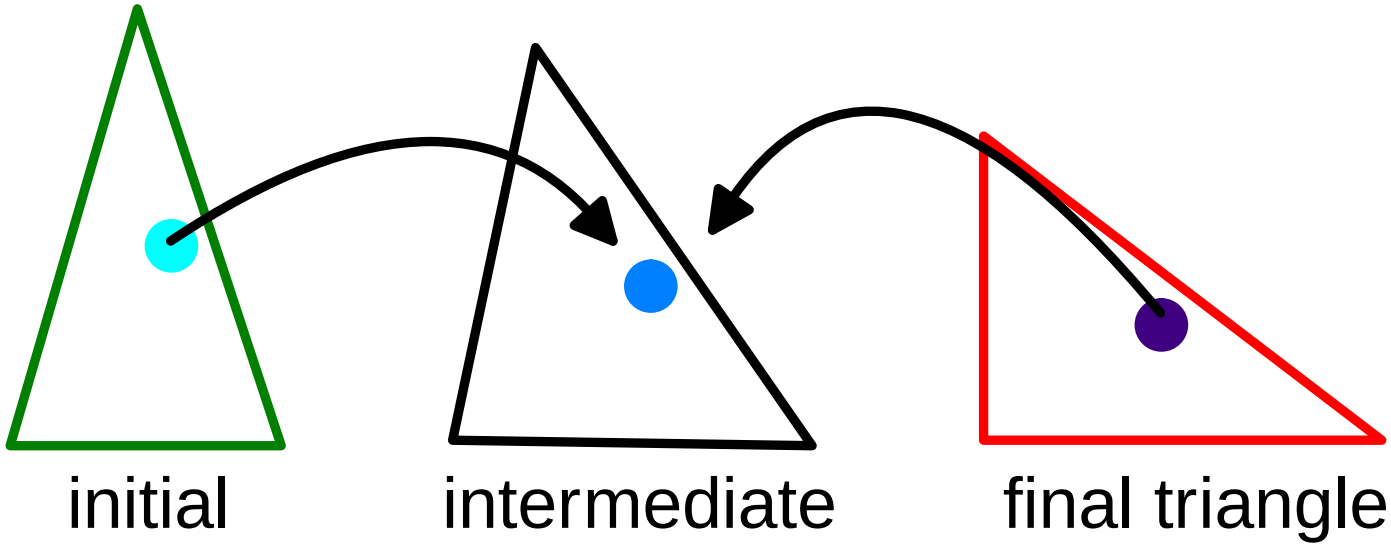
Shape and colour interpolation

- $\mathbf{q}$ lies inside the triangle $\mathbf{p}_1, \mathbf{p}_2, \mathbf{p}_3$ if and only if $\mathbf{q}$ can represented as a convex combination of the vertices, i.e.

$$\alpha_1, \alpha_2, \alpha_3 \geq 0.$$

Shape and colour interpolation

- In each triangle of an intermediate image, the grey- or colour value of a pixel is obtained as the corresponding convex combination of the corresponding two pixels in the initial and final image.



Colour interpolation in Java 2D

Reading the colour of a pixel (x, y) in a `BufferedImage bi`:

```
int rgbValue = bi.getRGB(x,y);
Color pixelColour = new Color(rgbValue);
```

Compute the RGB-values:

```
int red   = pixelColour.getRed();
int green = pixelColour.getGreen();
int blue  = pixelColour.getBlue();
```

Colour interpolation in Java 2D

Compute an interpolated colour with RGB-values (`rMix, gMix, bMix`).

Assign the interpolated colour to the pixel (x, y) in the `BufferedImage mixedBi`:

```
Color mixedColour =
                new Color(rMix, gMix, bMix);
mixedBi.setRGB(x,y,pixelColour.getRGB());
```

(see `MorphingCandS.java` and `TriangulatedImage.java`)