Grafiken programmieren mit Processing

Autor: Raimond Reichert

Inhaltsverzeichnis dieses Projekts

Links zu Processing (Englisch)

- Homepage von Processing
- Sprachübersicht: Kurzerklärungen zu allen Befehlen von Processing
- Tutorials: Kurze Anleitungen zu ausgewählten Themen
- Beispiele zu Grundlagen
- Beispiele zu weiterführenden Themen
- openprocessing.org, veröffentlichte Processing-Projekte

Quellen

Casey Reas, Ben Fry, John Maeda (2007). Processing: A Programming Handbook for Visual Designers and Artists. The MIT Press. Anschauliche Einführung in die Programmieren anhand von Processing. Richtet sich an Leser, die visuelle Darstellungen mit Hilfe von Programmen erstellen möchten.

Processing mit Eclipse programmieren

Processing ist primär dafür gedacht, innerhalb der Processing-Entwicklungsumgebung verwendet zu werden. Processing ist dann eine eigenständige Programmiersprache, die aber praktisch identisch ist zu Java. Daher kann Processing auch innerhalb von Eclipse verwendet werden. Dieses Eclipse-Projekt soll den Einstieg erleichern. Es basiert auf **Processing Version 1.5.1 (August 2011).**

Programmgerüst

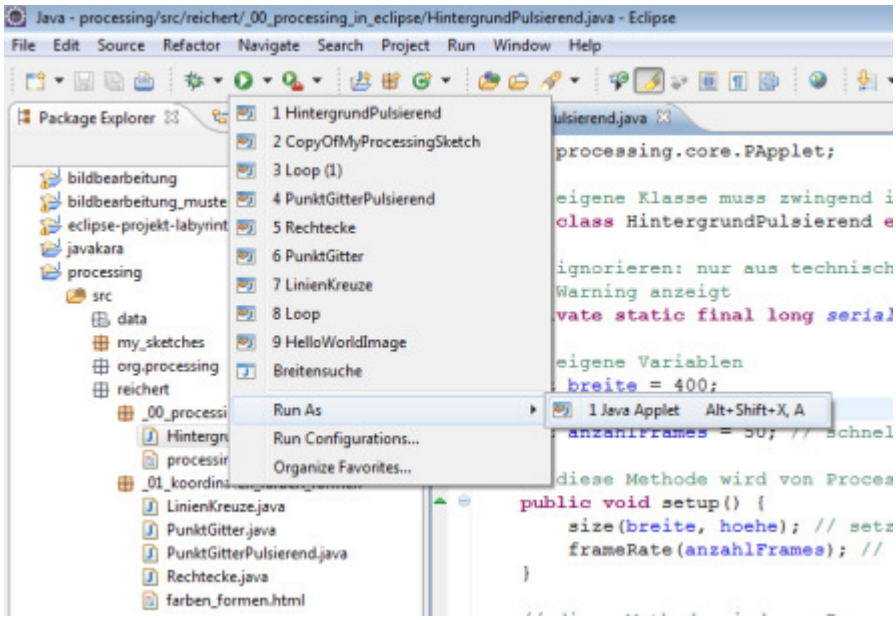
Eingeführte Befehle: size(breite, hoehe), frameRate(wiederholungen), background(r,g,b).

Das Programmgerüst sieht wie folgt aus:

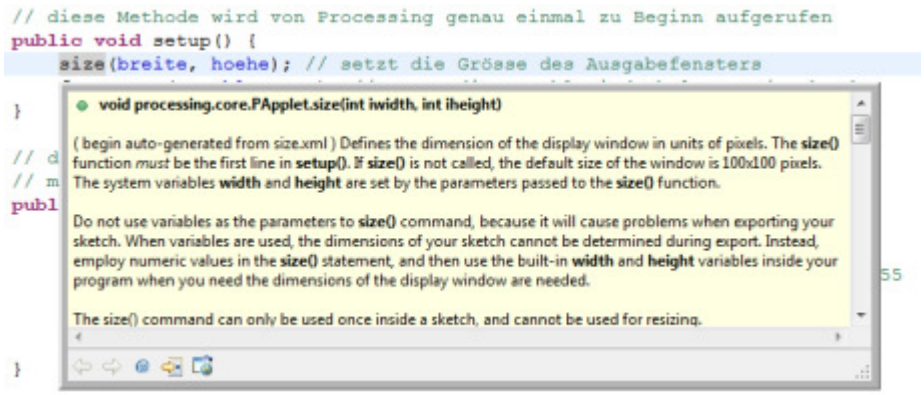
```
// Package-Deklaration muss Verzeichnis-Struktur entsprechen
package reichert._00_processing_in_eclipse;

// PApplet ist die Schnittstelle zu Processing
import processing.core.PApplet;

// die eigene Klasse muss zwingend in einer Datei mit gleichem Namen gespeichert sein
public class HintergrundPulsierend extends PApplet {

	// ignorieren: nur aus technischen Gründen, damit Eclipse keine unnötige
	// Warning anzeigt
	private static final long serialVersionUID = 1L;

	// eigene Variablen
	int breite = 400;
	int hoehe = 300;
	int anzahlFrames = 50; // schneller als normal

	// diese Methode wird von Processing genau einmal zu Beginn aufgerufen
	public void setup() {
		size(breite, hoehe); // setzt die Grösse des Ausgabefensters
		frameRate(anzahlFrames); // setzt die Anzahl Wiederholungen / Sekunde
	}

	// diese Methode wird von Processing pro Sekunde so oft aufgerufen, wie
	// mit frameRate(rate) angegeben
	public void draw() {
		// frameCount gibt an, zu wievielten Mal draw() aufgerufen wird.
		// die Anteile rot, grün, blau werden unterschiedlich schnell erhöht;
		// mittels % wird sichergestellt, dass die Werte immer im Bereich 0..255
		// bleiben.
		background(frameCount % 256, (frameCount * 2) % 256,
				(frameCount * 4) % 256); // setze Hintergrundsfarbe
	}

}
```

Verwendung von Eclipse

Ein Processing-Programm ist technisch ein Applet und wird daher via *Run* > *Run as* > *Java Applet* gestartet:



Innerhalb des Programmeditors können die Beschreibungen der einzelnen Methoden von Processing angezeigt werden, indem auf dem gewünschten Methoden-Name die Taste*F2* gedrückt wird:



Da in diesem Eclipse-Projekt zudem der Quellcode hinterlegt ist (Verzeichnis lib/src), kann der Quellcode jeder Processing-Methode mit der Taste *F3* (oder Rechtsklick auf Befehl > Open Declaration) angezeigt werden.

Koordinaten, Farben und Formen

Koordinatensystem

Eingeführte Befehle und Variablen: size(breite, hoehe); width, height.

Das Koordinatensystem beim Zeichnen mit dem Computer hat den Nullpunkt, die Koordiante (0,0), nicht in der Mitte des Bildschirmes, sondern links oben:

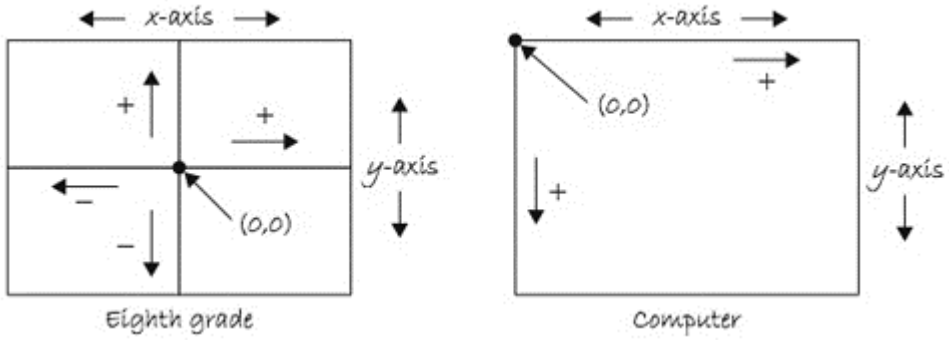


Bild von Coordinate System and Shapes, processing.org

Mit dem Befehl **size(breite, hoehe)** wird die initiale Grösse des Ausgabefensters von Processing gesetzt. Initiale Grösse, weil der Benutzer die Grösse des Fenster selber verändern kann. Die aktuelle **Grösse des Fensters** speichert Processing in den **Variablen width (Breite) und height (Höhe).**

Siehe auch Coordinate System and Shapes (processing.org).

Farben

Eingeführte Befehle: background(r,g,b), stroke(r,g,b), noStroke(), fill(r,g,b), noFill().

Farben werden in sehr vielen Programmen wie Adobe Photoshop, Microsoft Powerpoint u.v.m. häufig als Mischung von Rot-, Grün- und Blau-Bestandteilen angegeben (RGB). Jeder Farbwert liegt dabei im Bereich 0..255. Das ergibt 2^8 * 2^8 * 2^8 = 2^24 = 16'777'216 mögliche Farben. Grauwerte haben jeweils den gleichen Anteil rot, grün und blau, so dass 0..255 verschiedene Grauwerte (inkl. schwarz und weiss) dargestellt werden können. Siehe auch RGB-Farbraum (Wikipedia).

Farben können auch in Processing in der sogenannten RGB-Darstellung angegeben werden (siehe auch Color, processing.org). In Processing können folgende Farben gesetzt werden:

- Die **Hintergrundsfarbe** der gesamten Zeichenfläche kann mit background(r,g,b) gesetzt werden.

- Die **Vordergrundfarbe** zum Beispiel für die Umrandung eines Rechtecks kann mit stroke(r,g,b) gesetzt werden. Optional kann als vierter Parameter die Transparenz angegeben werden: stroke(r,b,g,transparenz). Soll keine Vordergrundsfarbe verwendet werden, kann noStroke() verwendet werden.
- Die **Füllfarbe** zum Beispiel für den Inhalt eines Rechtecks kann mit fill(r,g,b) gesetzt werden. Optional kann als vierter Parameter die Transparenz angegeben werden: fill(r,b,g,transparenz). Soll keine Füllfarbe verwendet werden, kann noFill() verwendet werden.

Ein Online-Program, um Farben zu mischen, finden Sie zum Beispiel unter http://www.kurztutorial.info/programme/farbenrechner/cmyk-rgb.htm.

2d-Grundformen

Eingeführte Befehle: point(x,y), line(x1,y1,x2,y2), rect(x,y,breite,hoehe), ellipse(x,y,breite,hoehe), triangle(x1,y1,x2,y2,x3,y3); strokeWeight(gewicht), smooth(), noSmooth().

Processing unterstützt die typischen Grundformen der zweidimensionalen Darstellung:

- Ein **Punkt** wird mit point(x,y) dargestellt.
- Eine **Strecke** wird mit line(x1,y1,x2,y2) dargestellt.
- Ein **Rechteck** wird mit rect(x,y,breite,hoehe) dargestellt.
- Eine **Ellipse** wird mit ellipse(x,y,breite,hoehe) dargestellt.
- Ein **Dreieck** wird mit triangle(x1,y1,x2,y2,x3,y3) dargestellt.

Als Linienfarbe wird die zuletzt mit stroke(r,g,b) gesetzte Farbe verwendet. Als Füllfarbe wird die zuletzt mit fill(r,g,b) gesetzte Farbe verwendet.

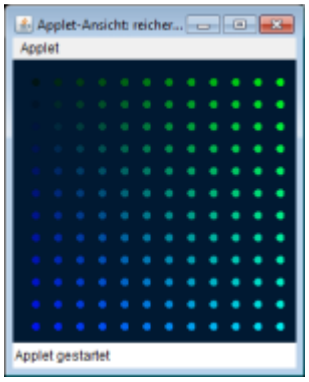
Die **Art der Darstellung** kann zudem über folgende Methoden beeinflusst werden:

- Die **Linienstärke** wird mit strokeWeight(breite) angegeben.
- Die **Kantenglättung** (Anti-Aliasing) kann mit smooth() aktiviert und mit noSmooth() deaktiviert werden.
- Die **Darstellung der Ecken** wird mit strokeJoin(MODE) festgelegt, wobei die Werte MITER, BEVEL, ROUND möglich sind.
- Die **Darstellung von Linienenden** wird mit strokeCap(MODE) festgelegt, wobei die Werte SQUARE, PROJECT, ROUND möglich sind.

Beispielprogramme

Beispielprogramm: Punktgitter

Das Programm zeichnet ein Gitter aus Punkten:

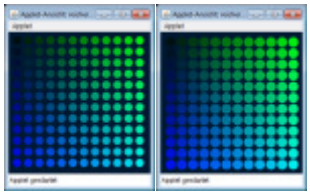


Die Punkte werden dabei eingefärbt in Abhängigkeit ihrer (x,y)-Koordinate:

```
for (int x = ABSTAND; x < BREITE; x = x + ABSTAND) {
        for (int y = ABSTAND; y < HOEHE; y = y + ABSTAND) {
                stroke(0, x, y);
                point(x, y);
        }
}
```

Source Code: PunktGitter.java

Eine Variation des Programms verändert fortlaufend die Grösse der Punkte, so dass der Eindruckt entsteht, die Punkte würden pulsieren:



Die Grösse der Punkte wird als Funktion des frameCount berechnet. frameCount gibt an, zu wievielten Mal draw() aufgerufen wird:

```
// punktStaerke wird die Werte 0, 1, ..., 2*ABSTAND, 0, 1, ...
// 2*ABSTAND, 0, 1, ... annehmen (% = Rest ganzzahliger Division)
int punktStaerke = frameCount % (2 * ABSTAND);
strokeWeight(punktStaerke);
```

Wie genau der Wert der Variable punktStaerke in Abhängigkeit der Zeit (Variable frameRate) berechnet wird, ist der Fantasie überlassen. Das folgende Beispiel verwendet trigonometrische Funktionen, um einen zyklisches Zu- und Abnehmen der Punktstärke zu erzeugen:

```
punktStaerke = (int) (ABSTAND + sin(radians(frameCount)) * ABSTAND);
```

Source Code: PunktGitterPulsierend.java

Beispielprogramm: Linien

Das Programm zeichnet ein Reihe von Kreuzen in X-Form:

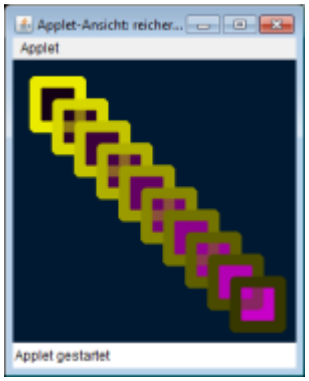
Die doppelt verschachtelte Schleife lässt die Variable x nicht grösser werden als die Variable y; dadurch entsteht die Dreiecksstruktur:

```
for (int y = 1; y < HOEHE - ABSTAND / 2; y = y + ABSTAND) {
        for (int x = ABSTAND / 2; x < y; x = x + ABSTAND) {
                line(x, y, x + ABSTAND / 2, y + ABSTAND / 2);
                line(x + ABSTAND / 2, y, x, y + ABSTAND / 2);
        }
}
```

Source Code: LinienKreuze.java

Beispielprogramm: Rechtecke

Das Programm zeichnet ein Serie von Rechtecken:



Die Rechtecke werden dabei abwechselnd halb-transparent gezeichnet, und ihre Füllfarbe sowie die Rahmenfarbe von einer Zählvariable abhängig gemacht:

```
// mache die Fläche der Rechtecke abwechselnd transparent
if (transparent) {
        fill(i, 0, i, 125); // 125 => mittlere Transparenz
```

```
} else {
        fill(i, 0, i);
}
transparent = !transparent;

stroke(255 - i, 255 - i, 0);
rect(i, i, SEITEN_LAENGE, SEITEN_LAENGE);
```

Source Code: LinienKreuze.java

Aufgaben

Aufgabe: Zufallsregen

Lassen Sie es mit Hilfe des Zufalls farbige Punkte regnen:

Processing stellt die Methode random(maximalWert) zur Verfügung, um Zufallszahlen zu berechnen. So kann eine zufällige X-Koordinate innerhalb der aktuellen Fensterbreite wie folgt erzeugt werden:

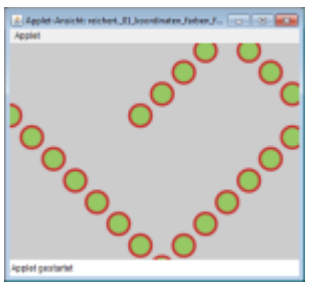
```
float x = random(width);
```

Source Code-Gerüst für Ihre Lösung: AufgabePunkteRegen.java

Variation: Sie können auch Ellipsen oder Rechtecke regnen lassen, gefüllt, nicht-gefüllt, transparent, nicht-transparent, die Farben über die Zeit blasser werden lassen – Ihrer Phantasie setzt Processing nur wenig Grenzen!

Aufgabe: Springender Ball

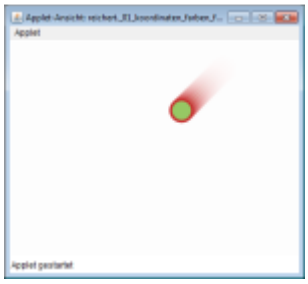
Schreiben Sie ein Programm, das einen Ball durch das Programmfenster springen lässt:



Für Ihr Programm müssen Sie dem Ball eine Geschwindigkeit in X-Richtung und in Y-Richtung geben. Bei jedem Aufruf von draw() aktualisieren Sie die aktuelle Position (x,y) des Balls basierend auf der aktuellen Geschwindigkeit. Dabei müssen Sie berücksichtigen, dass der Ball das Programmfenster mit Breite width und height (Processing-Variablen) nicht verlassen darf. Wenn der Ball also auf einen Fensterrand trifft, müssen Sie die Geschwindigkeit in X-Richtung und in Y-Richtung anpassen.

Source Code-Gerüst für Ihre Lösung: AufgabeSpringBall.java

Sie können Ihr Programm beliebig variieren. Sie können zum Beispiel bei draw() jedes Mal den Hintergrund mit background setzen; so sieht man immer nur den Ball an der aktuellen Position. Sie können aber auch die Spur stehen lassen, indem Sie den Hintergrund nicht jedes Mal setzen. Sie können den Hintergrund statt mit background auch mit rect mit einem halbtransparenten Rechteck zeichnen. So verblassen die Bälle der letzten Aufrufe von draw immer mehr, und es entsteht ein Schweif-Effekt:



Aufgabe: Geometrische Figuren

Schreiben Sie ein Programm, das irgendwelche geomtrischen Figuren zeichnet: Konzentrische Kreise (Kreise mit gleichem Mittelpunkt, aber grösser werdendem Radius), vielleicht mit Farbverlauf, Gitternetze aus Linien, gestapelte Rechtecke, … irgendein Programm, das die oben eingeführten Grundformen verwendet und ansprechend aussieht – und vielleicht animiert darstellt!

Bildbearbeitung

Bilder laden und anzeigen

Eingeführte Befehle: loadImage(dateiName), image(bild,x,y,breite,hoehe), tint(r,g,b,transparenz). Klasse: PImage.

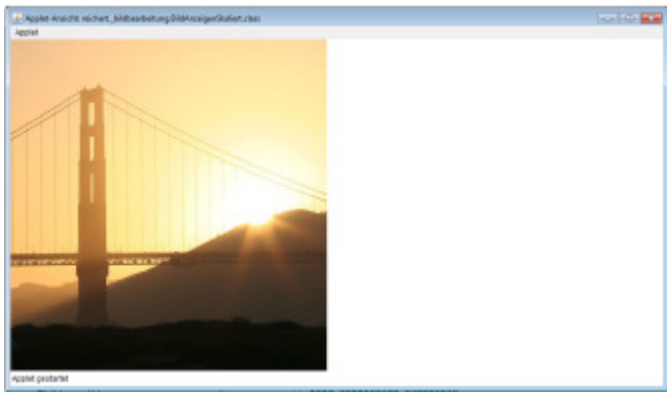
Das **Einlesen eines Bildes** übernimmt der Befehl PImage meinBild = loadImage(dateiName). Die Bilddatei muss dabei im Ordner src/data liegen. Unterstützte Bildformate sind GIF, JPEG, PNG.)

Die **Anzeige eines Bildes** geschieht mit dem Befehl image(meinBild,x,y,breite,hoehe). Das Bild wird dabei auf die angegebene Breite und Hoehe skaliert.

Der **Farbton** eines Bildes wird mit tint(r,g,b,transparenz) gesetzt und mit noTint() wieder zurückgesetzt.

Beispielprogramm: Bild skaliert anzeigen

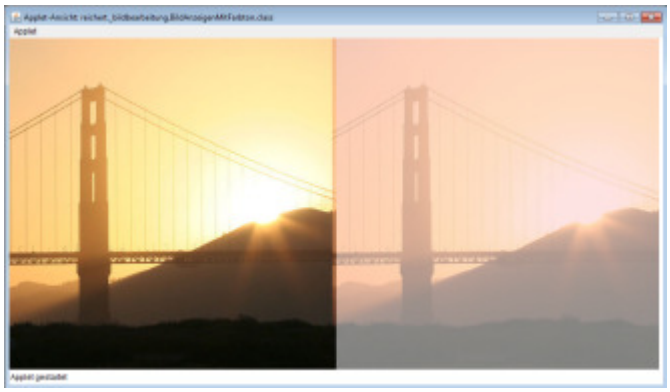
Das Programm skaliert ein Bild horizontal bis zu maximal doppelter Breite:



Source Code: BildAnzeigenSkaliert.java

Beispielprogramm: Eingefärbtes Bild anzeigen

Das Programm zeigt Originalbild und eine rötlich eingefärbte, halbtransparente Version des Bildes:



Source Code: BildAnzeigenMitFarbton.java

Bildbearbeitung: Struktur- und Farbveränderungen

Eingeführte Befehle: new PImage(breite, hoehe), loadPixels(), updatePixels(), red(farbe), green(farbe), blue(farbe), color(rot,gruen,blau)

Um ein Bild zu verändern, muss man typischerweise Pixel für Pixel neu berechnen. Processing erlaubt es, mit loadPixels() **alle Pixels eines Bildes** in einen Array von Integer-Zahlen zu laden. Jede Zahl repräsentiert die Farbe eines Pixels. Bildlich dargestellt ist der Zusammenhang zwischen dargestelltem Bild und dem Pixel-Array wie folgt:

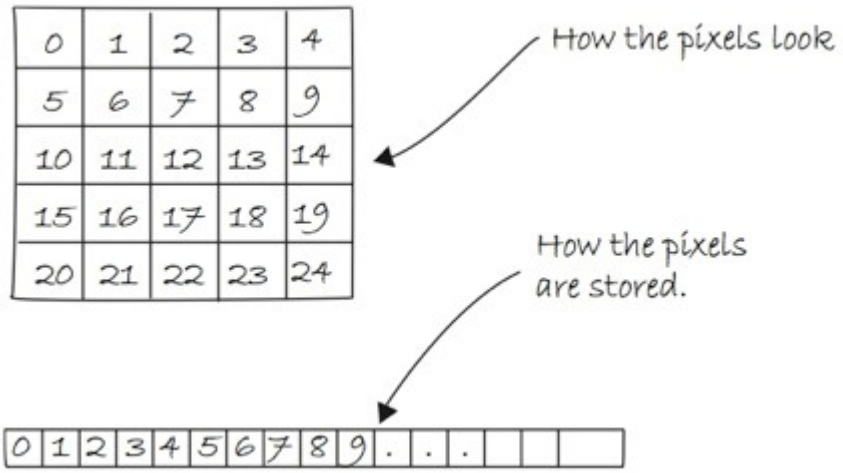


Bild von Images and Pixels, processing.org

Häufig möchte man die Pixels eines Bildes in Abhängigkeit von (x,y)-Koordinaten der Pixel verändern. Dazu muss man aus einer (x,y)-Koordinate den Array-Index berechnen. Grafisch sieht das wie folgt aus:

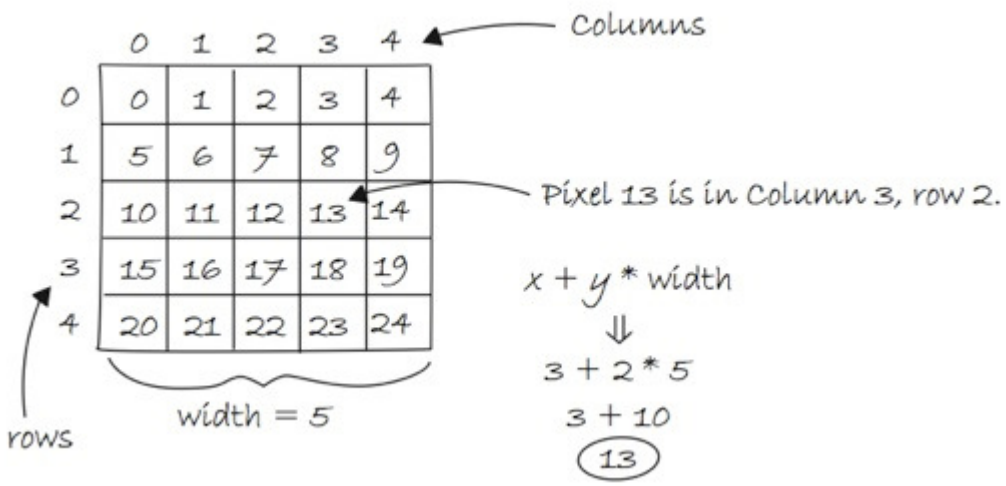


Bild von Images and Pixels, processing.org

Konkret gibt die folgende, selbstgeschriebene Methode getColor die Farbe an Koordinate (x,y) im Bild image zurück:

```
int getColor(PImage image, int x, int y) {
        return image.pixels[y * image.width + x];
}
```

Processing bietet die Methoden red(farbe), green(farbe) und blue(farbe) an, um **Rot-, Grün- und Blau-Anteil** von Farben zu extrahieren, so dass man sich nicht selbst um entsprechende Bit-Manipulationen kümmern muss. Der Parameter farbe ist dabei ein Integer, der eine Farbe in 32 Bit-Darstellung repräsentiert (3x8 Bit für die Farben, 8 Bit für Transparenz). Um umgekehrt aus Rot-, Grün- und Blau-Anteil einen **Integer zu erzeugen, der diese Farbe repräsentiert**, kann die Methode color(rot,gruen,blau) verwendet werden.

Beispielprogramm: Farbe rot entfernen

Betrachten wir ein Programm, das die Farbe rot entfernt:

Die Setup-Methode lädt zuerst das Original-Bild und erstellt dann ein leeres (schwarzes) Bild gleicher Grösse. Anschliessend wird für jeden Pixel im bearbeiteten Bild berechnet, indem als Rotanteil 0 und als Grün- und Blau-Anteil die Werte aus dem Originalbild verwendet werden:

```
original = loadImage("goldengate.jpg");
original.loadPixels(); // initialisiere original.pixels

size(original.width * 2, original.height);

// erstelle neues, leeres Bild
bearbeitet = new PImage(original.width, original.height);
bearbeitet.loadPixels(); // initialisiere bearbeitet.pixels
// bearbeitet.pixels enthält original.width * original.height
// schwarze Pixel

for (int i = 0; i < original.pixels.length; i++) {
        // Grün- und Blauanteil des i-ten Pixels ermitteln
        float gruen = green(original.pixels[i]);
        float blau = blue(original.pixels[i]);

        // neue Farbe berechnen mit Rotanteil=0,
        // Grün- und Blauanteil wie im Originalbild
        bearbeitet.pixels[i] = color(0, gruen, blau);
}
bearbeitet.updatePixels(); // aktualisiere bearbeitet.pixels
```

Source Code: BildbearbeitungRotEntfernen.java

Source Code zu Variation davon, mit animierter, zeilenweiser Rot-Entfernung: BildbearbeitungRotEntfernenAnimiert.java

Wenn Sie mit Farbwerten rechnen, können Ihnen folgende Hinweise hilfreich sein.

Farbwerte sind ganzzahlig. Wenn Sie nun zum Beispiel den Rotanteil eines jeden Pixels auf 20% des Orginalwertes setzen möchten, können Sie das wie folgt berechnen:

```
int neuerRotWert = (int) (red(originalFarbe) * 0.2);
```

Die Multiplikation von red(originalFarbe) mit 0.2 = 20/100 liefert als Resultat eine sog. Fliesskommazahl. War der Rotwert des Pixels zum Beispiel 87, wäre das Resultat 17.4. Aber für den neuen Rotwert braucht es wieder eine Ganzahl. Die Angabe von (int) nach der Zuweisung mit = sorgt dafür, dass die Fliesskommazahl in eine Ganzzahl umgewandelt wird. In diesem Beispiel könnten Sie alternativ auch schreiben:

```
int neuerRotWert = red(originalFarbe) * 20 / 100;
```

So wird eine ganzzahlige Multiplikation und Division durchgeführt. Das Resultat ist in beiden Formeln dasselbe.

Farbwerte sind ganze Zahlen im Bereich 0..255. Vielleicht verwenden Sie eine Berechnung, die einen Wert ausserhalb dieser Grenzen liefert. Sie könnten mit if-Anweisungen sicherstellen, dass der neue Wert nie unter Null oder über Null liegt. Angenommen, wir möchten den Rotanteil um 45% erhöhen, dann könnte ja der neue Rotwert über 255 sein:

```
int neuerRotWert = red(originalFarbe) * 145 / 100;
if (neuerRotWert > 255)  {
        neuerRotWert = 255;
}
```

Kürzer lässt sich das schreiben, wenn man eine Methode verwendet, die das Minimum zweier Werte zurückgibt:

```
int neuerRotWert = min(255, red(originalFarbe) * 145 / 100);
```

Analog kann auch max(0, irgendEineBerechnungFuerNeuenFarbwert) verwendet werden, um sicherzustellen, dass nie ein negativer Farbwert berechnet wird.

Beispielprogramm: Bild vertikal spiegeln

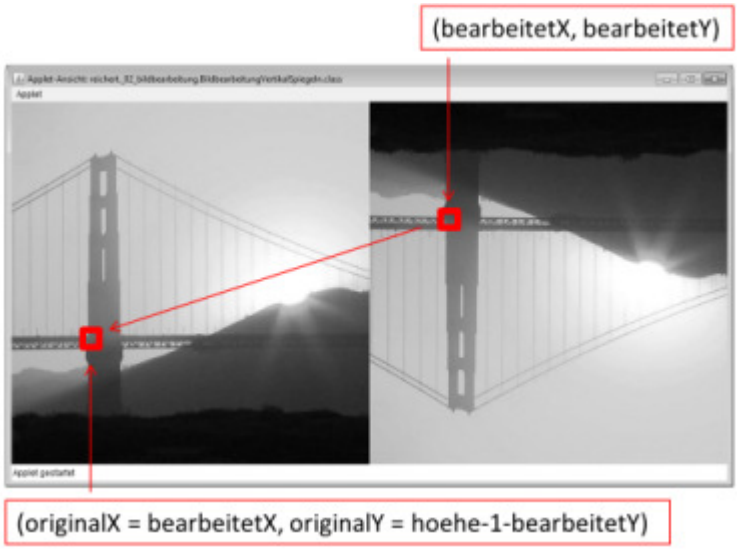
Betrachten wir nun ein Programm, das Bilder vertikal spiegelt:

Die Setup-Methode lädt zuerst das Original-Bild und erstellt dann ein leeres (schwarzes) Bild gleicher Grösse. Anschliessend wird das Bild zeilenweise bearbeitet (y-Richtung), und innerhalb jeder Zeile spaltenweise (x-Richtung). Für jeden Pixel im neuen, bearbeiteten Bild an der Koordinate (bearbeitetX, bearbeitetY) wird die Farbe bestimmt, indem die Farbe vom entsprechend gespiegelten Bildpunkt im Originalbild gelesen wird:

```
original = loadImage("goldengate.jpg");
original.loadPixels();

bearbeitet = new PImage(original.width, original.height);
bearbeitet.loadPixels();

for (int bearbeitetY = 0; bearbeitetY < bearbeitet.height; bearbeitetY++) {
        for (int bearbeitetX = 0; bearbeitetX < bearbeitet.width;
bearbeitetX++) {
                int originalX = bearbeitetX;
                int originalY = bearbeitet.height - 1 - bearbeitetY;
                int originalFarbe = getColor(original, originalX,
originalY);

                setColor(bearbeitet, bearbeitetX, bearbeitetY,
originalFarbe);
        }
```

```
}
bearbeitet.updatePixels();
size(original.width * 2, original.height);
```

Die folgende Grafik soll die Umrechnung der Koordinaten des Zielbildes in Koordinaten des Originalbildes veranschaulichen:



Die Struktur des obigen Programmes ist die **Struktur vieler Bildbearbeitungs-Algorithmen**:

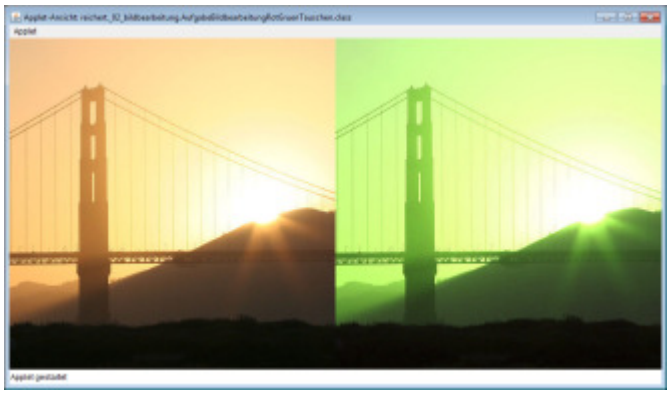
- Pixel für Pixel durch das neue Bild gehen (damit auch für alle Pixel eine Farbe gesetzt wird),
- und dann berechnen, von welcher Koordinate im Originalbild die Farbe gelesen werden soll.

Diese Koordinaten-Umrechnung kann beliebig komplex sein: So entstehen Bilder, die verzerrt, gedreht oder irgendwie transformiert werden. Auch beim Berechnen der neuen Farbe sind der Phantasie keine Grenzen gesetzt: Die neue Farbe kann auf der Farbe eines Pixels im Originalbild basieren, aber sie kann genausogut aus den Farben mehrerer Pixels im Originalbild basieren.

Source Code: BildbearbeitungVertikalSpiegeln.java

Aufgaben

Aufgabe Farbveränderungen: Rot und Grün vertauschen

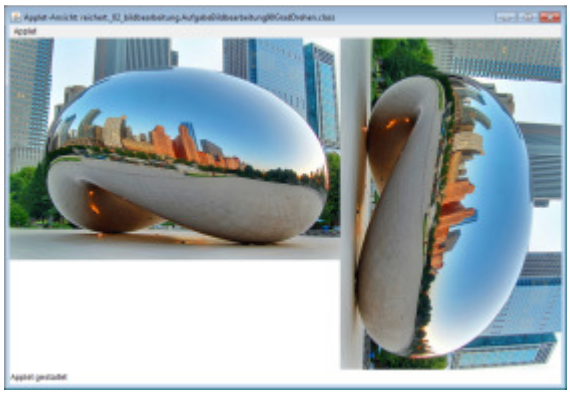


Schreiben Sie ein Programm, das für jeden Bildpunkt die Farben Rot und Grün vertauscht (oder ähnliche Farbveränderungen vornimmt). Sie können dafür die noch leere Klasse AufgabeBildbearbeitungRotGruenTauschen verwenden. Probieren Sie es mit einer einfachen Schleifen über alle Pixel sowie einer doppelt verschachtelten Schleife mit Index-Berechnung aus!

Aufgabe Bildstruktur verändern: Bild horizontal spiegeln

Erstellen Sie eine Klasse, die ein Bild horizontal spiegelt. Sie können dafür die noch leere Klasse AufgabeBildbearbeitungHorizontalSpiegeln verwenden.

Aufgabe Bildstruktur verändern: Bild um 90°, 180° und 270° drehen



Schreiben Sie drei Bildbearbeitungsprogramme:

1. Drehung um 90° nach rechts [1 Punkt]
2. Drehung um 180° (entspricht Punktspiegelung am Zentrum) [3 Punkte]
3. Drehung um 270° nach rechts (entspricht Drehung um 90° nach links) [1 Punkt]

Achten Sie darauf, dass bei Drehungen um 90° und 270° Höhe und Breite beim gedrehten Bild der Breite und Höhe des Originalbildes entsprechen!

Aufgabe Farbveränderungen: Bildfarben in Graustufen

Hintergrundsinformationen zu Graustufen: http://de.wikipedia.org/wiki/Graustufen.

Die Formel für Umrechnung von RGB-Farben in Graustufen: http://de.wikipedia.org/wiki/Grauwert.

Sie können auch mit anderen Formeln für die Grauwerte experimentieren – je nach Bild liefert eine andere Formel anschaulichere Grauwerte.

Aufgabe Farbveränderungen: Farbwerte der einzelnen Pixel neu berechnen

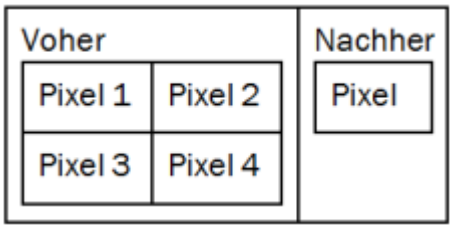
- Zum Beispiel ein Bild in Rot-, Grün-, oder Blaustufen (also nur eine Grundfarbe verwenden statt wie bei den Graustufen alle drei mit gleichem Wert).
- Zum Beispiel den Anteil einer Farbe erhöhen, den Anteil ein anderen Farbe verringern.
- Zum Beispiel Farben vertauschen, Rot zu Grün, Grün zu Blau, Blau zu Rot werden lassen.
- ...

Sie können selbst entscheiden, wie Sie die Farbwerte neu berechnen möchten!

Aufgabe: Bildgrösse halbieren

Schreiben Sie ein Programm, das die Grösse eines Bildes halbiert, horizontal und vertikal.

Abstrakt dargestellt würde ein Bild mit vier Pixel reduziert auf ein Bild mit einem Pixel:



Eine einfache Lösung ist dabei, jede zweite Spalte und jede zweite Zeile wegzulassen. Die Farben von "Pixel" in der Abbildung oben rechts wären dann einfach die Farben von "Pixel 1" in der Abbildung oben links. Allerdings leidet die Bildqualität erheblich darunter.

Aufgabe: Schreiben Sie ein Programm, das dieses einfache Verfahren umsetzt.

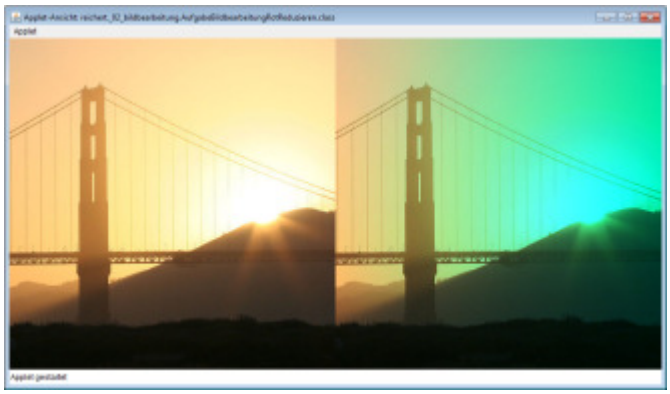
Eine anspruchsvollere Lösung berechnet die Farben von "Pixel"" in der Abbildung oben rechts unter Berücksichtigung von mehreren Pixeln im Orginalbild, zum Beispiel unter Berücksichtigung der vier abgebildeten Pixel oder unter Berücksichtigung aller acht Nachbarpixel berücksichtigen.

Aufgabe: Finden Sie ein Verfahren, das mehrere Pixel des Originalbildes verwendet, um so die Bildqualität zu steigern. Am besten, Sie experimentieren dazu mit verschiedenen Formeln für die Berechnung der Farben der neuen Pixel.

Aufgabe: Farben in Abhängigkeit von Position verändern

Schreiben Sie ein Programm, das die Farbwerte der Pixel in Abhängigkeit Ihrer Position verändert.

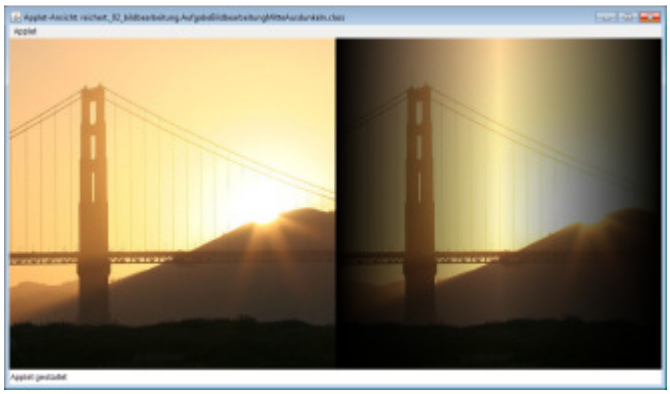
Sie könnten zum Beispiel zum Beispiel einen Farbwert von links nach rechts ausblenden, d.h. den neuen Farbwert von 100% des Originalfarbwertes ganz links kontinuierlich auf 0% ganz rechts reduzieren:



Hilfreich für solche Berechnungen sind folgende Formeln (analoge Formeln können Sie für Y-Richtung verwenden):

```
double deltaX = original.width - x;
// wie weit ist Pixel vom rechten Rand entfernt?
double prozentDeltaX = deltaX / original.width;
// Abstand des Pixel vom rechten Rand in Prozent
```

Sie könnten auch ein Bild von der Mitte her "ausdunkeln":



Hilfreiche Formeln dazu:

```
double deltaX = Math.abs(original.width / 2.0 - x);
// wie weit ist Pixel von der Mitte entfernt?
double deltaXProzent = 1 - deltaX / (original.width / 2);
// wie weit ist Pixel prozentual von der Mitte entfernt?
```

Sie könnten das Bild auch in horizontaler und vertikaler Richtung von der Mitte her ausdunkeln.

Sie könnten auch die Helligkeit eines Bilder in der Mitte am stärksten erhöhen, gegen den Bildrand hin immer weniger.

Sie könnten aber auch nur einen bestimmten Bildausschnitt verändern. Wenn Sie zum Beispiel ein Foto einer Person mit roten Augen nehmen, könnten Sie in den Bereichen der roten Augen den Rotwert reduzieren. Die Koordinaten können Sie mit Hilfe des Rahmenprogramms bestimmen.

Sie können selber wählen, welchen Effekt Sie berechnen wollen!

Interaktionen mit der Maus

Auf Maus reagieren: Variablen

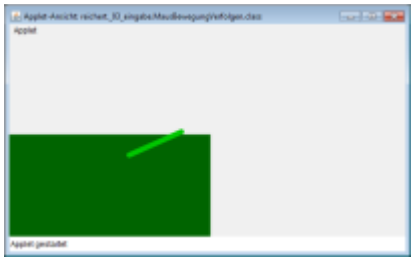
Eingeführte Variablen und Befehle: mouseX, mouseY, pmouseX, pmouseY, mousePressed, mouseButton, map(wert, originalWerteBereichStart, originalWerteBereichEnde, neuerWerteBereichStart, neuerWerteBereichEnde), constrain(wert, werteBereichStart, werteBereichEnde)

Die folgenden Beispiele zeigen, wie direkt in der Methode draw() auf die Maus-Eingaben des Benutzers reagiert werden kann.

Die aktuelle **Position des Mauszeigers ist als Koordinate (mouseX, mouseY)** in den entsprechenden Variablen gespeichert. Diese beiden Variablen werden vor jedem Aufruf von draw() aktualisiert. Die jeweils vorherige Position ist als Koordinate (pmouseX, pmouseY) in den entsprechenden Variablen gespeichert ("p" für "previous").

Beispielprogramm: Mausbewegung mit Linie und Rechteck verfolgen

Das Programm verbindet die vorherige Position der Maus mit der aktuellen Position, und fäbt den Quadranten ein, in dem sich die Maus gerade befindet:



Damit es nicht zu schnell geht, wird die FrameRate auf 10 Bilder / Sekunde runtergesetzt. Zudem wird in draw() jedes Mal die Hintergrundfarbe neu gesetzt, damit die letzte Zeichnung von draw() gelöscht wird.

Source Code: MausBewegungVerfolgen.java

Beispielprogramm: Berechnungen auf Mausposition

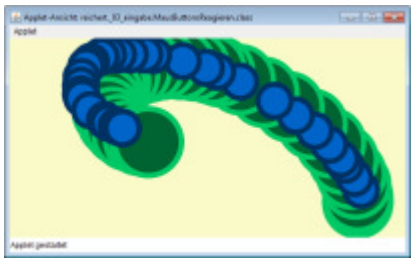
Das Program zeigt zu einem Bild das Originalbild und drei verschiebene Einfärbungen des Bildes, in Abhängigkeit von der Mausposition. Wichtig ist dabei, dass der Farbton im Wertebereich 0..255 liegen muss, auch wenn die Mauskoordinaten im Wertebereich 0..width (Fensterbreite) bzw. 0..height (Fensterhöhe) liegt. Für **Umrechnungen von einem Wertebereich in einen anderen** stellt Processing die **Methode map()** zur Verfügung:

```
map(mouseX, 0, width, 0, 255)
```

Source Code: MausPositionAlsFarbton.java

Beispielprogramm: Auf Maustasten reagieren

Das Programm zeigt, wie die Boole'sche Variable mousePressed anzeigt, ob eine Maustaste gedrückt ist, und wie die int-Variable mouseButton angibt, welche Maustaste gedrückt wurde:



In diesem Beispiel wird der Hintergrund in draw() nicht gelöscht, so dass der Benutzer eine "Zeichnung" machen kann.

Source Code: MausButtonsReagieren.java

Wichtig: Der obige Ansatz, direkt in der Methode draw() auf Mausereignisse zu reagieren, eignet sich nur für einfache Programme, wo nicht unterschieden werden muss zwischen "Maus gedrückt" (und noch nicht wieder losgelassen), "Maus losgelassen" oder "Maus geklickt". Das Programm MausButtonsKlickReagierenzeigt, wie die Methode mouseClicked verwendet werden kann, um nur bei Mausklicks (aber nicht bei gedrückter, bewegter Maus) Kreise zu zeichnen:

Source Code: MausButtonsKlickReagieren.java

Aufgabe: Bild um Mausposition herum grau färben

Schreiben Sie ein Programm, dass die Pixel im Bereich der Mausposition graufärbt. Wenn eine Maustaste gedrückt wird, sollen die Pixel im Rechteck (mouseX-20, mouseY-20)-(mouseX+20, mouseY+20) grau gefärbt werden:

Am besten schreiben Sie zunächst ein Programm, das nur den Pixel an der aktuellen Mausposition grau färbt. Dann erweitern Sie das Programm, so dass es die Pixel im Rechteck um die Mausposition herum grau färbt.

Wichtig ist dabei, dass Sie das **Originalbild unverändert lassen** und die Farbveränderung in einem neuen Bild speichern. Je nach Farbveränderung würden Sie sonst, wenn Sie nur auf einem Bild arbeiten, diese bei jedem Mausklick wieder verändern.

Wichtig ist zudem auch, dass Sie für das Lesen und Schreiben von Pixeln immer sicherstellen, dass die **(x,y)-Koordinaten der Pixel innerhalb des Bildes liegen** – Mauskoordinaten können auch ausserhalb liegen! Eine einfache Möglichkeit ist die **Methode** constrain(), hier an einem Beispiel illustriert:

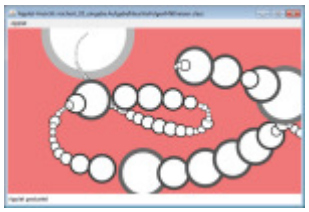
```
int constrainedX = constrain(x, 0, img.width - 1);
int constrainedY = constrain(y, 0, img.height - 1);
```

Die Variable constrainedX wird nie Werte kleiner 0 und nie grösser als img.width-1 werden und somit immer ein gültiger X-Wert im Bild img darstellen.

Source Code-Gerüst für Ihre Lösung: AufgabeBildLokalGraustufen.java

Aufgabe: Mausposition verfolgen in Abhängigkeit von Geschwindigkeit

Schreiben Sie ein Programm, dass die aktuelle Mausposition anzeigt, wobei zum Beispiel die Grösse eines Kreises oder seine Füllfarbe oder Rahmenfarbe oder Strichgrösse von der Mausgeschwindigkeit abhängen:



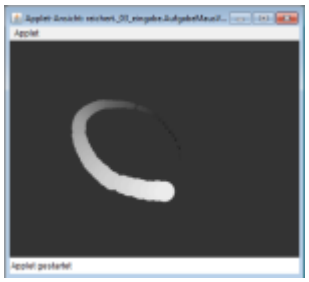
Für die Geschwindigkeit der brauchen Sie die aktuelle Position der Maus (mouseX, mouseY) sowie die vorherige Mausposition (pmouseX, pmouseY). Sie können die "Geschwindigkeit" auf verschiedene Arten berechen, und so verschiedene Effekte erzielen, zum Beispiel:

```
// "Geschwindigkeit" = X-Abstand + Y-Abstand; Funktion abs(wert) liefert
Absolutwert von wert
float speed = abs(mouseX - pmouseX) + abs(mouseY - pmouseY);

// "Geschwindigkeit" = geometrischer Abstand der beiden Mauspositionen;
// sqrt(wert) liefert Quadratwurzeln von wert
float speed = sqrt((mouseX - pmouseX) * (mouseX - pmouseX) + (mouseY -
pmouseY) * (mouseY - pmouseY));
```

Source Code-Gerüst für Ihre Lösung: AufgabeMausVerfolgenMitKreisen.java

Aufgabe: Mausposition verfolgen mit Verblassen der Spur

Schreiben Sie ein Programm, dass eine Spur der Mausbewegung anzeigt, die mit der Zeit verblasst:



Um eine Spur zu zeichnen, müssen Sie eine gewisse Anzahl Mauspositionen speichern. Sie können dazu zum Beispiel für die X- und Y-Positionen jeweils in einem Array speichern. Dann können Sie die Positionen zyklisch in diesen Arrays speichern: Wenn die Arrays "voll" sind, füllen Sie sie wieder von vorne. Das Programmgerüst dazu könnte wie folgt aussehen:

```
int num = 60;
float mx[] = new float[num];
float my[] = new float[num];

public void draw() {
        // Mausposition in Arrays speichern; als Index im Array wird
frameCount
        // verwendet, wobei mit % num jeweils der Rest der Ganzzahldivision
        // durch num (Länge des Arrays) genommen wird, so dass
        // mIndex die Werte 0..num-1, 0..num-1, ... annimmt.
        int mIndex = frameCount % num;
        mx[mIndex] = mouseX;
        my[mIndex] = mouseY;

        // hier die Mauspositionen anzeigen
}
```

Bei der Darstellung der Mauspositionen können Sie zum Beispiel die Grösse der Kreise oder ihre Füllfarben oder Rahmenfarben oder Strichgrössen verändern, damit der Effekt einer verblassenden Spur entsteht.

Source Code-Gerüst für Ihre Lösung: AufgabeMausVerfolgenMitSpur.java