Zukunftsmuster

analog, digital, generativ und interaktiv

Mustergestaltung mit Illustrator, Processing und CSS3

Bachelor-Thesis von Sebastian Welzel

HAWK Hochschule für angewandte Wissenschaft und Kunst
Hildesheim Holzminden Göttingen
Fakultät Gestaltung / Farbdesign
Renatastraße 11
31134 Hildesheim

Sommersemester 2015
Kompetenzfeld Farbdesign

Zukunftsmuster - analog, digital, generativ und interaktiv
Mustergestaltung mit Illustrator, Processing und CSS3
Bachelor-Thesis

Prüfer
Prof. Markus Schlegel
Dipl.-Des. Martin Brandes

Sebastian Welzel
Farbdesign
7. Semester

Inhalt

1. Einführung

Muster haben in der Menschheitsgeschichte eine lange Tradition. Im Laufe der Zeit haben sich in Regionen und Kulturen unterschiedliche Formen und Farben in Mustern etabliert, die sogar Auskunft über die Herkunft eines Musters geben können. So wie sich im Laufe der Zeit Formen und Farben entwickelt haben, sind auch immer wieder neue Werkzeuge und technische Möglichkeiten entstanden und erfunden worden, mit denen Muster erstellt werden konnten und die den Arbeitsprozess vereinfacht haben. Muster wurden reproduzierbar. Seit der Erfindung von Computern mit grafischer Oberfläche sind viele neue Grafik-Programme und andere Möglichkeiten entwickelt worden, die die Erstellung von digitalen und analogen Mustern noch einfacher machen. An dieser Stelle setzt die Thesis zum Thema Zukunftsmuster an.

Viele Farbdesign-Studenten kommen während ihres Studiums lediglich mit dem Vektorgrafik-Programm Adobe Illustrator in Berührung, mit dem statische Muster erstellt werden können.
Programme und einfache Programmiersprachen, mit denen Muster generativ, interaktiv und/oder reaktiv erstellt beziehungsweise automatisch und massenhaft generiert werden können, finden speziell in Farbdesign-Kursen noch keine Beachtung. Aus diesem Grund werden in dieser Thesis Techniken erklärt und Möglichkeiten aufgezeigt, um beispielhaft zu zeigen, wie Mustergestaltung in der Zukunft aussehen kann.

Erklärung zum Aufbau

Die Thesis gliedert sich in einen praktischen und einen theoretischen Teil.
Der vorliegende theoretische Teil behandelt zu Beginn die für die digitale Mustergestaltung notwendigen Grundlagen wie Farbsysteme, Farbkontraste und den allgemeinen Aufbau von Mustern.
Darauf aufbauend werden Programme und Werkzeuge für die Erstellung digitaler Muster in mehreren, voneinander unabhängigen Tutorials vorgestellt und erklärt.
Zum Abschluss wird in den Ergebnissen besonders herausgearbeitet, welche Vor- und Nachteile Processing und CSS3 gegenüber Adobe Illustrator haben, welche gemeinsam nutzbaren Schnittstellen es gibt und was das für die Arbeitsabläufe bei der Zukunftsmuster-Gestaltung bedeutet.

Farbcodierung

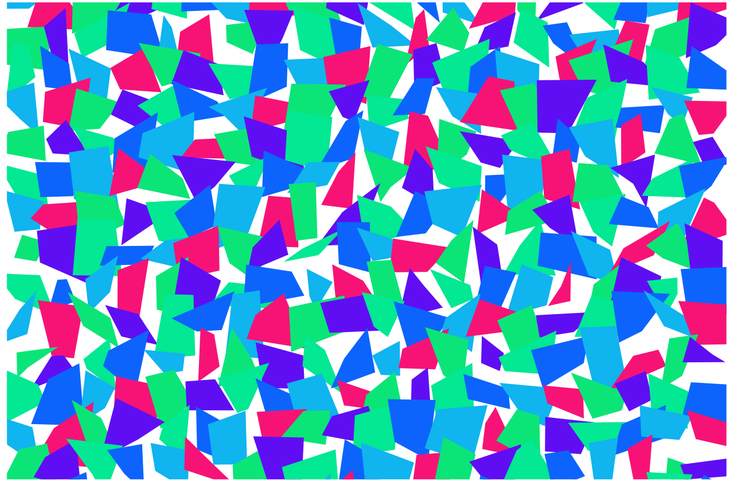
Da die Processing- und CSS3-Tutorials viele Zeilen Programmier-Code enthalten, wird dort eine durchgehende Farbcodierung eingesetzt, die das Lesen und Verstehen erleichtern soll. So werden **Farbwerte** beispielsweise rot hervorgehoben, **Formen** und **Winkelangaben** grün und **Positions-** und **Maßangaben** blau.

Die vorliegende Arbeit ist in einem Zeitraum von acht Wochen entstanden und beinhaltet teils komplexe technische Zusammenhänge, die hier so gut wie möglich auf das Thema Zukunftsmuster sowie die für die Bearbeitung ausgesuchten Programme und Techniken eingegrenzt wurden. Aus diesen Gründen besteht kein Anspruch auf Vollständigkeit.

5. Muster generieren mit Processing

5.1. Einführung

Das folgende Tutorial soll einen kleinen Einblick in die Syntax von Processing geben und zeigen, wie ein einfaches Raster mit Quadraten erstellt werden kann. In einem separaten Teil werden verschiedene Möglichkeiten erläutert, wie in Processing Farben beschrieben und wie mit ihnen gearbeitet werden kann.



Ein mit Processing generiertes unregelmäßiges Muster, ausgegeben als bearbeitbare PDF mit Vektor-Formen.

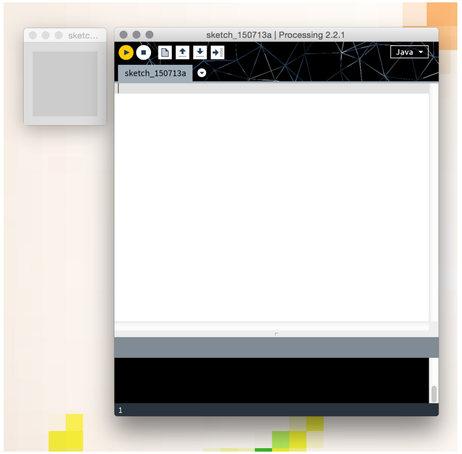
5.2. Processing

„Das Processing-Projekt wurde im Frühjahr 2001 von Ben Fry and Casey Reas mit einer kleinen Gruppe von Helfern initiiert und seitdem stetig weiterentwickelt. Hauptziel von Processing ist, visuell orientierten Menschen einen einfachen Zugang zur Programmierung zu ermöglichen.“[59]
In der nebenstehenden Grafik ist die Entwicklungsumgebung (großes Fenster) von Processing zu sehen. „Im mittleren Bereich, dem Editor, wird der Programmcode eingegeben. Die Toolbar darüber enthält Buttons, um das Programm zu starten und zu stoppen oder um Programme zu laden, zu speichern oder zu exportieren. Der untere Bereich ist die Konsole, in der Nachrichten ausgegeben werden können oder Fehlermeldungen erscheinen.“[60]
Das kleine graue Fenster ist das Programm, das in diesem Fall keinerlei Inhalt hat.

Nützliche Links
Programm-Download:
https://processing.org/download/
Referenz: https://processing.org/reference/

59 Bohnacker et. al., 2009, S. 168.

60 ebenda, S. 169.



Die Arbeitsumgebung von Processing. Rechts die Entwicklungsumgebung, links das Programm.

5.3. Dateistruktur und Syntax

In der **setup()**-Funktion werden Angaben gemacht, die nur einmal beim Start des Programms benötigt werden, hierzu gehört zum Beispiel die Fenster-Größe des Programms, die mit **size()** festgelegt wird (Angaben in Pixeln).

In den meisten Fällen sollen Programme so lange ausgeführt werden, bis sie vom Nutzer beendet werden. Dafür ist in Processing die **draw()**-Funktion gedacht. Alle darin enthaltenen Befehle werden standardmäßig 60 mal pro Sekunde aufgerufen, es sei denn es wird mit **frameRate()** im Setup-Bereich ein anderer Intervall festgelegt. Bei zu großem Rechenaufwand wird der Intervall automatisch verringert.[61]

Kommentare
„Je komplexer und trickreicher ein Programm ist, desto schwerer ist es für andere [...] das Programm zu durchschauen. Ein Programm, das nicht verstanden wird lässt sich nicht mehr modifizieren und erweitern. Kommentare im Programm helfen dabei, den Code verständlich zu halten.“[62]

61 Vgl. Bohnacker et. al., 2009, S. 173

62 Bohnacker et. al., 2009, S. 178

```
void setup() {
    size(500, 500);                 // size( Breite, Höhe );
    smooth();
    frameRate(30);
}

void draw() {
    background(255);                // Hintergrundfarbe weiß

    fill(0);                        // Füllfarbe schwarz
    noStroke();                     // keine Kontur

    rect(50, 50, 100, 100);         // zeichnet ein Rechteck


    noFill();                       // keine Füllfarbe
    stroke(random(0, 200));         // zufällige Konturfarbe in Grauwerten
    strokeWeight(2);                // Konturbreite in Pixeln

    ellipse(mouseX, mouseY, 50, 50);  // zeichnet einen Kreis um die Mauszeigerposition
}
```

Ein Beispielprogramm, in dem ein schwarzes Quadrat und ein Kreis mit Kontur variabel plaziert werden.
Quelle: eigener Code.

Einzeilige Kommentare werden mit zwei Schrägstrichen eingeleitet, der nachfolgende Text wird ignoriert. Mehrzeilige Kommentare beginnen mit /* und Enden mit */.

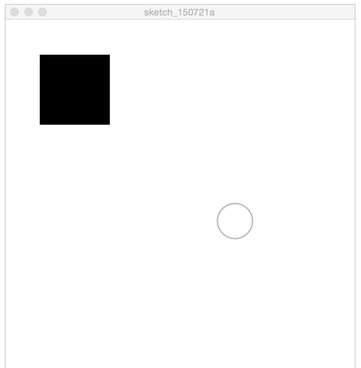
Farben für Objekte

Die Eigenschaften **background()**, **fill()** und **stroke()** legen Farben für den Hintergrund, die Füllfarbe und die Konturfarbe fest. Mehr dazu ab Seite 57.
Die Füll- und Konturfarben gelten für alle nachfolgenden Objekte, bis wieder neue Farben festgelegt und danach Objekte folgen.

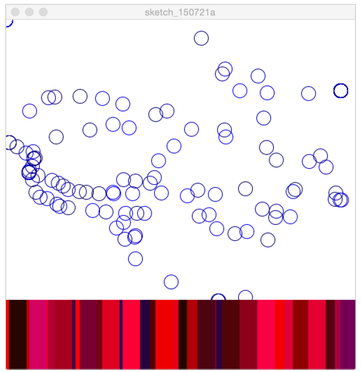
Objekte

Die Funktionen **rect()**, **ellipse()**, **line()**, **point()**, etc. zeichnen Rechtecke, Ellipsen/Kreise, Linien, Punkte, etc.. Wie die Syntaxen für die einzelnen Funktionen lauten steht in der Processing-Referenz[63].

63 URL: https://processing.org/reference/ [21.07.2015].



Ein Beispielprogramm, in dem ein schwarzes Quadrat und ein Kreis mit Kontur variabel plaziert werden.
Quelle: eigener Darstellung.



Ein Beispielprogramm, in dem rote Rechtecke und blaue Kreise auf den Hintergrund gezeichnet werden.

Hintergrundfarbe

Wenn die Hintergrundfarbe innerhalb von draw() festgelegt ist, wird der Hintergrund in jedem Frame neu gesetzt und die Inhalte gelöscht. Steht die Hintergrundfarbe innerhalb des Setups, wird der Hintergrund nur beim Start des Programms gefüllt und alle Objekte, die während der Laufzeit auf dem Hintergrund hinzugefügt werden, bleiben sichtbar.

Variablen

Statt Zahlenwerten können in Processing immer auch selbst definierte Variablen oder Systemvariablen wie beispielsweise **mouseX** und **mouseY** eingesetzt werden, die die horizontalen und vertikalen Mauspositionen einfügen.
Variablen haben den Vorteil, dass sie nur einmal definiert und anschließend überall eingesetzt werden können. Bei Änderungen muss der Wert nur einmal in der Variablendefinition geändert werden und wird überall, wo die Variable Verwendung findet, übernommen.

```
void setup() {
    size(500, 500);                    // size( Breite, Höhe );
    smooth();
    frameRate(10);

    background(255);                   // Hintergrundfarbe im Setup-Bereich
}

void draw() {

    fill(random(0, 255), random(0, 10), random(0, 100));    // zufällige Füllfarbe
    noStroke();                        // keine Kontur

    rect(mouseX, 400, 25, 100);        // zeichnet Rechtecke


    noFill();                          // keine Füllfarbe
    stroke(random(0, 10), random(0, 10), random(100, 255));    // zufällige Konturfarbe
    strokeWeight(1);                   // Konturbreite in Pixeln

    ellipse(mouseX, mouseY, 20, 20);   // zeichnet Kreise um die Mauszeigerposition
}
```

Ein Beispielprogramm, in dem rote Rechtecke und blaue Kreise auf den Hintergrund gezeichnet werden.
Quelle: eigener Code.

5.4. Raster generieren

Im folgenden wird kurz erklärt, wie ein Raster erstellt und mit zufallsfarbigen Rechtecken gefüllt werden kann.

Um ein Objekt, hier ein Rechteck, sowohl vertikal als auch horizontal zu wiederholen, wird für jede Achse die for()-Schleifen-Funktion verwendet. Eine Schleife wird so lange wiederholt, bis die in der Klammer definierte Bedingung erfüllt ist.[64] Schleifen haben zum Beispiel folgende Syntax:[65]

```
for ( Initialisierung; Bedingung; Aktualisierung) {
        auszuführende Funktionen
}
```

Beispiel
Initialisierung: Die Ganzzahl-Variable rasterY wird definiert und der Wert auf 0 gesetzt.
Bedingung: Der Variablen-Wert muss kleiner sein als der Variablen-Wert von rechteckAnzahl, hier: 20.
Aktualisierung: Wenn der Wert von rasterY unter 20 ist wird der Wert um eins erhöht.

64 Vgl. URL: https://processing.org/reference/for.html [22.07.2015]

65 Vgl. ebenda.

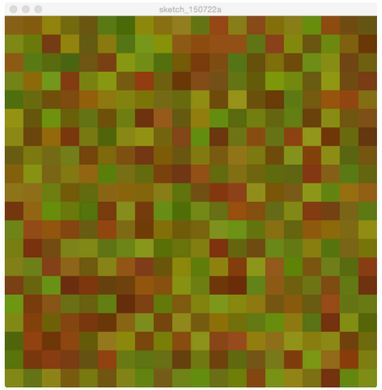
```
int rechteckAnzahl = 20;

void setup(){
    size(600, 600);
    frameRate(5);
}

void draw() {
    colorMode(HSB, 360, 100, 100, 100);  // Erklärung siehe Seite 56
    background(360, 0, 100);             // weißer Hintergrund
    noStroke();

    for (int rasterY=0; rasterY<rechteckAnzahl; rasterY++) {
        for (int rasterX=0; rasterX<rechteckAnzahl; rasterX++) {

            int posX = width/rechteckAnzahl * rasterX;
            int posY = height/rechteckAnzahl * rasterY;

            fill(random(20, 80), random(80, 90), random(40, 60), 100);
            rect(posX, posY, width/rechteckAnzahl, height/rechteckAnzahl);
        }
    }
}
```

Ein Beispielprogramm, in dem Quadrate mit zufälligen Farben in einem Raster angeordnet werden.
Quelle: eigener Code.

Die erste Funktion ist wieder eine Schleife, in der der Schleifen-Vorgang für die X-Achse rasterX wiederholt wird.

Mit den Variablen posX und posY werden schließlich x- und y-Positionen bestimmt. Die Breite bzw. Höhe des Programms wird durch die Anzahl der Rechtecke geteilt und mit dem jeweiligen Wert, den rasterX bzw. rasterY während des jeweiligen Schleifendurchlaufs hat, multipliziert.

Für jeden Schleifendurchlauf wird mit fill() eine neue, zufällige Farbe generiert.
Schlussendlich wird mit rect() ein Quadrat erstellt, mit den in den Variablen posX und posY gespeicherten Koordinaten-Werten und der errechneten Breite und Höhe (Breite und Höhe des Programms werden durch die Anzahl der Quadrate geteilt).



Ein Beispielprogramm, in dem Quadrate mit zufälligen Farben in einem Raster angeordnet werden.

5.5. Farbsysteme und Schreibweisen

Farbsysteme und deren Wertebereiche
In Processing gibt es viele verschiedene Möglichkeiten, Farbwerte zu definieren, jedoch beschränken sich diese auf lediglich zwei Farbsysteme, RGB und HSB. Um ein Farbsystem festzulegen wird die Funktion **colorMode()** verwendet, die sowohl im setup- als auch im draw-Bereich stehen kann und folgende Syntax aufweist:[66]

colorMode(Modus);
colorMode(Modus, max);
colorMode(Modus, max1, max2, max3);
colorMode(Modus, max1, max2, max3, maxA);

Modus: RGB oder HSB.
max: Legt den maximalen Wertebereich fest, in dem sich alle Farbwerte befinden müssen, zum Beispiel 255, 100 oder 1.
max1 – max3: Legen, abhängig vom gewählten Modus (Farbsystem), die maximalen Wertebereiche für rot, grün und blau (RGB) oder Farb-Winkel, Sättigung und Helligkeit (HSB) fest.
maxA: Maximaler Wertebereich für Alpha (Transparenz).

66 Vgl. URL: https://processing.org/reference/ [21.07.2015].

colorMode(RGB);
colorMode(RGB, 255);
colorMode(RGB, 255, 255, 255);
colorMode(RGB, 255, 255, 255, 255);

Je nach Anforderung kann die passende Schreibweise für colorMode, mit oder ohne angepasste Wertebereiche, verwendet werden. Wenn der Farb-Modus nicht definiert ist, wird standardmäßig RGB mit Wertebereichen von 0–255 verwendet.[67]

67 Vgl. ebenda.

colorMode(HSB);
colorMode(HSB, 360);
colorMode(HSB, 360, 100, 100);
colorMode(HSB, 360, 100, 100, 100);

Farbwerte definieren

Farbwerte werden typischerweise in die Funktionen **background()**, **fill()**, **stroke()** und **color()** eingesetzt. Die folgende Syntax ist für alle vier Funktionen anwendbar:[68]

```
background(rgb);
background(rgba);
background(rgb, alpha);
background(gray);
background(gray, alpha);
background(v1, v2, v3);
background(v1, v2, v3, alpha);
```

rgb: Ein Hex-RGB-Wert (siehe Hex-RGB) oder eine Variable, die RGB/HSB-Farbwerte enthält (siehe Seite 59).
rgba: Ein Hex-Alpha-RGB-Wert (siehe Hex-RGB) oder eine Variable, die RGBA/HSBA-Farbwerte enthält.
alpha: Transparenz-Wert als (separate) Zahl oder Variable.
gray: Ein einzelner Farbwert wird als Grau-Wert interpretiert, bspw. steht 0 für schwarz, 128 für mittelgrau und 255 für weiß (jeweils abhängig vom definierten maximalen Wertebereich).
v1–v3 legt die Farbwerte für rot, grün und blau (RGB) oder Farb-Winkel, Sättigung und Helligkeit (HSB) fest.

Mit den Funktionen für **red()**, **green()**, **blue()**, **hue()**, **saturation()**, **brightness()** und **alpha()** können auch einzelne Farb- bzw. Transparenzwerte vergeben werden.

Hex-RGB

Werte in der Hex-RGB-Schreibweise können in diesem Schema definiert werden: **#RRGGBB**.
Wenn ein Hex-RGB-Wert inklusive einem Alpha-Wert notiert werden soll, wird der zweistellige Alpha-Wert direkt vor die RGB-Werte gestellt. Am Anfang steht das Präfix „0x“: **0xAARRGGBB**. Gut zu wissen: Einzelne Rot-, Grün-, Blau- oder Alpha-Werte können nicht in der hexadezimalen Schreibweise notiert werden, sondern immer nur zusammen. Allerdings ist es möglich, einen Hex-RGB-Wert und einen dezimalen Alpha-Wert zu definieren, z. B. color(#006699, 191).[69]

68 Vgl. URL: https://processing.org/reference/background_.html [15.07.2015].

69 Vgl. URL: https://processing.org/reference/color_datatype.html [15.07.2015].

```
background(#FFEE00)     color(#FFEE00)
fill(#FFEE00)           #FFEE00
stroke(#FFEE00)
```

Beispiele für die Angabe von Hex-RGB-Werten in verschiedenen Funktionen. Die Funktion color() ist bei Angabe eines Hex-RGB-Wertes obsolet, für andere Funktionen gilt dies nicht.

Variablen

Einer der größten Vorteile ist, dass in Processing Farbwerte in Variablen gespeichert, an anderen Stellen wieder aufgerufen und im Bedarfsfall auch überschrieben oder angepasst werden können, zum Beispiel so:[70]

```
color c2;
color c1 = color(204, 153, 0);
c2 = #FFCC00;

fill(c1);
[...]
fill(c2);
[...]
color c3 = get(10, 50);
fill(c3);
[...]
```

70 Vgl. URL: https://processing.org/reference/color_datatype.html [15.07.2015].

Mit „color c1“ wird eine Farb-Variable mit dem Namen „c1“ definiert, der Farbwert wird nach dem Gleichheitszeichen festgelegt. Variablendefinition und Wertzuweisung können auch separat erfolgen wie bei „c2“.
Mit **get()** können die Farbwerte eines einzelnen Pixels ausgelesen und in der Variable gespeichert werden, dazu werden einfach die jeweiligen x- und y-Koordinaten des Pixels genannt.[71]
Sogar Systemvariablen wie bspw. **width**, **height**, **mouseX** und **mouseY** können als Farbwerte eingesetzt werden, sofern deren Werte die zulässigen Wertebereiche des Farbmodus nicht überschreiten, Beispiel: color(width, mouseX, 255, height).

71 Vgl. URL: https://processing.org/reference/get_.html [15.07.2015]

„Der direkte Einfluss auf 16.777.216 Farben ermöglicht auch einen anderen Zugang zur Farbe an sich.“[72]

72 Bohnacker et. al., 2009, S. 182

Zufallswerte

Eine spannende Möglichkeit ist die **random()**-Funktion, mit der zufällige Farbwerte generiert werden können. Die Syntax:[73]

random(high)
random(low, high)

high: Gibt einen zufälligen Zahlenwert zwischen 0 und der definierten Zahl aus, wobei diese nie ganz erreicht wird.[74]
low–high: Gibt einen zufälligen Zahlenwert zwischen den definierten Zahlen aus, auch hier wird der Maximalwert nie ganz erreicht.[75]

Die random()-Funktion gibt jedes mal, wenn sie durchlaufen oder aufgerufen wird, neue Werte aus.[76] Daher kann es sinnvoll sein, sie nicht in den draw-Bereich zu schreiben, sondern in den setup-Bereich oder eine separate Funktion, die nur bei Bedarf aufgerufen wird.

73 URL: https://processing.org/reference/random_.html [15.07.2015]
74 ebenda.
75 ebenda.
76 ebenda.

```
background(random(255), random(0, 100), random(200, 255));
```

Farbreihen

Mit der Funktion **lerpColor()** können Farbwerte interpoliert werden, die zwischen zwei definierten Werten liegen. Die Syntax:[77]

lerpColor(c1, c2, amt)

c1: Farbwert oder Farbwert-Variable der Startfarbe.
c2: Farbwert oder Farbwert-Variable der Endfarbe.
amt: Eine Zahl zwischen 0.0 und 1.0, die den Abstand der interpolierten Farbe von den Start- und Endfarben definiert. Abhängig von der gewünschten Anzahl der Abstufungen sollte der Abstand gleichmäßig gewählt werden, damit das Ergebnis ausgewogen ist.

77 URL: https://processing.org/reference/lerpColor_.html [15.07.2015]

```
void setup() {
size(400, 400);

}

void draw() {
colorMode(HSB, 360, 100, 100);
background(360);
noStroke();
color c1 = color(20, 100, 100);
color c4 = color(50, 100, 100);
color c2 = lerpColor(c1, c4, .33);
color c3 = lerpColor(c1, c4, .66);
fill(c1);
rect(0, 0, width/4, height);
fill(c2);
rect(100, 0, width/4, height);
fill(c3);
rect(200, 0, width/4, height);
fill(c4);
rect(300, 0, width/4, height);
}
```

Dieses Bespiel zeigt die lerpColor()-Funktion in der praktischen Anwendung. Das Ergebnis ist die nebenstehende Abbildung.

Eine mit der lerpColor()-Funktion interpolierte Farbreihe.

8. Verzeichnisse

8.3. Literaturverzeichnis

Bohnacker, Hartmut/Groß, Benedikt/Laub, Julia: Generative Gestaltung. Entwerfen Programmieren Visualisieren, Verlag Hermann Schmidt Mainz, Mainz, 2009.

Burger, Wilhelm und Burge, Mark James: Digitale Bildverarbeitung. Eine algorithmische Einführung mit Java, 3. Auflage, Springer Vieweg, Berlin, Heidelberg, 2015.
ISBN: 9783642046049 (eBook)
URL: http://link.springer.com/book/10.1007/978-3-642-04604-9 [11.07.2015]

Gull, Clemens und Münz, Stefan: HTML 5 Handbuch. Zukunftsorientierte Webseiten erstellen, Franzis Verlag GmbH, Haar, 2012.
ISBN: 9783645220811 (eBook)
URL: https://itunes.apple.com/de/book/html-5-handbuch/id527720127?mt=11 [11.07.2015]

Welzel, Sebastian: Farbe Struktur Oberfläche. Mustermappe, Hildesheim, 2012.