

Processing



Alexander Pann

Entdecke deine künstlerischen Fähigkeiten

B G / B R G C a r n e r i

G r a z

F a c h b e r e i c h s a r b e i t

2 0 1 1



Processing

© Alexander Pann 2011



Inhaltsverzeichnis

Vorwort	6
Processing allgemein	9
Die Processing IDE	10
Export von Processing Sketchen	11
Ähnliche bzw. auf Processing basierende Projekte.....	15
Bücher	16
Praktisches Projekt.....	33
Vorwort	33
Wichtige Bemerkungen zu den Erklärungen.....	34
Grundlagen und grundsätzlicher Aufbau eines Sketches.....	35
Importe und Kommentare.....	35
Mathematische Operatoren.....	38
Typenumwandlung.....	39
Anmerkung: Die Rolle der englischen Sprache in der Programmierung	40
Kontrollstrukturen	41
If-Else-Anweisung.....	41
Fehlermeldungen.....	43
Switch-Anweisung.....	47
While-Schleife	48
For-Schleife.....	49
Zuweisung.....	49
Processing IDE.....	50
Was tun bei unlösbaren Problemen und Bugs und wie erhält man Support?	53
Fortsetzung des Kapitels „Processing IDE“	54
Verschiedene Tipps zur IDE	55
Music Visual – Zeile 52 bis 66	55
OpenGL und Java OpenGL	58
Music Visual – Zeile 68 bis 73	60
Music Visual – Zeile 75 bis 78	61

Music Visual – Musikwiedergabe	61
Theorie zur „beat analysis“.....	63
Theorie zu MSAFluid.....	63
Music Visual – Zeile 80 bis 87	63
Music Visual – Die grafische Oberfläche	64
Button.....	65
Slider.....	65
Toggle	66
Textarea.....	67
Arrays.....	67
Erweiterte For-Schleifen.....	68
Die Methoden der GUI-Komponenten.....	69
Bitweise-AND-Zuweisung	71
Bitweise-OR-Zuweisung.....	71
Klassen – Aufbau	76
Sichtbarkeit von Variablen	76
Klassen	73
Interfaces.....	76
BeatListener.....	76
Konstanten	78
MSAParticle	80
MSAFluid – Particlesystem	83
MSAFluid – Hauptklasse	87
draw()	88
Perlin Noise.....	90
Music Visual – Zeile 110 bis 127	90
Music Visual – Zeile 129 bis 162	91
Inputfunktionen.....	93
Durchlauf des Programmes	95

Funktionsweise von Processing und weitere Funktionen.....111

Core.jar.....	112
PApplet.....	112
PFont.....	144
PStyle.....	145
PVektor.....	146
XML.....	148
PMatrix/PMatrix2D/PMatrix3D.....	148
PGraphics2D.....	162
PShape.....	164
PShapeSVG.....	165
Table	169
PIimage.....	169

Damals und heute: Die Veränderungen in Processing.....170

Änderungen zwischen Alpha (Release 0069) und Beta (Release 0085).....	170
Änderungen in Beta 0116+ (Annäherung an Version 1.0).....	172
Änderungen in Processing 1.0 (Revision 0162).....	173
Bevorstehende Änderungen in Processing 2.0.....	176

Nachwort.....177**Danksagung.....179****Anhang I – Der gesamte Quellcode von Music Visual180****Anhang II – Das Cover dieses Buches in Processing.....194**

Über den Autor



Alexander Pann wurde am 18. April 1994 in Graz, Österreich, geboren.

Seit 2004 besucht er das BG/BRG Carneri Gymnasium in Graz-Gedorf und hat sich in der Oberstufe für das Wahlpflichtfach Informatik entschieden, da er schon in früheren Jahren begonnen hat, sich für verschiedene Programmiersprachen zu interessieren.

Er hat dabei Erfahrungen in den Bereichen Processing, Java, und PHP gesammelt und sich Techniken der Webentwicklung (HTML, CSS, JavaScript, Ajax, PHP, MySQL, Java Applets) angeeignet.

Grundkenntnisse in Visual Basic und Umgang mit den beiden Betriebssystemen Windows und Linux ([K]Ubuntu, Knoppix) sowie deren Office-Anwendungen (Microsoft Office, LibreOffice, OpenOffice) sind dabei obligatorisch.

Die vorliegende Arbeit ist in den Sommerferien 2011 nach der 11. Schulstufe entstanden. Sie wurde als Fachbereichsarbeit zur Matura eingereicht. Alexander Pann hat sich die neue Programmiersprache „Processing“ im Selbststudium angeeignet und aufgrund ihrer leichten Erlernbarkeit zur Ergänzung des Unterrichtsfaches Informatik empfohlen.

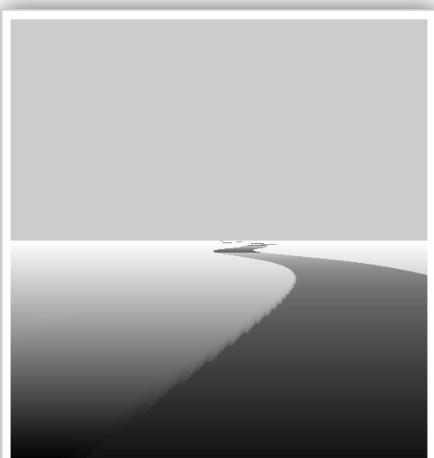
Der Autor sieht in Processing eine exzellente Möglichkeit der Kreativität junger Leute freien Lauf zu lassen und hat selbst schon sehr zahlreiche, sogenannte „Sketches“ in dieser Programmiersprache geschrieben. Er ist auch aktives Mitglied der offiziellen Processing Community und postet in diversen Foren Anregungen und Fragen.

Praktische Erfahrungen als Softwareentwickler hat er bereits als Berater in einem kommerziellen Unternehmen gesammelt: das internationale, im Bereich der Mikroelektronik tätige Unternehmen *austriamicrosystems* in Unterpremstätten bei Graz hat er im Bereich Webinterface (i-Foundry) tatkräftig unterstützt. Für die Zukunft ist das Studium der Informatik an der Technischen Universität Graz geplant.

Vorwort

Computer gehören heutzutage zum Alltag eines jeden Jugendlichen, fast jeder besitzt schon einen eigenen PC oder ein eigenes Handy. Vor zehn Jahren gab es zwar in den Schulklassen zwar auch schon Stand-PCs, doch damals war ein entscheidender Punkt ganz anders als heute: Damals war es etwas Besonderes. Wir haben uns für den Computer, genauer gesagt Computerspiele viel stärker interessiert und haben jede Pause darüber gestritten, wer als nächstes das Level spielen darf. Wir sind mit drei verschiedenen Spielen in zwei Schuljahren ausgekommen, heute besitzen Teenager oft über 100 Computerspiele und der Reiz hat trotz immer besserer Grafiken und Interaktionsmöglichkeiten meiner Meinung nach abgenommen. Der Computer wird von Erwachsenen als Arbeitsgerät verwendet, von Jugendlichen und auch Kindern erfahrungsgemäß um mit Freunden über das Internet zu kommunizieren. Die dafür nötigen Techniken werden immer komplizierter und komplexer, die Technologien

für einen hohen Sicherheitsstandard müssen ständig verbessert werden. Der durchschnittliche User will jedoch mit der Technik im Hintergrund nicht in Berührung kommen.



Muss man aber ein echter „Computerfreak“ sein um am Computer etwas zu erschaffen? Vielleicht ein soziales Netzwerk? Eine API für Wii-Remotes, ein eigenes Betriebssystem? Die Antwort ist möglicherweise „Ja“, da man tausende Zeilen Code formulieren müsste, doch meine Antwort lautet:

```
01 float h=256,t,T,x,y;void draw(){t+=x=.3-mouseY/h/5;T+=mouseX/h*x  
-x;noStroke();for(y=0;++;y<h){fill(h-y,85);rect(0,h+y,h*2,1);x=h  
/y+t;fill(x%20*9,85);x=T+noise(x/99)*60-30;rect(h-x*y-2*y,h+y,y*4,1);}}
```

Dieser 197 Zeichen lange Sketch heißt „Driving through Iceland“ und wurde der Gewinner des Processing Tiny Sketch Contest. Mit der Maus kann man sich auf der Straße nach links und rechts zu bewegen, wobei sich die Straße weiterbewegt. Der Sketch ist in Schwarz-weiß gehalten. Es ist ein kurzes und kompaktes Beispiel, wie einfach man mit Processing etwas Künstlerisches „in die Welt setzen“ kann, auch wenn man eigentlich auf diesem Gebiet nicht begabt ist. Einer der Gründe, warum ich mich sofort für die Programmiersprache Processing begeistert habe, ist sicherlich, dass ich meine Fantasien und Ideen in etwas Schöpferisches umsetzen kann, ohne dass ich einen Zeichenblock anfassen muss und mich danach für das Resultat schäme.

Ein weiteres Argument war für mich die Einfachheit der Befehle, die sich überwiegend direkt aus dem Englischen ableiten lassen. Ein Beispiel hierfür ist die Funktion “ellipse” mit der sich ein Kreis oder eben eine Ellipse darstellen lässt. Man erkennt daran, dass sich diese Sprache an Programmieranfänger richtet, jedoch haben auch erfahrene Programmierer an ihr große Freude, da sie sehr leicht zu erweitern ist.

Viele Programmiersprachen fordern vom Programmierer, dass er zahlreiche Befehle auswendig lernt. Processing besteht aus so wenigen Befehlen, dass sie auf der Processing Webseite in einer großen Tabelle aufgelistet werden können. Die Befehle haben sich im Laufe der Jahre auch fast nicht verändert, es kommen nur ab und zu neue Befehle hinzu. Aus diesem Grund ist es nicht nötig, jeden Monat umzulernen. Die einzelnen Processing Versionen sind ab Version 1.0 meist miteinander kompatibel. Sollte ein alter Befehl verwendet werden, wird die Processing Homepage geöffnet, und man wird über die Änderung informiert. Zusätzlich erscheint in der Konsole der Befehl, den man stattdessen verwenden soll.

Der Programmierer muss sich auch nicht um jedes einzelne Detail kümmern, sondern kann seinen Ideen freien Lauf lassen und sie in seinen Sketch einbringen. Noch ein weiterer Hauptgrund, dass ich bei dieser elektronischen Sprache geblieben bin, da viele ähnliche Sprachen, wie auch Java, C und Python viele zusätzliche Befehle benötigen. Ich möchte das anhand des Vergleiches von Processing und DirectX demonstrieren:

Die Aufgabenstellung lautet: *Es soll mit möglichst wenigen Zeilen Code ein Fenster auf dem Bildschirm angezeigt werden. Dargestellt werden soll ein Fenster, welches sich minimieren, maximieren und schließen lässt.*

Auf der nächsten Seite sind die nötigen Quellcodes beider Sprachen nebeneinander dargestellt. Im linken Bild werden jene Befehle gezeigt, die für die Programmierschnittstellen-sammlung DirectX relevant sind. Rechts ist die Implementierung in Processing ersichtlich. Wie selbst der Laie erkennen kann, ist die Realisierung mit Processing wesentlich einfacher und doch gleich effizient.



DirectX

```
// include the basic windows header file
#include <windows.h>
#include <windowsx.h>

// the WindowProc function prototype
LRESULT CALLBACK WindowProc(HWND hWnd,
    UINT message,
    WPARAM wParam,
    LPARAM lParam);

// the entry point for any Windows program
int WINAPI WinMain(HINSTANCE hInstance,
    HINSTANCE hPrevInstance,
    LPSTR lpCmdLine,
    int nCmdShow)
{
    // the handle for the window, filled by a function
    HWND hWnd;
    // this struct holds information for the window class
    WNDCLASSEX wc;
    ZeroMemory(&wc, sizeof(WNDCLASSEX));
    // fill in the struct with the needed information
    wc.cbSize = sizeof(WNDCLASSEX);
    wc.style = CS_HREDRAW | CS_VREDRAW;
    wc.lpfnWndProc = WindowProc;
    wc.hInstance = hInstance;
    wc.hCursor = LoadCursor(NULL, IDC_ARROW);
    wc.hbrBackground = (HBRUSH)COLOR_WINDOW;
    wc.lpszClassName = L"WindowClass1";
    // register the window class
    RegisterClassEx(&wc);
    // create the window and use the result as the handle
    hWnd = CreateWindowEx(NULL,
        L"WindowClass1", // name of the window class
        L"Our First Windowed Program", // title of the window
        WS_OVERLAPPEDWINDOW, // window style
        300, // x-position of the window
        300, // y-position of the window
        500, // width of the window
        400, // height of the window
        NULL, // we have no parent window, NULL
        NULL, // we aren't using menus, NULL
        hInstance, // application handle
        NULL); // used with multiple windows, NULL
    // display the window on the screen
    ShowWindow(hWnd, nCmdShow);
    // enter the main loop:
    // this struct holds Windows event messages
    MSG msg;
    // wait for the next message in the queue, store the result in 'msg'
    while(GetMessage(&msg, NULL, 0, 0))
    {
        // translate keystroke messages into the right format
        TranslateMessage(&msg);
        // send the message to the WindowProc function
        DispatchMessage(&msg);
    }
    // return this part of the WM_QUIT message to Windows
    return msg.wParam;
}

// this is the main message handler for the program
LRESULT CALLBACK WindowProc(HWND hWnd, UINT message, WPARAM wParam, LPARAM lParam)
{
    // sort through and find what code to run for the message given
    switch(message)
    {
        // this message is read when the window is closed
        case WM_DESTROY:
            // close the application entirely
            PostQuitMessage(0);
            return 0;
        } break;
    }
}
```

Kreativität, Geduld und elementare Englischkenntnisse. Zu dieser Arbeit: Dieses Werk ist dreigeteilt. Der erste Teil besteht aus einer allgemeinen Beschreibung von Processing. Der zweite Teil beschäftigt sich mit der praktischen Arbeit „Music Visual“. Im letzten Teil wird die Funktionsweise von Processing, sowie weitere Funktionen erklärt. Innerhalb der Erklärungen wird man durch zusätzliche Informationen in die Grundlagen der Programmierung mit Processing eingeführt.

Processing

```
01 void setup() {
02
03     size(500, 500);
04
05 }
```

Ich habe die Erfahrung gemacht, dass diese Einfachheit nicht bloß mich begeisterte, sondern die ganze Informatikgruppe unserer Schule. Ich bin davon überzeugt, dass der Informatikunterricht durch diese neue Programmiersprache attraktiver geworden ist, da sich herausstellen wird, dass jeder X-Beliebige, sei es ein erfahrener Computerbenutzer oder nur ein konventioneller Computerkonsument des 21. Jahrhunderts ein individuelles, originelles Werk am Computer hervorbringen kann ohne dafür 1000 Seiten dicke Bücher durchlesen zu müssen. Mit ein bisschen Übung, ist angefangen von der Ausgabe des Textes „Hallo Welt“ bis zu einem 2-dimensionalen Computerspiel mit Processing alles realisierbar. Man braucht nur etwas

Was ist Processing?

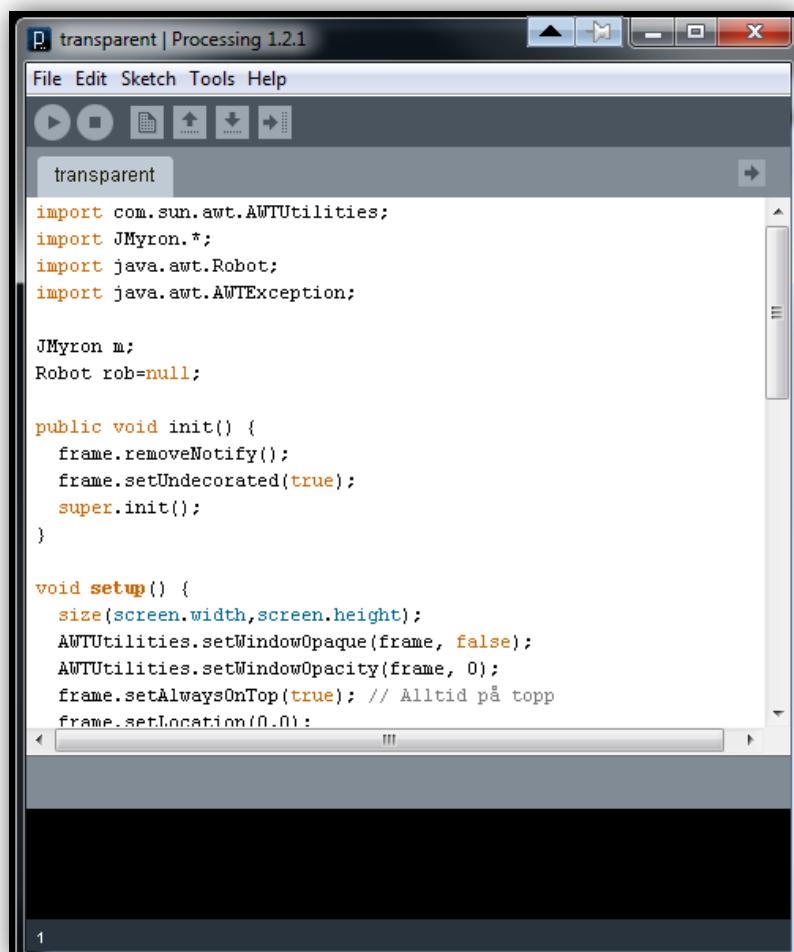
Processing ist eine unter der GNU General Public License und Lesser General Public License lizenzierte Open-Source-Programmiersprache, die bereits eine Entwicklungsumgebung, auf Englisch „Integrated Development Environment (IDE)“, beinhaltet. Die Einsatzgebiete der Sprache umfassen Grafik, Simulation und Animation und deshalb ist sie auch speziell für Künstler, Gestalter und Programmieranfänger geeignet, da sie eine sehr einfache Syntax, also einen einfachen Satzbau, besitzt und sich durch ihre einfachen Befehle auszeichnet. Aus programmiertechnischer Sicht ist sie eine spezialisierte, objektorientierte, stark typisierte Programmiersprache. Sie wird daher in einem kleinen Anwendungsbereich eingesetzt, besitzt einen speziellen Datentyp – das Objekt, von dem alle anderen Objekte abstammen. Stark typisiert ist sie, da schon vor der Laufzeit eines Programmes der Datentyp der einzelnen Variablen, in denen Werte gespeichert werden, feststeht. Processing ist ein quelloffenes Projekt und wurde am Massachusetts Institute of Technology (MIT) in Boston 2001 von Ben Fry (Broad Institute) und Casey Reas (UCLA Design Media Arts), ehemaligen Mitglieder der Aesthetics and Computation Group an der MIT Media Lab, initiiert.

Man kann sie als stark vereinfachte Version der Programmiersprache Java auffassen, da die meisten Funktionen, die auch als Methoden bezeichnet werden, von dem *Abstract Window Toolkit* der Java Foundation Classes abstammen, das für grafische Benutzeroberfläche konzipiert ist. Die Klassenbibliotheken und Third-Party-Libraries der Programmiersprache berücksichtigen vor allem die Gebiete Video, Grafik, Grafikformate, Sound, Animation, Typographie, 3D, Simulation, Datenzugriff und -transfer, sowie Netzwerkprotokolle. Es werden ständig neue Bibliotheken beigesteuert, zum Beispiel Bibliotheken für die Verarbeitung von Touchscreeneingaben, für die Visualisierung der Graffiti Markup Language und vieles mehr. Durch freie, herunterladbare Vorlagen ist es möglich freiwillig mitzuarbeiten, in dem man Bibliotheken veröffentlicht.

Einige andere Programmiersprachen haben Processing in der Vergangenheit beeinflusst, zu denen Design By Numbers, Java, OpenGL, PostScript und C gehören. Processing wurde im Jahr 2005 mit dem Prix Ars Electronica in der Kategorie Net Vision/Net Excellence ausgezeichnet, einem Kulturpreis, der seit 1987 im Rahmen des Ars Electronica Festivals für Kunst, Technologie und Gesellschaft vom Veranstalter ORF Oberösterreich und dem Land Oberösterreich vergeben wird.

Die Processing IDE

Processing beinhaltet ein sogenanntes "sketchbook", das man mit einer sehr schlichten Programmierumgebung für organisierte Projekte vergleichen kann. Jeder Processing Sketch ist eine Unterklasse der Java-Klasse *PApplet* und beinhaltet den größten Teil der Implementierungen der Sprache. Aufgrund dieser Tatsache kann Processing in einem Java Projekt integriert werden, wie später noch erklärt wird. Bei der Umwandlung in Java Code werden alle zusätzlichen Klassen als innere Klasse behandelt. Das heißt, dass statische Variablen und Methoden, die kein Objekt zur Instanziierung benötigen, nicht gestattet sind. Sollten diese trotzdem benötigt werden, muss man explizit Processing mitteilen, dass man im „Java Mode“ programmieren möchte. Nachdem dieser Schritt erfolgt ist, wird der Code in Byte-Code, ein Zwischencode von Java, umgewandelt und erst beim Ausführen in Maschinencode umgewandelt. Durch diesen unabhängigen Byte-Code, der auf jedem Betriebssystem erzeugt werden kann, kann man Processing als plattformunabhängig bezeichnen. Fälschlicherweise könnte man meinen, dass die Processing IDE auch komplett plattformunabhängig ist, sie ist zwar in Java geschrieben, jedoch wird sie auf den verschiedenen Betriebssystemen anders gestartet. Auf die Dateiendungen bezogen, erfolgt die Speicherung von Processing Code in Dateien mit der Endung *.pde*, bei der Konvertierung entsteht eine zweite Datei mit der Endung *.class*.



Export von Processing Sketchen

Es gibt zwei verschiedene Arten, um Processing Projekte zu präsentieren: in einem Java Applet im Webbrower oder als Stand-Alone-Desktopanwendung. Bei zuerst genannter Methode, wird ein Jar-Archiv erzeugt, dass normalweise Java Code beinhaltet. In diesem Archiv befinden sich nach dem Export folgenden Dateien:

- **index.html**

index.html ist eine Datei, die in HTML geschrieben ist und in der das Applet eingebunden und der Quellcode verlinkt wird. Zusätzlich befindet sich ein weiterer Link auf der Seite, der auf die Startseite der offiziellen Homepage von Processing zeigt.

- **XXX.jar**

Dieses Archiv beinhaltet alle für die Ausführungen nötigen Dateien, angefangen von den Standardklassen, die „core classes“ genannt werden, über die selbstgeschrieben Klassen bis hin zu allen Mediendateien, die sich in einem Ordner mit dem Namen „data“ befinden.

- **XXX.java**

Die Java Quellcodedatei wird vom Präprozessor aus der PDE-Datei generiert. Der eigentliche Sketch wird nun in Java umgeschrieben.

- **XXX.pde**

XXX.pde ist die Originaldatei, die auch von index.html verlinkt wird.

- **loading.gif**

Diese animierte Grafik wird beim Laden des Applets angezeigt. Wenn das Applet sehr schnell geladen wird, ist sie nicht sichtbar.

Die Alternative zu dieser Exportmethode ist die Erstellung einer ausführbaren Anwendung, für die Betriebssysteme Linux, Apple Macintosh und Microsoft Windows. Der Export befindet sich im Gegensatz zum Applet nicht in einem Ordner „applet“ sondern nennt sich „application.xxxx“, wobei xxxx für die jeweilige Plattform steht. Bei 64 Bit Computern ist zu beachten, dass für sie ein eigener Ordner angelegt wird, genauso wie für 32 Bit. Es ist jedoch möglich, dass sich eine 64 Bit Anwendung nicht auf einem 64 Bit Computer ausführen lässt.

In diesem Fall muss eine 32 Bit Anwendung erstellt werden, welche sich dann problemlos öffnen lassen sollte. Bei Desktopanwendungen kann der bereits erwähnte Java Modus eingesetzt werden. Dafür muss man den Sketch in eine Java Klasse einbetten, die die Klasse PApplet erweitert. Zusätzlich muss eine Methode **main()** hinzugefügt werden, damit Java weiß, wo der Programmcode gestartet wird. Die Entwickler der Sprache sprechen sich aber gegen diese Methode aus, da sie gegen das Konzept von Processing ist und Processing klar und verständlich sein sollte.

Die Art der Anwendung unterscheidet sich auf den verschiedenen Plattformen. Bei Mac OS X wird ein .app bundle erzeugt, wie bei regulären Anwendungen.

Bei Windows wird eine ausführbare Datei (.exe) erzeugt, wobei ein Ordner „lib“ existieren muss, damit die Anwendung laufen kann. Um alles in eine Exe-Datei packen zu können, muss ein Java EXE-Generator wie JSmooth oder launch4j verwendet werden. Für Anfänger ist diese Technik nicht empfehlenswert, da fortgeschrittenere Informatikkenntnisse nötig sind.

Bei Linux werden die Anwendungen über ein einfaches Shellskript gestartet, wodurch eine Unterstützung aller Unixsysteme gewährleistet ist.

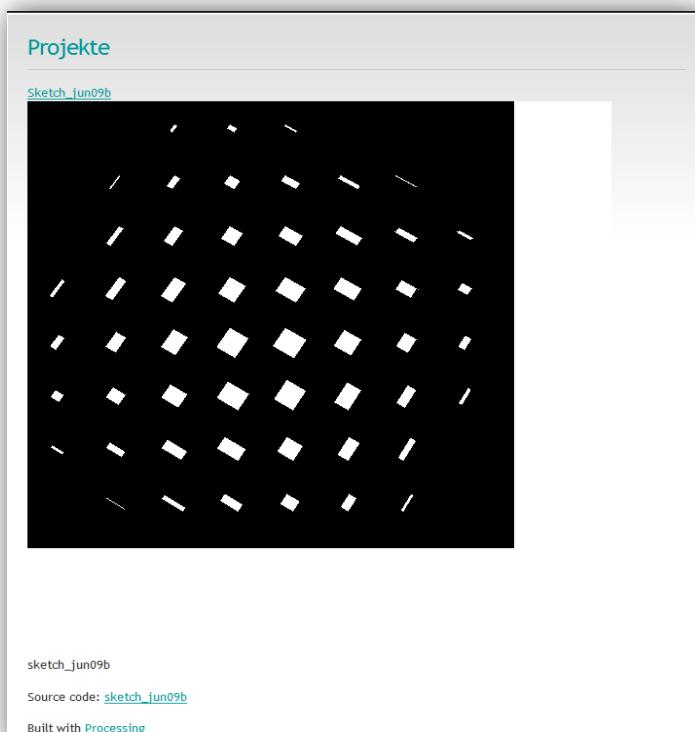
Wenn Sketche exportiert werden müssen, ist auf die Sicherheitseinschränkungen zu achten, die oft bei Applets und Mac-Anwendungen entstehen. Speziell beim Erweitern der Sprache durch Java gibt es im Internet oft Beschränkungen. Zum Beispiel ist es nicht möglich in einem Applet mit Java Befehlen den Mauscursor an eine andere Position zu bewegen.

Auf der Seite www.processing.org wird auch vor dem Einsatz von OpenGL gewarnt, da Processing nicht für solche komplizierten Befehle konzipiert ist. Daher wird man bei Fehlern, die in Zusammenhang mit OpenGL entstehen auf weniger Hilfe im offiziellen Forum stoßen, jedoch kann ich persönlich dieser Kritik nicht nachvollziehen. Ich persönlich habe die Erfahrung gemacht, dass viele Benutzer von Processing sich trotzdem an OpenGL heranwagen und aufgrund von häufig auftretenden Fehlern (sogenannte Bugs), viele Fragen zu diesem Thema haben.

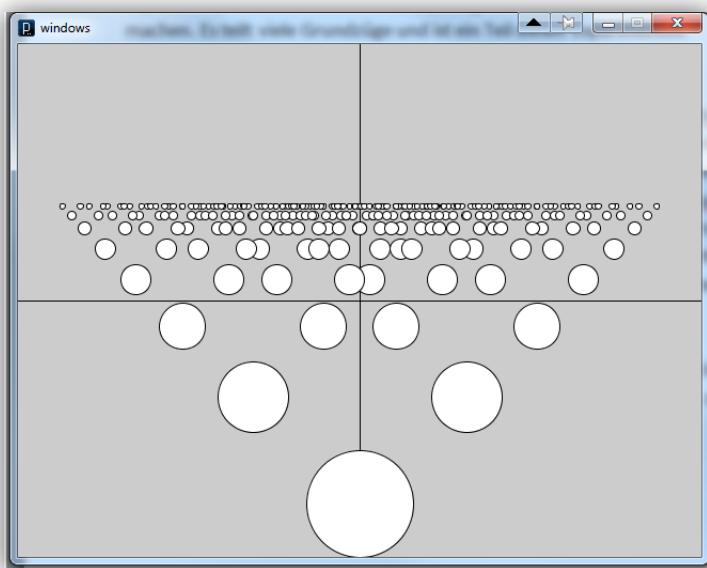
Beispiel für den Java Modus

```
01 Public class JavamodusBeispiel extends PApplet {  
02 // Dateiname: JavamodeBeispiel.pde  
03  
04 static public void main(String args[]) {  
05     PApplet.main(new String[] { "JavamodeBeispiel" });  
06 }  
07 static public void setup() {  
08     // Code  
09 }  
10 static public void draw() {  
11     // Code  
12 }  
13  
14 }
```

Beispiel für die Einbettung eines Applets in eine HTML-Seite. Dieses Beispiel zeigt eine einfache Interaktion des Users mit Hilfe der Maus.



Beispiel für eine Desktop-Anwendung



Es stellt sich die Frage, ob man Applets oder Anwendungen vorziehen sollte. Applets haben den Vorteil, dass das Programm direkt im Browser betrachtet werden kann und somit der Quellcode leicht mit anderen geteilt werden kann. Openprocessing.org ist die größte Seite, auf der man Sketches betrachten und mit anderen Menschen teilen kann. Es kann leider nicht vorausgesetzt werden, dass jeder User das Java Plugin im Browser installiert und

dieses auch auf den neuesten Stand aktualisiert hat. Viele Internetsurfer haben nicht das technische Grundwissen um diese Installationen und Updates durchzuführen, detaillierte Anleitungen oder Verlinkungen zu bestehenden Tutorials könnten hier Abhilfe schaffen. Hinzu kommt, dass nicht alle Features, die in der Desktopversion funktionieren, auch im Web ordnungsgemäß laufen.

Desktop-Anwendungen funktionieren meist tadellos, haben nichtsdestoweniger genauso viele Nachteile wie Applets. Zum einen sind sie für Betrachter ein Sicherheitsrisiko, da zum Beispiel in den ausführbaren Dateien von Windows Viren enthalten sein könnten und daher nur sehr ungern Sketches herunterladen bzw. ausführen wollen. Oft fehlt auch die Geduld, zu warten, bis der Sketch geöffnet werden kann. Heutzutage muss alles schnell gehen, daher sind Applets in diesem Fall besser.

Die richtige Entscheidung hängt von den Umständen ab. Wenn man die Möglichkeit hat, die Dateien ins Internet zu stellen, ist das Applet vorzuziehen, genauso wenn man das Resultat mit anderen Menschen über das Internet teilen möchte. Bei festen Installationen, wie das zum Beispiel in Kunstausstellungen der Fall ist, zahlen sich Desktop-Anwendungen aus, da man den Sketch leicht im Fullscreenmodus starten kann. In den meisten Fällen laufen Desktop-Anwendungen auch schneller und die Framerate ist höher, was zu einer flüssigeren Darstellungsweise führt. Dadurch können zum Beispiel auch rechenintensivere Berechnungen durchgeführt oder mehr Objekte angezeigt werden.

Ähnliche bzw. auf Processing basierende Projekte

- Design By Numbers

Processing basiert auf der Projekt „Design By Numbers Project“, ein Experiment, bei dem versucht wurde, Programmierung leicht unterrichtbar zu machen. Processing teilt viele Grundzüge mit Design By Numbers und ist ein Teil dieses Experimentes.

- Wiring, Arduino, and Fritzing

Aus Processing hat sich ein anderes Projekt entwickelt, dass sich „Wiring“ nennt. Es benützt die Processing IDE gemeinsam mit einer vereinfachten Version von C++, um Künstler die Programmierung von Mikrokontrollern nahe zu bringen. Es gibt zwei separate Hardwareprojekte, Wiring und Arduino, die beide die Wiring Umgebung und Sprache benutzen. Ein weiteres Projekt ist „Fritzing“, das Designern und Künstlern ermöglicht, ihre interaktiven Prototypen zu dokumentieren, um aus diesem interaktive Kunstwerke zu gestalten.

- Mobile Processing

Eine weitere Abzweigung des Projektes ist „Mobile Processing“ von Francis Li, das die Benutzung der Processing Sprache und IDE auf mobilen Geräten, die Java unterstützen, ermöglicht.

- Processing.js

„Processing.js“ ist eine JavaScript Portierung von Processing, speziell gestaltet für grafische Visualisierungen, Bilder sowie interaktive Inhalte und benötigt im Gegensatz zu Processing kein Java Plug-in. Sie benötigt hingegen eine sehr neue Webtechnologie (Stand: 2011), HTML5, um mit JavaScript 2D und 3D Inhalte zu rendern. Dafür wird das HTML canvas-Element verwendet, auf dem alle Inhalte gezeichnet werden. Alle modernen Browser haben dieses Element implementiert.

Die Entwicklung von Processing.js wurde von Studenten des Seneca College (Toronto, Kanada) nach dem ersten Release 2008 übernommen. Ein Team von Studenten beendete die Portierung, nachdem sie über 900 Fehler behoben und 12 Releases herausgegeben hatten. Das Projekt ist so erfolgreich, dass sich durch die Kooperation zwischen der Mozilla Foundation und dem Seneca College eine große Gemeinschaft bildete, die von David Humphrey, Al MacDonald und Corban Brook angeführt wird, die zurzeit die Hauptentwickler von Processing.js sind.

- Spde

Das Akronym „Spde“ steht für **S**cala **P**rocessing **D**evelopment **E**nvironment. Spde ersetzt die originale Processing Syntax und den Präprozessor durch die Programmiersprache Scala, die auch auf der Java Plattform läuft und somit dieselben Einschränkungen wie die nicht erlaubten statischen Objekte aufweist. Es gibt auch Vorteile, wie zum Beispiel den noch kürzer gehaltenen Programmcode und die Möglichkeit, funktional zu programmieren, wobei der ganze Programmcode nur aus Funktionen besteht.

- Processing in Clojure

„Clj-Processing“ ist eine Umsetzung von Processing in Clojure, eine Sprache der Programmiersprachenfamilie Lisp, die auch auf der Java Plattform läuft.

- Processing Monsters

„Processing Monsters“ ist ein Projekt von Lukas Vojir, mit dem Ziel, Interessierten die Sprache mit unterhaltenden Animationen beizubringen. Die „Monster“ sind einfache grafische Programme, die in Schwarz und Weiß gestaltet sind und auf Mausbewegungen reagieren. Die Anzahl an Monstern hat bereits die Marke 70 überschritten.

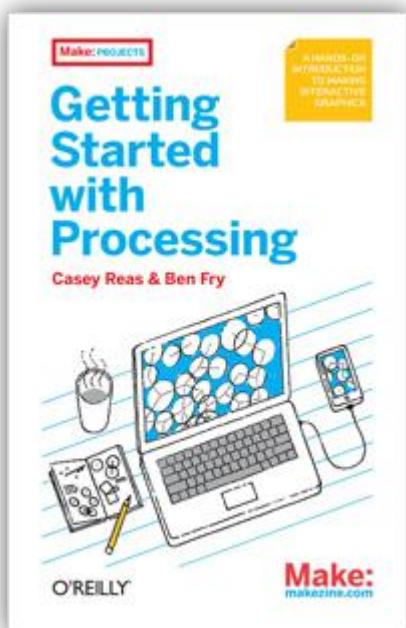
- iProcessing

„iProcessing“ ermöglicht es, native iPhone Anwendungen mit Processing zu programmieren. Möglich gemacht wurde dies durch die Integration von der Bibliothek Processing.js und einem JavaScript Application Framework für das iPhone.

Bücher

Es gibt zurzeit (Stand: 6.8.2011) 16 Bücher, die über diese Programmiersprache handeln, wobei elf Bücher in Englisch, die übrigen fünf in Deutsch, Französisch und Japanisch verfasst sind.

Hinweis: die folgenden Bücherbeschreibungen wurden von der offiziellen Processing-Homepage (<http://processing.org/learning/books/>) entnommen und von mir in das Deutsche übersetzt.



Getting Started with Processing

Casey Reas and Ben Fry.

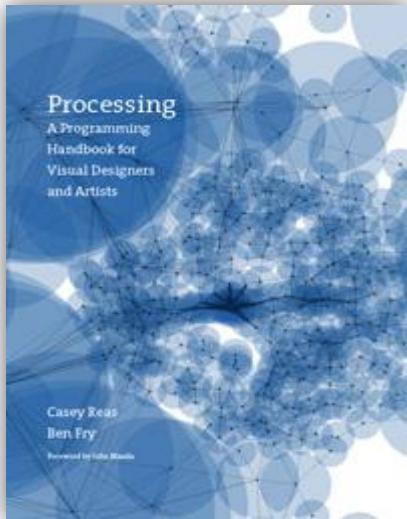
Veröffentlicht im Juni 2010, O'Reilly Media. 208 Seiten.

Preis: \$20

Dieses einfach geschriebene, kostengünstige Buch ist eine kurzgefasste Anleitung für Processing und interaktive Computergrafiken. Es ist von den Gründern von Processing geschrieben und hilft Schritt für Schritt die grundlegenden Programmierkonzepte zu verstehen. Man lernt mit Code etwas zu entwerfen, ein Programm mit ein paar Zeilen zu verfassen, das Ergebnis zu beobachten und es dann weiter zu verbessern. Es wurde geschrieben um den Lesern in 5 Punkten zu helfen:

- Das schnelle Lernen der Grundlagen, von Variablen bis zu Objekten.
- Das Verstehen der Grundlagen von Computergrafiken.
- Das Vertraut werden mit der Sprache Processing.
- Die Entwicklung von interaktiven Grafiken anhand von leicht verständlichen Projekten.
- Die Benützung der Arduino Boards.

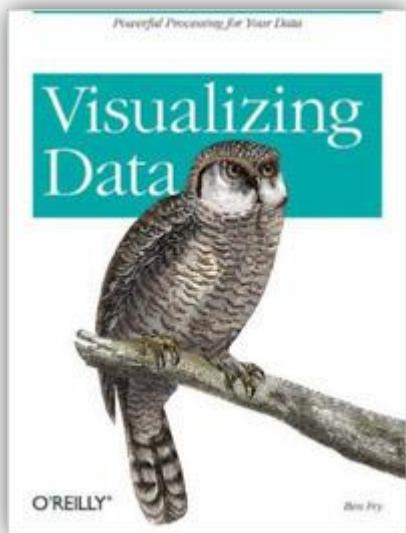
Processing: A Programming Handbook for Visual Designers and Artists



Casey Reas and Ben Fry (Foreword by John Maeda).
Veröffentlicht im August 2007, MIT Press. 736 Seiten.
Gebunde Ausgabe. Preis: \$52

Dieses Buch ist eine Einführung in die Konzepte der Computerprogrammierung mit einem Bezug zur bildenden Kunst. Die Zielgruppe sind erfahrene Computerbenutzer, die sich für die Erstellung von interaktiven und visuellen Arbeiten mit Software interessieren, jedoch nur wenig oder keine Erfahrung darin haben. Es ist das Ergebnis von sechs Jahren Softwareentwicklung und Unterrichtserfahrung. Die im Buch vorkommenden Ideen werden bis jetzt in Klassenräumen, Computerlabs, an Universitäten, Kunst- und Designschulen getestet.

Der Großteil des Buches ist in Abschnitte gegliedert, in denen ein bestimmter Teil der Software diskutiert wird und wie dieser mit Kunst in Verbindung gebracht werden kann. Diese Abschnitte erklären die Syntax und Konzepte von Software, wie zum Beispiel Variablen, Funktionen und die objektorientierte Programmierung. Sie decken Themen wie Fotografie und Zeichnen in Bezug auf Software ab. Diese Abschnitte beinhalten viele kurze, praxisorientierte Beispielprogramme mit Bildern und Erklärungen. Fortgeschrittene, professionelle Projekte von verschiedenen Gebieten, Animation, Performanz, und Typografie, werden von dessen Entwicklern in Interviews erklärt. Der Anhang besteht aus kurzen Einführungen in die künftigen Einsatzgebiete der Forschung, dazu gehören die Themen künstliches Sehen, Geräusche und Elektronik.



Visualizing Data

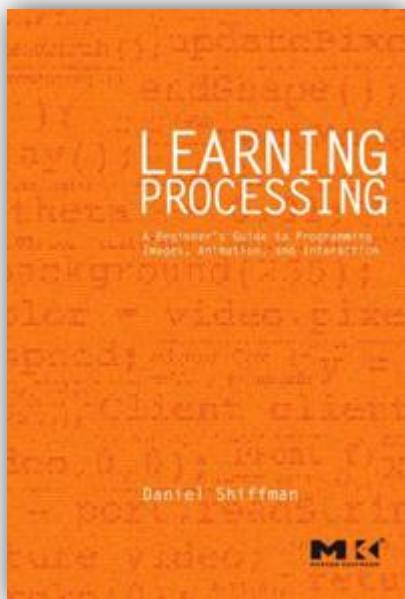
Ben Fry.

Veröffentlicht im Dezember 2007, O'Reilly. 384 Seiten.
Taschenbuch. Preis: \$40

Die O'Reilly Webseite schreibt: „ Wie kann man Nutzen aus Daten ziehen, die man möglicherweise ansonsten niemals genutzt hätte? Mit der Hilfe einer neuen leistungsfähigen Programmierumgebung [Processing], hilft Ihnen dieses Buch Daten im Web oder sonst wo auf präzise Art und Weise mit Hilfe von Interaktionen des Benutzers, Animationen und vielem mehr zu präsentieren. Sie werden die grundlegenden Darstellungsprinzipien kennenlernen, wie man die für Ihren Zweck richtige Technik auswählt und wie man interaktive Features bereitstellt um Benutzeroberflächen aus großen, komplexen Datenmengen zu designen.

Martin Wattenberg, ein Wissenschaftler und Künstler des IBM Watson Research Center sagt über dieses Buch: „ Dieser großartige, detaillierte Führer von einem der Besten der modernen Datengraphikdarstellung, erklärt Ihnen alles, was Sie brauchen um eine eigene Visualisation aus dem Nichts heraus zu programmieren. Möglicherweise sind die Beispiele, in denen Fry ein einfaches Konzept in ein schönes, effektives, fertiges Werk verwandelt am meisten wert.

Lesen Sie dieses Buch und Sie werden nie mehr im Leben abhängig sein, dass andere für Sie ihre Daten visualisieren.“



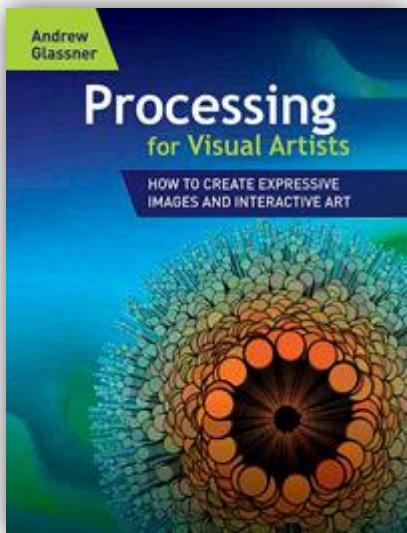
Learning Processing: A Beginner's Guide to Programming Images, Animation, and Interaction

Daniel Shiffman.

Veröffentlicht im August 2008, Morgan Kaufmann. 450 Seiten. Taschenbuch. Preis: \$52

Dan Shiffman sagt über sein Buch: „Dieses Buch erzählt eine Geschichte. Es ist eine Erzählung über Freiheit, über die ersten Schritte in Richtung des Verständnisses der Grundlagen der EDV, des Schreibens von eigenem Code und die Erstellung von eigenen Medien ohne den Einsatz von existierenden Softwarewerkzeugen. Dieses Geschichte ist für Sie.“

Zitat des Herausgebers: „Dieses Buch erklärt Ihnen die grundlegenden Baublöcke der Programmierung, die benötigt werden, um innovative graphische Applikationen mitsamt interaktiver Kunst, Live-Video-Verarbeitung und Datendarstellung zu erschaffen. Es ist eine einzigartige Anleitung und gibt Grafik- und Webdesignern, Künstlern, Illustratoren aus allen Branchen einen zusätzlichen Anstoß mit der Processing Entwicklungsumgebung zu arbeiten, indem Einführungen in die grundlegenden Prinzipien der Programmierung gegeben werden, die mit sorgfältigen Erklärungen von ausgewählten, fortgeschrittenen Techniken begleitet werden.“



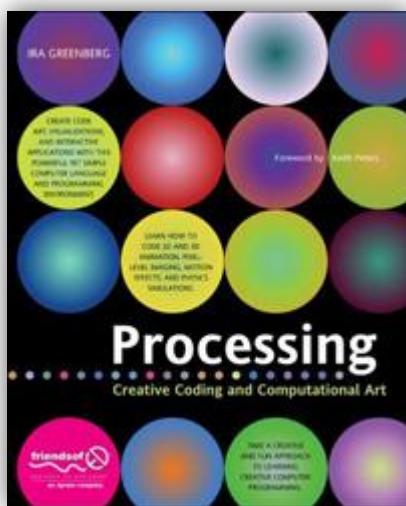
Processing for Visual Artists: How to Create Expressive Images and Interactive Art

Andrew S. Glassner.

Veröffentlicht im August 2010, A K Peters. Taschenbuch. Preis: \$70

Das Buch ist ein Spaziergang mit dem veteranen Autor Andrew Glassner auf einer Reise zu seinen Entdeckungen, auf dem er Processing Projekte von der Inspiration bis hin zur Realität auferichtet. Man kann den Schritten genau folgen, die er macht und Sie werden ganz genau sehen, wie jedes Projekt sich entwickelt, begleitet von großen und kleinen Fehlern, die er auf seinem Weg macht (und wie man sie löst!) und die Zeiten, in denen er die Richtung wechselt. Wenn Sie einmal die Ergebnisse sehen, verstehen Sie, warum die Programmierung so eine mächtige Fähigkeit ist, um sich auszudrücken.

Das Buch beinhaltet verschiedene Perspektiven von Anderen, da Glassner eine lange Erfahrung mit Computergrafiken hat. Seine Bio erzählt: "Dr. Andrew Glassner is a writer-director and a consultant in story structure, interactive fiction and computer graphics. Er began 1968 an 3D-Computergrafiken zu arbeiten und es erfolgte eine Karriere an der NYIT Computer Graphics Lab, der Delft University of Technology, der Bell Communications Research, der Xerox PARC und Microsoft Research. Er ist ein bekannter Autor und hat in einigen technischen Zeitschriften und Büchern über 3D-Modellierung, Rendering und Animation bis hin zu digitaler Soundsynthese Artikel veröffentlicht. Sein Buch '3D Computer Graphics: A Handbook for Artists and Designers' hat eine ganze Generation von Künstlern beeinflusst."



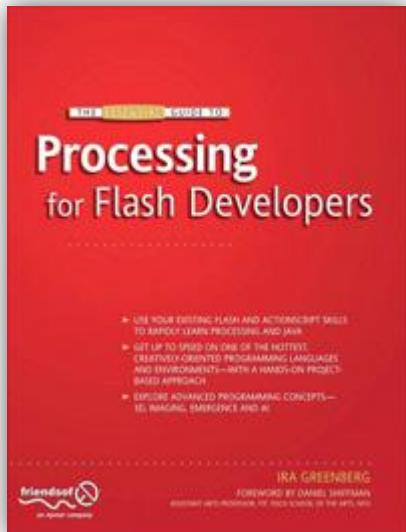
Processing: Creative Coding and Computational Art (Foundation)

Ira Greenberg (Foreword by Keith Peters).

Veröffentlicht im Mai 2007, Friends of Ed. 840 Seiten.

Gebundene Ausgabe. Preis: \$60

Die Freunde von Ed's Webseite schreiben: „Dieses Buch ist speziell für Künstler, Designer und andere kreative Profis und Studierende, die Programmierkunst, Grafikprogrammierung und rechenbetonte Ästhetik erforschen. Das Buch beinhaltet eine solide und umfassende Grundlage der Programmierung, inklusiv objekt-orientierter Programmierung und führt Sie in die leichtverständliche Programmiersprache Processing ein, sodass keine Vorkenntnisse in Programmierung notwendig sind. Das Buch beginnt bei der Programmierung von Linien, Kurven, Formen und Bewegung und setzt an dem Punkt fort, an dem Sie Processing verstanden haben und Sie schon beginnen können, Ihren Ideen freien Lauf zu lassen, mitsamt realistischer Physik, Interaktion und 3D! Im abschließenden Kapitel, werden Sie lernen, Ihre Processing Fähigkeiten mit der leistungsstarken Programmiersprache Java zu erweitern, aus der heraus Processing selbst geschrieben ist.“

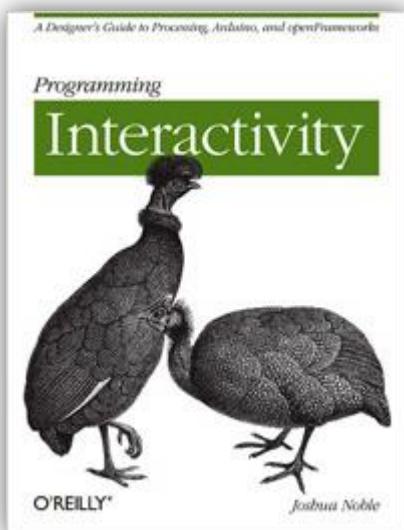


The Essential Guide to Processing for Flash Developers

Ira Greenberg (Foreword by Daniel Shiffman).

Veröffentlicht im Dezember 2009, Friends of Ed. 489 Seiten. Taschenbuch. . Preis: \$50

Ira erzählt: „Der unverzichtbare Führer in Processing für Flash Entwickler nimmt einen detaillierten, praxisorientierten Weg um Processing zu lernen, aufbauend auf Ihren Kenntnissen mit Flash und Erfahrungen mit ActionScript und objektorientierten Programmierkonzepten.“ Die ersten drei Kapitel sind für Neulinge, in denen die nötige Processing spezifische Programmiertheorie anhand von vielen Codebeispielen beigebracht wird. Der Rest des Buches ist auf Projekten basierend, einschließlich Animation von Charakteren, einer Partikel-Engine, eines seriösen Spiels mit künstlicher Intelligenz, einem cellular automata framework mitsamt einem Parsers für das .lif Dateiformat und 3D-Datenvisualisation. Jedes Projekt ist so strukturiert, dass weniger erfahrende Programmierer Fahrt aufnehmen können, während erfahrene Programmierer das Anfangsprojekt stehen lassen können und auf diesem basierend, eine komplexere Anwendung schaffen können. Das letzte Kapitel ist eine Einführung in den Processing Java mode, mit dem eine Verbindung zur Muttersprache von Processing, der Programmiersprache Java, ermöglicht wird.



Programming Interactivity: A Designer's Guide to Processing, Arduino, and openFrameworks

Joshua Noble.

Veröffentlicht im Juli 2009, O'Reilly. 736 Seiten. Taschenbuch. Preis: \$50

Die O'Reilly-Webseite schreibt: „Mache Sie coole Sachen. Wenn Sie ein Designer oder Künstler ohne Programmiererfahrung sind, wird dieses Buch Sie 2D- und 3D-Grafik, Sound, physikalische Interaktionen und elektronische Schaltkreise lehren um alle Arten von interessanten und fesselnden Erfahrungen – online und offline, zu kreieren. Programming Interactivity beschreibt Programmierung und technische, elektronische Grundlagen, und stellt drei gratis Tools vor, die speziell für Künstler und Designer entwickelt worden sind: Processing, Arduino, und OpenFrameworks.“



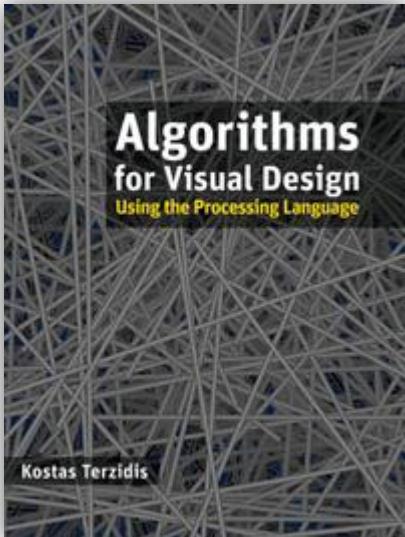
Generative Art

Matt Pearson.

Veröffentlicht im Mai 2011, Manning Publications. 300 Seiten. Taschenbuch. Preis: \$40

Matt bietet folgende Punkte an:

- eine komplette Anleitung für die Erstellung von generativen Graphiken zum Drucken, für Videos und das Web.
- Die Philosophie und praktische Anwendung der Programmiersprache als ein Werkzeug der Kunst.
- Eine Einführung in Processing für Anfänger und Tutorials in Themen wie Perlin Noise, Zufall, Fraktale, Emergence Agent Oriented Programming, 3-dimensionalen Zeichnen und Cellular Automata.
- Eine Vorstellung der Werke von Robert Hodgin, Jared Tarbell, Aaron Koblin, Casey Reas und vielen weiteren zeitgenössischen generativen Künstlern.
- 32 Seiten lange Abschnitte in Farbe.
- Ein Vorwort von Marius Watz.



Algorithms for Visual Design Using the Processing Language

Kostas Terzidis.

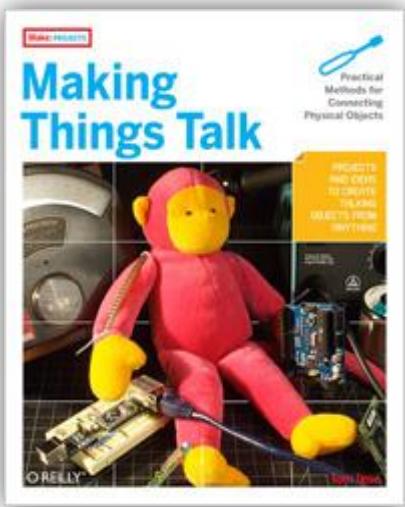
Veröffentlicht im Mai 2009, Wiley. 384 Seiten. Gebundene Ausgabe. Preis: \$60

Die Wiley Webseite schreibt: „Dieses Buch bietet eine Serie von gewöhnlichen Prozeduren an, die als Baustein dienen und Sie ermutigen, sie für Experimente zu nutzen, zu erforschen und Ihre Gedanken, Ideen und Prinzipien in potenzielle Lösungen zu verwandeln. Das Buch beinhaltet Themen wie strukturierte Formen, solid geometry, networking und Datenbanken, physical computing, Bildverarbeitung, grafische Benutzeroberflächen und noch mehr.“

Making Things Talk: Practical Methods for Connecting Physical Objects

Tom Igoe.

Veröffentlicht im September 2007, O'Reilly. 428 Seiten. Taschenbuch. Preis: \$30



Das Buch fokussiert auf über Netzwerk laufende, elektronische Geräte wie Arduino und Wiring, beinhaltet jedoch auch viele Beispiele, bei denen Processing für Grafiken verwendet wird. Die O'Reilly Webseite schreibt: „Durch eine Serie von einfachen Projekten, erklärt dieses Buch, wie man die eigenen Kreationen mit sich und der Umwelt kommunizieren lässt, indem man ein Netzwerk von intelligenten Geräten aufbaut. Egal ob Sie ein paar Sensoren zuhause für das Internet einbauen oder Sie ein Gerät entwickeln müssen, dass kabellos mit anderen Kreationen interagiert, Making Things Talk erklärt genau, was Sie benötigen.“

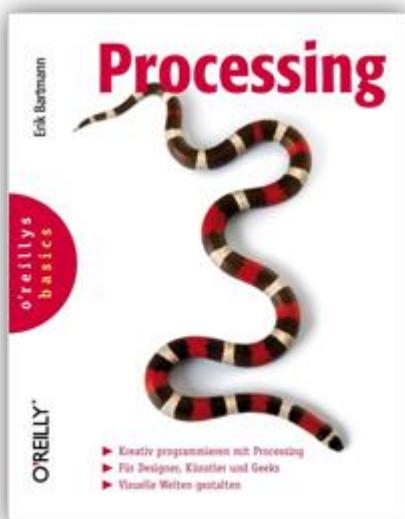


Generative Gestaltung

Hartmut Bohnacker, Benedikt Gross, Julia Laub und
Claudius Lazzeroni.

Veröffentlicht im November 2009, Schmidt Hermann
Verlag. 500 Seiten. Gebundene Ausgabe. Preis: \$75€
Text in Deutsch.

Dieses Buch ist außergewöhnlich. Das Design ist klar und raffiniert und von hoher inhaltlicher Qualität. Nicht so wie in vielen anderen Processing Büchern, werden nicht die Programmiergrundlagen diskutiert, sodass man sofort zu den aufregenden Beispielen fortfahren kann. Ein Zitat des Autors: „Wir wollen eine solide Grundlage bieten, um zu zeigen, wie dieser modifizierte Designprozess benutzt werden kann.“ Die Software in der Buchkategorie »Grundlegende Prinzipien« illustriert ein paar grundlegende Techniken auf den Grundsatz der vier Designbereiche: Farben, Formen, Typographie und Bilder. Dieses Repertoire ist erweitert im Bereich »Komplexe Methoden« durch das Kombinieren von ein paar Prinzipien anhand von sechs größeren Beispielen. In diesem Bereich gibt es auch Erklärungen für fortgeschrittene Techniken.



Processing, O'Reilly Basics

Erik Bartmann.

Veröffentlicht im September 2010, O'Reilly Verlag. 576 Seiten. Taschenbuch. Text in Deutsch. Preis: \$35€

Die OReilly.de Webseite schreibt: "Processing ist eine auf Grafik, Simulation und Animation spezialisierte objektorientierte Programmiersprache, die besonders für Menschen mit wenig Programmiererfahrung geeignet ist. Deshalb eignet sie sich vor allem für Künstler, Bastler und Programmieransteiger. Die aus Java abgeleitete Sprache wurde geschaffen, um schnell und effektiv mit relativ wenig Aufwand zu beeindruckenden Ergebnissen zu kommen. Processing führt den Leser zügig in die Programmieressentials ein und geht dann unmittelbar zur Programmierung grafisch anspruchsvoller Anwendungen über. Spielerisch wird dem Leser die 2D- und 3D-Programmierung, Textrendering, die Bildbearbeitung und darüber hinaus die Videomanipulation näher gebracht."

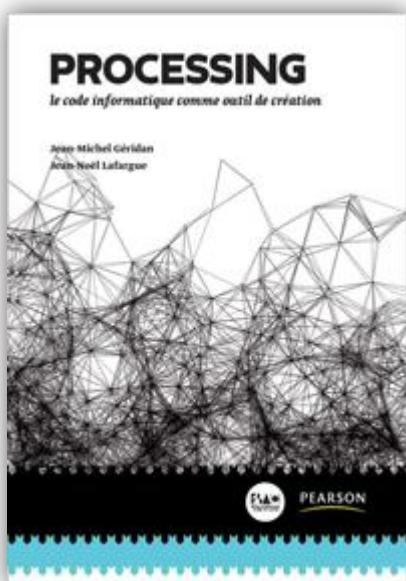


Processing – eine Einführung in die Programmierung

Andres Wanner

Version 1.1; veröffentlicht im Mai 2010, Lulu Press. 83 Seiten. Taschenbuch. Preis: \$40

Diese Publikation stammt aus dem Unterricht an der F+F Schule (Schweiz) und der Hochschule für Gestaltung und Kunst, FHNW. Dies ist zurzeit das ausführlichste deutschsprachige Lehrmittel für Processing. Die vorliegende Version 1.1 wurde im Hinblick auf die aktuelle Sprachversion von Processing überarbeitet. Sie enthält auch ein aktualisiertes Kapitel über Arduino von Hans Peter Wyss und Roland Broennimann.



Processing: Le code informatique comme outil de création

Jean-Michel Géridan und Jean-Noël Lafargue
Veröffentlicht im Februar 2011, Pearson Education.
300 Seiten. Gebundene Ausgabe. Preis: \$38€
Text in Französisch.

Zitat des Verlegers: "Steigern Sie Ihre Kreativität mit Processing! Erstellt von Künstlern für Künstler in einem Geist der Einfachheit und Konsistenz, gleicht die Verarbeitung von einem Schweizer-Messer der von Multimedia in der Programmierung, durch die alle Arten von Anwendungen in den Bereichen Design, Grafiken, animierte Bilder, Audio-, 3D-oder interaktive Kommunikation möglich werden. Mit Processing wird Computercode ein Material der bildenden Kunst, genauso wie auch Ton, Kohle oder Aquarell. Das Buch hilft Ihnen die Kontrolle über die Software zu erlangen und das Gestalten Ihrer ersten Entwürfe. Es behandelt die verschiedenen Aspekte der Sprache, von der Installation der Software, bis zur Produktion von PDF-Dokumenten, Videos, der dynamische Verarbeitung von XML-Daten und die Überwachung von elektronischen Geräten wie den Prototyping-Boards Arduino und Wiring. Unter Ausnutzung ihrer pädagogischen Erfahrung auf dem Gebiet, wollen die Autoren ihr Buch als ein umfassendes Nachschlagewerk über Processing und dem Leser erste Schritte in die Programmierung ermöglichen."



Built with Processing

Veröffentlicht im März 2007, BNN. 232 Seiten.

Taschenbuch.

Text in Japanisch. Preis: 3570 ¥

Anmerkung von Casey: "Ich erhielt eine Kopie von diesem Buch von den Autoren bei einer Reise nach Japan. Es ist ein sehr schön produziertes, farbiges Buch mit Abschnitten, die einen in Processing einführen, sowie vorgestellte Arbeiten, die mit Processing erstellt wurden (viele davon sind von dem Exhibition-Abschnitt der Processing Webseite) und eine Einführung in die Programmierung durch mit der Zeit schwieriger werdenden Beispielen. Der Großteil des Buches ist eine Einführung in die Programmierung. Es gibt viele gute Beispiele und der Code ist farbig dargestellt wie in der Processing Umgebung. Das Buch ist weniger umfangreich als das Buch von Greenberg und die Reas/Fry Bücher, jedoch erscheint es als eine gute, kurze Einführung."

Weiters wird Processing in folgenden Büchern und Projekten erwähnt:¹

Getting Started with Arduino von Massimo Banzi

Processing wird zur Kommunikation mit Arduino Boards verwendet.

Building Wireless Sensor Networks von Robert Faludi

Netzwerkbeispiele mit Processing werden erklärt.

Physical Computing: Sensing and Controlling the Physical World with Computers von Dan O'Sullivan und Tom Igoe

Es werden einige Beispiele angeführt, wie man Processing für RS-232-Kommunikation und künstlichen Sehen verwenden kann.

Aesthetic Computing bearbeitet von Paul Fishwick.

Casey Reas und Ben Fry fügten ein Kapitel mit dem Titel "Processing Code: Programming within the Context of Visual Art and Design" hinzu.

Hacking Roomba: ExtremeTech von Tod E. Kurt.

Processing wird vorgestellt und verwendet um eine Anwendung zu gestalten, die einen Roomba (ein Robotorstaubsauger) kontrolliert.

Analog In, Digital Out von Brendan Dawes.

Einige Projekte, die mit Processing kreiert wurden, werden dargestellt und diskutiert.

¹ Übernommen von <http://processing.org/learning/books/>

Bibliografie

- **Casey Reas und Fry Ben**, *Getting Started with Processing*, O'Reilly Media, 2010
- **Casey Reas und Ben Fry**, *Processing: A Programming Handbook for Visual Designers and Artists*, USA, MIT Press, 2007
- **Fry, Ben**, *Visualizing Data*, USA, O'Reilly, 2007
- **Shiffman, Daniel**, *Learning Processing: A Beginner's Guide to Programming Images, Animation, and Interaction*, USA, Morgan Kaufmann, 2008
- **S. Glassner, Andrew**, *Processing for Visual Artists: How to Create Expressive Images and Interactive Art*, USA, A K Peters, 2010
- **Greenberg ,Ira**, *Processing: Creative Coding and Computational Art (Foundation)*, USA, Friends of Ed, 2007
- **Greenberg ,Ira**, *The Essential Guide to Processing for Flash Developers*, USA, Friends of Ed., 2009
- **Noble, Joshua**, *Programming Interactivity: A Designer's Guide to Processing, Arduino, and openFrameworks*, USA, O'Reilly, 2009
- **Pearson, Matt**, *Generative Art*, Manning Publications ,2011
- **Terzidis, Kostas**, *Algorithms for Visual Design Using the Processing Language*, Wiley , 2009
- **Igoe, Tom**, *Making Things Talk: Practical Methods for Connecting Physical Objects*, USA, O'Reilly, 2007
- **Hartmut Bohnacker, Benedikt Gross, Julia Laub, und Claudio Lazzeroni**, *Generative Gestaltung*, Deutschland, Schmidt Hermann Verlag, 2009
- **Bartmann, Erik**, *Processing*, O'Reilly Basics, Deutschland, O'Reilly Verlag, 2010
- **Wanner, Andres**, *Processing - eine Einführung in die Programmierung*, USA, Lulu Press, 2010
- **Jean-Michel Géridan and Jean-Noël Lafargue**, *Processing : Le code informatique comme outil de création*, Frankreich, Pearson Education, 2011
- **Takashi Maekawa and Kotaro Tanaka**, *Built with Processing*, Japan, BNN, 2010

Praktisches Projekt – Music Visual

Vorwort

Programmiersprachen können nicht nur durch Auswendiglernen der Befehlsreferenz erlernt werden, man muss Projekte und Beispiele analysieren, verstehen und verändern. Genauso verhält es sich auch mit Processing, wobei man die Befehlsreferenz ohne Sorge weglegen kann, da sich Processing Sketche oft wie offene Bücher lesen. Nichts desto trotz ist diese Vorgehensweise nötig, da man sonst seine Fähigkeiten ab einen bestimmten Grad nicht mehr erweitern kann. Aus diesem Grund ist dieses praktische Projekt etwas komplizierter gestaltet als ein einfaches „Hallo-Welt“-Programm, um auch den fortgeschrittenen Lesern das Projekt interessanter zu gestalten. Ich habe mich entschieden nicht nur die Grundfunktionen von Processing zu zeigen, sondern habe auch einige externe Bibliotheken (auf Englisch „contributed libraries“ genannt) benutzt, um zu zeigen, wie viel in dieser Sprache steckt und wie leistungsstark sie ist. Um ein paar fortgeschrittene Techniken zu zeigen, werden einige Features von Java verwendet, der Java mode wird aber nicht verwendet, da Processing und nicht Java im Mittelpunkt des Sketches stehen sollte.

Da ich mich inzwischen für fortgeschrittene Projekte interessiere, ist es ein einfaches Musikvisualisierungsprogramm, bei dem man per Drag & Drop die Musikdateien in den Sketch ziehen kann und diese sofort abgespielt werden. Die Visualisierung erfolgt mit Hilfe von *MSAFluid*², einer Bibliothek für Flüssigkeitssimulationen und der Standardbibliothek *minim*, um den Sound zu analysieren. Das Einstellungsmenü ist ebenfalls mit Processing gestaltet, mit der Bibliothek *controlP5*. Man kann mit der Maus interaktiv in den Sketch eingreifen, indem man die durch den Rhythmus entstandenen Partikel ablenkt. Man kann auch ein Gerät oder Programm, dass das Protokoll Open Sound Control (OSC) unterstützt, anstelle der Maus über WLAN das Programm ansteuern. Als zusätzliches Feature gibt es einen 3D-Modus, in dem die Visualisation im 3-dimensionalen Raum betrachtet werden kann. Das Programm sollte folgende Themengebiete näher bringen: 3D (Kamera), Musik, GUI und Simulation. Das Projekt „Music Visual“ entstand in den Sommerferien 2011 für die Fachbereichsarbeit „Processing“ für das BG/BRG Carneri, Graz.

² Music Visual basiert auf folgenden Demo: http://memo.tv/files/memotv/msafluid_for_processing/index.html

Wichtige Bemerkungen zu den Erklärungen

1. Das Projekt wird Schritt für Schritt erklärt, wobei die Reihenfolge vom Schwierigkeitsgrad abhängt. Beginnend bei der Hauptfunktion *void setup()*, werden die einzelnen Befehle erklärt, sowie deren Hilfsklassen.
2. Die Variablen und Konstanten, die in diesem Projekt vorkommen, werden erst an den entsprechenden Stellen erklärt, das heißt die meisten Variablen können anfangs ignoriert werden.
3. Die einzelnen Codeabschnitte werden hier wie in der Processing-IDE farbig dargestellt, wobei Zeilennummern zur besseren Lesbarkeit eingefügt worden sind.

```
void setup() {  
    size(screen.width, screen.height, OPENGL);
```

4. Abschnitte, in denen Befehle vorkommen, die nicht zum Processing Standard oder zu den contributed libraries gehören, sondern Java Befehle sind, werden **orange** markiert.
5. Neue Befehle und Bezeichnungen werden **fett** geschrieben.
6. Verweise auf Artikel im Internet, sowie Funktionsnamen und Variablen sind *kursiv*.
7. *Ein Hinweis an Programmierer: Die primitiven Datentypen werden in dieser Arbeit großgeschrieben (Float statt float, Integer statt int). Gemeint sind allerdings die kleingeschriebenen Varianten und nicht die Java Wrapperklassen.*

Am Ende des praktischen Abschnitts sind Screenshots vom gestarteten Programm abgebildet und der Programmcode wird anhand von Diagrammen dargestellt. Alle in dieser Arbeit verwendeten Screenshots stammen von mir persönlich und wurden mit Hilfe des Tools „Code Formatter and Highlighter for processing und arduino“ von Anthony Mattox erstellt. Mehr über dieses Tool findet man auf der Homepage des Autors unter http://www.anthonymattox.com/code_formatter/

Grundlagen und grundsätzlicher Aufbau eines Sketches

Importe und Kommentare

Am Anfang eines Processing Sketches stehen die Bibliothekimporte. Importe können sowohl Standard-Bibliotheken (core libraries), sowie externe Bibliotheken (**contributed libraries**) sein. Welche Teile (**Pakete**) und Klassen der gesamten Bibliothek eingebunden werden, wird durch den **Punkt-Operator** bestimmt.

Beispiel 1:

- Hauptbibliothek: sojamo
 - Paket: drop

Beispiel 2:

- Hauptbibliothek: javax
 - Paket: media
 - Paket: OpenGL
 - Paket: GL

Eingebundene Bibliothek	Beschreibung, Herkunft
sojamo.drop	Drag and Drop, contributed
MSAFluid	Fluidsimulationen, contributed
processing.opengl	OpenGL Grafiken, contributed
javax.media.opengl.GL	OpenGL Grafiken, Java
ddf.minim.*	Sound, core
ddf.minim.analysis.*	Soundanalyse, core
javax.swing.JOptionPane	Dialog, Java
controlP5.*;	GUI (Graphical User Interface), contributed
peasy.*;	Kamera im 3D-Raum drehen, contributed
javax.swing.ImageIcon	Icon des Programmes, Java
java.awt.Cursor	Mauscursor, Java

Damit Processing weiß, dass es sich um einen Bibliotheksimport handelt, wird das Schlüsselwort **import** verwendet. Abgeschlossen werden die Anweisungen mit einem Strichpunkt (Semikolon). In den Zeilen 1 bis 8 sind einige Kommentare hinzugefügt worden. Es gibt zwei Arten von Kommentaren: Der einzeilige Kommentar beginnt mit zwei Slashes (//).

```
01  /*
02   * Music Visual
03   * Made by Alexander Pann; finished version 6.8.2011
04   * If you need help, press the help-button in the menu
05   * www.alexanderpann.at.tf
06  */
07
08 // imports
09 import sojamo.drop.*;
10 import msafluid.*;
11 import processing.opengl.*;
12 import javax.media.opengl.GL;
13 import ddf.minim.*;
14 import ddf.minim.analysis.*;
15 import javax.swing.JOptionPane;
16 import controlP5.*;
17 import peasy.*;
18 import javax.swing.ImageIcon;
19 import java.awt.Cursor;
```

Wenn man einen Kommentar über mehrere Zeilen verwenden möchte, kommt nur die zweite Variante in Frage, die auch in anderen Programmiersprachen oft verwendet wird. Am Beginn des Kommentars wird /* geschrieben, am Ende */. Kommentarzeichen dienen nur zum Kommentieren des Quellcodes, der Compiler ignoriert diese Zeilen. Es ist zu beachten, dass sich mehrzeilige Kommentare nicht ineinander verschachteln lassen:

```
01  /* /* Verschachteln ist nicht möglich */
02  /* // Das ist schon erlaubt */
03  // // Das auch
```

Variablen und Funktionen

Bevor die Hauptfunktion void setup() verwendet werden kann, sind Kenntnisse von Variablen und Funktionen notwendig.

Variablen sind Speicher für verschiedene Datentypen. Sie besitzen eine bestimmte Adresse und einen entsprechend großen, reservierten Speicherplatz. Da der Programmierer die Daten nicht mit Computerspeicheradressen ansprechen möchte, werden Aliase aus ganzen Wörtern verwendet.

Es gibt 2 Großgruppen von Datentypen, die **primitiven Datentypen** und die **Referenztypen**:

Liste der primitiven Datentypen

Abkürzung	Voller Name	Verwendung
int	Integer	Speicherung von Ganzzahlen
float	Float	Speicherung von KommaZahlen
double	Double	Erweiterung von Float mit höherer Genauigkeit
long	Long	Speicherung von großen Ganzzahlen
char	Character	Speicherung von einzelnen Zeichen
boolean	Boolean	Speicherung von Wahrheitswerten(wahr oder falsch)
color	Color	Speicherung von Farben (eigentlich ist color auch nur ein Integer)
byte	Byte	Speicherung von Bytes

Die Liste der Referenztypen ist frei erweiterbar, da der Programmierer selbst Typen erstellen

```
01 int antwort_auf_alles=42;
02 float pi=3.1415;
03 char c='c';
04
05 String begrüßung="Hallo";
```

kann mit Hilfe von sogenannten Klassen, die später genauer erklärt werden. Ein wichtiger Referenztyp ist **String**, in dem Texte gespeichert werden können.

Variablen werden auf folgende Weise deklariert und ein Wert gewiesen (**initialisiert**):

Auf der linken Seite wird der Datentyp angegeben, gefolgt vom Dateinamen, der nicht mit

```
01 int addieren(int summand,int summand) {
02     return summand+summand;
03 }
04
05 void test() {
06     println("Hallo Welt");
07 }
08
09 double pi() {
10     return 3,14159265;
11 }
12
13 println(addieren(1+1)); // 2
14 test();
15 println(pi()); // 3,14159265;
```

einer Zahl beginnen darf. Da Processing, wie Java, Unicodezeichen unterstützt, sind Umlaute und andere Sonderzeichen erlaubt, wie zum Beispiel in Zeile 7.

`setup()` ist also eine Funktion. Eine Funktion oder Methode ist ein Programmabschnitt, der mehrere Befehle enthält und mit einem eindeutigen Namen aufgerufen wird. Es können ihr Argumente übergeben werden, die im Block verarbeitet werden und sie kann beim Beenden dem Hauptprogramm einen Rückgabewert zurückgeben.

Wenn wir die Methode `addieren()` in Zeile 1 betrachten, kann folgendes gesagt werden:

- *int* ist der **Rückgabewert**, der mit dem Schlüsselwort **return** in Zeile 2 zurückgegeben wird.
- *addieren* ist der **Methodenname**.
- *int summand* ist jeweils eine Variable von Typ Integer. Die Argumente werden in runden Klammern hinter dem Funktionsnamen angegeben und werden mit Kommata getrennt. Wenn die Methode aufgerufen wird, muss die gleiche Anzahl an Argumenten angegeben werden wie beim Definieren der Methode, dass man **Deklaration** nennt. Der Datentyp muss auch dem definieren Datentyp entsprechen.
- Die geschwungenen Klammern öffnen und schließen den Programmblock, der in der Methode ausgeführt wird.
- Die in Zeile 1 vorkommenden Elemente nennt man zusammen **Signatur**.

Man kann die Funktion testen mit Hilfe der Methode `println()`. Sie akzeptiert alle primitiven Datentypen als Argument und gibt das Ergebnis auf der Konsole aus, die sich im untersten Abschnitt der Processing IDE befindet. Genauso wie die Methode in Zeile 5, ist ihr Rückgabewert **void**. Void bedeutet im Englischen „nichts“, das heißt sie hat keinen Rückgabewert. In der Zeile 9 ist eine weitere Beispiefunktion `pi()` die als Rückgabewert die Zahl Pi zurückgibt.

Mathematische Operatoren

Um gleich die Rechenfunktionen von Processing zu demonstrieren, werden mit dem **+**-**Operator** die Werte zusammengerechnet. Es werden folgende arithmetische Operatoren unterstützt: **Addition(+)**, **Subtraktion(-)**, **Multiplikation (*)**, **Division (/)**, und **Modulo (%)**. Es gilt auch Punkt vor Strich und es können Klammern gesetzt werden.

Beispiel:

- $1 + 2 = 3$
- $1 + 2 * 3 = 7$
- $(1 + 2) * 3 = 9$
- $11 \% 2 = 1$ // Der Module-Operator berechnet den Rest einer Division
- $1 - 2 + 3 * 4 / 5 = 1.4$

Typenumwandlung

Beim Dividieren ist zu beachten, dass mindestens eine der beiden Zahlen von Typ Float ist, da sonst gerundet wird.

Beispiel:

```

01 int a=1;
02 int b=2;
03
04 println(a/b);    // = 0
05 println(1/2);   // = 0
06
07 // jedoch
08 println(1/2.0); // = 0.5
09 println(1/2f);  // = 0.5
10
11 // andere Lösung
12 int aa=1;
13 float bb=2;
14
15 println(a/b); // = 0.5;
```

In Zeile 9 wurde eine andere Möglichkeit ausgenutzt, um ein richtiges Ergebnis zu bekommen. Die Zahl 2 wurde von einem Integer in ein Float umgewandelt (**gecastet**), indem an die Zahl ein f für Float angefügt wurde.

Wichtige Casts:

Nach Float	f
Nach Double	d

Es gibt noch eine zwei weitere Varianten:

```
01 float pi=3.1415;
02 int pi_gerundet=(int) pi;
03 int pi_gerundet2;
04
05 void setup() {
06   pi_gerundet2=int(pi);
07   println(pi_gerundet);
08   println(pi_gerundet2);
09 }
```

In der Zeile 2 wurde wieder ein cast durchgeführt, dabei wurde ein Float in ein Integer verwandelt. Um diese Funktion zu verwenden, muss der Datentyp, in den umgewandelt werden soll in runde Klammer vor den Ausdruck geschrieben werden. Sollte es für Processing nicht klar sein, welcher Wert umgewandelt werden soll, muss der Ausdruck auch in runden Klammern stehen. In diesem Beispiel ist das nicht nötig, da es nur eine Variable gibt. Die zweite Variante ist mit der Methode **int()** mit der man das Argument in ein Integer umwandeln kann. Sollte die Typen nicht „kompatibel“ sein, gibt es eine Fehlermeldung.

Void setup

Die Funktion **void setup()** ist einer der wichtigsten Methoden im Sketch. In ihr werden Variablen deklariert und initialisiert, die Fenstergröße und der Renderer festgelegt und Funktionen aufgerufen. Dies geschieht nur ein einziges Mal im ganzen Programm.

Void draw

Diese Funktion ist die Hauptfunktion, in der sich sozusagen alles abspielt. Sie wird bis zu 60 Mal in der Sekunde aufgerufen, je nachdem wie die Framerate gesetzt ist. Sie kann mit dem Befehl **frameRate(int)** gesetzt werden, *int* kann durch einen Wert von 1 bis mindestens 60 ersetzt werden. Die Standardrate ist 60, für flüssige Animationen werden 24 bis 25 Frames benötigt.

Anmerkung: Die Rolle der englischen Sprache in der Programmierung

In der Programmierung hat sich die englische Sprache durchgesetzt, Methoden werden mit englischen Bezeichnungen versehen und auch die Variablen werden in Englisch geschrieben. Die einzige Ausnahme sind oft Kommentare, die in der jeweiligen Landessprache geschrieben werden. Aus diesem Grund werden alle weiteren Beispiele nur mehr in Englisch geschrieben.

Kontrollstrukturen

In einem Programm werden meistens die einzelnen Funktionen nicht immer in der gleichen Reihenfolge und Häufigkeit aufgerufen. Dafür gibt es die sogenannten Kontrollstrukturen.

If-Else-Anweisung

Bei der If-Else-Anweisung wird – abhängig von einer Bedingung – entweder der eine oder der andere Programmblock ausgeführt. Der Ausdruck besteht aus mehreren Bedingungen, bei dem folgende Operatoren gültig sind:

Zeichen	Bedeutung	Beispiel
<	weniger als	1 < 2 -> true
<=	weniger oder gleich wie	3 <= 4 - 2 -> false
>	größer als	1 > 1 -> false
>=	größer oder gleich wie	2 == 10 / 5-> true
==*	ist gleich	2 == 3 ->false
!=	ist nicht gleich	5 != 10 ->true
&&	und	1 == 1 && 2 < 3 ->true
	oder	1 > 2 2 == 3 -> false

* Es wurde „==“ und nicht „=” gewählt, da dieser Operator bereits für Zuweisungen verwendet wird.

Beispiel:

```

01 if(expression) {
02 // commands
03 } else {
04 // commands
05 }
06 // example:
07 String apple_color="red";
08 if(apple_color=="red") {
09 println("The apple is red");
10 } else {
11 println("The apple isn't red");
12 }
```

Es gibt die gleiche Anweisung auch nur mit dem If-Teil:

```
01 boolean ok=true;
02
03 if(ok) {
04
05   println("perfekt");
06
07 }
```

Auffällig ist der Ausdruck in Zeile 3, der nur aus einer Variable besteht. Wenn ein boolescher Wert, also true oder false in einem Ausdruck verwendet wird, ist kein Operator mehr nötig, da schon eine Bedingung gegeben ist. Um weitere Bedingungen zu überprüfen, kann die Struktur **else if(Bedingung)** hinzugefügt werden.

Im folgenden Beispiel beachte man die fehlenden geschwungenen Klammern bei allen Wochentagen außer beim Dienstag. Sie sind nicht nötig, da nach der Bedingung nur ein Befehl ausgeführt wird. In den Zeilen 5 bis 8 kommen jedoch zwei *println()* Befehle vor und deshalb sind die geschwungenen Klammern notwendig. Würde man sie weglassen, würde der Compiler die If-Else-Anweisung nicht verstehen, weil er in Zeile 7 plötzlich einen neuen Befehl findet, obwohl die Anweisung noch nicht zu Ende ist. Das Ergebnis wäre eine Fehlermeldung.

```
01 String day_of_week="Friday";
02
03 if (day_of_week=="Monday")
04   println("I miss the weekend.");
05 else if (day_of_week=="Tuesday") {
06   println("I love tuesdays.");
07   println("This is my favourite day.");
08 }
09 else if (day_of_week=="Wednesday")
10   println("I have to work so long at this day.");
11 else if (day_of_week=="Thursday")
12   println("I am waiting for the weekend");
13 else
14   println("End of the week. Let's go to a party!");
```

Fehlermeldungen

Fehlermeldungen gehören zu jeder Programmiersprache, so auch zu Processing.

Grundsätzlich wird zwischen Exceptions und Errors unterschieden. Exceptions sind Ausnahmefehler, die während der Laufzeit des Programmes auftreten. Sie werden meist im weiteren Programmverlauf behandelt und der Absturz des Programmes wird verhindert.

Errors hingegen sind schwere Fehler, die einen sofortigen Absturz mit sich ziehen und nicht zur Laufzeit behandelt werden können. Zusätzlich erkennt der Präprozessor von Processing einige weitere Syntaxfehler.

Duplicate local variable x

```
01 int a = 1;  
02 int a = 2;
```

Lokale Variablen dürfen nicht zweimal deklariert werden, ansonsten erscheint diese Meldung. Abhilfe findet man, indem die beiden Variablen in einem anderen Sichtbarkeitsbereich deklariert werden, also zum Beispiel eine globale und eine lokale Variable definiert wird.

ArithException: / by zero

```
01 println(1 / 0);
```

Divisionen durch 0 sind in Processing nicht erlaubt und führen zu einer Fehlermeldung.

Missing a x() to go with that y()

```
01 translate(10,10);  
02 ellipse(0,0,50,50);  
03 popMatrix();
```

Viele Funktionen werden paarweise verwendet. Ein Aufruf der Endfunktion ohne Anfangsfunktion führt zur einer Exception im Animation Thread. Dies gilt zum Beispiel für die Paare pushMatrix()-popMatrix(),pushStyle()-popStyle() und beginShape()-endShape();
pushMatrix() cannot use push more than 32 times

```
01 for(int i=0;i<100;i++) pushMatrix();
```

Die Anfangsfunktion dieser Paare legt auf Daten auf einem Stapel, wie zum Beispiel pushMatrix(), die die aktuelle Matrix auf den Matrixenstapel legt, damit popMatrix() sie später wieder herunternehmen kann. Standardmäßig können nur 32 Objekte auf diese Stapeln gelegt werden und daher erfolgt zum Beispiel bei pushMatrix() die oben erwähnte Ausnahme. Der Java kongruente Error heißt StackOverflowError.

OutOfMemoryException

Da Processing objektorientiert ist, müssen die einzelnen Objekte im Arbeitsspeicher zwischengespeichert werden. Wenn es keinen freien Arbeitsspeicher mehr gibt, bricht das Programm ab und zeigt diese Exception.

NullPointerException

```
01 int[] ints = null;
02 println(ints[0]);
```

Diese Meldung betrifft "nur" die Objektvariablen und nicht die primitiven Datentypen. Erste re zeigen normalerweise auf einem Adresse im Arbeitsspeicher, die den Ort der gespeicherten Daten beinhaltet. Sollte dies nicht der Fall sein, liegt eine NullPointerException vor. Dieser Fehler kann sehr viele Ursachen haben. Einige mögliche Gründe sind:

- eine Objektvariable wurde nicht initialisiert
- eine Datei wurde nicht gefunden oder konnte nicht geladen werden
- eine Inputquelle liefert keine Daten
- eine Datei wird ausgelesen und das Dateiende wurde erreicht.

UnsupportedClassVersionError

Dieser Fehler tritt manchmal im Bezug auf contributed libraries oder tools auf. Wenn diese in einer anderen Version von Java geschrieben wurden, die von Processing nicht unterstützt wird, entsteht eine Inkompatibilität, die von dieser Meldung ausgedrückt wird.

Unexpected token

```
01 println("test")
02 int a = 0;
```

Processing erwartet bei dieser Exception an der auftretenden Stelle ein anderes Zeichen. Oft fehlt ein Zeichen, in diesem Fall ein Semikolon. Fehlende Strichpunkte werden meistens mit "Syntax error, maybe a missing semicolon?" und fehlende runde Klammern mit "Syntax er-

ror, maybe a missing right parenthesis?" gemeldet. Ob und wie der Fehler angezeigt wird, hängt von der Position im Quellcode und der Art des falschen/fehlenden Zeichens ab. Speziell bei Quellcode, die sich über mehrere Tabs erstrecken, erschweren die Anzeige des Fehlers. Processing deutet dann meistens darauf hin, dass der Fehler sich im 2.Tab befindet oder zeigt die Zeile überhaupt nicht an. Wenn der Fehler genau lokalisiert werden kann, wird die betroffene Zeile einem hellen Orange markiert. Die Ursache befindet sich allerdings oft in der vorherigen Zeile oder noch weiter vorher und daher sollte der Code von der betroffenen Stelle aufwärts untersucht werden.

Cannot convert from x to y

```
01 int a = "Hello world";
```

Konvertierungsfehler werden meist vor dem Start des Sketches gefunden.

IndexOutOfBoundsException

```
01 int numbers = { 1,6,3,5,3 };
02 println(numbers.length); // 5
03 println(numbers[5]);
```

Ein sehr häufiger Fehler ist, dass ein Arrayindex außerhalb der Länge des Arrays aufgerufen wird. Speziell bei Zählschleifen passiert es schnell, dass man entweder beim Index 1 beginnt und bei n. Element aufhört. das Letzte Element hat allerdings den Index n-1.

Infinity

```
01 println(pow(10,100));
```

Wenn ein berechnetes Ergebnis über die Kapazitäten von Processing reicht, wird "Infinity" zurückgegeben. Es ist zwar nicht wirklich ein Fehler, muss aber mit einer Funktionen der jeweilen Klasse des berechneten Datentyps überprüft werden. Zum Beispiel: Double.isInfinite(value), Float.isInfinite(value)

NaN

```
01 println(sqrt(-1));
```

"Not a number" ähnelt Infinity, sie ist zum Beispiel das Resultat von der Wurzel aus -1 (= komplexe Zahl: i). Sie wird auch mit einer entsprechenden Klassenfunktion wie Double.isNaN(a) festgestellt. Die zur Erkennung dieser speziellen Werte nötigen Funktionen gehören zum Standart von Java.

The method x() in the type y is not applicable for the arguments(...)

```
01 println("first argument","second argument?");
```

Wenn Methoden mit einer ungültigen Anzahl an Argumenten aufgerufen wird, wird vor der Ausführung noch darauf hingewiesen. In Java entspricht dies der `IllegalArgumentException`, die unter bestimmten Umständen auch in Processing geworfen werden kann.

You need to use "Import Library" to add `processing.opengl.PGraphicsOpenGL` to your sketch.

```
01 // missing import
02
03 void setup() {
04   size(100,100,OPENGL);
05 }
```

Obwohl OpenGL einer der beiden Hauptrenderer ab Version 2.0 ist, muss er trotzdem seit den ersten Versionen schon extra als Bibliothek importiert werden.

```
01 import processing.opengl.*;
02
03 void setup() {
04   size(100,100,OPENGL);
05 }
```

It looks like you're mixing "active" and "static" modes.

```
01 void active() {
02 }
03 println(active());
04
```

Wenn Processing ohne Setup und Draw im aktiven Modus verwendet wird, dürfen auch keine eigenen Funktionen definiert werden, da sonst der aktive mit dem statischen Modus gemischt wird. Die ausgelöste Exception ist eine Processing spezifische `SketchException`.

Switch-Anweisung

Da mit wachsender Anzahl an Bedingungen der Code unübersichtlich wird, gibt es eine andere, übersichtlichere Methode für solche Fälle. Der Befehl ist so aufgebaut:

```
01 switch(expression) {  
02  
03     case a1:  
04         // commands  
05         break;  
06     case a1:  
07         // commands  
08         break;  
09     default:  
10         // commands  
11     }  
12  
13     String day_of_week="Friday";  
14  
15     switch(day_of_week) {  
16  
17         case "Monday":  
18             println("I miss the weekend.");  
19             break;  
20  
21         case "Tuesday":  
22             println("I love tuesdays.");  
23             println("This is my favourite day.");  
24             break;  
25  
26         case "Wednesday":  
27             println("I have to work so long at this day.");  
28             break;  
29  
30         case "Thursday":  
31             println("I am waiting for the weekend");  
32             break;  
33  
34         case "Friday":  
35             println("End of the week. Let's go to a party!");  
36             break;  
37  
38     default:  
39         println("This isn't a day of the week!");  
40     }  
41 }
```

In Zeile 13 bis 41 ist die bereits beschriebene If-Else-Abfrage nochmals dargestellt. Die einzelnen Fälle werden mit einem **case** eingeleitet, gefolgt vom Wert und einem **Doppeltpunkt**. Darauf können beliebig viele Befehle folgen, bis zum nächsten case-Schlüsselwort. Um die Überprüfung abzubrechen, erfolgt am Ende das Schlüsselwort **break** und ein Semikolon. Eine fortgeschrittene Technik ist es, dieses Wort wegzulassen um einen Effekt zu erzeugen, der in der Fachsprache **fall through** genannt wird:

```
01 // fall through
02
03 int age=14;
04
05 switch(age) {
06 case 13:
07 case 14:
08 case 15:
09 case 16:
10 case 17:
11 case 18:
12 case 19:
13 println("Your a teenager!");
14 break;
15 default:
16 println("Your aren't a teenager!");
17 }
```

While-Schleife

Schleifen werden benutzt um Codestücke mehrmals zu wiederholen. Sie funktionieren vom Prinzip ähnlich wie anderen Kontrollstrukturen. Die While-Schleife ähnelt vom Aufbau der If-Schleife.

```
01 int i=1;
02
03 while(i<10) {
04   println(i*i);
05   i++;
06 }
07
08 while(true) {
09   println("This is a infinite loop, do not start the program!");
10 }
11
12 int j=1;
13
14 while(true) {
15   println(j*j);
16   j++;
17   if(j==10)
18     break;
19 }
```

In der Zeile 8 befindet sich eine Endlosschleife, die niemals so programmiert werden darf. Wenn das Programm ausgeführt werden würde, würde sich Processing „aufhängen“, weil es mit der Schleife überfordert ist. Sollte dieser Programmierfehler doch passieren, so schnell wie möglich den viereckigen Stopp-Button bzw. die ESC-Taste drücken. In den Zeilen 12 bis

18 werden das erste Beispiel und der fehlerhafte Teil zu einer neuen Technik zusammengebracht. Eine Endlosschleife lässt sich nämlich mit dem Schlüsselwort **break** abbrechen, das bereits von der Switch-Anweisung bekannt ist. In diesem Codeabschnitt ist noch eine Neuerung, nämlich, dass eine Integer Variable mit zwei Pluszeichen versehen ist. Im nächsten Abschnitt sollen nun Zuweisungen erklärt werden.

Zuweisungen

Es ist in den vorherigen Kapiteln schon oft vorgekommen, dass Variablen mit Hilfe des Gleichheitszeichens ein Wert zugewiesen worden ist. Es gibt aber noch viele andere Möglichkeiten:

Operator	Funktion	Beispiel
+=	Zuweisung mit Addition	a += 5
-=	Zuweisung mit Subtraktion	s -= 10
*=	Zuweisung mit Multiplikation	m *= 100
/=	Zuweisung mit Division	d /= 50f
%=	Zuweisung mit Modulo	mo %=2
x ++	Ausgabe, danach +1	
x --	Ausgabe, danach -1	
++ x	+1, danach Ausgabe	
-- x	-1, danach Ausgabe	

For-Schleife

Diese Schleife hat die gleiche Aufgabe wie die While-Schleife, besteht aber aus 3 Teilen:

```

01 for(int x=y;x<z;x++) {
02     // commands
03 }
04
05 for(int i=1;i<=100;i++)
06     println("The square of "+i+" is "+i*i);
07
08 for(int x=0,y=10;x<10;x++,y--)
09     println("X:"+x+" Y:"+y);
10
11

```

1. Ein oder mehrere Variablen werden initialisiert
2. Eine Bedingung wird aufgestellt

3. Einer oder mehreren in Punkt 1 verwendeten Variablen wird ein neuer Wert zugewiesen, meistens wird sie um 1 erhöht (**inkrementiert**) oder um 1 erniedrigt (**dekrementiert**).

In Zeile 10 werden zwei Variablen, X und Y verwendet, die einzelnen Befehle werden mit einem **Komma**, die 3 Abschnitte mit einem **Semikolon** getrennt.

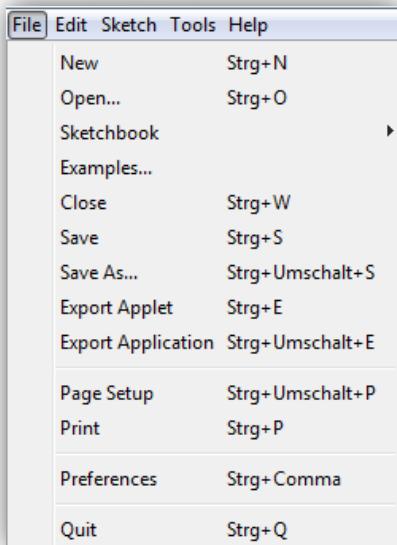
Nun sind alle Grundlagen erklärt, die für das Hauptprojekt notwendig sind. Nichts desto trotz erfolgen einige Erklärungen wie Klassen erst später, da sie etwas komplizierter sind.

Eine Übersicht aller Befehle kann unter <http://processing.org/reference/> gefunden werden. Es gibt auch einige Tutorials in Englisch über folgende Themen:

- Introduction
- Overview
- Coordinate Systems and Shapes
- Color
- Objects
- Two-Dimensional Arrays
- Images and Pixels
- Curves
- Strings and draw()ing Text
- 2D Transformations
- Trigonometry Primer I
- PVector
- Anatomy of a Program
- Processing in Eclipse

Processing IDE

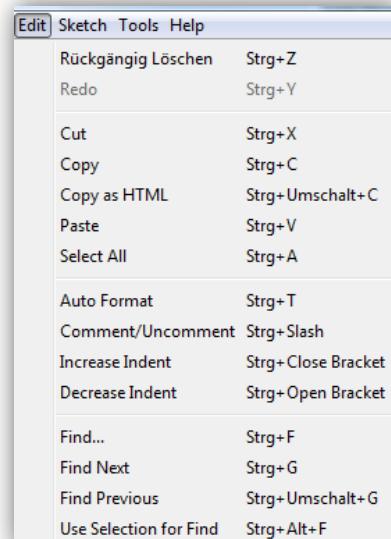
Bevor man beginnt, mit Processing zu programmieren, sollte man die Umgebung erkunden, in der man sich in nächster Zeit aufhalten wird. Die Processing IDE ist im Vergleich zu anderen Programmen mit recht wenigen Menüs ausgestattet, die trotzdem die wichtigsten Punkte enthalten, um erfolgreich ein Programm in Processing zu schreiben. Im Menü **File** befinden sich viele Einträge, die man gut von Office-Anwendungen kennt: **New**, **Open**, **Close**, **Save**, **Save As**, **Page setup()**, **Print** und **Preference**. Man kann auch Projekte, die man unter „Eigene Dateien“ im Processing Ordner gespeichert hat unter **Sketchbook** wieder öffnen. Wie der Titel verrät, ist das der persönliche Ort, wo man Sketche speichern und wieder laden kann. Wenn man den Eintrag **Examples** anklickt, öffnen sich Untermenüs mit Beispielprojek-



ten, angefangen vom Zeichnen eines einfachen Kreises bis hin zu einem 3-dimensionalen Feuertunnel. Alle hier aufgelisteten Beispiele sind auch auf der offiziellen Homepage zu finden. Ab Version 1.5 befinden sich die Beispiele in einem externen Fenster. Mit **Export Applet** kann man den Sketch als Applet exportieren, mit **Export Application** als Anwendung. Es erscheint ein kleines Fenster in dem man die gewünschten Betriebssysteme auswählen und den Fullscreenmodus aktivieren kann; wenn nötig kann auch ein Stopp-Button eingefügt werden. **Preference** ist das Einstellungsfenster, weitere Einstellungen können in der Datei **Preferences.txt** gemacht werden, deren Pfad am Ende des Dialogs angezeigt wird. Eine Einstellung, die bei größeren Berechnungen

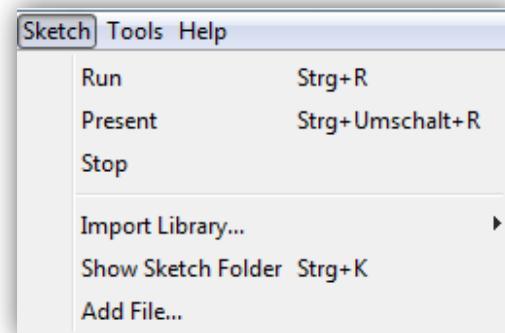
wichtig sein kann, ist **increase maximum available memory to**, weil der Speicher bei der Ausführung sehr schnell ausgehen kann. Auf Rechnern mit Windows 32 Bit lässt sich der Speicher auf ungefähr 1.5 GB ausweiten, bei Windows 64 Bit und mindestens 3GB freiem Speicher gibt es kein Limit. Mit Linux sind ungefähr 2 GB möglich, wobei man sich auf diese Angaben hier nicht völlig verlassen kann. Standardmäßig stehen Processing 256 MB zur Verfügung.

Im Menü **Edit** gibt es wieder einige Standardeinträge: **Rückgängig, Redo, Cut, Copy, Paste, Select All**. Zusätzlich zur Kopierfunktion gibt es eine erweiterte Funktion, mit der man den ausgewählten Bereich als HTML kopieren kann, um zum Beispiel den farbigen Quellcode im Internet zu dokumentieren. Eine wichtige Funktion ist **Auto Format**, bei der man sich den Tastenkürzel [Strg] +[T] merken sollte. Da speziell Anfänger von den vielen geschwungenen Klammern oft verwirrt sind, hilft diese Funktion den Code einzurücken, um ihn dadurch übersichtlicher und besser verständlich zu machen. Es kann bei einem schweren Syntaxfehler leider dazu kommen, dass die Einrückfunktion eine Fehlermeldung ausgibt und dann der Fehler nur schwer auffindbar ist. Weitere nützliche Erleichterungen sind die **Comment/Uncomment** Funktion, mit der Kommentare erzeugt und entfernt werden können, sowie **Increase/Decrease Indent** um die Einrückungen manuell zu ändern. Bei Betätigung vom Autoformat gehen sie allerdings wieder verloren.



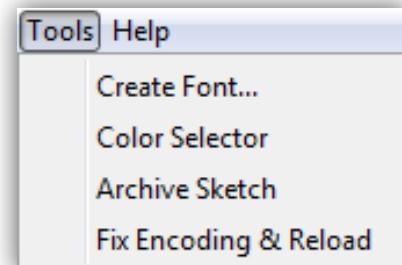
Der letzte Teil besteht aus Suchfunktionen, mit denen auch einzelne Zeilen ersetzt werden können: **Find**, **Find Next**, **Find Previous**, **Use Selection for Find**. Um einen Teil zu ersetzen, wird im *Find* Dialog, der zu suchende String angegeben, sowie in das nächste Textfeld der Ersatzstring. Es gibt weitere Möglichkeiten, wie zum Beispiel, dass man Groß- und Kleinschreibung ignoriert oder alle Suchergebnisse ersetzt. Bei der Suchfunktion sind reguläre Ausdrücke, also Platzhalter wie *, ?, + nicht erlaubt.

Im Menü **Sketch** befinden sich die wichtigsten Einträge. Mit **Run** startet man den Sketch, den Fullscreenmodus kann man mit **Present** erreichen. Alternativ kann man auch den dreieckigen Knopf in der Toolbar verwenden, der sich direkt unter dem Menü befindet. Hier befinden sich die am meisten verwendeten Aktionen zusammengefasst in einer Leiste. Um allerdings effektiv zu programmieren, sollte man die Shortcuts verwenden, die zum Starten des Sketches [Strg] + [R] und [Strg] + [Shift] + [R] lauten. Beenden kann man das eigene Programm mit der Escape [ESC]-Taste, oder über den Menüeintrag **Stop**. Da verwendete Bibliotheken teilweise einen nicht ganz so leicht merkbaren Namen tragen und man auch mal vergisst, welche Bibliotheken man heruntergeladen hat, kann man mit **Import Library** die import-Anweisung für die jeweilige Bibliothek automatisch generieren lassen. Contributed libraries findet man auf der Homepage von Processing und müssen in den Ordner **libraries** verschoben werden, der sich im Processing Installationsverzeichnis befindet, gefolgt von einem Neustart von Processing. Seit Version 1.5 befindet sich dieser unter „Installationsverzeichnis/modes/java/libraries“. Die Bibliotheken bestehen in der Regel aus vier Ordnern: **examples**, **library**, **reference** und **src**. Die für Processing wichtigen Dateien befinden sich unter *library* und können per Drag and Drop auch direkt in den Sketch gezogen werden. Dabei wird im Ordner des Sketches ein Ordner **code** erstellt, in dem sich nun die Dateien befinden. Diese Variante gilt jedoch als veraltet und sollte nur mehr bei Bibliotheken verwendet werden, die nicht für Processing sondern für Java konzipiert sind, da sie nur auf diese Weise eingebunden werden können. Auf gleiche Weise lassen sich auch Mediendateien hinzufügen, wobei hier ein Ordner **data** erstellt wird. Um diese beiden Ordner leicht zu erreichen, gibt es den Menüeintrag **Show Sketch Folder**. Man kann Dateien auch alternativ mit **Add File** über einen Dateibrowser hinzufügen.



Das Menü **Tools** beinhaltet einige nützliche Werkzeuge, die die Arbeit mit Processing erleichtern. Es gibt auch einige weitere „contributed tools“, die herunterladen werden können. Sie

müssen dem Ordner **tools** im Installationsverzeichnis hinzugefügt werden. Processing verwendet eigene Schriftarten, deren Endung „.vlw“ ist. Durch den Eintrag **Create Font** erscheint ein Dialog, bei dem man eine Schriftart wählen kann. Es lässt sich auch die Schriftgröße einstellen, wobei man hier aufpassen sollte, dass die Schrift nicht zu klein ist. Die Schriftgröße kann nämlich mit dem Befehl **fontSize()** später noch im Programm geändert werden. Wenn die geänderte Schriftgröße größer ist als die Schriftart selbst, wirkt sie pixelig. Um eine Schrift zu verwenden, lädt man sie als Erstes in eine Variable der Klasse **PFont** mit dem Befehl **loadFont()**. Der erste Parameter ist der Name der Schrift, wobei man folgende Dinge beachten muss: wenn man den Namen vergisst, wählt man am besten aus dem Menü *Show Sketch Folder* und sucht die Datei im Ordner data. Es muss auch die Dateiendung angeben werden, also zum Beispiel „Algerian.vlw“. Möchte man die Standardschriftart verwenden, lässt man den String leer. Als zweites Argument wird die Schriftgröße angegeben. Danach wird der Methode **textFont()** das PFont Objekt übergeben, man kann aber auch direkt loadFont in dieser Funktion aufrufen. Um einen Text am Bildschirm ausgeben zu können, wird der Befehl **text()** benötigt. Er akzeptiert als Parameter einen String, sowie die X- und Y-Koordinate. Bei Verwendung des OpenGL-Renderer kommt es oft zu Problemen.



Was tun bei „unlösbar“ Problemen und Bugs, wie erhält man Support?

Große Unterstützung bekommt man im Forum unter <http://forum.processing.org/>, die Troubleshooting-Seite im Wiki unter <http://wiki.processing.org/w/> hilft bei den gängigsten Problemen. Man sollte sich auch die anderen Seiten im Wiki anschauen, da auch nützliche Codesnippets und allgemeine Informationen zur Hardware etc. beschrieben werden. Bugs kann man in der Datenbank suchen unter <http://code.google.com/p/processing/issues/list>, wobei in den Kommentaren oft Lösungsmöglichkeiten (Workarounds) beschrieben werden. Die wichtigsten Fehler den einzelnen Processing Versionen können in einem Textdokument unter <http://processing.googlecode.com/svn/trunk/processing/build/shared/revisions.txt> nachgelesen werden.

Fortsetzung des Kapitels „Processing IDE“

Um zum vorherigen Thema zurückgelangen, werden die anderen beiden Tools erklärt. Das Tool **Archive Sketch** ermöglicht es den Sketch Ordner samt den darin enthaltenen Ordnern in ein Zip-Archiv zu speichern. Dadurch werden die Daten komprimiert, und der benötigte Speicherplatz verringert sich. Ein weiterer Vorteil ist, dass sich in diesem Format Daten leicht ins Internet hochladen lassen und wieder heruntergeladen werden können. Wenn man die ganze Ordnerstruktur ins Internet stellen würde, müsste man aus jeden einzelnen Ordner die Dateien am Desktop abspeichern und die „inneren“ Ordner manuell erstellen und die dementsprechenden Dateien hineinziehen müssen. Das letzte Tool ist in unseren Breiten nicht so wichtig, da es normalerweise keine Probleme mit Sonderzeichen gibt. In anderen Ländern kann dieser Menüeintrag gewählt werden, um die Kodierung des Sketches zu ändern, damit alle Sonderzeichen richtig angezeigt werden.

Das Menü **Help** enthält einige bereits genannte Links. Es gibt folgende Links:

About Processing	Zeigt das Ladebild (Splashscreen) noch einmal an
Environment	Informationen zur Processing IDE
Reference	Befehlsreferenz mit allen Befehlen
Getting Started	Einführungsdokument
Troubleshooting	Dokument, in dem die häufigen Bugs und Fehler aufgelistet werden
Frequently Asked Questions	Dokument, in dem die am häufigsten gestellten Fragen beantworten werden
Visit processing.org	Link zur offiziellen Homepage

Processing kann verwendet werden, um Java Applets und Anwendungen zu erzeugen, seit Version 1.5 kann man auch Anwendungen für die Android Plattform schreiben. Die einzige Voraussetzung ist, dass man das Android Software Development Kit installiert hat, das man kostenlos aus dem Internet laden kann. Es gibt auch einige neue Befehle die im Wiki unter <http://wiki.processing.org/w/Android> nachgelesen werden können. Um zwischen den beiden Varianten hin und her zu schalten, betätigt man den Button mit der Beschriftung **Standard**. Es erscheint ein Dropdownmenü, indem man zwischen Standard und Android wechselt kann. Unter diesem Button befindet sich ein Pfeil der nach rechts zeigt. Drückt man diesen Button, hat man die Möglichkeit, Tabs zu bearbeiten. Processing Sketche lassen sich in einzelne Tabs aufteilen, um sie übersichtlicher zu machen. Die Befehlszeilen in den anderen Tabs verhalten sich nämlich gleich, als würden sie direkt in den Haupttab am Ende eingefügt werden. Hierbei ist der Tab gemeint, der den gleichen Namen wie der Sketch hat. Die Funk-

tionen `setup()` und `draw()` können in jedem Tab notiert werden, sie dürfen jedoch nur einmal im ganzen Sketch vorkommen. Um neue Tabs hinzuzufügen, wird der Pfeil gedrückt und es erscheint ein Popupmenü. Nun können neue Tabs hinzugefügt und alte Tabs gelöscht, sowie unbenannt werden. Bei älteren Versionen von Processing wurden nicht einzelne Tabs mit **Delete** gelöscht, sondern der gesamte Sketch. Es lässt sich in diesem Menü auch zum vorherigen und nächsten Tab wechseln.

Verschiedene Tipps zur IDE

- Das Paragraphenzeichen (§) hinter dem Namen eines Tabs bedeutet, dass dieser seit der letzten Änderung nicht gespeichert wurde.
- In der linken, unteren Ecke wird die aktuelle Zeilennummer angezeigt. Wenn Text markiert wird, wird die Nummer der Anfangszeile bis zur Endzeile angezeigt.
- In der grauen Box über der Konsole gibt Processing Meldungen aus. Diese sind nur kopierbar, indem man die Ausgabe auf der Konsole sucht (meistens ganz oben) und von hier dort den Text kopiert.
- Die Höhe der Konsole lässt sich verändern, man muss dafür nur über der Infobox zu ziehen beginnen, der Cursor verändert sich dabei.
- Leerzeichen sind im Dateinamen nicht erlaubt, Processing wandelt sie automatisch in Unterstriche (_) um.

Music Visual – Zeile 52 bis 66

Der erste Aufruf in diesem Projekt erfolgt in Zeile 51, nämlich der `setup()` Funktion. Der erste Befehl, der fast immer in dieser Funktion aufgerufen wird, ist die Methode `size()`. Ihr müssen zwei fixe Argumente, nämlich die Höhe und Breite des Fensters mitgegeben werden, ein drittes, zusätzlich, auch **optionales** Argument genannt, ist ein Renderer, der mit Hilfe ein paar konstanter Variablen angegeben werden kann. Da dieses Programm im FullScreen-Modus laufen soll, werden zwei spezielle Variablen angeben, nämlich `screen.width` und `screen.height`. In diesen Integer Variablen befindet sich die Breite und Höhe des Bildschirms oder mit anderen Worten die maximale Größe des Fensters. Es ist auch der dritte **Parameter** angeben, der die folgenden Werte annehmen kann:

Name	Beschreibung
JAVA2D	Er ist der Standard-Renderer. Dieser Renderer unterstützt 2-dimensionales Zeichnen und bietet insgesamt eine hohe Bildqualität, ist im Allgemeinen aber langsamer als P2D
P2D	Ein schneller 2D-Renderer, der am besten

	mit Pixeldaten funktioniert, jedoch nicht so fehlerfrei wie der Java2D-Renderer ist
P3D	Ein schneller 3D-Renderer für das Internet. Er opfert Render-Qualität für schnelles 3D-Zeichnen
OPENGL	Sehr schneller 3D Grafik-Renderer der OpenGL-kompatible Grafikhardware benutzt, sollte sie verfügbar sein.
PDF	Erzeugt PDF-Dokumente

In Music Visual wird der OpenGL Renderer verwendet, da die MSAFluid Bibliothek diesen vorschreibt, da die hohe Geschwindigkeit des Renderers notwendig ist. In Zeile 53 und 54 wird versucht, den Renderer mit einem sogenannten **Rendering Hint** zu beeinflussen, dabei soll die Kantenglättung (Antialiasing) auf das Vierfache erhöht werden. In Zeile 52 wird der Hintergrund gesetzt. Farben werden in Processing Standardmäßig mit der RGB (Red, Green, Blue)-Palette angegeben. Es kann auch mit Hilfe von **colorMode(HSB)** die Palette auf HSB (Hue, Saturation, Brightness), also die Komponenten Farbton, Sättigung und Helligkeit, umgestellt werden. Sollte nur ein Grauton benötigt werden, reicht ein Argument von Typ Integer, mit einem Wert von 0 bis 255. Es kann auch ein Hexadezimalcode, wie er bei den Webfarben üblich ist, eingesetzt werden.

```
void setup() {
    size(screen.width, screen.height, OPENGL);
    background(0);
    if (ANTI_ALIASING_4X)
        hint(ENABLE_OPENGL_4X_SMOOTH);
```

In der Processing IDE unter Tools-> Color selector können Farben ausgewählt und der Code kopiert werden. Da der Hintergrund schwarz sein soll, wird dem Befehl **background** als Parameter eine 0 übergeben. In den Zeilen 58 bis 62 werden einige Einstellungen am Fenster geändert, in Java **Frame** genannt, die mit Processing nicht möglich sind:

- Zeile 58: Das Fenster wird in der linken oberen positioniert.
- Zeile 59: Das Fenster bekommt den Titel, der in der Konstante APP_TITLE definiert ist.
- Zeile 60: Ein neues Icon wird mit Hilfe der Processing Funktion **loadBytes()** vom Pfad geladen, der in der Konstante APP_ICON steht. Danach wird ein neues ImageIcon für das Objekt titlebaricon erstellt.

- Zeile 61: Das gerade erstellte Icon wird dem Frame zugewiesen.
- Zeile 62: Der Mauscursor passte für diese Anwendung nicht, also wurde er durch einen anderen Standardcursor ausgetauscht. Der Cursor wird wieder im Frameobjekt neu gesetzt und der neue Cursor aus der Java Klasse Cursor geladen.

Processing hat auf den Cursor zumindest einen eingeschränkten Zugriff, er kann mit **cursor()** und **noCursor()** ein- und ausgeblendet werden.

In Zeile 64 wird die Kamera für den 3D-Modus erstellt und positioniert. Die Variable *cam* wurde bereits in Zeile 47 deklariert und kann erst jetzt initialisiert werden, da die Variablen *width* und *height* erst in *setup()* existieren. Die Kamera wird dem Sketch zugewiesen mit dem Schlüsselwort **this**, das auf das PApplet Objekt zeigt. Das PApplet Objekt ist ein sehr wichtiges Objekt in Processing, da es viele Standardfunktionen und -variablen enthält und für sehr vieles zuständig ist. Wenn man zum Beispiel Processing in Java integrieren will, benötigt man die PApplet Klasse, um ein neues PApplet Objekt erstellen und es einem Fenster oder Frame hinzufügen zu können.

```
// FRAME
frame.setLocation(0, 0);
frame.setTitle(APP_TITLE);
ImageIcon titlebaricon = new ImageIcon(loadBytes(APP_ICON));
frame.setIconImage(titlebaricon.getImage());
frame.setCursor(Cursor.CROSSHAIR_CURSOR);

cam = new PeasyCam(this, width/2, height/2, 0, 940);
cam.setActive(false);
drop = new SDrop(this);
```

Die Kamera wird mit Hilfe der beiden bereits vorher erwähnten Variablen in der Mitte des Bildschirmes positioniert, indem man durch 2 dividiert, die Z-Koordinate liegt ungefähr bei 1000 Pixel, die Kameraposition ist daher ungefähr bei unseren Augen (Die Z-Koordinate 0 ist auf Höhe des Bildschirms, wenn sie positiv ist, kommt die Koordinate aus dem Bildschirm heraus auf den Betrachter zu, wenn sie negativ wäre, würde sie in den Bildschirm hineingehen). Nachdem die Kamera gesetzt ist, wird sie in der nächsten Zeile deaktiviert, da sie noch nicht benötigt wird. In Zeile 64 wird das SDrop Objekt initialisiert, das in Zeile 23 deklariert wurde.

OpenGL und Java OpenGL (JOGL)³

Da bereits erwähnte wurde, dass der OpenGL Renderer benutzt wird, wurde hier ein Kapitel zu diesem Thema eingeschoben.

OpenGL ist die Abkürzung für Open Graphics Library und ist eine Spezifikation für eine sprachen- und plattformunabhängige Programmierschnittstelle, mit der sich Anwendung mit 2D- und 3D-Computergrafiken entwickeln lassen. Die Schnittstelle besteht aus über 250 verschiedenen Funktionen, die es erlauben, komplexe 3-dimensionale Szenen mit einfachen Grundformen in Echtzeit darzustellen. Zudem können andere Organisationen, wie zum Beispiel Hersteller von Druckertreibern, proprietäre Erweiterungen entwickeln. OpenGL wurde 1992 von Silicon Graphics entwickelt und wird am meisten bei Computer-aided design, virtual reality, wissenschaftlichen Visualisationen, Informationsvisualisation, Flugsimulationen und Computerspielen verwendet. OpenGL wird geleitet von dem non-profit Industriekonsortium Khronos Group.

Implementierungen der OpenGL API erfolgt meist durch Systembibliotheken wie Mesa, einer freien Grafikbibliothek, oder auf einigen Betriebssystemen als Teil der Grafikkarten-Treiber. Diese führen entsprechend Befehle der Grafikkarte aus, insbesondere müssen auf der Grafikkarte nicht vorhandene Funktionen durch die CPU emuliert werden.

OpenGL weist zwei Hauptziele auf:

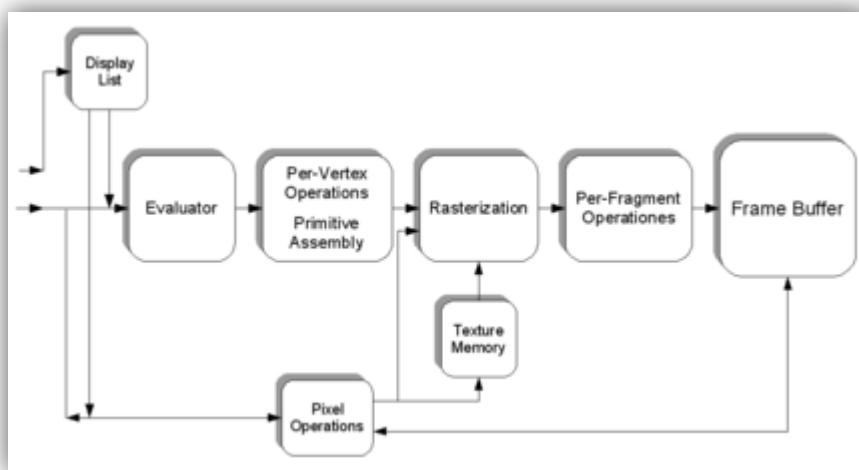
- Das Verstecken der Komplexität von Schnittstellen mit Hilfen verschiedener 3D-Beschleuniger durch das Präsentieren einer einheitlichen Schnittstelle.
- Das Verstecken von abweichenden Fähigkeiten der Hardwareplattformen, indem es eine Unterstützung für alle OpenGL Features für alle Implementierungen gibt (wenn es nötig ist, wird Software-Emulation eingesetzt).

Die Grundlegende Vorgangsweise ist, dass Grundformen wie Punkte, Linien und Polygone in Pixel umgewandelt werden. Dies geschieht durch eine Grafikpipeline, die als *OpenGL state machine* bekannt ist. Eine Grafikpipeline ist eine Modellvorstellung in der Computergrafik, die beschreibt, welche Schritte ein Grafiksystem zum Rendern durchführen muss. Die meisten OpenGL Befehle übergeben die Grundformen an die Graphikpipeline oder schreiben vor, wie die Pipeline mit diesen umzugehen hat. Vor der Einführung von OpenGL 2.0 hatte jeder Abschnitt der Pipeline eine fixe Funktion und war nur mit starken Einschränkungen konfigu-

³ Dieses Kapitel basiert auf <http://en.wikipedia.org/wiki/OpenGL> und <http://de.wikipedia.org/wiki/Jogl>

rierbar. OpenGL 2.0 bietet mehrere Abschnitte an, die voll programmierbar sind mit der Programmiersprache GLSL.

OpenGL ist eine systemnahe, prozedurale Schnittstelle, die vom Programmierer erwartet, dass er jeden einzelnen Schritt beschreibt, die für das Rendern der Szene notwendig sind. Das Gegenteil davon sind sogenannte beschreibende Schnittstellen, bei denen die Szene nur beschrieben werden muss und die Bibliothek selbst für die Details des Renderns zuständig ist. Das von OpenGL benutzte Design erwartet ein großes Wissen über die Grafikpipeline, lässt aber auch einen gewissen Spielraum zu, um neuartige Renderalgorithmen zu entwickeln.



Diese Grafik ist eine vereinfachte Version der Aufgaben der Grafikpipeline. Dabei wurden Features wie blending, logische Operationen und das Vertex Buffer Objekt nicht berücksichtigt.

OpenGL hatte historisch gesehen einen großen Einfluss auf die Entwicklung von 3D-Beschleunigern durch ein grundlegendes Level an Funktionalität, die nun bekannt ist als consumer-level hardware:

- Rasterung von Punkten, Linien, Polygonen als Grundformen
- Eine Pipeline für transforming und lightning
- Z-buffering
- Texture mapping
- Alpha blending

Es gibt drei Implementierungen in Java, nämlich Java Bindings for OpenGL (JSR 231), Java OpenGL (JOGL) und Lightweight Java Game Library (LWJGL), wobei JOGL in diesem Sketch verwendet wird.

Java OpenGL ist eine wrapper library, das heißt sie implementiert eine Spezifikation, die es erlaubt OpenGL in der Programmiersprache Java zu verwenden. Sie wurde von Kenneth Bradley Russel und Christopher John Kline entwickelt und von der Sun Microsystems Game

Technology Group weiterentwickelt. Seit 2010 ist es ein unabhängiges Open-Source-Projekt unter der BSD-Lizenz.

JOGL ermöglicht den Zugriff auf die meisten Features von Programmen in der Sprache C, mit Ausnahme des OpenGL Utility Toolkit (GLUT), da Java eigene Fenstertechniken wie das Abstract Window Toolkit (AWT), Swing und andere Erweiterungen besitzt. Die grundlegende API in C, die auch mit dem Begriff Windowing API in Verbindung gebracht wird, wird in JOGL durch Java Native Interface (JNI) Aufrufe erreicht. Daher muss das Betriebssystem OpenGL unterstützen, damit JOGL funktioniert. Die prozedural und state machine beschaffene Eigenschaft von OpenGL ist ein Widerspruch zu den typischen Methoden der Programmierung mit Java, was viele Programmierer als lästig empfinden. Trotzdem wird die Umwandlung von existierenden C Anwendungen durch die Überführung der OpenGL C API in Java Methoden leichter. Die dünne Schicht an Abstraktion macht die Ausführung zur Laufzeit ziemlich effizient, aber aus Erfahrung ist es schwieriger mit JOGL zu programmieren als mit Bibliotheken, die ein höheres Abstraktionslevel besitzen, wie zum Beispiel Java3D. Da jedoch der Code automatisch generiert wurde, können Änderungen an OpenGL schnell zu JOGL hinzugefügt werden.

Music Visual – Zeile 68 bis 73

Für die richtige Darstellung bei Größenänderungen des Fensters benötigt OpenGL das Seitenverhältnis, das in den Zeilen 70 bis 72 berechnet wird. Zuerst wird die invertierte Höhe (=Höhe⁻¹) ausgerechnet und dann mit der Breite multipliziert. In der Zeile 73 wird das Verhältnis nochmal mit sich selbst multipliziert und wiederum in eine Variable abgespeichert. Nun wird die Methode **initTUIO()** im Tab TuioHandler aufgerufen die ein neues Objekt der Klasse TuioProcessing erstellt (Der TuioHandler wird später näher beschrieben).

```
// MSA
invWidth = 1.0f/width;
invHeight = 1.0f/height;
aspectRatio = width * invHeight;
aspectRatio2 = aspectRatio * aspectRatio;
initTUIO();
```



```
18 void initTUIO() {
19     tuioClient = new TuioProcessing(this);
20 }
21 }
```

Music Visual – Zeile 75 bis 78

In Zeile 76 wird das in Zeile 30 deklarierte Objekt *minim* initialisiert und in der darauffolgenden Zeile werden die Fehlerausgaben zur Konsole unterdrückt, da dies nicht für dieses Projekt relevant ist.

```
75 // MINIM
76 minim = new Minim(this);
77 minim.debugOff();
78 newSong(START_SONG);
```

Bis auf das Erstellen des Fensters gab es noch keine visuellen oder akustischen Ausgaben, in Zeile 78 wird jedoch die Funktion **newSong()** aufgerufen, die bereits den akustischen Teil von Music Visual startet.

Music Visual – Musikwiedergabe

Die Methode erwart als Parameter den Pfad zum neuen Lied, dazu wird der Wert der Konstanten *START_SONG* übergeben. Der Startsong heißt „Elektro Kardiogramm“ und ist vom Interpreten „Kraftwerk“ aus dem Album Minim Maximum aus dem Jahr 2003.

Gleich am Anfang fallen zwei Besonderheiten auf: die Funktion **song.pause()** ist von einer sogenannten try-catch-Anweisung umgeben und der Song wird gleich am Anfang pausiert. Das hat zum einen den Grund das mögliche Fehler unterdrückt werden, in dem die Befehle in den try-Block gegeben werden, der catch-Block für die Fehlerausgaben bleibt leer. Der Song wird pausiert, da es möglich ist, dass der Song ausgetauscht wurde und somit noch der alte Song läuft. Danach wird in einer erneuten Try-Catch-Anweisung der neue Song mit dem *minim* Objekt geladen und in Zeile 192 abgespielt. Die ID3-Tags des Songs werden ebenfalls ausgelesen, sollte es sich um eine Audiodatei im MP3-Format handeln. Das ist ein Format für Zusatzinformationen (Metadaten), die in Audiodateien des MP3-Formats enthalten sein können. Die Bibliothek *minim* stellt dafür die Klasse *AudioMetaData* zu Verfügung:

Funktion	Beschreibung	Funktion	Beschreibung
album()	Album	fileName()	Dateiname
comment()	Kommentare	genre()	Genre
composer()	Komponist	orchestra()	Orchester
copyright()	Copyright	publisher()	Herausgeber
date()	Datum	author()	Autor
disc()	Cdnummer	track()	Tracknummer

Die Daten werden in den String *metadata* gespeichert, in dem sie mit dem **Plus-Operator** verbunden werden. Um einen Zeilenumbruch zu erzeugen, muss eine bestimmte Escape-Sequenz eingefügt werden. Letzteres sind Zeichenfolgen für häufig benötigte Steuerzeichen, wie Zeilenumbrüche, Tabulatorsprünge und weitere. Am Ende der Zeile ein „\n“ benötigt. Da der Backslash ein Sonderzeichen in Processing ist, kann er nicht ohne weiters verwendet werden, er muss sich selbst **maskieren** und daher lautet die Zeichenfolge \\n.

```

209 metadata=
210   LABEL_META_TITLE    +title      +"\\n"+
211   LABEL_META_AUTHOR   +author     +"\\n"+
212   LABEL_META_ALBUM    +album      +"\\n"+
213   LABEL_META_COMMENT  +comment    +"\\n"+
214   LABEL_META_COMPOSER +composer   +"\\n"+
215   LABEL_META_COPYRIGHT +copyright +"\\n"+
216   LABEL_META_DATE     +date      +"\\n"+
217   LABEL_META_DISC     +disc      +"\\n"+
218   LABEL_META_ENCODED   +encoded    +"\\n"+
219   LABEL_META_FILENAME  +filename  +"\\n"+
220   LABEL_META_GENRE    +genre     +"\\n"+
221   LABEL_META_ORCHESTRA +orchestra +"\\n"+
222   LABEL_META_PUBLISHER +publisher +"\\n"+
223   LABEL_META_TRACK+track;
224 }
225 catch(Exception e) {
226   message(LABEL_SONG_LOADING_ERROR);
227 }
228

```

In Zeile 226 befindet sich eine Methode, mit der eine Fehlermeldung ausgegeben wird, sollte es im Try-Block zu Fehlern gekommen sein. Die Methode **message()** erwartet einen String als Parameter. Die Klasse JOptionPane zeigt durch die Funktion **showMessageDialog()** einen Dialog an, der sich mit dem Ok-Button beenden lässt. Das erste Argument erwartet sich ein übergeordnetes Objekt für den Dialog, damit es sich an ihm positionieren kann. Es wird mit *this* wieder das aktuelle PApplet Objekt angegeben und somit wird die Meldung in der Mitte des Sketches geöffnet.

In Zeile 229 wird ein neues *BeatDetect* erzeugt. Die Klasse benötigt dafür von *song* die Buffergröße und die Samplerate. Zusätzlich wird die Empfindlichkeit des Beatdetektors eingestellt. Die Zeilen 233 bis 236 werden ausgelassen, da das Prinzip von MSAFluid noch nicht erklärt wurde.

```

228
229 beat = new BeatDetect(song.bufferSize(), song.sampleRate());
230 beat.setSensitivity(BEAT_SENSITIVITY);
231 b1 = new BeatListener(beat, song);
232
233 fluidSolver = new MSAFluidSolver2D((int)(FLUID_WIDTH), (int)(FLUID_WIDTH * height/width));
234 fluidSolver.enableRGB(true).setFadeSpeed(MSA_FADE_SPEED).setDeltaT(MSA_DELTA_T).setVisc(MSA
235 imgFluid = createImage(fluidSolver.getWidth(), fluidSolver.getHeight(), RGB);
236 particleSystem = new ParticleSystem();
237
238

```

Theorie zur „beat analysis“

Die verwendete Bibliothek *minim.analysis* untersucht das Musikstück auf charakteristische Merkmale. Dazu muss die Musik zuerst in ihr Frequenzspektrum zerlegt werden. Das mathematische Verfahren hierfür ist die Fouriertransformation, die ein Signal in ein Frequenzspektrum zerlegt. Im speziellen wird hierfür die FFT (Fast Fourier Transformation) angewandt. Hierbei handelt es sich um eine Analyse eines Signales, im gegebenen Fall eines Musikstückes, welches bereits in einer digitalen Form vorliegt, das heißt bereits digital „abgetastet“ wurde. In dieser Arbeit wird auf die mathematischen Grundlagen nicht eingegangen. Nur so viel sei gesagt: Nach der Zerlegung des Musikstückes in sein Frequenzspektrum wird nach charakteristischen Eigenschaften des Schlagzeuges gesucht. Es wird zum Beispiel nach einem charakteristischen „Fingerabdruck“ eines bestimmten Schlaginstrumentes, zum Beispiel „snare“ gesucht um seine Intensität graphisch darzustellen. Die einzelnen Schlaginstrumente kann man dabei auch auswählen. Der Vorgang wird letztendlich „beat analysis“ genannt und findet Verwendungen in Music Visual.

Theorie zu MSAFluid

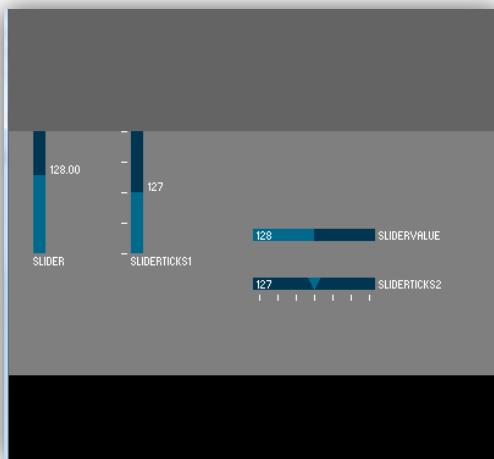
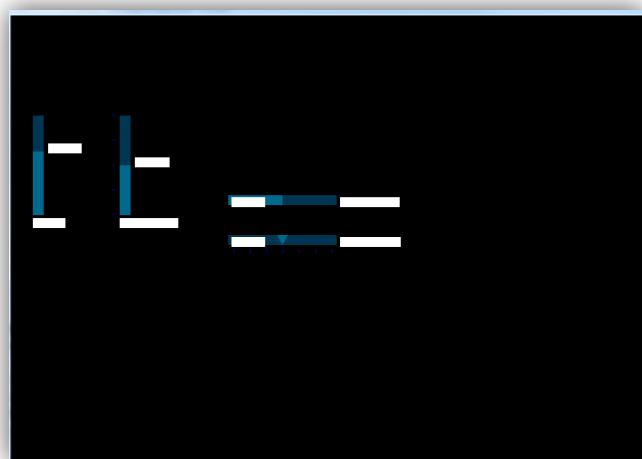
MSAFluid verwendet die Navier-Stokes-Gleichungen, einem präzisen mathematischen Modell für die meisten Flüssigkeiten, die in der Natur vorkommen. Da am Computer Grafiken überzeugend aussehen und schnell berechnet werden sollen, sind die verwendeten Algorithmen nicht so genau wie physikalische Lösungen, dafür stabiler. Der Algorithmus ist optimiert für eine Echtzeitdarstellung für graphisch-künstlerische Programme. Das Ergebnis am Bildschirm kann als Flüssigkeit oder auch als Rauch interpretiert werden. Einer Veränderung der einzelnen Parameter ermöglicht eine unendliche Anzahl an verschiedenen Bildsequenzen.

Music Visual – Zeile 80 bis 87

ControlP5 ist eine Bibliothek, mit der man die wichtigsten grafischen Komponenten in den Sketch einbauen kann, die man für ein User Interface benötigt. Sie besteht aus leichtgewichtigen Komponenten, das heißt, die Komponenten werden nicht vom Betriebssystem dargestellt, sondern werden von der Bibliothek gezeichnet.

Das Gegenstück dazu ist das Fenster eines jeden Processing Sketches, das als schwergewichtige Komponente durch die Java Klasse *java.awt* dargestellt wird.

In Zeile 81 wird das Hauptobjekt der controlp5 Bibliothek erstellt und in Zeile 87 wird die Methode *initgui()* aufgerufen, doch beim Testen gab es Probleme mit der Bibliothek:

Orginal*Music Visual*

Antwort auf das Problem mit der GUI bezüglich dem MSAFluidDemo, das im Projekt integriert ist:

"I believe it's the blending being done in the particleSystem. If you turn off the blending, it works, so maybe the glBlendFunc needs to be reset. In any case, you can work around this."
(jeff_g -> <http://forum.processing.org/topic/controlP5-OpenGL>)

Das Problem konnte behoben werden durch den Befehl controlP5.setAutodraw(false) in der Zeile 85, indem die Komponenten nicht automatisch neu gezeichnet werden, sondern erst an der richtigen Stelle in draw(). Diese Antwort ist ein weiteres Beispiel, wie man spontan auftretende, unerwartete Probleme durch die Processing Community relativ leicht lösen kann.

Music Visual – Die grafische Oberfläche

Vor der Methode **initgui()** werden einige Variablen deklariert, die von Bedeutung sind: Die Integer Variablen *gui_hue*, *gui_saturation*, *gui_brightness* sind Variablen in denen die HSB-Farbe der Partikel des Fluidsystems gespeichert ist. Standardmäßig sind die Farben abhängig von der Framerate, wenn jedoch in der GUI der Haken „Einfarbig“ aktiviert ist, wird diese Farbe verwendet. Der boolesche Wert dieser Checkbox wird in *onecolor* gespeichert. Weitere Variablen sind *fluid_pause* und *gui_show* die beide selbst erklärend sind.

```

01 int gui_hue=0;
02 int gui_saturation=100;
03 int gui_brightness=100;
04 boolean fluid_pause=false;
05 boolean onecolor=false;
06 boolean gui_show=false;
```

Es werden insgesamt vier verschiedene Komponenten verwendet um das Menü zu gestalten. Zwei weiteren Hilfsklassen dienen nur zum allgemeinen Ansprechen der Komponenten: **Controller** und **ControllerGroup**

Button	Slider	Toggle	Textarea
menu_button	hue_slider	status_toggle	id3tags_textarea
exit_button	saturation_slider	fluid_pause_toggle	
help_button	brightness_slider	free_rotate_toggle	
info_button		one_color_toggle	

Button

Neue Buttons können mit der Hilfe von **controlP5.addButton()** erstellt werden. Die Parameter, die angegeben werden müssen, sind ein Name, in dem keine Leerzeichen vorkommen dürfen, da dieser als Funktionsname dienen wird, irgendeinen Wert für die Position im 2-dimensionalen Raum, sowie Breite und Höhe des Buttons. Die Methode erzeugt einen Button, der dem Objekt zugewiesen werden kann. Mit der Methode **setLabel()** der Klasse Button kann das **Label**, also die Beschriftung, verändert werden. Es wird wieder eine vorher definierte Konstante verwendet.

Slider

Neue Slider können mit der Funktion **controlP5.addSlider()** erzeugt werden. Die Parameter unterscheiden sich nur geringfügig von Button. Notwendige Parameter sind ein Name, ein Minimal- und Maximalwert für den Schieberegler, den Standardwert, sowie Position und Größe. Zusätzlich zur Funktion **setLabel()** wird **setNumberOfTickMarks()** verwendet, um die Anzahl der Striche des Schiebereglers zu bestimmen. Diese Striche zeigen dem Benutzer, auf welchen Wert sie gerade eingestellt sind. Je mehr Striche es gibt, desto genauer lässt sich der korrekte Wert feststellen.

```
08 Button menu_button, exit_button, help_button, info_button;
09 Slider hue_slider,saturation_slider,brightness_slider;
10 Toggle status_toggle, fluid_pause_toggle, free_rotate_toggle, one_color_toggle;
11 Textarea id3tags_textarea;
12 DropdownList visualisation_dropdown;
13 controlP5.Controller[] elements;
14 controlP5.ControllerGroup[] elements2;
15
16 void initgui() {
17     menu_button = controlP5.addButton("menu_button", 0, 0, 0, 100, 19);
18     hue_slider=controlP5.addSlider("hue_slider",gui_hue,360,0,20,100,10,100);
19     hue_slider.setLabel(LABEL_GUI_HUE);
20     hue_slider.setNumberOfTickMarks(10);
21
22     saturation_slider=controlP5.addSlider("saturation_slider",0,100,gui_saturation,70,100,10,10);
23     saturation_slider.setLabel(LABEL_GUI_SATURATION);
24     saturation_slider.setNumberOfTickMarks(10);
25
26     brightness_slider=controlP5.addSlider("brightness_slider",0,100,gui_brightness,120,100,10,10);
27     brightness_slider.setLabel(LABEL_GUI_BRIGHTNESS);
28     brightness_slider.setNumberOfTickMarks(10);
29
30     status_toggle=controlP5.addToggle("status_toggle",false,20,30,20,20);
31     status_toggle.setLabel(LABEL_GUI_PAUSE_SONG);
32
33     fluid_pause_toggle=controlP5.addToggle("fluid_pause_toggle",false,100,30,20,20);
34     fluid_pause_toggle.setLabel(LABEL_GUI_PAUSE_VISUAL);
35
36     free_rotate_toggle=controlP5.addToggle("free_rotate_toggle",false,220,30,20,20);
37     free_rotate_toggle.setLabel(LABEL_GUI_START_3D);
38
39     id3tags_textarea= controlP5.addTextarea("id3tags_textarea",metadata,20,250,400,500);
40
41     one_color_toggle=controlP5.addToggle("one_color_toggle",false,320,30,20,20);
42     one_color_toggle.setLabel(LABEL_GUI_UNI_COLOR);
43
44     help_button = controlP5.addButton("help_button", 0, 100,0, 100, 19);
45     help_button.setLabel(LABEL_GUI_HELP);
46
47     exit_button = controlP5.addButton("exit_button", 0, 200,0, 100, 19);
48     exit_button.setLabel(LABEL_GUI_QUIT);
49
50     info_button = controlP5.addButton("info_button", 0, width/2-222,height/2, 444, 80);
51     hide_button("info_button");
52
53
54
```

Toggle

Toggle Buttons schalten zwischen zwei Zuständen beim Anklicken hin und her, meistens zwischen an und aus. Die bekanntesten Vertreter sind die Checkboxen bei HTML Formularen auf Webseiten. Diese lassen sich leicht mit einer Variable von Typ Boolean steuern. Neue Toggle lassen sich durch **controlP5.addToggle()** erstellen, wobei die Argumente dem Button gleichen, anstelle eines Integer Wertes für den Wert, wird ein boolescher Wert verwendet, also true oder false.

Textarea

Textareas werden verwendet um mehrzeilige Strings anzuzeigen. Für einzeilige Texte werden Labels verwendet. Die Initialisierung erfolgt durch **controlP5.addTextArea()**, die Parameter sind wieder gleich wie bei Button und Toggle mit dem Unterschied, dass diesmal der Text, der angezeigt werden soll als String angegeben wird. Da in der Methode *newSong()* die Variable *metadata* mit Informationen befüllt wurde, kann diese hier gleich zum Einsatz kommen.

In Zeile 53 ist eine Hilfsfunktion, die es ermöglicht, Komponenten auszublenden. ControlP5 bietet die Möglichkeit alle Objekte über die Hauptklasse mit der Funktion **controller()** anzusprechen. Es wird das jeweilige Objekt zurückgegeben und es kann direkt darauf zugegriffen werden. In *hide_button()* in Zeile 66 wird dies genutzt, um den Button mit **setVisible(false)** ausblenden zu lassen. *Der Name der Methode controller() ist nicht zufällig gewählt, da Controller die Super- oder Elternklasse von Button ist und damit alle Eigenschaften von Controller geerbt hat. Diese Behauptung muss schlicht und ergreifend ohne weitere Erklärungen hingenommen werden, da die Abschweifung vom Thema zu groß wäre.* Alle verwendeten Komponenten, außer Textarea stammen von dieser Klasse ab. Dadurch können sie alle hintereinander in einem sogenannten Array abgespeichert werden.

Arrays

Arrays sind Datenstrukturen, vereinfacht gesagt Variablen, in denen viele weitere Variablen gespeichert werden können. Diese Einträge können über eine eindeutige Nummer angesprochen werden. Die Variablen werden durchnummeriert, beginnend bei der **Zahl 0 (und nicht 1!).** Arrays erkennt man an den eckigen Klammern, die bei der Deklaration hinter den Datentyp geschrieben werden. Initialisieren kann man Arrays entweder gleich beim Deklarieren, indem man wie bei einer Zuweisung das Gleichheitszeichen verwendet und die Werte mit Kommata voneinander getrennt in geschwungenen Klammern angibt. Eine andere Möglichkeit ist, nur die Anzahl der Einträge anzugeben indem man das Schlüsselwort **new** benutzt, gefolgt vom Datentyp. In eckigen Klammern wird zum Schluss noch die Anzahl angegeben. Die Werte können nun in *setup()* oder *draw()* zugewiesen werden, wie zu vermuten war, in eckigen Klammern in Form einer Zuweisung. In Zeile 15 ist eine weitere Variante bei dem die beiden vorherigen Schritte zusammengefasst wurden. Man beachte, dass der in Zeile 16 gezeigter Versuch nicht gültig ist, da die Anzahl der Einträge in den geschwungenen Klammern bereits die Arraygröße vorgibt.

```

01 int[] integers;
02 float[] floats;
03 String[] names;
04 int[] answers=new int[10];
05 float[] coins = { 1,0.5f,0.2f,1,0.5f};
06 String[] colors;
07
08 void setup() {
09   answers[0]=1;
10   answers[1]=4;
11   answers[2]=3;
12   answers[3]=1;
13   answers[4]=2;
14   // ...
15   colors=new String[] { "blue", "green", "black" };
16   names = new String[3]{ "Tod", "Michael", "James" }; // not possible
17 }
```

Erweiterte For-Schleifen

Um die in Zeile 57 aufgerufenen Methoden *hidogui()* besser zu verstehen, muss man die verkürzte Variante der For-Schleife kennen. Da in der Methode alle GUI-Komponenten versteckt werden sollen, muss jede Komponente einzeln angesprochen werden. Es wäre möglich gewesen, jedes einzeln anzusprechen. Da der Code allgemein und leicht zu warten sein sollte, wurden in Zeile 54 alle Komponenten in das Array *elements* gegeben, dessen Typ *controlP5.Controller* ist. Wie bereits vorher beschrieben ist dies die Superklasse von allen Komponenten und daher kann dieses Array alle Komponenten aufnehmen. Die einzige Ausnahme ist *Textarea*, die von *ControllerGroup* abstammt und daher ein eigenes Array benötigt.

```

124 void hidogui() {
125   for(controlP5.Controller cur:elements)
126     cur.hide();
127   for(controlP5.ControllerGroup cur:elements2)
128     cur.hide();
129   menu_button.setLabel(LABEL_GUI_MENU_SHOW);
130 }
131
132 void showgui() {
133   for(controlP5.Controller cur:elements)
134     cur.show();
135   for(controlP5.ControllerGroup cur:elements2)
136     cur.show();
137   menu_button.setLabel(LABEL_GUI_MENU_HIDE);
138 }
139 }
```

In Zeile 124 kommt die neue Schleife zum ersten Mal zum Einsatz. In den runden Klammern ist auf der rechten Seite das Array angegeben, getrennt mit einem Doppelpunkt von der lin-

ken Seite. Auf der anderen Seite ist ein neues Objekt mit dem Datentyp der Arrayelemente erzeugt worden. Über dieses Objekt, das hier *cur* genannt wurde, kann die *Methoden hide()* der Klasse Controller bzw. ControllerGroup aufgerufen werden. In den Zeilen 132 bis 138 befindet sich die umgekehrte Funktion *showgui()*, die mit **show()** wieder alle Elemente anzeigt. Der Aufbau ist analog zu *hidogui()*.

Die Methoden der GUI-Komponenten

Jede Komponente der controlP5 Bibliothek hat eine eigene Methode mit dem Rückgabewert void, um sie interaktiv zu machen, damit eine Aktion passiert, wenn man einen Button drückt, einen Toggle umschaltet oder bei einem Slider den Wert verstellt.

Beim Erstellen einer neuen Komponente musste das erste Element ein eindeutiger Name ohne Leerzeichen sein. Dieser Name wird nun für den Methodennamen verwendet. Die Methoden haben auch ein Argument, sie akzeptieren einen Integer Wert, beziehungsweise einen Boolean, wenn es sich um einen Toggle handelt. ControlP5 gibt dann den Wert der Komponente an diese Funktion. Betrachtet man zuerst die Methoden der Toggle ergibt sich folgende Tabelle:

Methodename	Flag=true	Flag=false
one_color_toggle()	Die Partikelfarbe wird umgestellt auf die alternative Farbe	Die Partikelfarbe wird zurückgesetzt auf das Original
free_rotate_toggle()	3-D-Modus wird gestartet	3-D-Modus wird beendet
fluid_pause_toggle()	Die Simulation wird gestoppt	Die Simulation wird fortgesetzt
status_toggle()	Der Song wird pausiert	Der Song wird fortgesetzt.

```
77 void one_color_toggle(boolean flag) {
78     if(flag) {
79         onecolor=true;
80     } else {
81         onecolor=false;
82     }
83 }
84
85 void free_rotate_toggle(boolean flag) {
86     if(flag) {
87         free_rotate_toggle.setLabel(LABEL_GUI_3D_STARTED);
88         cam.setActive(true);
89     } else {
90         free_rotate_toggle.setLabel(LABEL_GUI_3D_QUIT);
91         cam.setActive(false);
92     }
93 }
94
95 void fluid_pause_toggle(boolean flag) {
96     if(flag) {
97         fluid_pause=true;
98         fluid_pause_toggle.setLabel(LABEL_GUI_VISUAL_CONTINUE);
99     } else {
100        fluid_pause=false;
101        fluid_pause_toggle.setLabel(LABEL_GUI_VISUAL_PAUSE);
102    }
103 }
104
105 void status_toggle(boolean flag) {
106     if(flag) {
107         song.pause();
108         status_toggle.setLabel(LABEL_GUI_SONG_CONTINUE);
109     } else {
110         song.play();
111         status_toggle.setLabel(LABEL_GUI_SONG_PAUSE);
112     }
113 }
114 }
```

Um die Kamera zu aktivieren wird die Funktion **setActive()** der Peasycam Klasse verwendet. Die Methode `song.pause()` in Zeile 107 ist bereits bekannt, neu ist die reverse Variante in Zeile 110, nämlich **song.play()**.

Nun werden die Methoden der Button Objekte untersucht. Hier finden sich einige neue Funktionen. In Zeile 62 wird der Button *info* versteckt. Dieser gibt einige Meldungen aus, zum Beispiel wenn der 3D-Modus gestartet wird oder wenn der Button *help* gedrückt wurde. Da Meldungen nur eine kurze Mitteilung sein sollten, lässt sich dieser Button per Mausklick wieder ausblenden. Die dafür zuständige Methode in Zeile 64 hat einen etwas ungünstig gewählten Namen, da sie als Funktion eines Buttons durchgehen könnte, jedoch wird die Methoden nur einmal verwendet und zwar zwei Zeilen darüber. In den Zeilen 68 bis 71 befindet sich die Funktion **captionLabel()** mit der man auf das Label des Buttons zugreifen kann. Der Button besitzt ein eigenes Objekt der Klasse Label und man kann daher gleich mit **set()** den Wert verändern. In Zeile 74 kommt wieder einmal eine Funktion der Processing

core library an den Zug, **exit()** beendet nämlich den Sketch. Etwas abgeschieden vom restlichen Code wird ab Zeile 115 das gesamte Menü angezeigt oder versteckt, je nachdem ob *gui_show* true oder false ist.

```

114
115 void menu_button(int value) {
116   gui_show=!gui_show;
117   if(gui_show)
118     showgui();
119   else {
120     hidegui();
121   }
122 }
```

Um den Wert einer booleschen Variable umzudrehen, gibt es zwei Varianten: Entweder man macht weist die Variable sich selbst zu mit dem Gleichheitszeichen und fügt ein **Rufzeichen** hinzu vor der Zuweisungsseite, damit der Befehl in Worten lautet: Weise der Variable den gegenteiligen Wert von sich selbst zu. Die schnellste Variante ist eine bitweise Zuweisung. Dabei wird der Binärcode, also Nullen und Einsen kombiniert. Es gibt drei Arten.

Bitweise-AND-Zuweisung:

	Kombination 1	Kombination2	Kombination3	Kombination4
a	true	false	false	true
b	false	true	false	true
a & b	false	false	false	true

Bei dieser Zuweisung wird kontrolliert ob in beiden Variablen das Bit gesetzt ist. Wenn es zutrifft, ist das Ergebnis true, ansonsten false.

Bitweise-OR-Zuweisung:

	Kombination 1	Kombination2	Kombination3	Kombination4
a	true	false	false	true
b	false	true	false	true
a b	true	true	false	true

Bei der Or-Zuweisung muss nur einmal der Wert true sein, darf aber auch bei beiden true sein. Daher ist das Ergebnis hier bei drei Kombination true, nur bei einer false.

Bitweise-XOR-Zuweisung:

	Kombination 1	Kombination2	Kombination3	Kombination4
a	<i>true</i>	<i>false</i>	<i>false</i>	<i>true</i>
b	<i>false</i>	<i>true</i>	<i>false</i>	<i>true</i>
a ^ b	<i>true</i>	<i>true</i>	<i>false</i>	<i>false</i>

Die letzte Zuweisung nennt sich exklusive Or-Zuweisung, da nur eine Variable true sein darf. Diese Variante eignet sich perfekt dafür, Werte umzudrehen. Daher hätte man das vorherige Problem so lösen können: gui_show ^= true.

Die drei Slider brauchen keine langen Erklärungen, der geänderte Wert wird einfach in eine Variable gespeichert. Dies geschieht in den Zeilen 140 bis 150.

```

140 void hue_slider(int value) {
141   gui_hue=value;
142 }
143
144 void saturation_slider(int value) {
145   gui_saturation=value;
146 }
147
148 void brightness_slider(int value) {
149   gui_brightness=value;
150 }
```

Klassen – Aufbau

Da die grafische Oberfläche nun implementiert ist, kann nun ein neues Thema angegangen werden, dass eigentlich ein Feature von Java ist, jedoch in Processing sehr oft benötigt wird. Gemeint sind sogenannte Klassen. Mit ihnen kann man eigene Referenztypen erstellen und neue Objekte mit ihnen instanziieren. Klassen bestehen aus Methoden und ein oder mehreren Konstruktoren, mit denen eine neue Instanz erzeugt werden kann. Natürlich besitzt eine Klasse auch Variablen, die Sichtbarkeit der Variablen spielt hierbei auch oft eine Rolle.

Sichtbarkeit von Variablen

In Processing selbst gibt es nur lokale und globale Variablen. Lokale Variablen sind Variablen die in Funktionen wie setup(), draw() oder eigenen Funktionen deklariert wurden. Sie sind nur in diesem Bereich sichtbar. Globale Variablen werden außerhalb von Funktionen deklariert und sind überall gültig. Sollte es eine lokale und eine globale Variable mit dem gleichen

Namen geben, wird die lokale Variable benutzt, ab dem Zeitpunkt, ab dem die lokale Variable nicht mehr existiert, wird wieder die globale Variable benutzt. In Java gibt es weitere Sichtbarkeiten:

Sichtbarkeit	Innerhalb eines Pakets	Abgeleiteten Klassen	Außerhalb des Pakets
Private	Unsichtbar	Unsichtbar	unsichtbar
Default	Sichtbar	unsichtbar	Unsichtbar
Protected	Sichtbar	Sichtbar	Unsichtbar
Public	Sichtbar	Sichtbar	sichtbar

Wenn Klassen verwendet werden, sind in Processing höchstens drei von 4 Sichtbarkeiten wichtig. Speziell in Bibliotheken werden Klassen verwendet, damit nicht alle Variablen sichtbar sind. Die Variablen die sichtbar sein sollen, müssen mit dem Schlüsselwort **public** vor dem Typ gekennzeichnet werden, sollte sie beim Importieren nicht sichtbar sein, muss entweder **private** oder kein Schlüsselwort verwendet werden, es kommt darauf an, ob die Variable im Paket der Bibliothek sichtbar sein soll oder nicht.

Klassen

Klassen werden mit dem Schlüsselwort **class** eingeleitet, gefolgt von geschwungenen Klammern, die wie bei anderen Strukturen, den Programmblöck abgrenzen. Danach können Vari-

```

01 class Tv {
02
03   color colour;
04   int width;
05   int height;
06   boolean on=false;
07
08   void turnOn() {
09     on=true;
10     println("Tv is turned on");
11   }
12
13   void turnOff() {
14     on=false;
15     println("Tv is turned off");
16   }
17 }
```

ablen wie in einem normalen Sketch definiert werden, analog verhält es sich auch mit den Funktionen. Zum besseren Verständnis wird hier eine Klasse *Tv* programmiert, mit der man einen neuen Fernseher mit einer bestimmten Höhe, Breite und Farbe instanziiieren kann. Er

soll Funktionen zum Ein- und Ausschalten besitzen und man soll auch den Status ausgeben können, also ob entweder Ein oder Aus ist.

Die entsprechenden Variablen, noch mit Default Sichtbarkeit, sind in diesem Beispiel bereits gesetzt worden, sowie die Funktionen *turnOn()* und *turnOff()*. Bevor Die Statusfunktion gezeigt wird, muss eine fortgeschrittene Struktur gezeigt werden, nämlich die verkürzte If-Else Anweisung (**ternärer Operator**)

```
01 (expression)?true_output:false_output;
02
03 int apples=1;
04
05 String text_to_say="I have got "+apples;
06
07 text_to_say+=(apples>1)?" apples":" apple";
08 text_to_say+=" in my pocket.";
09
10 println(text_to_say); // I have got 1 apple in my pocket.
```

Der Aufbau ist in Zeile 1 beschrieben. Um die Ausgabe von true_output und false_output zu verstehen, kann man sich das mit einer Art Pseudofunktion vorstellen:

```
01 {
02 if(expression)
03   return true_output;
04 else
05   return false_output
06 }
```

Im vorherigen Beispiel wird diese Technik genutzt um den Satz grammatisch richtig darzustellen. Die Funktion *getStatus()* verwendet diese Struktur. Man beachte, dass die Struktur nochmals in runden Klammern steht, da der Compiler sonst ausgibt: „cannot convert from String to boolean“. Durch die zusätzlichen Klammern betrachtet er den Teil nicht mehr als String sondern als eigenen Ausdruck.

```
01 class Tv {  
02     color colour;  
03     int width;  
04     int height;  
05     boolean on=false;  
06  
07     // void turnOn()  
08     // void turnOff()  
09  
10     void getStatus() {  
11         println("The Tv ist turned "+((on)?"on":"off"));  
12     }  
13 }  
14 }
```

Im letzten Schritt machen wir die Variable *on* nur für die dafür vorgesehenen Funktionen sichtbar, damit der Fernseher nicht von außen unerlaubterweise eingeschalten werden kann. Um ein neues Objekt erzeugen zu können, wurde noch ein Konstruktor nach den Variablen eingefügt. Er ist von Aufbau her wie eine normale Funktion, mit dem Unterschied, dass es keinen Rückgabewert gibt. Um den Klassenvariablen Werte zuweisen zu können, bräuchte man eigentlich nur den Namen zu schreiben, jedoch haben die Parameter denselben Namen wie die Klassenvariablen. Um auf die Klassen internen Variablen daher trotzdem zugreifen zu können, gibt es das Schlüsselwort **this**, das bereits früher auf eine Klasse gezeigt hat, nämlich PApplet.

```
01 class Tv {  
02     color colour;  
03     int width;  
04     int height;  
05     private boolean on=false;  
06  
07     Tv(int width,int height,color colour) {  
08         this.width=width;  
09         this.height=height;  
10         this.colour=colour;  
11     }  
12  
13     void turnOn() {  
14         on=true;  
15         println("Tv is turned on");  
16     }  
17  
18     void turnOff() {  
19         on=false;  
20         println("Tv is turned off");  
21     }  
22  
23     void getStatus() {  
24         println("The Tv ist turned "+((on)?"on":"off"));  
25     }  
26 }  
27 }
```

In der Klasse BeatListener wird noch eine erweiterte Java-Technik verwendet, die zur vollständigen Dokumentation an dieser Stelle erklärt werden soll.

Interfaces

Interfaces sind abstrakte Typen, die von einer Klasse implementiert werden können. Das heißt sie bestehen nur aus Signaturen von Funktionen und führen keinen Code aus. Ein Beispiel dazu wäre `void doanything();` In der Klasse werden die Funktionen vollständig ausgeschrieben, samt Programmcode, der sich in den Funktionen befinden soll. Wenn eine Klasse ein Interface implementieren soll oder muss, wird das Schlüsselwort **implements** hinter dem Klassennamen verwendet, gefolgt vom Namen des Interfaces. Es können auch mehrere Interfaces implementiert werden, die mit Beistrichen voneinander getrennt werden.

BeatListener

Die Klasse kommt mit zwei privaten Objekten aus, BeatDetect aus dem Paket *minim.analysis*, sowie AudioPlayer aus dem Hauptpaket *minim*. Der Konstruktor erwartet auch genau dieselben Klassen und initialisiert sie mit Hilfe des Schlüsselwortes `this`. Zeile 11 ist nicht von Bedeutung, da zum Verständnis der Quellcode der Bibliothek nötig wäre. BeatListener implementiert das Interface AudioListener, das die Funktion `samples()` fordert. Es gibt jedoch zweimal die gleiche Funktion mit unterschiedlich vielen Parametern, dies ist in Processing aufgrund von Java möglich. Diese Technik nennt sich „Überladen“ und funktioniert nur mit unterschiedlichen Datentypen. Wenn nur die Variablennamen unterschiedlich sind, weiß der Compiler nicht, welche Funktion er aufrufen muss. Beide Funktionen rufen die Funktion `detect()` der Klasse BeatDetect auf. Somit ist die Klasse BeatDetect auch fertig.

TuioHandler

Bevor der wichtigste Teil, die Bibliothek MSA Fluid an der Reihe ist, wird der TuioHandler durchgenommen, der es ermöglicht, Daten, die über das Open-Sound-Control (OSC) Protokoll über WLAN gesendet werden, auszulesen. Als Sender der Daten können viele Programme verwendet werden, bei Music Visual wird die iPhone Applikation **TuioPad** verwendet. Andere Apps wie OSCemote, iOSC, Hex OSC S oder MSA Remote sollten auch funktionieren, wobei nur Letzteres getestet wurde.

Die Funktion `initTuioClient()` wurde bereits beim `setup()` vorgestellt. In Zeile 13 ist ein Faktor von 0.02 angegeben, da der iPhone Bildschirm so klein ist und daher die Geschwindigkeiten der Partikel viel zu schnell wären, wenn die Geschwindigkeit der Berührungsänderungen übertragen werden würde. In der nächsten Zeile ist ein Faktor von 0.001, damit sich die Par-

tikel trotzdem weiterbewegen, auch wenn der Cursor gerade stationär ist. Der TuioHandler bietet einige Funktion an, um die OSC Daten auszulesen:

void updateTUIO()	Hauptfunktion bei Änderungen
void addTuioObject(TuioObject tobj)	Bei hinzugefügten Objekten
void updateTuioObject(TuioObject tobj)	Bei geänderten Objekten
void removeTuioObject(TuioObject tobj)	Bei einem entfernten Objekt
void addTuioCursor(TuioCursor tcur)	Bei einem hinzugefügten Cursor
void updateTuioCursor(TuioCursor tcur)	Bei einem geänderten Cursor
void removeTuioCursor(TuioCursor tcur)	Bei einem entfernten Cursor
void refresh(TuioTime bundleTime)	Bei einem Refresh

Die einzige Funktion, die verwendet wird, ist **updateTUIO()**, alle anderen werden zwar ange- schrieben, jedoch nicht benutzt. In Zeile 21 werden die Cursors, die vom iPhone übertragen wurden, in die Variable *tuioCursorList* vom Typ Vector gespeichert. **Vector ist eine Java Klasse, die ähnlich wie Arrays funktioniert.** Danach wandert das Programm mit einer Schleife durch den Vector bis zum letzten Element (=Anzahl der Einträge – 1), das durch die Funktion **size()** auffindbar ist. Es wird danach ein neuer TuioCursor erzeugt, indem mit der Funktion **elementAt(i)** das Element geholt wird, i ist die Zählvariable der For-Schleife. Um die Geschwindigkeit des Cursors zu bestimmen, nimmt man den Rückgabewert der Funktion **getX- Speed()** und multipliziert ihn mit dem Faktor *tuioCursorSpeedMult*, der bereits früher im Programm definiert wurde.

```

19
20 void updateTUIO() {
21     Vector tuioCursorList = tuioClient.getTuioCursors();
22     for (int i=0;i<tuioCursorList.size();i++) {
23         TuioCursor tcur = (TuioCursor)tuioCursorList.elementAt(i);
24         float vx = tcur.getXSpeed() * tuioCursorSpeedMult;
25         float vy = tcur.getYSpeed() * tuioCursorSpeedMult;
26         if(vx == 0 && vy == 0) {
27             vx = random(-tuioStationaryForce, tuioStationaryForce);
28             vy = random(-tuioStationaryForce, tuioStationaryForce);
29         }
30         for(int ii=0;ii<MSA_NEW_FORCES;ii++)
31             addForce(tcur.getX(), tcur.getY(), vx, vy);
32     }
33 }
```

Analog dazu wird mit dem y-Wert vorgegangen, sodass es jetzt zwei Float Variablen, vx und vy, gibt. Da es möglich ist, dass die Geschwindigkeit Null beträgt, wird in Zeile 26 eine If- Abfrage gemacht, die bei Eintreten der Änderung die beiden Geschwindigkeitsvariablen auf einem zufälligen Wert zwischen *-tuioStationaryForce* und *+tuioStationaryForce* setzt. Dazu wird die Processing Funktion **random()** verwendet, die auf zwei Arten benutzt werden kann:

entweder es wird nur ein Parameter angegeben und der zurückgegebene Wert liegt zwischen 0 und diesem Wert oder es werden zwei Werte angegeben, wobei der Erste als untere, der Zweite als obere Grenze der Zufallszahl dient. In der Zeile 31 wird eine neue Kraft genau an diesem Punkt im Sketch hinzugefügt, an dem der Finger den iPhone Bildschirm berührt.

Die Koordinaten bekommt man mit **getX()** und **getY()** vom aktuellen TuioCursor Objekt, die Kräfte, die das dritte und vierte Argument bilden, wurden bereits berechnet und müssen nur mehr eingefügt werden. Da eine einzelne Kraft alleine zu wenig ist, wird eine Schleife verwendet, die bis zum Wert von **MSA_NEW_FORCES** zählt. Der Wert beträgt 5 und kann, wie auch alle anderen Konstanten, angepasst werden.

Konstanten

Konstanten können nur einmal initialisiert werden. Man könnte sagen, sie haben eine read-only-Eigenschaft. Konstanten werden immer mit dem Schlüsselwort **final** eingeleitet. In diesem Projekt wird zusätzlich das Schlüsselwort **static** verwendet, das angibt, dass man auf diese Variable von überall zugreifen kann, ohne dass man eine Instanz dieser Klasse machen muss. Hier ist dies eindeutig nur optional, da die Konstanten nicht in einer Klasse sind, es wurde trotzdem geschrieben, da Konstanten in Java normalerweise auf diese Art geschrieben werden. Würde man eine Bibliothek mit ein oder mehreren Klassen für Processing schreiben, hätte das einen Sinn, da man auf diese Konstanten dann direkt zugreifen könnte.

Liste aller Konstanten und deren Werte:

Name der Konstante	Wert
APP_TITLE	„Music!“
APP_ICON	„icon.gif“
ANTI_ALIASING_4X	true
START_SONG	„Elektrokardiogramm.mp3“;
SOURCE_NOISE_INC	0.005
BEAT_SENSITIVITY	300
MSA_FADE_SPEED	0.003
MSA_DELTA_T	0.5
MSA_VISC	0.0001
MSA_NEW_FORCES	10
MSA_STANDARD_FORCES	true
ALT1_XOFF_INC	0.01
ALT1_THISALPHA_DEC	0.001
ALT1_THISALPHA_2_DEC	0.01

ALT1_THISALPHA_3_DEC	0.005
LABEL_3D_ENTER	„3-D Modus kann durch ENTER beendet werden“
LABEL_HELP	"ALT + Maus = Menü bewegen #"+ " ALT + [H] = Menü verstecken/anzeigen #+ " [C] = Hilfe anzeigen"
LABEL_ONLY_SOUND	„Bitte nur Sounddateien verwenden.“
LABEL_FILE_ADDED	„Datei hinzugefügt!“
LABEL_URL_ADDED	„URL hinzugefügt!“
LABEL_URL_INVALID	„Die URL ist ungültig!“
LABEL_META_TITLE	„Titel:“
LABEL_META_AUTHOR	„Autor:“
LABEL_META_ALBUM	„Album:“
LABEL_META_COMMENT	„Kommentar:“
LABEL_META_COMPOSER	„Komponist:“
LABEL_META_COPYRIGHT	„Copyright:“
LABEL_META_DATE	„Datum:“
LABEL_META_DISC	„CD:“
LABEL_META_ENCODED	„Encoded:“
LABEL_META_FILENAME	„Dateiname:“
LABEL_META_GENRE	„Genre:“
LABEL_META_SONGLENGTH	„Songlänge:“
LABEL_META_ORCHESTRA	„Orchester:“
LABEL_META_PUBLISHER	„Herausgeber:“
LABEL_META_TRACK=	„Songnummer:“
LABEL_SONG_LOADING_ERROR	„Es ist ein Fehler aufgetreten beim Laden des Liedes.“
LABEL_START_3D	“3-D-Modus starten”
LABEL_GUI_HUE	“Farbton”
LABEL_GUI_SATURATION	“Sättigung”
LABEL_GUI_BRIGHTNESS	“Helligkeit”
LABEL_GUI_PAUSE_SONG	“Song pausieren”
LABEL_GUI_PAUSE_VISUAL	Visualisation pausieren
LABEL_GUI_START_3D	“3-D-Modus starten”
LABEL_GUI_UNI_COLOR	Einfarbig
LABEL_GUI_HELP	“Hilfe”
LABEL_GUI_QUIT	“Beenden”;
LABEL_GUI_3D_STARTED	“3-D-Modus gestartet”;
LABEL_GUI_3D_QUIT	“3-D-Modus beenden”

MSAParticle

MSAFluid ist aus drei Teilen aufgebaut: Dem Tab MSA, der mit der Funktion `draw()` von der Wichtigkeit vergleichbar ist, der Klasse MSAParticleSystem, die alle Partikel enthält und der Klasse MSAParticle, die als Erstes behandelt wird. Es gibt zwei Konstanten, die in Zeile 8 und 9 deklariert wurden: **MOMENTUM** (Impuls) und **FLUID_FORCE** (Kraft). Da bereits die Funktion zum Hinzufügen der Partikeln verwendet worden ist, ist bekannt, dass es eine Position und Geschwindigkeiten in Richtung **x** und **y** gibt.

```
08     final static float MOMENTUM = 0.5;
09     final static float FLUID_FORCE = 0.6;
10
11     float x, y;
12     float vx, vy;
13     float radius;      // particle's size
14     float alpha;
15     float mass;
16
17     void init(float x, float y) {
18         this.x = x;
19         this.y = y;
20         vx = 0;
21         vy = 0;
22         radius = 5;
23         alpha = random(0.3, 1);
24         mass = random(0.1, 1);
25     }
```

Zusätzlich zu diesen Float Variablen kommt eine Variable **radius** vor, der die Größe des Partikels bestimmt, einen Transparenz-Wert, der bestimmt, wie sehr ein Partikel zurzeit sichtbar ist und in Zeile 15 eine Variable für die Masse. Die Funktion **init()** in Zeile 17 weist diesen Variablen einen Wert zu, die Position kommt aus den Parametern zu Stande, die Kräfte werden auf 0 gesetzt, der Radius standardmäßig auf 5 und der Alpha-Wert und die Masse bekommen einen zufälligen Wert zwischen 0.3 beziehungsweise 0.1 bis 1.

Nun folgt die Funktion **update()**. Obwohl der Rückgabewert der Funktion `void` ist, wird sie in Zeile 31 verlassen, falls der Alpha-Wert 0 ist, also der Partikel unsichtbar ist. Dies zeigt, dass `void` eindeutig zu den Rückgabewerten gehört, obwohl er nichts zurückgibt. In Zeile 32 bis 35 werden die Fluid-Informationen gelesen und die Geschwindigkeit hinzugefügt, der Befehl **getIndexFromNormalizedPosition()** gibt einen Fluid-Zellenindex zurück für die normalisierten Positionskoordinaten. Daher wird **x** mit der invertierten Breite des Fensters multipliziert, analog dazu die **y**-Koordinate. Dann wird in Zeile 38 bis 39 die Position geändert, in dem die Geschwindigkeiten dazugezählt werden.

```
28 void update() {
29     // only update if particle is visible
30     if(alpha == 0) return;
31
32     // read fluid info and add to velocity
33     int fluidIndex = fluidSolver.getIndexForNormalizedPosition(x * invWidth, y * invHeight);
34     vx = fluidSolver.u[fluidIndex] * width * mass * FLUID_FORCE + vx * MOMENTUM;
35     vy = fluidSolver.v[fluidIndex] * height * mass * FLUID_FORCE + vy * MOMENTUM;
36
37     // update position
38     x += vx * (beat.isKick() ? 2 : 1);
39     y += vy * (beat.isKick() ? 2 : 1);
40
41     // bounce off edges
42     if(x < 0) {
43         x = 0;
44         vx *= -1;
45     }
46     else if(x > width) {
47         x = width;
48         vx *= -1;
49     }
50
51     if(y < 0) {
52         y = 0;
53         vy *= -1;
54     }
55     else if(y > height) {
56         y = height;
57         vy *= -1;
58     }
59
60     // hackish way to make particles glitter when they slow down a lot
61     if(vx * vx + vy * vy < 1) {
62         vx = random(-1, 1);
63         vy = random(-1, 1);
64     }
65
66     // fade out a bit (and kill if alpha == 0);
67     alpha *= 0.999;
68     if(alpha < 0.01) alpha = 0;
69
70 }
71 }
```

Bevor dies jedoch geschieht werden die Geschwindigkeiten mit 2 multipliziert, sollte die BeatDetector-Klasse einen „kick“ erkannt haben. Es ist nach dem Update möglich, dass der Partikel den Bildschirm auf einer Seite verlassen hat und daher müssen die Geschwindigkeiten für den nächsten Durchlauf umgedreht werden. Die Vorgehensweise ist wie folgt in den Zeilen 42 bis 58: Zuerst wird in einer If-Abfrage überprüft, ob der Wert für x oder y zu klein beziehungsweise zu groß ist. Sollte der Fall eingetreten sein, wird die Koordinate auf jene Koordinate umgeändert, an der der Partikel den Bildschirm verlassen hat. Dann wird die dazugehörige Geschwindigkeit umgedreht. In den Zeilen 60 bis 64 befindet sich ein Trick, wie man die Partikel zum Funkeln bringen kann, wenn sie sehr langsam werden.

Wenn $vx^2 + vy^2 < 1$ ist, dann werden die beiden Geschwindigkeiten auf einen Zufallswert zwischen -1 und 1 gesetzt. Wenn man Music Visual startet, sieht man, dass die Partikel ein bisschen schwächer werden, da in Zeile 67 der Alpha-Wert mit 0.999 multipliziert wird. Da-

mit am Ende der Alpha-Wert nicht 0.0000000.... lautet, wird er, wenn er kleiner als 0.01 ist, automatisch auf 0 gesetzt.

updateVertexArrays ist eine weitere Funktion dieser Klasse. Ein Vertex ist normalerweise ein Punkt in einem Vieleck (Polygon). Sie werden meistens in einem Array gespeichert, da man dann mit einer Schleife alle Punkte der Figur zeichnen kann. Übergeben wird die Indexposition, ab der die Werte eingefügt werden sollen und zwei FloatBuffer, die aus dem Java Paket *java.nio.FloatBuffer* stammen. Es werden vier Werte gespeichert, also ist die Anfangsposition $i * 4$.

Beispiel: Es wird angenommen, dass n die Anzahl der bereits gespeicherten Werte ist, i die Anzahl der Wertpakete und vi die zu berechnende Anfangsposition.

$n=8$ daher $i=2$ (da $8/4=2$)

Berechnung durch i :

$vi = i * 4 = 8$, daraus lässt sich wieder n schließen, da $vi=n$

Es werden hier vier Werte gespeichert, da jeder Punkt zwei Koordinaten und auch jeder Partikel zwei Punkte besitzt, da einer der beiden Punkte der Punkt mit der alten Position ist. Es wird der Befehl **put()** verwendet, um folgende Werte einzuspeichern: die alte X-Position (die hinzugefügte Geschwindigkeit wird wieder subtrahiert), die alte Y-Position, die neue X- und die neue Y-Position.

```

72     void updateVertexArrays(int i, FloatBuffer posBuffer, FloatBuffer colBuffer) {
73         int vi = i * 4;
74         posBuffer.put(vi++, x - vx);
75         posBuffer.put(vi++, y - vy);
76         posBuffer.put(vi++, x);
77         posBuffer.put(vi++, y);
78
79         int ci = i * 6;
80         colBuffer.put(ci++, alpha);
81         colBuffer.put(ci++, alpha);
82         colBuffer.put(ci++, alpha);
83         colBuffer.put(ci++, alpha);
84         colBuffer.put(ci++, alpha);
85         colBuffer.put(ci++, alpha);
86         colBuffer.put(ci++, alpha);
87     }
88

```

Die Indexvariable vi wird dabei als ersten Parameter eingesetzt und inkrementiert. Ähnliches wird mit dem Alpha-Wert durchgeführt, es gibt wieder eine Indexvariable, die diesmal ci genannt und sechs Mal der Alpha-Wert im *colBuffer* gespeichert wird. Als letztes besitzt diese Klasse eine einfache Funktion **drawOldSchool()**, mit der die Partikel auf eine klassische Weise gezeichnet werden.

Es wird als Parameter das OpenGL Objekt *GL* übergeben, das Zugriff auf die Standardfunktionen von OpenGL bietet. Mit dem Befehl **glColor3f()** wird in Zeile 91 die Farbe auf den Alpha-Wert (ein Grauwert in diesem Fall) gesetzt. Dann wird in den folgenden zwei Zeilen von der letzten zur aktuellen Position eine Linie gezogen.

MSA Fluid – Particlesystem

Bevor mit dem Particlesystem losgelegt werden kann, wird das Paket `com.sun.opengl.util`. für einige Funktionen benötigt. Es wird in Zeile 8 eingebunden. Das Partikelsystem enthält einige Variablen, die bereits aus der Particle Klasse bekannt sind. In Zeile 24 und 25 sind die beiden FloatBuffer deklariert, in Zeile 28 gibt es eine Integer Variable `curlIndex`. Die Partikeln werden in Zeile 30 in einem Array mit dem Namen `particles` gespeichert, die maximale Anzahl an Partikeln wird in Zeile 27 mit einer Maximalmenge von 5000 festgelegt. Nach den Variablen erfolgt, wie bereits gewohnt, der Konstruktor, der parameterlos ist. Dem Array wird in Zeile 24 die maximale Anzahl an Particle Objekten in die eckigen Klammern geschrieben, die man allgemein „Felder“ nennen kann.

```
22  
23 class ParticleSystem {  
24     FloatBuffer posArray;  
25     FloatBuffer colArray;  
26  
27     final static int maxParticles = 5000;  
28     int curIndex;  
29  
30     Particle[] particles;  
31  
32     ParticleSystem() {  
33         particles = new Particle[maxParticles];  
34         for(int i=0; i<maxParticles; i++) particles[i] = new Particle();  
35         curIndex = 0;  
36  
37         posArray = BufferUtil.newFloatBuffer(maxParticles * 2 * 2); // 2 coordinates per point, 2 per  
38         colArray = BufferUtil.newFloatBuffer(maxParticles * 3 * 2);  
39     }  
40 }
```

Der Index wird auf 0 gesetzt, da es noch keine Partikel gibt. Die beiden FloatBuffer werden ebenfalls initialisiert mit Hilfe der Funktion `newFloatBuffer()`, aus der Java Hilfsklasse `BufferUtil`. Als Argument wird die maximale Anzahl an Koordinaten mitgegeben, die sich so berechnet:

`size=maxParticles * 2 * 2`

Die nächste Funktion `updateAndDraw()()` führt die Änderungen an den Partikeln durch und zeichnet sie mit Hilfe des OpenGL Graphics Objekt von Processing. Um es zu erzeugen, wird es von PGraphics Objekt g, das standardmäßig in jedem Sketch existiert, in ein PGraphicsOpenGL Objekt gecastet.

Um ein Objekt der benötigten Klasse GL zu erzeugen, wird die Funktion `beginGL()` der neuen Variable `pgl` verwendet.

```

42 void updateAndDraw() {
43     PGraphicsOpenGL pg1 = (PGraphicsOpenGL) g;           // processings opengl graphics object
44     GL gl = pg1.beginGL();                                // JOGL's GL object
45
46     gl.glEnable( GL.GL_BLEND );                          // enable blending
47     if(!drawFluid) fadeToColor(gl, 0, 0, 0, 0.05);
48

```

Um Farben mischen zu können wir mit **glEnable()** mit dem Argument **GL.GL_BLEND** blending aktiviert. glEnable() aktiviert bestimmte Fähigkeiten von OpenGL, es können dabei sehr viele verschiedene Argumente angegeben werden. In Zeile 47 wird überprüft, ob gerade die Partikel gezeichnet werden, wenn nicht, werden sie nach schwarz abgeblendet mit der Hilfsfunktion **fadeToColor()**. Sie erwartet als Parameter das GL Objekt, die Farbe in die abgeblendet werden soll in RGB-Form und die Geschwindigkeit. In Zeile 12 wird die Blendvariante mit **glBlendFunc()** angegeben. Der erste Parameter gibt an, wie der Rot-, Grün-, Blau- und Alphaanteil des Quell-, der zweite Parameter wie der des Ziel-Blendfaktors berechnet wird.

Folgende Parameter sind für die erste Variable gültig:

GL_ZERO	GL_ONE
GL_SRC_COLOR	GL_ONE_MINUS_SRC_COLOR
GL_DST_COLOR	GL_ONE_MINUS_DST_COLOR
GL_SRC_ALPHA	GL_ONE_MINUS_SRC_ALPHA
GL_DST_ALPHA	GL_ONE_MINUS_DST_ALPHA
GL_CONSTANT_COLOR	GL_ONE_MINUS_CONSTANT_COLOR
GL_CONSTANT_ALPHA	GL_ONE_MINUS_CONSTANT_ALPHA
GL_SRC_ALPHA_SATURATE	

Der zweite Parameter kann folgende Werte annehmen:

GL_ZERO	GL_ONE
GL_SRC_COLOR	GL_ONE_MINUS_SRC_COLOR
GL_DST_COLOR	GL_ONE_MINUS_DST_COLOR
GL_SRC_ALPHA	GL_ONE_MINUS_SRC_ALPHA
GL_DST_ALPHA	GL_ONE_MINUS_DST_ALPHA
GL_CONSTANT_COLOR	GL_ONE_MINUS_CONSTANT_COLOR
GL_CONSTANT_ALPHA	GL_ONE_MINUS_CONSTANT_ALPHA

Die fettgedruckten Konstanten sind die Standardwerte.

In diesem Projekt wird die Funktion wie folgt aufgerufen:

gl.glBlendFunc(GL.GL_SRC_ALPHA, GL.GL_ONE_MINUS_SRC_ALPHA)

Durch diese Kombination wird eine Transparenz erzeugt und die Ebenen werden von hinten nach vorne sortiert. In Zeile 13 wird die Farbe mit dem bereits bekannten Befehl `glColor4f` gesetzt. Die „4“ in dem Befehl heißt, dass vier Parameter erwartet werden, das „f“, dass diese vom Typ Float sein müssen. Als Alpha-Wert wird die Geschwindigkeit angeben. Darauf folgt der Befehl `glBegin()` der das Zeichnen einer Figur einleitet. Als Argument gibt es eine größere Auswahl an Formen:

<code>GL_POINTS</code>	Einzelne Punkte
<code>GL_LINES</code>	Einzelne Linien
<code>GL_LINE_STRIP</code>	Miteinander verbundene Linien
<code>GL_LINE_LOOP</code>	Wie <code>GL_LINE_STRIP</code> , das Ende wird mit dem Anfang wieder verbunden
<code>GL_TRIANGLES</code>	Einzelne Dreiecke
<code>GL_TRIANGLE_STRIP</code>	Spezielle Variante des Dreiecks
<code>GL_TRIANGLE_FAN</code>	Spezielle Variante des Dreiecks
<code>GL_QUADS</code>	Einzelne Vierecke
<code>GL_QUAD_STRIP</code>	Verbundene Vierecke
<code>GL_POLYGON</code>	Ganze Polygone

Es werden hier Vierecke gezeichnet, also wird die Konstante `GL.GL_QUADS` innerhalb der runden Klammern eingefügt. Danach werden in den folgenden Zeilen vier Punkte gezeichnet, wobei sich in jeder Ecke des Bildschirmes einer befindet. OpenGL verarbeitet die Befehle schrittweise.

```

 9
10
11 void fadeToColor(GL gl, float r, float g, float b, float speed) {
12     gl.glBlendFunc(GL.GL_SRC_ALPHA, GL.GL_ONE_MINUS_SRC_ALPHA);
13     gl.glColor4f(r, g, b, speed);
14     gl.glBegin(GL.GL_QUADS);
15     gl.glVertex2f(0, 0);
16     gl.glVertex2f(width, 0);
17     gl.glVertex2f(width, height);
18     gl.glVertex2f(0, height);
19     gl.glEnd();
20 }
21

```

Wenn die Punkte im Uhrzeigersinn oder vice versa gezeichnet werden, wird dadurch ein Rechteck über den ganzen Sketch gezogen. Nun werden mit `drawOldSchool()` die Linien gezeichnet. In Zeile 50 wird mit `glEnable()` eingestellt, dass die Punkte rund gezeichnet werden, in der nächsten Zeile wird die Liniendicke mit `glLineWidth()` auf 1px geändert. Der nächste Befehl ist altbekannt, diesmal wird `glBegin()` die Konstante `GL.GL_LINES` übergeben.

```

49     gl.glBlendFunc(GL.GL_ONE, GL.GL_ONE); // additive blending (ignore alpha)
50     gl.glEnable(GL.GL_LINE_SMOOTH); // make points round
51     gl.glLineWidth(1);
52     gl.glBegin(GL.GL_LINES); // start drawing points
53     for(int i=0; i<maxParticles; i++) {
54       if(particles[i].alpha > 0) {
55         particles[i].update();
56         particles[i].drawOldSchool(gl); // use oldschool rendering
57       }
58     }
59   gl.glEnd();
60
61   gl.glDisable(GL.GL_BLEND);
62   pgl.endGL();
63
64
65 }
```

Jetzt wird das Particle Array durchlaufen und dabei überprüft ob der Alpha-Wert der einzelnen Partikel größer als 0 ist und gegebenfalls werden die durchgenommenen Funktionen `update()` und `drawOldSchool()` in den Zeilen 55 bis 56 mit den entsprechenden Parametern aufgerufen. Schlussendlich wird in Zeile 59 GL mit `glEnd()` beendet, genauso wie in Zeile 63 bei `pgl` mit `endGL()`. Da alle aktivierten Funktionen auch deaktiviert werden sollten, wird die Blendfunktion vorher noch in Zeile 62 mit `glDisable(GL.GL_BLEND)` ausgeschalten.

Um Partikel in das Array aufzunehmen, wurde eine Funktion `addParticle()` geschrieben. Eine weitere Funktion `addParticles()` ist dafür gedacht, dass, wenn mehrere Partikel hinzugefügt werden sollen, diese Funktion in einer Schleife die Funktion `addParticle()` so oft aufruft, wie es im Parameter `count` steht.

```

66
67 void addParticles(float x, float y, int count ) {
68   for(int i=0; i<count; i++) addParticle(x + random(-15, 15), y + random(-15, 15));
69 }
70
71
72 void addParticle(float x, float y) {
73   particles[curIndex].init(x, y);
74   curIndex++;
75   if(curIndex >= maxParticles) curIndex = 0;
76 }
77 }
```

Die x- und y-Position wird dabei in Zeile 68 mit `random()` um bis zu 15 Pixel verschoben. Die eigentliche Funktion in Zeile 7 ruft an dem Arrayindex `curIndex` im Array `particles` die Funktion `init()` mit der übergebenen Position auf. Dann wird `curIndex` inkrementiert und es erfolgt eine Abfrage, ob schon zu viele Partikel im Array sind. Wenn dies der Fall ist, beginnt der Index in Zeile 75 wieder bei 0 zu zählen.

MSA Fluid – Hauptklasse

Betrachtet man die Variablen Deklarationen, sind die Variablen in Zeile 10 und 11 schon bekannt, in Zeile 8 wird die Breite der Fluidsimulation in der Konstante FLUID_WIDTH festgelegt. In Zeile 15 befindet sich das Partikelsystem, in Zeile 19 eine boolesche Variable, die auf true gesetzt ist. Besonders ins Auge sticht die Klasse in Zeile 13, ein Variable der Klasse MSAFluidSolver2D. Sie ist die Hauptklasse der MSA Fluid Bibliothek, denn ohne sie wäre sozusagen gar nichts möglich. Genauso auffällig ist das **PImage** Objekt in Zeile 17. Es wurde zwar erklärt, dass MSAFluid mit OpenGL berechnet und gezeichnet wird, ganz stimmt das jedoch nicht. Das Ergebnis wird in das Bild *imgFluid* gezeichnet und dann erst angezeigt.

In der nachfolgenden Funktion wird eine Kraft zur Simulation hinzugefügt, einfacher gesagt: neue Partikeln kommen hinzu. **mouseDragged()** ist eine Funktion, die von Anfang an genau so wie **setup()** und **draw()** verwendet werden kann um Befehle auszuführen, wenn die Maus gedrückt und bewegt wird. Im Englisch spricht man hier von „drag“, bekannt geworden durch „Drag and Drop“.

```

21 void mouseDragged() {
22     float mouseNormX = mouseX * invWidth;
23     float mouseNormY = mouseY * invHeight;
24     float mouseVelX = (mouseX - pmouseX) * invWidth;
25     float mouseVelY = (mouseY - pmouseY) * invHeight;
26     addForce(mouseNormX, mouseNormY, mouseVelX, mouseVelY);
27 }
28
29 }
```

Um eine Kraft hinzuzufügen, müssen alle Argumente normalisiert sein. Daher wird die aktuelle Mausposition mit invWidth beziehungsweise invHeight multipliziert. Die Geschwindigkeit ergibt sich aus $\text{mouseX} - \text{pmouseX}$ und $\text{mouseY} - \text{pmouseY}$. Beide Ergebnisse werden gleich wie die Mausposition normalisiert. Nun kann der Befehl **addForce()** in Zeile 27 mit den normalisierten Mausposition und Geschwindigkeiten aufgerufen werden. In Zeile 31 beginnt die Funktion **addForce()**. Gleich am Anfang wird durch eine einfache mathematische Rechnung eine gute Balance zwischen den X- und Y-Komponenten der Geschwindigkeit mit der Seitenverhältnis des Bildschirmes berechnet.

Die Formel laut: $\text{speed} = \text{dx}^2 + \text{dy}^2 * \text{aspect_radio}^2$

Wenn die Geschwindigkeit größer als 0 ist, wird in Zeile 35 mit einer If-Abfrage fortgesetzt. Wenn x und y nicht im Bereich 0 bis 1 liegen, werden sie in den Zeilen 35 bis 38 an die nächste Grenze angepasst. Also wenn x zum Beispiel kleiner als 0 ist, wird x der Wert 0 zugewiesen. In Zeile 40 und 41 wird jeweils ein Faktor für die Farbwerte und die Geschwindigkeit in

die Variablen `colorMult` und `velocityMult` geschrieben, die gleich verwendet werden. Wie bereits bei einem früheren Codeabschnitt wird hier die Funktion `getIndexForNormalizedPosition()` des Objekts `fluidSolver` benötigt.

Im Kapitel graphische Oberfläche wurde bereits eine alternative Farbvariable beschrieben, die `gui_hui` heißt. Die neue Farbe wird in das Color Objekt `drawColor` gespeichert, das in Zeile 45 deklariert wurde. Der Farbmodus wird auf die HSB-Palette umgeändert, wobei der Farbton einen Wert von 0 bis 360 und die Sättigung und Helligkeit von 0 bis 100 annehmen kann. Dann erfolgt eine verkürzte If-Abfrage, bei der die Variable `onecolor` bestimmt, ob die ausgewählte Farbe oder eine Farbe nach der Formel berechnet werden soll.

Die Formel lautet: `hue= ((x+y) *180+frameCount)%360`

frameCount enthält die Anzahl der Frames, die bis zum jetzigen Zeitpunkt ausgeführt wurden. Der Modulo-Operator wird in dieser Zeile verwendet um einen Wert zwischen 0 und 360 zu bekommen, da der Rest bei einer Division durch 360 nie kleiner als 0 und größer als 360 sein kann. In Zeile 49 wird aus der berechneten Variable und zwei weiteren Variablen der Benutzeroberfläche eine neue Farbe zusammengesetzt mit der Funktion `color()`. Danach muss der Farbmodus auf RGB umgestellt werden, im Bereich 0 bis 1. Processing und Java verwendeten zwar Werte von 0 bis 255, bei OpenGL gibt es jedoch nur Werte von 0 bis 1, daher ist diese Änderung unumgänglich. Die einzelnen Farbtöne Rot, Grün und Blau werden mit den Funktionen `red()`, `green()` und `blue()` extrahiert und werden für kräftigere Farben mit dem Farbmultiplikator multipliziert. Dann werden die Werte mit `+=` an der Stelle index in die Arrays `rOld`, `gOld` und `bOld` hinzugefügt. In Zeile 56 wird im Partikelsystem `addParticles()` aufgerufen, wobei insgesamt 10 neue Partikel erstellt werden. Analog zu den Farben geschieht es mit den Geschwindigkeiten, die vorgesehenen Arrays heißen `uOld` und `vOld`.

draw()

Last but not least, die Funktion `draw()`. Sie ist das Kernstück eines jeden Sketches. Ohne sie gäbe es keine Bewegungen, interaktive Elemente und vieles mehr. Es gibt allerdings auch Sketches, die nur aus einer `setup()` Funktion und anderen Funktionen bestehen, die zum Beispiel die fraktal erscheinende Menge Mandelbrot berechnen.

Nach dem Befehl `background(0)` erfolgt hier die Aufbereitung der gesammelten Drag & Drop-Daten, die das `dropEvent()` in Zeile 166 in verschiedene Daten aufgeteilt hat. Sollte eine Datei in den Sketch gezogen worden sein, wurde die Variable `dropped` auf `true` gesetzt, da hier wird in Zeile 94 eine entsprechende If-Abfrage gemacht. Bei Eintreten des Falles wird die Variable wieder auf `false` gesetzt, damit beim nächsten Durchlauf dieses Programmteiles der

Code nicht noch einmal ausgeführt wird. Es gibt drei Variablen die Informationen enthalten können:

dropfilepath	Pfad zu einer Datei
dropfile	Datei
dropfileurl	Pfad zu einer Datei im Internet

dropfile und *dropfileurl* wurden bereits bei der Initialisierung mit Null versehen, *dropfilepath* wurde ein leerer String zugewiesen.

```

91 void draw() {
92   background(0);
93   // Drag & Drop
94   if (dropped) {
95     dropped=false;
96     if (dropfilepath!="") {
97       newSong(dropfilepath);
98       dropfilepath="";
99     }
100    if (dropfile!= null) {
101      newSong(dropfile.toString());
102      dropfile=null;
103    }
104    if (dropfileurl != null) {
105      newSong(dropfileurl.toString());
106      dropfileurl=null;
107    }
108 }
```

Daher kann der Pfad in einer If-Abfrage überprüft werden, ob er einen Text enthält und gegebenfalls die Funktion `newSong()` mit dem entsprechenden Parameter aufgerufen werden. *dropfilepath* wird wieder resetet, indem ein leerer String zugewiesen wird. Ähnlich verhält es sich mit den beiden Referenztypen der Klasse File und URL, wobei überprüft wird, ob sie gleich Null sind. Sie werden auch wieder auf null gesetzt, wenn die Bedingung wahr ist. Da die Methode `newSong()` nur mit einem String aufgerufen werden kann, muss in Zeile 101 und 105 die Funktion `toString()` aufgerufen werden, die jedes Objekt in Processing besitzt.

Bevor ab Zeile 109 der Abschnitt mit der MSA Fluid Bibliothek erklärt werden kann, müssen ein paar Funktionen, die früher in der Funktion `newSong()` von Zeile 233 bis 236 ausgelassen worden sind, nachgeholt werden.

In Zeile 233 wird ein neues Objekt der Klasse `MSAFluidSolver2D` instanziiert, wobei die Breite und Höhe angegeben wird. Die Breite ist in der Konstante `FLUID_WIDTH` festgelegt, die sich im Tab MSA gleich am Anfang befindet. Die Höhe errechnet sich aus dieser Konstante mal die Höhe des Sketches, dividiert durch die Breite. Wenn diese Variable angepasst wird,

ist darauf zu achten, dass der Wert nicht zu hoch gesetzt wird, da die zusätzlich anfallenden Berechnungen den Sketch verlangsamen. Der Standardwert ist 120 und kann zum Beispiel auf 60 halbiert oder auf 320 erhöht werden. Darauf folgend wird die Option **enableRGB()** des Objektes *fluidSolver* auf true gesetzt, um die RGB-Farbpalette zu aktivieren. Die Funktion gibt wieder das Objekt zurück, von dem es aufgerufen wurde und daher kann gleich mit einem Punkt getrennt, die nächste Methode aufgerufen werden. **setFadeSpeed()** ist eine optionale *setup()* Methode (genauso wie *enableRGB()*) und bestimmt wie schnell die Farbe der Fluidsimulation sich aufgelöst und ausgeblendet wird. Die darauffolgende Funktion **setDeltaT()** fügt einen Zeitschritt zu *fluidSolver* hinzu. **setVisc()** setzt den Wert der Viskosität, also das Maß für die Zähflüssigkeit eines Fluids. Das Gegenteil der Viskosität ist die Fluidität, ein Maß für die Fließfähigkeit eines Fluids. Je größer die Viskosität, desto weniger fließfähig ist das Fluid. In der letzten Zeile wird noch ein neues Partikelsystem erstellt.

Perlin Noise⁴

Um pseudo-zufällige Kräfte in der Simulation zu platzieren, wird die Technik Perlin Noise verwendet. Das ist ein vom Computer generierter, visueller Effekt, der von Ken Perlin 1982 für den Film „Tron“ entwickelt worden ist. Er bekam dafür den Oscar für Technical Achievement von der Academy of Motion Arts and Sciences. Der Effekt kann verwendet werden, um Elemente aus der Nature zu simulieren, besonders nützlich ist er wenn der Computerspeicher limitiert ist. Perlin Noise ist eine prozedurale Texturengrundform, die bei Visual-Effects-Künstlern benutzt wird, um den Realismus in Computergrafiken zu steigern. Die Funktion ist nicht wirklich zufällig, da alle Details die gleiche Größe haben. Durch diese Eigenschaft sind die Grafiken leicht regulierbar.

Perlin Noise ist im Allgemeinen eine Funktion mit den Parametern (x,y), (x,yz) oder (x,y,z,time), kann aber für jede Anzahl an Dimension definiert werden. Processing unterstützt die Funktion nur bis zur dritten Dimension. Bei Perlin Noise werden Frequenzen überlagert, der Zufallsfaktor ist abhängig von der Frequenz. Je höher die Frequenz ist, desto näher entfernt vom letzten Pixel wird ein neuer Wert berechnet. Je niedriger die Frequenz, desto mehr Werte müssen zwischen zwei Pixeln interpoliert werden, dadurch entsteht ein sogenannter „flacher Gradient“.

Music Visual – Zeile 110 bis 127

Zuerst wird überprüft, ob die Fluidsimulation überhaupt laufen soll, danach ob die zufälligen Kräfte hinzugefügt werden müssen. Um die Funktion **noise()** benutzen zu können, wird eine Variable benötigt, deren Wert sich ändert. In Zeile 39 und 50 wurden daher bereits die Vari-

⁴ Basierend auf <http://de.wikipedia.org/wiki/Perlin-Noise>

ablen *sourceXOff* und *sourceYOff* definiert. Bei jedem Durchlauf werden die beiden Variablen erhöht, wobei der Wert der Erhöhung kleiner als 1 sein sollte. Werte zwischen 0.005-0.03 sind erfahrungsgemäß eine gute Wahl. Danach wird die Funktion *noise()* angewendet, indem die vorher erwähnten Variablen als Argument übergeben werden und das Ergebnis mit der Länge und Breite des gewünschten Bereiches multipliziert wird.

```

109 // general
110 if (!fluid_pause) {
111   if (MSA_STANDART_FORCES) {
112     sourceXOff+=SOURCE_NOISE_INC;
113     sourceYOff+=SOURCE_NOISE_INC;
114     curSourceX=noise(sourceXOff)*width;
115     curSourceY=noise(sourceYOff)*height;
116     mouseNormX = curSourceX * invWidth;
117     mouseNormY = curSourceY * invHeight;
118     mouseVelX = (curSourceX - lastSourceX) * invWidth;
119     mouseVelY = (curSourceY - lastSourceY) * invHeight;
120     for (int i=0;i<MSA_NEW_FORCES;i++)
121       addForce(mouseNormX, mouseNormY, mouseVelX, mouseVelY);
122     lastSourceX=curSourceX;
123     lastSourceY=curSourceY;
124   }
125 // msa fluid
126 updateTUIO();
127 fluidSolver.update();
128

```

In Zeile 115 und 116 wurde mit *width* bzw. *height* multipliziert, in der Praxis werden diese Werte aber nicht erreicht. Das Ergebnis wird in die Variablen *curSourceX* und *curSourceY* gespeichert. Die Befehle in den Zeilen 116 bis 119 wurden bereits in einer anderen Funktion benutzt, man hätte diesen Teil auch in einer eigenen Funktion zusammenfassen können, genauso wie das Hinzufügen der Kräfte. Der einzige Unterschied ist hierbei die Berechnung der Geschwindigkeit, indem hier die letzte „zufällige“ Position in die Variablen *lastSourceX* und *lastSourceY* gespeichert wurde. In Zeile 126 werden die OSC-Eingaben verarbeitet und deswegen wird *fluidsolver* in Zeile 127 aktualisiert.

Music Visual – Zeile 129 bis 162

In Zeile 130 wird die Funktion **getNumCells()** verwendet, um mit Hilfe einer Schleife alle Pixel vom *fluidsolver* in das *PlImage* Objekt *imgFluid* zu übertragen. Damit die Farben heller sind werden sie mit 2 multipliziert, der konstante Wert ist in der Variable *d* gespeichert. Auf die einzelnen Pixel greift man bei *fluidSolver* Objekten und der Klasse *PlImage* durch das Array **pixels** zu, wobei zwei zusätzlich Befehle vonnöten sind: Zum einen müssen Pixel geladen werden, wenn sie an einen anderen Ort kopiert werden, wie es in Zeile 132 der Fall ist. In *fluidsolver* sind die Pixel anders gespeichert als in der Klasse *PlImage*, hier müsste man zusätzlich den Befehl **loadPixels()** verwenden. In Zeile 134 kommt der zweite Befehl ins Spiel,

der die Pixel ändert, er nennt sich **updatePixels()**. Da die Pixel geändert wurden, kann das Bild gezeichnet werden. Da das Problem mit der grafischen Oberfläche schon ausführlich diskutiert wurde, ist der Befehl **draw()** in Zeile 138 klar, da in **setup()** eingestellt wurde, dass das Menü nicht automatisch gezeichnet wird. Würde man die Reihenfolge der beiden letzten Befehle tauschen, gäbe es wieder Probleme mit dem Menü. Zuletzt wird auch das Partikel-system geändert und neu gezeichnet und zwar in Zeile 140.

```

129     if (drawFluid) {
130         for (int i=0; i<fluidSolver.getNumCells(); i++) {
131             int d = 2;
132             imgFluid.pixels[i] = color(fluidSolver.r[i] * d, fluidSolver.g[i] * d, fluidSolver.b[i] * d);
133         }
134         imgFluid.updatePixels();
135     }
136 } // !fluid_pause
137 image(imgFluid, 0, 0, width, height);
138 controlP5.draw(); // workaround
139 if (!fluid_pause) {
140     particleSystem.updateAndDraw();
141     // minim
142     fill(255);
143     if (beat.isKick() || beat.isSnare() || beat.isHat() ) {
144         float pseuX=random(width);
145         float pseuY=random(height);
146         mouseNormX = pseuX * invWidth;
147         mouseNormY = pseuY * invHeight;
148         mouseVelX = (pseuX - lastpseuX) * invWidth;
149         mouseVelY = (pseuY - lastpseuY) * invHeight;
150         for (int i=0;i<MSA_NEW_FORCES;i++)
151             addForce(mouseNormX, mouseNormY, mouseVelX, mouseVelY);
152         lastpseuX=pseuX;
153         lastpseuY=pseuY;
154     }
155 } // !fluid_pause
156
157 // free-rotate mode
158 if (cam.isActive()) {
159     controlP5.controller("info_button").setVisible(true);
160     controlP5.controller("info_button").captionLabel().set(LABEL_3D_ENTER);
161 }
162
163 }
```

Zusätzlich zu den mit Perlin Noise positionierten Kräften, werden auch Kräfte hinzugefügt, wenn der BeatDetector eine bestimmte Spur in der Musik erkennt. Dazu werden in Zeile 143 die Funktionen **isKick()**, **isSnare()** und **isHat()** aufgerufen. Die Zeilen 146 bis 148 sind gleich wie immer, die Position wird diesmal mit der Methode *random()* bestimmt. Alle weiteren Befehle sind fortwährend gleich geblieben.

Die letzten Funktionen in **draw()** ist eine Überprüfung, ob die Kamera aktiv ist. Wenn die Abfrage mit **true** endet, wird der Infobutton sichtbar gemacht und mit dem entsprechenden Text versehen, dass der 3D-Modus gestartet wurde.

Inputfunktionen

Zur Abschlussfunktion **keyPressed()** von Music Visual, gibt es noch eine kurze Einführung in die Möglichkeiten, Inputeingaben auszulesen.

Mit Processing selbst lassen sich Eingabe der Maus, sowie der Tastatur auslesen:

mouseClicked()	mouseDragged()
mouseMoved()	mousePressed()
mouseReleased()	keyPressed()
keyReleased()	keyTyped()
mousePressed	keyPressed

```

01 void draw() {
02   if(mousePressed) {
03     println("The mouse is been pressed!");
04   }
05 }
06
07 void mouseReleased() {
08   println("The mouse isn't pressed any more.");
09 }
10
11 void mouseDragged() {
12   println("Maybe a Drag & Drop?");
13 }
```

In der letzten Zeile sind zwei zusätzliche Variablen angeführt, die man im draw() mit Hilfe einer If-Abfrage benutzen kann. Die speziellen Variablen für Mauseingaben wurden bereits vorgestellt: *mouseX*, *mouseY*, *pmouseX* und *pmouseY*. Das „p“ in letzteren Befehlen steht für „previous“.

Für Keyboardeingaben gibt es nochmals zwei weitere Befehle. *key* enthält die letzte gedrückte Taste, mit *keyCode* lassen sich Spezialtasten wie [Shift] und [Strg] abfragen. In folgendem etwas aufwändigeren Beispiel wird deutlich, dass sich die Funktionen mit den speziellen Variablen auch kombinieren lassen:

```
01 color fillVal = color(126);
02 boolean newcolor=false;
03
04 void draw() {
05   if (keyPressed && !newcolor) {
06     if (key == 'b' || key == 'B') {
07       fill(100);
08     }
09   } else {
10     fill(fillVal);
11   }
12   rect(25, 25, 50, 50);
13 }
14
15 void keyPressed() {
16   newcolor=true;
17   if (key == CODED) {
18     if (keyCode == UP) {
19       fillVal = 255;
20     }
21     else if (keyCode == DOWN) {
22       fillVal = 0;
23     }
24   }
25   else {
26     fillVal = 126;
27     newcolor=false;
28   }
29 }
30 }
```

Mit contributed libraries gibt es eine noch größere Palette an Möglichkeiten. Man kann Eingaben verarbeiten von:

- Lego Mindstorms NXT Robotern
- Joysticks und Joypads
- Apple Sudden Motion Sensor in PowerBooks and MacBooks
- Light Sensor in MacBook Pro
- Novation's Launchpad
- Touchatag RFID Readers
- Graphics tablets

Durchlauf des Programmes

Auf den folgenden Seiten befinden sich Screenshots von Music Visual, auf denen das Programm im Presentmodus, also im Fullscreenmodus aus der Processing Umgebung heraus gestartet wurde.

Um die Screenshots zu erstellen, wurde eine If-Abfrage in `keyPressed()` hinzugefügt, die überprüft, ob die Taste „S“ gedrückt wurde. Wenn dies der Fall ist wird die Funktion `saveFrame()` mit einem Bildnamen als Argument aufgerufen. Um mehrere Screenshots erstellen zu können, können Rauten (#) in den Dateinamen eingefügt werden, damit die Bilder durchnummert werden.

```
01 if(key == 's')  
02   saveFrame("snapshot####.jpg");
```

Getestet wurde das Projekt auf einem Fujitsu Esprimo P1510, mit einem Intel Core i7 Prozessor und einer Taktrate von 2,83 GHz. Das installierte Betriebssystem ist Microsoft Windows 7 mit einer 64-Bit-Architektur.

Der Sketch lief durchgehend mit gleichbleibenden 60 Frames, der Standardeinstellung von Processing. Die maximale Frameanzahl bei 5 Versuchen waren 127 Frames. Auf alten Rechnern ist es möglich, dass einige Einstellungen verändert werden müssen, um ein flüssiges Laufen des Programms zu erreichen. Man kann zum Beispiel die Größe der Fluidsimulation oder die maximale Anzahl an Partikeln vermindern. Sollte der Fullscreen-Modus nichtig sein, kann auch das Fenster verkleinert werden. Eine Einstellung, die man immer vornehmen muss, ist die Positionierung der Kamera für die 3D-Darstellung, die experimentell an die eigenen Bedürfnisse angepasst wird. Um sie automatisch an den Bildschirm anzupassen, müsste die Position der Kamera in Bezug auf die Größe und Breite des Monitors berechnet werden.

Meldungen während der Ausführung

Während der Ausführung werden in der Konsole speziell beim Start einige Zeilen geschrieben.

„PeasyCam v092“

Diese Meldung zeigt nur, dass die Bibliothek *Peasycam* verwendet wird.

„sojamo.drop 0.1.4 infos, comments, questions at <http://www.sojamo.de/libraries/drop>“

Diese Meldung gleicht der Ersten, ausgelöst durch die Bibliothek *SDrop*.

„===== JavaSound Minim Error =====

===== Don't know the ID3 code PRIV”

Diese Meldung tritt 5-mal auf. Sie signalisiert, dass die Bibliothek *minim* oder genauer gesagt die JavaSound API ein ID3 Tag der MP3-Datei nicht erkannt hat. Eigentlich ist es keine Fehlermeldung sondern nur eine Warnung und ist abhängig von der Sounddatei.

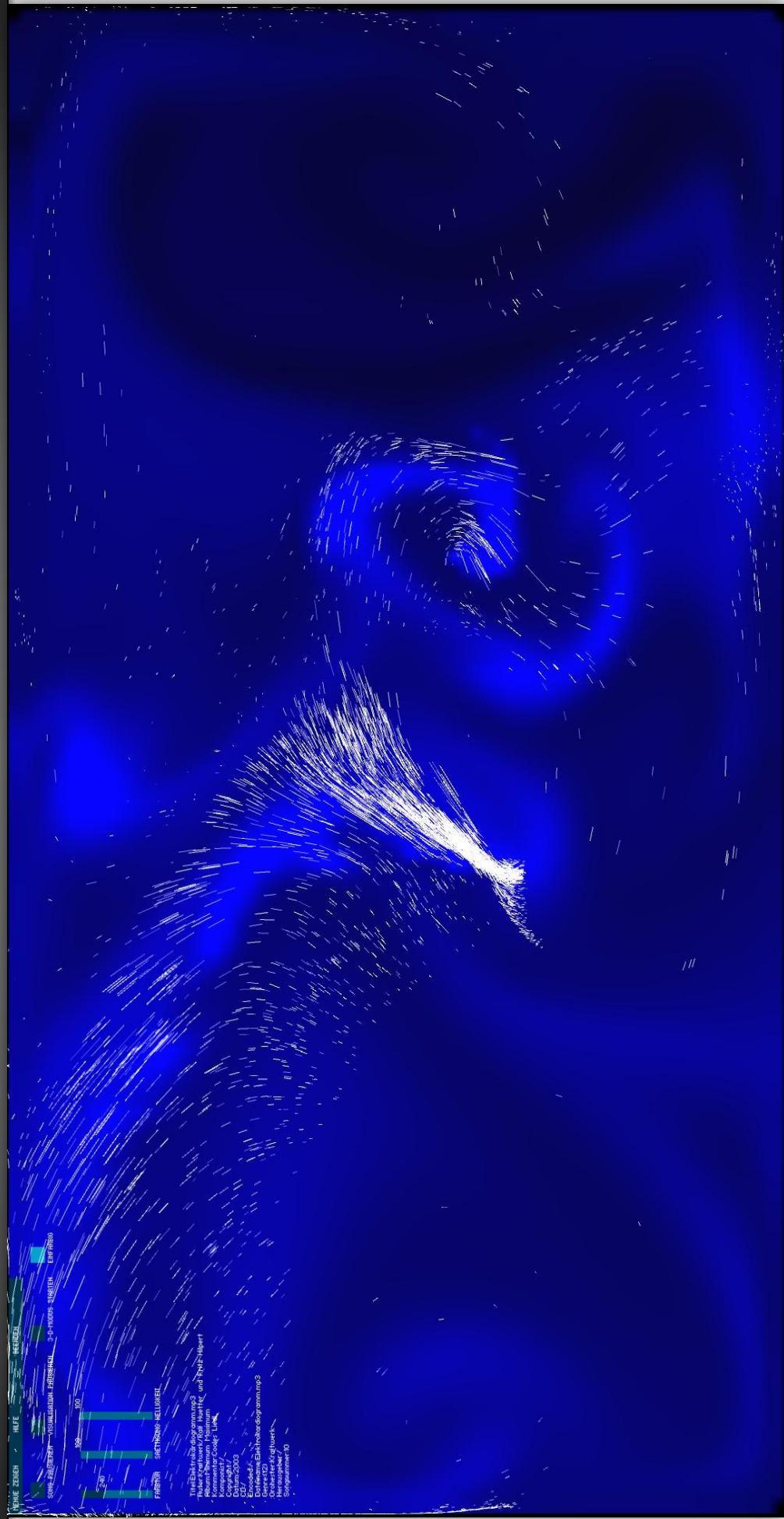
„controlP5.BitFontRenderer getWidth WARNUNG: You are using a character that is not supported by controlP5's BitFont-Renderer; you could use ControlFont instead (see the ControlP5controlFont example).“

Diese Warnung bedeutet, dass controlP5 ein Zeichen nicht anzeigen kann, da es nicht unterstützt wird. Der Fehler wird nicht durch Music Visual ausgelöst, sondern durch diese Bibliothek. Die Liste aller gültigen Zeichen:

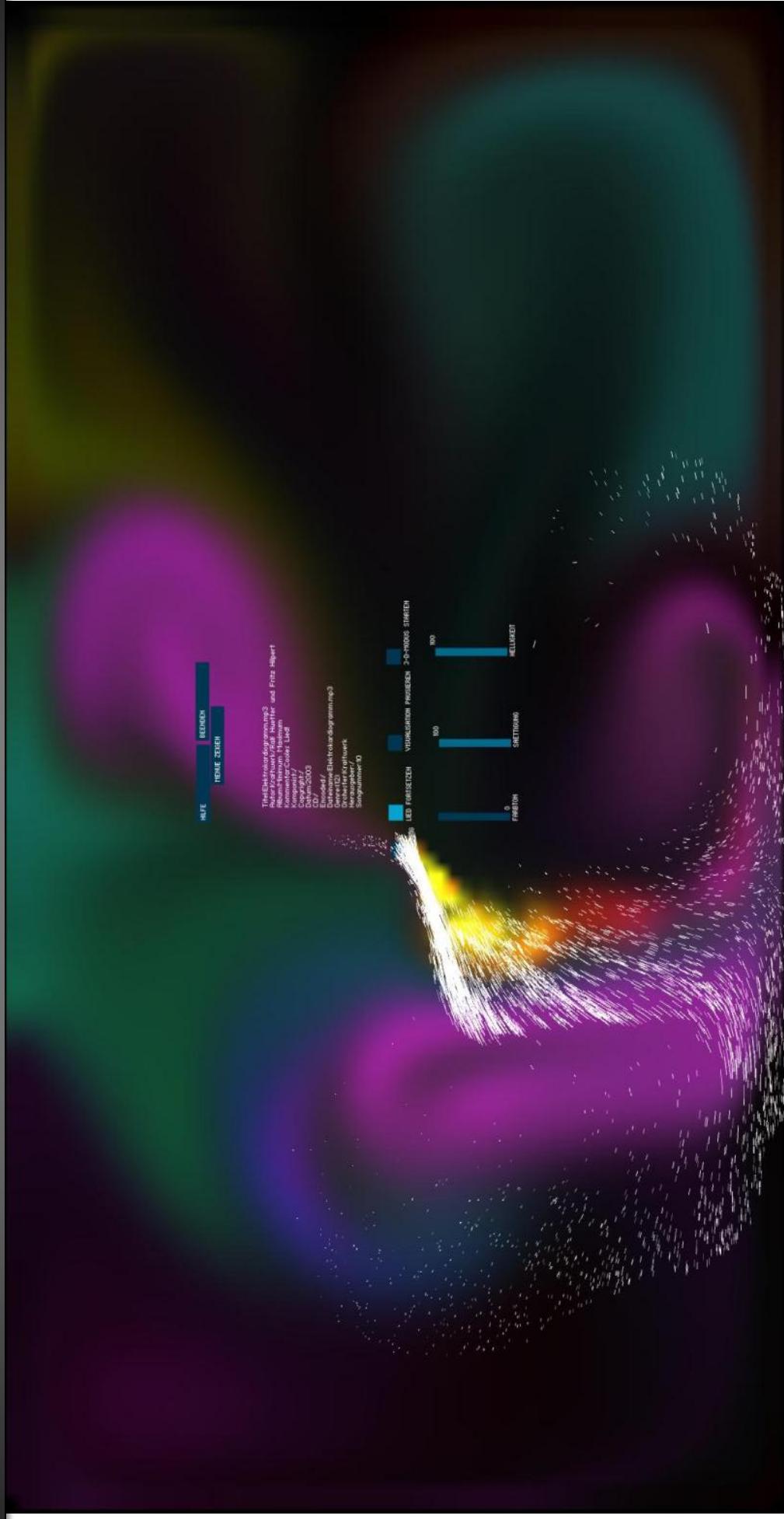
!	"	#	\$	%	&	'	(
)	*	+	,	-	.	/	0
1	2	3	4	5	6	7	8
9	:	;	<	=	>	?	@
A	B	C	D	E	F	G	H
I	J	K	L	N	O	P	Q
R	S	T	U	V	W	X	Y
Z	[\]	^	_	`	*
a	b	c	d	e	f	g	h
i	j	k	l	m	n	o	p
q	r	s	t	u	v	w	x
y	z	{		}			



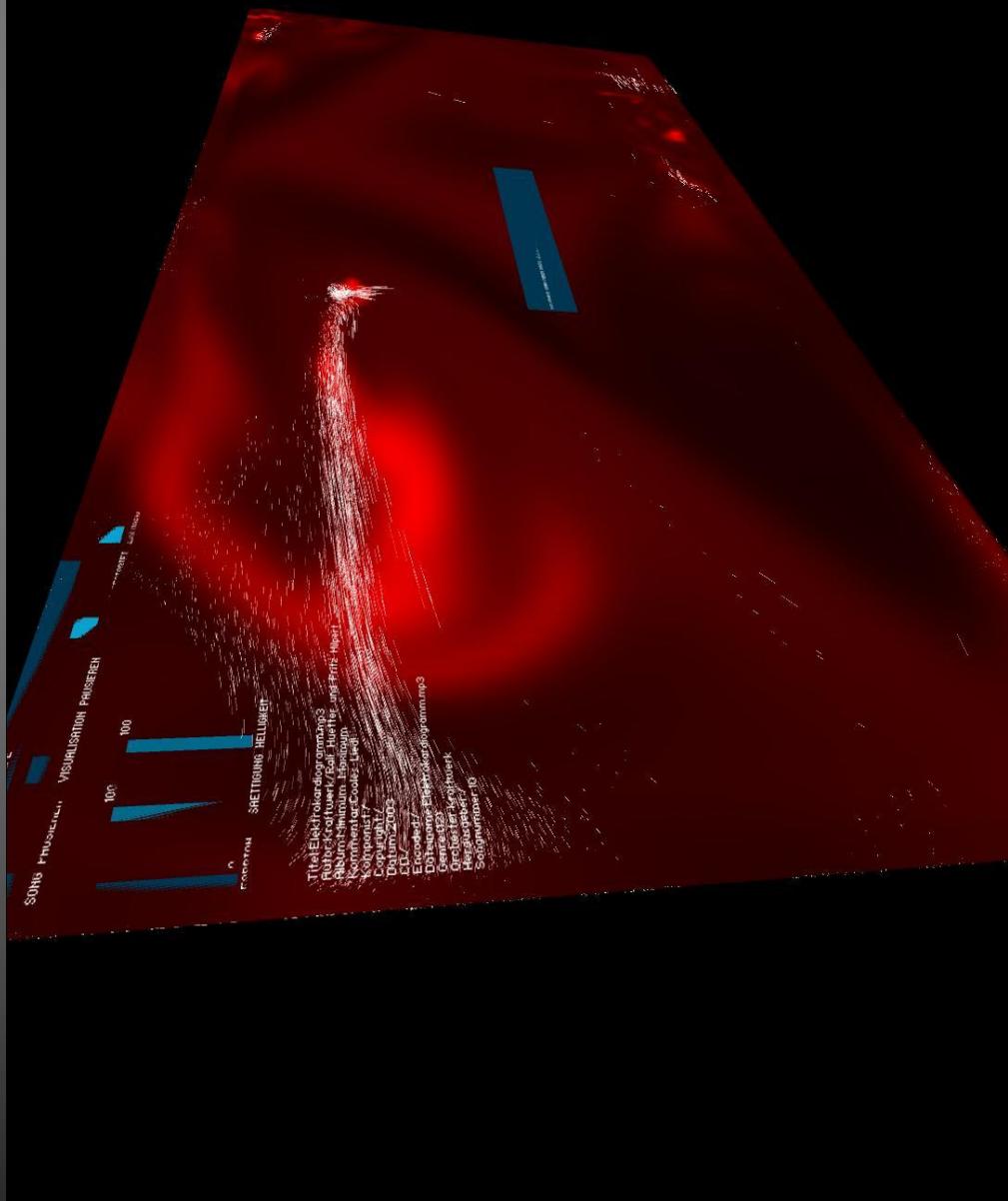
Screenshot von Music Visual mit Standard-Einstellungen



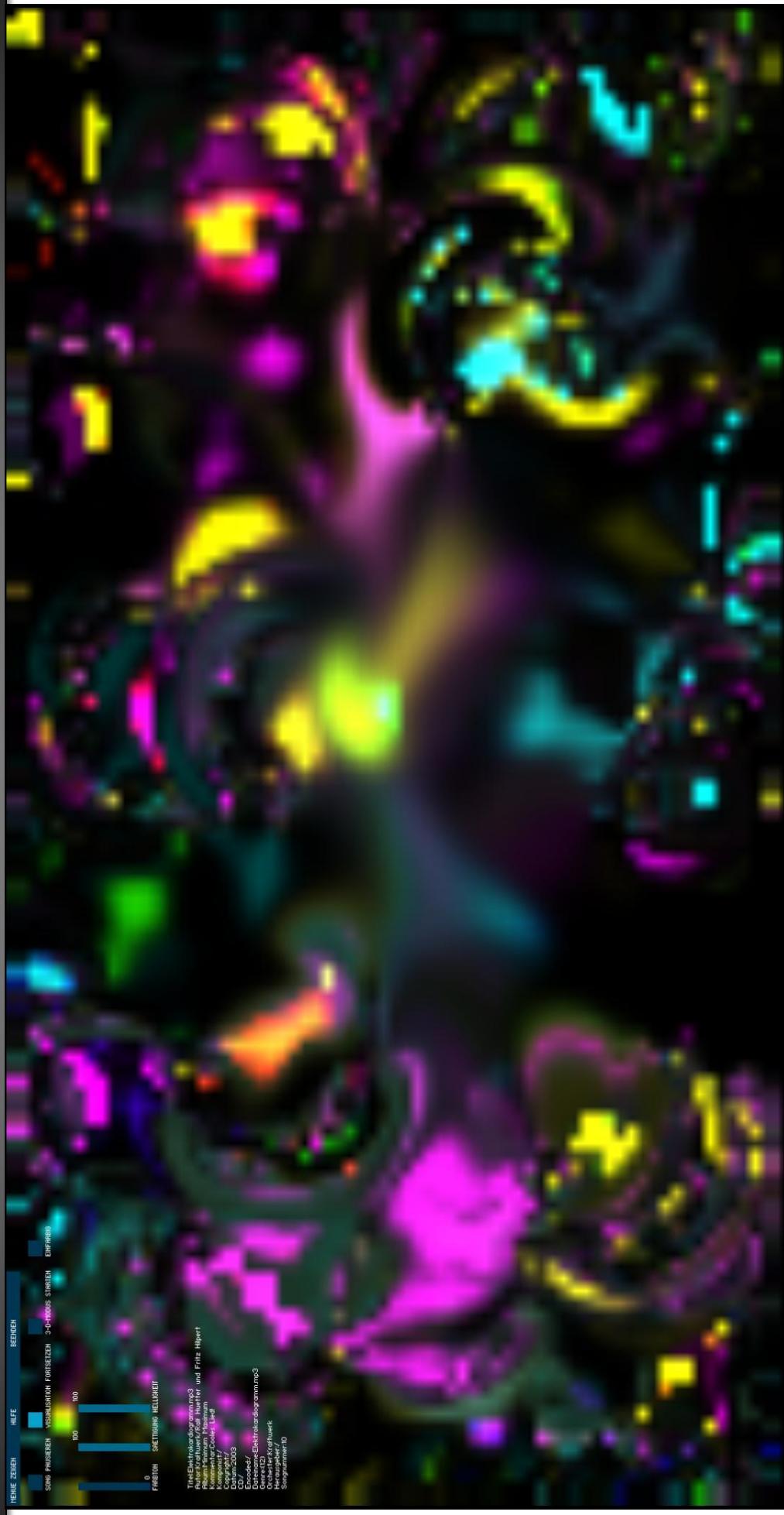
Screenshot von Music Visual mit Option „Einfarbig“ und selbst gewählten Blauton



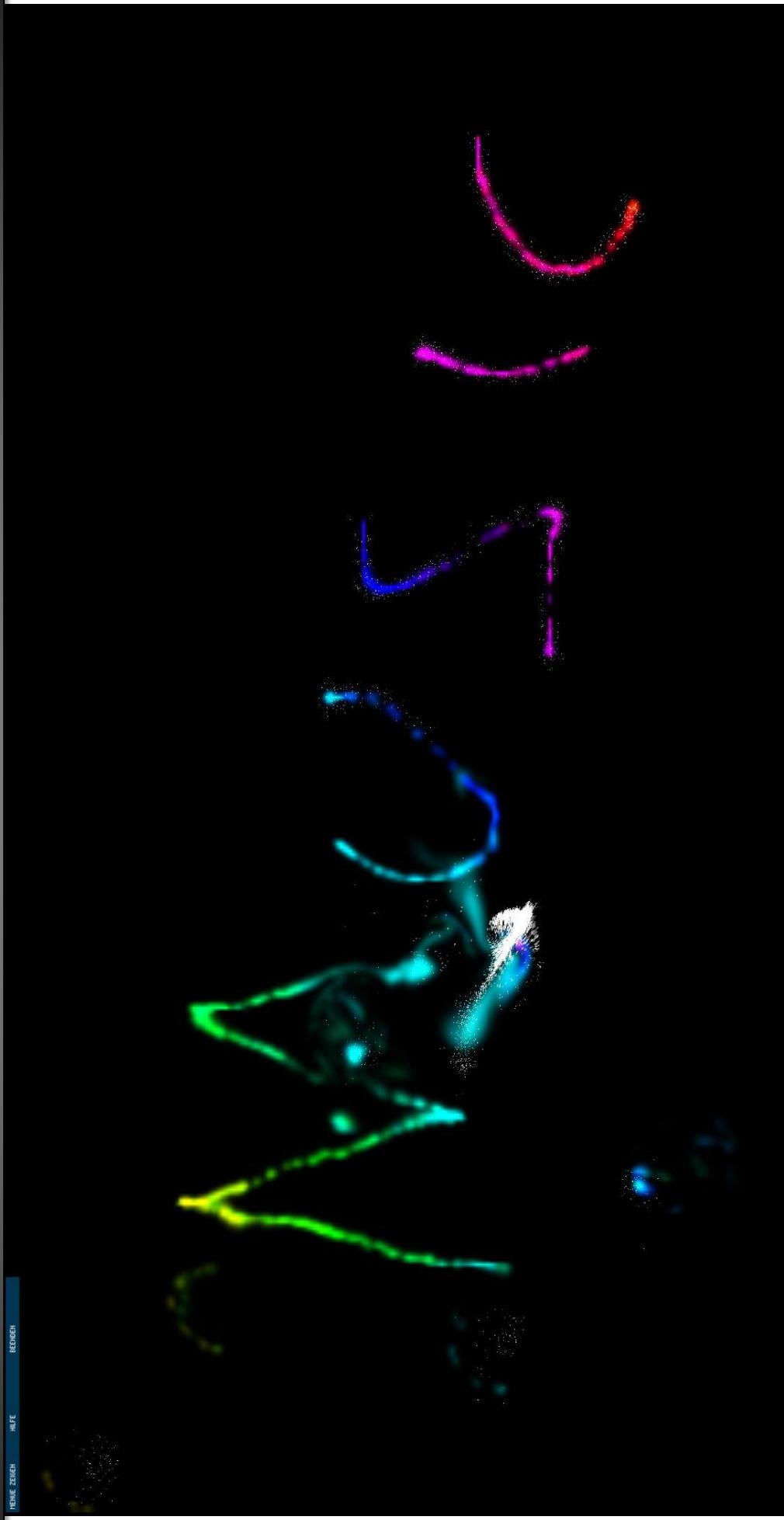
Screenshot von Music Visual mit vom User positionierten Einstellungskomponenten



Screenshot von Music Visual im 3D-Modus



Screenshot von Music Visual mit veränderter Viskosität (0.1 statt 0.0001) im Pause-Modus (Pixel-Effekt)



Screenshot von Music Visual mit veränderter Konstante **FLUID_WIDTH** (420 statt 120); Text mit Maus geschrieben



Flussdiagramme

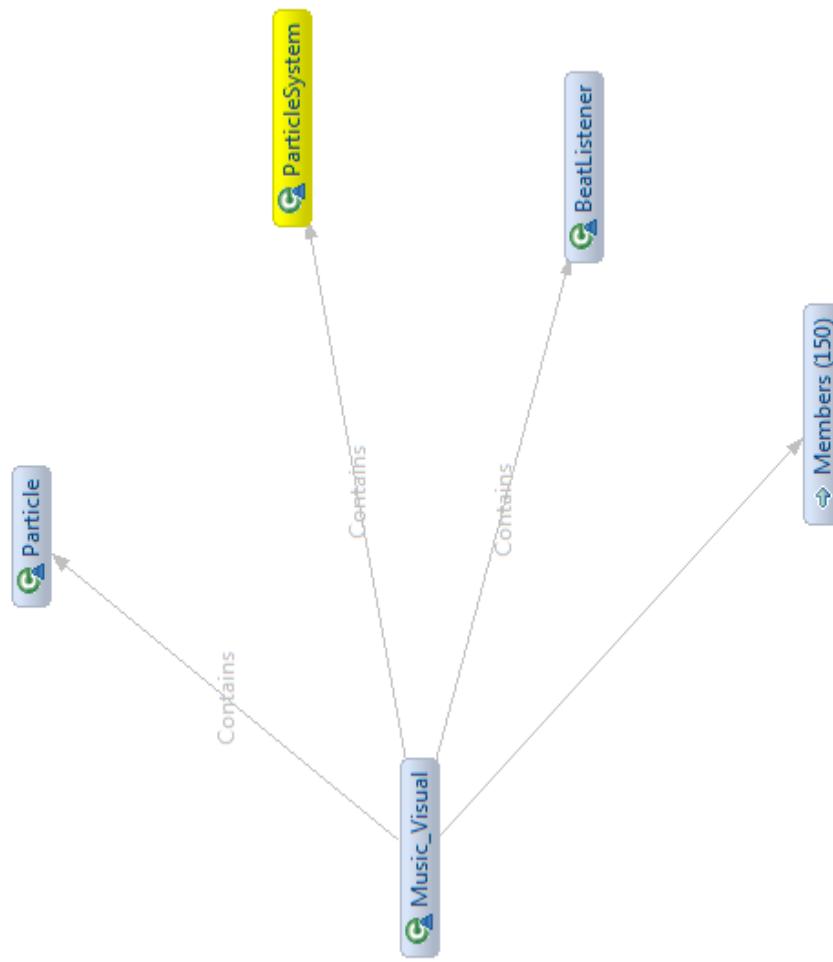
Auf den folgenden Seiten sind einige Flussdiagramme abgebildet, die das gesamte Programm in einzelne Portionen zerlegen, um noch einmal die Aufteilung der einzelnen Codeabschnitte zu veranschaulichen.

Die Flussdiagramme wurden erstellt, indem der Processing Code in Java Code umgewandelt und dann grafisch dargestellt wurde. Da dies mit der Processing IDE nicht möglich ist, wurde der Code mit Hilfe eines Processing Plugins in die Entwicklungsumgebung Eclipse portiert, die den Entwickler in Java und einigen anderen Programmiersprachen unterstützt. Sie kann unter <http://www.eclipse.org/> heruntergeladen werden.

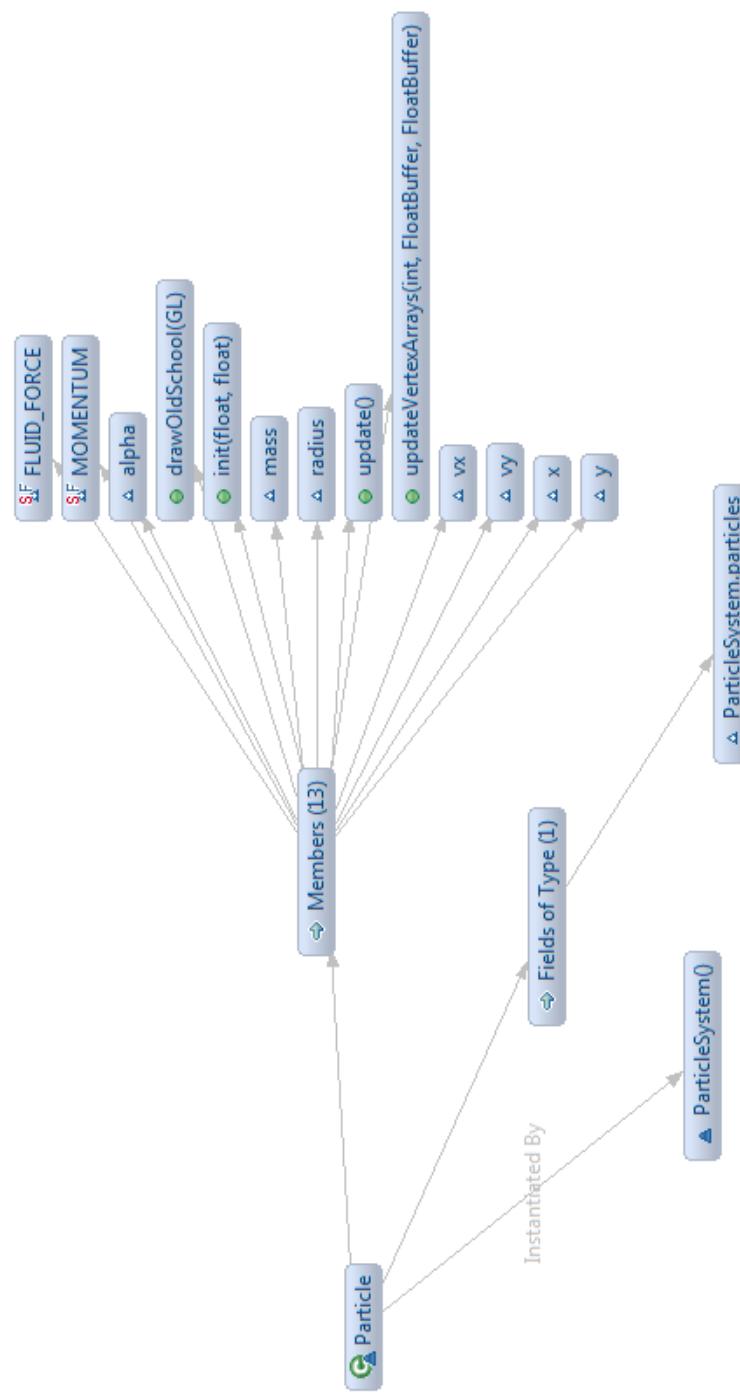
Da Eclipse so leicht erweiterbar ist, wurde ein weiteres Plugin eingesetzt, dass sich *nWire* nennt. Obwohl es ein kostenpflichtiges Programm ist, gibt es eine Testversion, die sich 30 Tage lang uneingeschränkt nutzen lässt. Das Plugin lässt sich über den Update-Manager von Eclipse installieren, der sich im Hauptmenü unter „Help“ -> „Install New Software“ öffnen lässt. In die Adresszeile muss man dann die Updateseite eingeben, die <http://update.nwiresoftware.com/> lautet. Man kann von dieser Homepage auch Eclipse mit dem vorinstallierten Plugin herunterladen.

Sind diese Schritte vollbracht, muss nur mehr die entsprechende Java Datei markiert werden und mit der Kombination [Strg] + [4] gelangt man in die textbasierte Veranschaulichung des Codes. In den grafischen Modus gelangt man durch das Symbol „Open Visualization View“, das sich auf der rechten Seite befindet.

Das Diagramm kann dann modifiziert werden, indem zum Beispiele einzelne Funktionen geöffnet werden, oder ein anders Objekt in den Mittelpunkt gerückt mit „Focus on Selection“. Die Grafik kann auch abgespeichert werden mit der Funktion „Save Image“.

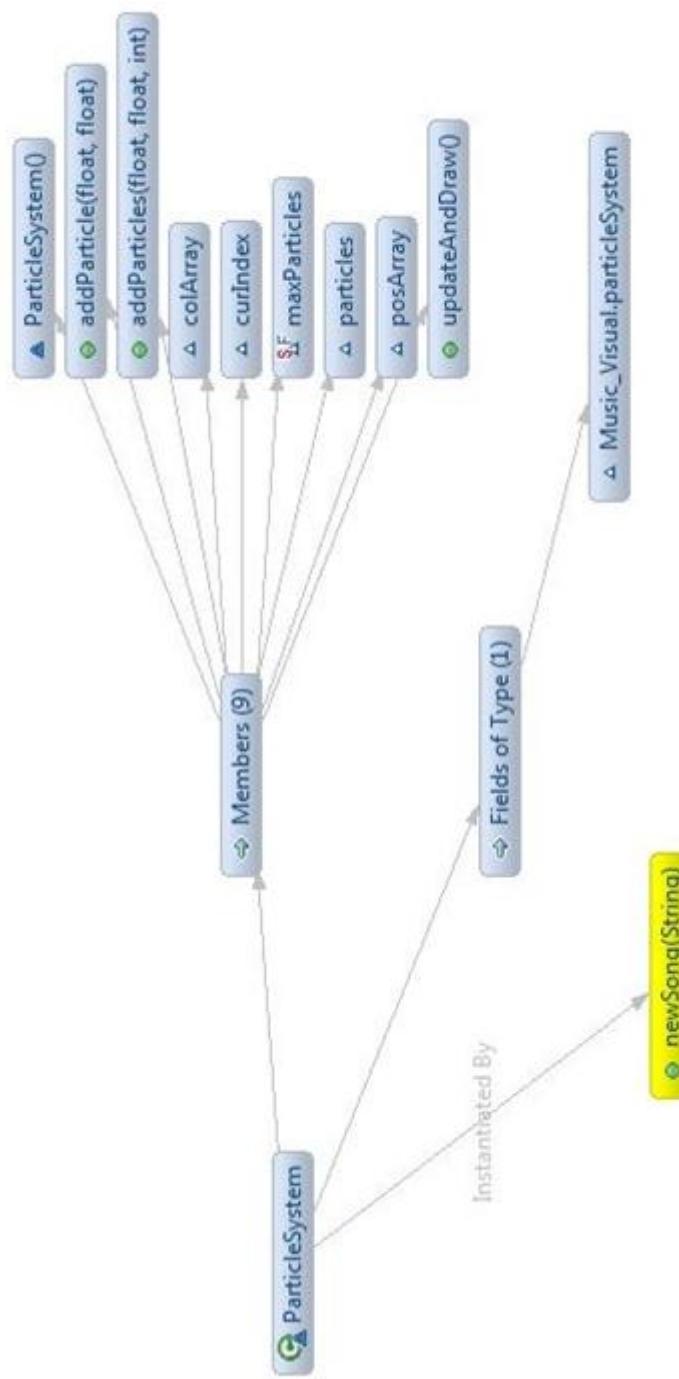


Hier sieht man eine allgemeine Übersicht des gesamten Programmes. Der Grafik kann man entnehmen, dass `Music Visual` aus 150 „Members“ besteht. Mit `Members` sind sowohl Variablen, also auch Methoden gemeint. Zusätzlich kommen in den einzelnen Klassen noch die Konstruktoren dazu. `Music Visual` besteht aus diesen Membern, sowie aus drei Klassen, nämlich `Particle`, `ParticleSystem` und `BeatListener`.

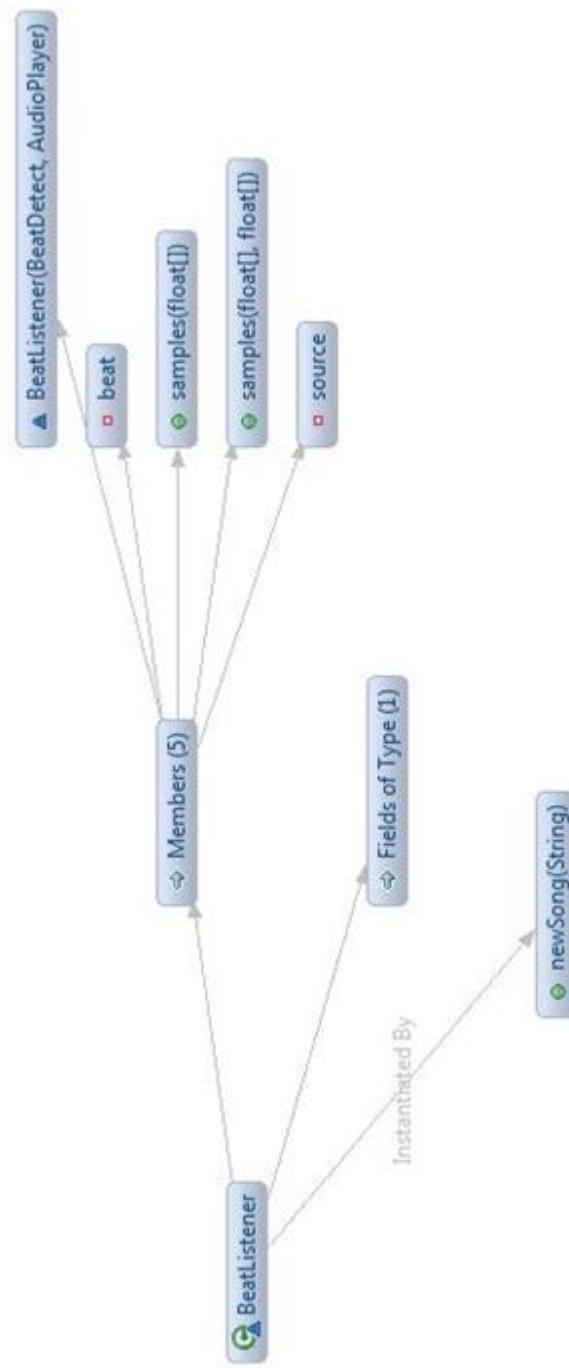


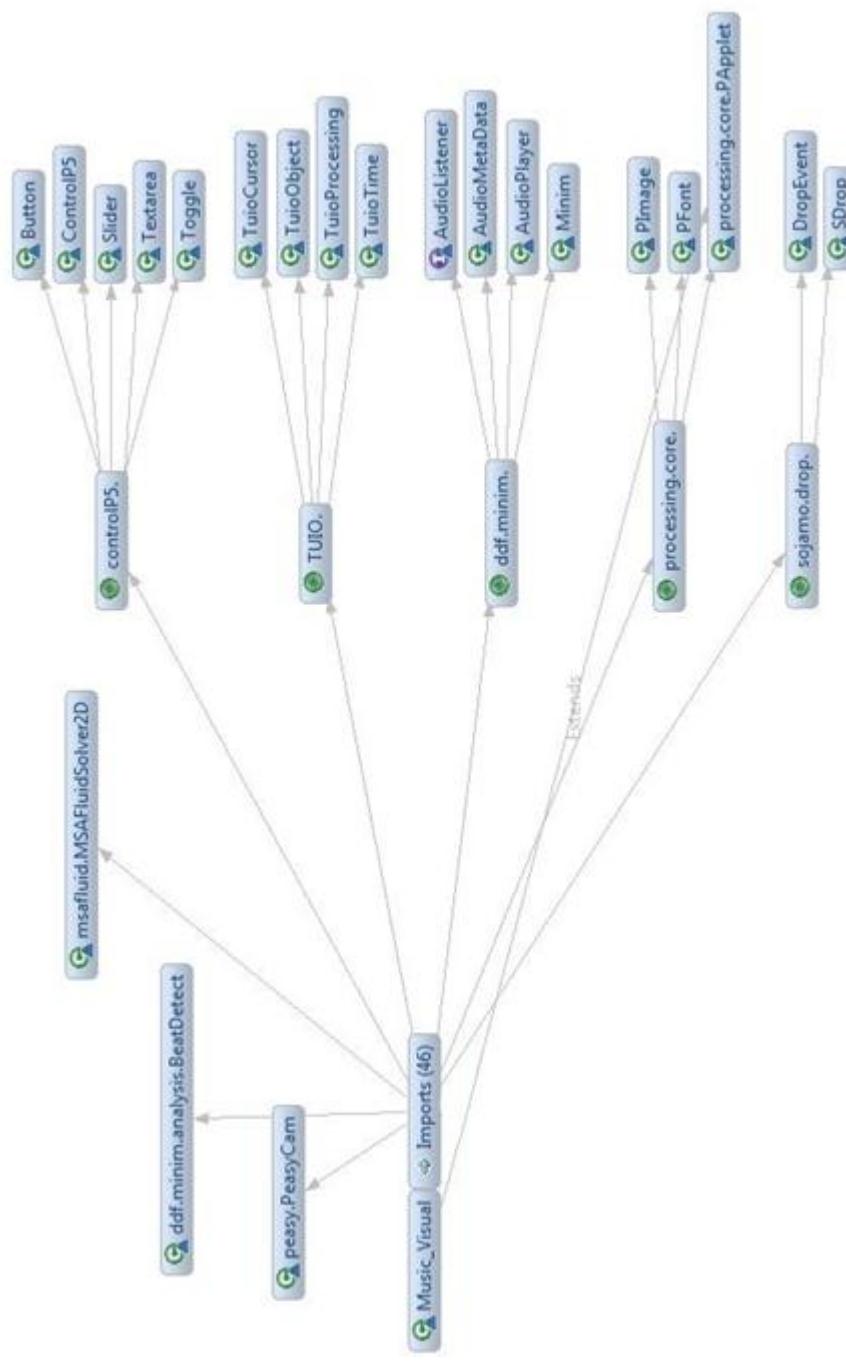
Die Klasse Particle besteht aus dreizehn Members, wobei auch zwei Konstanten vorkommen. „Instantiated By“ bedeutet in diesem Diagramm, dass ein neues Objekt dieser Klasse von der Klasse ParticleSystem erstellt wird. „Fields of Type“ gibt noch genauer Informationen über die Instanzierung an. Particle wird nämlich als Typ des Arrays `particles` in bereits vorher erwähnter Klasse verwendet.

Die Klasse ParticleSystem ähnelt im Aufbau der Particle Klasse. Sie besitzt neun Member wird instanziiert in der Funktion newSong(). Es gibt eine Variable mit diesem Typ die sich *particleSystem* nennt. Die verschiedenen Komponenten sind übrigens mit verschiedenen Symbolen versehen. Ein Dreieck bedeutet Konstruktor, ein grüner Kreis, dass es eine Funktion ist und ein nicht ausgefülltes Dreieck symbolisiert eine Variable. Konstanten werden extra mit den Buchstaben „SF“ hervorgehoben.



BeatListener besteht nur aus 5 Membern, wobei hier nochmal auf die Funktion `samples()` hingedeutet wird, die es in zwei Varianten mit unterschiedlichen Argumenten gibt. Auch ein Konstruktor ist wieder vorhanden. Fields of Type wurde diesmal nicht erweitert, da sie nur auf den Hauptsketch verweist. Die Klasse wird wieder von der Funktion `newSong()` heraus verwendet, wobei hier nicht so deutlich gemacht werden kann, wie viele Aufgaben diese Funktion außer einen neuen Song abzuspielen, hat.





Diese etwas weiter verzweigte Grafik stellt die ganzen Importe da, die aus Processing heraus getätigert wurden, sowie welche Klassen benutzt wurden. Die Zahl 46 neben „Imports“ vermag einen Fehler in der Grafik aufgedeckt zu haben, allerdings stimmt es, dass insgesamt so viele Importe erfolgt sind. Alle nicht dargestellten Importe sind Java Bibliotheken, die beim Sketch zur Anwendung kamen. Man denke dabei zum Beispiel an die Pakete javax.swing zum Zeichnen des Fensters, sowie javax.media.opengl, da die von Processing verwendete OpenGL Bibliothek auf diese zugreift.



Quellen der Bilder

Alle in dieser Arbeit vorkommenden Screenshots wurden eigenständig gemacht und zugeschnitten. Alle weiteren Bilder wurden ungeschnitten in die Arbeit aufgenommen.

- Seite 12 - 25: <http://processing.org/img/learning/books/>
- Seite 45: http://en.wikipedia.org/wiki/File:Pipeline_OpenGL_%28en%29.png

Quellenangaben der Texte

Alle Quellen, die ohne ein Datum angeführt wurden, wurden am 26.5.2011 bzw. zwischen dem 6.8.2011 und 10.8.2011 benutzt. Um verwaiste Links zu vermeiden, wurden teilweise nur die Verweise zu den Startseiten der jeweiligen Quellen angegeben.

- <http://processing.org/>
- http://de.wikipedia.org/wiki/Prix_Ars_Electronica (26.5.2011)
- http://en.wikipedia.org/wiki/Processing_%28programming_language%29 (26.5.2011)
- <http://de.wikipedia.org/wiki/Processing> (26.5.2011)
- http://wiki.processing.org/w/Export_Info_and_Tips (26.5.2011)
- <http://en.wikipedia.org/wiki/Processing.js> (26.5.2011)
- <http://processingjs.org/> (26.5.2011)
- <http://de.wikipedia.org/wiki/Escape-Sequenz> (7.8.2011)
- <http://wiki.delphigl.com/index.php/glBlendFunc> (7.8.2011)
- <http://wiki.delphigl.com/index.php/glBegin> (7.8.2011)
- <http://en.wikipedia.org/wiki/OpenGL> (8.8.2011)
- <http://de.wikipedia.org/wiki/OpenGL> (8.8.2011)
- http://en.wikipedia.org/wiki/Java_OpenGL (8.8.2011)
- <http://www.dgp.toronto.edu/people/stam/reality/Research/pdf/GDC03.pdf> (8.8.2011)
- http://en.wikipedia.org/wiki/Perlin_noise (9.8.2011)
- <http://olli.informatik.uni-oldenburg.de/Graffiti3/graffiti/flow10/page7.html ff.> (20.8.2011)



Funktionsweise von Processing und weitere Funktionen

Processing wurde bereits allgemein behandelt und anhand des Projektes „Music Visual“. Das folgende Kapitel behandelt Processing „hinter den Kulissen“ und soll Einblicke in den Quellcode dieser Programmiersprache zeigen. Die Erklärungen sind recht allgemein gehalten, das heißt, dass der Programmiercode nicht konkret erklärt wird, sondern nur in Worten zusammengefasst ist. Es werden auch einige neue Methoden vorgestellt. Weniger relevante Funktionen wurden unter Umständen ausgelassen. Der Quellcode von Processing ist bei Google Project Hosting gelagert und kann mit einem Subversion-Klient mit dem Befehl

```
svn checkout http://processing.googlecode.com/svn/trunk/ processing-read-only
```

in einem Verzeichnis auf dem eigenen Computer abgespeichert werden. Subversion (SVN) ist eine freie Software zur Versionsverwaltung von Dateien und Verzeichnissen. Um die Software nutzen zu können, muss ein Paket von <http://subversion.apache.org/packages.html> heruntergeladen werden. Subversion kann sowohl über die Konsole als auch über ein Programm mit grafischer Oberfläche gesteuert werden. Um den Code online betrachten zu kön-

The screenshot shows the main page of the Processing project on Google Code. At the top, there's a logo with a blue house-like icon and the word "processing". Below it, a subtitle reads "An open source programming language and environment for images, animation, and interactions." A navigation bar includes links for "Project Home", "Downloads", "Wiki", "Issues", "Source", "Summary", "Updates", and "People". On the left, a sidebar titled "Project Information" contains sections for "Activity" (with a "High" status), "Code license" (GNU Lesser GPL), and "Members" (listing f., r., b., and 12 committers). A "Featured" section highlights "Downloads" with links to "processing-1.5.1-linux.tgz", "processing-1.5.1-macosx.zip", and "processing-1.5.1-windows.zip", with a "Show all" link. Another sidebar titled "Links" lists "External links" including "Processing.org", "(incomplete) Core Javadoc", and "(incomplete) Javadoc". The main content area contains general information about Processing, instructions for contributing, and a note about the current state of development.

nen kann man auf der Startseite des Projektes unter <http://code.google.com/p/processing/> auf „Source“ klicken. Auf dieser Seite befinden sich auch andere Informationen wie die neuesten Updates, die Mitglieder dieses Projekts sowie ein Wiki. Von hier aus kann auch die

Bugtrackingdatenbank erreicht werden, sowie ein Verzeichnis mit allen möglichen Downloadversionen von Processing. Es können auch ältere Versionen heruntergeladen werden, die auch ausdrücklich als veraltet markiert sind. Es kann auch das Javadoc von Processing erreicht werden, eine Dokumentation aller Klassen und Funktionen, die in Processing verwendet werden können. Dieses Kapitel wurde mit Hilfe des Quellcodes von Processing 1.5.1 geschrieben. Die hier erwähnten Funktionsweisen können von der aktuellen Version abweichen.

Core.jar

Dieses Archiv beinhaltet die grundlegenden Klassen, die ein Processing Sketch benötigt, um lauffähig zu sein. Im Quellcode befinden sich diese Dateien unter
processing/core/src/processing/core

Das Archiv beinhaltet folgende Dateien:

PApplet.java	PConstants.java
PFont.java	PGraphics.java
PGraphicsJava2D.java	PIImage.java
PMatrix.java	PMatrix2D.java
PMatrix3D.java	PShape.java
PShape2D.java	PStyle.java
PVector.java	Table.java
XML.java	

PApplet

PApplet ist die Basisklasse für alle Sketche, die processing.core verwenden. Sie erweitert die Klasse Applet aus der Fensterverwaltung AWT. Sie erweitert nicht JApplet (Swing) aus historischen Gründen, da Processing Java 1.1 unterstützt, bei dem es Swing noch nicht gab. In späteren Versionen wie Versionen 1.5.1 wurde die Abwärtskompatibilität auf Java 1.5 beschränkt, da die Entwickler sich auf die Zukunft fokussieren wollten und durch die neuen „timing facilities“ die Stabilität der Animationen verbessert werden. Ein weiterer Grund ist, dass die Entwickler von Swing nicht sehr viel halten, da sie die schwergewichtigen AWT Komponenten vorziehen, die vom Betriebssystem gezeichnet werden. Swing kann trotzdem verwendet werden, allerdings ist zu beachten, dass keine anderen Komponenten über den

Processing Sketch gezeichnet werden können, da die Oberfläche von Processing sich automatisch aktualisiert.

Alle Aufgaben, die mit Animationen zu tun haben, passieren im „Processing Animation Thread“. Die `setup()` und `draw()` Methode, sowie Events wie Mausbewegungen und Tastatureingaben, die durch den „event dispatch thread“ oder „event thread(EDT)“ ausgelöst und eingereiht, um sicher bearbeitet zu werden, werden hier bearbeitet. Genauso wie mit JApplet und Applet verhält es sich mit dem Fenster, bei dem auch die AWT Variante der Swing Komponente JFrame vorgezogen wurde. Wie bereits erwähnt, können Sketche in Java Applikationen eingebunden werden, um zum Beispiel ein Menü mit einem WYSIWYG-Editor mit Java Komponenten zu erzeugen. Das Fenster, in dem der Sketch sich befindet, lässt sich für spezielle Zwecke auch vergrößern und verkleinern, indem `frame.setResizable(true)` und `frame.setSize()` mit den gewünschten Größenangaben aufgerufen werden. Da der Standard Animation Thread mit 60 Frames in der Sekunde läuft, kann die Java Anwendung schwerfällig wirken. Um dies zu verhindern muss entweder mit `frameRate()` eine niedrigere Framerate gesetzt werden oder man deaktiviert `draw()` in `setup()` mit `noLoop()` und aktiviert sie nur wenn es nötig ist mit `loop()`. Alternativ kann auch der Befehl `redraw()` verwendet werden, um während `noLoop()` die Methode `draw()` trotzdem aufzurufen.

PApplet erweitert Applet und implementiert einige Konstanten und Listener für verschiedene Inputeingaben. Zuerst wird die Umgebung, in der der Sketch läuft unter die Lupe genommen. Es wird die Java Version und das Betriebssystem in Variablen gespeichert. Für Mac OS X kann der Quartz Renderer anstatt des Standard Renderers mit einer Variablen bestimmt werden. Gerendert wird das PApplet von einem Objekt der Klasse PGraphics. Es werden am Anfang auch ein paar Konstanten festgelegt wie die Standardgröße eines Applets, die mit 100 festgelegt wurde, sowie die minimale Größe des Fensters, die Aufgrund von Mac OS auf 128 Pixel gesetzt wurde. Die Farbwerte aller Pixel des Sketches werden in ein Array gespeichert, dessen Größe sich mit `width*height` berechnen lässt. Es werden eine Reihe von Variablen definiert wie `width`, `height`, `mouseX`, `mouseY` sowie `pmouseX` und `pmouseY`. Die beiden letzten Variablen benötigen Hilfsvariablen, um beim ersten Frame den Wert der Mausposition anzunehmen, damit sie nicht in den ersten Frames den Wert 0 tragen. Weitere Variablen, die zur Inputabfrage nötig sind, werden deklariert und initialisiert sowie diverse Variablen, wie zum Beispiel `frameCount` und `frameRate`, aber auch Konstanten für die Position des Fensters. Initialisierungen finden hauptsächlich in der Funktion `void init()` statt, in der zum Beispiel die Bildschirmdimension gespeichert wird und einige spezielle Objekte eine

Instanz der Klasse *RegisteredMethods* bekommen. Die Größe des Fensters wird voreingestellt, sowie der Renderer, der standardmäßig Java2D lautet.

Die Methode **void start()** wird vom Browser oder Applet Viewer aufgerufen, die das Applet informiert, dass die Ausführung gestartet werden soll. Dabei wird der Animation Thread gestartet, bevor die Funktion *paint()* aufgerufen wird. Die gegenteilige Funktion, die dem Applet befiehlt zu stoppen, heißt **void stop()**, es wird jedoch nicht garantiert, dass sie immer aufgerufen wird (wenn der Browser zum Beispiel beendet wird, oder beim Wechseln zwischen zwei Webseiten wird sie schon mal ausgelassen). Wenn das Applet die Ressourcen freigeben soll, für die Speicherplatz reserviert wurde, wird die Funktion **void destroy()** aufgerufen.

Applet besitzt auch eine sogenannte innere Klasse, die sich *RegisteredMethods* nennt. Objekte ihrer Klasse registrieren oder melden Mausevents, Keyboardevents, Events, bei denen das Fenster vergrößert wird und noch einige weitere Events ab.

Die nächste Funktion ist eine sehr bekannte Funktion, nämlich **void setup()**. Sie wird nur einmal ganz am Anfang aufgerufen und sollte danach nicht mehr aufgerufen werden. Variablen, die hier deklariert werden, sind außerhalb der Funktion nicht sichtbar.

Nach dieser Funktion wird **void draw()** fortlaufend ausgeführt, bis der Befehl *noLoop()* aufgerufen oder das Programm beendet wird. Den Abstand der Ausführungen kann man mit *frameRate()* bzw. **delay()** bestimmen. Auch wenn die Funktion selbst nicht benötigt wird, muss sie in einem Sketch existieren um Events wie Mauseingaben abzufangen.

Eine weitere Methode ist **void resizeRenderer()**, mit der die Größe des Sketches verändert wird. Es sollten keine Variablen an *void size()* übergeben werden, da es sonst Probleme beim Exportieren des Sketches gibt. Die Funktion kann außerdem nur einmal benutzt werden, Änderungen der Fenstergröße sind mit dieser Methode nicht möglich. Auch muss man beachten, ob die Werte, die in der Variable *screen* gespeichert sind, wirklich einen Fullscreen erzeugen, sollte man die Werte nicht direkt eintragen. Da auf manchen PCs einfach die gesamte Anzahl an Pixeln verwendet wird, ist es möglich, dass ein Bildschirm mit 800 x 600 plötzlich 1600 x 300 unterstützt. In diesem Fall muss man die verschiedenen Renderer und Größen ausprobieren. Eine weitere Gefahr dieser Funktion ist, dass kein Code in *setup()* vor ihr geschrieben werden darf, da er möglicherweise mehrfach ausgeführt wird.

Eine andere Funktion, die in dieser Klasse untergebracht ist, ist **createGraphics()**, mit der sich off-screen Grafikpuffer erstellen lassen. Dabei wird im Hintergrund auf ein Grafikobjekt

gezeichnet, dass später im Sketch angezeigt wird. Dadurch lässt sich das Programm in mehrere Ebenen aufteilen. Der OPENGL Renderer unterstützt die Methode aus technischen Gründen nicht, der DXF Renderer sollte in dieser Kombination auch nicht benutzt werden, sondern mit **beginRaw()** und **endRaw()**. Alle Befehle müssen zwischen den beiden Funktionen geschrieben werden, auch Methoden, die das Zeichnen beeinflussen, wie beispielsweise *smooth()* oder *colorMode()*, da der Renderer wissen muss, wann nicht mehr gezeichnet wird, damit das Objekt aktualisiert werden kann. Im Gegensatz zum Hauptfenster können diese Grafiken transparent sein. Mit dem Befehl **save()** kann die Grafik im PNG oder TGA Format abgespeichert werden. Sollten besonders große Szenen kreiert werden, muss unter Umständen der verfügbare Speicher in den Einstellungen erhöht werden. **createGraphics()** ist die für den Programmierer sichtbare Funktion, intern gibt es zusätzlich die Funktion *PImage makeGraphics()*.

Eine Alternative dazu ist, ein neues *PImage* Objekt zu erstellen, mit dem Bilder angezeigt werden können und auf dessen Pixel man zugreifen kann. Instanziert wird ein neues *PImage* mit *PImage createImage()*. Diese Methode ist der Variante **new PImage()** vorzuziehen. Es muss als Argument die Größe sowie das Format angegeben werden. Mit Format ist das Farbmodell gemeint, dass entweder RGB, ARGB oder ALPHA lauten kann. Der dritte Parameter darf nicht vergessen werden, da sonst ein Fehler auftreten kann.

Um den Bildschirm neu zu zeichnen, stellt das Java Applet eine Methode **void update()** bereit, in der ein Graphics Objekt der Methode **void paint()** übergeben wird. Der erste Frame wird nicht von dieser Funktion, sondern vom Betriebssystem gezeichnet, ansonsten wird das zugeordnete PGraphics Objekt gezeichnet.

Auch der Animation Thread besitzt eine eigene Hauptmethode, die sich **run()** nennt. Wenn zum Beispiel der Sketch pausiert ist, versucht der Thread 100 Millisekunden zu schlafen. Eine weitere Aufgabe ist es den Fokus für das Fenster zu erlangen und die Funktion *handledraw()* aufzurufen, die einen einzelnen Frame rendert. Die Methode wartet auf das Aktualisieren und Zeichnen, bevor der nächste Frame gezeichnet wird, da das Zeichnen manchmal in einem anderen Thread stattfindet.

Die Funktion **handleDisplay()** hat die Aufgabe das Zeichnen abzubrechen, wenn der Renderer noch nicht bereit ist. Die Hauptfunktion besteht darin, dass hier Variablen, die mit der Framerate und der Maus zu tun haben, aktualisiert werden. Dazu wird die Funktion **handleMouseEvent()** aufgerufen. Weiters werden Maus- und Tastaturevents aus der Wartschlanke entfernt und die sogenannte *preMethods* und *postMethods* behandelt, die vor bzw. nach

den anderen Methoden ausgeführt werden. Am Schluss wird die Methode `paint()` aufgerufen.

Manchmal muss `draw()` manuell ausgeführt werden, die Funktion `redraw()` setzt daher ein Zeichen, dass eine Aktualisierung notwendig ist. `draw()` wird zwar nicht sofort aufgerufen, würde aber ohne Hilfe erst nach bis zu 10 Sekunden ausgeführt werden, da der Thread erst aufgeweckt werden muss.

Öfters kam bereits der Begriff Listener vor, die nächste Methode heiße **addListener()**. Listener sind Boten einer Unterfunktion, die die Hauptfunktion bei Änderungen benachrichtigen. Wenn das Hauptprogramm nach neuen Änderungen Ausschau halten würde, würde das viel zu viel Rechenaufwand kosten und man müsste mit Geschwindigkeitsverlust rechnen. In dieser Methode werden Listener für die Maus, Mausbewegungen, die Tastatur, den Fokus und Komponenten registriert.

Wie die Mauseingaben behandelt werden, wurde schon erklärt, doch noch ist unklar, wer die gesamten Mausfunktionen wie `mousePressed()` und `mouseClicked()` bearbeitet. Zuständig dafür ist die Funktion **checkMouseEvent()**, an die ein Objekt der Klasse `MouseEvent` übergeben wird. Analog verhält es sich mit den Tastatureingaben, diesmal heißt die Funktion **checkKeyEvent()**. Um Tasten abzufangen, die dem American Standard Code for Information Interchange (ASCII) entsprechen, verwendet man die Variable **key**, ansonsten **keyCode**. Die speziellen Tasten Backspace, Tab, Enter, Escape und Delete gehören zur ASCII Spezifikation und können daher mit `key` abgefragt werden. Wenn der Sketch auf mehreren Plattformen laufen soll, muss man darauf achten, dass beim Macintosh zusätzlich zur Enter-Taste auch die Return-Taste abgefragt werden muss. Wenn eine Taste gehalten wird, wird `keyPressed()` und auch `keyReleased()` mehrere Male aufgerufen. Die Anzahl der Aufrufe ist vom Betriebssystem und von den Einstellungen des Computers abhängig.

Die nächste Funktion bezieht sich nur auf Applets und wurde daher im praktischen Projekt nicht erwähnt. Applets werden mit Hilfe des applet-Tags in eine Webseite eingebunden. Es können auch Parameter mit dem gleichnamigen Tags mitgegeben werden. Diese können mit **String param()** einzelnen ausgelesen werden. Es muss dazu das *name-Attribut* des Tags als Parameter übergeben werden. Es kann auch der Text der Statuszeile im Browser mit **status()** verändert werden, sowie eine neue Internetseite mit **link()** geöffnet werden. Das erste Argument ist eine vollständige Adresse, das zweite Argument muss variabel, je nach Browser und Betriebssystem gesetzt werden, die Grundfunktion ist jedoch das Setzen des Titels der neuen Seite.

Noch eine neue Funktion ist **open()**, mit der sich Programme und Dateien vom Betriebssystem öffnen lassen. Sie funktioniert bei Windows über die Kommandozeile, bei Mac OS X wird der Befehl „open“ verwendet und bei Linux versucht der Befehl zuerst „gnome-open“, dann „kde-open“, bei einem Misserfolg wird der Befehl ohne Änderung an die Konsole weitergegeben.

Im Gegensatz dazu gibt es in PApplet einige Funktionen, die sich um das Auflösen von Ressourcen und um das Beenden kümmern. Um ein Applet mit einer Fehlermeldung zu beenden, gibt es die Funktion **die()**. Die Standardfunktion ohne Fehlermeldung lautet **exit()** und kann im Sketch aufgerufen werden.

Um wieder einen grafischen Befehl zu betrachten, ist **cursor()** eine gute Wahl. Er ist bereits bekannt, wurde aber nicht ausführlich erklärt. Mit ihm lässt sich nicht nur der Cursor anzeigen, sondern auch verändern. Zur Verfügung stehen die Standardcursor, die man mit den Konstanten ARROW, CROSS, HAND, MOVE, TEXT, WAIT aufrufen kann. Es lässt sich auch ein PImage Objekt übergeben, das Symbol sollte entweder 32 x 32 oder 16 x 16 groß sein. Als weitere Argumente muss man den horizontalen und vertikalen Hotspot angeben, der Punkt, an dem der Mauszeiger nicht nur ein statisches Bild zeigt, sondern auch interaktiv handelt. Es gibt allerdings im Web die Einschränkung, dass benutzerdefinierte Cursor nicht erlaubt sind und auch nicht alle Standardcursor unterstützt werden. Bei Linux gibt es den Nachteil, dass der Cursor nur zweifarbig sein kann.

Eine weitere Methode, die eine sehr wichtige Hilfe beim Fehlersuchen (debuggen) ist, ist **println()**. Sie gibt auf der Konsole einen Text aus. Sie basiert auf dem gleichnamigen Befehl der Java Klasse *System.out*. Um einen Fehler in roter Schrift auszugeben, kann man *System.error.println()* verwenden. Die letzten beiden Buchstaben in *println* stehen für „line“ und können auch weggelassen werden, um keinen Zeilenumbruch hervorzurufen. Grundsätzlich sollten nur primitive Datentypen als Argument angegeben werden, Processing versucht aber auch Objekte als String darzustellen.

In jedem Sketch kommen Rechenoperationen vor, manchmal nur simple, andere Male auch komplexere Operationen. Daher sind nun die mathematischen Befehle am Zug. Die meisten von ihnen funktionieren so einfach, dass man sie in einer Tabelle darstellen kann:

Funktion	Berechnung mit Java Befehlen
float abs(float n)	<code>return (n < 0) ? -n : n</code>
float sq(float a)	<code>return a*a</code>
float sqrt(float a)	<code>return (float)Math.sqrt(a)</code>

float log(float a)	return (float)Math.log(a)
float exp(float a)	return (float)Math.exp(a)
float pow(float a, float b)	return (float)Math.pow(a, b)
int/float/double max(a, b)	return (a > b) ? a : b
int/float/double max(a, b, c)	return (a > b) ? ((a > c) ? a : c) : ((b > c) ? b : c)
int/float/double min(a, b)	return (a < b) ? a : b
int/float/double min(a, b, c)	return (a < b) ? ((a < c) ? a : c) : ((b < c) ? b : c)
int/float constrain(amt, low, high)	return (amt < low) ? low : ((amt > high) ? high : amt)
float sin(float angle)	return (float)Math.cos(angle)
float cos(float angle)	return (float)Math.cos(angle)
float tan(float angle)	return (float)Math.tan(angle)
float asin(float value)	return (float)Math.asin(value)
float acos(float value)	return (float)Math.acos(value)
float atan(float value)	return (float)Math.atan(value)
float atan2(float a, float b)	return (float)Math.atan2(a, b)
float degrees(float radians)	return radians * RAD_TO_DEG
radians(float degrees)	return degrees * DEG_TO_RAD

int floor(float what)	return (int) Math.floor(what)
int round(float what)	return (int) Math.round(what)
float mag(float a, float b)	return (float)Math.sqrt(a*a + b*b)
float dist(float x1, float y1, float x2, float y2)	return sqrt(sq(x2-x1) + sq(y2-y1))
float dist(float x1, float y1, float z1, float x2, float y2, float z2)	return sqrt(sq(x2-x1) + sq(y2-y1) + sq(z2-z1))
float lerp(float start, float stop, float amt)	return start + (stop-start) * amt
float norm(float value, float start, float stop)	return (value - start) / (stop - start)
float/double map(value, istart, istop, ostart, ostop)	return ostart + (ostop - ostart) * ((value - istart) / (istop - istart))

Nicht nur Mathematik findet sich in vielen Sketchen, auch der Zufall spielt eine große Rolle. Die Funktion **random()** generiert Zufallszahlen. Intern wird die Klasse *Random* verwendet.

Wenn als Argument 0 angegeben wird, wird aufgrund eines Fehlers zur Sicherheit 0 zurückgegeben. Ansonsten wird die Funktion **nextFloat()** aufgerufen, die eine Zufallszahl zwischen 0 und 1 zurückgibt. Dann wird diese mit dem gegebenen Parameter multipliziert und returned. Soll eine pseudo-zufällige Reihenfolge an Zahlen erfolgen, kann die Funktion **void randomSeed()** mit einer gewählten Zahl aufgerufen werden.

Perlin Noise ist eine Alternative um „Zufallszahlen“ zu erzeugen, die nur leicht sich von der vorherigen Zahl unterscheiden. Diese Funktion wurde bereits in einem kleinen Kapitel behandelt und deshalb wird nur mehr die Methode **noiseDetail()** erwähnt, mit der sich die Anzahl der Oktaven ändern lässt. Es gibt auch eine Funktion, die *randomSeed()* entspricht und zwar **noiseSeed()**.

Betrachtet man als Nächstes die Bildfunktionen, gibt es die Methode **loadImage()**, die ein PImage zurückgibt, bei einem Fehler ist das Ergebnis Null und eine Fehlermeldung wird auf der Konsole ausgegeben. Die Meldung unterbricht das Programm nicht, wenn der Wert auf null überprüft wird, ansonsten gibt es eine NullPointerException.

Es gibt zwei Arten von Fehlern: Errors, das sind schwerwiegende Fehler im Programm sowie Exceptions, deren Unterkategorie RuntimeException ist. Hierbei handelt es sich um Laufzeitfehler, die während des Programmverlaufs auftreten. Die Kategorie vereint alle Fehler, die auf Grund eines Programmierfehlers auftreten können. Dazu gehören:

- fehlerhafte Typkonvertierung
- Zugriff über die Grenzen eines Arrays hinaus
- Zugriff auf einen leeren Zeiger (diese „zeigen“ auf eine Adresse im Speicher)

Alle nicht von RuntimeException abgeleiteten Klassen repräsentieren Fehler, die sich durch ungünstige Umstände und nicht durch fehlerhaften Code ergeben:

- Fehler beim Lesen einer Datei
- Fehlfunktion externer Dienste
- Zu wenige Systemressourcen

loadImage() kann jedoch noch eine andere Fehlermeldung ausgeben, wenn ein Applet versucht, ein Bild von einem anderen Server zu laden. Um diese Sicherheitsmaßnahme zu umgehen, muss ein Applet mit bestimmten Programmen signiert werden. Processing bietet unter <http://processing.org/hacks/doku.php?id=hacks:signapplet> eine Anleitung, wie dies funktioniert. Sollte die Dateiendung nicht ersichtlich sein, kann sie als zweites Argument an

gegeben werden. Wenn das Bild einen Parameter besitzt, der weggeschnitten werden muss, wie zum Beispiel filename.jpg?blah=blah=something=that, kann ein weiteres optionales Argument mit diesem String drangehängt werden. Je nachdem, welches Dateiformat benutzt wird, werden unterschiedliche Varianten genutzt, um das Bild zu laden. Der erste Schritt besteht jedoch immer darin, die Bytes mit *loadBytes()* in ein Array zu laden. Für manche Endungen wird die Bibliothek *javax.imageio* verwendet, bei jpeg, gif und png sollte man stattdessen *createImage()* verwenden. Diese Funktion kann man in Java mit *Toolkit.getDefaultToolkit().createImage()* aufrufen. Das Toolkit wurde übrigens schon vorher häufig verwendet, um zum Beispiel die Bildschirmauflösung auszulesen. Sollte das Bildformat unklar sein, wird versucht mit der Java Klasse *ImageIO* das Format zu erkennen und bei Übereinstimmung mit einem gültigen Format wird das Bild mit einer Funktion geladen und die Parameter angehängt.

Sollte ein Bild sehr groß sein, kann es mit der Funktion **requestImage()** in einem anderen Thread geladen werden, damit der Sketch nicht in *setup()* einfriert. Die Höhe und Breite der Grafik während des Ladens beträgt 0, bei Fehlern werden sie auf -1 gesetzt. Asynchron geladene Bilder verbessern die Performance erheblich, speziell wenn von einem Server im Internet geladen wird. Die Parameter sind identisch mit *loadImage()*. Aus technischen Gründen gibt es nur 4 Threads, da der Webbrowser nur 4 Verbindungen erlaubt. Daher können nicht mehr als 4 Bilder gleichzeitig geladen werden.

Des Öfters war die Rede von *Threads*. Threads sind Programmteile, die nebeneinander ausgeführt werden um Code „gleichzeitig“ verarbeiten können. Tatsächlich springt das Programm nur schnell zwischen den einzelnen Threads hin und her, damit es den Anschein hat, dass der Computer „Multitasking“ kann. Mit mehreren Prozessorenkernen ist diese Gleichzeitigkeit allerdings wirklich erreichbar.

Ein ganz spezielles Format haben Vektorgrafiken, deren Endung .svg ist. Diese Formen sollten in *setup()* mit der Funktion **loadShape()** aus dem Ordner *data* geladen werden. Ansonsten sind die Rückgabewerte und Hinweise der Funktion *loadImage()* zu entnehmen. Analog zur Imagefunktion gibt es **createShape()**, mit der sich eine neue Vektorgrafik mit einer vorbestimmten Größe erzeugen lässt. Grafiken, mit der Endung .svgz werden zuvor noch entpackt und im XML-Format gelesen.

Damit ist das Thema Grafiken abgeschlossen und das Thema „FONT I/O“ kann untersucht werden. Fonts oder Schriftarten können aus demselben Ordner geladen werden, nachdem sie mit dem Menüeintrag „Create Font...“ im Werkzeugmenü erstellt wurden. Die einzelnen

Buchstaben werden bei Processing nicht als Vektorgrafik angezeigt, sondern als normales Bild. Mit dem Befehl **hint(ENABLE_NATIVE_FONTS)**, kann die entsprechende Schriftart auf dem Computer des Users gesucht und angezeigt werden. Dafür verwendet man den Standard Java2D Renderer oder die auf ihm basierende Bibliothek PDF.

Es kann auch die Standardschriftart „Lucida sans“ mit **createDefaultFont()** in der übergebenen Größe erzeugt werden. Diese Variante ruft die allgemeine Variante **createFont()** auf, die einen Schriftartnamen, die Größe und die Glattheit als Boolean akzeptiert. Dabei wird eine auf dem Computer installierte Schriftart in das Format .vlw umgewandelt, das von Processing benötigt wird. Mit **PFont.list()** lassen sich die erkannten Schriftarten eruieren. Aufgrund von Limitierungen der Sprache Java können nicht alle Fonts auf allen Betriebssystemen benutzt werden. Wenn der Sketch mit anderen Leuten geteilt oder im Web veröffentlicht wird, sollte eine .ttf oder .otf Version der Schrift im *data* Ordner beiliegen, da andere die verwendete Schriftart möglicherweise nicht installiert haben. Es sollten auch nur Fonts eingebunden werden, die legal erwerbar sind. **createFont()** geht gleich wie das Tool zum Erstellen von Schriftarten vor und erzeugt eine Kopie in Form des Formates Bitmap an. Wenn der Standard Renderer genutzt wird, wird zuerst versucht die native Schriftart anzuzeigen, die sich durch eine gute Performance und Qualität auszeichnet, mit der P2D, P3D, und OPENGL Einstellung wird immer die Bitmap Version benutzt. Diese hat den Vorteil, dass sie die Erscheinung und Geschwindigkeit erheblich steigern kann, wenn die Schriftart beim Exportieren nicht vorhanden ist, kann das in einem schlechten Resultat enden. Um nur bestimmte Zeichen zu generieren, kann der Funktion **createFont()** als vierter Argument ein Array der Zeichen mitgegeben werden.

Es können mit Processing nicht nur Bilder gezeigt werden, sondern auch Dateien und Ordner selektiert werden. Um einen plattformabhängigen Dateidialog anzuzeigen, wird **selectInput()** verwendet, die den ganzen Pfad zur ausgewählten Datei bzw. bei einem Fehler Null zurückgibt. Dieser Dialog ist zum Öffnen einer Datei gedacht, um sie zu Speichern wird **selectOutput()** benutzt. Wenn eine vorhandene Datei selektiert ist, wird diese ersetzt. Als Alternative kann auch ein neuer Ordner und eine neue Datei zum Beschreiben erstellt werden. Um nur einen Ordner auszuwählen, kommt nur **selectFolder()** in Frage. Im Gegensatz zu den letzten beiden Methoden, die einen eigenen Dialog verwenden, nutzt **selectFolder()** den eigenen Dialog von Java, indem ein neues JFileChooser Objekt instanziert wird.

Um die ausgewählten Objekte zu lesen und beschreiben, muss der Abschnitt „Readers und Writers“ genauer betrachtet werden. Um die einzelnen Zeilen einer Datei als String einzulesen, verwendet man **createReader()**. Man übergibt den Dateinamen und erhält im Gegenzug ein Objekt der Klasse BufferedReader. Wie man dies in einem Sketch verwendet, gibt dieses Beispiel von der Referenzseite Aufschluss:

```
01
02
03 BufferedReader reader;
04 String line;
05
06 void setup() {
07   // Open the file from the createWriter() example
08   reader = createReader("positions.txt");
09 }
10
11 void draw() {
12   try {
13     line = reader.readLine();
14   } catch (IOException e) {
15     e.printStackTrace();
16     line = null;
17   }
18   if (line == null) {
19     // Stop reading because of an error or file is empty
20     noLoop();
21   } else {
22     String[] pieces = split(line, TAB);
23     int x = int(pieces[0]);
24     int y = int(pieces[1]);
25     point(x, y);
26   }
27 }
```

Um den Reader in `setup()` zu verwenden, könnte man den Codeabschnitt, der in die Try-catch-Anweisung gepackt ist, zusätzlich in eine While-Schleife geben und die Abfrage ungefähr so gestalten: `while((line=reader.readLine()) != null)` ... Wichtig ist dabei, dass die Zuweisung nur aus einem Gleichheitszeichen besteht und mit runde Klammern umschlossen werden muss, um die Zuweisung als Ausdruck verwenden zu können. Java Programmierer, die Processing benutzen, können auch ein Objekt vom Typ File oder InputStream übergeben.

Um eine Datei zu beschreiben, wird **createWriter()** verwendet. Analog kann statt einem String wieder ein File bzw. OutputStream übergeben werden. Betrachtet man dazu das Referenzbeispiel, fällt einem auf, dass der Writer einfacher zu bedienen ist.

```
01  PrintWriter output;
02
03  void setup() {
04      // Create a new file in the sketch directory
05      output = createWriter("positions.txt");
06  }
07
08  void draw() {
09      point(mouseX, mouseY);
10      output.println(mouseX + "t" + mouseY); // Write the coordinate to the file
11  }
12
13  void keyPressed() {
14      output.flush(); // Writes the remaining data to the file
15      output.close(); // Finishes the file
16      exit(); // Stops the program
17 }
```

Bei den Inputmethoden gibt es eine Variante für fortgeschrittene Programmierer, um einen Java InputStream zu öffnen. Möglich ist dies durch **openStream()**. Der angehängte Dateiname kann entweder eine URL sein, zum Beispiel `openStream("http://processing.org")`, eine Datei aus dem *data* Ordner oder der vollständige Pfad zu einer lokalen Datei, wenn der Sketch als Programm ausgeführt wird. Wenn der Sketch nicht online steht, wird auch auf Groß- und Kleinschreibung geachtet, damit die Datei beim Export gefunden wird. Im Web spielt dies eine Rolle, in der Processing Entwicklungsumgebung unter Windows oder Mac OS wird das ignoriert. Sollte das gewünschte Objekt nicht existieren, wird Null zurückgegeben, wenn die Datei mit .gz endet, wird der Stream mit gzip entpackt. Sollte die automatische Dekomprimierung nicht erfolgen, muss **createInputRaw()** verwendet werden. Dateien im Internet werden übrigens nicht nur im *data* Ordner gesucht, sondern im ganzen Verzeichnis.

Sehr häufig wurde bereits **loadBytes()** verwendet. Dabei werden die Bytes in ein Bytearray gespeichert. Die Datei muss sich im *data* Ordner bzw. auf dem gleichen Webserver befinden. Verwendet werden dazu die Klassen `BufferedInputStream` und `ByteArrayOutputStream`.

Diese Form von Daten ist zur Datenvisualisierung unpraktisch. Es lassen sich nämlich auch mit **loadStrings()** die einzelnen Zeilen als String in ein Array speichern. Die Methode funktioniert im Hintergrund ähnlich wie der Einsatz von `createReader()`. Eine entscheidende Funktion ist `System.arraycopy()` mit der die Strings in das Array kopiert werden.

Passend zu `createInput()` gibt es auch **createOutput()**, die einen OutputStream für einen gegebenen Pfad oder eine Datei erstellt. Die Datei entsteht im Sketch Ordner oder im gleichen Ordner wie die exportierte Applikation. Wenn der Pfad noch nicht existiert, wird der entsprechende Ordner erstellt. Fehler werden auf der Konsole ausgegeben, der Rückgabewert lautet dann Null. Es sind einige erweiterte Schritte nötig um die Funktion nutzen zu können:

- Das Erstellen eines FileOutputStream Objekt
- Das Feststellen des exakten Pfades zur Datei
- Das Bearbeiten von Exceptions

Um einen Stream in eine Datei zu speichern, gibt es **saveStream()**. Grundlegend wird nur `saveBytes(blah, loadBytes())` aufgerufen, jedoch auf eine effizientere Art. Die hilfreiche Klasse hierzu ist `FileOutputStream`.

Gleich wie für Streams gibt es eine Funktion für Bytearrays, nämlich **saveByte()**. Als erstes Argument ist der Dateiname nötig und als zweites das Bufferarray, in dem die Bytes gespeichert sind. Die Funktion nutzt diesmal `OutputStream`.

Aller guten Dinge sind drei und daher gibt es auch einen Weg, String Arrays zu speichern, die Lösung heißt **saveStrings()**. Vom Prinzip her ähnelt sie sehr `saveByte()`.

Nun ein kurzer Überblick über die Funktionen die mit Ordner zu tun haben. Eine versteckte Funktion ist **sketchPath()**, mit der sich der Pfad des Sketches vor den angegebenen Pfad stellen lässt. Im Webbrower wird dieser aus Sicherheitsgründen auf null gesetzt. Wenn die Funktion `init()` nie aufgerufen wurde, gibt sie einen Fehler zurück.

sketchFile() funktioniert gleich, mit dem Unterschied, dass ein Zwischenordner entsteht, damit Objekte ordnungsgemäß gespeichert werden können. Bei beiden Funktionen wird nicht der `data` Ordner verwendet. Das hat den Grund, dass sich die Dateien beim Applet in einem Jar-Archiv befinden und auf diesen ständig zugegriffen wird.

Noch zwei Funktionen, die nicht online dokumentiert sind, sind **dataFile()** und **createPath()**. Die erste Methode gibt den vollständigen Pfad zu einer Datei im Medienordner zurück, die Zweite erstellt einen neuen Unterordner am übergebenen Pfad.

Bevor die Arrayfunktionen durchgenommen werden, sollten noch die Sortierfunktionen beachtet werden. Sie heißen alle **sort()** und akzeptieren alle möglichen Varianten an Parametern und geben verschiedene Typen zurück. Intern werden immer die Java Funktionen `System.arraycopy()` und `Array.sort()` ausgenutzt.

Die Arraymethoden wären zu langsam, um hier durchgenommen zu werden. Grundsätzlich werden immer neue Arrays erstellt oder kleinere Teile von Arrays zurückgegeben. Hilfreich dabei ist wieder `System.arraycopy()`.

Funktionsname	Aufgabe
arrayCopy()	Kopiert ein Array in neues Array
expand()	Erweitert die Größe eines Arrays
append()	Fügt ein neues Element hinzu
shorten()	Entfernt ein Element und gibt es zurück
splice()	Fügt ein Array einem anderen Array hinzu
subset()	Extrahiert einen Teil des Arrays und gibt in zurück
concat()	Fügt zwei Arrays zusammen
reverse()	Dreht die Reihfolge im Array um

Bei den String Funktionen wird in dieser Arbeit auch nicht weiter ins Detail eingegangen. Strings sind eigentlich unveränderlich, dadurch kann man nur entweder einen neuen String instanziieren oder einen StringBuffer verwenden.

Funktionsname	Aufgabe
trim()	Entfernt alle Arten von Leerzeichen
join()	Fügt ein String Array zusammen, getrennt durch ein Zeichen
splitTokens()	Erstellt ein String Array aufgrund einiger Trennzeichen
split()	Erstellt ein String Array aufgrund eines Trennzeichens
match()	Gibt ein String Array mit den auf das Suchmuster passenden Strings zurück.
matchAll()	Ähnlich wie match()

Nachdem viele dokumentierte Funktionen gezeigt wurden, sind jetzt einige interne Funktionen an der Reihe, die eine große Rolle bei der Konvertierung von Datentypen haben. Viele von ihnen wurden in den ersten Versionen von Processing programmiert und später wieder entfernt.

Casts für Boolean:

Signatur	Veraltet	Signatur	Veraltet
parseBoolean(char what)	Ja	parseBoolean(char what[])	ja
parseBoolean(int what)	Nein	parseBoolean(byte what[])	ja
parseBoolean(float what)	Ja	parseBoolean(int what[])	Nein
parseBoolean(String what)	nein	parseBoolean(float what[])	Ja

Casts für Byte:

Signatur	Veraltet	Signatur	Veraltet
parseByte(boolean what[])	Nein	parseByte(String what[])	Ja
parseByte(char what[])	Nein		
parseByte(int what[])	Nein		
parseByte(float what[])	Nein		

Casts für Char:

Signatur	Veraltet	Signatur	Veraltet
parseChar(boolean what)	Ja	parseChar(String what[])	Ja
parseChar(byte what)	Nein	parseChar(boolean what[])	Ja
parseChar(int what)	Nein	parseChar(byte what[])	Nein
parseChar(float what)	Ja	parseChar(int what[])	Nein
parseChar(String what)	Ja	parseChar(float what[])	Ja

Casts für Int:

Signatur	Veraltet	Signatur	Veraltet
parseInt(boolean what)	Nein	parseInt(String w, int otherwise)	Nein
parseInt(byte what)	Nein	parseInt(boolean what[])	Nein
parseInt(char what)	Nein	parseInt(byte what[])	Nein
parseInt(float what)	Nein	parseInt(char what[])	Nein
parseInt(String what)	Nein	parseInt(float what[])	Nein
parseInt(String what[])	Nein	parseInt(String w ,int missing)	Nein

Casts für Float:

Signatur	Veraltet	Signatur	Veraltet
parseFloat(boolean what[])	Ja	parseFloat(int what[])	Nein
parseFloat(char what[])	Ja	parseFloat(String what[])	Nein
parseFloat(byte what[])	Nein	parseFloat(String w[], int miss)	Nein

Casts für String:

Signatur	Veraltet	Signatur	Veraltet
str(boolean what)	Nein	str(boolean what[])	Nein
str(byte what)	Nein	str(byte what[])	Nein
str(char what)	Nein	str(char what[])	Nein
str(int what)	Nein	str(int what[])	Nein
str(float what)	Nein	str(float what[])	Nein

Für Integer und Float gibt es zusätzliche Funktionen, mit denen sie in Strings formatiert werden können. Eine davon heißt **nf()**. Zusätzlich zur Zahl kann man entweder die Anzahl der Zahlen vor dem nächsten Punkt angeben oder wie viele Zahlen auf der rechten und wie viele auf der linken Seite des Punktes stehen. Dabei muss man den zweiten und dritten Parameter in Form einer Ganzzahl angeben. Mit **nfc()** können auch Arrays mit Zahlen formatiert werden. Dabei muss als zweites Argument die linke Seite des Punktes nicht angegeben werden. Um ein Vorzeichen vor den Zahlen zu erlangen, muss die Funktion **nfp()** verwendet werden. Die letzte Variante **nfs()** setzt statt dem Pluszeichen ein Leerzeichen, damit positive mit den negativen Zahlen auf gleicher Höhe stehen.

Um nicht zu sehr in den Binärkode einzudringen, wird das Kapitel Farben und Umwandlungen von Farbcodes nur oberflächlich angeschnitten. Farben werden in Processing im *color* Objekten gespeichert. Eigentlich bestehen sie aus einem Integer. Wenn man das ARGB-Farbmodell nimmt, wird jede Komponente in einem gemeinsamen Integer gespeichert. Da ein Integer 32 Bit oder 4 Byte an Daten speichern kann, hat jede Komponente 8 Bit an Speicherplatz. Um die einzelnen Farben zu extrahieren, muss man eine logischen Verschiebung mit den Operator `<<` (nach links) und `>>` (nach rechts) machen. Unter einer logischen Verschiebung versteht man in der Informatik einen bitweisen Operator, der alle Bits des Operanden um eine bestimmte Anzahl an Stellen verschiebt:

```

01  println(10 >> 1); // == 1010 -> 0101
02  /*
03   16 | 8 | 4 | 2 | 1
04   1   0   1   0   # 8+2=10
05   -> 1   0   1   # 4+1=5
06  */
07
08  println(100 >> 2); // 1100100 -> 0011001
09  /*
10   64 | 32 | 16 | 8 | 4 | 2 | 1
11   1   1   0   0   1   0   0 # 64+32+4=100
12   -> -> 1   1   0   0   1 # 16+8+1 =25
13  */

```

Um das Ergebnis in eine neue Variable speichern zu können, muss mit Hilfe der bitweisen Oder-Verknüpfung verglichen werden, welche Bits gesetzt sind. Eine Variable, in der alle Bits gesetzt sind, bekommt man durch 0xFF. Folglich bekommt man die RGB-Farben auf diese Weise:

```
01 int red = (rgb >> 16) & 0xFF;
02 int green = (rgb >> 8) & 0xFF;
03 int blue = rgb & 0xFF;
```

In Processing läuft dies alles etwas komplizierter ab, aber ein wichtiger Schritt ist, dass, wenn ein zu hoher oder niedriger Wert dem Farbobjekt gegeben wurde, der Wert auf das Maximum oder Minimum gesetzt wird.

Nun werden einige Processing interne Aktionen beschrieben. Zunächst werden an dem Frame einige allgemeine Listener hinzugefügt, die zum Beispiel regeln, was passiert, wenn das Fenster geschlossen wird. In einem Byte Array befinden sich auch die Daten vom Logo, damit es schneller geladen werden kann. Danach werden die Kommandozeilenargumente abgearbeitet, die mit zwei Bindestrichen beginnen. Das passiert in der Methode **runSketch()**, an die String args[] und das Applet übergeben werden. Wenn kein Argument angegeben wurde, wird das Programm sofort beendet. Ansonsten werden einige Einstellungen vorgenommen, die durch die übergebenen Strings aktiviert werden:

Parameter	Funktion
--location=x,y	Gibt die Position des Applets an
--present	Aktiviert den Fullscreen-Modus
--exclusive	Exklusiver Fullscreen, der andere Fenster und Interaktionen mit anderen Monitoren deaktiviert; wie ein „Spiele-Modus“
--hide-stop	Versteckt den Stopbutton
--stop-color=#xxxxxx	Verändert die Farbe des Buttons
--bgcolor=#xxxxxx	Verändert die Hintergrundfarbe des Fensters
--sketch-path	Ändert den Ort, an den Dateien durch die Funktionen saveStrings() und saveFrame() gespeichert werden
--display=n	Wählt den Monitor aus
--external	Ist gesetzt, wenn das Applet von der IDE aufgerufen wird.
--editor-location=x,y	Position des Editors der IDE

Die letzten beiden Argumente werden nur von der Processing IDE verwendet. Nun wird ein neues Frame Objekt instanziert und mit verschiedensten Einstellungen modifiziert, wie der Position, der Farbe und so weiter. Es werden Listener und der Stopbutton hinzugefügt und abschließend der Frame mit `setVisible()` sichtbar gemacht. Die Methode `runSketch()` wird ganz am Anfang von der Hauptfunktion eines jeden Java Programmes aus aufgerufen:

`public static void main(final String[] args);` Die Argumente werden direkt übergeben.

Widmet man sich wieder den Exportfunktionen, fehlt noch eine bedeutende Methode. Mit **beginRecord()** und **endRecord()** lassen sich alle Zeichenfunktionen, sowie die Ausgabe des Fensters in eine Datei umleiten. Dazu muss ein Renderer und der Dateiname angeben werden. Intern dient als Rekorder ein PGraphics Objekt.

Um aus 3D Daten Vektoren zu exportieren, kann **beginRaw()** in Kombination mit **endRaw()** aufgerufen werden. Die Formen werden dabei genommen, bevor sie auf den Bildschirm gezeichnet werden. An dieser Stelle ist die gesamte Szene nur eine lange Liste an individuellen Linien und Dreiecken. Das heißt, dass Formen, die mit `sphere()` erstellt wurden, aus 100en Dreiecken bestehen und nicht etwa nur aus einem Objekt. Mehrteilige Formen wie Kurven werden auch als individuelle Teile gerendert. Es können 2D sowie 3D Renderer verwendet werden, wobei **hint(ENABLE_DEPTH_SORT)** die Erscheinung von 3D-Grafik verbessern kann. Um einen Hintergrund zu zeichnen, muss mit **rect(0, 0, width, height)** ein Rechteck über den gesamten Sketch gezogen werden, da `background()` nicht als Form erkannt wird.

Nicht immer will man Daten exportieren, manchmal will man nur einzelne Pixel manipulieren. `loadPixels()` und `updatePixels()` sind bereits bekannt. Beim ersten Befehl werden die Pixel in ein Array geladen, beim Zweiten wird für die entsprechenden Pixel die Funktion `updatePixels(int x1, int y1, int x2, int y2)` mit den richtigen Parametern aufgerufen.

Das eine oder andere Mal werden Objekte durch den Renderer nicht ideal angezeigt, mit Hints und Hacks kann man hier nachhelfen:

Hint/Hack	Funktion
<code>hint(ENABLE_OPENGL_4X_SMOOTH)</code>	Aktiviert 4x anti-aliasing für OpenGL
<code>hint(DISABLE_OPENGL_2X_SMOOTH)</code>	Deaktiviert 2x anti-aliasing für OpenGL
<code>hint(ENABLE_NATIVE_FONTS)</code>	Aktiviert native Fonts
<code>hint(DISABLE_DEPTH_TEST)</code>	Deaktiviert die Z-Sortierung in 3D
<code>hint(ENABLE_DEPTH_SORT)</code>	Aktiviert die Z-Sortierung für Linien und Dreiecke
<code>hint(DISABLE_OPENGL_ERROR_REPORT)</code>	Deaktiviert die Überprüfung auf Fehler bei OpenGL

Früher wurden Hints mit `unhint()` rückgängig gemacht, bei späteren Versionen wurden die jeweils gegenteiligen Konstanten(ENABLE/DISABLE) eingeführt.

Nun werden einige Möglichkeiten gezeigt, wie man verschiedene Formen in Processing zeichnen kann. Um komplexe Formen zu zeichnen, eignet sich `beginShape()` bzw. `endShape()`. In der Anfangsfunktion gibt man die Art der Form an, zur Verfügung stehen die Konstanten POINTS, LINES, TRIANGLES, TRIANGLE_FAN, TRIANGLE_STRIP, QUADS, QUAD_STRIP, die alle aus dem Kapitel OpenGL und Java OpenGL geläufig sind. Der Endfunktion kann die optionale Konstante CLOSE übergeben werden, die angibt, dass das Ende der Form mit dem Anfang verbunden werden soll. Um einzelne Punkte hinzuzufügen, wird `vertex()` benutzt, für Kurven stehen die Funktionen `curveVertex()` und `bezierVertex()` zur Verfügung. `curveVertex()` muss mindestens vier Mal hintereinander aufgerufen werden, damit eine Kurve entsteht. `bezierVertex()` verlangt von Haus aus sechs bzw. neun Argumenten, je nachdem, ob es eine z-Koordinate gibt oder nicht. Beide Methoden können auch als Grundformen mit `curve()` und `bezier()` genutzt werden. Um ein Viereck darzustellen, kann `quadraticVertex()` mit vier oder sechs Argumenten aufgerufen werden.

Diese entstehenden Formen können auch mit `texture()` texturiert werden. Als Argument muss ein PImage erfolgen. Es lässt sich dabei der Koordinatenraum ändern, wenn es nötig ist. Es gibt zwei Optionen, IMAGE, das aktuelle Koordinatensystem oder NORMALIZED. Den Unterschied kann man am besten anhand eines Beispiels sehen. Wenn ein Bild 100 x 200 Pixel groß ist, benötigt die Abbildung mit IMAGE auf einem gesamten Quadrat die Punkte (0,0) (0,100) (100,200) (0,200). Die Gleiche Abbildung würde mit NORMALIZED (0,0) (0,1) (1,1) (0,1) benötigen. Mit `noTexture()` lässt sich die Textur wieder deaktivieren.

Um primitive Formen nutzen zu können, muss keine neue Form kreiert werden. Es reichen einfache englische Befehle um die Formen zu beschreiben.

Befehl Variante 1	Befehl Variante 2
<code>point(float x, float y)</code>	<code>point(float x, float y, float z)</code>
<code>line(float x1, float y1, float x2, float y2)</code>	<code>line(float x1, float y1, float z1, float x2, float y2, float z2)</code>
<code>triangle(float x1, float y1, float x2, float y2, float x3, float y3)</code>	
<code>quad(float x1, float y1, float x2, float y2, float x3, float y3, float x4, float y4)</code>	
<code>rect(float a, float b, float c, float d)</code>	<code>rect(float a, float b, float c, float d, float r)</code>
<code>rect(float a, float b, float c, float d, float tl, float tr, float br, float bl)</code>	
<code>arc(float a, float b, float c, float d, float start, float stop)</code>	
<code>box(float size)</code>	<code>box(float w, float h, float d)</code>

```
sphere(float r)  
ellipse(float a, float b, float c, float d)
```

Einige dieser Befehle können durch weitere Befehle konfiguriert werden. Die angegebenen Koordinaten beziehen sich bei normalen Einstellungen auf die linke obere Ecke der Form. Mit **rectMode()** bzw. **ellipseMode()** lässt sich dies ändern. Die Standardeinstellung ist CORNER, wenn CORNERS benutzt wird, definieren die ersten zwei Koordinaten eine Ecke, die anderen beiden Koordinaten legen die gegenüberliegende Ecke fest. Um das Objekt mittig zu platzieren, gibt es die Konstante CENTER, mit RADIUS werden die letzten Koordinaten als Radius verwendet.

Auch die 3-dimensionale Form Sphere besitzt eine Funktion, die sich **sphereDetail()** nennt. Eine Kugel besteht aus einzelnen Vertex Objekten, in der Standardeinstellung besteht sie aus 30 Teilen, das heißt alle $360/30 = 30$ Grad ein Teilstück. Die komplexen Formen wurden bereits beschrieben, hier kommen noch einige Ergänzungen: Mit **bezierPoint()** lassen sich Punkte einer Bézierkurve berechnen. Die Verwendung kann am besten anhand eines kleinen Beispiels demonstriert werden:

```
01 noFill();  
02 bezier(85, 20, 10, 10, 90, 90, 15, 80);  
03 fill(255);  
04 int steps = 10;  
05 for (int i = 0; i <= steps; i++) {  
06   float t = i / float(steps);  
07   float x = bezierPoint(85, 10, 90, 15, t);  
08   float y = bezierPoint(20, 10, 90, 80, t);  
09   ellipse(x, y, 5, 5);  
10 }
```

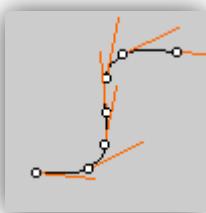
Als Ausgabe erhält man eine Bézierkurve, bei der die einzelnen Punkte mit einer Ellipse markiert wurden. Als Argument übergibt man jeweils die x- bzw. y-Koordinaten der Kurve und gibt ein Teilstück von 0 bis 1 an.

Mit **bezierTangent()** kann die Tangente der Kurve berechnet werden. Eine Tangente ist in der Geometrie eine Gerade, die eine Kurve in einem bestimmten Punkt berührt. Auf der offiziellen Processing Homepage findet man hierzu wieder ein Beispiel.⁵

⁵ Siehe http://processing.org/reference/bezierTangent_.html

```

01 noFill();
02 bezier(85, 20, 10, 10, 90, 90, 15, 80);
03 int steps = 6;
04 fill(255);
05 for (int i = 0; i <= steps; i++) {
06   float t = i / float(steps);
07   // Get the location of the point
08   float x = bezierPoint(85, 10, 90, 15, t);
09   float y = bezierPoint(20, 10, 90, 80, t);
10   // Get the tangent points
11   float tx = bezierTangent(85, 10, 90, 15, t);
12   float ty = bezierTangent(20, 10, 90, 80, t);
13   // Calculate an angle from the tangent points
14   float a = atan2(ty, tx);
15   a += PI;
16   stroke(255, 102, 0);
17   line(x, y, cos(a)*30 + x, sin(a)*30 + y);
18   // This following line of code makes a line
19   // inverse of the above line
20   //line(x, y, cos(a)*-30 + x, sin(a)*-30 + y);
21   stroke(0);
22   ellipse(x, y, 5, 5);
23 }
```



Es wird in diesem Beispiel wieder die Funktion *bezierPoint()* und die neue Methode *bezierTangent()* mit den gleichen Argumenten verwendet. Um den Winkel zwischen den Tangentenpunkten zu berechnen, kann der Befehl **atan2()** benutzt werden.

Um die Auflösung der Bézierkurve zu ändern, gibt es die Funktion **bezierDetail()**. Die Standardeinstellung ist 20 und kann je nach Belieben angepasst werden. Diese Methode funktioniert allerdings nur mit dem P3D oder OpenGL Renderer, der Standard Java2D Renderer ignoriert diese Information.

Analog zu *bezierPoint()* kann für normale Kurven **curvePoint()** eingesetzt werden. Für Tangenten entspricht dies der Funktion **curveTangent()**. Die Parameter gleichen denen der Bézier-Varianten. Zusätzlich zu **curveDetail()** gibt es **curveTightness()**, dessen Argument bestimmt, wie sehr die Kurve zu den gegebenen Punkten passt. Der Standardwert 0.0 definiert die Kurve als Catmull-Rom-Spline, eine Kurve, die nach Edwin Catmull und Raphael Rom benannt wurde. Ein Wert von 1.0 lässt die Punkte mit geraden Linien verbinden. Wenn der Wert im Bereich -5 bis 5 liegt, wird die Auslenkung der Kurve verändert, die definierten Punkte der Kurve bleiben dabei an der gleichen Stelle.

Runde Objekte erscheinen bekanntlich bei Computerberechnungen eckig. Daher muss eine Kantenglättung(anti-aliasing) durchgeführt werden. Bei Processing wird diese mit **smooth()** aktiviert und mit **noSmooth()** deaktiviert. Durch diese Berechnungen geht die Framerate der Anwendung zurück, man erreicht dadurch aber eine bessere visuelle Darstellung. Außerdem wird die Qualität von Bildern verbessert, deren Größe verändert wurde. Wenn Antialiasing

deaktiviert wird, erscheinen sowohl Bilder als auch Fonts nicht mehr so rund. Beim OpenGL Renderer ist diese Einstellung automatisch aktiviert und kann nicht abgestellt werden. In Java wird hierbei ein Rendering Hint gesetzt.

Um beim Thema Bilder zu bleiben, sehe man sich die Image und Shape Funktionen an. Bilder werden wie bereits bekannt mit **image()** angezeigt, wobei die Parameter variieren können. Es muss mindestens ein PImage Objekt an erster Stelle übergeben werden, sowie die Position, an der es dargestellt werden soll. Zusätzlich kann man zum Beispiel die gewünschte Höhe und Breite des Bildes angeben. Die Positionierung kann mit **imageMode()** und den früher erwähnten Konstanten geändert werden. Die zu zeigenden Grafiken müssen sich im *data* Ordner befinden. Sollte eine GIF-Grafik geladen werden, die Transparenz beinhaltet, bleibt diese im Sketch enthalten. Animationen können auf diese Art nicht dargestellt werden, dazu muss eine contributed Library wie **gifAnimation** verwendet werden.

Vektorgrafiken werden bekanntlich mit **shape()** dargestellt, die Parameter gleichen der *image()* Funktion. Die Parameter kann man analog zu den vorherigen Methoden mit **shapeMode()** beeinflussen.

Als Nächstes sind die textspezifischen Kommandos an der Reihe. Um den Text vertikal und horizontal zu positionieren, kann man **textAlign()** benutzen. LEFT, CENTER und RIGHT sind die gültigen Konstanten, das zweite Argument ist optional. Wenn es nicht gesetzt ist, wird es standardmäßig auf BASELINE gestellt, ansonsten sind TOP, CENTER und BOTTOM Alternativen. Wenn bei **text()** die Breite und Höhe angegeben wird, wird BASELINE ignoriert und als TOP behandelt. Ansonsten würde der Text außerhalb der Box gezeichnet werden, da BASELINE die Standardeinstellung ist. BASELINE ist keine gute Variante, wenn der Text in einem Rechteck gezeichnet werden soll. Die vertikale Ausrichtung basiert auf den Wert von **textAscent()**. Dieser Wert wird von vielen Schriftarten nicht korrekt spezifiziert. Es kann notwendig sein, einen Abstand von ein paar Pixeln einzubringen, damit der Abstand richtig gestellt wird. Es kann auch ein gewisser Prozentsatz von *textAscent()* oder **textDescent()** genommen werden, damit der Hack auch funktioniert, wenn die Schriftgröße verändert wird.

Eine weitere wichtige Textfunktion ist **textFont()**, die schon ausreichend dokumentiert wurde. Der Zeilenabstand lässt sich mit **textLeading()** konfigurieren. Der Abstand zwischen den Linien wird in Pixel angeben und allen darauffolgenden Aufrufen von **text()** verwendet.

Die Methode **textMode()** darf auf keinen Fall mit Funktionen wie **rectMode()** oder **ellipseMode()** und niemals mit **textAlign()** verwechselt werden. **textMode()** bestimmt, wie der Text

auf dem Bildschirm gerendert wird. Im Standardmodus MODEL kann der Text im 2- und 3-dimensionalen Raum rotiert, skaliert und positioniert werden. Bei einer Veränderung zu SCREEN, werden die Buchstaben direkt in den Vordergrund des Fensters gezeichnet und die Renderqualität und - geschwindigkeit steigt mit dem P2D und P3D Renderer. Mit OPENGL und JAVA2D dauert das Rendern länger, dafür sind die gesetzten Pixel genauer als bei den anderen beiden Renderern. Mit *textMode(SCREEN)* werden die Buchstaben in tatsächlicher Größe gezeichnet und Aufrufe von *textSize()* haben keine Wirkung. Außerdem wird die z-Koordinate bei der Funktion *text()* nicht beachtet, da der Bildschirm bekanntlich flach ist. Der SHAPE Modus rendert den Text, indem die Umrisse der einzelnen Buchstaben, anstatt als Textur verwendet werden. Dieser Modus wird nur von PDF und OPENGL Renderer unterstützt, bei PDF muss der Befehl vor allen anderen Befehlen erfolgen, die etwas auf den Bildschirm zeichnen. Wenn die Umrisse nicht verfügbar sind, wird Anstelle *textMode(MODEL)* ausgeführt. In OPENGL kann der Modus mit *beginRaw()* kombiniert werden um exakten, vektorbasierenden Text in eine 2D oder 3D Ausgabedatei, zum Beispiel DFX oder PDF zu exportieren. Dieser Modus ist nicht für OPENGL optimiert und es sollte bei Problemen der MODEL Modus angewandt werden.

Oft ist es im Sketch nötig, die Breite des Textes zu berechnen, um ihn an das Kunstwerk anpassen zu können. Processing bietet dafür den Befehl **textWidth()** an, der die Breite als Float zurückgibt. Der wichtigsten Funktion **text()** kann ein String oder Char übergeben werden. Wenn *textMode()* und Java 2D aktiviert ist, werden einzelne Buchstaben nur langsam gezeichnet, da *loadPixels()* und *updatePixels()* aufgerufen werden müssen. Zeilenumbrüche im Text können durch „\n“ erreicht werden (Unix newline oder linefeed char, ASCII 10). Der Zeilenumbruch von Windows und Mac OS (Wagenrücklauf/ carriage return) wird nicht akzeptiert und wird ignoriert. Zusätzlich zu einem einzelnen Char können auch ganze Arrays vom Typ Char geschrieben werden, da String eigentlich Char Arrays sind und sie in der Funktion *text()* in Arrays umgewandelt werden. Zusätzlich muss der Start- und Endindex im Array als zweites und drittes Argument mitgegeben werden. Die Textfunktion wird auch im 3-dimensionalen Raum mit einer dritten Koordinate unterstützt. Ein besonders Feature ist, dass Zahlen ähnlich wie mit *nf()* auf eine einfache Weise formatiert werden.

Nun ist es an der Zeit die Möglichkeiten kennenzulernen, Transformationen, Rotationen und Skalierungen vorzunehmen. Eine Hilfsfunktion ist **pushMatrix()**, die die aktuelle Transformationsmatrix auf den Matrixstack legt. Vektoren sind aus dem Mathematikunterricht bekannt, mit ihnen kann man Punkte im Raum beschreiben. In der Geometrie gibt es Matrizen. Sie eignen sich besonders für das Beschreiben von Drehungen und Spiegelungen. Ein sogenannter Stack lässt sich im Deutschen gut als Stapel erklären. Ein Matrixstack ist daher ein

Stapel mit Matrizen. Er weist einige Merkmale auf: Neue Werte, die auf den Stack gelegt werden, liegen ganz oben. Der oberste Wert kann vom Stack wieder entnommen werden. Dieses Prinzip nennt man in der Informatik „Last In First Out“ (LIFO). Zugriffe auf Werte, die sich im Stapel befinden und nicht oben liegen, sind nicht erlaubt. `pushMatrix()` speichert das aktuelle Koordinatensystem auf den Stack, die Methode `popMatrix()` stellt das frühere Koordinatensystem wieder her.

Um Objekte in x-, y- und z-Richtung zu verschieben, kann `translate()` angewandt werden. Für die Verschiebung im 3-dimensionalen Raum ist der entsprechende Renderer notwendig. Wenn mehrere Male der Befehl ausgeführt wird, werden die Transformationen addiert. Wenn zum Beispiel nach `translate(50, 50, 0)` `translate(-50,-50,0)` geschrieben wird, heben sich die beiden Befehle gegenseitig auf. Damit außer das darauffolgende Objekt keine anderen Objekte beeinflusst werden, kann er mit `pushMatrix()` und `popMatrix()` eingeklammert werden. Am Ende von `draw()` wird `translate()` wieder zurückgesetzt.

Auf die gleiche Weise finden Rotationen mit `rotate()` statt. Der Winkel muss im Bereich 0 bis 2 Pi in Radianen angegeben werden. Um die Arbeit zu erleichtern, kann man die Konstanten QUARTER_PI, HALF_PI, PI und TWO_PI einsetzen. Sollte der Winkel größer als erlaubt sein, beginnt Processing wieder bei 0 zu zählen, bis der Winkel im Bereich 0 bis 2 Pi liegt. Der Winkel kann auch von Grad in Radianen und vice versa mit `radians()` bzw. `degrees()` umgerechnet werden. Um einzelne Achsen zu rotieren, kann einer der Methoden `rotateX()`, `rotateY()` oder `rotateZ()` verwendet werden. Es ist dabei zu beachten, dass Objekte an der relativen Position zum Ursprung rotiert werden. Damit sie sich um sich selbst drehen, müssen sie vorher mit `translate()` verschoben werden. Es kann auch eine Drehung anhand eines Vektors erfolgen, in der Form `rotate(float angle, float vx, float vy, float vz)`.

Der dritte angesprochene Punkt sind Skalierungen mit Hilfe von `scale()`. Damit die Größe verändert wird, werden die einzelnen Teile der Form erweitert oder kontrahiert. Objekte skalieren immer abhängig von deren Abstammung. Als Argument wird ein Prozentsatz angegeben. Um zum Beispiel eine Form auf 200% zu vergrößern, wird die Funktion mit `scale(2.0)` aufgerufen. Wie auch bei den vorherigen Methoden kann die Beeinflussung der Funktion abgegrenzt werden mit `pushMatrix()` und `popMatrix()`.

Ein weiterer Matrixbefehl ist `resetMatrix()`, mit der die aktuelle Matrix durch die Einheitsmatrix oder Identitätsmatrix ersetzt wird. Das ist eine quadratische Matrix, die nur aus Nullen besteht, ausgenommen den Hauptdiagonalen, die mit Einsen gefüllt sind. In OpenGL gibt es einen identischen Befehl: `glLoadIdentity()`.

Die aktuelle Matrix kann auch mit einer anderen durch **applyMatrix()** multipliziert werden. Dieser Vorgang ist sehr langsam und sollte vermieden werden. Übergeben wird dabei ein Objekt der Klasse PMatrix, PMatrix2D, PMatrix3D oder eine 4 x 4 Matrix in Form von Float Werten. Die äquivalente Funktion in OpenGL lautet *glMultMatrix()*.

Um die aktuelle Matrix abzuspeichern, wird mit **getMatrix()** die Matrix zurückgeben, damit sie einem Objekt zugewiesen werden kann. Es kann auch das Objekt direkt als Parameter übergeben werden, damit es kopiert wird. Eine neue Matrix kann durch das Übergeben von Null erreicht werden. **setMatrix()** erlaubt das Definieren einer neuen Transformationsmatrix. Zum Testen kann die Matrix mit **printMatrix()** auf der Konsole angezeigt werden.

Sehr umfassend ist auch die Anzahl der Befehle, die mit der Kamera in Verbindung gebracht werden. Die einfache Konfiguration der Kamera erfolgt mit **camera()**. Die ersten drei Koordinaten bestimmen die Position des Auges der Kamera, die nächsten den Punkt am Bildschirm, auf den die Kamera gerichtet ist und die letzten drei Argumente welche Achsen aufwärts gerichtet sind. Die Grafiken können dadurch aus verschiedenen Winkeln betrachtet werden. Die parameterlose Version setzt die Kamera an die Standardposition, dabei zeigt sie auf die Mitte des Bildschirms und die Y-Achse ist nach oben gerichtet. Dies entspricht `camera(width/2.0, height/2.0, (height/2.0) / tan(PI*60.0 / 360.0), * width/2.0, height/2.0, 0, 0, 1, 0)`.

OpenGL hat dafür die Funktion *gluLookAt()*, bei der die Kameraeinstellung davor jedoch nicht gelöscht wird.

Für weitere Einstellungen wird **beginCamera()** mit abschließenden **endCamera()** genutzt. Die Funktion ist zwar für eine größere Kontrolle über die Kamerabewegungen nützlich, aber für die meisten User nicht nötig, da *camera()* ausreichend ist. Die eingeschlossenen Funktionen werden alle Transformationen wie *rotate()* und *translate()*, die davor in *draw()* erschienen ersetzen, die Kameratransformation wird dabei nicht automatisch ersetzt. Daher sollten Kamerafunktionen am Anfang von *draw()* platziert werden, *camera()* kann nach *beginCamera()* geschrieben werden, wenn die Kamera zurückgesetzt werden soll, bevor neue Transformationen angewendet werden. Diese Funktion setzt den Matrixmodus zur Kameramatrix und somit beeinflussen Aufrufe wie *translate()*, *rotate()*, *applyMatrix()* und *resetMatrix()* die Kamera. *beginCamera()* und *endCamera()* können nicht ineinander verschachtelt werden.

Gleich wie bei Matrizen kann die Kameramatrix mit **printCamera()** ausgegeben werden. Um den Koordinatenursprung zu ändern, kann die Funktion **ortho()** aufgerufen werden. Sie er-

wartet sechs Argumente, die man unter http://processing.org/reference/ortho_.html nachschlagen kann. Der parameterlose Aufruf wird in `ortho(0, width, 0, height, -10, 10)` geändert.

Die Perspektive lässt sich mit **`perspective()`** ändern. Der Befehl wendet eine perspektivische Projektion mit Verkürzung an und entfernte Objekte erscheinen kleiner als nähere Objekte. Die Parameter definieren einen Sichtbereich in Form einer abgestumpften Pyramide. Objekte, die sich in der Nähe der Vorderseite des Bereiches befinden, erscheinen in der tatsächlichen Größe, während weiter weg entfernte Objekte kleiner erscheinen. Diese Projektion simuliert die Perspektive der Welt viel genauer als die orthografische Projektion, bei der der Ansichtsbereich wie ein Kasten geformt ist. Ohne Parameter sind die Standardwerte: `perspective(PI/3.0, width/height, cameraZ/10.0, cameraZ*10.0)`, wobei `cameraZ ((height/2.0) / tan(PI*60.0/360.0))` ist.

Eine neue Perspektivenmatrix kann mit **`frustum()`** eingestellt werden. Dabei muss die linke, rechte, untere und obere Koordinate der sogenannten „Clip-Ebene“ definiert werden, sowie das naheste und das am weitesten entfernte Objekt der Clip-Ebene. Alle Parameter müssen in Form eines Float bereitgestellt werden. Die Methode funktioniert wie `glFrustum()`, außer, dass die aktuelle Perspektivenmatrix ausgelöscht und nicht mit sich selbst multipliziert wird.

2D und 3D Koordinaten müssen das ein oder andere Mal umgerechnet werden. Um dies zu erleichtern gibt es von beiden Seiten eine Funktion, die das erledigt. Um 3D Koordinaten umzuwandeln, werden die drei Funktionen **`screenX()`**, **`screenY()`** und **`screenZ()`** verwendet, die eine 2D Koordinate zurückgeben, an der die 3-dimensionale Koordinate am Monitor erscheinen würde. Um 2D Koordinaten in den 3-dimensionalen Raum zu transferieren, werden **`modelX()`**, **`modelY()`** und **`modelZ()`** angewandt. Sie wandeln die Koordinaten in eine Position im Modelbereich um. Dabei werden aktuelle Transformationen wie Skalierungen, Rotationen und Translationen beachtet.

Um Abwechslung ins Thema zu bringen, werden nun einige Funktionen erklärt, die den Stil eines Sketches verändern können. Gleich wie `pushMatrix()` für Matrizen gibt es für den Stil eine eigene Funktion, die den Wirkungsbereich einschränkt. **`pushStyle()`** speichert die aktuellen Stilinformationen in eine Objekt der Klasse `PStyle`, die mit **`popStyle()`** wiederhergestellt werden. Die beiden Methoden müssen immer gemeinsam im Programmcode vorkommen.

In Processing Sketchen kommen die unterschiedlichsten Linien vor, sei es die Umrandung eines Kreises, eines Rechtecks oder eine anderen Form. Die Dicke der Linien wird mit **`strokeWeight()`** gesetzt. P2D, P3D und OPENGL produzieren unschöne Ergebnisse bei Serien von verbundenen Linien, zum Beispiel bei Polygonen, Dreiecken oder Ellipsen, wenn `stroke-`

Weight() gesetzt ist. Dieser Bug wurde bis zur Version 1.5.1 nicht ausgebessert. Bei OpenGL gibt es zusätzlich die Limitation, dass der minimale und maximale Wert von *strokeWeight()* von der Grafikkarte und der OpenGL Implementierung des Betriebssystems gesteuert wird und somit der Maximalwert beispielsweise auf 10 Pixel limitiert ist.

Wenn Linien aufeinander treffen gibt es mehrere Methoden sie zu rendern. Sie können vier-eckig, erweitert oder rund sein. Die zuständige Funktion ist **strokeCap()** und akzeptiert die Konstanten SQUARE, PROJECT und ROUND, wobei Letztere der Standardwert ist. Es ist zu beachten, dass diese Methode mit dem P3D und OPENGL Renderer nicht verfügbar ist.

Bilder lassen sich auch mit einer speziellen Funktion einfärben, nämlich **tint()**. Die reversible Variante ist **noTint()**. Farben werden gleich wie bei *fill()* mitgegeben. Um ein Bild transparent zu machen, muss der Grauwert auf weiß(255) gesetzt sein und dann kann als zweites Argument der Alphawert angegeben werden.

Um Formen mit Farbe zu füllen, wird **fill()** angewandt. Das Farbmodell ist anfangs RGB, die Werte können zwischen 0 und 255 liegen. Hexadezimalfarben, Grauabstufungen und Transparenz sind außerdem möglich. Die gegenteilige Methode ist **noFill()**.

Mit den Grundfunktionen von Processing lassen sich zwar nur zwei verschiedene 3-dimensionale Figuren rendern, dafür gibt es aber die bereits vorgestellten Kamerafunktionen und es können Lichter zur Szene hinzugefügt werden.

Um zu bestimmen, wie Lichter Formen und Oberflächen beeinflussen, kann der aktuelle Normalvektor gesetzt werden. Processing versucht zwar automatisch diesen den Formen zuzuweisen, da dies noch nicht ganz ausgereift ist, ist diese Funktion eine bessere Möglichkeit, um mehr Kontrolle über den eigenen Sketch zu erlangen. Diese Methode ist identisch zu *glNormal3f()* in OpenGL.

Die Reflexion der Umgebung an Formen kann mit **ambient()** in Kombination mit Umgebungslichtern(ambient lights) geregelt werden. Die Farbkomponenten der Parameter definieren die Reflexion. Beispielsweise im Standardfarbmodus mit den Einstellungen v1=255, v2=126, v3=0, würde das gesamte rote und das halbe grüne Licht reflektiert werden. Um die Glanzfarbe des Materials einer Form zu ändern, muss **specular()** aufgerufen werden. Mit diesem Glanzlicht ist das Licht gemeint, dass von der Oberfläche in eine gewünschte Richtung abprallt, im Gegensatz zu einem gestreuten Licht, dass in alle Richtungen abprallt. Um den Grad des Glanzes zu setzen, wird **shininess()** mit einem Float Parameter ausgeführt.

Das ausstrahlende Licht der Form kann mit **emissive()** bestimmt werden, als Parameter wird eine Farbe übergeben. Voreingestellte Lichter können mit **lights()** aktiviert und **noLights()** deaktiviert werden. Die folgende Tabelle zeigt die Einstellungen:

Licht	Einstellung
ambientLight()	128, 128, 128
directionalLight()	128, 128, 128, 0, 0, -1
lightFalloff()	1, 0, 0
lightSpecular()	0, 0, 0

Mit **ambientLight()** kann eine Lichtquelle hinzugefügt werden, deren Licht nicht von einer bestimmten Richtung kommt. Die Lichtstrahlen prallen von überall ab, sodass Objekte gleichmäßig von allen Seiten beleuchtet sind. Umgebungslichter werden immer in Kombination mit anderen Lichtern benutzt. Lichter müssen in *draw()* inkludiert werden, damit sie im Sketch persistent bleiben. Die Farbe ist abhängig von der Farbe, die als Parameter übernommen wird und vom aktuellen Farbmodus, der mit *colorMode()* verändert wird. Die Position der Quelle muss im 3-dimensionalen Raum mit 3 Koordinaten angegeben werden.

Mit **directionalLight()** kann ein gerichtetes Licht hinzugefügt werden. Direktes Licht kommt von einer Richtung und ist stärker, wenn es eine Oberfläche voll und ganz trifft und schwächer, wenn dies in einem schwachen Winkel passiert. Nachdem es die Oberfläche getroffen hat, streut das direkte Licht in alle Richtungen. Zusätzlich zur Farbe muss deshalb die Richtung angegeben werden, aus der es kommt. Wenn die Parameter nx, ny und nz lauten und ny auf -1 gesetzt wurde, werden die Objekte von oben beleuchtet.

Um ein Punktlicht zu platzieren, gibt es **pointLight()**. Die Argumente orientieren sich an den anderen Funktionen. Um ein Spotlicht, sozusagen ein Scheinwerferlicht in den Sketch zu bringen, kann **spotLight()** mit 11 Argumenten ausgeführt werden. Die ersten drei Werte geben die Farbe an, die nächsten die Position und die darauffolgenden drei die Richtung, in die das Licht scheint. Zusätzlich sind der Winkel, sowie die Konzentration notwendig.

Da bei Lichtern Farben eine große Rolle spielen, müssen die Farbfunktionen auch verstanden werden. Bevor dieses Thema angegangen werden kann, hat der Befehl **background()** Vorrang. Diese Funktion ändert die Hintergrundfarbe des Processing Fensters. Die Fläche, die eingefärbt wird, zählt nicht als Form. Der Standardhintergrund ist in hellem Grau gehalten. In *draw()* wird sie am Anfang eines jeden Frames verwendet, um die Hintergrundfarbe zurückzusetzen. Es ist wichtig, dass dies am Anfang geschieht, da sonst die bereits gezeichneten

Formen wieder übermalt werden. Es können auch Bilder als Hintergrund benutzt werden, allerdings muss die Größe der Grafik mit dem Hintergrund übereinstimmen, ansonsten gibt es eine Fehlermeldung. Um ein Bild an die Größe anzupassen, verwendet man bei einem Bild den Befehl **resize(width, height)**, zum Beispiel bei einer Grafik ,b': b.resize(width, height). Als Hintergrund benutzte Bilder ignorieren die aktuelle *tint()* Einstellung. Es ist nicht möglich den Hintergrund transparent zu machen, das funktioniert nur mit *createGraphics()*. Erfahrene User können den Frame transparent machen. Die Anleitung dazu findet man unter http://java.sun.com/developer/technicalArticles/GUI/translucent_shaped_windows/ Es ist sehr verlockend, dass man versucht den Bildschirm bei jedem Frame nur teilweise zu löschen, das ist jedoch mit dieser Funktion nicht möglich. Wenn sie aufgerufen wird, werden die Pixel durch andere Pixeln ersetzt, die gleich transparent sind. Um eine halbtransparente Fläche zu bekommen, muss der Befehl **fill()** verwendet werden.

Wie Farben funktionieren, wurde bereits erklärt. Die folgende Auflistung gibt Aufschluss darüber, welche Möglichkeiten es gibt, um Komponenten des HSB und RGB Modus zu filtern. **alpha()** filtert die Transparenz.

RGB	HSB
float red()	float hue()
float green()	float saturation()
float blue()	float brightness()

Es gibt eine zusätzliche Farbfunktion, die sich **lerpColor()** nennt. Sie berechnet einen Farbton zwischen zwei Farben an einer bestimmten Steigung. Zusätzlich zu den zwei color Objekten muss ein Wert zwischen 0 und 1 angegeben werden, der bestimmt, welcher Farbe der neue Ton näher ist. In der Dokumentation wird vom *amt* Parameter gesprochen und die Technik nennt sich Interpolation. Daher ist der neue Punkt bei einem Wert von 0.5 genau in der Mitte von beiden Farben. Abgesehen von einigen internen Hilfsfunktionen ist nun fast das Ende der Klasse PApplet erreicht. Nun erfolgen noch einige Funktionen, mit denen man Pixel manipulieren kann.

Eine Funktion, die speziell für Anfänger gedacht ist, gibt den Farbwert eines bestimmten Pixels zurück. **get()** returned genauer gesagt einen ARGB color Typ. Wenn die Koordinate außerhalb des Bildes ist, wird 0 zurückgegeben (eigentlich schwarz, aber vollständig transparent). Wenn das Bild im RGB Format ist, wie zum Beispiel ein PVideo Objekt, werden die ho-

hen Bits gesetzt, um Fälle zu vermeiden, in denen sie noch nicht gesetzt wurden. Wenn das Bild im ALPHA Format vorliegt, wird Weiß und dessen Alphawert zurückgegeben. Durch diese beiden Überprüfungen ist die Methode etwas langsam; um die Sache effizienter zu erledigen, sollte man das *pixels* Array direkt adressieren. Es kann auch ein ganzer Bereich an Pixeln aufgeschnappt werden, indem die Breite und Höhe zusätzlich angegeben wird. Dadurch ist der Rückgabewert ein PImage. Um eine Kopie der ganzen Grafik zu erlangen kann *get()* parameterlos aufgerufen werden, gleichzustellend mit *get(0, 0, width, height)*.

Um Pixel zu setzen, muss **set()** oder das *pixels* Array verwendet werden. Gleiche Wirkung wie *set(x, y, #000000)*, erzeugt man mit *pixels[]* durch *pixels[y*width+x] = #000000*. Seit der Version 1.0 wird dabei *imageMode()* nicht beachtet. Diese Variante ist auch sehr effektiv um die Pixeln eines PImage Objektes zu setzen, dass anstelle der Farbe gesetzt werden kann.

Mit **mask()** kann der Alphakanal gesetzt werden. Schwarze Farben in der Quelle erscheinen im folgenden Bild transparent und die Farbe Weiß lässt Dinge undurchsichtig erscheinen. Genau genommen wird der blaue Anteil als Alphafarbe verwendet. Bei einer Grafik in Grautönen ist das korrekt, bei einem farbigen Bild liegt die Genauigkeit nicht bei 100%. Für eine genauere Umwandlung muss zuerst **filter(GRAY)** benutzt werden, um ein Bild mit Graustufen zu erlangen.

Es gibt einen zweiten Anwendungszweck dieser Funktion bei Objekten vom Typ PImage. Sie verdeckt Teile eines Bildes beim Anzeigen, indem ein anderes Bild geladen wird und dieses als Alphakanal genutzt wird. Dieses Mask Image sollte nur Graustufen enthalten und dieselben Maße wie das Orginalbild besitzen. Es kann auch ein Integer Array übergeben werden, dass Alpha Werte enthält. Dabei ist zu beachten, dass das Array gleich lang sein muss wie das *pixels* Array des Bildes und die Grauwerte sollten zwischen 0 und 255 liegen.

filter() kann nicht nur ein Bild in Graustufungen umrechnen, sondern auch einige andere Effekte anwenden. Die in Klammern angegeben Werte sind die Standardwerte:

Filter	Effekt
THRESHOLD (0.5)	Konvertiert ein Bild in Schwarz Weiß, abhängig ob die Pixeln über dem definierten Grenzbereich(zwischen 0.0 und 1.0) liegen oder nicht.
GRAY	Konvertiert die Farben des Bildes in Graustufungen.
INVERT	Invertiert jeden Pixel der Grafik.
POSTERIZE -	Limitiert die Anzahl der Farben der einzelnen Kanäle des Bildes, abhängig vom Parameter.

BLUR (1)	Führt einen Gaussian Blur durch, bei dem das Bild weichgezeichnet wird. Der Radius wird als Argument mitgegeben.
OPAQUE	Setzt den Alphakanal auf undurchsichtig.
ERODE	Reduziert die hellen Flächen mit der Menge, die an die Funktion übergeben wurde.
DILATE	Erhöht die hellen Flächen mit der Menge, die an die Funktion übergeben wurde

Bereich des Hintergrundes bzw. eines Bildes können mit der Funktion **copy()** an eine andere Stelle kopiert werden. Wenn die Quelleregion nicht dieselbe Größe wie die Zielregion hat, wird die Skalierung automatisch getätigt. Es werden zwar keine Alphainformationen bei diesem Vorgang benutzt, die transparente Farbe wird aber trotzdem kopiert. Die Funktion wird mit 8 bzw. 9 Argumenten aufgerufen, je nachdem ob die Quelle der Hintergrund oder ein PImage ist.

Farbe und Bilder lassen sich in Processing mischen. Es gibt dafür die Methode **blendColor()** bzw. die auf PImage bezogene Funktion **blend()**. Beide akzeptieren als Argument verschiedene Konstanten, die in folgenden Tabellen aufgelistet sind:

blendColor(c1, c2, MODE)

BLEND	MULTIPLY
ADD	SCREEN
SUBTRACT	OVERLAY
DARKEST	HARD_LIGHT
LIGHTEST	SOFT_LIGHT
DIFFERENCE	DODGE
EXCLUSION	BURN

blend(srcImg, x, y, width, height, dx, dy, dwidth, dheight, MODE)

BLEND	Lineare Interpolation von Farben. $C = A * \text{factor} + B$
ADD	Additive Mischung mit weißem Clip. $C = \min(A * \text{factor} + B, 255)$
SUBTRACT	Subtraktive Mischung mit schwarzem Clip. $C = \max(B - A * \text{factor}, 0)$
DARKEST	Nur die dunkelsten Farben bleiben bestehen. $C = \min(A * \text{factor}, B)$

LIGHTEST	Nur die hellsten Farben bleiben bestehen. $C = \max(A * \text{factor}, B)$
DIFFERENCE	Zieht Farben vom darunterliegenden Bild ab.
EXCLUSION	Ähnlich wie DIFFERENCE, nur weniger stark.
MULTIPLY	Multipliziert die Farben, das Ergebnis wird immer dunkler sein.
SCREEN	Gegenteil von MULTIPLY, bei dem der invertierte Wert der Farbe benutzt wird.
OVERLAY	Mischung aus MULTIPLY und SCREEN. Dunkle Werte werden multipliziert, helle Werte benutzen SCREEN.
HARD_LIGHT	SCREEN wenn größer als 50% grau, ansonsten MULTIPLY.
SOFT_LIGHT	Mischung aus DARKEST und LIGHTEST. Ähnlich wie OVERLAY, aber nicht so grell.
DODGE	Verstärkt helle Farben und erhöht den Kontrast, dunkle Farben werden ignoriert.
BURN	Gegenteil von DODGE, dunkle werden verändert und helle Farben ignoriert.

Einige Funktionen, die bereits in ein vorheriges Kapitel gehört hätten und nicht dazu gepasst haben, sind die Zeit- und Datumsfunktionen. Processing kommuniziert dabei mit der Uhr auf dem Computer.

Funktionsname	Rückgabe
int day()	Der heutige Tag als Wert zwischen 1 und 31
int hour()	Die jetzige Stunde als Wert zwischen 0 und 23
int millis()	Die Millisekunden, die seit dem Start des Applets vergangen sind
int minute()	Die jetzige Minute als Wert zwischen 0 und 59
int month()	Das jetzige Monat als Wert zwischen 1 und 12
int second()	Die jetzige Sekunde als Wert zwischen 0 und 59
int year()	Das Jahr als ganze Zahl(z.B. 2011)

PConsts

PConsts.java ist ein Interface, das viele Konstanten vorgibt. Die meisten Konstanten sind Integer, deren Werte von 0 bis n durchnummeriert werden. Ein Teil davon sind auch vom Typ Float, wie zum Beispiel die Pi-Konstanten, wobei hier deutlich wird, wie ungenau Pi bei Processing verwendet wird. Einige Fehlermeldungen sind in Form von Strings gespeichert.

PFont

PFont ist die Klasse, in der Schriftarten gespeichert werden. Sie implementiert die Klasse PConstants. Die Entwickler nennen sie „Grayscale bitmap font class“, da die Schriftarten als Bitmap Grafik abgespeichert werden. Zunächst ein Überblick über die Variablen. Hinweis: Eine Glyphe ist die grafische Darstellung eines Schriftzeichens:

Variable	Bedeutung
int glyphCount	Anzahl der Glyphen; muss nicht mit der Arraylänge übereinstimmen.
Glyph glyphs	Die Glyphendaten.
String name	Name der Schriftart, sowie sie von Java erkannt wurde.
String psname	Name der Schriftart in Postscript (einer Seitenbeschreibungssprache), in der aus dem Font ein Bitmap gemacht wird.
int size	Die Orginalgröße der Schrift, als sie erstellt wurde.
boolean smooth	Gibt an ob die Schrift „rund“ gerendert wird (benötigt für native Impl.)
int ascent	Aszendent der Schrift.
int descent	Deszendent der Schrift.
int[] ascii	Eine Tabelle mit den ASCII Schriftzeichen.

boolean lazy	Gibt an, ob die Schriftart dynamisch geladen wird.
Font font	Wird für vom Stream geladene Schriften verwendet.
boolean stream	Gibt an, ob die Schrift von einem Stream geladen wurde
boolean subsetting	Bestimmt, ob <code>getFont()</code> null returnen soll, damit die native Schriftart benutzt wird
boolean fontSearched	True, wenn versucht wurde, eine native AWT Version der Schriftart zu finden
Font[] fonts	Array der nativen Fonts
BufferedImage lazyImage	Variablen, die benötigt werden, wenn die Schriftart dynamisch geladen wird. Standardmäßig z.B. bei <code>createFont()</code>
FontMetrics lazyMetrics	
int[] lazySamples	
HashMap<PGraphics, Object> cacheMap	In dieser Klasse können zusätzliche Metadaten für Unterklassen gespeichert werden.

Im Konstruktor werden einige Variablen initialisiert und bei den nativen Fonts zusätzliche Rendering Hints aktiviert. Es werden neue Glyph Objekte instanziert und in das Array ge-

schrieben. Zwei besondere Buchstaben, nämlich ‚d‘ und ‚p‘ werden zusätzlich in das Array eingefügt, damit der Aszendent und Deszendent berechnet werden kann. Die Position des ASCII Zeichensatzes wird im Array ascii zwischengespeichert. Sollte an das PFont Objekt ein InputStream übergeben werden, wird er in einen DataInputStream umgewandelt, damit wichtige Informationen mit **readInt()** ausgelesen werden können. Dann werden die Bitmaps ausgelesen und es wird versucht eine native Schriftart zu finden.

Viele in dieser Klasse geschriebenen Funktionen werden nicht erwähnt, da die Variablendeclarationen über die Speicherung der Daten genug Auskunft geben. Jedoch gibt es zwei Funktionen, die ein Processing User im Sketch benutzen kann.

Mit **list()** bekommt man ein Array mit allen auf dem System installierten Schriftarten. Die Liste bietet die Namen jeder Schriftart an, die man in **createFont()** einsetzen kann und somit Schriften dynamisch geladen werden. Um eine mit Processing erstellt Schriftart aus dem *data* Ordner zu laden, muss **loadFont()** mitsamt dem vollständigen Namen der Schriftart aufgerufen werden. Wie bereits in einem anderen Kapitel erwähnt, können native Fonts mit **hint(ENABLE_NATIVE_FONTS)** aktiviert werden, wenn der JAVA2D Renderer im Einsatz ist.

Zum Abschluss der Klasse muss die innere Klasse Glyph betrachten werden, die wichtige Informationen über die einzelnen Schriftzeichen enthält. Sie besitzt jeweils Methoden um die Headerinformationen des Zeichens und das Bitmap zu beschreiben und zu lesen. An die Funktionen wird immer ein DataOutputStream bzw. DataInputStream übergeben.

Variablenname	Gespeicherte Information
PImage image	Bitmap der Glyphe
int value	Zeichen
int height	Höhe der Glyphe
int width	Breite der Glyphe
int index	Index der Glyphe
int topExtent	Obere Abgrenzung
int leftExtent	Linke Abgrenzung

PStyle

Sie ist die kleinste Klasse in den core libraries, da sie nur einige Konstanten implementiert:

int imageMode	int strokeColor
int rectMode	float strokeWeight

int ellipseMode	int strokeCap
int shapeMode	Int strokeJoin
int colorMode	float ambientR, ambientG, ambientB
float colorModeX	float specularR, specularG, specularB
float colorModeY	float emissiveR, emissiveG, emissiveB
float colorModeZ	float shininess
float colorModeA	PFont textFont
boolean tint	int textAlign
int tintColor	int textAlignY
boolean fill	int textMode
int fillColor	float textSize
boolean stroke	float textLeading

PVektor

Mit dieser Klasse werden Vektoren in Processing beschrieben. Um Objekte als String darstellen zu können, muss in Java die Klasse Serializable implementiert werden, in PVektor wird dies getan. Die Klasse besteht aus sehr wenigen Variablen, wie die folgende Tabelle beweist:

Float x	Float y
Float y	Float[] array

Im Konstruktor werden die x-, y- und z-Koordinate gesetzt, Wenn z nicht angegeben ist, ist sie standardmäßig 0. Die Methode **set()** übernimmt diese Aufgabe nach der Initialisierung, sie übernimmt auch einen anderen PVektor oder ein Float Array als Parameter. Um den Vektor oder ein Float Array zu erlangen, kann die Methode **get()** mit und ohne Parameter aufgerufen werden. Das übergebene Array wird je nach Länge mit zwei oder drei Einträgen befüllt.

Um Werte oder Vektor zu addieren oder subtrahieren, kann **add()** bzw. **sub()** verwendet werden. Wenn drei Float übergeben werden, gibt es keinen Rückgabewert, bei zwei PVektor wird ein neuer PVektor zurückgegeben. Alternativ kann als drittes Argument auch ein PVektor übergeben werden, der durch die Funktion überschrieben wird.

Mit dem gleichen Prinzip funktionieren auch die anderen Methoden. Vektoren werden mit **mult()** multipliziert und **div()** dividiert. Es kann auch durch einen einzelnen Integer als zweites Argument dividiert werden.

Der euklidische Abstand zwischen zwei Punkten, in dem Fall zwischen zwei Vektoren, wird mit **dist()** so errechnet:

```

01 public float dist(PVector v) {
02     float dx = x - v.x;
03     float dy = y - v.y;
04     float dz = z - v.z;
05     return (float) Math.sqrt(dx*dx + dy*dy + dz*dz);
06 }
```

Das Skalarprodukt wird mit **dot()** ausgerechnet, dabei werden die gleichen Koordinaten multipliziert und dann werden alle zusammengerechnet. Die Formel ist vom Aufbau wie beim vorherigen Beispiel In Zeile 5 in der Funktion *sqrt()*.

Das Kreuzprodukt wird mit **cross()** errechnet. Die Formel dafür lautet:

$$(a_1, a_2, a_3) \times (b_1, b_2, b_3) = (a_2 b_3 - a_3 b_2, a_3 b_1 - a_1 b_3, a_1 b_2 - a_2 b_1)$$

Die Magnitude wird mit **mag()** bestimmt, sie wird aus der Wurzel des Skalarprodukts berechnet.

Um den Vektor zu normalisieren, kann die Funktion **normalise()** helfen. Sie rechnet die Magnitude aus und dividiert den Vektor durch diese, falls es nötig ist. Die Magnitude kann auch mit **limit()** limitiert werden. Dabei wird der Vektor normalisiert und mit dem Maximumwert multipliziert, sollte die Magnitude größer als der Maximalwert sein.

Eine neue Magnitude kann gesetzt werden mit **setmag()**, wobei vor der Länge auch der zu ändernde Vektor als Parameter angegeben werden kann. Der Vektor wird dann normalisiert und mit der Länge multipliziert.

Bei 2D Vektoren kann auch die Rotation mit **head2D()** kalkuliert werden. Er berechnet sich aus $\text{atan2}(-y, x) * -1$. Der Vektor kann mit **rotate()** gedreht werden. Die Ergebnisse der Rechnung sind: $x = x * \cos(\theta) - y * \sin(\theta)$ und $y = x * \sin(\theta) + y * \cos(\theta)$. Theta ist der übergebene Winkel.

Es lassen sich zwei Vektoren auch mit **lerp()** interpolieren, wobei die gleichnamige, allgemeine Processing Funktion lerp() aufgerufen wird. Um den Winkel zwischen den beiden Vektoren zu berechnen, wird **angleBetween()** angewandt. Zuerst wird das Skalarprodukt gebildet und die beiden Magnituden(v1mag, v2mag) werden ausgerechnet.

Dann wird der Winkel so berechnet: $\alpha = \text{acos}(dot / (v1mag * v2mag))$

Zu guter Letzt kann aus dem Vektor mit **array()** ein Array gebildet werden. Dabei werden die einzelnen Koordinaten in das bereits deklarierte Array *array* gespeichert.

XML

Die Extensible Markup Language ist eine Auszeichnungssprache, die erlaubt, Daten hierarchisch und strukturiert darzustellen. Beispielsprachen sind RSS, XHTML und Scalable Vector Graphics(SVG). Processing auch XML-Dateien lesen und über die Klasse XMLElement schreiben. Ab den Zeitpunkt, ab den die Arbeit geschrieben wurde, wurde das XMLElement durch PNode ersetzt, daher gibt es nur eine kurze Einführung: Über ein DocumentBuilderFactory Objekt wird ein neuer DocumentBuilder instanziert. Damit wird ein neues Objekt der Klasse Document erstellt, mit dem man auf die einzelnen Knoten in einer XML Datei zugreifen kann. Mit den folgenden Funktionen können Daten ausgelesen werden:

Funktion	Aufgabe
int getChildCount()	Gibt die Anzahl der Kinder zurück.
XMLElement getChild()	Gibt ein einzelnes Kind zurück.
XMLElement[] getChildren()	Gibt alle Kinder als ein XElement Array zurück.
String getContent()	Gibt den Inhalt eines Elements zurück.
int getInt()	Gibt das Attribute eines Elements als Integer zurück..
float getFloat()	Gibt das Attribute eines Elements als Float zurück
String getString()	Gibt das Attribute eines Elements als String zurück
String getName()	Gibt den Namen des Elements zurück

PMatrix/PMatrix2D/PMatrix3D

PMatrix ist ein Interface, das erwartet, dass einige Funktionen für Matrizenoperationen implementiert werden. PMatrix2D und PMatrix3D nehmen diese Implementation für die jeweilige Dimension vor. Damit eine Funktion nicht zweimal erklärt werden muss, wird hier nur auf die Klasse PMatrix3D fokussiert.

1	0	0	0
0	1	0	0
0	0	1	0
0	0	0	1

Processing verwendet eine 4 x 4 Matrix, bei der die einzelnen Elemente als Float gespeichert sind. Sie sind folgendermaßen durchnummieriert: m00...m03, m10...m13... bis m33. Bei einem parameterlosen Konstruktorauftruf wird die Funktion **reset()** aufgerufen, die dann die Funktion **set()** zum Initialisieren der Elemente aufruft. Ihr werden die Werte aller Elemente übergeben, die entweder 0 oder 1 lauten können. Die Matrix

ist quadratisch und hat sogenannte Diagonalen, die schräg durch das Schema verlaufen. Die Hauptdiagonale verläuft vom oberen linken zum unteren rechten Ende der Matrix, die Nebendiagonale beginnt rechts oben. Bei der Initialisierung bilden die Elemente mit dem Wert 1 die Hauptdiagonale. Es können die Elemente auch direkt an den Konstruktor übergeben werden, der sie an `set()` weiterleitet. `set()` kann auch ein PMatrix Objekt übergeben werden, um die aktuelle Matrix in das übergebene Objekt zu kopieren. Auch die Übergabe eines Float Arrays und einer 3 x 3 Matrix ist möglich, wobei hier die Matrix vervollständigt wird, sodass die Hauptgerade wieder aus Einsen besteht. Die hierdurch gebildete Matrix nennt sich Einheits- oder Identitätsmatrix. Nennt man diese Matrix I und bestimmt \vec{x} als beliebigen Vektor, dann gilt: $I\vec{x} = \vec{x}$. Das heißt, dass, wenn man die Identitätsmatrix mit einem Vektor multipliziert, das Ergebnis der Vektor ist. Dieser Trick wird genutzt, damit die Matrix initialisiert werden kann. Doch warum soll mit einem Vektor multipliziert werden? Die Koordinatenpunkte der einzelnen Formen, die transformiert werden sollen, werden mit dieser Matrix multipliziert, die durch verschiedene Transformationen im Laufe der Zeit verändert wird. Man könnte dies auch als Funktion interpretieren: $T(x) = A\vec{x}$. $T(A)$ ist eine Funktion, die eine Transformation durchführt, A ist die sogenannte Transformationsmatrix und \vec{x} ist ein Vektor, der die Koordinaten des Punktes enthält. Bei einer Matrixmultiplikation mit einem Vektor ist auf die Reihenfolge zu achten, die Matrix muss hierbei vorne stehen.

Die Transformationen lassen sich also in der Matrix speichern. Ganz allgemein gefragt: wo wird welche Transformation gespeichert?

1	0	0	0
0	1	0	0
0	0	1	0
0	0	0	1

Die einzelnen Spalten dieser Matrix werden als Spaltenvektor bezeichnet. Der erste Spaltenvektor speichert ein paar Transformationen an der X-Koordinate, nämlich in unserem Fall Rotationen, Skalierungen und Scherungen. Genauso verhält es sich mit dem zweiten und dritten Spaltenvektor, die für die y- bzw. z-Koordinate zuständig sind. In der vierten Spalte werden die Translationen gespeichert.

*cx	*cy	*cz	...
*cx	*cy	*cz	...
*cx	*cy	*cz	...
*cx	*cy	*cz	...

`rotateX(unterhalb)`

Um das Genannte zu demonstrieren, wird die Funktion `scale()` hergenommen. Sie lässt Skalierungen in alle 3 Richtungen zu. Bei der Skalierung werden die Zahlenwerte der Koordinaten x bis z mit jeweils einem konstanten Faktor multipliziert. Man nehme dafür die drei Faktoren cx, cy, cz. Die Matrix ändert sich so, wie es die Tabelle auf der linken Seite zeigt. Es verändern sich dabei alle Elemente, die nicht mit „...“ gekennzeichnet sind. Alle anderen werden mit dem jeweiligen Faktor multipliziert.

1	0	0	0
0	c	-s	0
0	s	s	0
0	0	0	1

rotateY(unterhalb)

c	0	s	0
0	1	0	0
-s	0	c	0
0	0	0	1

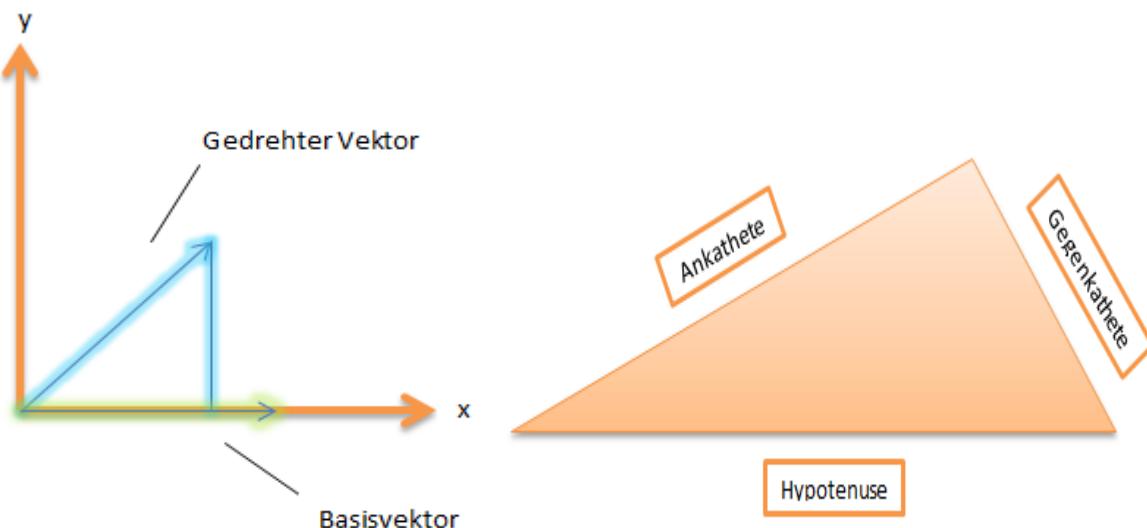
rotateZ(unterhalb)

c	-s	0	0
s	c	0	0
0	0	1	0
0	0	0	1

Die nächste Matrix zeigt die Veränderungen an der Matrix anhand der Rotation um die x-Achse mit der Funktion **rotateX()**. Dabei geht man von der Einheitsmatrix aus und lässt die erste Spalten stehen, da die x-Achse sich nicht verändert, sondern nur um sich selbst gegen den Uhrzeigersinn dreht. Daher verändern sich nur die y- und z-Koordinate. Das neue x berechnet sich aus $x \cos(\Phi) - y \sin(\Phi)$ und y aus $x \sin(\Phi) + y \cos(\Phi)$. Die Berechnungen für die Rotation der anderen beiden Achsen funktionieren auch über die Winkelfunktionen, wie die beiden anderen Tabellen zeigen. $c=\cos(\Phi)$ $s=\sin(\Phi)$.

Achtung: Anschließend wird die alte Matrix mit der neuen durch die Funktion **apply()** multipliziert. Die neuen Elemente entstehen, indem die Änderungen mit den alten Elementen multipliziert und ihnen die Summe aller Werte ihrer Spalte zugewiesen werden.

Man benötigt dafür die sogenannten Basisvektoren, mit denen sich ein ganzer n-dimensional Raum darstellen lässt. Da eine Koordinate gedreht wird, wird der Vektor für den 2-dimensionalen Raum benötigt, den man in der Identitätsmatrix ablesen kann: $\begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}$. Dann zeichnet man die beiden Spaltenvektoren in ein Koordinatensystem ein und zeichnet jeweils einen Vektor ein, der vom Basisvektor aus gedreht wurde. Von der Spitze des neuen Vektors aus, wird so eine Linie gezogen, dass ein rechtwinkeliges Dreieck entsteht. Mit Hilfe des Wissen $\sin = \frac{\text{Gegenkathete des Winkels}}{\text{Hypotenuse}}$ und $\cos = \frac{\text{Ankathete des Winkels}}{\text{Hypotenuse}}$ kann die neue x- und y-Koordinate berechnet werden.



Eine Translation mit **translate()** ist eine affine Transformation, da gilt $\vec{x}' = A\vec{x} + \vec{b}$. Sie wird in der vierten Spalte durchgeführt, der Vektor \vec{b} setzt sich aus den linearen Transformationen zusammen, die vorher erklärt wurden. Die Spalten der jeweiligen Reihen werden mit der übergebenen Koordinate tx, ty, oder tz multipliziert. Es wird die Summe der Ergebnisse gebildet und als vierter Element eingesetzt. Die Matrix sieht dann wie in der Tabelle aus.

Translate

...	$tx * m_{00} + ty * m_{01} + tz * m_{02}$
...	$tx * m_{10} + ty * m_{11} + tz * m_{12}$
...	$tx * m_{20} + ty * m_{21} + tz * m_{22}$
...	$tx * m_{30} + ty * m_{31} + tz * m_{32}$

Zur Erinnerung: m_{00} bis m_{33} sind die Einträge in der Matrix. Wenn ein Element um sich selbst gedreht werden soll, muss es vorher mit **translate()** verschoben werden, damit es nicht um den Koordinatenursprung gedreht wird. **translate(0,0)** hat keine Wirkung, stattdessen kann man zum Beispiel die Koordinaten der zu drehenden Form an **translate()** übergeben und sie dann an der Position 0,0 zeichnen.

shearX

1	t	0	0
0	1	0	0
0	0	1	0
0	0	0	1

Bei der Scherung verändert sich der Winkel zwischen den Achsen der Koordinaten. Technisch gesehen, werden die Punkte nur mit einer Rotationsmatrix multipliziert. Scherung kann im 2- und 3-dimensionalen Raum stattfinden, Processing unterstützt jedoch nur 2D mit den Funktionen **shearX()** und **shearY()**. Ihnen wird ein Winkel als Parameter übergeben.

shearY

1	0	0	0
t	1	0	0
0	0	1	0
0	0	0	1

Alle bis jetzt erwähnten Transformationen sind lineare Transformationen. Das heißt es gilt: $\vec{x}' = A\vec{x}$. Um noch einmal die Charakteristika dieser Transformationen zusammenzufassen, sind in dieser Liste die gemeinsamen Eigenschaften festgehalten worden, wobei die Skalierung ausgeschlossen wurde:

- Es wird immer vom Normalvektor ausgegangen
- Es werden die ersten 3 Spalten modifiziert
- Die 4. Spalte bleibt unangetastet.
- Die 4. Reihe bleibt unberührt.

Obwohl die Skalierung ein Spezialfall ist, gehört sie in diese Gruppe.

Dies waren die Hauptfunktionen, die zuständig für die Transformationen sind. Zusätzlich zu **apply()** gibt es **preApply()**, mit der man auf der linken Seite eine andere Matrix anwenden kann. Die Methode **mult()** erlaubt es einen Vektor mit der jetzigen Matrix zu multiplizieren, dabei werden die einzelnen Reihenelemente mit der Koordinate x-z vom Quellvektor multi-

pliziert (bis auf die vierte Spalte, da es nur drei Koordinaten gibt) und als Summe in die einzelnen Koordinaten des Zielvektors geschrieben. Diese Methode gibt es in allen möglichen Kombinationen, zum Beispiel, dass nur mit einer bestimmten Reihe multipliziert wird.

PGraphics

PGraphics.java ist die Hauptklasse, die die Grundfunktionen der Processing Schnittstelle implementiert, im Vergleich zur Klasse Applet, die nur die Funktionen aufruft. Wenn es nötig ist, dass in einen off-screen Grafikpuffer gezeichnet wird, kann diese Klasse verwendet werden. Ein neues PGraphics Objekt wird mit **createGraphics()** konstruiert. Die Methoden **beginDraw()** und **endDraw()** sind nötig, um den Puffer zu erstellen und ihn wieder zu zerstören. Die Methoden in dieser Klasse sind sehr vielzählig, allerdings sind die meisten schon aus der Klasse PApplet bekannt. Daher werden nur mehr die Funktionsweisen vorgestellt.

Um eine Unterklass zu erstellen und das PGraphics Objekt zuzuweisen, müssen keine Parameter an den Konstruktor der Unterklass übergeben werden, da ein paar Funktionen des „Hosting“ Applets aufgerufen werden um die Attribute zu spezifizieren. **setParent()** wird aufgerufen, um das zuständige Applet auszuwählen. **setPrimary()** wird mit einem Boolean aufgerufen und kann festlegen, dass dieses PGraphics die Hauptzeichenoberfläche ist, die vom Sketch verwendet wird. **setPath()** wird aufgerufen, wenn der Renderer einen Dateinamen oder einen Ausgabepfad benötigt, wie zum Beispiel der PDF oder DXF Renderer. Am Schluss wird **setSize()** aufgerufen, da es an dieser Stelle sicher ist, die Initialisierung zu beenden. PGraphics enthält viele Methoden, die mit sogenannten Sichtbarkeitsmodifizierern versehen sind, die bereits vor der praktischen Arbeit erwähnt wurden. Funktionen, die mit **protected** versehen sind, werden oft von verschiedenen Renderern benötigt, sind jedoch nicht **public**, da sie nicht von Usern von Processing erreicht werden sollen. Zum Beispiel sollten *textModeCheck()* oder *vertexTexture()* nie direkt vom User aufgerufen werden.

Ein weiteres wichtiges Thema sind Warnungen und Ausnahmen, die von diversen Methoden ausgelöst werden. Methoden, die nicht verfügbar sind, zeigen generell eine Warnung an, damit sie durch ihr fehlendes Vorhandensein nicht eine andere Ausnahme verursachen. Beispielsweise gibt die Methode *getMatrix()* null zurück, wenn sie nicht vorhanden ist und eine Ausnahme wird hervorgerufen, dass die Funktion nicht verfügbar ist, anstatt, dass gewartet wird, dass eine NullPointerException erfolgt, wenn der Sketch versucht diese Methode zu nutzen. Seit der Version 1.49 werden die Warnungen nur einmal angezeigt und Ausnahmen wurden durch Warnungen ersetzt, sofern es möglich war.

Eine sehr ähnliche Klasse zu PGraphics ist PImage, da PGraphics eine Unterklasse von PImage ist, damit es in einer ähnlichen Art und Weise gezeichnet und manipuliert werden kann. Daher wurden viele Methoden vererbt, viele sind jedoch unter Umständen nicht verfügbar: Zum Beispiel *resize()* ist nicht implementiert, genauso wie *mask()*, wobei es von der Situation abhängt. Daher stellt sich nun die Frage: Was befindet sich alles in PGraphics und was nicht?

Generell ist zu sagen, dass aufgrund der Vorteile von Unterklassen so viel wie möglich in PGraphics platziert wurde. Interpolationscode für Bézierkurven und Implementationen von *strokeCap()* Methoden werden hier verarbeitet. Features, die abhängig vom Renderer sind, sind in der eigenen Unterkasse untergebracht. Als Beispiel kann man hier die Matrixoperationen nennen, die vom Renderer abhängig sind. Java 2D benutzt eigene affine Transformationen, P2D benutzt PMatrix2D, sowie PMatrix3D. Eine gute OpenGL Implementation soll künftig erfolgen, also in einer Version, die höher als 1.5 ist, die alle Matrixoperationen mit der Grafikkarte durchführen wird. Die Beleuchtung fällt in die gleiche Kategorie, wobei die grundlegenden Materialeinstellungen wie direktes Licht, Umgebungslicht etc. weiterhin von PGraphics verarbeitet werden, da die Standardlogik von *colorMode()* benutzt wird. Unterklassen sollten dann Methoden wie *emmissiveFromCalc()* überschreiben, bei der eine gültige Farbe zuvor definiert wurde, die in verschiedenen Funktionen mit der Endung „*Calc*“ verwendet werden kann.

Nun zur eigentlichen Klasse. Wie bereits erwähnt, wird die Klasse PImage erweitert, sowie einige Konstanten implementiert. Es werden sehr viele Variablen für die unterschiedlichsten Zwecke verwendet. Allgemeine, sowie für Berechnungen nützliche Variablen sind:

protected int width1	Breite der Grafik – 1
protected int height1	Höhe der Grafik - 1
public int pixelCount	Anzahl der Pixel (Höhe * Breite)
public boolean smooth = false	Antialiasing
protected boolean settingsInitiated	Wert, ob <i>defaults()</i> das 1.Mal aufgerufen wurde
protected PGraphics raw	Objekt für die Funktion <i>beginRaw()/endRaw()</i>
protected String path	Pfad zur einer Datei, für den Render
protected boolean primarySurface	Gibt an, ob es sich um die Hauptzeichenfläche des Sketches handelt
protected boolean[] hints	Beinhaltet die Rendering Hints

Die weiteren Variablen behandeln verschiedene Themen:

public int colorMode	Aktuelle Farbmodus
public float colorModeX	Maximaler Rot/Farbtön
public float colorModeY	Maximaler Grünton/Sättigungswert
public float colorModeZ	Maximaler Blauton/Helligkeitswert
public float colorModeA	Maximaler Alphawert
boolean colorModeScale	Gib an, ob die Farben im Bereich 0 bis 1 verwendet werden
boolean colorModeDefault	Gibt an ob der Farbmodus RGB,255 ist

Einfärbung

public boolean tint	Gibt an, ob <i>tint()</i> aktiviert ist
public int tintColor	verwendete Farbe
protected boolean tintAlpha	verwendete Alphawert
protected float tintR, tintG, tintB, tintA	Einzelne Farbkomponenten
protected int tintRi, tintGi, tintBi, tintAi	Hilfsvariable

Füllfarbe

public boolean fill;	Gibt an, ob <i>fill()</i> aktiviert ist
public int fillColor = 0xfffffff	verwendete Farbe (Standard: weiß)
protected boolean fillAlpha	verwendete Alphawert
protected float fillR, fillG, fillB, fillA	einzelnen Farbkomponenten
protected int fillRi, fillGi, fillBi, fillAi	Hilfsvariable

Rahmenfarbe

public boolean stroke;	Gibt an, ob <i>stroke()</i> verwendet wird.
public int strokeColor = 0xff000000	verwendete Farbe (Standard: weiß)
protected boolean strokeAlpha	verwendete Alphawert
protected float strokeR, strokeG, strokeB, strokeA	einzelne Farbkomponenten
protected int strokeRi, strokeGi, strokeBi, strokeAi	Hilfsvariable
protected final float DEFAULT_STROKE_WEIGHT = 1	Konstante für Standarddicke
protected final int DEFAULT_STROKE_JOIN = MITER	Konstante für die Standard Kantenverbindungsart

protected final int DEFAULT_STROKE_CAP = ROUND	Konstante für die Standardlinienart
public float strokeWeight = DEFAULT_STROKE_WEIGHT	verwendete Liniendicke
public int strokeJoin = DEFAULT_STROKE_JOIN	verwendete Kantenverbindungsart
public int strokeCap = DEFAULT_STROKE_CAP	verwendete Kantenart

Positionierung von Formen

public int rectMode	Viereckmodus
public int ellipseMode	Ellipsenmodus
public int shapeMode	Formenmodus
public int imageMode	Grafikmodus

Text und Schrift

public PFont textField	verwendete Schriftart
public int textAlign = LEFT	Textausrichtung auf der x-Achse
public int textAlignY = BASELINE	Textausrichtung auf der y-Achse
public int textMode = MODEL	Textmodus
public float textSize	Schriftgröße
public float textLeading	Zeilenzwischenraum der Zeichen

Materialien

public float ambientR, ambientG, ambientB
public float emissiveR, emissiveG, emissiveB
public float specularR, specularG, specularB
public float shininess

Stapel der Styleeigenschaften

final int STYLE_STACK_DEPTH = 64	Standardstapeltiefe
PStyle[] styleStack =new PStyle[STYLE_STACK_DEPTH]	Stapel aus PStyle Objekten
int styleStackDepth	Stapeltiefe

Hintergrund und Matrix

public int backgroundColor = 0xffCCCCCC	Hintergrundfarbe
protected boolean backgroundAlpha	verwendete Alphawert
protected float backgrounders, backgroundG, backgroundB, backgroundA	Farbkomponenten des Hintergrunds
static final int MATRIX_STACK_DEPTH	Matrixstapeltiefe

Plmage

public Image image	für diesen Renderer assoziiertes Image Objekt
protected float calcR, calcG, calcB, calcA	Hilfsvariable
protected int calcRi, calcGi, calcBi, calcAi	Hilfsvariable
protected int calcColor	Hilfsvariable
protected boolean calcAlpha	Hilfsvariable
int cacheHsbKey	letzte zu HSB konvertierte RGB-Wert
float[] cacheHsbValue = new float[3]	Enthält das Ergebnis der letzten Konvertierung

Shape

protected int shape	Art der Form
public static final int DEFAULT_VERTICES = 512	Konstante, mit der Standardanzahl an Punkten(Vertex/Vertices)
protected float vertices[][] = new float[DEFAULT_VERTICES][VERTEX_FIELD_COUNT]	
protected int vertexCount	Anzahl der Punkte

Bézierkurve und Kurve

protected boolean bezierInitiated = false	Gibt an, ob Bézier initiiert wurde
public int bezierDetail = 20	Detailierungsgrad der Bézierkurve
protected PMatrix3D bezierBasisMatrix	Hilfsmatrix für Bézier und Kurve
protected PMatrix3D bezierDrawMatrix	Hilfsmatrix
protected boolean curveInitiated = false	Gibt an, ob die Kurve initiiert wurde
protected int curveDetail = 20	Detailierungsgrad der Kurve
public float curveTightness = 0	Enge der Kurve

protected PMatrix3D curveBasisMatrix	Catmull-rom Basismatrix
protected PMatrix3D curveDrawMatrix	Catmull-rom Basismatrix
protected PMatrix3D bezierBasisInverse	Hilfsmatrix
protected PMatrix3D curveToBezierMatrix	Hilfsmatrix

Sinus/Kosinus Referenztabelle

final protected float sinLUT[]	Sinuseinträge
final protected float cosLUT[]	Kosinuseinträge
final protected float SINCOS_PRECISION = 0.5f	Allgemeine Präzision der Werte
final protected int SINCOS_LENGTH	Anzahl der Werte

Interne Textpuffer

protected float textX, textY, textZ;	Letzte Textposition
protected char[] textBuffer = new char[8 * 1024];	Textbuffer
protected int textBreakCount	Anzahl der Textumbrüche
protected int[] textBreakStart	Array der Startpositionen der Umbrüche
protected int[] textBreakStop	Array der Endpositionen der Umbrüche

Diverses

protected final int NORMAL_MODE_AUTO = 0	Konstante für Modus AUTO
protected final int NORMAL_MODE_SHAPE = 1	Konstante für Modus SHAPE
protected final int NORMAL_MODE_VERTEX = 2	Konstante für Modus VERTEX
protected int normalMode	Standardmodus
protected boolean autoNormal	Gibt an, ob der Standardmodus aktiviert ist
public float normalX, normalY, normalZ	Normalvektoren

Texturen und Sphären

public int textureMode	Texturmodus
public float textureU	Breite der Textur
public float textureV	Höhe der Textur
public PImage textureImage	Grafik, die als Textur verwendet wird

float sphereX[], sphereY[], sphereZ[]	Position der Kugel
public int sphereDetailU = 0	Detailierungsgrad in Richtung x-Achse
public int sphereDetailV = 0	Detailierungsgrad in Richtung y-Achse

Da die verwendeten Variablen nun beschrieben wurden, kann nun zur eigentlichen Klasse übergegangen werden. Der Konstruktor ist leer und es ist bekannt, dass die Konfigurationen mit `setParent()`, `setPrimary()`, `setPath()` und `setSize()` durchgeführt werden. Zu letzterer Funktion ist zu sagen, dass zusätzlich zu den neuen Zuweisungen ein neuer Speicher mit **allocate()** reserviert wird und die neuen Einstellungen mit **reapplySettings()** angewandt werden. `allocate()` muss vom jeweiligen Renderer selbst implementiert werden. Die Methode **dispose()**, die aufgerufen wird, wenn der Sketch beendet und ein Renderer ausgewählt wird, wurde leer gelassen. Weiters werden `beginDraw()` und `endDraw()` ignoriert, sowie die Methode `flush()`, die aufgerufen wird, wenn mit dem P3D Renderer verschiedene Objekte geschrieben werden. Die bereits erwähnte Funktion `checkSettings()` ruft **defaultSettings()** auf, wenn die Einstellungen nicht initiiert wurden. Nun zu den Standardsteinstellungen:

```

01 noSmooth();
02 colorMode(RGB, 255);
03 fill(255);
04 stroke(0);
05 strokeWeight(DEFAULT_STROKE_WEIGHT);
06 strokeJoin(DEFAULT_STROKE_JOIN);
07 strokeCap(DEFAULT_STROKE_CAP);
08 shape = 0;
09 rectMode(CORNER);
10 ellipseMode(DIAMETER);
11 autoNormal = true;
12 textStyle = null;
13 textSize = 12;
14 textLeading = 14;
15 textAlign = LEFT;
16 textMode = MODEL;
17 if (primarySurface) {
18     background(backgroudColor);
19 }
20 settingsInitiated = true;
21

```

`reapplySettings()` wird von manchen Methoden wie `textFont()` aufgerufen, da sie den Grafikkontext beeinflussen oder Parameter vom Kontext benötigen. Die Funktion wird nur von `allocate()` aufgerufen, die nach `size()` innerhalb der Funktion `beginDraw()` vorkommt. `ReapplySettings()` wendet nur diejenigen Einstellungen an, die aktiviert sind.

Nun, da alle Einstellungen gesetzt sind, können Formen gezeichnet werden. Wenn `beginShape()` ohne Argumente aufgerufen wird, nimmt Processing an, dass es sich um ein Polygon handelt. Die Art der Form wird einfach in einer Variable festgehalten. Die erste Hilfsfunktion, der man begegnet, ist `edge()`, mit der festgelegt wird, ob der nächste Punkt in der Form ein Teil einer Ecke ist. In OpenGL nennt sich dieser Befehl `glEdgeFlag()`.

Um die Form zu zeichnen, ist das Setzen des Normalvektors mit `normal()` für jeden Punkt wichtig. Der Vektor wird dabei als Float übergeben. Andere Funktionen verändern nur den Wert von einzelnen Variablen, die zum Beispiel in If-Anweisungen benötigt werden. Zu diesen Befehlen gehört `textureMode()`, `texture()` und `noTexture()`.

Mehr Technik befindet sich dagegen in der geschützten Funktion `vertexCheck()`, in der die Punkte der Formen in ein anderes Array kopiert werden. Davor allerdings muss erst einmal `vertex()` aufgerufen werden. Zusätzlich zur Position des neuen Punktes werden auch andere Informationen wie die Linien- und Füllfarbe und Materialinformationen in das Array gespeichert, sowie überprüft ob der vorherige Punkt mit dem Neuen identisch ist. Wenn die Funktion aktiviert ist, wird auch gleich der Normalvektor automatisch berechnet.

Wenn die Informationen über den neuen Punkt bereits vorhanden sind, kann auch `vertexFields()` ein Array übergeben werden, das in das entsprechende Array kopiert wird.

Da die Form definiert wurde, muss nur `endShape()` mit dem optionalen Argument CLOSE geschrieben werden, um eventuell den Endpunkt mit dem Anfangspunkt zu verbinden. Etwas komplizierter dagegen sind Bézierpunkte, die über die Matrizenrechnung berechnet werden. (*Die folgende Beschreibung ist dem Wikipediartikel „Bézierskurve“ entnommen:*)

Bézierskurven werden anhand einer Formel berechnet, bei Processing erfolgt dies über Matrizenoperationen. Sie wurde Anfang der 1960er Jahre von Pierre Bézier bei Renault und Paul de Casteljau bei Citroën für computerunterstützte Konstruktion(CAD) gleichzeitig und unabhängig voneinander entwickelt. Es ist in der Computergrafik eine sehr beliebte Form um Kurven zu zeichnen aufgrund ihrer grafischen Eleganz und der einfachen Berechnung. Grundsätzlich werden 4 Kontrollpunkte verwendet, um die Kurve zu beschreiben. Benennt man diese P_0 bis P_3 und definiert einen Parameter t für die Funktion, also $Q: t \rightarrow Q(t) = \begin{pmatrix} z(t) \\ y(t) \end{pmatrix}$,

$$\begin{aligned} Q(t) = & (1 - t)^3 * P_0 \\ & + 3t * (1 - t)^2 * P_1 \\ & + 3t^2 * (1 - t) * P_2 \\ & + t^3 * P_3 \end{aligned}$$

sieht die Funktion, wie auf der linken Seite aus. Die Funktion kann, wie bereits erwähnt auch mit Matrizen berechnet werden. Die Formel dazu sieht so aus: $Q(t) = G_B * M_B * T$. M_B und T sind Matrizen, in

denen die Informationen über die Kurve gespeichert sind. G_B stellt die Eigenschaften einer bestimmten Bézierkurve dar und wird daher Bézier-Geometriematrix genannt. Die Elemente der Matrix entsprechen den Kontrollpunkten.

M_B stellt die allgemeinen Eigenschaften einer jeden Bézierkurve da und wird Bézier-Basismatrix genannt. Sie ist auf der linken Seite abgebildet.

-1	3	-3	1
3	-6	3	0
-3	3	0	0
1	0	0	0

In Processing können auch quadratische Bézierkurven berechnet werden, allerdings werden sie in dieser Arbeit nicht näher erläutert. Normale Kurven werden mit **curveVertex()** Aufrufen gestaltet. Bevor die Kurvenpunkte in ein Array gespeichert werden, wird die Methode **curveVertexCheck()** aufgerufen, um Fehler zu vermeiden. Danach wird die Kurve mit **curveInitCheck()** ein weiteres Mal überprüft. Wenn die Kurve eine Catmull-Rom Kurve ist, wird das neue Vertex mit **curveVertexSegment()** hinzugefügt. Primitive Formen wie Linien und Ellipsen werden auf dieselbe Weise gezeichnet. Je nach Anzahl der Punkte in der Form wird entsprechend oft **vertex()** aufgerufen. Es wird in diesen Funktionen auch der Modus der Form beachtet. Auch die 3-dimensionale Box wird mit Hilfe von 24 Punkten gezeichnet. Die Anzahl der Punkte bei der Kugel ist abhängig vom Detaillierungsgrad.

Die primitive Variante der Bézierkurve wurde bereits erläutert. Die Tangente, die in **bezierTangent()** berechnet wird, erwartet sich die Koordinaten der Kontrollpunkte sowie den Abschnitt. Wenn die übergebenen Werte a, b, c, d und t genannt werden, wir sie wie in der

$$\begin{aligned} \text{Tangente} = & 3t^2 * (-a+3 * \\ & b - 2*c + d) + \\ & 6*t * (a - 2*b + c) + \\ & 3 * (-a + b) \end{aligned}$$

Grafik berechnet. **curvePoint()**, **curveTangent()** und **curveInit()** werden wieder über Matrizen berechnet, wobei die letzte der genannten Funktionen für die Catmull-Romkurve benötigt wird. **curveDetail()** und **curveTightness()** verändern nur den Wert einer Variable. Sowohl Bézier als auch Catmull-Rom verwenden eine zusätzliche Methode **splineForward()**, bei der eine spezielle Matrix aufgestellt wird, die das Kurven rendern beschleunigt. Sie basiert auf einer speziellen Anzahl an Kurvensegmenten und fügt inkrementell einen jeden Punkt eines Segments hinzu, anstatt mathematisch eine Gleichung dritten Grades zu lösen.

Wenn man sich die Funktion ansieht, die PImage Objekte zeichnen, erkennt man, dass sie auch auf die gleiche Art gezeichnet werden. Die Funktion **image()** ruft **imageImpl()** auf, in der mit **beginShape(QUADS)** ein neues Quadrat erstellt wird. Das Bild wird in das Quadrat eingefügt, indem vor den Punkten der Befehl **texture()** mit dem PImage Objekt aufgerufen wird.

Nur Vektorgrafiken, die mit `shape()` angezeigt werden, funktionieren anders. Die Klasse `PShape` stellt dazu die Methode `draw()` bereit. Das nächste anstehende Thema wäre „Text und Schrift“. Da es aber schon ausführlich behandelt wurde, gibt es nur mehr einen kleinen Nachtrag. Die Methoden, die zuständig sind, besitzen meistens eine zweite Methode, die mit „`Impl`“ endet. Die Hauptmethode konfiguriert die Einstellungen und die Hauptarbeit erfolgt in der Implementation.

Viele Methoden, die Transformationen und Projektionen durchführen oder die Kamera verändern werden zum Beispiel in `PMatrix2D` ausgeführt. Daher sind diese Methoden in `PGraphics` zwar implementiert, zeigen bei einem Aufruf aber nur eine Warnung an, dass sie nicht implementiert sind. Die Implementierung erfolgt in anderen Klassen, die die Funktionen überschreiben.

Alle den Stil verändernden Methoden sind allerdings implementiert, wie zum Beispiel `pushStyle()` und `popStyle()`, die das `PStyle` Array verändern. Angewendet werden die neuen Stileinstellungen mit der Funktion `style()`, an die ein `PStyle` Objekt übergeben wird. Darin werden Funktionen wie `fill()` bis hin zu `textLeading()` mit Argumenten gefüllt. Mit `getStyle()` werden die Einstellungen in ein `PStyle` Objekt gespeichert und zurückgegeben. Viele in dieser Kategorie befindliche Methoden verändern nur den Wert bestimmter Variablen, wobei zusätzlich eine Hilfsfunktionen mit der Endung „`fromCalc`“ zum Beispiel `tintFromCalc()` anfallende Berechnungen durchführen. Genauso funktionieren die Methoden, die das Material modifizieren, die das Licht betreffenden Funktionen sind hier nicht implementiert. Auch das Setzen des Hintergrunds erfolgt gleich und alle bereits erwähnten Funktionen, die sich um das `color` Objekt drehen sind hier implementiert.

Abgeschlossen wird die Klasse `PGraphics` durch die Warnungen und Ausnahmen. Die folgende Tabelle zeigt alle Funktionen, die ein String als Parameter erwarten.

Methode	Funktion
<code>showWarning()</code>	Zeigt einen Renderfehler an, wenn er nicht bereits gezeigt wurde.
<code>showDepthWarning()</code>	Warnt, dass die Methode nur in 3D vorhanden ist.
<code>showDepthWarningXYZ()</code>	Warnt, dass eine Methode die 3 Dimensionsparameter aufnehmen kann, beim aktuellen Renderer nur 2 Dimensionen erwartet.
<code>showMethodWarning()</code>	Warnt, dass die Methode nicht verfügbar ist.

showVariationWarning()	Warnt, dass eine bestimmte Variante dieser Methode nicht verfügbar ist.
showMissingWarning()	Warnt, dass eine Methode unter Umständen nicht implementiert ist.
showException()	Zeigt einen Renderer spezifischen Fehler anhand einer RuntimeException an.
defaultFontOrDeath()	Stoppt das Programm, wenn trotz des Versuchs die Standardschriftart zu erstellen ein Fehler auftaucht.

PGraphics2D

PGraphics2D ist die Unterklasse für PGraphics, die eine Implementierung der Grafikfunktionen mit Hilfe von Java2D vornimmt. Der Standard Renderer P2D benutzt auch Java2D, ist jedoch nicht für Pixelmanipulation geeignet. Wenn das Java Graphics2D aus einem besonderen Anlass benötigt wird, kann es durch *PGraphics2D g2= ((PGraphicsJava2D)g)* P2D benutzt werden. Dadurch kann man Java2D betreffende Aktionen direkt durchführen, allerdings ist diese Variante für keine Art von Formen unterstützt. Das heißt, dass man es benutzen kann, sich aber nicht wundern darf, wenn das Programm abstürzt und mit einer Fehlermeldung endet.

Aufgrund der vielen direkten Aufrufe von Java Funktionen werden recht wenige Variablen benötigt:

Variable	Beschreibung
Graphics2D g2	Hauptobjekt, das für alles zuständig ist.
BufferedImage offscreen	Buffer, in den im Hintergrund gezeichnet werden kann.
GeneralPath gpath	Geometrischer Pfad, der gerade Linien, quadratische und kubische Bézierkurven speichern kann.
boolean breakShape	Gibt an, ob die Form am nächsten Punkt geknickt wird.
float[] curveCoordX	x-Koordinaten für interne Kurvenberechnung.
float[] curveCoordY	y-Koordinaten für interne Kurvenberechnung.
float[] curveDrawX	x-Koordinaten für interne Kurvenberechnung.
float[] curveDrawY	y-Koordinaten für interne Kurvenberechnung.
int transformcount	Anzahl der Transformationen (Zählvariable)
AffineTransform transforms- tack[]	Array, das alle affinen Transformationen enthält.

Line2D.Float line	Objekt, mit dem eine neue Linie instanziert wird.
Ellipse2D.Float ellipse	Objekt, mit dem eine neue Ellipse instanziert wird.
Rectangle2D.Float rect	Objekt, mit dem ein neues Rechteck instanziert wird.
Arc2D.Float arc	Objekt, mit dem eine neue Linie instanziert wird.
Color tintColorObject	Farbe für die Einfärbung von Objekten.
Color fillColorObject	Farbe für das Füllen von Objekten.
boolean fillGradient	Gibt an, ob die Füllung aus einem Farbverlauf besteht.
paint fillGradientObject	Beinhaltet einen Farbverlauf.
Color strokeColorObject	Farbe des Rahmens von Formen.
boolean strokeGradient	Gibt an, ob der Rahmen aus einem Farbverlauf besteht.
Paint strokeGradientObject	Beinhaltet einen Farbverlauf.

Viele Funktionen wurden bereits in PGraphics implementiert und es werden in PGraphics2D nur mehr einige Funktionen erweitert. Zum Nennen ist zum Beispiel `vertex()`, in der einzelne Formen wie POINTS, TRIANGLES, QUADS implementiert, sowie einige Warnungen hinzugefügt wurden. Auch die Funktionen, die Kurven und Bögen darstellen, wurden leicht verändert. Linien, Dreiecke und Vierecke werden in einem geometrischen Pfad gespeichert und dann gezeichnet. Vektorengrafiken werden auch über den bereits genannten Pfad gerendert. Eine wichtige Funktion, deren Programmblöck erstmals nicht leer ist, ist der von `smooth()` und `noSmooth()`, in denen Rendering Hints gesetzt werden mit Hilfe von `setRenderingHint()`. Die Hints sind Konstanten in der Klasse `RenderingHints`. Dabei wird an die Funktion ein Schlüssel mit einem passenden Wert mitgegeben, der Klasse `Object` ist.

`smooth()`

KEY_ANTIALIASING	VALUE_ANTIALIAS_ON
Einstellung, dass anti-aliasing verwendet wird	
KEY_INTERPOLATION	VALUE_INTERPOLATION_BICUBIC
Einstellung, dass Interpolation verwendet wird	

`noSmooth()`

KEY_ANTIALIASING	VALUE_ANTIALIAS_OFF
Einstellung, dass anti-aliasing nicht verwendet wird	
KEY_INTERPOLATION	VALUE_INTERPOLATION_NEAREST_NEIGHBOR
Einstellung, dass Interpolation nur zwischen Nachbarobjekten stattfindet.	

Um das Zeichnen von Bildern zu beschleunigen, wurde die Klasse *ImageCache* geschrieben, die das zuletzt angezeigte Bild, sowie dessen Färbung enthält. Wenn eine neue Grafik gezeichnet werden soll, wird zuerst über überprüft, ob das Bild im Cache die gleiche Färbung hat. Sollte dies der Fall sein kann es direkt mit *drawImage()* gezeichnet werden. Ansonsten wird vorher das Bild umgefärbt. Der Aufbau der Klasse ist nicht so entscheidend, entscheidend ist nur das Wissen, dass sie existiert.

Weiters wird in PGraphics2D der Textaszendent und -deszendent, sowie die Textbreite berechnet. Hilfreich ist dabei die Java Klasse *FontMetrics*. Den Text betreffend ist auch die Funktion *textLineImpl()*, die den Text mit Hilfe der Funktion *drawChars()* vom Graphics2D schreibt.

Abgesehen von einzelnen Matrix Befehlen wie *pushMatrix()*, *popMatrix()* und *applyMatrix()* und einigen anderen werden nur „depth warnings“ angezeigt, die auch bei den Kamera- und Projektionsfunktionen, sowie bei den Methoden zur SCREEN und MODEL Transformation zum Einsatz kommen. Ansonsten werden weitere Warnungen angezeigt und ein paar kleiner Funktionen verändert, die jedoch nicht nennenswert sind und dem Quelltext entnommen werden können.

Die einzigen Implementierungen, die hier noch stattfinden, sind Methoden, die Pixelmanipulationen durchführen oder Pixel auslesen. Benutzt wird immer die Java Klasse *WritableRaster*, mit der Pixel geschrieben und gelesen werden können. *loadPixels()* und *get()* rufen darunter *getDataElements()*, *updatePixels()* und *get()* *getDataElements()* auf. Eine sehr ähnliche Funktion, die gut zu den anderen passt, ist *copy()*, die sich aus den Methoden *drawImage()* und *copyArea()* von g2 (das Graphics2D Objekt) Nutzen zieht.

PShape

PShape ist ein Datentyp, mit dem Formen gespeichert werden können, in Version 1.5 nur in Form von Vektorgrafiken im SVG Format. Bevor eine Form benutzt werden kann, muss sie mit **loadShape()** geladen werden. Die **shape()** Funktion wird benutzt, um die Form ins Fenster zu zeichnen. Die Methode **loadShape()** unterstützt nur Dateien, die mit Inkscape und Adobe Illustrator erstellt wurden, da es keine vollständige SVG Implementation ist. Um das Wiederholen von bereits beschriebenen Funktionen zu vermeiden, gibt es wieder nur eine kleine Einführung in diese Klasse.

Gezeichnet werden die Formen mit **drawGeometry()**, an die ein PGraphics Objekt übergeben wird. Die Vektorgrafiken werden mit *beginShape()* eingeleitet, dann werden mit *vertex()* die einzelnen Punkte gezeichnet und am Schluss wird *endShape()* aufgerufen. Man erkennt, dass sie gleich wie andere komplexe Formen gezeichnet werden. Für Kurven existiert die Funktion **drawPath()**, die folgende Pfade im 2- und 3-dimensionalen Raum zeichnen kann: VERTEX, QUAD_BEZIER_VERTEX, BEZIER_VERTEX und CURVE_VERTEX. Da SVG auf XML basiert, gibt es einige typische Funktionen, mit denen eine SVG Datei ausgelesen werden kann:

Funktion	Aufgabe
PShape getParent()	Gibt das Elternelement zurück
int getChildCount()	Gibt die Anzahl der Kinderelemente zurück
PShape[] getChildren()	Gibt die Kinder zurück
PShape getChild()	Gibt ein Kind zurück
PShape findChild()	Sucht ein Kind in der untersten Ebene
void addChild()	Fügt ein Kind hinzu
void removeChild()	Entfernt ein Kind
void addName()	Fügt eine Form zur Lookup-Tabelle hinzu
int getChildIndex()	Gibt den Index eines Kindes zurück

Die übergebenen Parameter sind meistens ein PShape Objekt oder ein Integer. Zusätzlich gibt es einige Hilfsfunktionen wie *getFamily()*, *getPrimitive()*, *getParams()*, *getVertexCount()*... Außerdem wurden in dieser Klasse neue Transformationsfunktionen geschrieben, die zuerst eine neue Matrix für den 2- bzw. 3-dimensionalen Raum schaffen und dann die entsprechende Funktion aufrufen. Die Initialisierung der Matrix findet in **checkMatrix()** statt, an der die Dimension als Integer übergeben wird.

PShapeSVG

PShapeSVG ist der eigentliche Parser für SVG Dateien, diese Klasse analysiert also die Dateien und erstellt anhand der gefunden Objekte neue Objekte der Klasse PShape. Diese Klasse ist absichtlich nicht voll implementiert, da die Bibliothek so klein sein soll, dass sie in ein Applet in der Größenordnung von 25 bis 40 KB eingebettet werden kann. Eine bessere SVG Bibliothek für Import- und Exportfunktionen ist das Batik SVG Toolkit von der Apache Software Foundation, das unter <http://xmlgraphics.apache.org/batik/> im Web gefunden werden kann. Die Klasse basiert auf der Bibliothek Candy von Michael Chang, einem ehemaligen Studenten von UCLA Design | Media Arts. Sie wurde für Processing umgeschrieben und erweitert von Ben Fry.

PShapeSVG erweitert die Klasse PShape und enthält die folgenden Variablen:

Variable	Funktion
XML element	Zugriff auf XML Objekte
float opacity	Transparenz
float strokeOpacity	Transparenz des Rahmens
float fillOpacity	Transparenz der Füllung
Gradient strokeGradient	Farbverlauf des Rahmens
Paint strokeGradientPaint	Hilfsobjekt für einen Farbverlauf
String strokeName	ID eines anderen Objektes
Gradient fillGradient	Farbverlauf der Füllung
Paint fillGradientPaint	Hilfsobjekt für einen Farbverlauf
String fillName	ID eines anderen Objektes

An den Konstruktor muss ein XML Element übergeben werden, bei dem überprüft wird, ob es sich um eine SVG Datei handelt. Danach wird die Höhe und Breite ausgelesen. Es kann als erstes Argument auch ein Elternelement vom Typ PShapeSVG übergeben werden. Als zweites Element müssen die Einstellungen als XML Objekt angehängt werden, sowie ein Boolean, der bestimmt, ob die Kinder ausgelesen werden. Wenn das Elternelement gleich Null ist, werden Standardeinstellungen für den Stil vorgenommen, ansonsten werden die Einstellungen übernommen. Dann werden einige Informationen, wie die ID ausgelesen und die Methoden **parseColors()** und **parseChildren()** bei den Kinderelementen aufgerufen, sofern der Wert auf wahr gesetzt ist. **parseChildren()** ruft **getChildren()** auf und speichert somit die Kinder in einem Array. Dann werden die Kinder mit **addChild()** hinzugefügt.

parseChild() erwartet hingegen nur ein XML Element, bei dem der Name überprüft wird. Es gibt viele Varianten, wie der Name gedeutet werden kann:

Wert	Bedeutung
g	Enthält ein Objekt, ein neues PShapeSVG Objekt wird erzeugt
defs	Enthält Informationen über einen Farbverlauf; neues PShapeSVG Objekt
line	Enthält eine Linie; neues PShapeSVG Objekt
circle	Enthält einen Kreis; neues PShapeSVG Objekt
ellipse	Enthält eine Ellipse; neues PShapeSVG Objekt
rect	Enthält ein Viereck; neues PShapeSVG Objekt
polygon	Enthält ein Polygon; neues PShapeSVG Objekt
polyline	Enthält einen Linienzug (Polyline) ; neues PShapeSVG Objekt
path	Enthält einen Pfad; neues PShapeSVG Objekt

radialGradient	Enthält einen radialen Farbverlauf; neues RadialGradient Objekt
linearGradient	Enthält einen linearen Farbverlauf; neues LinearGradient Objekt
font	Enthält eine Schriftart; neues Font Objekt
metadata	Enthält Metadaten; werden ignoriert
text	Enthält Text; wird nicht unterstützt
filter	Enthält einen Filter; wird nicht unterstützt
mask	Enthält eine Maske; wird nicht unterstützt
pattern	Enthält ein Muster; wird nicht unterstützt
stop	Ende eines Farbverlaufs
sodipodi:namedview	Warnungen in Inkscape Dateien
null	Enthält wahrscheinlich nur Leerzeichen

Nachdem ein neues, entsprechendes Objekt instanziert wurde, wird bei manchen Übereinstimmungen das Objekt nicht zurückgegeben, sondern in der Variable *shape* gespeichert und danach eine Funktion aufgerufen, um die Daten weiter zu analysieren.

parseLine() holt sich die Koordinaten des Anfangs- und Endpunktes. **parseEllipse()** holt sich die Position, sowie den Radius. Wenn es eine Ellipse ist, gibt es für den Radius zwei Werte, bei einem Kreis nur einen Wert. **parseRect()** holt sich die Position, sowie die Breite und Höhe des Rechtecks. **parsePoly()** extrahiert zuerst die Punkte aus dem Attribut „points“ und speichert sie dann in zwei Arrays, eines für jede Koordinate, ab.

parsePath() geht auf eine ähnliche Weise vor, die Daten liegen hier jedoch nach dem Extrahieren in Form eines Char Arrays vor. Mit einer Schleife werden die einzelnen Zeichen durchlaufen und je nachdem, welche Form der Buchstabe symbolisiert, wird die Form separat von den anderen gezeichnet oder nicht. Die Zeichen bedeuten folgendes:

Buchstabe	Bedeutung
M	Linie zu einem absoluten Punkt.
m/L/I	Linie zu einem relativen Punkt.
H	Horizontale Linie zu einem absoluten Punkt.
h/V/v	Horizontale Linie zu einem relativen Punkt.
C	Kurve zu einem absoluten Punkt.
c	Kurve zu einem relativen Punkt.
S	Bézierkurve zu einem absoluten Punkt.
s	Bézierkurve zu einem relativen Punkt.
Q	Quadratische Kurve zu einem absoluten Punkt.
q	Quadratische Kurve zu einem relativen Punkt.

T	Quadratische Kurve zu einem absoluten Punkt.
t	Quadratische Kurve zu einem relativen Punkt.
Z/z	Ende der Form.

parsePathVertex() speichert die einzelnen Punkte in Arrays. **parsePathCode()** speichert einen speziellen Code der Punkte ab. Alle jetzt genannten Funktionen benutzen die zwei gerade vorgestellten Methoden: *parsePathCode()*, *parsePathMoveto()*, *parsePathLineto()*, *parsePathCurveto()* und *parsePathQuadto()*.

Es werden auch Transformationen mit *parseTransform()* bzw. *parseSingleTransform()* ausgewertet und in ein PMatrix2D Objekt gespeichert. Es gibt einige Transformationen, die an der Matrix durchgeführt werden können, indem die gefundenen Argumente eingesetzt werden:

String	Bedeutung
matrix	Es wurde eine Matrix übergeben.
translate	Eine Translation soll durchgeführt werden.
scale	Die Form soll skaliert werden.
rotate	Die Form soll gedreht werden.
skewX	Die Form soll in x-Richtung verzerrt werden.
skewY	Die Form soll in y-Richtung verzerrt werden.

Die Funktion **parseColors()** kann mehr Eigenschaften erkennen. Wenn Eigenschaften nicht direkt gefunden werden, sind sie meist im Attribut „style“, das ein String Array zurückgibt:

String	Bedeutung
fill	Der Füllwert
fill-opacity	Transparenz der Füllung
stroke	Rahmenfarbe
stroke-width	Breite des Rahmens
stroke-linecap	Art, wie Linien dargestellt werden
stroke-linejoin	Art, wie Linien verbunden werden
stroke-opacity	Transparenz des Rahmens
opacity	Transparenz

Wenn ein Wert gefunden wird, wird die entsprechende Funktion auch aufgerufen. Bei der Transparenz würde diese zum Beispiel *setOpacity()* heißen, wobei das übergebene Argument immer ein String ist.

Abgesehen von ein paar anderen analysierenden Methoden gibt es einige innere Klassen, die bereits ganz am Anfang in einer Tabelle vorkamen. LinearGradient erweitert Gradient und speichert nur die 2 Punkte, die den Farbverlauf beschreiben. Transformationen werden mit Hilfe eines Objektes der Klasse AffineTransform ausgeführt. RadialGradient ist fast eine Kopie der vorherigen Klasse mit dem Unterschied, dass statt einem zweiten Punkt der Radius festgelegt wird.

Die Hauptimplementierung erfolgt in LinearGradientPaint bzw. RadialGradientPaint, die beide die Klasse Paint erweitern. Berechnet werden die Farbverläufe durch **calcGradientPaint()**. Weitere innere Klassen sind Font, FontFace und FontGlyph, die der Klasse PFont sehr ähneln. Ein Unterschied ist, dass die Eigenschaften aus der SVG Datei gelesen werden.

Zuletzt kann das SVG Dokument mit **print()** ausgegeben werden, sowie ein spezielles Kindelement anhand der ID zurückgegeben werden mit **PShape getChild()**. Wenn die SVG Datei mit der Hand editiert wird, muss das ID Tag geändert werden. In Illustrator können diese Tags durch das Erweitern der Ebenenpalette erreicht werden.

Table

Table konvertiert csv (comma-separated values) in tsv (tab-separated values) Dateien. Die Klasse gehört zwar zu den core libraries, meiner Meinung nach ist sie aber nicht relevant, um die Funktionsweise von Processing zu verstehen. Außerdem können die Dateien genauso gut mit einer Desktopanwendung umgewandelt werden.

PImage

PImage speichert bekanntlich Grafiken und es ist möglich die einzelnen Pixel zu manipulieren und auszulesen. Die Klasse enthält bis auf einige bereits aus anderen Klassen geläufigen Funktionen hauptsächlich die Umsetzung der Filterfunktionen. Dieses Thema ist so komplex und lang, dass es den Rahmen dieser Arbeit sprengen würde. Daher habe ich auf diese Klasse verzichtet.

Damals und heute: Die Veränderungen in Processing

Wie jede Programmiersprache wurde Processing eines Tages initiiert und es gab die erste Version dieser Sprache. Dieses Entwicklungsstadium nennt sich Alpha. Die ersten Testversionen, die von jedem oder einem ausgewählten Benutzerkreis getestet werden, nennen sich Beta. Seit der ersten Version von Processing hat sich viel getan. Die größten Änderungen erfolgten in Version 1.0 (erschienen im November 2008). Zukünftige sowie alte Änderungen können im Processing Wiki unter <http://wiki.processing.org/w/Changes> nachgelesen werden.

Änderungen zwischen Alpha(Release 0069) und Beta (Release 0085)

- *loop()* wurde ersetzt durch *draw()*. Wenn die Anwendung nur einmal die Funktion *draw()* aufrufen soll, muss die Methode *noLoop()* innerhalb von *setup()* verwendet werden.
- *size()* sollte als aller Erstes in *setup()* aufgerufen werden, ansonsten werden andere Befehle, die vor diesem geschrieben wurde, ignoriert. Die Einstellungen für *fill()* oder *stroke()* zum Beispiel würden nach dem Aufruf von *size()* verschwinden.
- 2D und 3D Grafiken werden nicht mehr länger vermischt. Um 3D Grafiken nutzen zu können, muss statt *size(200,200)* die neue Art *size(200, 200, P3D)* aufgerufen werden.
- Man findet unter Umständen, dass der Code in einem Beta Release langsamer läuft, als in einem Alpha Release. Der neue Renderer kann bei manchen Dingen langsamer sein, bei anderen jedoch schneller. Wenn man schnelle Pixeloperationen durchführen möchte, sollte der P3D Renderer verwendet werden (bis P2D verfügbar ist). Um Linien und das Resultat der Funktionen *strokeWeight()* und *strokeCap()* in einer hohen Qualität zu bekommen, kann JAVA2D verwendet werden.
- Fortgeschrittene User, die in den vorherigen Release eigene BGraphics Objekte gemacht haben, sollten BGraphics durch PGraphics ersetzen und die Zeichenbefehle zwischen den Befehlen *beginDraw()* und *endDraw()* einschließen.
- Um den OpenGL Renderer zu benutzen, muss der Befehl *size(200,200)* auf *size(200, 200, OPENGL)* umgeändert werden Import Library->opengl ausgewählt werden. Die OpenGL Unterstützung ist noch nicht vollständig, aber sie erlaubt es viel größere Bildschirmgrößen zu verwenden und beschleunigte Grafiken zu erstellen, die von besseren Grafikkarten Gebrauch machen.
- Wenn das *pixel[]* Array des Hauptfensters oder einer Grafik benutzt wird, muss zuvor *loadPixels()* aufgerufen werden. Wenn man mit dem Bearbeiten fertig ist, rufe man *updatePixels()* auf. Das muss so sein, da der neue JAVA2D und OpenGL Renderer beide die Bilddaten aufbereiten müssen, bevor sie angezeigt werden, eine Operation, die viel Zeit kostet.

- Events von Bibliotheken werden nun anders behandelt, da mehr als ein serieller Port oder ein Videoeingabegerät mit Processing Beta benutzt werden können. Die Dokumentation von *movieEvent()*, *captureEvent()* und *serialEvent()* geben mehr Auskunft darüber.
- Die Schriftarten sollten nun eine exaktere Größe haben, wenn die Schriftgröße seltsam aussieht, kann man versuchen über den Dialog „Create Font“ eine neue Schriftart zu erstellen. Diese Funktion befindet sich im neuesten Release von Processing. Es wird wärmstens empfohlen dies beim Aktualisieren von Projekten von Alpha zu Beta durchzuführen.
- angleMode()* wurde entfernt, da Radianen für alle Funktionen benutzt werden. Wer Radianen nicht möchte, kann Grad in Radianen durch die Methode *radians()* umwandeln. Zum Beispiel: *sin(radians(90))*
- Anstatt vier Mal *bezierVertex(x, y)* aufzurufen, sollten einmal *vertex(x,y)*, gefolgt von *bezierVertex(cx1, cx1, cy1, cx2, cy2, x2, y2)* (zwei Kontrollpunkte, gefolgt von einem Zielpunkt) benutzt werden.
- Die Bibliotheken für Schnittstellen, Video und Netzwerk wurde komplett überarbeitet. Die Referenzseite bietet nähere Details zu diesen Themen.

Lexikalische Änderungen sowie Änderungen der Standardwerte

BImage, BFont ...	PImage, PFont ...
push()	pushMatrix()
pop()	popMatrix()
textureMode(NORMAL_SPACE)	textureMode(NORMALIZED)
textureMode(IMAGE_SPACE)	textureMode(IMAGE)
textMode(ALIGN_LEFT)	textAlign(LEFT)
textMode(ALIGN_RIGHT)	textAlign(RIGHT)
textMode(ALIGN_CENTER)	textAlign(CENTER)
textSpace(SCREEN_SPACE)	textMode(SCREEN)
textSpace(OBJECT_SPACE)	textMode(MODEL)
font.width(String s)	textWidth(String s)
font.ascent()	textAscent()
font.descent()	textDescent()
alpha()	mask()
Previous default ellipseMode was CORNER	ellipseMode(CENTER)
RGB for images	ARGB (more technically accurate)
CENTER_DIAMETER	CENTER
objectX, objectY, objectZ	modelX, modelY, modelZ
curveSegments()	curveDetail()
bezierSegments()	bezierDetail()

splitStrings()	split() (changed in rev 0069)
splitInts()	toInt(split()) (changed in rev 0069)
splitFloats()	toFloat(split())(changed in rev 0069)

Änderungen in Beta 0116+ (Annäherung an Version 1.0)

- framerate() wird jetzt *frameRate()* genannt, um den Processing Konventionen besser zu entsprechen. Es machte keinen Sinn eine *frameCount* Variable und eine *framerate()* Methode als „verunstalteten Cousin“ zu haben.
- Die Standardframerate beträgt jetzt 60 Frames in der Sekunde. Diese Änderung wird einige Eigenheiten von Sketches zerstören, die schneller laufen, als es benötigt wird und die die Systemressourcen auf deinem Computer drosseln. Um diese Einstellung zu deaktivieren, kann die Framerate nach Belieben höher gesetzt werden.
- *beginShape()* wurde verändert. LINE_LOOP und LINE_STRIP wurden entfernt, da sie unnötig waren und nicht zum Rest von Processing passten. *beginShape(POLYGON)* ist nicht mehr länger empfohlen, man sollte einfach einfach *beginShape()* ohne Parameter, wenn komplexe Formen gezeichnet werden.
- *endShape()* mit dem CLOSE Parameter verbindet das Ende der Form mit dem Anfang. Auf die gleiche Art funktionierten POLYGON und LINE_LOOP.
- Die *blend()* Methoden für einzelne Pixel wurde entfernt, sie waren zu viel des Guten. Außerdem wird die Funktion, die Farben anstatt von Bilddaten mischt, jetzt *blendColor()* genannt.
- Neben *blendColor()* wurde die Methode *lerpColor()* hinzugefügt. Sie funktioniert gleich wie *lerp(...)* nur eben mit Farben.
- writer() und reader() heißen jetzt createWriter() und createReader().
- *toInt()* und *toFloat()* sind jetzt *parseInt()* und *parseFloat()*, um an die JavaScript Schreibweise anzuknüpfen.
- Text wird mit den Java2D Einstellungen standardmäßig nicht mehr durch native Fonts dargestellt, das heißt sie werden langsamer und hässlicher dargestellt.
- Threading wurde signifikant verändert. Einige dieser Änderungen wurden in der letzten Minute rückgängig gemacht, also werden weitere Änderungen kommen, die einige „seltsame“ InterruptedException Probleme lösen werden.
- Die PGraphics Klassen wurden alle überarbeitet und umbenannt.
 - PGraphics ist eine abstrakte Klasse, auf der weitere PGraphics Unterklassen aufbauen können.

- PGraphics2D (P2D) beinhaltet die ehemaligen Inhalte von PGraphics, die auf das Rendern im 2-dimensionalen Raum spezifiziert sind. Wenn die Klasse fertig ist, wird dies einmal der P2D Renderer sein. Im Release 0126 ist P2D noch nicht fertig und kann noch nicht genutzt werden.
- PGraphics3D, früher PGraphics3, bleibt fast unverändert.
- PGraphicsJava2D (JAVA2D), früher PGraphics2D, ist der Renderer, der benutzt wird, wenn man `size(w, h)` oder `size(w, h, JAVA2D)` aufruft. Er bleibt der Standard Render, wenn kein anderer definiert wurde. Er ist langsamer, aber genauer als andere.
- PGraphicsOpenGL (OPENGL) ist der neue Name für PGraphicsGL.
- *new PGraphics()* sollte nicht mehr benutzt werden, um PGraphics Objekte zu instanzieren. Anstelle dessen benutze *createGraphics()*, welches das Verbinden der neuen Grafik zum Elternsketch übernimmt und die Funktion *save()* zum Laufen bringt.
- Aus demselben Grund sollte *createImage(width, height, format)* statt *new PImage()* genommen werden, wenn ein neues Bild erstellt wird.
- Die neueren Releases beinhalten eine bessere Unterstützung für Grafiken im ARGB Format. Das ist praktisch für die Funktionen *createImage()* bzw. *createGraphics()*, da eine transparente Oberfläche generiert werden kann. Versichere dich, dass ein Format wie PNG benutzt wird, dass Transparent unterstützt. Die Unterstützung ist noch nicht fertig und es gibt noch eine Handvoll Fehler.
- *beginFrame()* und *endFrame()* werden nun *beginDraw()* und *endDraw()* genannt. Benutze diese Funktionen um Zeichenfunktionen herum, wenn ein PGraphics Objekt durch *createGraphics()* erstellt wurde. Die Methode *defaults()* sollte nicht/muss nicht benutzt werden, sondern nur *beginDraw()* und *endDraw()*.

Änderungen in Processing 1.0 (Revision 0162)

Die größten Änderungen in den Monaten vor der Version 1.0 sind die folgenden:

- **Bibliotheken:** Alle Bibliotheken müssen in einem Ordner mit dem Namen *libraries* im Sketchbook Ordner platziert werden. Man verwendet nicht den Hauptbibliotheken-ordner im Processing Installationsverzeichnis, da dieser für core libraries reserviert ist und er bei Mac OS X nicht sichtbar ist.
- **XML:** Die XML Bibliothek ist seit Neuestem automatisch eingebunden, daher wird man sie im „Import Library“-Menü nicht mehr finden. Außerdem wurde die XML Bibliothek seit 0135 sehr verbessert, sodass sie mit viel mehr verschiedenen Dokumenten klarkommt.

- **Processing.app:** Das Max OS X Release von Processing ist nun eine einzige .app Datei, wie es sich für eine OS X Applikation gehört.
- **Processing.exe:** Das Windows Release hat einen neuen Starter, der auf launch4j basiert. Unglücklicherweise haben manche Computer mit dem neuen Starter ein Problem und wir können das Problem nicht auffinden. PDE Dateien können jetzt auch durch einen Doppelklick in Windows geöffnet werden.
- **OpenGL:** Alle OpenGL Sketche benutzen zweifaches Anti-aliasing. Das heißt, dass Sketche immer geglättet erscheinen und *smooth()* und *noSmooth()* ignoriert werden. Um zum alten Verhalten in den Beta Releases zurückzukehren, muss die Referenz von *hint()* betrachtet werden.
- **P2D und P3D:** Der P2D Renderer ist zurück (siehe Referenz von *size()*) und Anti-aliasing ist für P2D und P3D aktiviert worden. Die Unterstützung ist unglücklicherweise unvollständig und daher können manchmal dünne Linien innerhalb der Formen entstehen. Das ist ein Bug mit sehr großer Priorität, der in den zukünftigen Releases behoben wird.
- **Candy und PShape:** Die Candy SVG Bibliothek wurde zu den core libraries aufgenommen, die eine neue Funktion *loadShape()* und das Objekt *PShape* mit sich bringt. Die speziellen Stärken von *PShape* werden in zukünftigen Releases eingeführt. Zur jetzigen Zeit funktioniert *loadShape()* am besten mit dem Standard Render(JAVA2D). Komplexe Formen werden oft zackig erscheinen oder nicht einmal gerendert mit P2D, P3D und OPENGL. Es wurde an einer besseren Unterstützung für SVG Dateien gearbeitet, die mit Inkscape erstellt wurden.
- **PVector:** Es wurde eine neue Klasse *PVector* implementiert, die eine einfache 3-dimensionale Vektor (auch bekannt als Punkt oder Tupel) Klasse ist. Sie ist praktisch um Punktdaten zu speichern oder Operationen an 3D Punkten vorzunehmen.
- **Werkzeuge:** Eine neue Werkzeugschnittstelle wurde für Entwickler kreiert, die Code besteuern möchten, der die Processing Entwicklungsumgebung in einer Art und Weise erweitert. Es gibt bereits einige kreative Werkzeuge wie „Color Selector 2.0“, „Rot13 Code Mangler“ und „I Am Rich“. Mehr Informationen zu diesem Thema findet man auf der Entwicklerseite für Werkzeuge. Ähnlich wie Bibliotheken werden Werkzeuge in einem Ordner installiert, der sich im Sketchbook Ordner befindet.

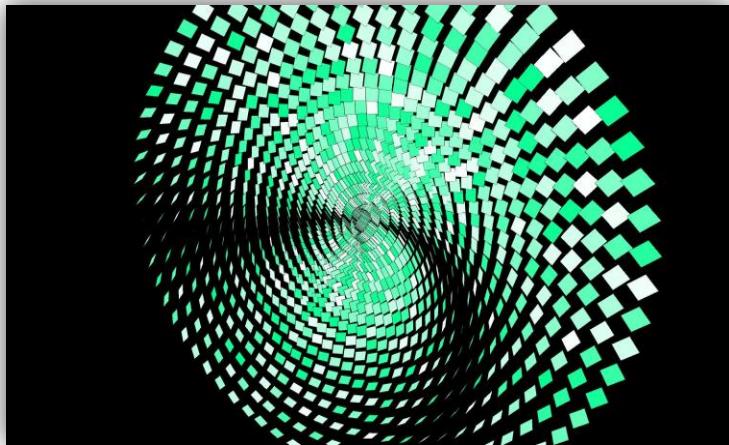
- **Asynchrone Bilder:** Es wurde eine neue Funktion `requestImage()` hinzugefügt, die eine Grafik im Hintergrund lädt, damit der Sketch nicht einfriert, wenn große Bilder bei einer schlechten Internetverbindung geladen werden.
- **Sounds:** Wir haben die Minim Bibliothek von Damien Di Fede zum Download hinzugefügt. Sie verwendet die JavaSound Schnittstelle, um eine Audio Bibliothek bereitzustellen, die sich leicht bedienen lässt. Die Bibliothek besteht nur aus wenigen Befehlen, bietet jedoch vergleichsweise viel Flexibilität für fortgeschrittene User. Besten Dank an Damien für seine harte Arbeit an dieser großartigen Bibliothek.
- **Present:** Der Presentmodus (Vollbild) wird anders gehandhabt in 1.0. Wenn er innerhalb der PDE gestartet ist, benutzt nur Mac OS X den exklusiven Bildschirmmodus mit dem Presentmodus, Windows und Linux machen ein Vollbildfenster. Außerhalb der PDE, erstellen alle drei ein nicht dekoriertes Fenster in der Größe des ganzen Bildschirms und bei einem Mac wird eine Option in die Datei Info.plist geschrieben, damit das Dock und die Menüleiste verschwinden (da es programmtechnisch von innerhalb der Java Applikation nicht möglich ist.)
- **Compiler:** Der alte Jikes Compiler wurde entfernt und stattdessen wird der ECJ Compiler vom Eclipse Projekt benutzt. Es wurde auch versucht, die Qualität der Fehlermeldungen zu verbessern, da noch immer einige Prachtstücke vorhanden sind, die wie Fehlermeldungen von Mainframe Computern in 1970er Filmen klingen.
- **Internationalisierung:** Um eine bessere internationalisierte Unterstützung zu erlangen, haben wir die Kodierung auf UTF-8 geändert, wenn Sketche geladen und gespeichert werden. Sketche, die Zeichen enthalten, die nicht dem ASCII Standard entsprechen und in Processing 1.4 oder früher geöffnet wurden, sahen seltsam aus. Entstellter Text und seltsame Zeichen sind Folgen von Verwendungen von Umlauten, Cedillen und japanischen Zeichen. Wenn das passiert, muss „Fix Encoding & Reload“ im Werkzeugmenü gedrückt werden. Dadurch wird der Sketch neu geladen und man kann ihn in richtigen UTF-8 Versionen neu speichern.
- **Java - Linux and Windows** inkludieren jetzt Java 6 update 10 im Download. Es gibt noch immer keine Unterstützung für die Java 1.5 Syntax, aber es wird gehofft, dass die bessere Performanz in Java 6 helfen wird, damit die Anwendungen stabil laufen.

Bevorstehende Änderungen in Processing 2.0

Seit dem 18.Juni 2011 werden große Änderungen in Hinblick auf Processing 2.0 welches im späten Sommer oder früheren Herbst herausgegeben wird, gemacht. Beginnend mit Revision 0198 gibt es einige Dinge, die neu bzw. alt sind, repariert oder entfernt wurden.

- **P2D und P3D** wurden entfernt. P2D wird deswegen einfach versuchen, den Java2D und P3D den OpenGL Renderer zu benutzen. Diese Änderungen erleichtern uns die Arbeit und es können die Bemühungen der Entwickler auf bessere Synchronität der Befehle bei den verschiedenen Implementationen fokussieren werden. Dadurch wird zum Beispiel das Zusammenspiel mit JavaScript auf der Android Plattform besser.
- **OpenGL 2:** Eine neue Version der OpenGL Bibliothek wurde implementiert und die alte Variante wurde entfernt. Die neue Bibliothek basiert auf Andres Colubris Android Arbeit (und seinen Erfahrungen mit der Entwicklung der GLGraphics Bibliothek). Alle großartigen Dinge von Android wurden rückportiert zur Desktopversion von Processing, sodass wir eine sehr schnelle OpenGL Bibliothek besitzen.
- **Video:** Das Programm QuickTime für die Java Videobibliothek wurde entfernt und nun wird eine abgeänderte Version von Andres Colubris GSVideo Bibliothek benutzt. Zurzeit muss GStreamer auf dem Betriebssystem installiert sein, aber dies wird vor dem Release von 2.0 bereinigen.
- **Das XMLElement** gibt es nicht mehr, aber es wurde PNode dafür hinzugefügt. XML Daten werden mit *loadNode(„blah.xml“)* innerhalb von PApplet gelesen. Die Methoden haben sich nicht geändert, außer dass *getAttribute()* jetzt *getXxx()* ist, zum Beispiel wurde aus *getIntAttribute()* einfach *getInt()*. Auch JSON Daten werden mit PNode und *loadNode()* in der Zukunft unterstützt werden. Eine weitere hinzugefügte Funktion ist *PNodeXML.parse(String)*, die ein PNode Objekt von einem String returned, der XML Daten enthält.
- **screen.width** und **screen.height** wurden durch screenWidth und screenHeight ersetzt. Die alten Funktionen waren nur halb dokumentiert und stimmten nicht mit der Processing Syntax überein.
- **delay()** wurde entfernt. Niemand verstand, was es tat und wenn sie es taten, verstanden sie nicht, warum es dort ist. Eine große Quelle von Verwirrung, speziell bei Anfängern.
- Runde Vierecke sind durch *rect(x, y, w, h, radius)* und *rect(x, y, w, g, tl, tr, br, bl)* möglich.

Nachwort



Processing ist für mich eine Sprache, die ich mit Deutsch und Englisch vergleiche. Mit ihr lässt sich vielleicht sogar mehr ausdrücken als mit lebenden Sprachen, Grafiken (Bilder) sagen bekanntlich mehr als 1000 Worte. Man bedenke, dass bei Processing dieses sich auch teilweise 24-mal in der Sekunde ändert! Daher hat es mir Spaß gemacht, diese Arbeit zu schreiben

und es dauerte einige Zeit, bis ich die notwendigen Kenntnisse für diese Arbeit angeeignet hatte. Die meisten Informationen habe ich im Laufe der Jahre durch den Einsatz von Processing gelernt, bei den Matrizen half mir die Seite www.khanacademy.org, die ich besonders empfehlen kann. Auf dieser Seite werden mathematische Themen vom einfachen Einmaleins bis hin zum Stoff für Aufnahmeprüfungen an Unis auf Englisch erklärt. In den Videos kann man auf einer Art virtueller Tafel die Schritte des Lehrers nachvollziehen, sollte man ihn akustisch nicht verstehen, können bei diesen YouTube Videos Untertitel in verschiedenen Sprachen eingeblendet werden (leider noch nicht in Deutsch).

Um auf den Laufenden zu bleiben, empfehle ich die Processing Website hin und wieder zu besuchen und die Ankündigungen auf der Startseite sowie der Entwicklerseite unter <http://code.google.com/p/processing/> zu lesen. Eine andere Variante ist das Processing RSS Feed zu abonnieren, um zum Beispiel Neuigkeiten über das Handy abrufen zu können. Processing ist eine überschaubare Programmiersprache und man bekommt Änderungen normalerweise rasch mit, was auch ein weiterer Grund ist, warum ich sie schätze. Sie ist einfach, verständlich, überschaubar und liefert großartige Ergebnisse.

Es ist sehr schade, dass zum jetzigen Zeitpunkt (22.8.2011) die Version 2.0 noch nicht veröffentlicht wurde, da ich gerne auch schon die neuen Features beschrieben hätte. Ich blicke schon in die Zukunft von Processing, in der Sketche durch die neue OpenGL Implementierung noch schneller sein werden. Es ist auch ein großer Fortschritt, dass sich die Entwickler nur mehr auf zwei Renderer spezialisieren, da dadurch die Entwicklung neuer Funktionen



und die Behebung von Fehlern schneller erfolgen kann. Wer sich jetzt denkt: „Nein, ich möchte mich an eine kompliziertere Programmiersprache heranwagen.“, sollte Processing durch den Einsatz von Java erweitern, so wie es in „Music Visual“ getan wurde. Ansonsten kann man verschiedene Programmiersprachen sehr gut auf der Webseite http://de.wikipedia.org/wiki/Liste_von_Hallo-Welt-Programmen/Programmiersprachen vergleichen. Wenn man längere Beispiele sucht, wird man auf dieser Seite <http://99-bottles-of-beer.net/abc.html> fündig.

Allerdings kommt man bei keiner Programmiersprache so schnell zu einem grafischen Resultat wie bei Processing. Ich persönlich habe auch Processing.js probiert und war speziell von der Touchscreen-Unterstützung begeistert. Da man zum Beispiel auf dem iPhone nur schwer kostenlos Apps entwickeln kann, da Flash und Java nicht unterstützt werden, ist Processing.js eine gute Alternative, um Webapps zu erstellen.

Trotzdem meiner geringen Kenntnisse in der Elektronik habe ich mir auch ein Arduinoboard gekauft, um es mit Processing zu steuern. Es ist für mich faszinierend, was mit Processing alles möglich ist. Auch wenn man sich Processing anhand der Befehlsreferenz selbst beibringen kann, kann ich das Buch „Processing – A Programming Handbook for Visual Designers and Artists“ von den Gründern Casey Reas und Ben Fry empfehlen, das in Englisch verfasst wurde. Der englischen Sprache kann man ohnedies nicht entfliehen, da es fast ausschließlich englische Quellen gibt. Andere Tutorials im Internet kann man in der Übersicht unter <http://processing.org/learning/> finden.

Ich hoffe, dass diese Arbeit Sie ermutigt hat, einen Blick auf Processing zu werfen.

„*Processing – Entdecke deine künstlerischen Fähigkeiten*“



Danksagung

Ich möchte mich insbesondere bei meinem Betreuer, Herrn Mag. Bernhard Kopp bedanken, weil er es mir ermöglicht hat, die Programmiersprache Processing in BG/BRG Carneri Graz zu etablieren.

Weiters danke ich meinem Klassenvorstand für die Aufrechterhaltung des guten Klimas in unserer Klasse bis in die 11. Schulstufe.

Unserer Direktorin, Frau Mag. Dr. Barbara Bruglacher, möchte ich danken, weil sie mir ermöglicht hat, am Informatikunterricht auch in der 12. Schulstufe teilzunehmen.

Für das Korrekturlesen dieses Skriptums hat meine gesamte Familie einen Beitrag geleistet, auch dafür möchte ich herzlich danken.



Anhang I – Der gesamte Quellcode von Music Visual

```
01  /*
02   * Music Visual
03   * Made by Alexander Pann; finished version 6.8.2011
04   * If you need help, press the help-button in the menu
05   * www.alexanderpann.at.tf
06  */
07
08 // imports
09 import sojamo.drop.*;
10 import msafluid.*;
11 import processing.opengl.*;
12 import javax.media.opengl.GL;
13 import ddf.minim.*;
14 import ddf.minim.analysis.*;
15 import javax.swing.JOptionPane;
16 import controlP5.*;
17 import peasy.*;
18 import javax.swing.ImageIcon;
19 import java.awt.Cursor;
20
21
22 // SDrop variables
23 SDrop drop;
24 File dropfile=null;
25 URL dropfileurl=null;
26 boolean dropped=false;
27 String dropfilepath="";
28
29 // minim variables
30 Minim minim;
31 AudioPlayer song;
32 BeatDetect beat;
33 BeatListener bl;
34 AudioMetaData meta;
35 String metadata;
36 float kickSize, snareSize, hatSize;
37
38 // msa variables
39 float sourceXOff=random(100);
40 float sourceYOff=random(100);
41 float mouseNormX, mouseNormY, mouseVelX, mouseVelY;
42 float lastpseuX, lastpseuY;
43 float lastSourceX, lastSourceY;
44 float curSourceX, curSourceY;
45
46 // various Variables
47 PeasyCam cam;
48 ControlP5 controlP5;
49 PFont font;
50
51 void setup() {
52   size(screen.width, screen.height, OPENGL);
53   background(0);
54   if (ANTI_ALIASING_4X)
55     hint( ENABLE_OPENGL_4X_SMOOTH );
56
57 // FRAME
58 frame.setLocation(0, 0);
59 frame.setTitle(APP_TITLE);
60 ImageIcon titlebaricon = new ImageIcon(loadBytes(APP_ICON));
```

```
61    frame.setIconImage(titlebaricon.getImage());
62    frame.setCursor(Cursor.CROSSHAIR_CURSOR);
63
64    cam = new PeasyCam(this, width/2, height/2, 0, 940);
65    cam.setActive(false);
66    drop = new SDrop(this);
67
68    // MSA
69    invWidth = 1.0f/width;
70    invHeight = 1.0f/height;
71    aspectRatio = width * invHeight;
72    aspectRatio2 = aspectRatio * aspectRatio;
73    initUI();
74
75    // MINIM
76    minim = new Minim(this);
77    minim.debugOff();
78    newSong(START_SONG);
79
80    // GUI
81    controlP5 = new ControlP5(this);
82    /*
83     * workaround
84     * question: http://forum.processing.org/topic/controlp5-opengl
85     */ controlP5.setAutoDraw(false);
86
87    initgui();
88
89
90
91 void draw() {
92    background(0);
93    // Drag & Drop
94    if (dropped) {
95      dropped=false;
96      if (dropfilepath!="") {
97        newSong(dropfilepath);
98        dropfilepath="";
99      }
100     if (dropfile!= null) {
101       newSong(dropfile.toString());
102       dropfile=null;
103     }
104     if (dropfileurl != null) {
105       newSong(dropfileurl.toString());
106       dropfileurl=null;
107     }
108   }
109   // general
110   if (!fluid_pause) {
111     if (MSA_STANDART_FORCES) {
112       sourceXOff+=SOURCE_NOISE_INC;
113       sourceYOff+=SOURCE_NOISE_INC;
114       curSourceX=noise(sourceXOff)*width;
115       curSourceY=noise(sourceYOff)*height;
116       mouseNormX = curSourceX * invWidth;
117       mouseNormY = curSourceY * invHeight;
118       mouseVelX = (curSourceX - lastSourceX) * invWidth;
119       mouseVelY = (curSourceY - lastSourceY) * invHeight;
120       for (int i=0;i<MSA_NEW_FORCES;i++)
121         addForce(mouseNormX, mouseNormY, mouseVelX, mouseVelY);
122       lastSourceX=curSourceX;
123       lastSourceY=curSourceY;
124     }
125   }
126   // msa fluid
```

```
126 updateTUIO();
127 fluidSolver.update();
128
129 if (drawFluid) {
130     for (int i=0; i<fluidSolver.getNumCells(); i++) {
131         int d = 2;
132         imgFluid.pixels[i] = color(fluidSolver.r[i] * d, fluidSolver.g[i] * d, fluidSolver.b[·
133     }
134     imgFluid.updatePixels();
135 }
136 } // !fluid_pause
137 image(imgFluid, 0, 0, width, height);
138 controlP5.draw(); // workaround
139 if (!fluid_pause) {
140     particleSystem.updateAndDraw();
141     // minm
142     fill(255);
143     if (beat.isKick() || beat.isSnare() || beat.isHat() ) {
144         float pseuX=random(width);
145         float pseuY=random(height);
146         mouseNormX = pseuX * invWidth;
147         mouseNormY = pseuY * invHeight;
148         mouseVelX = (pseuX - lastpseuX) * invWidth;
149         mouseVelY = (pseuY - lastpseuY) * invHeight;
150         for (int i=0;i<MSA_NEW_FORCES;i++)
151             addForce(mouseNormX, mouseNormY, mouseVelX, mouseVelY);
152         lastpseuX=pseuX;
153         lastpseuY=pseuY;
154     }
155 } // !fluid_pause
156
157 // free-rotate mode
158 if (cam.isActive()) {
159     controlP5.controller("info_button").setVisible(true);
160     controlP5.controller("info_button").captionLabel().set(LABEL_3D_ENTER);
161 }
162
163 //Sdrop
164
165 void dropEvent(DropEvent theDropEvent) {
166     if (theDropEvent.isImage())
167         message(LABEL_ONLY_SOUND);
168     else if ((dropfile=theDropEvent.file())!=null)
169         println(LABEL_FILE_ADDED);
170     else if (theDropEvent.isURL()) {
171         try {
172             dropfileurl=new URL(theDropEvent.url());
173             println(LABEL_URL_ADDED);
174         }
175         catch(Exception e) {
176             println(LABEL_URL_INVALID);
177         }
178     }
179     dropped=(dropfilepath!="" || dropfile!= null || dropfileurl != null) ? true:false;
180 }
181
182 void newSong(String title) {
183     try {
184         song.pause();
185     }
186     catch(Exception e) {
187     }
188
189     try {
```

```
191 song = minim.loadFile(title, 2048);
192 song.play();
193 meta = song.getMetaData();
194 String author=(meta.author().isEmpty())?"/":meta.author();
195 String album=(meta.album().isEmpty())?"/":meta.album();
196 String comment=(meta.comment().isEmpty())?"/":meta.comment();
197 String composer=(meta.composer().isEmpty())?"/":meta.composer();
198 String copyright=(meta.copyright().isEmpty())?"/":meta.copyright();
199 String date=(meta.date().isEmpty())?"/":meta.date();
200 String disc=(meta.disc().isEmpty())?"/":meta.disc();
201 String encoded=(meta.encoded().isEmpty())?"/":meta.encoded();
202 String filename=(meta.fileName().isEmpty())?"/":meta.fileName();
203 String genre=(meta.genre().isEmpty())?"/":meta.genre();
204 String orchestra=(meta.orchestra().isEmpty())?"/":meta.orchestra();
205 String publisher=(meta.publisher().isEmpty())?"/":meta.publisher();
206 String songtitle=(meta.title().isEmpty())?"/":meta.title();
207 String track=(meta.track()<=0)?"/":Integer.toString(meta.track());
208
209 metadata=
210     LABEL_META_TITLE      +title      +"\\n"
211     LABEL_META_AUTHOR    +author     +"\\n"
212     LABEL_META_ALBUM     +album      +"\\n"
213     LABEL_META_COMMENT   +comment    +"\\n"
214     LABEL_META_COMPOSER  +composer   +"\\n"
215     LABEL_META_COPYRIGHT +copyright +"\\n"
216     LABEL_META_DATE      +date      +"\\n"
217     LABEL_META_DISC      +disc      +"\\n"
218     LABEL_META_ENCODED   +encoded    +"\\n"
219     LABEL_META_FILENAME  +filename  +"\\n"
220     LABEL_META_GENRE     +genre     +"\\n"
221     LABEL_META_ORCHESTRA +orchestra +"\\n"
222     LABEL_META_PUBLISHER +publisher +"\\n"
223     LABEL_META_TRACK+track;
224 }
225 catch(Exception e) {
226     message(LABEL_SONG_LOADING_ERROR);
227 }
228
229 beat = new BeatDetect(song.bufferSize(), song.sampleRate());
230 beat.setSensitivity(BEAT_SENSITIVITY);
231 b1 = new BeatListener(beat, song);
232
233 fluidSolver = new MSAFluidSolver2D((int)(FLUID_WIDTH), (int)(FLUID_WIDTH * height/width));
234 fluidSolver.enableRGB(true).setFadeSpeed(MSA_FADE_SPEED).setDeltaT(MSA_DELTA_T).setVisc(MSA_
235 imgFluid = createImage(fluidSolver.getWidth(), fluidSolver.getHeight(), RGB);
236 particleSystem = new ParticleSystem();
237 }
238
239 void keyPressed() {
240     if (cam.isActive()) {
241         if (key == ENTER) {
242             free_rotate_toggle.setState(false);
243             free_rotate_toggle.setLabel(LABEL_START_3D);
244             cam.reset();
245             cam.setActive(false);
246         }
247         if (key == 'c')
248             help_button(0);
249     }
250
251     void message(String message) {
252         JOptionPane.showMessageDialog(this, message);
253     }
```

Hinweis: Der abgebildete Quelltext enthält nicht die Funktion `saveframe()`, mit der die Screenshots gemacht wurden. Dazu müsste im Haupttab `keyPressed()` durch folgenden Code ersetzt werden:

```
01 void keyPressed() {
02   if (cam.isActive())
03     if (key == ENTER) {
04       free_rotate_toggle.setState(false);
05       free_rotate_toggle.setLabel(LABEL_START_3D);
06       cam.reset();
07       cam.setActive(false);
08     }
09   if (key == 'c')
10     help_button(0);
11   if(key == 's')
12     saveFrame("snapshot####.jpg");
13 }
```

BeatListener

```
01 // need for beatdetect
02
03 class BeatListener implements AudioListener
04 {
05   private BeatDetect beat;
06   private AudioPlayer source;
07
08   BeatListener(BeatDetect beat, AudioPlayer source)
09   {
10     this.source = source;
11     this.source.addListener(this);
12     this.beat = beat;
13   }
14
15   void samples(float[] samps)
16   {
17     beat.detect(source.mix);
18   }
19
20   void samples(float[] sampsL, float[] sampsR)
21   {
22     beat.detect(source.mix);
23   }
24 }
```

Constants

```
01 final static String APP_TITLE="Music!";
02 final static String APP_ICON="icon.gif";
03 final static boolean ANTI_ALIASING_4X=true;
04
05 final static String START_SONG="Elektrokardiogramm.mp3";
06 final static double SOURCE_NOISE_INC=0.005;
07 final static int BEAT_SENSITIVITY=300;
08
09 final static float MSA_FADE_SPEED=0.003;
10 final static float MSA_DELTA_T=0.5;
11 final static float MSA_VISC=0.0001;
12 final static int MSA_NEW_FORCES=10;
13 final static boolean MSA_STANDART_FORCES=true;
14
15 final static float ALT1_XOFF_INC=0.01;
16 final static float ALT1_THISALPHA_DEC=0.001;
17 final static float ALT1_THISALPHA_2_DEC=0.01;
18 final static float ALT1_THISALPHA_3_DEC=0.005;
19
20 final static String LABEL_3D_ENTER="3-D Modus kann durch ENTER beendet werden";
21 final static String LABEL_HELP="ALT + Maus = Menue bewegen #"+
22 " ALT + [H] = Menue verstecken/anzeigen #"+
23 " [C] = Hilfe anzeigen";
24 final static String LABEL_ONLY_SOUND="Bitte nur Sounddateien verwenden.";
25 final static String LABEL_FILE_ADDED="Datei hinzugefuegt!";
26 final static String LABEL_URL_ADDED="URL hinzugefuegt!";
27 final static String LABEL_URL_INVALID="Die URL ist unguelig!";
28 final static String LABEL_META_TITLE="Titel:";
29 final static String LABEL_META_AUTHOR="Autor:";
30 final static String LABEL_META_ALBUM="Album:";
31 final static String LABEL_META_COMMENT="Kommentar:";
32 final static String LABEL_META_COMPOSER="Komponist:";
33 final static String LABEL_META_COPYRIGHT="Copyright:";
34 final static String LABEL_META_DATE="Datum:";
35 final static String LABEL_META_DISC="CD:";
36 final static String LABEL_META_ENCODED="Encoded:";
37 final static String LABEL_META_FILENAME="Dateiname:";
38 final static String LABEL_META_GENRE="Genre:";
39 final static String LABEL_META_SONGLENGTH="Songlaenge:";
40 final static String LABEL_META_ORCHESTRA="Orchester:";
41 final static String LABEL_META_PUBLISHER="Herausgeber:";
42 final static String LABEL_META_TRACK="Songnummer:";
43 final static String LABEL_SONG_LOADING_ERROR="Es ist ein Fehler aufgetreten beim Laden des Liedes";
44 final static String LABEL_START_3D="3-D-Modus starten";
45 final static String LABEL_GUI_HUE="Farbton";
46 final static String LABEL_GUI_SATURATION="Saettigung";
47 final static String LABEL_GUI_BRIGHTNESS="Helligkeit";
48 final static String LABEL_GUI_PAUSE_SONG="Song pausieren";
49 final static String LABEL_GUI_PAUSE_VISUAL="Visualisation pausieren";
50 final static String LABEL_GUI_START_3D="3-D-Modus starten";
51 final static String LABEL_GUI_UNI_COLOR="Einfarbig";
52 final static String LABEL_GUI_HELP="Hilfe";
53 final static String LABEL_GUI_QUIT="Beenden";
54 final static String LABEL_GUI_3D_STARTED="3-D-Modus gestartet";
55 final static String LABEL_GUI_3D_QUIT="3-D-Modus beenden";
56 final static String LABEL_GUI_VISUAL_CONTINUE="Visualisation fortsetzen";
57 final static String LABEL_GUI_VISUAL_PAUSE="Visualisation pausieren";
58 final static String LABEL_GUI_SONG_CONTINUE="Lied fortsetzen";
59 final static String LABEL_GUI_SONG_PAUSE="Lied pausieren";
60 final static String LABEL_GUI_MENU_SHOW="Menue zeigen";
61 final static String LABEL_GUI_MENU_HIDE="Menue zeigen";
```

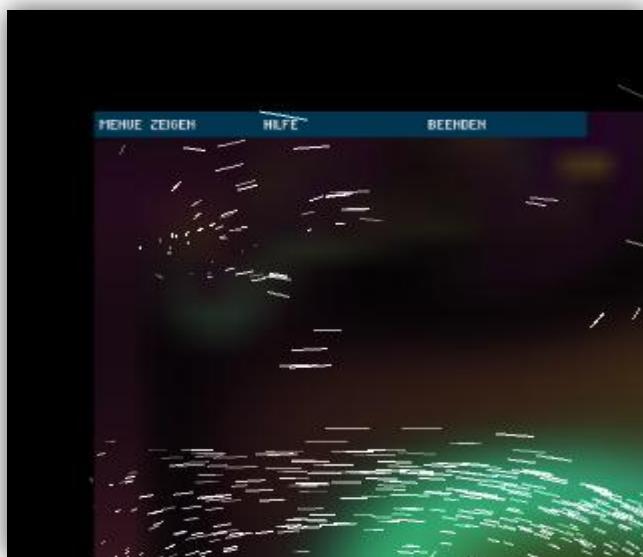
GUI

```
01 int gui_hue=0;
02 int gui_saturation=100;
03 int gui_brightness=100;
04 boolean fluid_pause=false;
05 boolean onecolor=false;
06 boolean gui_show=false;
07
08 Button menu_button, exit_button, help_button, info_button;
09 Slider hue_slider,saturation_slider,brightness_slider;
10 Toggle status_toggle, fluid_pause_toggle, free_rotate_toggle, one_color_toggle;
11 Textarea id3tags_textarea;
12 controlP5.Controller[] elements;
13 controlP5.ControllerGroup[] elements2;
14
15 void initgui() {
16
17     menu_button = controlP5.addButton("menu_button", 0, 0, 0, 100, 19);
18
19     hue_slider=controlP5.addSlider("hue_slider",gui_hue,360,0,20,100,10,100);
20     hue_slider.setLabel(LABEL_GUI_HUE);
21     hue_slider.setNumberOfTypeMarks(10);
22
23     saturation_slider=controlP5.addSlider("saturation_slider",0,100,gui_saturation,70,100,10,10);
24     saturation_slider.setLabel(LABEL_GUI_SATURATION);
25     saturation_slider.setNumberOfTypeMarks(10);
26
27     brightness_slider=controlP5.addSlider("brightness_slider",0,100,gui_brightness,120,100,10,10);
28     brightness_slider.setLabel(LABEL_GUI_BRIGHTNESS);
29     brightness_slider.setNumberOfTypeMarks(10);
30
31     status_toggle=controlP5.addToggle("status_toggle",false,20,30,20,20);
32     status_toggle.setLabel(LABEL_GUI_PAUSE_SONG);
33
34     fluid_pause_toggle=controlP5.addToggle("fluid_pause_toggle",false,100,30,20,20);
35     fluid_pause_toggle.setLabel(LABEL_GUI_PAUSE_VISUAL);
36
37     free_rotate_toggle=controlP5.addToggle("free_rotate_toggle",false,220,30,20,20);
38     free_rotate_toggle.setLabel(LABEL_GUI_START_3D);
39
40     id3tags_textarea= controlP5.addTextarea("id3tags_textarea",metadata,20,250,400,500);
41
42     one_color_toggle=controlP5.addToggle("one_color_toggle",false,320,30,20,20);
43     one_color_toggle.setLabel(LABEL_GUI_UNI_COLOR);
44
45     help_button = controlP5.addButton("help_button", 0, 100,0, 100, 19);
46     help_button.setLabel(LABEL_GUI_HELP);
47
48     exit_button = controlP5.addButton("exit_button", 0, 200,0, 100, 19);
49     exit_button.setLabel(LABEL_GUI_QUIT);
50
51     info_button = controlP5.addButton("info_button", 0, width/2-222,height/2, 444, 80);
52     hide_button("info_button");
53
54     elements = new controlP5.Controller[] { hue_slider,saturation_slider,brightness_slider,status_toggle };
55     elements2 = new controlP5.ControllerGroup[] { id3tags_textarea };
56
57     hidogui();
58 }
59
60
61     void info_button(int value) {
62         hide_button("info_button");
63 }
```

```
63 }
64 void hide_button(String name) {
65     controlP5.controller(name).setVisible(false);
66 }
67
68 void help_button(int value) {
69     controlP5.controller("info_button").setVisible(true);
70     controlP5.controller("info_button").captionLabel().set(LABEL_HELP);
71 }
72
73 void exit_button(int value) {
74     exit();
75 }
76
77 void one_color_toggle(boolean flag) {
78     if(flag) {
79         onecolor=true;
80     } else {
81         onecolor=false;
82     }
83 }
84
85 void free_rotate_toggle(boolean flag) {
86     if(flag) {
87         free_rotate_toggle.setLabel(LABEL_GUI_3D_STARTED);
88         cam.setActive(true);
89     } else {
90         free_rotate_toggle.setLabel(LABEL_GUI_3D_QUIT);
91         cam.setActive(false);
92     }
93 }
94
95 void fluid_pause_toggle(boolean flag) {
96     if(flag) {
97         fluid_pause=true;
98         fluid_pause_toggle.setLabel(LABEL_GUI_VISUAL_CONTINUE);
99     } else {
100        fluid_pause=false;
101        fluid_pause_toggle.setLabel(LABEL_GUI_VISUAL_PAUSE);
102    }
103 }
104
105 void status_toggle(boolean flag) {
106     if(flag) {
107         song.pause();
108         status_toggle.setLabel(LABEL_GUI_SONG_CONTINUE);
109     } else {
110         song.play();
111         status_toggle.setLabel(LABEL_GUI_SONG_PAUSE);
112     }
113 }
114
115 void menu_button(int value) {
116     gui_show=!gui_show;
117     if(gui_show)
118         showgui();
119     else {
120         hidegui();
121     }
122 }
123
124 void hidegui() {
125     for(controlP5.Controller cur:elements)
126         cur.hide();
127     for(controlP5.ControllerGroup cur:elements2)
```

```
128     cur.hide();
129     menu_button.setLabel(LABEL_GUI_MENU_SHOW);
130 }
131
132 void showgui() {
133   for(controlP5.Controller cur:elements)
134     cur.show();
135   for(controlP5.ControllerGroup cur:elements2)
136     cur.show();
137   menu_button.setLabel(LABEL_GUI_MENU_HIDE);
138 }
139
140 void hue_slider(int value) {
141   gui_hue=value;
142 }
143
144 void saturation_slider(int value) {
145   gui_saturation=value;
146 }
147
148 void brightness_slider(int value) {
149   gui_brightness=value;
150 }
```

Damit der Sketch Musiv Visual korrekt dargestellt wird, muss die Zeile 64 im Haupttab, in der die Kamera positioniert wird, geändert werden. [cam = new PeasyCam(this, width/2, height/2, 0, 940)]. Der Wert 940 kann experimentell an die jeweilige Bildschirmauflösung angepasst werden. Sollte der Wert übernommen werden, füllt die Simulation möglicherweise nicht den ganzen Bildschirm aus und es entstehen schwarze Ränder an den Seiten.



MSA

```
01 ****
02 Copyright (c) 2008, 2009, Memo Akten, www.memo.tv
03 *** The Mega Super Awesome Visuals Company ***
04 * All rights reserved.
05 *
06 ****
07
08 final float FLUID_WIDTH = 120;
09
10 float invWidth, invHeight; // inverse of screen dimensions
11 float aspectRatio, aspectRatio2;
12
13 MSAFluidSolver2D fluidSolver;
14
15 ParticleSystem particleSystem;
16
17 PImage imgFluid;
18
19 boolean drawFluid = true;
20
21
22 void mouseDragged() {
23     float mouseNormX = mouseX * invWidth;
24     float mouseNormY = mouseY * invHeight;
25     float mouseVelX = (mouseX - pmouseX) * invWidth;
26     float mouseVelY = (mouseY - pmouseY) * invHeight;
27     addForce(mouseNormX, mouseNormY, mouseVelX, mouseVelY);
28 }
29
30 // add force and dye to fluid, and create particles
31 void addForce(float x, float y, float dx, float dy) {
32     float speed = dx * dx + dy * dy * aspectRatio2; // balance the x and y compo
33
34     if(speed > 0) {
35         if(x<0) x = 0;
36         else if(x>1) x = 1;
37         if(y<0) y = 0;
38         else if(y>1) y = 1;
39
40         float colorMult = 5;
41         float velocityMult = 30.0f;
42
43         int index = fluidSolver.getIndexForNormalizedPosition(x, y);
44
45         color drawColor;
46
47         colorMode(HSB, 360, 100, 100);
48         float hue = onecolor?gui_hue:((x + y) * 180 + frameCount) % 360;
49         drawColor = color(hue,gui_saturation,gui_brightness); /* gui_hue */
50         colorMode(RGB, 1);
51
52         fluidSolver.rOld[index] += red(drawColor) * colorMult;
53         fluidSolver.gOld[index] += green(drawColor) * colorMult;
54         fluidSolver.bOld[index] += blue(drawColor) * colorMult;
55
56         particleSystem.addParticles(x * width, y * height, 10);
57         fluidSolver.uOld[index] += dx * velocityMult;
58         fluidSolver.vOld[index] += dy * velocityMult;
59     }
60 }
```

MSAParticle

```
01  ****
02  Copyright (c) 2008, 2009, Memo Akten, www.memo.tv
03  *** The Mega Super Awesome Visuals Company ***
04  * All rights reserved.
05  ****
06
07  class Particle {
08      final static float MOMENTUM = 0.5;
09      final static float FLUID_FORCE = 0.6;
10
11      float x, y;
12      float vx, vy;          // particle's size
13      float radius;         // particle's size
14      float alpha;
15      float mass;
16
17      void init(float x, float y) {
18          this.x = x;
19          this.y = y;
20          vx = 0;
21          vy = 0;
22          radius = 5;
23          alpha = random(0.3, 1);
24          mass = random(0.1, 1);
25      }
26
27
28      void update() {
29          // only update if particle is visible
30          if(alpha == 0) return;
31
32          // read fluid info and add to velocity
33          int fluidIndex = fluidSolver.getIndexForNormalizedPosition(x * invWidth, y * invHeight);
34          vx = fluidSolver.u[fluidIndex] * width * mass * FLUID_FORCE + vx * MOMENTUM;
35          vy = fluidSolver.v[fluidIndex] * height * mass * FLUID_FORCE + vy * MOMENTUM;
36
37          // update position
38          x += vx * (beat.isKick() ? 1 : -1);
39          y += vy * (beat.isKick() ? 1 : -1);
40
41          // bounce off edges
42          if(x < 0) {
43              x = 0;
44              vx *= -1;
45          } else if(x > width) {
46              x = width;
47              vx *= -1;
48          }
49
50          if(y < 0) {
51              y = 0;
52              vy *= -1;
53          } else if(y > height) {
54              y = height;
55              vy *= -1;
56          }
57
58          // hackish way to make particles glitter when they slow down a lot
59          if(vx * vx + vy * vy < 1) {
60              vx = random(-1, 1);
61          }
62      }
```

```

63         vy = random(-1, 1);
64     }
65
66     // fade out a bit (and kill if alpha == 0);
67     alpha *= 0.999;
68     if(alpha < 0.01) alpha = 0;
69   }
70
71
72   void updateVertexArrays(int i, FloatBuffer posBuffer, FloatBuffer colBuffer) {
73     int vi = i * 4;
74     posBuffer.put(vi++, x - vx);
75     posBuffer.put(vi++, y - vy);
76     posBuffer.put(vi++, x);
77     posBuffer.put(vi++, y);
78
79     int ci = i * 6;
80     colBuffer.put(ci++, alpha);
81     colBuffer.put(ci++, alpha);
82     colBuffer.put(ci++, alpha);
83     colBuffer.put(ci++, alpha);
84     colBuffer.put(ci++, alpha);
85     colBuffer.put(ci++, alpha);
86   }
87
88
89   void drawOldSchool(GL gl) {
90     gl.glColor3f(alpha, alpha, alpha);
91     gl.glVertex2f(x-vx, y-vy);
92     gl.glVertex2f(x, y);
93   }
94
95 }
96 }
```

MSAParticleSystem

```

01  /**************************************************************************
02  Copyright (c) 2008, 2009, Memo Akten, www.memo.tv
03  *** The Mega Super Awesome Visuals Company ***
04  * All rights reserved.
05  * ****
06
07  import java.nio.FloatBuffer;
08  import com.sun.opengl.util.*;
09
10
11  void fadeToColor(GL gl, float r, float g, float b, float speed) {
12    gl.glBlendFunc(GL.GL_SRC_ALPHA, GL.GL_ONE_MINUS_SRC_ALPHA);
13    gl.glColor4f(r, g, b, speed);
14    gl.glBegin(GL.GL_QUADS);
15    gl.glVertex2f(0, 0);
16    gl.glVertex2f(width, 0);
17    gl.glVertex2f(width, height);
18    gl.glVertex2f(0, height);
19    gl.glEnd();
20  }
21
22 }
```

```
23 class ParticleSystem {  
24     FloatBuffer posArray;  
25     FloatBuffer colArray;  
26  
27     final static int maxParticles = 5000;  
28     int curIndex;  
29  
30     Particle[] particles;  
31  
32     ParticleSystem() {  
33         particles = new Particle[maxParticles];  
34         for(int i=0; i<maxParticles; i++) particles[i] = new Particle();  
35         curIndex = 0;  
36  
37         posArray = BufferUtil.newFloatBuffer(maxParticles * 2 * 2); // 2 coordinates per |  
38         colArray = BufferUtil.newFloatBuffer(maxParticles * 3 * 2);  
39     }  
40  
41     void updateAndDraw() {  
42         PGraphicsOpenGL pg1 = (PGraphicsOpenGL) g; // processings opengl graphic:  
43         GL gl = pg1.beginGL(); // JOGL's GL object  
44  
45         gl.glEnable(GL.GL_BLEND); // enable blending  
46         if(!drawFluid) fadeToColor(gl, 0, 0, 0, 0.05);  
47  
48         gl.glBlendFunc(GL.GL_ONE, GL.GL_ONE); // additive blending (ignore alpha)  
49         gl.glEnable(GL.GL_LINE_SMOOTH); // make points round  
50         gl.glLineWidth(1);  
51         gl.glBegin(GL.GL_LINES); // start drawing points  
52         for(int i=0; i<maxParticles; i++) {  
53             if(particles[i].alpha > 0) {  
54                 particles[i].update();  
55                 particles[i].drawOldSchool(gl); // use oldschool rendering  
56             }  
57         }  
58         gl.glEnd();  
59  
60         gl.glDisable(GL.GL_BLEND);  
61         pg1.endGL();  
62     }  
63  
64     void addParticles(float x, float y, int count) {  
65         for(int i=0; i<count; i++) addParticle(x + random(-15, 15), y + random(-15, 15))  
66     }  
67  
68     void addParticle(float x, float y) {  
69         particles[curIndex].init(x, y);  
70         curIndex++;  
71         if(curIndex >= maxParticles) curIndex = 0;  
72     }  
73 }
```

Hinweis: Der Quellcode der Bibliothek MSAFluid darf kopiert, verändert und verbreitet werden, da er mit der BSD (Berkeley Software Distribution) Lizenz, dessen Urtyp von der University of California, Berkeley stammt, lizenziert wurde. Der Copyright-Vermerk darf jedoch nicht entfernt werden.

TuioHandler

```
01  /**************************************************************************
02
03  Copyright (c) 2008, 2009, Memo Akten, www.memo.tv
04  *** The Mega Super Awesome Visuals Company ***
05  * All rights reserved.
06  *
07  * ****
08  */
09
10 import TUIO.*;
11 TuioProcessing tuioClient;
12
13 final static float tuioCursorSpeedMult = 0.02f; // the iphone screen is so small, ea:
14 final static float tuioStationaryForce = 0.001; // force exerted when cursor is stat:
15
16 TuioPoint tuioLastTap;           // stores last tap information (to detect double tap)
17
18 void initTUIO() {
19     tuioClient = new TuioProcessing(this);
20 }
21
22
23 void updateTUIO() {
24     Vector tuioCursorList = tuioClient.getTuioCursors();
25     for (int i=0;i<tuioCursorList.size();i++) {
26         TuioCursor tcur = (TuioCursor)tuioCursorList.elementAt(i);
27         float vx = tcur.getXSpeed() * tuioCursorSpeedMult;
28         float vy = tcur.getYSpeed() * tuioCursorSpeedMult;
29         if(vx == 0 && vy == 0) {
30             vx = random(-tuioStationaryForce, tuioStationaryForce);
31             vy = random(-tuioStationaryForce, tuioStationaryForce);
32         }
33         for(int ii=0;ii<MSA_NEW_FORCES;ii++)
34             addForce(tcur.getX(), tcur.getY(), vx, vy);
35     }
36 }
37
38
39 void addTuioObject(TuioObject tobj) {
40 }
41
42 void updateTuioObject(TuioObject tobj) {
43 }
44
45 void removeTuioObject(TuioObject tobj) {
46 }
47
48
49 void addTuioCursor(TuioCursor tcur) {
50     tuioLastTap = new TuioPoint(tcur); // store info for next tap
51 }
52
53 void updateTuioCursor(TuioCursor tcur) {
54 }
55
56 void removeTuioCursor(TuioCursor tcur) {
57 }
58
59 void refresh(TuioTime bundleTime) {
60 }
```

Anhang II – Cover dieses Buches in Processing

```
01 float startx, starty;
02 float XrectSizeFrom=5;
03 float XrectSizeTo=200;
04 float YrectSizeFrom=5;
05 float YrectSizeTo=200;
06 int times=5;
07
08 void setup() {
09   size(screen.width,screen.height);
10   colorMode(HSB);
11   //noCursor();
12   smooth();
13   //textFont(loadFont("Batang-48.vlw"));
14   //textFont(loadFont("BellMTBold-48.vlw"));
15   render();
16 }
17
18 void draw() {
19 }
20
21 void keyReleased() {
22   if(key==' ') render();
23   if (key=='s') saveFrame("##.jpg");
24 }
25
26 class Part {
27   int transparency;
28   float x, y;
29   float win;
30   float hue, sat, br;
31   int itered;
32   float offset;
33   Part(float x, float y, int itered) {
34     this.x=x;
35     this.y=y;
36     this.itered=itered;
37     this.offset=random(TWO_PI);
38     this.win=random(TWO_PI);
39     this.hue=random(140,160); // 140-160
40     this.sat=255;
41     this.br=255;
42     fill(#FAD412, 50);
43     fill(hue, sat, br, 100);
44     beginShape();
45     float w=random(XrectSizeFrom,XrectSizeTo);
46     float h=random(YrectSizeFrom,YrectSizeTo);
47     for (float i=offset;i<TWO_PI+offset;i+=TWO_PI/4) {
48       float newx=x+sin(i)*w;
49       float newy=y+cos(i)*h;
50       ;
51       stroke(random(255));
52       vertex(newx, newy);
53       if (itered!=0 && random(1) > 0.5) {
54         new Part(newx,newy, itered-1);
55       }
56     }
57     endShape();
58   }
59 }
60
61 void render() {
62   startx=width/2;
```



```
63  starty=height/2;
64  background(0);
65  new Part(startx, starty, times);
66  textSize(35);
67  fill(255);
68  //text("Processing", 150, 200);
69  textSize(15);
70  // text("Entdecke deine künstlerischen Fähigkeiten", 150, 250);
71 }
```

Im oben gelisteten Programm wird grundsätzlich ein Kreis gezeichnet, jedoch nur vier Punkte davon dargestellt und zu einem Viereck verbunden. Zu 50% wird der Funktion die jeweilige Koordinatn der Punkte übergeben. Die Funktion wird nochmals aufgerufen und die Prozedur wiederholt.

Eine Funktion die immer wieder aufgerufen wird, nennt man auch eine rekursive Funktion.

