An Introduction to Processing and Music Visualization

Christopher Pramerdorfer*
Vienna University of Technology

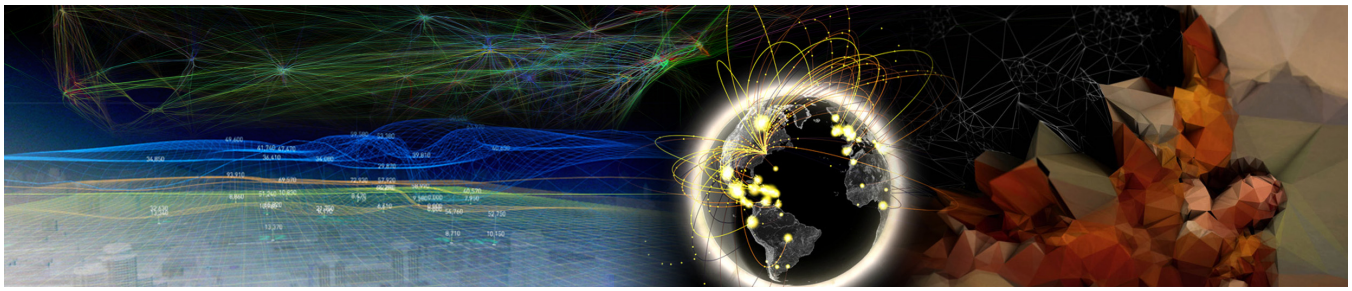

Figure 1: Processing is used in various disciplines, from art to visualization. This Figure is a composition of visualizations created with Processing. Some of the projects these visualizations come from are discussed in section 3.3.

Abstract

Processing is a programming language and environment targeted at artists and designers. It is used in various environments for different purposes, from scientific data visualization to art installations. This paper is comprised of two parts. The purpose of the first part is to provide an insight into Processing and its community as well as to showcase some projects that were created using Processing. The second part focuses on visualization of music. It gives a concise introduction to the theory of how information can be derived from sound signals and presents examples of how that information can be visualized using Processing.

Keywords: processing, processing.org, visualization, music visualization, audio visualization

1 Introduction

Visual abstractions of things and facts, like maps or diagrams, have been used for centuries to aid humans in thinking and communication. Such graphical representations of data or concepts are called *visualizations*. Nowadays in the information age enormous amounts of data are produced. Techniques are required that help humans access that information by supporting their cognition. One of these techniques is the processing of information in a visual way, referred to as *information visualization* or *data visualization*.

One of the greatest benefits of data visualization is the sheer amount of information that can be rapidly interpreted if it is presented well. Visual displays provide the highest bandwidth channel from the computer to the human. We acquire more information through vision than through all the other senses combined [Ware 2004]. With the help of modern computers, interactive visualizations can be prepared automatically at time of use and according to the users needs.

Processing is a programming language and environment based on the Java programming language [Oracle 2010]. Originally targeted at artists and designers, Processing has evolved into a full-blown design and prototyping tool used for large-scale installation work, motion graphics, and complex data visualization [Fry 2008]. Processing runs on all common operating systems, is free to use, easy to learn, and provides sophisticated drawing functionality. This makes Processing a great tool for visualization tasks, amongst others.

Figure 2: The Processing logo.

The goal of the first part of this paper is to provide an introduction to the Processing project. It explains the language differences to Java and addresses some core characteristics and functionality of the API. Furthermore, it introduces the development environment of Processing and how Processing can be used from within Java programs. The first part then continues with an overview on the development of the project and on the community. It closes with a showcase of some projects created with Processing. The second part is about music visualization. It starts with an introduction to how audio data is processed in the computer and provides an overview on how information can be extracted from audio

*e-mail: e0626747@student.tuwien.ac.at

data streams. It then presents approaches on how that information can be used to create realtime visualizations of music in Processing.

2 Processing

The Processing project consists of several parts, which are discussed in this part of the paper. It includes a programming language based on Java, an API that provides the core functionality and several libraries that enhance this functionality, as well as a development environment.

Processing can be downloaded from the official homepage [Processing 2010]. Once downloaded and extracted, one can start the so called *Processing Development Environment* (PDE) by executing a file named *processing* or similar (depending on the operating system). The PDE is the primary development environment for Processing. Figure 3 depicts a screenshot.

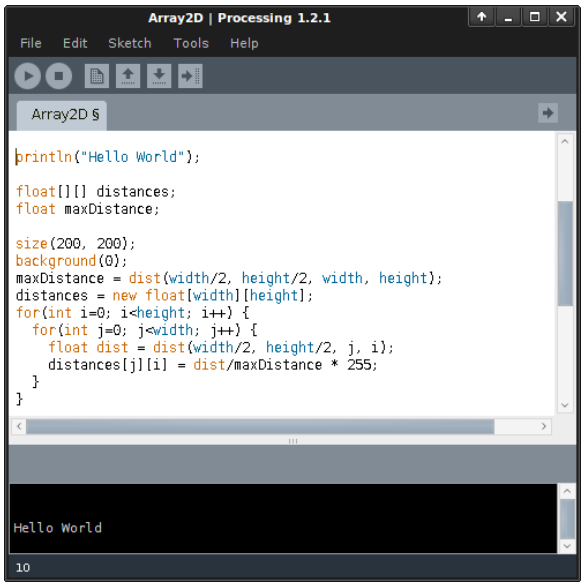

Figure 3: The Processing Development Environment

2.1 The Language

Programs written with Processing are called *sketches*. When the PDE has started, a new sketch is created automatically. Users can start with programming directly, without having to worry about things like project configuration. For instance, the following lines of code:

```
size(400, 200);
background(#FFFFFF);
fill(#33FF00);
ellipse(170, 110, 120, 100);
for(int i = 20; i <= 180; i += 5) {
  line(275, i, 375, i);
}
stroke(#FF0000);
strokeWeight(4);
noFill();
bezier(250, 20, 10, 10, 90, 90, 50, 180);
```

produce the output shown in Figure 4 when the sketch is run. Sketches can be run by clicking at the play symbol at the top left of the PDE or by utilizing the shortcut `ctrl + r`.

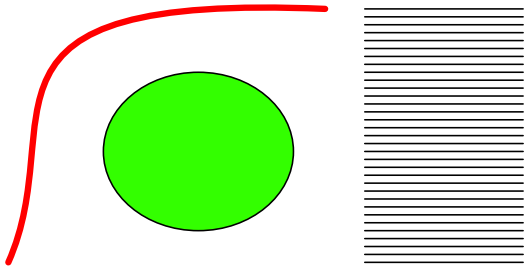

Figure 4: Drawing shapes and lines with Processing.

The previous code provides a basic overview of the syntax of Processing, which is very similar to C or Java. The functionality of Processing is utilized by using functions, which are named in a self-explanatory way. For instance the function `background` lets the user change the background of the resulting image, while `ellipse` draws ellipses. Specified coordinates and dimensions map directly to device pixels by default (the Processing coordinate system is explained in detail in section 2.2.3). Many Processing functions alter the internal state. For instance `fill` sets the fill color of objects to a specific color. This fill color will be used in subsequent drawing operations until another color is specified by calling `fill` again, or until `noFill` is called, which tells Processing to no longer fill shapes. State changes affect only subsequent operations. This makes programming in Processing a sequential process, which greatly simplifies development as it enables users to create Processing applications a few lines at a time. This is referred to as *sketching* by the Processing community.

> „Processing is for writing software to make images, animations, and interactions. The idea is to write a single line of code, and have a circle show up on the screen. Add a few more lines of code, and the circle follows the mouse. Another line of code, and the circle changes color when the mouse is pressed. We call this sketching with code. You write one line, then add another, then another, and so on. The result is a program created one piece at a time."
> – Casey Reas and Ben Fry [Reas and Fry 2010].

The Processing syntax is a dialect of Java. It hides certain parts of Java in order to allow for faster coding and help users not familiar with Java and object oriented programming. Aside from this, both languages are identical. When a Processing sketch is run in the PDE, the Processing preprocessor automatically converts the code to Java source code, which is then compiled to Java byte code. More precisely, the preprocessor creates a new class that extends `PApplet` and moves all code defined in the sketch into this class. For example the following Processing code:

```
background(255);
point(10, 10);
```

in a sketch called Test is converted to:

```
import processing.core.*;
```

```
public class Test extends PApplet {
  public void setup() {
    background(255);
    point(10, 10);
  }
}
```

This means that Processing programs correspond to Java classes. Therefore, the Java API can be accessed as in Java by using import statements. `PApplet` contains all the core API functionality of Processing, that is all Processing functions are implemented inside `PApplet` as Java methods. Because sketches are merely classes that inherit from `PApplet`, they can use all its functionality. `PApplet` is defined in the Java package `processing.core`, which is imported automatically by the preprocessor.

The `setup` method is called once when the program starts, thus the code above will execute only once. This mode is referred to as the basic mode of the preprocessor. It is sufficient only in special circumstances as it creates static images. Most of the time the so called continuous or active mode is used.[1] To enable this mode the Processing code must contain the functions `setup` and `draw`.[2] For instance the code:

```
void setup() {
  size(300, 200);
  stroke(0);
}

void draw() {
  background(255);
  rect(mouseX, mouseY, 50, 50);
}
```

in a Processing sketch called Test is converted to:

```
import processing.core.*;

public class Test extends PApplet {
  public void setup() {
    size(300, 200);
    stroke(0);
  }

  public void draw() {
    background(255);
    rect(mouseX, mouseY, 50, 50);
  }
}
```

when the sketch is run in the PDE. The `setup` method is again called only once when the program is started. Therefore, it should be used to define initial environment properties such as screen size, and to load resources such as fonts or images. After `setup` has finished, the program continually calls the `draw` method, by default at a rate of 60 times per second. This allows for interactivity. For instance, the above code produces a rectangle that follows the mouse. This illustrates how easy it is to create simple interactive applications in Processing.

Sketches can be extended with additional source files, which are referred to in the PDE as *tabs*. Every tab can contain one or more class definitions. The preprocessor treats classes defined in tabs as inner classes of the class created from the sketch. Therefore, classes defined in tabs can access functions defined in the sketch as well as those defined in `PApplet`. However, static fields and methods are not available to these classes as Java does not support static members for inner classes.

One can also create a pure Java source file by creating a tab with the name postfix `.java`. Classes defined this way are not converted to inner classes by the preprocessor, so if a class needs access to methods of the host `PApplet`, a reference must be passed to it. The following example shows how this can be accomplished.

```
import processing.core.*;

public class Shape {
  PApplet _p;
  public Shape(PApplet p) { _p = p; }
  public void draw() { // use _p to draw ... }
}
```

The above class defined in the tab `Shape.java` can be instantiated and used in the sketch by writing:

```
Shape s = new Shape(this);
s.draw();
```

While Java and Processing are almost identical languages, there are some differences. In Java, floating point values are stored as `double` by default. The Processing language and API instead use the `float` data type because of memory and speed advantages. As a result, `float f = 1.0` is valid syntax in Processing, but not in Java. The preprocessor tags all non-integer values as `float` by adding an `f` to them (i.e., `1.0` becomes `1.0f`).

The Processing language supports a literal for specifying colors in the form `#XXYYZZ` or `#AAXXYYZZ`, as introduced in previous code examples. In the RGB color mode (the default) this literal works like for instance in HTML or CSS: `AA` is a hexadecimal value that defines the alpha value of the color, `XX` is the red value, `YY` is green and `ZZ` specifies the blue fraction of the color. Processing also offers a HSB color mode, in which `XX` specifies the hue, `YY` the saturation and `ZZ` the brightness of the color. The preprocessor maps color literals to Java `int` values (i.e., `#FF5500` becomes `0xFFFF5500`).

Another difference is that Processing supports typecasts in the style `type(value)`, e.g. `int(value)` or `float(value)`. The preprocessor converts these constructs to use corresponding `PApplet` methods, like `parseInt(value)` and `parseFloat(value)`, respectively.

2.2 Features of Processing

Processing was created to make it easier to develop visually oriented applications with emphasis on animation and provide users with instant feedback through interaction [Fry 2008]. As introduced in the previous section, Processing enables users to draw shapes in an easy and intuitive way. But the Processing API allows for much more. This section gives a

[1] The preprocessor offers a third mode called Java mode, which can be activated by writing a Java class definition inside the sketch. However, if Processing is to be used at this level, it is often better to use its API directly from Java, as discussed in section 2.3.

[2] One can specify any number of additional functions, which are converted to methods by the preprocessor. Unless specified explicitly, all generated methods are public.

short overview of some important features and characteristics. An extensive API documentation is available on the official website [Processing 2010].

2.2.1 File Input and Output

All local files to be used in Processing must reside inside the `data` directory inside the sketch folder. This eliminates path issues when Processing applications are redistributed. Files inside the `data` folder are easily accessible via the Processing API, which provides functions for loading special types of data. For example, `loadStrings` reads a text file into an array of `String` objects, while `loadShape` allows users to load shapes stored in Scalable Vector Graphics (SVG) format, as shown in the following code example:

```
String[] data = loadStrings("data.txt");
PShape shape = loadShape("shape.svg");
```

`PShape` is one of many utility classes that are part of the Processing API. While `PShape` stores shapes, others store fonts (`PFont`), images (`PImage`), or vectors (`PVector`). Processing also provides functions for saving data, like `saveStrings`, which saves `String[]` arrays to text files. These functions do not write to the `data` directory, but to the directory the sketch is located in.

2.2.2 Support for Interactivity

Interactivity is an important aspect in many visualizations. Therefore, one of the core characteristics of Processing is simple access to information regarding the user, such as mouse position or whether a mouse button or key was pressed. This functionality is implemented in an intuitive way: When an event occurs that involves the user, Processing automatically executes a corresponding event handler function, if available. Event information is stored in variables.[3] The following Processing code demonstrates this:

```
void mousePressed() {
  println("Clicked: " + mouseX + "," + mouseY);
}

void keyPressed() {
  println("Pressed: " + key);
}
```

The above code informs the user when he clicks the mouse or presses a keyboard key. `mouseX` and `mouseY` hold the current mouse cursor position, while `key` stores the character of the keyboard key the user pressed most recently.

2.2.3 Coordinate System and 3D

Processing uses a Cartesian coordinate system with the origin in the upper-left corner of the window. Pixels are drawn to the right and below the coordinate, thus if the image is $m \times n$ pixels in dimension, the last visible pixel in the lower-right corner is at position $(m-1, n-1)$. Coordinates may be specified as integer or float values, the latter are rounded to integers automatically. This means that $(0.4, 0.4)$ maps to the pixel position $(0, 0)$, while $(0.6, 0.6)$ maps to $(1, 1)$.

Processing supports three dimensional geometry. In 3D mode, Processing uses a left-handed Cartesian coordinate system. The z-coordinate is zero at the surface of the image, with negative z-values moving back in space, as shown in Figure 5. Processing supports boxes and spheres as three dimensional primitives. Custom two and three dimensional shapes can be created by specifying a number of vertices that define the shape endpoints, like so:

```
beginShape();
vertex(50, 50);          // x, y (2D)
vertex(50, 100, 100);    // x, y, z (3D)
vertex(100, 75, 75);
endShape(CLOSE);
```

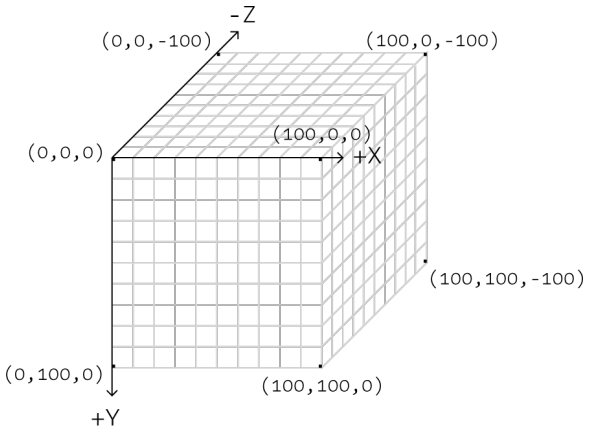

Figure 5: Illustration of the coordinate system used in Processing, assuming a window size of 100×100 pixels. Image taken from [Reas and Fry 2007].

The Processing API includes a number of functions that are helpful when working with the coordinate system. A feature found in most languages that support geometry is coordinate transformations. In Processing, this functionality is provided by the `translate`, `scale` and `rotate` functions, amongst others. Simply spoken, these functions transform the coordinate system (see Figure 7). They are commonly used when drawing three dimensional geometry. for instance, they make it possible to draw 2D primitives in 3D space, as shown in the following code example:

```
rect(0, 0, 50, 50);
translate(30, 30, 50);  // x, y, z
rect(0, 0, 50, 50);
```

Internally, coordinate transformations are implemented as affine transformations by using matrices. A consequence of this is that subsequent calls to transform functions accumulate the effect, e.g. `translate(10, 10)` followed by `translate(20,10)` is the same as `translate(30, 20)`. In order to produce consistent results, the transformation matrix (and thus the coordinate system) is therefore reset every time the `draw` method executes. Two important functions in this context are `pushMatrix` and `popMatrix`. `pushMatrix` adds a new coordinate system to the matrix stack, which initially contains only the default coordinate system. All coordinate transformations are carried out on the currently active coordinate system. Therefore, users can revert trans-

[3] In Processing event handling is implemented in a thread-safe way. Thus, it is save to access any functions and variables from within event handler functions. This is in contrast to other implementations such as in Java AWT (on which Processing's implementation is based).

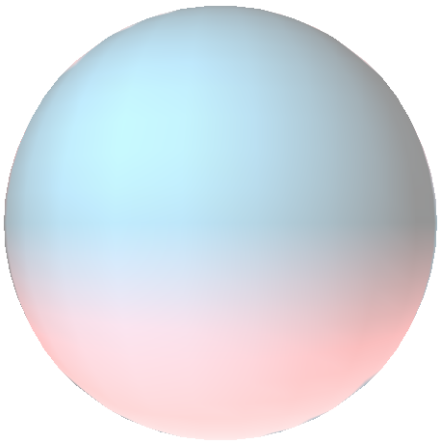

Figure 6: A sphere lit by three lights, created with Processing.

formations by calling `popMatrix`, which removes the current coordinate system and all its associated transformations. Figure 7 illustrates this behavior.

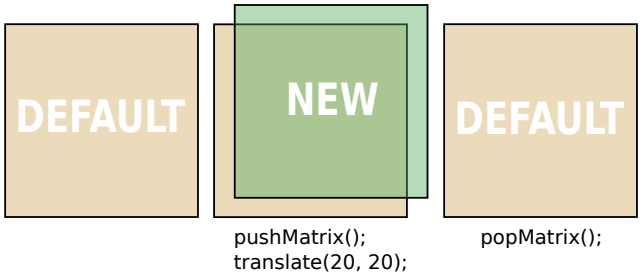

Figure 7: Illustration of functions that relate to the coordinate system: `pushMatrix` adds a new coordinate system to the stack, `translate` translates the coordinate system, and `popMatrix` removes the topmost coordinate system.

Processing also supports more advanced topics such as camera control in three dimensional space, light support, and texturing. These functionality together with transforms discussed in this section allows users to create more sophisticated images. As an example, the image shown in Figure 6 was created with the help of the functions `sphere`, `translate` and three lights.

2.2.4 Image Manipulation

The Processing API makes it easy to read, manipulate, display, and save images. `loadImage` loads an image from inside the `data` directory into a `PImage` object:

```
PImage img = loadImage("img.jpg");
```

The functions of `PImage` allow users to access pixels, to save images to disk or to resize and copy images or parts thereof. Two powerful functions `PImage` offers are `blend`, which blends two images together, and `filter`, which applies filters such as blur. These functions were used to create the image shown in Figure 8. `PImage` objects are compatible with many Processing functions, such as `background`, `image` or `copy`. These functions can be used to display images or image parts. Moreover, Processing supports all functions of the `PImage` class, such as `blur` or `filter`.

Figure 8: An example of image manipulation in Processing. The left and middle parts were manipulated with the `filter` function. For the right part the image was blended with itself using `blend`.

2.2.5 Renderer Alternatives

The actual drawing of two and three dimensional geometry is carried out by a so called renderer. Processing supports five different renderers. The renderer is specified with the `size` function, e.g. `size(300, 200, P2D)`. The default renderer is `Java2D`, a 2D renderer which utilizes Oracles Java2D graphics library. It provides high image quality at the cost of performance. An alternative for rendering two dimensional geometry is `P2D`, Processing's own software-based 2D renderer. `P2D` offers lower quality but is often faster than `Java2D`, especially when dealing with pixel-based data [Processing 2010]. A different renderer is `PDF`, which draws 2D graphics directly to an Adobe PDF file. This mode is useful when vector shapes are needed for high resolution output or printing.

When 3D geometry is to be drawn, a 3D renderer must be used. Processing offers two alternatives. `P3D` is Processing's own implementation for drawing in 3D space. Since it is software based, it is relatively slow and not as accurate as other renderers (see Figure 9). Processing also supports OpenGL rendering through the Java Bindings for OpenGL (JOGL) library. This renderer offers high image quality and the best performance in complex 3D projects, it is however dependent on native libraries and works only on systems with a graphics card that supports OpenGL [Processing 2010]. The OpenGL renderer can be enabled by using `OPENGL` as the third argument of the `size` function. While this renderer offers high quality and speed, it has some drawbacks and limitations. The GLGraphics library introduced in section 3.2 provides an alternative that is also based on OpenGL.

Renderers have different rendering features, which can be enabled or disabled with the `hint` function. For instance,

`hint(ENABLE_OPENGL_4X_SMOOTH)` tells the OpenGL renderer to use anti aliasing, while `hint(DISABLE_DEPTH_TEST)` disables the z-Buffer of the 3D renderers.

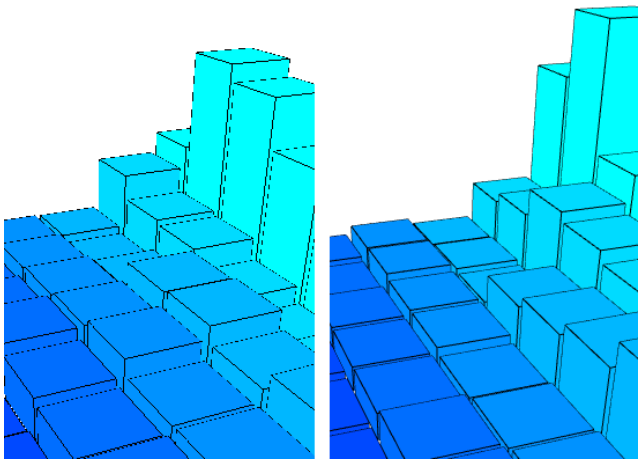

Figure 9: Quality differences between the `P3D` (left) and `OPENGL` (right) renderers. `P3D` does not support anti aliasing, which results in clearly visible aliasing artifacts at box edges.

2.3 Development with Processing

The Processing Development Environment (PDE), introduced at the beginning of this section, is the default development environment for Processing. It comes with a small set of features and is designed as a simple introduction to programming or for testing ideas (which is referred to as *sketching* by the Processing community) [Fry 2008].

The advantages of the PDE are its ease of use and level of integration. It handles preprocessing and compiling automatically and hidden from the user, imports libraries on demand and automatically copies files to be used in the sketch to the `data` folder. The PDE also has the ability to export Processing projects to Java Archives, Java Applets or standalone applications. Another helpful feature is its ability to create bitmap fonts from fonts installed on the system. Processing uses a special `.vlw` font format that stores letters as images. The advantage of these fonts is that they make text rendering system independent as fonts are distributed with the sketch. The disadvantage is that bitmap fonts cannot be scaled without quality loss.

An alternative to the PDE is the official Processing Eclipse plug-in, which enables users to create Processing applications from within Eclipse [Eclipse 2010]. At the time of writing, the plug-in is in beta phase and lacks some features of the PDE, such as tab support and export functionality. More information on the plug-in is available at [Fry 2010].

The PDE is not designed for larger projects. In these cases it is often better to use the Processing API directly within Java. This is possible since the Processing functionality is available in the form of Java classes, as introduced in section 2.1. The core API and libraries are merely Java Archives, which can be imported and used in Java projects. The advantage of this approach is flexibility: It allows programmers to use any Java IDE they like and enables easy integration of Processing into Java projects. Because the `PApplet` class derives from `java.awt.Component`, visuals created with Processing can even be incorporated into existing GUI applications, as depicted in Figure 10. The downside is that the Processing language is no longer available. The following code demonstrates the concepts of integrating Processing and Java:

```
import processing.core.PApplet;

class MySketch extends PApplet {
  public static void main(String[] args) {
    PApplet.main(new String[] { "MySketch" });
  }

  public void setup() { ... }
  public void draw() { ... }
}
```

The `PApplet.main` method initializes and starts the sketch with the name that was passed to it (`MySketch` in the above example). This code resembles the code that is generated by the Processing preprocessor when a sketch is run from within the PDE (see section 2.1). In order for the above code to work, the Java archive `core.jar` containing the Processing core functionality must be located in the Java classpath.

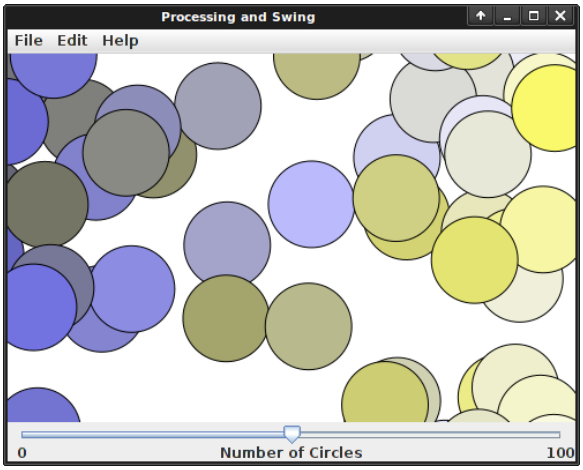

Figure 10: A Processing sketch embedded in a Java Swing GUI. A Swing slider can be used to change the number of circles created by the Processing sketch.

3 Community

3.1 The Origin of Processing

Processing was created in 2001 by Ben Fry and Casey Reas when they were graduate students at MIT Media Lab, working with Professor John Maeda. Processing has its roots in a language called Design By Numbers (DBN), which was created by Maeda with contributions from his students, including Fry and Reas. Design by Numbers was developed for teaching programming concepts to artists and designers with no prior experience in programming. It is easy to learn but not suited for creating advanced applications as it lacks essential features.

When designing Processing the goal of Fry and Reas was to create an environment that allows for easy testing of design

ideas in code, unlike for instance in C++ or Java. The other goal was to make a language for teaching design and art students how to program and to give more technical students an easier way to work with graphics [Reas and Fry 2010]. Thus, Processing is based on Design By Numbers, which makes it easy to learn and use, while providing an extensive feature set that makes it possible to create advanced visual applications.

Early alpha versions of Processing were used by students in workshops at MIT in 2001. In late 2001 and 2002 the first institutions began using the software in their projects, including the Musashino Art University in Tokio and the Hochschule für Gestaltung und Kunst in Basel. The first beta version was released in 2005. In the same year Casey Reas and Ben Fry received the Prix Ars Electronica Golden Nica award for Processing, one of the worlds most important awards in electronic art and culture. The first stable version 1.0 followed in 2008.

3.2 Processing Today

Today, Processing is used throughout the world by students, artists, design professionals, and researchers for learning, prototyping and production [Processing 2010]. The strong community is a major part in the development process, over the last years the software has evolved through conversation between Reas, Fry and Processing users. The popularity of Processing has led to many Processing-based projects, some of which are introduced in this section. Figure 11 shows the relationships between languages and technologies Processing is based on as well as some of those who are based on Processing.

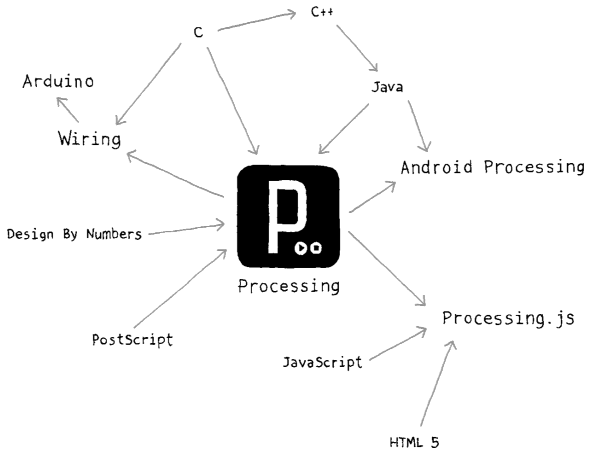

Figure 11: The „family tree" of Processing. An arrow from A to B means „B is based on A", e.g. Processing.js is based on Processing, JavaScript and HTML5. Illustration taken from [Reas and Fry 2010].

Wiring [Wiring 2010] and **Arduino** [Arduino 2010] are platforms for physical computing. They come with microcontrollers and Processing-based languages and development environments to program them. These platforms enable users to receive sensor input and control lights or servo motors, for instance. They are commonly used in art installations.

Processing.js [ProcessingJS 2010] is a JavaScript library that allows Processing code to be run by HTML5 compatible browsers. While the PDE can export Processing sketches to Java applets that run inside a web browser, these applets have the disadvantage that they require Java to be installed on client systems. Processing.js on the other hand uses Java Script and the HTML5 canvas functionality for drawing. This eliminates the need for additional software.

Android Processing [ProcessingAndroid 2010] is a port of Processing that runs on the Android platform. Android is a free and popular mobile operating system developed by the Open Handset Alliance, which is led by Google. At the time of writing, Android Processing is still in development but already includes features such an advanced 3D renderer based on OpenGL and accelerometer support. Figure 12 shows an example.

Figure 12: An Android smartphone running a 3D Processing sketch. Image taken from [Colubri 2010].

As mentioned before, Processing is a community effort. The community has contributed more than 100 libraries that extend the features of Processing. Some of them have become part of the official Processing distribution, such as the Minim audio library discussed in section 4.3. A comprehensive listing is available at the Processing website [Processing 2010]. A particularly useful library is GLGraphics by Andres Colubri [Colubri 2010], which is introduced at this point.

Processing includes an OpenGL renderer that utilizes the graphics card (GPU) to draw geometry (see section 2.2.5). The main benefit of doing graphics related computation on the GPU is speed. Modern GPUs have hundreds of cores that work in parallel and are designed for this type of computations. Without GPU support, larger 3D scenes can become too complex for the CPU to render, which results in low frame rates. Therefore, the OpenGL renderer is usually the best choice when working with 3D scenes. However, it lacks advanced features such as off-screen rendering support and particularly support for the OpenGL Shading Language (GLSL). The GLGraphics library includes an OpenGL based renderer that brings speed improvements and includes support for the features mentioned, amongst others. More information and tutorials are available on the project website [GLGraphics 2010].

GLSL is a programing language based on C that allows programmers to utilize the shaders of the GPU. These shader programs can be used to access and modify vertex data of geometry as well as color and other attributes of each pixel of an image. This allows for advanced effects such as complex

geometry transformations, realistic lighting effects, shadows, and smoke. Shaders can be used to calculate complex rendering effects with a high degree of flexibility. Figure 13 shows an example. For a reference on the topic of GLSL refer to [Rost et al. 2009].

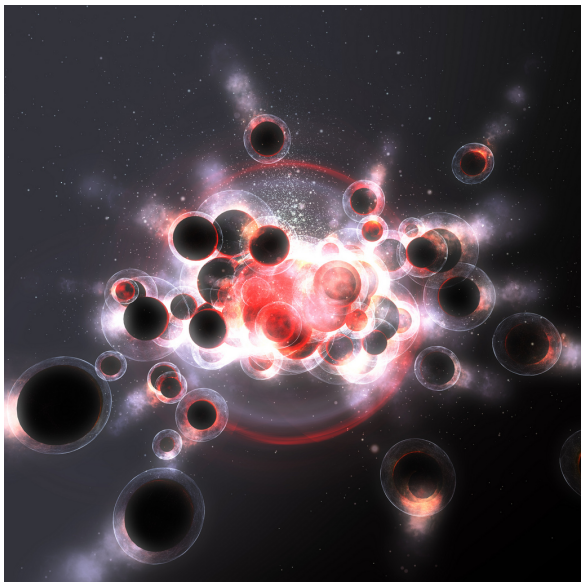

Figure 13: A complex scene that includes effects such as lighting, smoke and particle effects. Image created with Processing by Robert Hodgin [Hodgin 2010]. More projects by Hodgin are introduced in section 4.3.

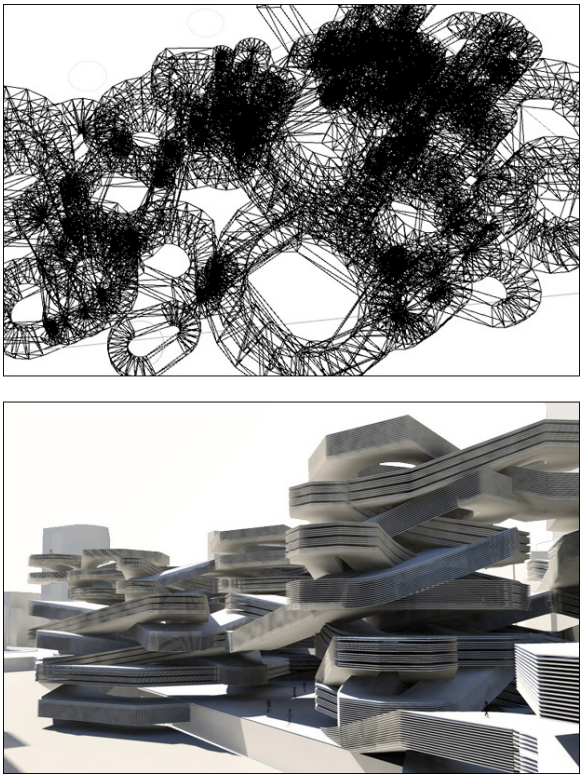

Figure 14: Wireframe structure generated with Processing (top) and rendered version (bottom), as shown on the website of the Computing Kaizen project [Columbia 2010].

3.3 Selected Projects

This section showcases selected projects that were created with the help of Processing. In addition, projects focused on music visualization are presented in section 4.3. A large collection of projects created with Processing is available on its website [Processing 2010]. OpenProcessing [OpenProcessing 2010], an online community platform for sharing and discussing Processing sketches, hosts an extensive collection of open-source Processing projects.

Computing Kaizen was a graduate design studio at the Columbia University Graduate School of Architecture Planning and Preservation. It explored evolutionary architectural structures and their potential to anticipate change and internalize complex relationships. The project used Processing to create intelligent building blocks that could self-organize into innovative forms, as depicted in Figure 14. Further information and illustrations as well as source code is available on the project website [Columbia 2010].

In the Air is a project which aims to visualize the microscopic and invisible agents of air, such as gases, particles, pollen or diseases, in order to see how they perform, react and interact with the rest of the city. The project was created and first used in Madrid and is also in use in Budapest and Santiago de Chile, according to the project website [Medialab-Prado 2008]. The project uses air data coming from sensor stations that are spread out in the city to visualize air pollution over time in an interactive Processing program (Figure 15 (top)). Furthermore, air pollution data is visualized via so called diffuse facades, which are wall-mounted devices that emit vaporized water. This water vapor is colorized with light according to air pollution data coming from the nearest sensor station (Figure 15 (bottom)).

Branching Morphogenesis is an art installation that was on exhibition at the Futurelab of the Ars Electronica Center in Linz (see Figure 16). According to the project website [Sabin and Jones 2008] „Branching Morphogenesis explores fundamental processes in living systems and their potential application in architecture. The project investigates part-to-whole relationships revealed during the generation of branched structures formed in real-time by interacting lung endothelial cells placed within a 3D matrix environment. The installation materializes five slices in time that capture the force network exerted by interacting vascular cells upon their matrix environment. The time lapses manifest as five vertical, interconnected layers made from over 75,000 cable zip ties. Gallery visitors are invited to walk around and in-between the layers, and immerse themselves within an organic and newly created „Datascape" fusing dynamic cellular change with the body and human occupation, all through the constraints of a ready-made."

While the projects introduced so far come from the areas science and art, many creative and interesting Processing projects come from individuals. One example is the **Laser Harp** [Hobley 2010], a musical instrument made of light. It is driven by Arduino, which was introduced in section 3.2. A Laser Harp is a device that emits a fan of light beams. When a beam is cut the device creates MIDI data that can be fed into a synthesizer. This allows the performer to create music by moving their hands in the beams of the Laser Harp.

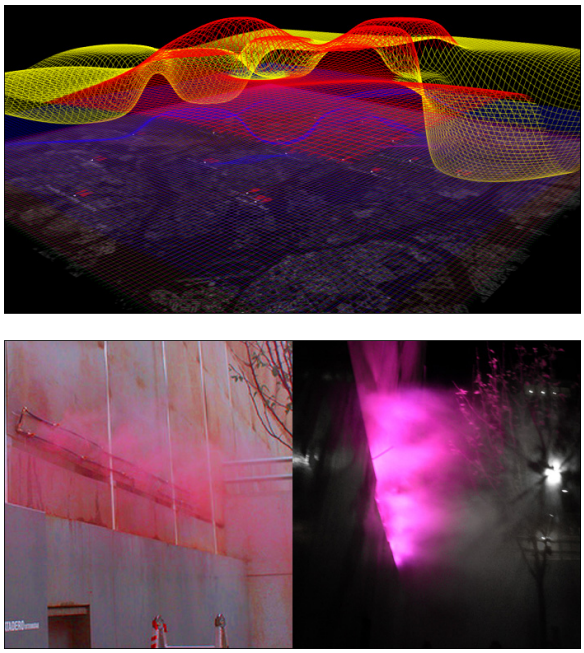

Figure 15: Top: Visualization of the pollution of Madrid's air. Every mesh color refers to a specific contaminant, while the height of the mesh at a given point describes the degree of pollution. Bottom: Prototype of a diffuse facade emitting colorized water vapor based on current pollution statistics (bottom). Images courtesy of [Medialab-Prado 2008].

Figure 17 shows an image of the device. More projects by individuals can be found at the Processing website [Processing 2010]. A list of 40 interesting Arduino projects is available at [HackNMod 2010].

4 Music Visualization

The previous part of this paper gave an introduction to Processing and its features and capabilities. It also presented some projects that were created with the help of Processing. The following sections are dedicated to one of many areas Processing is used: visualization of music. What is referred to as music visualization is the generation of imagery based on music data. This paper addresses real-time approaches, that is generation of imagery based on data coming from music as it is played.

One of the first attempts to electronically visualize music was the Atari Video Music, first released in 1976 [Fourney and Fels 2009]. The device was a console designed to be connected to a HIFI stereo system and a television set. It used the audio data coming in from the stereo system to create images on the TV set in real time. The Atari Video Music had twelve buttons and five knobs that enabled users to create custom video effects. Figure 18 shows a picture.

Software-based music visualization became widespread in the mid to late 1990s with media players such as Winamp. Today, many popular media players support music visualization, such as Apple iTunes (see Figure 19) and Windows Media Player. There are also standalone applications such as G-Force [O'Meara 2010], which can optionally act as a plug-in for media players.

Figure 16: Photo of the Branching Morphogenesis project at Ars Electronica Center in Linz, taken from [Sabin and Jones 2008].

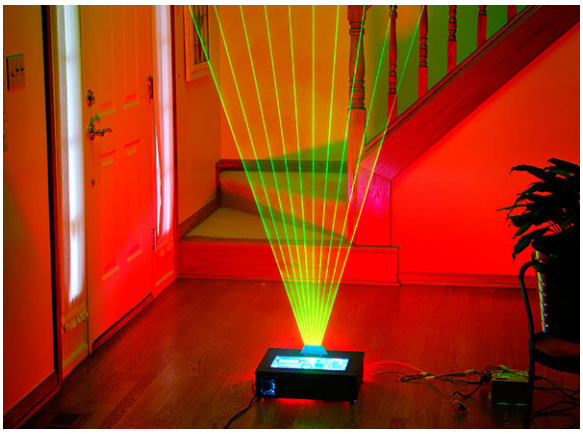

Figure 17: A Laser Harp, an Arduino-based musical instrument made of light. Photo taken from [Hobley 2010].

The following sections give an introduction on the representation, access, and analysis of digital music. Then, the paper introduces approaches on how information derived from music data can be used to create visualizations with the help of Processing.

4.1 Digital Music

What humans perceive as sound is an oscillation of pressure transmitted through matter, usually through air. The change of pressure in a specific spatial position is a continuous process, dependent on time: it can be regarded as a continuous-time signal. This means the pressure can be described by a continuous function $f(t)$, with t being an instant in time. However, continuous-time signals cannot be processed using computer technology. It is therefore required to first transform them to discrete-time signals (see Figure 20). This conversation is carried out by evaluating the amplitude of $f(t)$ at defined time intervals T. The resulting discrete-time signal $x(n)$ with $n \in \mathbb{Z}$ is then given by

$$x(n) = f(T\,n) \tag{1}$$

Figure 18: Picture of the Atari Video Music, one of the first electronic music visualizers. Image taken from `http://en.wikipedia.org/wiki/Atari_Video_Music`.

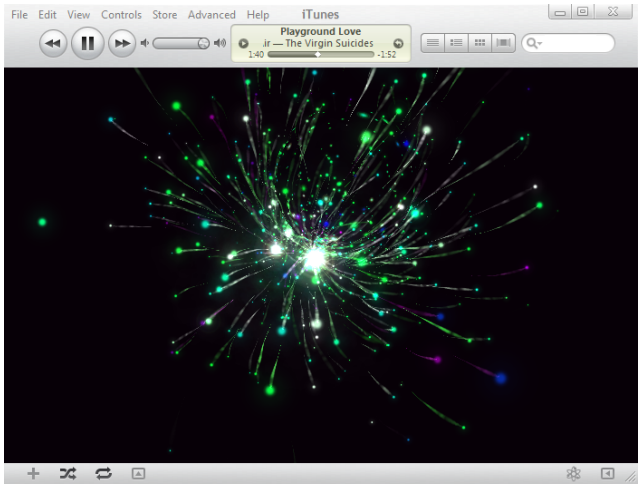

Figure 19: Music visualization is supported by many popular music players, such as iTunes by Apple.

This transformation process is called *sampling*, with T being the sampling interval. The values $x(n)$ are called samples. The inverse of T is the *sample rate* of the signal, also referred to as sampling rate or sampling frequency. Since T is a time unit, the unit of sample rate is hertz (Hz). The sample rate describes the number of samples per second. For instance the Audio CD standard has a sample rate of 44100 Hz. This value is also used by default in audio encoders like MP3.

In practice equation (1) does not apply as the exact values $f(T\,n)$ often cannot be stored exactly on the computer. Hence, the sampled values must be rounded, which is referred to as *quantization*. The Audio CD standard uses a 16-bit quantization, that is sampled values of f can be mapped to $2^{16} = 65536$ discrete values. The process of sampling and quantization is called *pulse-code modulation* (PCM). Audio on the computer is processed as a PCM data stream. Programmers can access, evaluate and modify this audio stream, which is referred to as digital audio signal processing.

4.2 Information Extraction

The approach of extracting information from audio signals is called *content-based audio processing*. It is one of many disciplines to *music information retrieval* (MIR). MIR is

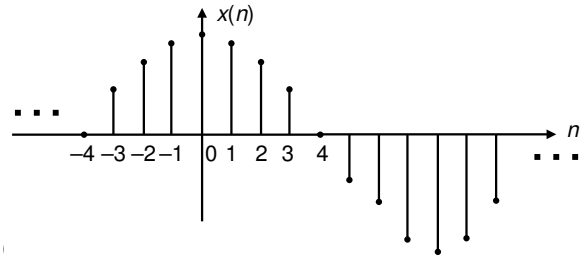

Figure 20: A discrete-time signal $x(n)$. Image courtesy of [Diniz et al. 2010].

a young and active multidisciplinary research domain that addresses the development of methods for computation of semantics and similarity within music.

Many features can be computed from audio signals. Since these features describe aspects of the audio data, they are called *descriptors*. In this paper, the terms feature and descriptor are used synonymously. In the context of audio processing, there are three types of descriptors [Polotti and Rocchesso 2008]: Low-level descriptors are computed from the signal, either directly or after transformation. Mid-level descriptors describe features like music genre or tonality. Algorithms for calculating this kind of information compare features of the audio signal to reference data for classification purposes. High-level descriptors describe features that embrace semantics, like „happy" or „sad". Since perception of high-level features is subjective, algorithms compute these features based on user ratings for other songs.

Mid-level and high-level descriptors describe features that are understandable and thus relevant to users. On the other hand, low-level features are often too abstract and thus not musically-meaningful when presented directly to users. For instance, the amplitude mean of a signal chunk is not meaningful to users because it does not directly correspond to something the user can hear. There are however low-level features that correlate with perceptual attributes. Also, low-level features are required when computing mid- and high-level features, which emphasizes their importance. Because low-level descriptors can be computed directly from the signal and independently from other data, they are practical in realtime audio visualization. Therefore, this section aims to give an introduction on how some low-level features can be extracted and used. It focuses on features that are meaningful to the user, e.g., features that correlate with perceptual attributes. For a comprehensive overview refer to [Polotti and Rocchesso 2008].

The source of content-based audio processing is the PCM audio signal, which is first divided into frames. These frames, representing a time interval at the range of a few ms of the signal, are the basis for further computation. The typical frame length is about 20ms [Polotti and Rocchesso 2008]. Before calculations are carried out, a tapered window function (e.g., a Gaussian or Hanning window) is applied to each frame to minimize the discontinuities at the beginning and end. In addition, consecutive frames are often considered with some overlap, which allows for smoother analysis. Then for each frame one scalar value per descriptor is calculated.

4.2.1 Temporal Features

Many audio features can be computed directly from the temporal representation of these frames via simple statistics, such as the mean, maximum and range of the amplitude of the samples in a frame, the energy, or the zero-crossing rate (ZCR). These features are referred to as *temporal features*.

Two features that loosely correspond to *loudness*, that is the perceived intensity of the sound, are the Amplitude Envelope and the Root Mean Square (RMS). The former descriptor is the maximum of the absolute sample values of a frame. The RMS descriptor is calculated by squaring the samples of a frame f and taking the root of the mean of these values:

$$\mathrm{RMS}(f) = \sqrt{\frac{1}{n}\sum_{n} f(n)^2} \tag{2}$$

A simple and effective way to approximate loudness is to take the 0.23th power of the calculated RMS values [Polotti and Rocchesso 2008]. Loudness approximations from temporal features are vague because loudness is affected by parameters such as sound frequency or bandwidth. The two descriptors discussed are commonly used by algorithms that detect rhythmic structure, such as beat detectors.

The ZCR descriptor is based on the number of times the sign of the samples in a frame f changes. If that number is Z, then the ZCR is defined as [Harrington and Cassidy 1999]

$$\mathrm{ZCR}(f) = \frac{Z\, f_s}{2N} \tag{3}$$

where f_s is the sampling frequency of the signal and N the number of samples in the frame. The ZCR is correlated to the *pitch*, which represents the perceived fundamental frequency of a sound. In the special case of the signal being a sine wave, the frequency of the wave, which equals the pitch, is identical to the zero crossing rate.

4.2.2 Frequency Domain Features

Many audio features can be computed after transforming the signal to the frequency domain. In the transformation process, the input signal is decomposed into its constituent frequencies, called the frequency spectrum. More precisely, the input signal is decomposed into a sum of complex numbers that represent the amplitude and phase of the different sinusoidal components of the input signal. In many applications like audio processing, the phase information is usually not important and thus discarded. Since the spectrum reveals the frequencies an audio signal is composed of, many useful features can be derived from it. The transformation is usually carried out by applying a Discrete Fourier Transform to each signal frame. This approach is called Short-Time Fourier Transform (STFT). An illustration is depicted in Figure 21.

One of the drawbacks of the STFT is its fixed resolution: The STFT yields either good frequency resolution or good time resolution, depending on the window size that is used. In this context, the window size corresponds to the frame size. When calculating the STFT from a signal, a wide frame size gives good frequency resolution but poor time resolution and vice versa. For example, the signal shown in Figure 22 is of the following form:

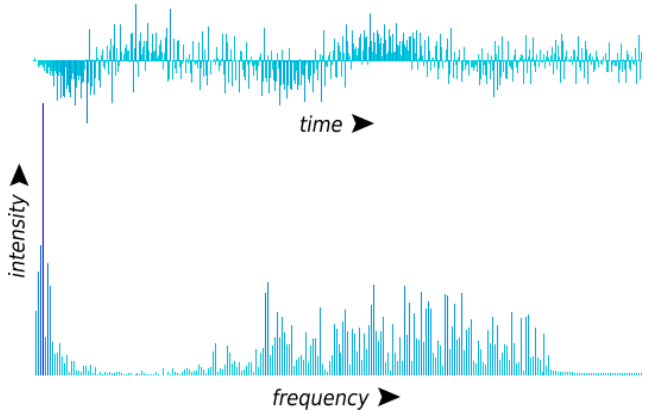

Figure 21: Visualization of the samples of an audio frame containing 1024 samples (top) and the frequency distribution of the frame data, obtained via STFT (bottom). Visualization created in Processing.

$$x(t) = \begin{cases} \cos(2\pi 10\, t/\mathrm{s}) & 0\,\mathrm{s} \le t < 5\,\mathrm{s} \\ \cos(2\pi 25\, t/\mathrm{s}) & 5\,\mathrm{s} \le t < 10\,\mathrm{s} \\ \cos(2\pi 50\, t/\mathrm{s}) & 10\,\mathrm{s} \le t < 15\,\mathrm{s} \\ \cos(2\pi 100\, t/\mathrm{s}) & 15\,\mathrm{s} \le t < 20\,\mathrm{s} \end{cases}$$

This means that the signal is a single sine wave at any given time t. In the spectral domain, a sinusoidal signal would have only one peak at the frequency of the sine, with the intensities of all other frequencies being zero. This is however clearly not the case in Figure 22, which was created with a window size of 25 ms. While the time resolution is high as one can see in the abrupt change of the spectrogram at 5, 10 and 15 seconds, the frequency resolution is poor: In every five second segment of the visualization there should only be one horizontal line representing the sine frequency.

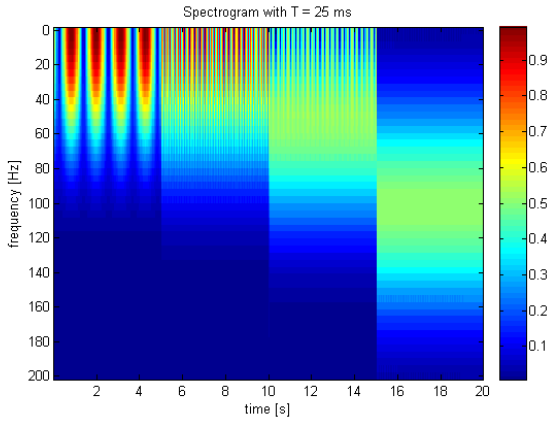

Figure 22: A spectrogram of an audio file, created with a STFT. For more information on spectrograms see section 4.3.2. Image taken from `http://en.wikipedia.org/wiki/STFT`.

As depicted in Figure 23, the frequency resolution increases drastically as the window size is increased to 1000 ms, while the time resolution decreases. If the data computed from the FFT is to be used for real-time music visualization, the

window size should not be too high as otherwise the generated imagery would appear out of sync with the music.

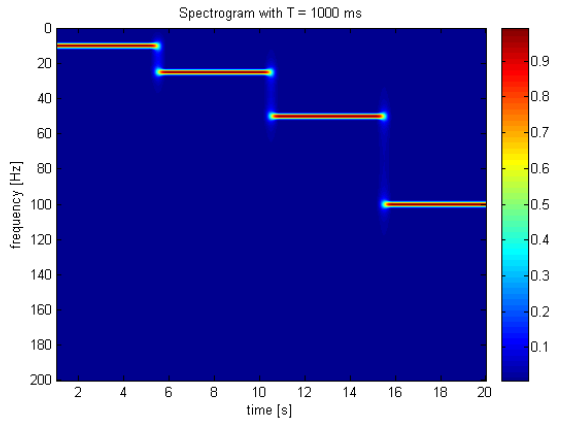

Figure 23: A spectrogram of the same audio file as in Figure 22, created with a STFT window size of 1000 ms. Image taken from `http://en.wikipedia.org/wiki/STFT`.

There are alternatives to the STFT, such as the Wavelet transform, which was developed to overcome the resolution problems of the STFT [Tzanetakis et al. 2001]. In contrast to the STFT, the Wavelet transform provides high time resolution and low frequency resolution for high frequencies and low time resolution and high frequency resolution for low frequencies. These time-frequency resolution characteristics are similar to those of the human ear, which makes the Wavelet transform especially applicable for transforming audio data. The Wavelet transform is however more complex to compute than the Fourier transform and the resolution disadvantage is usually not an issue in music information retrieval and music visualization.

Many features can be extracted directly from the spectral representation, including the spectrum energy, energy values in several sub-bands and statistical features such as the mean, kurtosis, skewness and other values that describe properties of the frequency distribution of a frame. A descriptor that has a robust connection with the impression of „brightness" of a sound is the spectral centroid [Schubert et al. 2004]. It is defined as

$$\mathrm{SC}(f) = \frac{\sum_b f(b)\, a(b)}{\sum_b a(b)} \tag{4}$$

where $f(b)$ is the center frequency of the frequency bin b and $a(b)$ the amplitude of b. The spectral centroid corresponds to the central tendency of the frequency distribution. It is the foundation of other statistical descriptors such as the spectral spread, which describes the variance of the frequency distribution, and the spectral skewness, which describes its level of asymmetry.

While some descriptors computed directly from the audio spectrum are to some extent correlated with perceptional attributes, they represent rather simple concepts and thus are not very powerful. More sophisticated descriptors require additional transformation steps to process. Figure 24 shows some common transformations along with the resulting descriptors.

Hearing is not a purely mechanical phenomenon. Thus, the spectrum of sound is not equal to what humans perceive

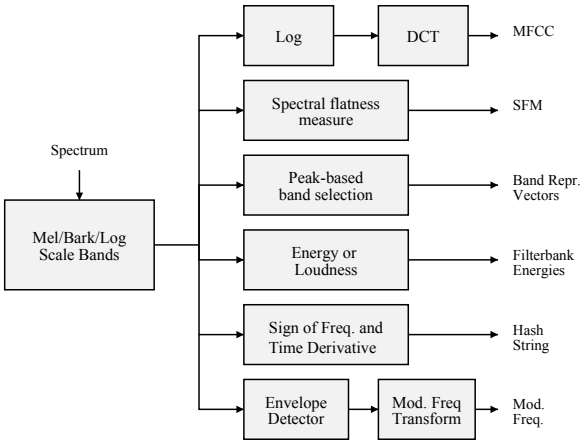

Figure 24: Common spectrum based transforms and descriptors. Image taken from [Polotti and Rocchesso 2008].

when hearing this sound. For instance, loudness is amongst others dependent on frequency, which means that the relative intensities of the frequency bins of the spectrum do not correspond to the intensities humans perceive these frequencies. The study of the relation between sounds and sensations is called *psychoacoustics* [Plack 2005]. The first step of more sophisticated methods is to transform the spectrum in order to approximate hearing perception. Common transforms to accomplish this are the Mel, Bark and Log scales. The Log scale is obtained by taking the logarithms of the amplitudes to simulate loudness perception [Polotti and Rocchesso 2008]. The perceived musical pitch of sound is approximately proportional to the logarithm of frequency. The Mel and Bark scales account for this fact. They are approximately logarithmic in frequency at the high-frequency end, but nearly linear at frequencies below 1 kHz. The Mel scale is used in MFCC computation, which is described below. The formula to convert a frequency f to Mel m is

$$m = 1127 \ln\left(\frac{f}{700} + 1\right) \tag{5}$$

A similar scale is the Bark scale, which corresponds to the 24 critical bands of hearing (the frequency bands the incoming sound is split into in the inner ear). The intensities of Bark scale bands correlate strongly with the loudness [Polotti and Rocchesso 2008].

One of the more advanced descriptors derived from the frequency domain are *Mel Frequency Cepstral Coefficients* (MFCCs). MFCCs are the dominant features used for speech recognition and are also suited to process music [Logan 2000]. The MFCCs model the shape of the spectrum in a compressed form. They are calculated by converting the spectrum of a frame to Mel scale and taking the logarithms of the amplitudes. Then, the discrete cosine transform is applied, which results in a number of coefficients (MFCCs). Usually, only the lower coefficients are used, as they describe the coarse envelope of the spectrum of the frame [Logan 2000].

4.2.3 Temporal Evolution of Features

The descriptors described above are extracted from frames and are therefore called frame feature values. These values approximately describe instantaneous features in the audio

signal, that is features at a given time. Another approach is to focus on the temporal evolution of these features, i.e., the degree of change of feature values between consecutive frames. For instance, the Spectral Flux is modeled to describe the temporal change of the spectrum. It is the Euclidean distance between the normalized frequency distributions of two consecutive frames, and can be regarded as a measure of the rate at which spectrum changes locally [Polotti and Rocchesso 2008].

Descriptors that describe the temporal change of features can further be used to determine region boundaries of sound data, a process which is referred to as *segmentation*. In the most generic way, a region of audio data is a group of consecutive frames that share some kind of similarity. Algorithms can determine region boundaries by using the amount of change of a feature vector: If this amount is higher than a specified threshold, a boundary is detected. This approach is called model-free segmentation.

4.3 Music Visualization with Processing

The preceding section discussed how information can be derived from audio data from a theoretical point of view. In this section approaches are introduced on how this information can be obtained and utilized to create different kinds of visualizations in Processing. Furthermore, this section showcases selected music visualization projects created by the Processing community.

4.3.1 Accessing Audio Data

The Processing core API does not support audio, but several libraries have been developed by the community that provide this functionality. One of them is Minim [Fede 2010] by Damien Di Fede. It is part of the Processing download and includes features for sound access, processing and analysis, amongst others. Minim supports playback of WAV, AIFF, AU, SND, as well as MP3 files, and provides means to access the audio signal from within Processing. For instance, a file `song.mp3` that resides in the Processing `data` directory can be played back with the following Processing code:

```
Minim minim = new Minim(this);
AudioPlayer s = minim.loadFile("song.mp3");
s.play();
```

The `AudioPlayer` class can be used to control playback. It also allows direct access to the signal of the playing song via audio buffers, which are accessible from the fields `left`, `right`, and `mix`. These buffers provide buffered and synchronized access to the next samples to be played back. The buffers thus correspond to audio frames introduced in section 4.2 and are therefore the foundation of feature extraction and visualization. While Minim also provides access to buffers that hold the signals of the left and right speaker, respectively, visualizations discussed in this paper utilize only the `mix` buffer, which contains a mix of both channels. The number of samples n the buffers contain may be specified when an audio file is loaded via `Minim.loadFile`. By default n is 1024. At a sample rate of 44100 Hz, a buffer length of $n = 1024$ relates to $1000/44100 \cdot 1024 \approx 23$ ms of music, which is a common frame size magnitude.

4.3.2 Frame Visualizations

A simple approach to music visualization is to visualize the frame data – that is the buffer data – in raw form. This can be accomplished easily by accessing the `mix` buffer from inside the `draw` method. The buffer data can then be visualized by mapping the buffer length to the width of the window and the sample values[4] to the window height, like so:

```
background(255);  // reset the display window

float ySilence = height/2;
int numFrames = s.mix.size();  // s = AudioPlayer

for(int i = 0; i < numFrames; i++) {
  float x = map(i, 0, numFrames -1, 0, width);
  float y = map(s.mix.get(i), 1, -1, 0, height);
  line(x, ySilence, x, y);
}
```

The result is a graph of a short part of the audio signal, referred to as its *waveform*. Figure 25 shows the result of such a mapping where colors were used to emphasize the intensity of the signal. This illustrates how easy it is to create simple music visualization in Processing. Note that because every frame value is drawn as an distinct vertical line, the width of the Processing window must be at least equal to the buffer size in order not to lose information in the visualization process. On the other hand, if the window width is larger than the buffer size this visualization approach can lead to gaps as the number of lines is mapped to the window width. One solution is to place control points at the calculated `x` and `y` positions and then connect these points by lines. This could be implemented with Processing's `beginShape` and `vertex` functions, for example.

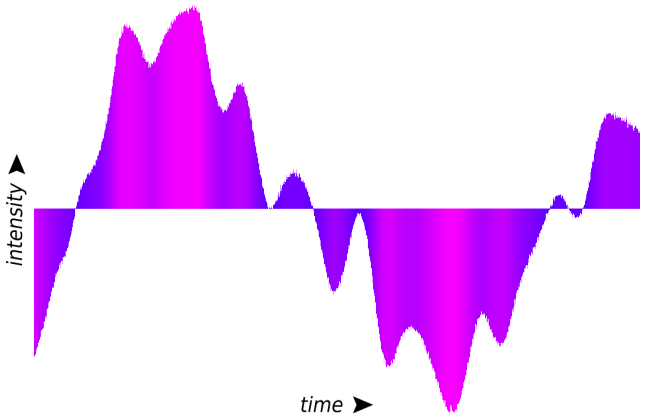

Figure 25: Visualization of a Waveform in Processing.

The approach just described can also be used to visualize the frame data in the frequency domain. Minim provides a STFT implementation that can be used for transformation purposes (see section 4.2.2). The buffer data can be transformed with the following code:

```
FFT f = new FFT(s.bufferSize(), s.sampleRate());
f.forward(s.mix);  // s = AudioPlayer
```

The intensity of a specific frequency band b can then be retrieved by calling `f.getBand(b)`. The number of bands is

[4]The sample values inside the buffers are normalized to the interval $[-1, 1]$, with 0 being silence.

$n/2 + 1$ with n being the number of samples in the buffer. The `FFT` class offers many useful methods that can be used to calculate intensity averages or map bands to frequencies, amongst others. It can also be configured to automatically apply a Hamming window to the buffer data before transformation, which smooths the results. In the simplest form, the data obtained can be visualized by mapping the number of frequency bands to the window width and the intensities of the bands to the window height, which resembles the approach used for the waveform visualization. The band intensities are not normalized. Therefore the maximum intenity has to be calculated first. Figure 21 shows an illustration of the result.

While the two visualizations described above show the audio data in its raw form and are thus „exact", they are not very useful in practice because they contain too much information. As a result, it is hard to perceive more than drastic changes in intensity when the visualizations are updated 30 times per second or more often, which is required in realtime visualization.[5]

A problem of the frequency band visualization introduced above is that it is not very meaningful because the intensities of the frequency bands do not directly correspond to human perception, as described in section 4.2.2. An improvement is to first take the logarithm of the band intensities and then accumulate the bins. The accumulation could be carried out logarithmically, which is supported by Minim's `FFT` class. Another approach that yields similar results is the calculation of a Mel scale from the frequency data. Visualizations based on this accumulated data correlate better with what users hear. They are also easier to follow as they display less information. Figure 26 shows a simple visualization based on a Mel scale.

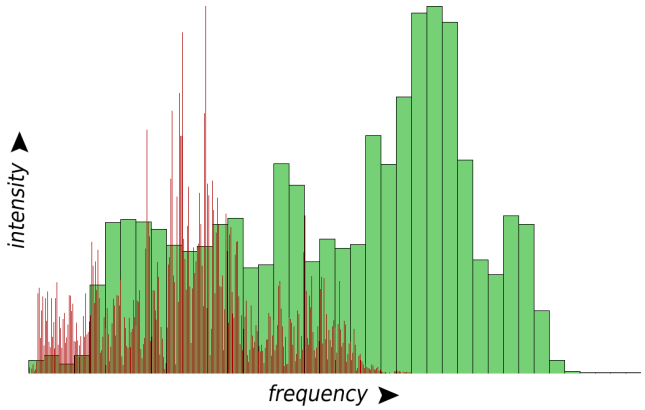

Figure 26: Visualization of the frequency distribution of a frame (red) and Mel scale visualization based on the same data (green).

Another useful visualization is the *spectrogram*, which shows how the frequency distribution of a signal varies with time. The spectrogram can reveal patterns which can be used for identification purposes. At the simplest form, a spectrogram has three dimensions: The horizontal axis represents time, the vertical axis frequency, and the color indicates the intensity of a given frequency band. This means a particular frequency at a particular time is represented by the color of each point in the visualization. Spectrograms of sinusoidal signals are shown in Figures 22 and 23. Figure 27 shows another example.

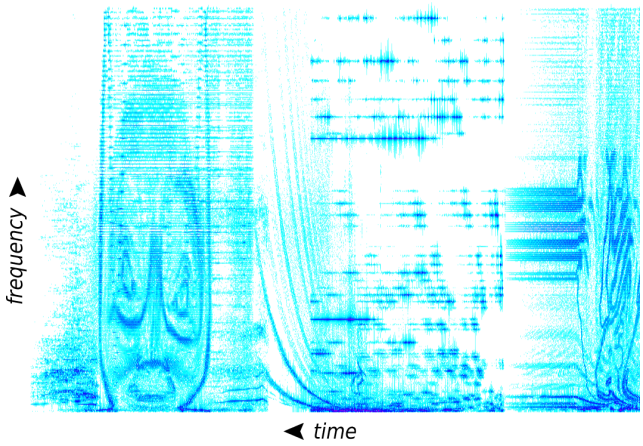

Figure 27: Spectrogram of a part of an experimental track by Aphex Twin, created with Processing. Darker areas indicate frequencies with higher intensities.

Implementation of a spectrogram visualization in Processing is not as straightforward as the visualizations described so far. Basically, a spectrogram is computed by mapping a counter that represents time to the horizontal position of the display window, the frequency bands to the height of the window and the intensity of the bands to a color. The problem is that data computed in former calls to `draw` are needed. An approach would be to use a collection such as a `LinkedList` that could hold the frequency data computed during the current and former calls to `draw`. One could then read the last w values from the collection (with w being the width of the display window) and use that data for drawing. Older values could be deleted from the collection. The problem with this approach is that it is very inefficient as up to $w \cdot n$ (n denotes the number of frequency bands) values have to be accessed and drawn every time `draw` is called.

The solution to this type of problems is based on the usage of so called off-screen buffers. In the simplest case, such a buffer is just an image (an object of the `PImage` class) whose pixel values are changed. Moreover, Processing provides the `PGraphics` class, an off-screen buffer that supports the same functions for drawing as Processing and can be displayed like an ordinary image.[6] By using a buffer, the spectrogram visualization can be implemented in an efficient way. First, a buffer with the width w and the height n is initialized, along with a counter c that is set to zero. Then, at every invocation of `draw`, c is incremented and frequency information based on the current audio frame is used to draw row c of the off-screen buffer as described above. Then, the buffer is displayed with Processing's `image` function.

This method works until $c > w$, at which point the buffer width is exceeded. The solution is to reset c to zero when this happens, that is to start a new iteration. With this approach, values from the current iteration overwrite values from the last iteration. Then, the old and new parts of the

[5] At a sample size of 44100 Hz and a sample size of 1024 practical frame rates are around 43 as $43 \cdot 1024 \approx 1000$ ms, which means that nearly the complete signal data can be used for visualization purposes.

[6] The `PGraphics` class does not work with Processing's OpenGL renderer. However, the GLGraphics library introduced in section 3.2 supports an OpenGL based off-screen buffer.

buffer are displayed in a shifted way, as illustrated in Figure 28. This can be done with the `copy` function, which allows users to display certain parts of an image.

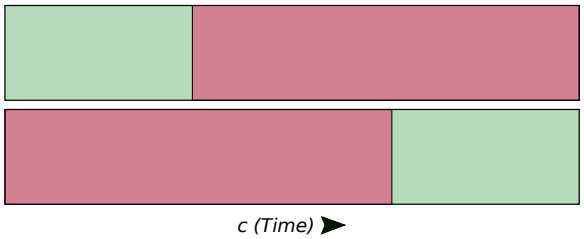

Figure 28: Illustration of off-screen buffer usage for the creation of the Spectrogram visualization. The top image illustrates the buffer. The green section is created with data from the current iteration, the red section is from the last iteration. The bottom image depicts how the buffer content is displayed.

4.3.3 Generative Visualizations

The visualizations described in the last section were based on raw frame data. This section discusses *generative visualizations*, that is visualizations that use information derived from the audio data to generate shapes and effects. The focus of these visualizations is to produce beautiful images, not to correspond to the perceived music as precisely as possible. Thus, they are often based on simple audio descriptors.

One approach to generative visualizations is to start with the frequency distribution of the audio data and modify and use this data in some abstract way. An example is Audio-driven Landscape by Robert Hodgin [Hodgin 2010], which creates a three dimensional landscape from the frequency distribution. For this purpose the frequency data is first smoothed and then used to create a landscape by mapping time to the x-axis, frequency to the z-axis and intensity of the frequency bands to the y-axis and color. This approach is also used to create three dimensional spectrograms. Figure 29 shows an image of the visualization.

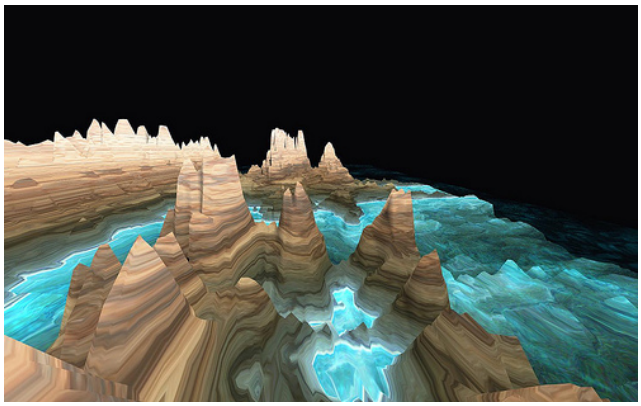

Figure 29: Audio-driven Landscape by Robert Hodgin. Image taken from [Hodgin 2010].

Another approach is to draw the frequency bands independently, as done in the Narratives visualization by Matthias Dittrich. In this visualization, frequency bands are drawn individually side by side, according to their frequency and intensity. Figure 30 shows an illustration. The Processing sketch is available on the project website [Dittrich 2009].

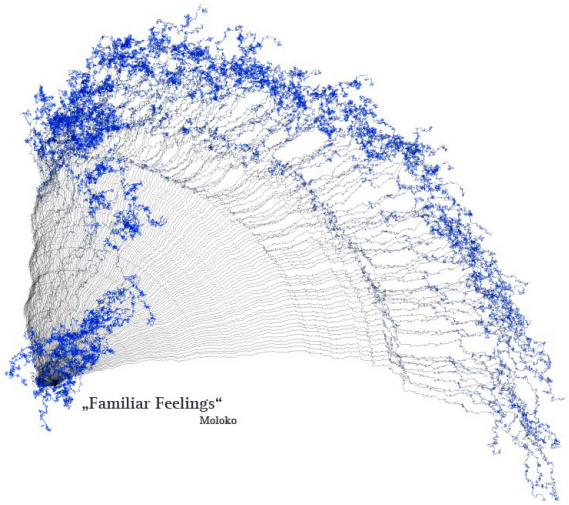

Figure 30: Visualization of the song Familiar Feelings by Moloko, created with Narratives. Illustration based on an image from [Dittrich 2009].

Other generative visualizations utilize simple descriptors such as RMS (equation 2) and beat-detection algorithms, which are often based on these descriptors.[7] An example is the Bubbles visualization, shown in Figure 31.

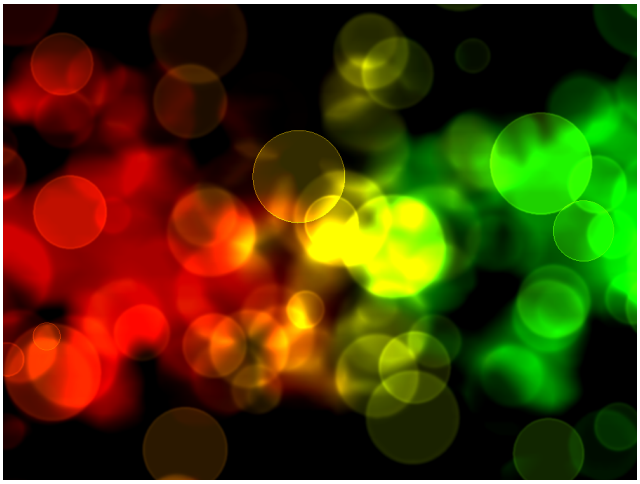

Figure 31: The Bubbles visualization. Bubbles are spawned according to the intensity of the music. Created with Processing [Pramerdorfer 2010].

The Bubbles visualization is described in more detail to illustrate some characteristics of this kind of visualizations. The visualization spawns bubbles based on the RMS descriptor, i.e., based on the intensity of the music played. The number and size of bubbles spawned is related to the RMS of the frame data. However, there is no strict mapping. Instead, the bubbles visualization relies on randomness in order to be able to produce different results every time, which makes

[7] Minim offers the `BeatDetect` class for this purpose.

it more exciting to watch. This is a characteristic many generative visualizations share.

The basics of the Bubbles visualization are as follows. At every invocation of the `draw` function a number of random values r_i in the interval $[0, 1]$ are calculated. Then, the RMS of the current audio frame is calculated, which is a value between 0 and 1. This value is transformed and then compared to every value r_i. If it is greater than r_i, a bubble is spawned. This means that the number of bubbles spawned not only depends on the intensity of the music played, but also on randomness. As a result, there is a very small chance that no bubbles are spawned even if the calculated RMS is nearly 1. In practice, this makes the visualization more interesting to watch as it makes it less predictable while the correlation between sound intensity and bubble spawn rate is still apparent. The spawn positions are also chosen randomly. For example, the vertical position is calculated based on a Gaussian distribution with the mean at the center of the display window. The variance is chosen depending on the window height so that the bubbles are spawned over the total available area, with an emphasis on the center.

When a bubble is spawned, it grows in size over a few frames, then it stays this size and becomes blurred until it disappears. While these effects can be done with Processing's `filter` function, it is inflexible and slow. Therefore, these effects are implemented as GLSL shaders by utilizing the GLGraphics library, which was introduced in section 3.2. The rate at which bubbles disappear is related to the number of new bubbles spawned recently. This adaptive behavior is important, in the case of the Bubbles visualization it ensures that the bubbles do not flood the display window. The source code of the Bubbles visualization is available on the project website of Processing Audio Visualization [Pramerdorfer 2010] together with other visualizations created with Processing.

Robert Hodgin is the author of complex generative visualizations like Solar, which is shown in Figure 32. Solar is based on black spheres. Each sphere relates to a certain frequency band and uses its intensity to adjust its mass and charge, i.e., Solar integrates physics. These physical attributes pull the spheres together or apart, depending on the frequency distribution of the audio data. The lighting and particle effects are also based on this information. Videos of Solar are available at [Hodgin 2010]. Another visualization by Hodgin, called Magnetosphere, is the default iTunes visualizer at the time of writing.

The possibilities for generative visualizations are limited only by the creativity of the designers. For instance, designers could use more complex descriptors like MFCCs to model forces that could act on geometry. Processing and libraries created by the community provide users with the necessary tools to create complex visualizations in an intuitive way.

Conclusion

This paper provides an introduction to Processing. The language differences to Java, the basic functionality Processing provides as well as development environments are discussed. An overview of the Processing community and selected projects are presented. The second part of this paper deals with music visualization, one of many application areas of Processing. After discussing how music can be accessed with the computer and how information can be derived from audio data streams, approaches to use this information to create visualizations with the help of Processing as well as projects by the community are introduced.

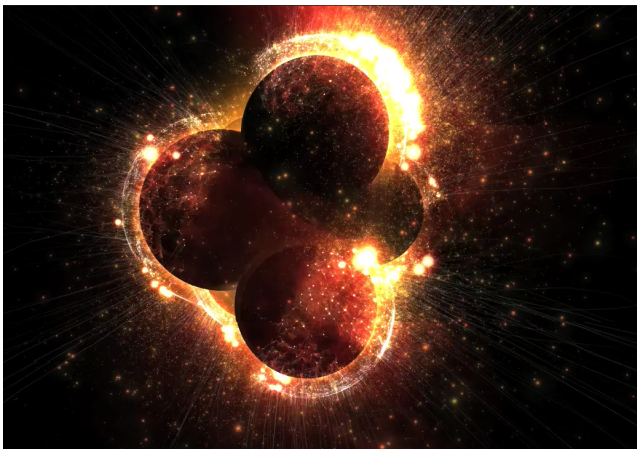

Figure 32: Solar by Robert Hodgin, created with Processing. Image based on a video from [Hodgin 2010].

References

Arduino, 2010. The official arduino website. `http://www.arduino.cc/` (retrieved on 2011-01-16).

Bohnacker, H., Gross, B., and Laub, J. 2009. *Generative Gestaltung: Entwerfen. Programmieren. Visualisieren.*, 1 ed. Schmidt Hermann Verlag.

Colubri, 2010. Website of andres colubri. `http://codeanticode.wordpress.com/` (retrieved on 2011-01-16).

Columbia, 2010. Project website of computing kaizen. `http://proxyarch.com/kaizen/` (retrieved on 2011-01-16).

Daubechies, I. 2002. The wavelet transform, time-frequency localization and signal analysis. *IEEE Transactions on Information Theory 36*, 5, 961–1005.

Diniz, P., da Silva, E. A., and Netto, S. L. 2010. *Digital Signal Processing: System Analysis and Design*, 2 ed. Cambridge University Press.

Dittrich, M., 2009. Project website of narratives. `http://matthiasdittrich.com/projekte/narratives/visualisation/index.html` (retrieved on 2011-01-16).

Eclipse, 2010. The official eclipse website. `http://eclipse.org/` (retrieved on 2011-01-16).

Fede, D. D., 2010. The official minim website. `http://code.compartmental.net/tools/minim/` (retrieved on 2011-01-16).

Fourney, D., and Fels, D. 2009. Creating access to music through visualization. In *Science and Technology for Humanity (TIC-STH), 2009 IEEE Toronto International Conference*, IEEE, 939–944.

Fry, B. 2008. *Visualizing Data: Exploring and Explaining Data with the Processing Environment*, 1 ed. O'Reilly Media.

Fry, B., 2010. Website of the processing eclipse plugin. http://wiki.processing.org/w/Eclipse_Plug_In (retrieved on 2011-01-16).

Glassner, A. 2010. *Processing for Visual Artists: How to Create Expressive Images and Interactive Art*. A K Peters Ltd.

GLGraphics, 2010. Project website of the glgraphics library. http://glgraphics.sourceforge.net/ (retrieved on 2011-01-16).

Greenberg, I. 2007. *Processing: Creative Coding and Computational Art*, 1 ed. friends of ED.

HackNMod, 2010. Top 40 arduino projects of the web. http://hacknmod.com/hack/top-40-arduino-projects-of-the-web/ (retrieved on 2011-01-16).

Harrington, J., and Cassidy, S. 1999. *Techniques in Speech Acoustics*. Springer-Verlag New York, Inc.

Hobley, S., 2010. Project website of laser harp. http://www.stephenhobley.com/blog/laser-harp-2009/ (retrieved on 2011-01-16).

Hodgin, 2010. Website of robert hodgin. http://www.flight404.com/ (retrieved on 2011-01-16).

Logan, B. 2000. Mel frequency cepstral coefficients for music modeling. In *International Symposium on Music Information Retrieval*, vol. 28.

Lu, L., Liu, D., and Zhang, H. 2005. Automatic mood detection and tracking of music audio signals. *IEEE Transactions on Audio, Speech, and Language Processing 14*, 1, 5–18.

Medialab-Prado, 2008. Project website of in the air. http://intheair.es/ (retrieved on 2011-01-16).

Noble, J. 2009. *Programming Interactivity: A Designer's Guide to Processing, Arduino, and openframeworks*, 1 ed. O'Reilly Media.

O'Meara, A., 2010. Official website of the g-force music visualization. http://www.soundspectrum.com/g-force/ (retrieved on 2011-01-16).

OpenProcessing, 2010. The openprocessing website. http://openprocessing.org/ (retrieved on 2011-01-16).

Oracle, 2010. The official java website. http://java.com/en/ (retrieved on 2011-01-16).

Plack, C. J. 2005. *The Sense of Hearing*, 1 ed. Psychology Press.

Polotti, P., and Rocchesso, D. 2008. *Sound to Sense, Sense to Sound: A state of the art in Sound and Music Computing*. Logos Berlin.

Pramerdorfer, C., 2010. Website of processing audio visualization. http://web.student.tuwien.ac.at/~e0626747/pav/ (retrieved on 2011-01-16).

Processing, 2010. The official processing website. http://processing.org/ (retrieved on 2011-01-16).

ProcessingAndroid, 2010. The official processing android website. http://wiki.processing.org/w/Android (retrieved on 2011-01-16).

ProcessingJS, 2010. The official processing.js website. http://processingjs.org/ (retrieved on 2011-01-16).

Reas, C., and Fry, B. 2007. *Processing: A Programming Handbook for Visual Designers and Artists*. The MIT Press.

Reas, C., and Fry, B. 2010. *Getting Started with Processing*, 1 ed. Make.

Rost, R. J., Licea-Kane, B., Ginsburg, D., Kessenich, J. M., Lichtenbelt, B., Malan, H., and Weiblen, M. 2009. *OpenGL Shading Language*, 3 ed. Addison-Wesley Professional.

Sabin, J. E., and Jones, P. L., 2008. Project website of branching morphogenesis. http://sabin-jones.com/arselectronica.html (retrieved on 2011-01-16).

Saito, S., Kameoka, H., Takahashi, K., Nishimoto, T., and Sagayama, S. 2008. Specmurt analysis of polyphonic music signals. *IEEE Transactions on Audio, Speech, and Language Processing 16*, 3, 639–650.

Schubert, E., Wolfe, J., and Tarnopolsky, A. 2004. Spectral centroid and timbre in complex, multiple instrumental textures. In *Proceedings of the International Conference on Music Perception and Cognition, North Western University, Illinois*, 112–116.

Shiffman, D. 2008. *Learning Processing: A Beginner's Guide to Programming Images, Animation, and Interaction*. Morgan Kaufmann.

Terzidis, K. 2009. *Algorithms for Visual Design Using the Processing Language*. Wiley.

Tzanetakis, G., Essl, G., and Cook, P. 2001. Audio analysis using the discrete wavelet transform. In *Proceedings of the Conference in Acoustics and Music Theory Applications*.

Ware, C. 2004. *Information Visualization*, 2 ed. Morgan Kaufmann.

Wiring, 2010. The official wiring website. http://wiring.org.co/ (retrieved on 2011-01-16).