

CodeDraw

A Graphics Library for Novice Programmers

BACHELOR'S THESIS

submitted in partial fulfillment of the requirements for the degree of

Bachelor of Science

in

Software and Information Engineering

by

Niklas Kraßnig

Registration Number 11706718

to the Faculty of Informatics

at the TU Wien

Advisor: Dr. techn. Stefan Podlipnig

Vienna, 1st October, 2022

Niklas Kraßnig

Stefan Podlipnig

Abstract

Many computer science educators incorporate graphics into their CS1 courses, however, there are only a handful of graphics libraries in plain Java that are suitable for novice programmers. Those available lack features or documentation, hence, we developed CodeDraw. CodeDraw is a new 2D graphics library specifically created for beginners that allows the creation of images, animations and even interactive programs. This thesis discusses in depth the features and design decisions of this library and compares CodeDraw with similar libraries using various examples.

Danksagung

Vorerst möchte ich mich bei allen Beteiligten der Lehrveranstaltung Einführung in die Programmierung 1 bedanken. Das Feedback von Studierenden und Lehrenden hat viel zur Entwicklung beigetragen. Vielen Dank an den Lehrveranstaltungsleiter und meinen Betreuer, Stefan Podlipnig, der mir gegenüber das Vertrauen aufgebracht hat, diese Bibliothek einzusetzen und mich beim Schreiben dieser Arbeit unterstützt hat. Besonderer Dank geht auch an Nikolaus Kasyan, der Teile der Bibliothek entwickelt hat und mich bei dem Design der Bibliothek immer eifrig unterstützt hat. Des Weiteren möchte ich mich bei Konstantin Lackner für die Idee des InstantDraw Modus bedanken, der es den Studierenden wesentlich leichter macht, ihre Anwendungen zu debuggen. Bei ihm und Florentina Meister bedanken ich mich auch vielmals dafür, dass sie die Arbeit korrigiert und verständlicher gemacht haben. Danke, dass ihr mir geholfen habt, CodeDraw möglich zu machen.

Contents

| | |
|--|------------|
| Abstract | iii |
| Contents | vii |
| 1 Introduction | 1 |
| 1.1 What is CodeDraw | 1 |
| 1.2 What CodeDraw is not | 2 |
| 1.3 About this Thesis | 2 |
| 2 CodeDraw Features | 3 |
| 2.1 Drawing Shapes | 3 |
| 2.2 Animation | 4 |
| 2.3 The Palette | 4 |
| 2.4 Images | 5 |
| 2.5 Transparency | 6 |
| 2.6 Image Editing | 6 |
| 2.7 Display Options | 7 |
| 2.8 Debugging CodeDraw | 8 |
| 2.9 Linear Transformations | 8 |
| 2.10 High DPI Support | 10 |
| 2.11 Text Formatting | 11 |
| 2.12 Custom Cursor | 12 |
| 2.13 Canvas and Window Position | 12 |
| 2.14 Events | 13 |
| 2.14.1 Event Listener (v1) | 13 |
| 2.14.2 EventScanner (v2) | 14 |
| 2.14.3 Enhanced EventScanner (v3) | 15 |
| 2.14.4 Game Development with the EventScanner | 16 |
| 2.14.5 Inversion of Control (v3) | 19 |
| 2.15 Paths | 21 |
| 3 CodeDraw at the Vienna University of Technology | 23 |
| 3.1 Programming Courses at the Vienna University of Technology | 23 |

| | | |
|----------|---|-----------|
| 3.2 | Why develop CodeDraw | 24 |
| 3.3 | How CodeDraw is used in Introduction to Programming 1 | 24 |
| 4 | Learning Programming Concepts through CodeDraw | 27 |
| 4.1 | Objects as a Resource | 27 |
| 4.2 | Show is Slow | 27 |
| 4.3 | Data Composition, Enumeration Types and Immutability | 28 |
| 4.4 | Chapter Conclusion | 28 |
| 5 | Design Comparison and Related Work | 29 |
| 5.1 | Orientation of the Pixel Grid | 30 |
| 5.2 | Origin of Shapes | 30 |
| 5.3 | Inversion of Control | 30 |
| 5.4 | Double Buffer vs. Single Buffer | 31 |
| 5.5 | Angles and Rotation | 32 |
| 5.6 | Static Image Comparison | 33 |
| 5.7 | Animation Comparison | 35 |
| 5.8 | Interactive Program Comparison | 40 |
| 5.8.1 | ObjectDraw | 41 |
| 5.8.2 | Java AWT | 42 |
| 5.8.3 | StdDraw | 43 |
| 5.8.4 | Processing | 44 |
| 5.8.5 | CodeDraw Animation | 45 |
| 5.8.6 | CodeDraw | 46 |
| 6 | Conclusion | 47 |
| | List of Figures | 49 |
| | Bibliography | 51 |

Introduction


Many courses that teach programming incorporate computer graphics to provide students with more engaging exercises. At the Vienna University of Technology we use Java to teach students programming but found that, when it comes to plain Java, there are not many options for beginner graphics libraries.


Hence, we developed a graphics library named CodeDraw. It is specifically aimed at first semester students and can be used to create images, animations and interactive programs. The entire library is documented and the documentation, source code, binaries and most current version can be found in the CodeDraw repository.¹

After this introduction, the second chapter of this thesis describes the features and capabilities of the CodeDraw library. Alternatively, the *Introduction to CodeDraw* in the CodeDraw repository can be read to get a shorter and more surface level overview of the features of CodeDraw.² The third chapter looks at how the Vienna University of Technologies teaches its courses and how that set the prerequisites for this library. The fourth chapter discusses different programming concepts present in the CodeDraw library. Oftentimes concepts like these are taught through constructed exercises, whereas with this library, they're acquired incidentally while producing graphical output. The last chapter compares the design decision of the CodeDraw library to other educational libraries and talks about the design decisions of the library in general.

1.1 What is CodeDraw

CodeDraw is a 2D Java graphics library created for introductory programming. Its main design goal is to be easy to learn for novice programmers, while avoiding to teach any

¹CodeDraw repository  <https://github.com/Krassnig/CodeDraw>

²Introduction to CodeDraw v3  https://github.com/Krassnig/CodeDraw/tree/release_v3.0.0/INTRODUCTION.md

software anti-patterns. It provides a 2D pixel grid on which different shapes and images can be drawn. The drawn objects are then displayed in a window.

There are two ways to write CodeDraw programs. The first is procedural and does not require writing functions or classes. The second way is using an `Animation` interface where a `draw(Image canvas)` method has to be implemented. This draw method is then called in regular intervals by the library.

CodeDraw is built on top of Swing/AWT. It hides all the complexities of Swing/AWT and avoids exposing any Swing/AWT methods and classes. It also hides the AWT thread. When users work with CodeDraw they never have to use any other threads than the main thread. All concurrency is handled by the library.

1.2 What CodeDraw is not

CodeDraw does not support graphical user interface elements such as buttons or menus. While other libraries like ObjectDraw [BDM01] create and modify their shapes through objects, CodeDraw can only draw shapes onto a canvas through methods. Once shapes have been drawn in CodeDraw, they can no longer be accessed, modified or moved. Furthermore, CodeDraw is neither a 3D graphics library nor a full game engine. You can use CodeDraw to develop games, but for larger games it would probably be better to choose a proper game engine.

1.3 About this Thesis

This thesis describes version 3.0.0 of CodeDraw unless stated otherwise.³ All versions of CodeDraw as well as many examples from this thesis are available on the Github repository.⁴ CodeDraw uses semantic versioning.⁵

All of the graphical results of the CodeDraw library are produced through the Java AWT library. Many of the design decisions of CodeDraw were inspired by StdDraw, Java AWTs Graphics2D class, Processing and HTMLs Canvas graphics. The focus of the CodeDraw project is to make graphics as easy and accessible as possible for beginners.

All links in this thesis were last accessed on fourth of October 2022. Additionally, each link can be accessed on the Wayback Machine by clicking on the *WBM* letters in front of each link.

³CodeDraw version 3 [WBM https://github.com/Krassnig/CodeDraw/tree/release_v3.0.0](https://github.com/Krassnig/CodeDraw/tree/release_v3.0.0)

⁴CodeDraw repository [WBM https://github.com/Krassnig/CodeDraw](https://github.com/Krassnig/CodeDraw)

⁵Semantic versioning [WBM https://semver.org/](https://semver.org/)

CodeDraw Features

2.1 Drawing Shapes

CodeDraw provides a window with a 2D pixel grid that is modeled as an object. As illustrated in figure 2.1, shapes can be drawn onto the pixel grid by calling different draw methods. In line 1 the size of the pixel grid is specified through the `CodeDraw` constructor. Once the size is set, it cannot be changed. After an instance of the `CodeDraw` window has been created, one can choose to draw from a variety of shapes. For example, in line 2 the outline of a rectangle with a width of 200px and a height of 100px is drawn. The top left corner of the rectangle will be 100px below and 100px to the right of the top left corner of the canvas. To draw filled shapes as can be seen in line 3 the `fill` prefix is used instead.

```
1 CodeDraw cd = new CodeDraw(500, 500);
2 cd.drawRectangle(100, 100, 200, 100);
3 cd.fillRectangle(100, 300, 200, 100);
4 cd.setColor(Palette.ORANGE);
5 cd.fillCircle(200, 200, 50);
6 cd.show();
```

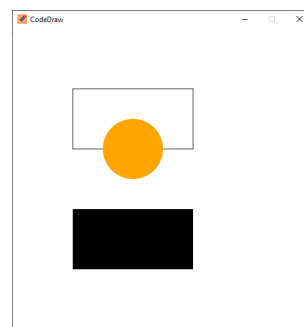


Figure 2.1: An example of how to draw basic shapes using CodeDraw.

The state of the `CodeDraw` object represents its drawing configuration. By changing the state, the way shapes are drawn is changed. The color of all succeeding shapes can be set through `setColor(Color color)`. In line 4 the color is changed and, therefore, the circle on line 5 is orange.

Lastly, in line 6, the `show()` method has to be called to display everything that has been drawn into the buffer until now.

2.2 Animation

Animations are implemented by calling `show()` inside a loop. In each iteration a different image is drawn. For example, a clock that ticks every second uses an endless loop that waits for one second after drawing twelve dots and the clock's second hand. In order to pause for one second between each frame the `show(long waitMilliseconds)` method has to be called. Before drawing a new frame the entire canvas needs to be cleared by calling the `clear()` method.

```
CodeDraw cd = new CodeDraw(400, 400);
for (double sec = -Math.PI / 2; !cd.isClosed(); sec += Math.PI / 30) {
    cd.clear();
    cd.drawLine(200, 200, Math.cos(sec)*100+200, Math.sin(sec)*100+200);
    for (double j = 0; j < Math.PI * 2; j += Math.PI / 6) {
        cd.fillCircle(Math.cos(j) * 100 + 200, Math.sin(j) * 100 + 200, 4);
    }
    cd.show(1000);
}
```

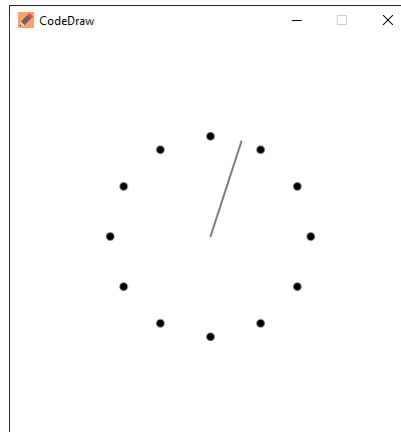


Figure 2.2: One frame of the clock animation in CodeDraw after 3 seconds.

2.3 The Palette

Java AWT only offers 13 predefined colors in its color class.¹ In contrast, CodeDraw offers around 140 colors through the `Palette` class which allows the user to pick from a larger pool of colors. The color names were taken from the predefined CSS colors.²

¹AWT color class [WBM https://docs.oracle.com/javase/9/docs/api/java/awt/Color.html](https://docs.oracle.com/javase/9/docs/api/java/awt/Color.html)

²List of CSS colors [WBM https://www.w3.org/TR/2022/REC-css-color-3-20220118/#svg-color](https://www.w3.org/TR/2022/REC-css-color-3-20220118/#svg-color)

To create custom colors the user can either use the `Color` class itself or call the color creation methods of the `Palette` class as seen in the list below.

- `Palette.fromRGB(int red, int green, int blue)` creates a color from the red, green and blue primary colors.
- `Palette.fromRGBA(int red, int green, int blue, int transparency)` additionally allows to set the transparency value.
- `Palette.fromHSV(int hue, int saturation, int brightness)` allows the creation of colors from the HSV model.
- `Palette.randomColor()` creates a random color that is not transparent.
- `Palette.fromGrayscale(int grayscale)` creates a color where all three color components have the same value.
- `Palette.fromBaseColor(Color color, int transparency)` creates a new color based on the base color but with the transparency value changed.

Java AWT's `Color` class was the only class that was not wrapped by `CodeDraw`. The reason for that is that in Introduction to Programming 1 we generally recommend IntelliJ to our students and IntelliJ has a feature where it displays the color of a `Color` variable, but does not do so for custom classes. Thus, it becomes very hard to tell what the color based on the name alone will look like.

2.4 Images

`CodeDraw` has an `Image` class that is very similar to the `CodeDraw` class but does not include the graphical user interface. The `CodeDraw` class actually extends the `Image` class to implement its drawing capabilities. Additionally, the `Image` class offers an easy way to load images from the file system without users having to handle exceptions. For example, the code below creates a window with the image exactly wrapping around the window.

```
Image myImage = Image.fromFile("path/to/image.png");
CodeDraw cd = new CodeDraw(myImage.getWidth(), myImage.getHeight());
cd.drawImage(0, 0, myImage);
cd.show();
```

There are other static methods for creating images. `Image.fromUrl(String)` loads an image from the Internet. `Image.fromResource(String)` loads it from the resource folder. `Image.fromBase64String(String)` creates images from images that are Base64 encoded strings. To create a plain white image the constructor can be called with the

```
Image testImage = new Image(100, 100);
testImage.fillCircle(50, 50, 50);
Image.save(testImage, "path/to/image.png", ImageFormat.PNG);
```

Figure 2.3: How to save a PNG image to the file system.

size specified through the width and height parameter `new Image(int width, int height)`. If another color should be used, the background color can be given as an additional parameter to the constructor `new Image(int, int, Color)`.

Also, images can be saved to the file system by calling the static method `Image.save(Image image, String path, ImageFormat format)`.

2.5 Transparency

To create a transparent image the background color in the constructor can be specified as `Palette.TRANSPARENT`. If an image file is already transparent, it will also be transparent once opened with CodeDraw.

The *draw over* property defines whether transparent shapes and images are drawn over existing pixels or whether they replace the pixels that are already present. Per default or by calling `setDrawOver(true)` every drawing operation will draw over what is already there. This means that a 50% transparent red square that is drawn over a blue background will appear purple, the colors will mix. This would not set the pixel values to transparent values since the background will always show through the drawn pixel. By calling `setDrawOver(false)` the drawing behavior is changed to replace pixels. The 50% transparent red square on the blue background will no longer be purple but instead be transparent red, since the pixel values get replaced by 50% transparent red.

Since the CodeDraw window is not see-through, once an image is drawn onto a canvas, the background will be made white and the transparent image will be drawn over the white background. In the example above, this will result in the the 50% transparent red turning into light red when displayed.

2.6 Image Editing

Basic image editing operations are available as static methods such as `crop`, `scale`, `rotate` and `mirror`.

In figure 2.4 line 2 cuts out a rectangle from `myImage` at the (100, 100) coordinate with a width of 200 pixels and a height of 100 pixels. The width and height of the cropped image will be 200 by 100 pixels. In line 3 the `rotateClockwise()` method rotates the cropped image clockwise by 90°, which would give it a size of 100 by 200 pixels. Line 4 mirrors the image along the vertical axis.

```

1 Image myImage = Image.fromFile("plant.png");
2 Image cropped = Image.crop(myImage, 100, 100, 200, 100);
3 Image rotated = Image.rotateClockwise(cropped);
4 Image mirrored = Image.mirrorVertically(rotated);

```

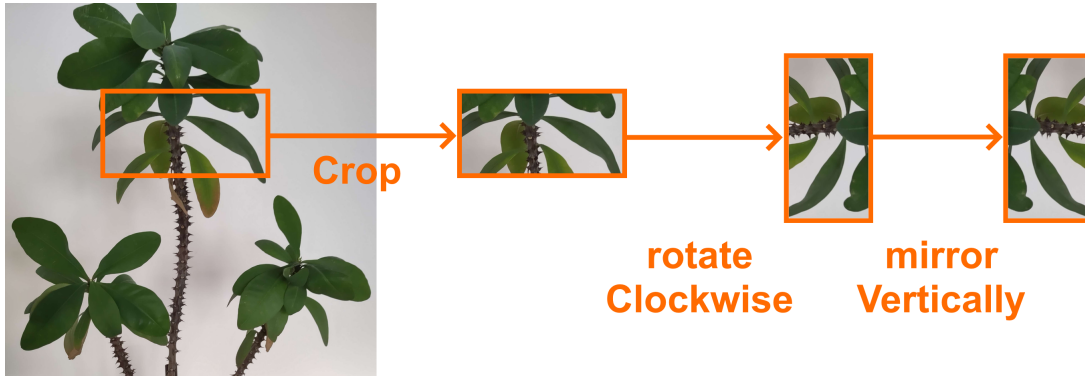


Figure 2.4: An image of a plant being edited using CodeDraw. This image was taken by Nikolaus Kasyan.

2.7 Display Options

CodeDraw has three window classes `CodeDraw`, `BorderlessWindow` and `FullScreen`. `CodeDraw` is the default style similar to most application windows with a title, minimize button and close button. `BorderlessWindow` is just the canvas without any surrounding user interface. The `FullScreen` class covers the whole screen. Since its size is dependent on the screen the width and height is determined by the screen resolution.

```

FullScreen fs = new FullScreen();
fs.drawText(100, 100, "Hello World!");
fs.show();

```

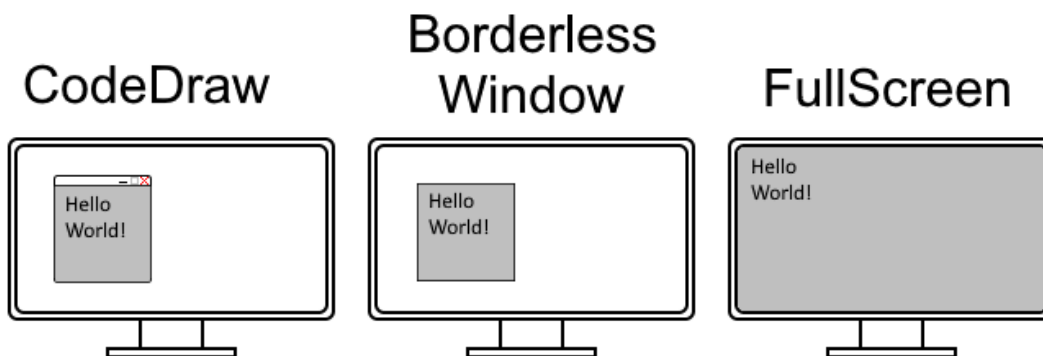


Figure 2.5: Different display options in CodeDraw.

All three inherit from the `Image` class and all provide the same drawing capabilities. The difference lies in their available graphical user interface methods. For example, the position of the `FullScreen` window cannot be changed and therefore does not have `setWindowPosition()` and `setCanvasPosition()` methods.

2.8 Debugging CodeDraw

When a program with CodeDraw reaches a breakpoint, that breakpoint pauses all threads per default, this also blocks the Java AWT thread running in the background and freezes the CodeDraw window. To avoid this behavior, the debugger settings need to be changed to only block the main thread. In IntelliJ this can be done by right clicking a breakpoint and selecting the *Thread* option.

A common issue often encountered by user while debugging is the inability to tell what has been drawn since the last `show()` command was called, as CodeDraw only displays changes after calling `show()`.³ One unpractical solution is to put `show()` after every drawing command. To alleviate this issue, CodeDraw has an instant draw mode that can be activated by setting `setInstantDraw(boolean)` to `true`. When instant draw is activated, each draw operation instantly displays what has been drawn and also guarantees that every time a draw method returns, the change will already be displayed on the canvas. However, the instant draw mode has the drawback of being very slow.

To make debugging even more convenient there is a `setAlwaysOnTop(boolean)` property. Enabling this always displays the CodeDraw window on top of every other window including the IDE. Users can then switch between the CodeDraw window and their IDE without the window hiding behind their IDE.

2.9 Linear Transformations

Drawn shapes can be transformed before they are drawn by setting a specific linear transformation through `setTransformation(Matrix2D transformation)`. The transformation will then be applied each time before drawing a shape. The multiplication method in the `Matrix2D` class applies the transformation, represented by these matrices, right to left as per usual.

In contrast, all none multiplication methods in the `Matrix2D` class apply the transformation left to right. For example, when the `rotate()` method is called on some matrix M , a rotational matrix R would be created and then multiplied with matrix M . However, in the `rotate` method, this multiplication is calculated using $R \cdot M$, not $M \cdot R$. This way of implementing the matrix methods should be more intuitive since the operations are applied in the same direction English is read in. So if the user writes `Matrix2D.IDENTITY.translate(100, 200).rotate(Math.PI / 4);` the translation would be applied first, then the rotation. However, if the user were to write a

³The reason behind having to call `show()` each time is described in section 5.4.

similar looking program like in figure 2.6, the rotation would be applied first and then the translation.

```
Matrix2D t = Matrix2D.IDENTITY.translate(100, 200);
Matrix2D r = Matrix2D.IDENTITY.rotate(Math.PI / 2);
Matrix2D m2 = t.multiply(r);
```

Figure 2.6: An example of matrix multiplication.

The standard Matrix operations are implemented as methods, e.g. `translate(double dx, double dy)`, `rotate(double angle)`, `scale(double x, double y)` and additionally, there is also a `shear(double shearX, double shearY)` and a `mirror(double angle)` method. Each of those methods has an *at* variant that lets users specify a location where the operation is applied. For example, `rotate(double angle)` rotates the coordinate system around the (0, 0) coordinate, but `rotateAt(double x, double y, double angle)` lets users rotate the coordinate system around any arbitrary point.

```
1 CodeDraw cd = new CodeDraw(200, 200);
2 cd.setTransformation(Matrix2D.IDENTITY.rotateAt(100, 100, Math.PI / 4));
3 cd.getTextFormat().setTextOrigin(TextOrigin.CENTER);
4 cd.drawText(100, 100, "Hello World!");
5 cd.show();
```

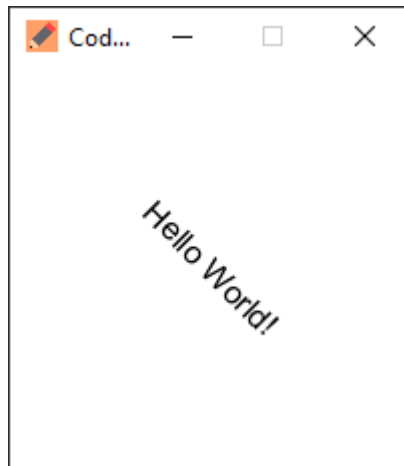


Figure 2.7: Rotating text by 45° around the (100, 100) coordinate in CodeDraw.

Figure 2.7 illustrates such a rotation where the text “Hello World!” is tilted clockwise 45 degrees at the center of the canvas instead of the (0, 0) coordinate. Line 3 changes the text origin from the top left to the center of the drawn text.

Like all state variables in CodeDraw, this modifies all subsequently drawn shapes. If a rectangle would be drawn afterwards, it would also be rotated by 45 degrees around the (100, 100) coordinate. To reset this behavior, the user can either specify the identity matrix `cd.setTransformation(Matrix2D.IDENTITY)` as a transformation or

call `cd.setTransformationToIdentity()`. The identity matrix does not apply any transformations and represents the default value.

The matrix transformation can also be used to change the pixel scale of CodeDraw to a percent based scale, where the range is 0 to 1 and the center of the canvas is always 0.5. In figure 2.8 a circle is drawn at the center of the canvas, with the circle radius being $600 * 0.1 = 60$ pixels.

```
CodeDraw cd = new CodeDraw();
cd.setTransformation(
    Matrix2D.IDENTITY.scale(cd.getWidth(), cd.getHeight())
);
cd.fillCircle(0.5, 0.5, 0.1);
cd.show();
```

Figure 2.8: Using linear transformation to create a coordinate system that goes from 0 to 1.

Changing the origin point to the bottom left is also theoretically possible with matrix transformations, by translating and rotating the canvas. The problem with that being that the text and images are then turned upside down and rectangular shapes are drawn from the bottom left.

2.10 High DPI Support

DPI stands for *dots per inch* and describes how many pixels are on a screen relative to its size.⁴ A high DPI-device is a device that has a high screen resolution compared to its screen size and to display graphical user interfaces on such devices, applications are usually upscaled. Instead of one pixel being one pixel it might be upscaled to 1.25 pixels. If done properly, applications will not appear tiny but instead appropriate for their screen size. If upscaling is done improperly, the application will be larger but the text and lines will look blurry.

Java AWT automatically supports high DPI rendering by making the window large enough to fit the scaling factor. AWT does this by automatically upscaling its canvas, but not the one CodeDraw uses. The pixels are then interpolated to achieve the upscaled resolution, but none of the interpolation options is ideal for every scenario.

Instead, CodeDraw *secretly* upscales the resolution of the canvas users draw on. To ensure that the upscaling factor is large enough even when the CodeDraw window is moved between different screens, the largest upscaling factor of all screens is taken. This upscaling factor is then rounded to the next largest integer, so a scaling factor of 1.25 gets rounded to 2. The numerical values of the pixel grid as viewed from the outside remain the same, but on the inside the resolution is doubled. If a user specifies to draw a

⁴DPI can also be described with PPI or pixels per inch.

dot at the 10.5 coordinate and CodeDraw has an upscaling factor of 2, the pixel position of the dot gets translated to the 21st pixel.

Once the image is given to AWT to display, it is actually scaled down using a bi-cubic interpolation. This gives much better results than upscaling from a lower resolution.

2.11 Text Formatting

In CodeDraw text can be formatted by calling the `getTextFormat()` method on an instance of the CodeDraw class. The `TextFormat` class is a simple data object where the user can set the font-family, font-size, boldness, italic, underline and strike-through. Additionally, there is the `TextOrigin` property which specifies where the text should be drawn relative to the specified point in the `drawText(int x, int y, String text)` method. Per default Java AWT ignores newlines, but the `drawText()` method in CodeDraw supports new lines and breaks the text.

```
CodeDraw cd = new CodeDraw(500, 100);
TextFormat tf = cd.getTextFormat();
tf.setTextOrigin(TextOrigin.CENTER);
tf.setFontSize(25);
tf.setItalic(true);
cd.drawText(
    cd.getWidth() / 2.0,
    cd.getHeight() / 2.0,
    "Italic text at the center of the canvas!"
);
cd.show();
```

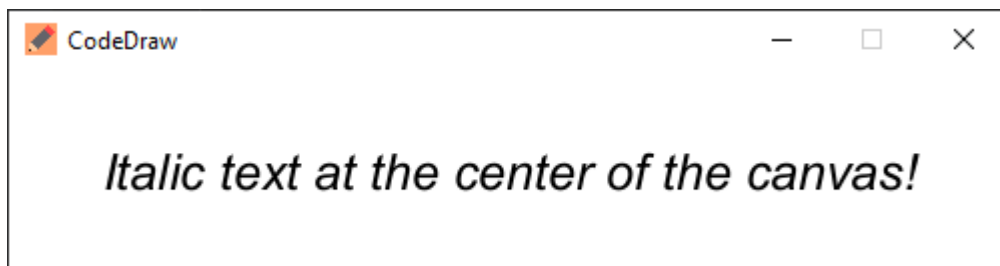


Figure 2.9: An example of how text can be stylized in the CodeDraw library.

Although Java AWT seems to offer more formatting options, they do not change the output. For example, setting a different font weight only changes between normal and bold at a certain numerical threshold.

This feature was developed by Nikolaus Kasyan in cooperation with the author of this thesis.

2.12 Custom Cursor

The cursor can be replaced or changed by calling `setCursorStyle(CursorStyle)` on a `CodeDraw` object. The `CursorStyle` class provides a list of common cursors. Alternatively, users can create their own custom cursor by providing an image and optionally defining the click position of the cursor on that image. Otherwise, the (0, 0) pixel of the image is taken as the click position.

```
CodeDraw cd = new CodeDraw();
cd.setCursorStyle(CursorStyle.WAIT);
```

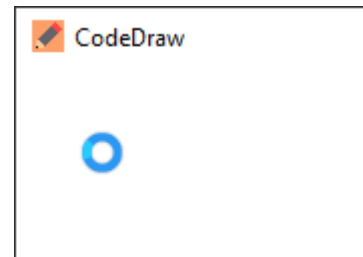


Figure 2.10: CodeDraw with a loading cursor.

This feature was developed by Nikolaus Kasyan in cooperation with the author of this thesis.

2.13 Canvas and Window Position

In the `CodeDraw` class both the canvas position and the window position can be changed by calling `setCanvasPositionX(double)` or `setWindowPositionX(double)`. The x and y component of the position must be accessed separately by calling their respective getters and setters, to avoid the use of objects.

```
CodeDraw cd = new CodeDraw();
cd.setWindowPositionX(100);
cd.setWindowPositionY(100);
```

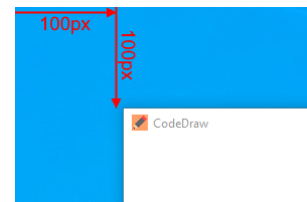


Figure 2.11: A `CodeDraw` window positioned 100 pixel away from the top left corner of the main screen.

The code in figure 2.11 places the window 100 pixel below and to the right of the main screen. There is a small mismatch of around 8 pixel that is caused by the shadow of the window.

The `FullScreen` class and the `BorderlessWindow` class also have window positions but these are more restricted. The `FullScreen` class has getters for the window position only because the `FullScreen` window is permanently attached to one screen that cannot

```
CodeDraw cd = new CodeDraw();
cd.setCanvasPositionX(100);
cd.setCanvasPositionY(100);
```

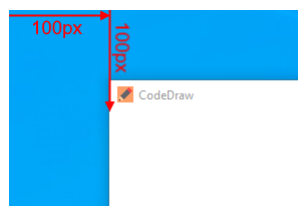


Figure 2.12: A CodeDraw window positioned so that the (0, 0) coordinate of the canvas is 100 pixels away from the top left corner of the main screen.

be changed. The `BorderlessWindow` has a changeable window position but no canvas position since the canvas position would be the same as the window position.

2.14 Events

There are two ways events can be handled in CodeDraw version 3. The first option is the `EventScanner` that can be queried similarly to the `Scanner`. The second option is the `Animation` interface where the event methods are overwritten and their implementation updates the state of the application. The `Animation` interface is only available in version 3, while the `EventScanner` is available in versions 2 and 3. Versions 1 and 2 of CodeDraw let users pass event listeners in the form of lambdas that then got executed on the CodeDraw event loop.

2.14.1 Event Listener (v1)

In version 1 of CodeDraw events were handled through the event listeners where lambdas had to be passed on to event methods like `onMouseMove(EventHandler eventHandler)`. This has the advantage of not losing any events but is very complicated to implement, since local variables of the surrounding method cannot be modified from inside a lambda. This made it very hard to change the state of the program as users had to introduce global variables to update the state of their program based on events.

```
public interface EventHandler<TSender, TArgs> {
    void handle(TSender sender, TArgs args);
}
```

Figure 2.13: The `EventHandler` interface. `TSender` is always the `CodeDraw` class and `TArgs` are the event arguments.

Another issue was that events were executed on a separate CodeDraw event thread but rendering happened on the main thread. This meant that, depending on how the user updated their state, concurrency errors could appear. Since CodeDraw is intended to be used by CS1 students, this is not something we could or would want to teach them.

2.14.2 EventScanner (v2)

In version 2 of CodeDraw the EventScanner was introduced. It matches its implementation almost exactly to the `java.util.Scanner` class. This avoids concurrency issues from the users perspective, it also allows the modification of local variables and no events that are produced by Java AWT are lost. Internally, the EventScanner works like a concurrent blocking queue and mirrors the blocking behavior of the Scanner class.

```
1 CodeDraw cd = new CodeDraw();
2 EventScanner es = cd.getEventScanner();
3 cd.drawText(200, 200, "Move your mouse over here.");
4 cd.show();
5 cd.setColor(Palette.RED);
6
7 while (!cd.isClosed()) {
8     while (es.hasEventNow()) {
9         if (es.hasMouseMoveEvent()) {
10            MouseMoveEvent e = es.nextMouseMoveEvent();
11            cd.fillRect(e.getX() - 5, e.getY() - 5, 10);
12        } else {
13            es.nextEvent();
14        }
15    }
16    cd.show(16);
17 }
```

Figure 2.14: A simple EventScanner program that draws a red square wherever the mouse moves.

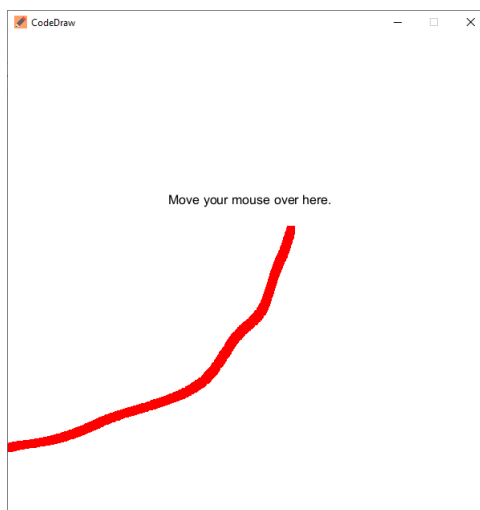


Figure 2.15: The output of the EventScanner program in figure 2.14 when moving the mouse over the CodeDraw window.

The example in figure 2.14 is implemented using version 3 of CodeDraw and draws a square wherever the mouse is moved. First, a CodeDraw object has to be created and the EventScanner has to be declared in lines 1 and 2.

In line 7 an endless loop is created, in which each iteration of the loop will show one frame. The inner loop on line 8 consumes all currently available events. If there are no more events available, the inner loop stops and the changes are displayed. The inner loop has to stop quite frequently to allow the changes to be displayed in the CodeDraw window. The `show()` command could be placed inside the inner loop but this would significantly slow down the program because quickly moving the mouse over the canvas generates a significant amount of mouse move events. These would not be processed quickly enough if rendering happened for each mouse move event individually.

Inside the inner loop, in each iteration exactly one event is processed. To check which event is currently at the head of the queue a *has* method can be called, as can be observed in line 9 by the use of the `hasMouseEvent()` method. If there is a mouse move event, the `nextMouseEvent()` method should be called to remove the element from the head of the queue and return it as the method result, as seen in line 10. The returned event can then be used to update the state of the application. In line 11, a red square is drawn around the mouse position. If the head of the queue does not have a mouse move event, the event must be discarded. This is done in line 13 by calling `nextEvent()`. If this is not done, the head of the event queue remains the same and the inner loop would be stuck in an endless loop.

In line 16, after processing all currently available events, the `show()` method is called to display the drawn rectangles.

A problem that can arise when using the EventScanner is that users might think that they understood the Scanner and therefore the EventScanner, but still struggle to implement their desired behavior. The main reason is that the requirements on the EventScanner are more complicated than on the Scanner. In principle, it would be as easy to read one event from the EventScanner as it is to read an integer from the Scanner class. However, when implementing an interactive application, the user does not want to process just one mouse click from the window. The user wants to continuously handle input. Often, it is also undesirable to block rendering when no event happens. Additionally, a lot of unexpected events are generated while executing a CodeDraw program and users always have to write code to discard events they do not wish to process. This makes using the EventScanner a bit complicated, but once the pattern has been learned, it is easy to create interactive application with it.

2.14.3 Enhanced EventScanner (v3)

There is another way to use the EventScanner. Using both the fact that the EventScanner is iterable in version 3 and that Java has switch type matching, the result in figure 2.16 is created. The `foreach` loop consumes all available events and the switch type matching branches for the different event types. There is also less room for error when

```
CodeDraw cd = new CodeDraw();

cd.drawText(200, 200, "Move your mouse over here.");
cd.show();
cd.setColor(Palette.RED);

while (!cd.isClosed()) {
    for (var e : cd.getEventScanner()) {
        switch (e) {
            case MouseEvent a
                -> cd.fillSquare(a.getX() - 5, a.getY() - 5, 10);
            default
                -> { }
        }
    }

    cd.show(16);
}
```

Figure 2.16: An enhanced EventScanner program that draws a red square wherever the mouse moves.

implementing event handling this way. There cannot be a mismatch between the *has* and the *next* method, the discarding of unused events happens by simply ignoring the event and users cannot accidentally handle two events at once by using consecutive **ifs** instead of **else ifs**.

2.14.4 Game Development with the EventScanner

To get a better understanding of the `Animation` interface and compare it to the `EventScanner`, the game from this section is implemented with the `EventScanner` and with the `Animation` interface in the next section.

A typical game, as shown in figure 2.18, would consist of a game state such as the position of 200,000 particles and the mouse position. The game state could be other things as well, like the level or health of a character, the map of the game or the enemies within the game. A game would also typically have a game loop where each iteration calculates one frame. The game loop usually consists of three parts: event handling, game logic and rendering.

The event handling section processes all interactions of the user with the `CodeDraw` window and updates the state of the game accordingly. In this particle example just the mouse position is updated. Other games could process the WASD keys to move the player or the space bar to make an attack.

The game logic simulates the world and transforms the game state from the previous state to the next state, e.g., simulating the actions of enemies or non-player-characters.

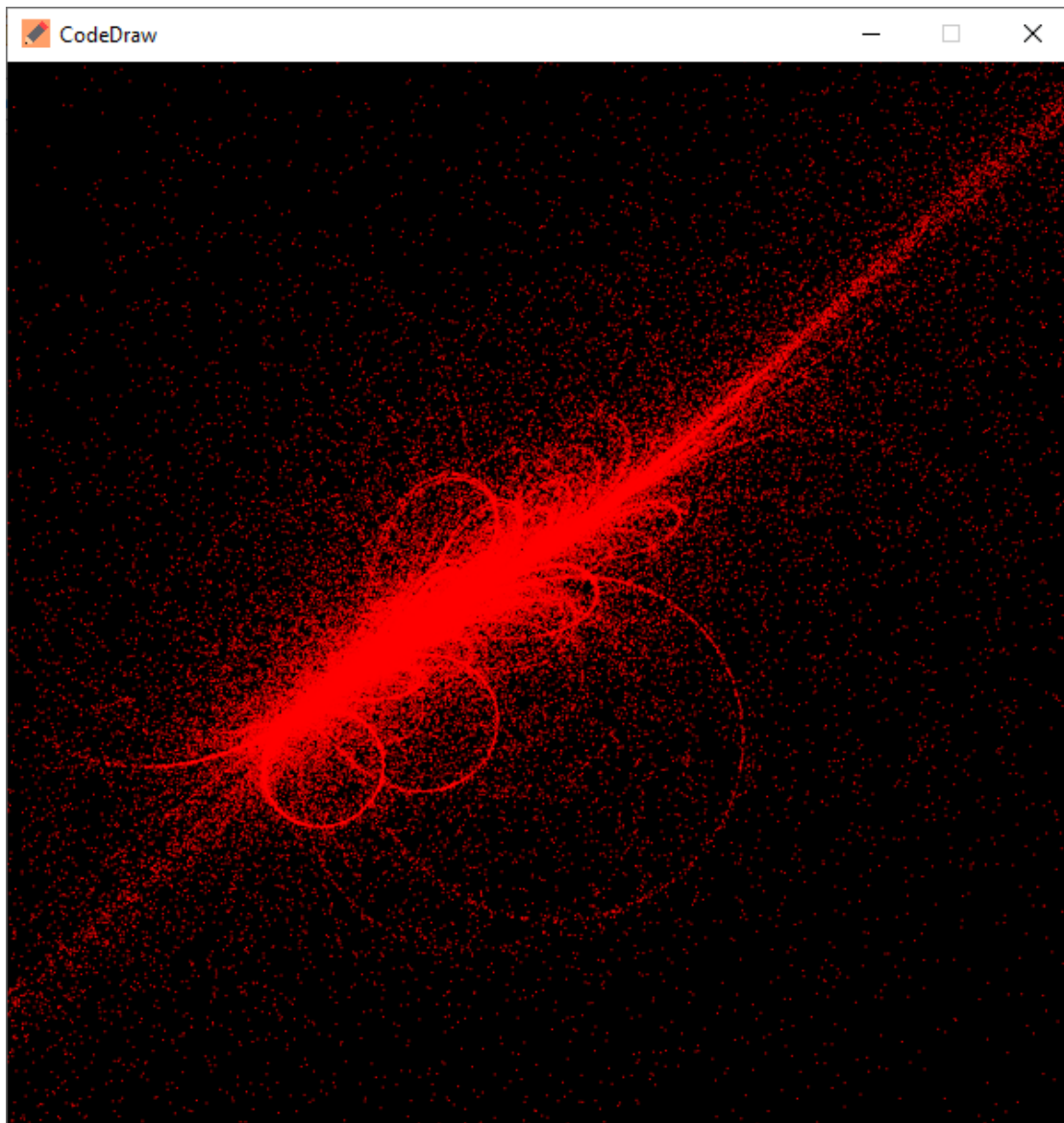


Figure 2.17: An example output of the particle program in figure 2.18 being executed.

In this particle example in figure 2.18, the position of each particle is updated. The closer the particle is to the mouse, the faster it moves in the direction of the current mouse position.

The rendering displays the state of the game in a visually appealing way. Usually before rendering you want to clear the canvas and render each frame from a blank canvas. In this case the canvas is cleared black and then for each particle a red pixel is drawn.

This structure is of course just a suggestion. It will work with many common interactive

2. CODEDRAW FEATURES

```
CodeDraw cd = new CodeDraw();
EventScanner es = cd.getEventScanner();

final int particleCount = 100000;
final double exponent = -0.25 * Math.sqrt(2) - 0.5;
final double speed = 100;

// game state
int mouseX = cd.getWidth() / 2;
int mouseY = cd.getHeight() / 2;
double[] particlesX = new double[particleCount];
double[] particlesY = new double[particleCount];
for (int i = 0; i < particleCount; i++) {
    particlesX[i] = Math.random() * cd.getWidth();
    particlesY[i] = Math.random() * cd.getHeight();
}

// game loop
while (!cd.isClosed()) {
    // event handling
    while (es.hasEventNow()) {
        if (es.hasMouseMoveEvent()) {
            MouseMoveEvent a = es.nextMouseMoveEvent();
            mouseX = a.getX();
            mouseY = a.getY();
        } else {
            es.nextEvent();
        }
    }

    // game logic
    for (int i = 0; i < particleCount; i++) {
        double distanceX = mouseX - particlesX[i];
        double distanceY = mouseY - particlesY[i];

        double movement = speed *
            Math.pow(distanceX * distanceX + distanceY * distanceY, exponent);

        particlesX[i] += distanceX * movement;
        particlesY[i] += distanceY * movement;
    }

    // rendering
    cd.clear(Palette.BLACK);
    for (int i = 0; i < particleCount; i++) {
        cd.setPixel((int)particlesX[i], (int)particlesY[i], Palette.RED);
    }
    cd.show(8);
}
```

18 Figure 2.18: This program produces particles that follow the mouse cursor. The closer the mouse cursor is to individual particles the faster they will move.

programs, but there are situations where other structures might be more convenient. For example, in certain programs you might want the event handling to be blocking, since no changes happen without the user interacting with the CodeDraw window, or you might want to change the order of handling events, simulating the world and drawing.

2.14.5 Inversion of Control (v3)

Version 3 of CodeDraw introduces the `Animation` interface which can be passed to the `CodeDraw.run(Animation, ...)` method. The `run` method then uses the implementation of the `Animation` interface to run the user's program. The same method exists in the `FullScreen` and `BorderlessWindow` class. Since control is passed onto CodeDraw with the `run` method, the main thread executes the `draw` and `event` methods which avoid concurrency issues. It also reduces the amount of control structures the user has to implement. Instead of control structures the user has to implement several object oriented patterns, such as inheriting from the `Animation` interface, overriding methods, creating object variables and using the constructors.

The equivalent to figure 2.18 can be seen in figure 2.19. The state of the program is modeled as object variables instead of local variables. The user has to implement the `draw(Image canvas)` method which is called in regular intervals depending on the set frame rate. The canvas to draw on is supplied as an argument to the `draw` method. Events are implemented by overriding methods such as `onMouseMove(MouseMoveEvent event)` and then changing the state of the object variables.

Optionally the `simulate()` method can be overridden, in order not to put the game logic in the `draw` method. The advantage of the `simulate()` method is that the world simulation happens independently of the drawing of frames. If the user's computer is under heavy load, frames might be dropped by CodeDraw since the computer cannot keep up. This would not happen with the `simulate` method which is guaranteed to execute.

In the example in figure 2.19 the game state is now modeled as the state of the object and the initialization of the game state has been moved into the constructor. The rendering happens in the `draw` method. The update of the particles happens in the `simulate` method and the update of the mouse position happens in the `onMouseMove(MouseMoveEvent)` method.

```
import codedraw.*;

public class GameDevelopmentOOP implements Animation {
    public static void main(String[] args) {
        CodeDraw.run(new GameDevelopmentOOP(600, 600), 600, 600, 60, 60);
    }

    private static final int PARTICLE_COUNT = 100000;
    private static final double EXPONENT = -0.25 * Math.sqrt(2) - 0.5;
    private static final double SPEED = 100;

    private int mouseX;
    private int mouseY;
    private double[] particlesX;
    private double[] particlesY;

    public GameDevelopmentOOP(int width, int height) {
        mouseX = width / 2;
        mouseY = height / 2;
        particlesX = new double[PARTICLE_COUNT];
        particlesY = new double[PARTICLE_COUNT];

        for (int i = 0; i < PARTICLE_COUNT; i++) {
            particlesX[i] = Math.random() * width;
            particlesY[i] = Math.random() * height;
        }
    }

    @Override
    public void simulate() {
        for (int i = 0; i < PARTICLE_COUNT; i++) {
            double distanceX = mouseX - particlesX[i];
            double distanceY = mouseY - particlesY[i];

            double movement = SPEED *
                Math.pow(distanceX * distanceX + distanceY * distanceY, EXPONENT);

            particlesX[i] += distanceX * movement;
            particlesY[i] += distanceY * movement;
        }
    }

    @Override
    public void draw(Image canvas) {
        canvas.clear(Palette.BLACK);
        for (int i = 0; i < PARTICLE_COUNT; i++) {
            cd.setPixel((int)particlesX[i], (int)particlesY[i], Palette.RED);
        }
    }

    @Override
    public void onMouseMove(MouseMoveEvent event) {
        mouseX = event.getX();
        mouseY = event.getY();
    }
}
```

20

Figure 2.19: The same program as in figure 2.18 but written with inversion of control.

2.15 Paths

More complicated shapes can be created by calling `fillPathStartingAt(double x, double y)` or `drawPathStartingAt(double x, double y)`. To continue the path method chaining is used to specify one of four line segments: `lineTo()`, `curveTo()`, `bezierTo()`, `arcTo()`; At the end `complete()` has to be called to close and draw the shape.

The example in figure 2.20 draws a section of a dart board, which starts at the bottom right (200, 300) as defined in line 3 and draws an arc clockwise around the center (300, 300) in line 4. Then a line is drawn from (229, 299) to (158, 158) in line 5. The (229, 229) coordinate does not have to be specified since it is stored in the underlying Path object that is used to enable the method chaining. Next, in line 6, another arc is drawn that goes counter-clockwise. Finally, in line 7, `complete()` is called and the last line is drawn to complete the shape. No coordinates have to be specified since the Path object knows the starting point of the shape. If `complete()` is not called, the shape is not drawn onto the canvas.

```

1 CodeDraw cd = new CodeDraw();
2 double angle = Math.PI / 4; // 45°
3 cd.fillPathStartingAt(200, 300)
4   .arcTo(300, 300, angle)
5   .lineTo(300 - Math.cos(angle) * 200, 300 - Math.sin(angle) * 200)
6   .arcTo(300, 300, -angle)
7   .complete();
8 cd.show();

```

$$300 - \cos\left(\frac{\pi}{4}\right) * 200 = 300 - \sin\left(\frac{\pi}{4}\right) * 200 = 158$$

Figure 2.20: A segment of a dart board as an example for a custom shape.

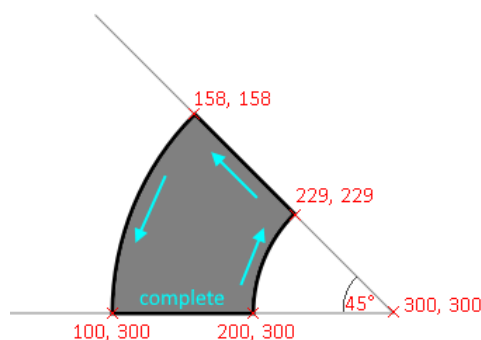


Figure 2.21: An illustration on how the path in figure 2.20 would be drawn.

CodeDraw at the Vienna University of Technology


CodeDraw is designed to fit the needs of the Vienna University of Technology. To understand the design decisions that went into CodeDraw one also has to understand how our university structures its courses.


3.1 Programming Courses at the Vienna University of Technology

Before the start of their first semester at the Vienna University of Technology, students can voluntarily attend a course named PROLOG. In this course the Processing language is used to introduce students to fundamental programming concepts. The PROLOG course lasts for about 2 weeks and afterwards the semester begins.

Our CS1 course “Introduction to Programming 1/Einführung in die Programmierung 1” mainly focuses on structured programming concepts (sequence, loops, branches), procedural programming concepts (methods, variables, arrays), recursion and basic algorithms, using the Java programming language. Students are not taught how to implement classes, just how to use them to the extent that is necessary. Only in the last few lectures a short introduction to object orientation is given, this, however, is optional and not graded.¹ Previously, StdDraw was used as a graphical library in this course.

There are two courses which could be described as CS2, “Introduction to Programming 2/Einführung in die Programmierung 2”² and “Algorithms and Datastructures/Algorithmen

¹Introduction to Programming 1 course page  <https://tiss.tuwien.ac.at/course/courseDetails.xhtml?courseNr=185A91>

²Introduction to Programming 2 course page  <https://tiss.tuwien.ac.at/course/courseDetails.xhtml?courseNr=185A92>

und Datenstrukturen”³. Introduction to Programming 2 teaches object oriented programming. Both courses and several other courses at the Vienna University of Technology use Java.

3.2 Why develop CodeDraw

Since Introduction to Programming 1 specifically does not teach object orientation, a library written for this lecture must support that approach of teaching. Ideally, students should be able to observe a library written in an object oriented style while not having to understand object orientation. Additionally, there should not be a stark difference between the library and Processing, which is used in the PROLOG course beforehand.

Before developing CodeDraw, we used StdDraw at Vienna University of Technology. However, StdDraw was written in a singleton pattern that made it accessible from anywhere within the program. This pattern forced students to write the word *StdDraw* in front of every method and means that all getters and setters are static. Additionally, events cannot be handled properly and section 5.8.3 discusses this in more detail. Furthermore, the coordinate grid starts at the bottom left which is quite unusual for any programming graphics library. Despite there being the Draw library which does not use the singleton pattern, all of these issues led the author to develop CodeDraw in order to give students a well rounded experience.

3.3 How CodeDraw is used in Introduction to Programming 1

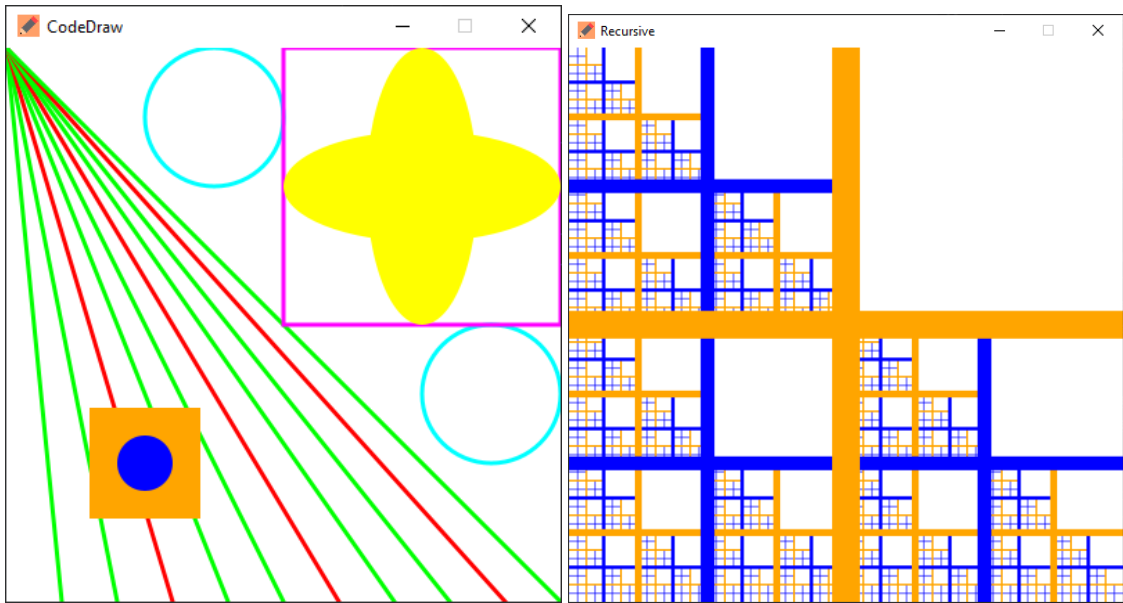
There are six exercise sheets that are handed out throughout the semester.⁴ In the first five exercise sheets there are five to six exercises each. One or two requiring the use of CodeDraw, with exercises increasing in difficulty until the last exercise sheet. In the last exercise sheet students have to implement a game. Some parts are already implemented because it would be too difficult for most students to build a game without at least some guidance. Thus, students mainly have to complete well documented methods that have not yet been implemented.

During the semester we also give students a creative exercise where they can create anything they want but have to use at least one loop and one branch while utilizing CodeDraw. In total the final program should have around 50 to 200 statements. It gives students more freedom to set loose their creativeness while not being so complex as to overburden them.

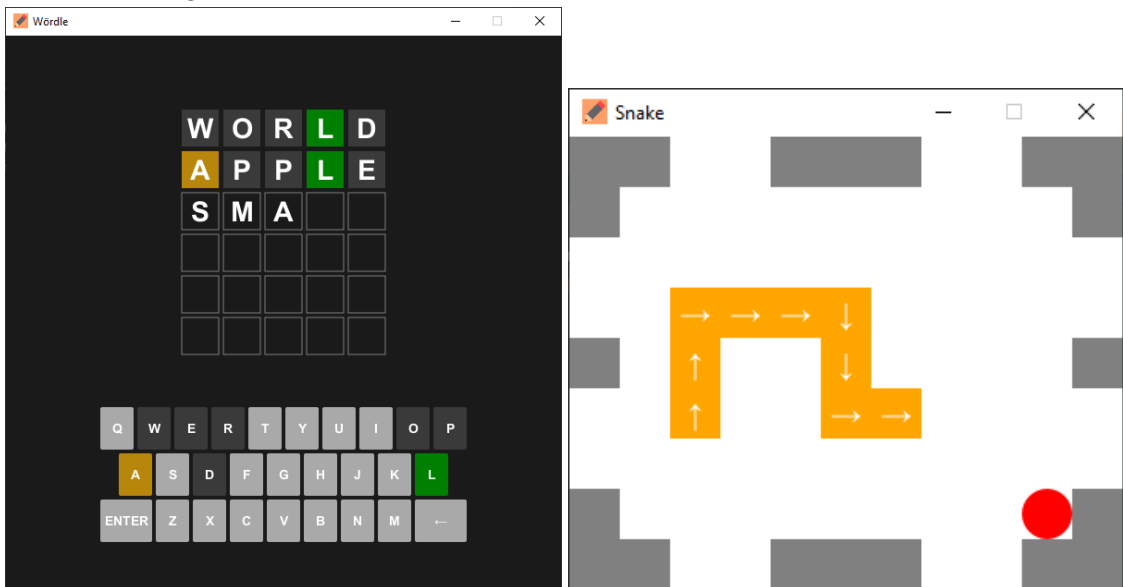
³Algorithms and Data Structures course page [WBM https://tiss.tuwien.ac.at/course/courseDetails.xhtml?courseNr=186866](https://tiss.tuwien.ac.at/course/courseDetails.xhtml?courseNr=186866)

⁴Some details on the exercises can be found on the Introduction to Programming 1 course page [WBM https://tiss.tuwien.ac.at/course/courseDetails.xhtml?courseNr=185A91](https://tiss.tuwien.ac.at/course/courseDetails.xhtml?courseNr=185A91)

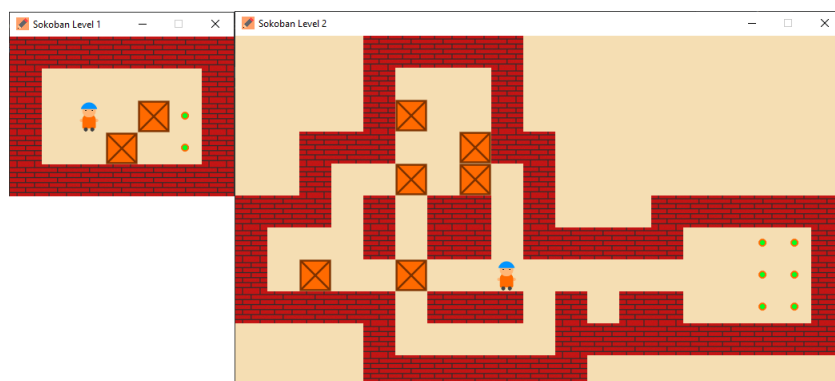
Below in figure 3.1 are some examples of what we do with CodeDraw in Introduction to Programming 1. Figure 3.1 (a) and 3.1 (b) show examples of exercises given throughout the semester. Figures 3.1 (c) (d) and (e) show games that are implemented with CodeDraw and might be given to students as a final exercise in the future.



(a) A simple exercise to familiarize students with the coordinate system, the use of colors and the order of drawing. (b) Creating a recursive pattern in CodeDraw.



(c) Implementing Wordle in CodeDraw. (d) Implementing a simple version of Snake in CodeDraw.



(e) Implementing Sokoban in CodeDraw.

Figure 3.1: Games and patterns that are implemented using CodeDraw.

Learning Programming Concepts through CodeDraw

There are a number of programming concepts that students learn to some degree while using CodeDraw. They do not necessarily need to fully comprehend those concepts to use CodeDraw, but once these concepts are explained, students will have an easier time understanding them.

4.1 Objects as a Resource

Modelling the drawing window as an object leads students to associate the visible window with an object. Although students who learn programming for the first time might not understand what a constructor or an instance variable is, they are able to create an object, draw something and then call `show()`. This is also an acceptable amount of overhead to create graphical output.

Additionally, the constructor of CodeDraw is an analogy for resource creation. This means that each time the CodeDraw constructor is called, a new window appears on the screen. The resource, that is the window, can then only be drawn onto once the constructor has been called. Just as other resources the CodeDraw class also has a close method, that, when called, closes the CodeDraw window. This is a common pattern used in the Java standard library when handling IO.

4.2 Show is Slow

Compared to the various draw methods the `show()` method is slow. Students often do not realize that certain methods are slower than others and that they should avoid calling them unnecessarily. By moving `show()` outside of loops or at the end of the main

```
CodeDraw cd = new CodeDraw();
cd.drawLine(100, 100, 200, 300);
cd.show(2000);
cd.close();
```

Figure 4.1: Code that opens a CodeDraw window, draws a line and closes it after 2 seconds.

method there is a very noticeable speed up, especially when drawing a lot of shapes. In Introduction to Programming 1 students do not have to consider the performance of their applications, but this might be the first time they notice that some programs are faster than others.

4.3 Data Composition, Enumeration Types and Immutability

One of the first concepts taught surrounding object orientation is usually data composition, where new data types are built by combining already existing data types. The `TextFormat` class composes several text formatting options into one object that can be accessed by calling `getTextFormat()` on a `CodeDraw` object.

There are several places in the `CodeDraw` library where enumeration types are used. For example, the `TextFormat` class defines different underline types through the `Underline` enumeration type and defines the origin of the text relative to the specified location through the `TextOrigin` enumeration type. The corner of some shapes can be changed by setting a different `Corner` value on a `CodeDraw` object. When drawing images onto a canvas an `Interpolation` can optionally be defined through an enumeration type. There are also (partial) enumeration types such as `Color` or `CursorStyle` which have some predefined values, but custom values can be created.

Immutability means that once created, the state of an object can no longer be changed. Although the `CodeDraw` and `Image` classes are not immutable, their width and height is. However, there are other classes such as the event classes and the `Matrix2D` class which are fully immutable. The `Matrix2D` is build as to allow the application of successive matrix transformations through method chaining.

4.4 Chapter Conclusion

Although `CodeDraw` does not nearly cover everything about programming and its design patterns, for a few basic principles it offers a practical analogy. By using `CodeDraw`, students will already be familiar with the contexts these patterns might appear in.

Design Comparison and Related Work

There are several libraries written in Java which work similarly to CodeDraw. They also try to make creating images and animations simple and accessible with a focus on education. This section is not a comprehensive comparison of these libraries but rather a short overview on how their programming interfaces are designed.

ACM¹ and ObjectDraw² [BDM01] model their shapes as modifiable objects. The usage of the ACM library is described in an experience report by Schuster [Sch10]. StdDraw, Draw and Processing model their shapes as functions. StdDraw and Draw³ [SW17] are two versions of the (almost) same interface but Draw can be instantiated as an object while StdDraw is static only. Processing⁴ [RF14] is strictly speaking not a Java library since it uses its own dialect of Java and simplifies a lot of Java's boilerplate. Processing is included in this comparison because it is quite popular, used at our University in the PROLOG course and influenced some design decisions of CodeDraw.

Purposefully excluded are Java libraries that are complete game engines because their focus is less on teaching and more on game development. They usually offer quite a large feature set, but in return take longer to fully comprehend.

¹The ACM library page [WBM https://cs.stanford.edu/people/eroberts/jtf/](https://cs.stanford.edu/people/eroberts/jtf/)

²ObjectDraw page [WBM https://cseweb.ucsd.edu/~ricko/CSE11/links.html](https://cseweb.ucsd.edu/~ricko/CSE11/links.html)

³The StdLib library that contains StdDraw and Draw [WBM https://introcs.cs.princeton.edu/java/stdlib/](https://introcs.cs.princeton.edu/java/stdlib/)

⁴The Processing website [WBM https://processing.org/](https://processing.org/)

5.1 Orientation of the Pixel Grid

There are two common approaches to the orientation of coordinate systems. The mathematical approach in which the $(0, 0)$ coordinate is at the bottom left and coordinates increase to the top right. StdDraw and Draw use this approach. However, most graphical libraries take a different approach and place their origin coordinate at the top left and increase coordinates to the bottom right. ObjectDraw, ACM Graphics, Processing and CodeDraw all take this approach. Many common non-educational libraries also have their origin at the top left: HTML Canvas Graphics⁵, Java AWT Graphics⁶, C# Windows Forms⁷.

The disadvantage of having the origin at the top left in CS1 is that most people are only familiar with the coordinate grid taught in mathematics. It is hard for some people to get used to another coordinate origin.

5.2 Origin of Shapes

Many graphics libraries place the origin of their shapes at the top left: HTML Canvas Graphics, Java AWT Graphics, C# Windows Forms, ACM Graphics and ObjectDraw. StdDraw and Draw place the origin coordinate at the center of each shape. Processing mixes both approaches: circular shapes have their origin at the center and rectangular shapes have their origin at the top left. There are options in Processing to change the origin of shapes.

CodeDraw took Processing's approach and places the origin of rectangular shapes at the top left and the origin of circular shapes at their center but there is no option to change the origin of shapes in CodeDraw.

5.3 Inversion of Control

Inversion of control means that the library or framework calls the code that the programmer writes. This gives libraries a lot of control over what happens when and lets libraries better support their users. The disadvantage of using inversion of control in Java is that it can be complicated if you are new to programming. Users either need to inherit from a class or interface, use annotation, or pass method references to the library. These are all complicated concepts to understand for beginners and therefore not suitable for beginner libraries. However, Processing can still use inversion of control and avoid these complicated concepts by using a transpiler.

⁵Mozilla web docs about Drawing Graphics **WBM** https://developer.mozilla.org/en-US/docs/Learn/JavaScript/Client-side_web_APIs/Drawing_graphics#2d_canvas_basics

⁶Oracle documentation about coordinates **WBM** <https://docs.oracle.com/javase/tutorial/2d/overview/coordinate.html>

⁷Microsoft .NET documentation **WBM** <https://learn.microsoft.com/en-us/dotnet/desktop/winforms/advanced/types-of-coordinate-systems?view=netframeworkdesktop-4.8>

Originally, CodeDraw was designed with a procedural approach in which the user has to call methods in CodeDraw for each change and inversion of control was purposefully avoided. In version 3 CodeDraw supports inversion of control through the `Animation` interface, which is implemented by only using the existing procedural implementation of CodeDraw.

`StdDraw` and `Draw` use a procedural approach only. `ACM Graphics` and `ObjectDraw` use inversion of control in some parts of their libraries. For example, inheritance can be used to implement events.

5.4 Double Buffer vs. Single Buffer

Buffering in this context describes the approach of using memory regions to store the displayed graphical user interface. The number of buffers influences the performance and behavior of a graphics library.

If drawing happens on the main thread with a single buffer, each draw operation must run mutually exclusive to the concurrently running AWT thread. If the AWT thread decides to frequently re-render, it could slow down drawing and from the user's perspective the AWT thread would sometimes render their changes and sometimes it would not. Therefore, the user would still have to invoke a re-render to make sure their changes are applied at a specific point in time. From the user's perspective this would lead to inconsistent behavior where sometimes changes appear on screen and sometimes they do not. Another solution is rendering changes each time something is drawn, however, this would slow down drawing significantly.

Double buffering uses two buffers, one buffer on which the user draws and another buffer that Java AWT accesses to draw the window. By using double buffering, the mutually exclusive zone is reduced to just copying the contents of the first buffer into the second buffer. Double buffering has very consistent behavior since the changes are displayed exactly when the library copies the contents of the first buffer into the second and are not displayed if the contents are not copied.

`StdDraw` always uses two buffers. When double buffering is disabled and a shape is drawn onto the first buffer, the buffer is immediately copied onto the second buffer and displayed on screen. This will happen for every drawing call and is the default option. The advantage is that each change is immediately shown on screen and no separate call for copying the contents of the first buffer into the second is necessary. The disadvantage is that the copying is slow and this slowdown is noticeable if a lot of shapes are drawn.

To solve this issue, `StdDraw` has a built-in `StdDraw.enableDoubleBuffering()` method. When double buffering is enabled, the user has to call `StdDraw.show()` to display changes and `StdDraw.pause(int)` to pause between renders. The draw operations then draw onto the first buffer and only when `StdDraw.show()` is called, the changes are copied into the second buffer and displayed.

CodeDraw always uses double buffering which has the drawback of users always having to call `show()`. The advantage is that they do not have to know what double buffering is and by the time they do animations, they are already familiar with calling `show()`. The `show(long waitMilliseconds)` method also combines the showing and waiting into a single method call. The time it takes to copy the first buffer into the second is subtracted from the total delay and the thread sleeps for the remaining delay.

5.5 Angles and Rotation

Angles have to be used every time there is some central point around which some section of a circle is drawn or when a rotation has to be applied. There are a few places in the CodeDraw library where that happens, namely `drawArc()`, `drawPie()`, `fillPie()`, `Matrix2D` rotation and when drawing arcs by using paths.

In mathematics the $(0, 0)$ coordinate is at the bottom left and coordinates increase to the top right. The sine and cosine functions in such coordinate systems produce a coordinate starting at 3 o'clock going counter-clockwise around the $(0, 0)$ point.

```
drawArc(  
    centerX = 100,  
    centerY = 100,  
    radius = 100,  
    startAngleInRadians = alpha,  
    lengthInRadians = PI / 2  
)  
  
drawLine(  
    startX = 100,  
    startY = 100,  
    endX = cos(alpha) * 100 + 100,  
    endY = sin(alpha) * 100 + 100,  
)
```

What convention should be used to define the start and the direction of such an arc? CodeDraw stays consistent with the behavior of the sine and cosine functions. Since most programming graphics libraries including CodeDraw have the $(0, 0)$ coordinate in the top left and increase the coordinate to the bottom right, a line drawn in such a coordinate system would start at the 3 o'clock and go clockwise. The length in `drawArc()` is interpreted as going clockwise since an increase in the `alpha` angle would also go clockwise in the `drawLine()` example. This keeps consistency between the trigonometric functions and the related methods in CodeDraw.

5.6 Static Image Comparison

To illustrate how these different libraries approach drawing, a simple example is used. Draw a red line from the top left to the bottom right and write the text “Hello World!”. The window should be 400 by 400 pixels in size. This example is simple enough to compare the basic structure and show the overhead of using each of those graphics libraries.

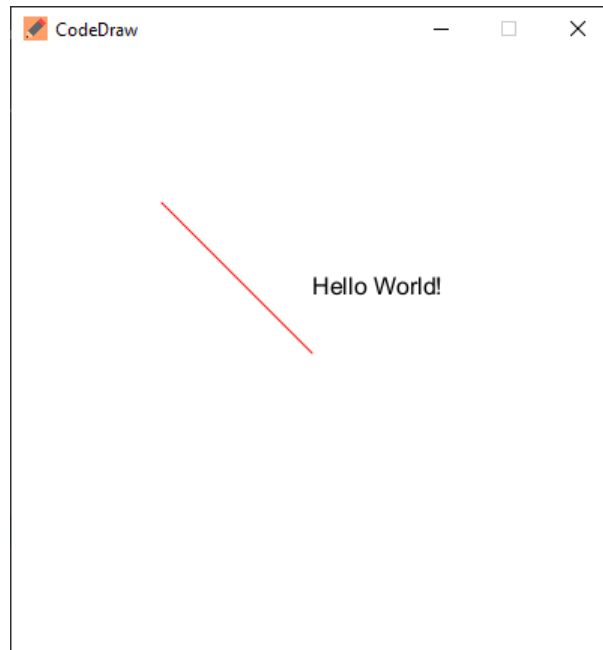


Figure 5.1: The typical output of each graphics library when drawing a red line and the words "Hello World!".

The output of each library looks roughly like the output in figure 5.1. The output mostly varies based on whether these libraries use anti-aliasing and how exactly they align text. As a baseline, pure Java AWT is also included. Draw and StdDraw have no imports since they are in the default package. Processing has the advantage of being transpiled to Java and therefore there is no need to write a class or main method.

```

(a) ACM
import acm.graphics.*;

import javax.swing.*;
import java.awt.*;

public class HelloWorldLine {
    public static void main(String[] args) {
        JFrame frame = new JFrame();
        frame.setSize(400, 400);

        GCanvas gc = new GCanvas();
        frame.getContentPane()
            .add(BorderLayout.CENTER, gc);

        GLine line = new GLine(100, 100, 200, 200);
        line.setColor(Color.RED);
        gc.add(line);

        GLabel label =
            new GLabel("Hello World!", 200, 150);
        gc.add(label);

        frame.show();
    }
}

(b) Java AWT
import javax.swing.*;
import java.awt.*;

public class HelloWorldLine {
    public static void main(String[] args) {
        JFrame frame = new JFrame();
        frame.setContentPane(new MyPanel());
        frame.pack();
        frame.setVisible(true);
    }

    private static class MyPanel extends JPanel {
        public MyPanel() {
            setPreferredSize(new Dimension(400, 400));
        }

        @Override
        protected void paintComponent(Graphics g) {
            super.paintComponent(g);
            g.clearRect(0, 0, 400, 400);

            g.setColor(Color.RED);
            g.drawLine(100, 100, 200, 200);

            g.setColor(Color.BLACK);
            g.drawString("Hello World!", 200, 150);
        }
    }
}

(c) CodeDraw
import codedraw.*;

public class HelloWorldLine {
    public static void main(String[] args) {
        CodeDraw cd = new CodeDraw(400, 400);

        cd.setColor(Palette.RED);
        cd.drawLine(100, 100, 200, 200);

        cd.setColor(Palette.BLACK);
        cd.drawText(200, 150, "Hello World!");

        cd.show();
    }
}

(d) CodeDraw's Animation interface
import codedraw.*;

public class HelloWorldLine implements Animation {
    public static void main(String[] args) {
        CodeDraw.run(new HelloWorldLine(), 400, 400, 1);
    }

    @Override
    public void draw(Image canvas) {
        canvas.clear();
        canvas.setColor(Palette.RED);
        canvas.drawLine(100, 100, 200, 200);
        canvas.setColor(Palette.BLACK);
        canvas.drawText(200, 150, "Hello World!");
    }
}

(e) Draw
public class HelloWorldLine {
    public static void main(String[] args) {
        Draw d = new Draw();
        d.setCanvasSize(400, 400);

        d.setPenColor(Draw.RED);
        d.line(0.25, 0.75, 0.5, 0.5);

        d.setPenColor(Draw.BLACK);
        d.text(0.5, 0.70, "Hello World!");
    }
}

(f) StdDraw
public class HelloWorldLine {
    public static void main(String[] args) {
        StdDraw.setCanvasSize(400, 400);

        StdDraw.setPenColor(StdDraw.RED);
        StdDraw.line(0.25, 0.75, 0.5, 0.5);

        StdDraw.setPenColor(StdDraw.BLACK);
        StdDraw.text(0.5, 0.70, "Hello World!");
    }
}

(g) Processing
size(400, 400);
background(#FFFFFF);
stroke(#FF0000);
line(100, 100, 200, 200);
fill(#000000);
text("Hello World", 200, 150);

(h) ObjectDraw
import objectdraw.*;
import java.awt.*;

public class HelloWorldLine {
    public static void main(String[] args) {
        AWTFrameCanvas canvas = new AWTFrameCanvas(400, 400);

        Line line = new Line(100, 100, 200, 200, canvas);
        line.setColor(Color.RED);
        line.show();

        Text text = new Text("Hello World!", 200, 150, canvas);
        text.show();
    }
}

```

Figure 5.2: Producing a red line and the text "Hello World!" in different libraries.

5.7 Animation Comparison

To compare how animations and user interactions are implemented, only CodeDraw, ObjectDraw, StdDraw, Processing and Java AWT are used. As an example a clock with 12 dots for each hour and a second hand showing the time is implemented.

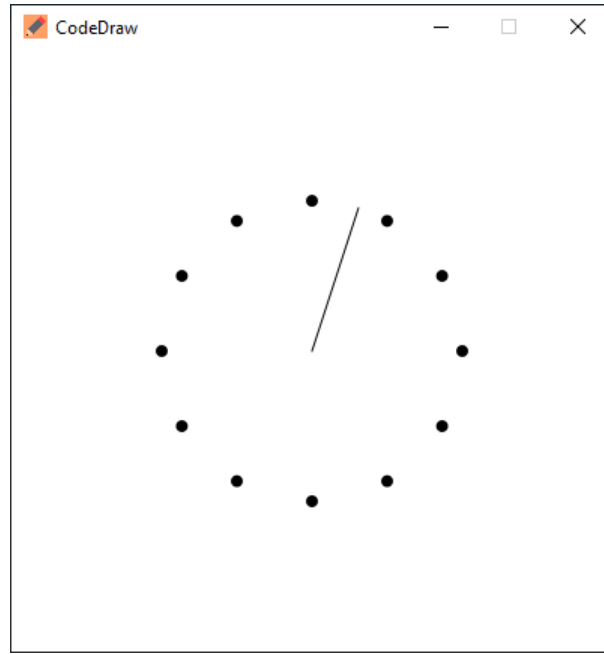


Figure 5.3: The typical output of each graphics library when implementing the clock example.

CodeDraw, ObjectDraw and StdDraw are all implemented using a loop that pauses for one second in each iteration. When using inversion of control in CodeDraw or in Processing, the frame rate has to be limited to one frame per second. Each rendered frame also increases the angle of the second hand. In Java AWT you have to both call your code once per second from the main thread and increase the angle at each render on the AWT thread.

This is not an accurate clock, all examples will drift over time. Especially the CodeDraw, ObjectDraw and StdDraw example will take longer because execution time plus sleep time is more than one second.

```
import codedraw.*;

public class Clock {
    public static void main(String[] args) {
        CodeDraw cd = new CodeDraw(400, 400);

        for (double sec = -Math.PI / 2; !cd.isClosed(); sec += Math.PI / 30) {
            cd.clear();

            cd.drawLine(200, 200, Math.cos(sec) * 100 + 200, Math.sin(sec) * 100 + 200);

            for (double j = 0; j < Math.PI * 2; j += Math.PI / 6) {
                cd.fillCircle(Math.cos(j) * 100 + 200, Math.sin(j) * 100 + 200, 4);
            }

            cd.show(1000);
        }
    }
}
```

Figure 5.4: CodeDraw animation example.

```
import codedraw.*;

public class Clock implements Animation {
    public static void main(String[] args) {
        CodeDraw.run(new Clock(), 400, 400, 1, 1);
    }

    private double sec = -Math.PI / 2;

    @Override
    public void simulate() {
        sec += Math.PI / 30;
    }

    @Override
    public void draw(Image canvas) {
        canvas.clear();
        canvas.drawLine(200, 200, Math.cos(sec) * 100 + 200, Math.sin(sec) * 100 + 200);

        for (double j = 0; j < Math.PI * 2; j += Math.PI / 6) {
            canvas.fillCircle(Math.cos(j) * 100 + 200, Math.sin(j) * 100 + 200, 4);
        }
    }
}
```

Figure 5.5: CodeDraw animation example using the Animation interface.

```

public class Clock {
    public static void main(String[] args) {
        StdDraw.setCanvasSize(400, 400);
        StdDraw.setScale(0, 400);
        StdDraw.enableDoubleBuffering();

        for (double sec = Math.PI / 2; true; sec -= Math.PI / 30) {
            StdDraw.clear();

            StdDraw.line(200, 200, Math.cos(sec) * 100 + 200, Math.sin(sec) * 100 + 200);

            for (double j = 0; j < Math.PI * 2; j += Math.PI / 6) {
                StdDraw.filledCircle(Math.cos(j) * 100 + 200, Math.sin(j) * 100 + 200, 4);
            }

            StdDraw.show();
            StdDraw.pause(1000);
        }
    }
}

```

Figure 5.6: StdDraw animation example.

```

import objectdraw.*;

public class Clock {
    public static void main(String[] args) {
        AWTFrameCanvas canvas = new AWTFrameCanvas(400, 400);

        for (double j = 0; j <= Math.PI * 2; j += Math.PI / 6) {
            FilledOval dot =
                new FilledOval(Math.cos(j) * 100 + 196, Math.sin(j) * 100 + 196, 8, 8, canvas);
        }

        Line line = new Line(200, 200, 200, 100, canvas);

        for (double sec = -Math.PI / 2; true; sec += Math.PI / 30) {
            line.setEnd(Math.cos(sec) * 100 + 200, Math.sin(sec) * 100 + 200);

            ActiveObject.pause(1000);
        }
    }
}

```

Figure 5.7: ObjectDraw animation example.

```
import javax.swing.*;
import java.awt.*;

public class Clock {
    public static void main(String[] args) throws InterruptedException {
        JFrame frame = new JFrame();
        frame.setContentPane(new MyPanel());
        frame.pack();
        frame.setVisible(true);

        while (true) {
            Thread.sleep(1000);
            frame.repaint();
        }
    }

    private static class MyPanel extends JPanel {
        public MyPanel() {
            setPreferredSize(new Dimension(400, 400));
        }

        private double angle = -Math.PI / 2;

        @Override
        protected void paintComponent(Graphics g) {
            super.paintComponent(g);

            g.clearRect(0, 0, 400, 400);

            g.drawLine(
                200, 200,
                (int)(Math.cos(angle) * 100 + 200),
                (int)(Math.sin(angle) * 100 + 200)
            );

            for (double j = 0; j < Math.PI * 2; j += Math.PI / 6) {
                g.fillOval(
                    (int)(Math.cos(j) * 100 + 196),
                    (int)(Math.sin(j) * 100 + 196),
                    8, 8
                );
            }

            angle += Math.PI / 30;
        }
    }
}
```

Figure 5.8: Java AWT animation example.

```
void setup() {
  size(400, 400);
  frameRate(1);
}

float sec = -PI / 2;

void draw() {
  background(#FFFFFF);

  line(200, 200, cos(sec) * 100 + 200, sin(sec) * 100 + 200);

  for (float j = 0; j < PI * 2; j += PI / 6) {
    fill(#000000);
    circle(cos(j) * 100 + 200, sin(j) * 100 + 200, 8);
  }

  sec += PI / 30;
}
```

Figure 5.9: Processing animation example.

5.8 Interactive Program Comparison

User interaction is enabled by giving the user of the library the necessary information to change what is being displayed on the canvas based on the interaction of the user with the interface.

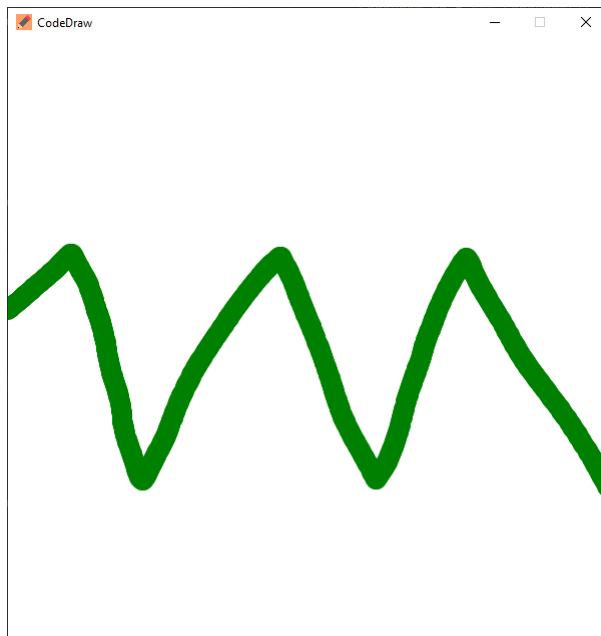


Figure 5.10: The typical output of an interactive graphics program in which the cursor draws a green line.

5.8.1 ObjectDraw

ObjectDraw exposes the Java AWT event listeners and users have to implement either a class or write a lambda expression. This can lead to concurrency issues since the Java AWT thread executes the event listeners while users implement their logic on the main thread. From the inside of a Java lambda expression a user cannot assign a value to a local variable that is on the outside of that expression. Users either have to implement global variables or implement objects with state.

```
import objectdraw.*;

import java.awt.*;
import java.awt.event.*;

public class YouDraw {
    public static void main(String[] args) {
        AWTFrameCanvas canvas = new AWTFrameCanvas(600, 600);

        canvas.addMouseMotionListener(new MouseMotionAdapter() {
            @Override
            public void mouseMoved(MouseEvent e) {
                FilledOval circle =
                    new FilledOval(e.getX() - 10, e.getY() - 10, 20, 20, canvas);
                circle.setColor(Color.GREEN);
            }
        });
    }
}
```

Figure 5.11: Implementing an interactive application in ObjectDraw.

5.8.2 Java AWT

Java AWT is also implemented through an event listener. The way in which this example is implemented avoids concurrency issues since the `paintComponent()` method and the `MouseMotionAdapter` both get executed on the same Java AWT thread.

```
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
import java.awt.image.BufferedImage;

public class YouDraw {
    public static void main(String[] args) {
        JFrame frame = new JFrame();
        frame.setContentPane(new MyPanel());
        frame.pack();
        frame.setVisible(true);
    }

    private static class MyPanel extends JPanel {
        public MyPanel() {
            image = new BufferedImage(600, 600, BufferedImage.TYPE_INT_ARGB);
            g = image.createGraphics();

            setPreferredSize(new Dimension(600, 600));
            addMouseListener(new MouseMotionAdapter() {
                @Override
                public void mouseMoved(MouseEvent e) {
                    g.setColor(Color.RED);
                    g.fillRect(e.getX() - 5, e.getY() - 5, 10, 10);
                    repaint();
                }
            });
        }

        private BufferedImage image;
        private Graphics2D g;

        @Override
        protected void paintComponent(Graphics g) {
            super.paintComponent(g);
            g.drawImage(image, 0, 0, Color.WHITE, null);
        }
    }
}
```

Figure 5.12: Implementing an interactive application in Java AWT.

5.8.3 StdDraw

StdDraw does not have inversion of control, but offers simple static methods that can be called at any time, e.g., to check whether the mouse is pressed `StdDraw.isMousePressed()`. The main advantage of this approach is that it is easily understood. The disadvantage is events are often lost and the program seems buggy when it does not react.

There are several reasons why events are lost in StdDraw:

If the main thread is suspended for a short moment all events during that time are lost.

If the user decides to pause after rendering events happening during that time are lost.

If some part of the program takes too long to calculate events during that time are lost.

```
public class YouDraw {
    public static void main(String[] args) {
        StdDraw.setCanvasSize(600, 600);
        StdDraw.setPenColor(StdDraw.GREEN);
        StdDraw.enableDoubleBuffering();

        while (true) {
            StdDraw.filledCircle(StdDraw.mouseX(), StdDraw.mouseY(), 10D/512);
            StdDraw.show();
            StdDraw.pause(16);
        }
    }
}
```

Figure 5.13: Drawing a circle wherever the mouse moves in StdDraw with double buffering.

```
public class YouDraw {
    public static void main(String[] args) {
        StdDraw.setCanvasSize(600, 600);
        StdDraw.setPenColor(StdDraw.GREEN);

        while (true) {
            StdDraw.filledCircle(StdDraw.mouseX(), StdDraw.mouseY(), 10D/512);
        }
    }
}
```

Figure 5.14: Drawing a circle wherever the mouse moves in StdDraw without double buffering.

There are two possible implementations for reading the mouse input. One with double buffering in figure 5.13 and the other one without in figure 5.14. Both lose a large amount of mouse motion events since the main thread can only poll to see where the mouse is at that moment. The double buffering approach seems to work better. In both implementations the first render will also create a circle at the (0, 0) coordinate since

there is no value for the mouse position if the mouse has not yet been moved over the StdDraw window.

When reading text from StdDraw or Draw there is an interface that is somewhat similar to the `java.util.Scanner` class. A user can use `StdDraw.hasNextKeyTyped()` and `StdDraw.nextKeyTyped()` to read characters but neither is blocking, which differs from the implementation of the `Scanner`.

5.8.4 Processing

In Processing a similar solution to the one in StdDraw *can* be implemented since the mouse position can be accessed while drawing. This creates the same issue StdDraw has where many events are lost and a circle is drawn at the beginning at the (0, 0) coordinate.

```
void setup() {
    size(600, 600);
    background(#FFFFFF);
}

void draw() {
    noStroke();
    fill(#00FF00);
    circle(mouseX, mouseY, 20);
}
```

Figure 5.15: Implementing an interactive application in Processing (suboptimal).

Since Processing uses inversion of control, there are event methods which can be overridden. When using those methods no events are lost and no circle is drawn at the (0, 0) coordinate. Events are handled on the same thread that calls the `draw()` and the `mouseMoved()` method. It is also easy for users to implement that kind of event handling because of the custom syntax of Processing.

```
void setup() {
    size(600, 600);
    background(#FFFFFF);
}

void mouseMoved() {
    noStroke();
    fill(#00FF00);
    circle(mouseX, mouseY, 20);
}
```

Figure 5.16: Implementing an interactive application in Processing (ideal).

5.8.5 CodeDraw Animation

When the `Animation` interface is used, method overriding can be used to handle events. There is no way to access the canvas directly in overridden event methods. The events either have to be saved in object variables or processed and then saved in the object variables. In the example below the move events are saved in the `image` object variable. The `Animation` interface has no concurrency issues either since it is executed on the main thread and uses the `EventScanner` in the background.

```
import codedraw.*;

public class YouDraw implements Animation {
    public static void main(String[] args) {
        CodeDraw.run(new YouDraw(), 600, 600, 60);
    }

    private Image image = new Image(600, 600);

    @Override
    public void draw(Image canvas) {
        canvas.drawImage(0, 0, image);
    }

    @Override
    public void onMouseMove(MouseMoveEvent event) {
        image.setColor(Palette.GREEN);
        image.fillCircle(event.getX(), event.getY(), 8);
    }
}
```

Figure 5.17: Implementing an interactive application in CodeDraw using the `Animation` interface.

5.8.6 CodeDraw

Plain CodeDraw has the EventScanner, which is essentially just a blocking concurrent queue. In an endless loop each frame is drawn. In the inner loop before calling `show()`, all currently available events are consumed. The user then has to check which events are at the front of the queue. To check whether a mouse event is in the front of the queue `hasMouseMoveEvent()` can be called and then `nextMouseMoveEvent()` has to be called to retrieve the event from the front of the queue. Events that are not used also have to be discarded in the else branch with `nextEvent()`. The EventScanner is written in a thread safe way to avoid any potential concurrency issues.

```
import codedraw.*;

public class YouDraw {
    public static void main(String[] args) {
        CodeDraw cd = new CodeDraw();
        EventScanner es = cd.getEventScanner();

        cd.setColor(Palette.GREEN);

        while (!cd.isClosed()) {
            while (es.hasEventNow()) {
                if (es.hasMouseMoveEvent()) {
                    MouseMoveEvent a = es.nextMouseMoveEvent();
                    cd.fillCircle(a.getX(), a.getY(), 10);
                } else {
                    es.nextEvent();
                }
            }

            cd.show(16);
        }
    }
}
```

Figure 5.18: Implementing an interactive application in CodeDraw.

CHAPTER 6

Conclusion

CodeDraw offers everything you need for students to take the first step in creating graphical output. The overhead is minimal, even someone who does not know programming at all could create simple static images. CodeDraw hides the complexities as much as is reasonable and everything is documented with descriptions and examples. If some time is spent explaining the EventScanner even CS1 students can write interactive programs and small games, all without writing functions or classes. It also supports inversion of control which reduces the amount of control structures the user has to write.

Thank you for reading my thesis and I hope you have fun programming with CodeDraw!

List of Figures

| | | |
|------|---|----|
| 2.1 | An example of how to draw basic shapes using CodeDraw. | 3 |
| 2.2 | One frame of the clock animation in CodeDraw after 3 seconds. | 4 |
| 2.3 | How to save a PNG image to the file system. | 6 |
| 2.4 | An image of a plant being edited using CodeDraw. This image was taken by Nikolaus Kasyan. | 7 |
| 2.5 | Different display options in CodeDraw. | 7 |
| 2.6 | An example of matrix multiplication. | 9 |
| 2.7 | Rotating text by 45° around the (100, 100) coordinate in CodeDraw. . . . | 9 |
| 2.8 | Using linear transformation to create a coordinate system that goes from 0 to 1. | 10 |
| 2.9 | An example of how text can be stylized in the CodeDraw library. | 11 |
| 2.10 | CodeDraw with a loading cursor. | 12 |
| 2.11 | A CodeDraw window positioned 100 pixel away from the top left corner of the main screen. | 12 |
| 2.12 | A CodeDraw window positioned so that the (0, 0) coordinate of the canvas is 100 pixels away from the top left corner of the main screen. | 13 |
| 2.13 | The EventHandler interface. TSender is always the CodeDraw class and TArgs are the event arguments. | 13 |
| 2.14 | A simple EventScanner program that draws a red square wherever the mouse moves. | 14 |
| 2.15 | The output of the EventScanner program in figure 2.14 when moving the mouse over the CodeDraw window. | 14 |
| 2.16 | An enhanced EventScanner program that draws a red square wherever the mouse moves. | 16 |
| 2.17 | An example output of the particle program in figure 2.18 being executed. | 17 |
| 2.18 | This program produces particles that follow the mouse cursor. The closer the mouse cursor is to individual particles the faster they will move. | 18 |
| 2.19 | The same program as in figure 2.18 but written with inversion of control. | 20 |
| 2.20 | A segment of a dart board as an example for a custom shape. | 21 |
| 2.21 | An illustration on how the path in figure 2.20 would be drawn. | 21 |
| 3.1 | Games and patterns that are implemented using CodeDraw. | 26 |
| | | 49 |

| | | |
|------|---|----|
| 4.1 | Code that opens a CodeDraw window, draws a line and closes it after 2 seconds. | 28 |
| 5.1 | The typical output of each graphics library when drawing a red line and the words "Hello World!". | 33 |
| 5.2 | Producing a red line and the text "Hello World!" in different libraries. . . | 34 |
| 5.3 | The typical output of each graphics library when implementing the clock example. | 35 |
| 5.4 | CodeDraw animation example. | 36 |
| 5.5 | CodeDraw animation example using the <code>Animation</code> interface. | 36 |
| 5.6 | <code>StdDraw</code> animation example. | 37 |
| 5.7 | <code>ObjectDraw</code> animation example. | 37 |
| 5.8 | Java AWT animation example. | 38 |
| 5.9 | Processing animation example. | 39 |
| 5.10 | The typical output of an interactive graphics program in which the cursor draws a green line. | 40 |
| 5.11 | Implementing an interactive application in <code>ObjectDraw</code> | 41 |
| 5.12 | Implementing an interactive application in Java AWT. | 42 |
| 5.13 | Drawing a circle wherever the mouse moves in <code>StdDraw</code> with double buffering. | 43 |
| 5.14 | Drawing a circle wherever the mouse moves in <code>StdDraw</code> without double buffering. | 43 |
| 5.15 | Implementing an interactive application in Processing (suboptimal). | 44 |
| 5.16 | Implementing an interactive application in Processing (ideal). | 44 |
| 5.17 | Implementing an interactive application in CodeDraw using the <code>Animation</code> interface. | 45 |
| 5.18 | Implementing an interactive application in CodeDraw. | 46 |

Bibliography

- [BDM01] Kim B. Bruce, Andrea Danyluk, and Thomas Murtagh. A Library to Support a Graphics-Based Object-First Approach to CS 1. In *Proceedings of the Thirty-Second SIGCSE Technical Symposium on Computer Science Education*, SIGCSE '01, page 6–10, New York, NY, USA, 2001. Association for Computing Machinery.
- [RF14] Casey Reas and Ben Fry. *Processing: A Programming Handbook for Visual Designers*. MIT Press, 2nd edition, 2014.
- [Sch10] Daniel L. Schuster. CS1, Arcade Games and the Free Java Book. In *Proceedings of the 41st ACM Technical Symposium on Computer Science Education*, SIGCSE '10, page 549–553, New York, NY, USA, 2010. Association for Computing Machinery.
- [SW17] Robert Sedgewick and Kevin Wayne. *Introduction to Programming in Java: An Interdisciplinary Approach*. Addison-Wesley Professional, 2nd edition, 2017.