DIPARTIMENTO DI INGEGNERIA DELL'INFORMAZIONE

CORSO DI LAUREA IN INGEGNERIA DELL'INFORMAZIONE

ENHANCEMENT OF PERFORATED ROLLS FOR PLAYER PIANO: DESIGN AND DEVELOPMENT OF SOFTWARE FOR CONVERTING SCANS INTO MIDI FORMAT

Relatore: Prof. Sergio Canazza

Correlatore: Dott. Pierluigi Bontempi

Laureando: Federico Frontini

ANNO ACCADEMICO 2022-2023
Data di Laurea 29/09/2023

Abstract

Piano rolls are long strips of paper with perforations used in mechanical player pianos to produce music, but the frailty of the paper in which the rolls are stored and the need for expensive tools to reproduce them are limiting access to interested parties. There are two kinds of rolls: metronomic rolls which simply encode the timing and duration of notes, similar to sheet music, and reproducing rolls which also capture the subtle nuances of a pianist's performance.
In this thesis, we use high-quality scans of metronomic rolls provided by the laboratory of the Italian Mechanical Music Association, AMMILAB, and the source code of a similar project to develop in C# a software capable of converting those scans into MIDI files.
While at this point the software presented here is still in the early stages of development and should be quite similar to the one employed by AMMILAB in scope, a streamlined management of the code should make adding functionalities and updating the source code in the future significantly easier.

Acknowledgments

I would like to take a moment to extend my heartfelt appreciation to those without whom this work would not have come to fruition.
First of all, I am deeply grateful to Prof. Sergio Cannazza and Dr. Pierluigi Bontempi for their mentorship and guidance during the research process.
I am also extremely thankful to Prof. Pietro Zappalà and Dr. Matteo Malosio for their expertise and readiness to help me throughout the whole undertaking.
On a more personal note, I would not be here without the unwavering support of my family, especially my dearest grandmother, my father, my aunt, and my boyfriend.
My thoughts and prayers go to the memory of my late mother: I dedicate the end of this journey to her.

Contents

Abstract **I**

Acknowledgments **III**

1 Introduction: understanding piano rolls **1**
1.1 Piano rolls and player pianos 1
1.2 The historical relevance of the rolls 3

2 A brief history of the digitization of piano rolls **7**
2.1 Reasons for digitizing piano rolls 7
2.2 From early efforts to modern optical scanners 8

3 Choice of instruments **15**
3.1 C# and AvaloniaUI 15
3.2 The MVVM design pattern 17

4 Discussion of the code **23**
4.1 Handling .rcf configuration files 24
4.2 Why is it necessary to write a Logger 27
4.3 Contour detection using EmguCV 29

Appendixes **39**

A A brief overview of the MIDI protocol **39**
A.1 A bridge between music and computers 39
A.2 The fundamentals of MIDI protocol 41
A.3 MIDI messages 44

List of Figures

1.1 A reproducing piano and a player piano 2
1.2 Close up of a piano roll 3
1.3 A Haines Brothers Ampico reproducing piano 4

2.1 A Duo-Art Weber reproducing piano 9
2.2 The modern successor of ProRecord 10
2.3 The apparatus built by AMMILAB 12

3.1 The logic behind the MVVM design pattern 19

4.1 A section of perforated piano roll before processing 30
4.2 Greyscale and blurred section of a perforeted piano roll 31
4.3 The result of the findContour algorithm 32
4.4 findContour with a filter based on shape 33
4.5 findContour with a filter based on shape and distance 34
4.6 The filtered contours interpolated to follow the dynamic line 34

A.1 A perforated piano roll and a DAW digital workstation 40
A.2 A Minimoog, in its 2016 rerelease 42
A.3 A Minimoog, in its 2016 rerelease 43
A.4 Format of a NOTE ON message 44

List of Tables

3.1 C# versus Python 16
3.2 Qt versus Avalonia 18

Chapter 1

Introduction: understanding piano rolls

1.1 Piano rolls and player pianos

In his doctorate thesis[13], Peter Phillips simply defines piano rolls as a

> paper medium containing a recording, in which a roll is a single entity.

In more detail, piano rolls are a kind of symbolic musical storage medium consisting of a scroll of paper rolled onto a metallic cylinder, punched with holes representing musical notes, and designed to be played autonomously, with little human intervention. In conjunction with a piece of specialized equipment for reproduction, they were an early example of "Automatophonic"[7], a mechanism that was meant to replace the need of a musician but not the instrument itself.

As a means of bringing music to a broader audience, piano rolls were introduced in the late 19th century thanks to the invention of the pneumatic mechanism and gradually replaced earlier contraptions such as the perforated discs or the wood and metal barrels mostly used with music boxes or barrel organs. At a time when few people could actually play the piano or any kind of music, these early recordings could provide entertainment in the home of the upper middle class of the period.

To read and play these rolls one had to use a player piano, a self-playing acoustic piano whose keys could be moved mechanically. These instruments were not simply devices on which to listen to music, but fully functioning pianos as well. During playback though, the instrument reproduced the music as the roll was

Figure 1.1: *A reproducing piano and a player piano*

unfolded. Where the holes appear in the paper dictates the notes the player piano would procuce. When air passes through the holes, it generates a pneumatic force that mechanically pushes the piano keys or the pedals according to the position of the holes themselves.

A human operator was only needed to manually power the pneumatic pump, at least until the introduction of electric motors completely eliminated the need for human intervention.
The earliest kind of piano rolls, called metronomic piano rolls, while relatively inexpensive to produce presented the significant shortcoming of not being able to contain expressive dynamics and sported a limited musical range; all notes played with the same volume, severely limiting the expressiveness and pleasantness of the reproduced music. These rolls were often just metronomic transcription of the score and were intended to be played on foot-pumped pianos by a so-called pianolist, whose job was to transform the mechanical performance into music. They were also generally crafted meticulously by hand to give the pianolist the chance to bring their own interpretation to the piece without compromising its accuracy. A convention, attended by representatives of the roll-making industry and by player piano manufacturers, was held December 10, 1908, in Buffalo, New York, at the Iroquois Hotel where it was decided the standards for

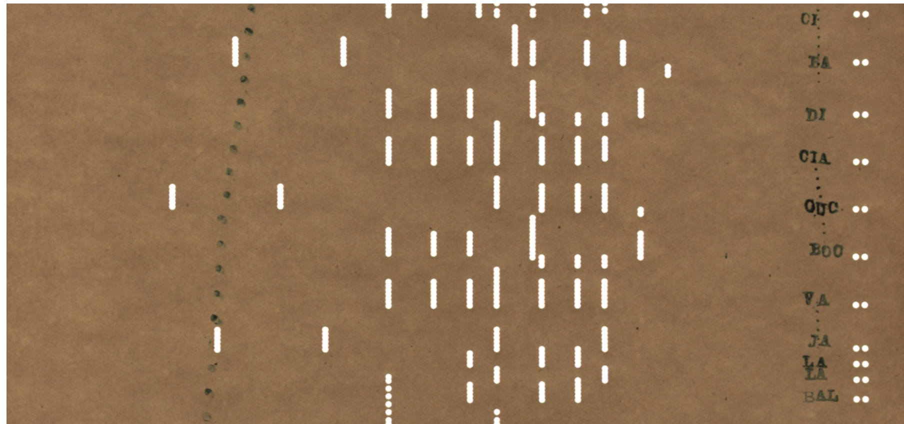

Figure 1.2: *Close up of a piano roll*

88-note piano roll hole spacing[5]. After that date, two standards remained into use, the aforementioned 88-note, and a cheaper 65-note. In both cases, the width of the roll was set at 286 millimeters, making it possible for either kind of roll to be played on any player or reproducing piano.

1.2 The historical relevance of the rolls

In truth, the piano rolls reached widespread popularity from the early 20th century to the 1930s mainly due to the introduction of reproducing rolls that could reproduce notes and the pedals and dynamic variations, narrowing the gap between reproduced music and live performance and generally employing an electric pump so that they could be played completely autonomously.
Generally speaking, a piano roll is divided into three groups of holes with a broad central area controlling the pitches and duration of the notes, a left edge activating the bass notes and the sustain pedal, and a right edge in charge of the treble notes' expression. In the roll's central area, pitches increase from left to right, similar to a piano keyboard. A note's duration depends on the length of its chain of holes, which may be continuous or with a minimum distance between subsequent holes usually set at 1mm. These bridges leave such thin paper between subsequent holes that they never block the passage of air during reproduction and so are invisible to the tracker bar. The first hole in a vertical chain matches a key-down event and the end of the chain is a key-up event. We can conveniently visualize a piano roll with a Cartesian plane or a matrix, with notes and other

functions on the x-axis and the performance progression on the y-axis. Each moment in time can thus be summarized just by reading the corresponding row on the roll.
Some reproducing rolls also had three columns on both edges of the piano roll that managed an ingenious volume-level regulation mechanism. These ports could generate eight different volume levels independently at each side of the roll, and it was also possible to increase and decrease the volume step by step.
During playback, the roll moves at a certain speed printed at the beginning of the roll in units of feet per ten minutes and it is important to note that this measure of tempo might not match the more usual beats per minute.
More so, while the paper is being pulled through the piano, as the roll builds momentum the speed increases and needs to be compensated. Thus, before reproducing the roll a device was needed to adjust the speed according to the specific roll, with a technician manually adjusting a lever in order to do so.

Figure 1.3: *A Haines Brothers Ampico reproducing piano*

A reproducing roll could be obtained with a recording piano operated in tandem with the pianist by a technician that could capture in real-time the performance of the artist down to the finest details and with a high degree of fidelity.
It is this capacity to reproduce the performance of the pianist that attracted the interest of some of the most famous composers and pianists of their time who decided to record their own work. Thanks to the reproducing rolls we have recordings of Debussy, Ravel, Mahler, Richard Strauss, Rachmaninoff, Josef Lhévinne, and Moritz Rosenthal among others making them the oldest generation of pi-

anists ever recorded.[14]
Even so, there is a certain mistrust from researchers and enthusiasts alike toward piano rolls, especially when used to research piano practices of famous pianists[8]. The reason for this is that piano rolls are often heard through recordings made on poorly adjusted instruments or not perfectly reconditioned player pianos. At the time though, competing early acoustic recordings could not match the high quality of a live piano for playback as they were crippled by poor sound quality, high noise, and limited recording length. Due to the limitations of the audio recording technology of the time, a number of artists didn't record their work this way, further elevating the piano rolls importance to historians and musicologists alike.
These advantages of reproducing piano rolls over acoustic recordings ensured significant commercial success in the first decades of the 20th century with competing companies producing their own reproducing roll systems. The most important of these include Welte-Mignon, Hupfeld, Ampico, Duo-Art and Philips Duca but due to variations in the size of the roll, configuration of holes, etc., each system is incompatible with the others. Playing any reproducing roll requires a suitable player piano specifically manufactured for that format of roll. Even worse, the formats further evolved over time so that later rolls by the same manufacturer simply do not work on older reproducing pianos further crippling potential interest in researching their importance and value both as an historical medium and as a musical record.

Chapter 2

A brief history of the digitization of piano rolls

2.1 Reasons for digitizing piano rolls

Museums and enthusiasts alike keep large collections of piano rolls, in varying degrees of conservation. In Europe, the two largest collections are the one hosted at the Musée Mécanique in Paris and the one displayed in the Historisches Museum of Basel. Despite their historical significance and the relative availability of large collections of piano rolls, there are two main difficulties preventing scholars from accessing and studying them.

First of all, piano rolls are made of paper. The frailty of the material is often a problem for historians and there is a very real possibility of tearing the roll by mistake even with the utmost care. Worse, many of the rolls are already brittle and damaged, at risk of quite literally disintegrating. It is no surprise then that curators of larger collections or in possession of certain uncommon piano rolls are often hesitant to allow further study.

The second issue is more practical. Unfortunately, having a piano roll and a reproducing piano is not enough to assure good audio quality, let alone a standard piano: a reproducing piano in very good condition is needed, which is possible only with a restored or reconditioned instrument. It is not cheap nor easy to find one, which is a hard sell for prospective scholars, and the number of restored player pianos is also decreasing over time.

Digitizing piano rolls may help in solving both issues and at the time of writing, the process can be summarised as follows: the roll is first spooled through a scanning device to produce a picture that can be digitally restored or enhanced.

The roll as it spools triggers the scanning device, building up a two-dimensional image one row at a time that may be full color or greyscale. Various methods have been tried with optical scanning through a repurposed CIS module being the most straightforward[17]. For most purposes, a greyscale image is more than enough, both to contain costs and file size. As we shall see in the next part of the thesis, for the conversion of the piano roll scan to MIDI format it is enough to have a greyscale image as it is better recognized by computer vision algorithms[11].
A light source may be placed behind the roll so that the holes appear white if it benefits the scanning process. The apparatus developed by AMMILAB can work both ways, even though the process bears better results with a light source in place.
This certainly serves the purpose of archival preservation but does not allow piano rolls to be heard as the musical documents they were intended to be. After the scan, the roll can be digitally repaired and printed anew or, as it shall be discussed in this thesis, it may be further processed into a MIDI file.
Scanning a piano roll is by itself a gentler process than playing it and the obtained scans can then be used to create brand new rolls or they can be further processed into MIDI files to reach a wider audience (as discussed earlier) with the clear advantage of not having to worry about compatibility between brands and models. As long as the holes are not obscured or torn apart, it is possible to scan the piano roll without expensive repair work. In the case of reproducing rolls, the note dynamics are controlled by the tracks down the sides and an emulator is necessary to mimic the physical process that controls the suction levels and the velocity with which the hammers hit the strings[19].

2.2 From early efforts to modern optical scanners

Enthusiasts of piano roll technology have been trying to digitize and convert piano rolls since the 70s, at first only to obtain functional libraries for their Ampico or Duo-Art reproducing pianos.

Early experiments with fiber optics were immediately recognized as a dead end but the development of the MIDI protocol and the commercial success of the Yamaha Disklavier introducing another form of reproducing piano in the market revitalized research and interest in the matter. In 1976, Wayne Stahnke managed to successfully digitize piano rolls with a pneumatic-electric roll reader to be stored on cassettes and launched this system, called the Anpico CC-3, on the market. In the same years, three pneumatic roll readers were used to convert 2500

Figure 2.1: *A Duo-Art Weber reproducing piano*

reproducing and player rolls for the Pianocorder library, one of the first solenoid-operated pianos. Despite the technical achievement, however, that conversion had a low sampling rate, meaning the converted data lost expressiveness.
The MIDI protocol introduction in late 1983 helped bring new interest in the piano roll scene. When Yamaha bought and shut down the company marketing the Pianocorder and started refusing to sell Disklavier player pianos to third parties, an American company called PianoDisc developed their retrofit system for acoustic pianos by the same name. Unlike the Disklavier, it could fit any piano, not just Yamaha's, and it was cheaper. The PianoDisc needed a library of high-quality music to showcase the capabilities of this retrofit system and for sale. Thanks to a specific device called ProRecord, a MIDI record strip under the keys of an Ampico reproducing piano, the company digitized their piano rolls and started selling this so-called Masterpiece Collection on floppy disks. These are still on

sale today.

Figure 2.2: *The modern successor of ProRecord*

It is highly likely that both PianoDisc recordings and those of yet another new competitor trying to insert itself into this niche but growing market, the QRS Pianomation player System, were recorded by someone pedalling a push-up pianola sitting before a MIDI piano. Unfortunately, these records were still underwhelming quality-wise in part due to how quickly QRS and Burgett Inc needed a library to support their efforts against Yamaha: the problem was yet again achieving the same quality of the reproducing rolls on the new MIDI instruments as they would be heard on reconditioned pneumatic instruments.
Even though the MIDI conversions of piano rolls throughout the 80s were still low quality, there was finally a serious alternative to pneumatic reproducing pianos, lowering the barrier to further studies from interested parties that didn't have access to an original reproducing piano. An original restored reproducing piano was no longer mandatory, one had merely to adapt a retrofit system such as the Pianomation and the PianoDisc player system to a standard acoustic piano. In 1993, a project led by Artis Wodehouse involving the conversion to MIDI files of a handful of Gershwin's piano rolls took place, thanks to the interest of the

University of Michigan, the Gershwin family, and Nonesuch Records. The process involved restoring and digitizing rolls recorded by Gershwin in the 1910s, subject to wear due to repeated use. Researchers took high-res photos of each roll before digitally enhancing their quality and accuracy and converting them into MIDI data which was in turn used to recreate the original performance on a modern player piano. Even though the rolls were still serviceable and thus it would have been possible to merely insert them into a reproducing piano system replicating key and pedal movements, this approach aimed to overcome the limitations of the medium while maintaining the musical content and nuances of Gershwin's original performance achieving the highest level of audio quality and fidelity.

It is of note that American software developer Richard Brandle specifically wrote software to create the dynamics of each playing note and this process in the contest of digitalizing piano rolls is now commonly referred to as emulation. An emulator's task is to mimic this process to produce comparable note velocities in the MIDI files. Brandle then started to sell this software called WindPlay, incorporating emulators for Ampico, Duo-Art, and Welte rolls. The software was somehow inconvenient to users as it required the purchase of a companion program called Wind that had to be used to convert MIDI files into the proprietary bar/ann standard.

In 1996, the first optical roll scanner based on a Logitech handheld paper scanner was developed by Stahnke and Richard Tonneson. These scans were used to generate punch-for-punch replica rolls of the digitized rolls. This is especially of note because, in the following years, the standard approach to digitizing piano rolls would include optical roll scanners, even though the punch matrix computer files and the bar/ann standard fell out of fashion.

In 2001, Richard Stibbons founded the International Association of Mechanical Music Preservationists (IAMMP), promoting the use of flatbed scanners to convert and digitize piano rolls to MIDI standard files, an approach that has been followed since with the Italian Mechanical Musical Association and the University of Pavia supporting a research group that eventually developed the first optical scanner to use a digital camera called the SISAR project. This scanner was used to digitize the roll collection of the University of Pavia and its blueprints were the basis for a similar project by Anthony Robinson and Stanford University in 2014. Both projects are notable because previous attempts commonly used obsolete repurposed Mustek A3 flatbeds, nothing more than rows of sensors, a rod len, and a strip light. The main limitation of this kind of scanner is the depth of focus, very narrow. Unfortunately, one needs a large enough flatbed scanner to

repurpose, which is starting to get hard to find. AMMI lab decided instead to use a conventional camera, with the piano roll placed at some distance from the camera. The distance at which the camera is placed can be adjusted, so wider rolls can be processed if the needs manifest.

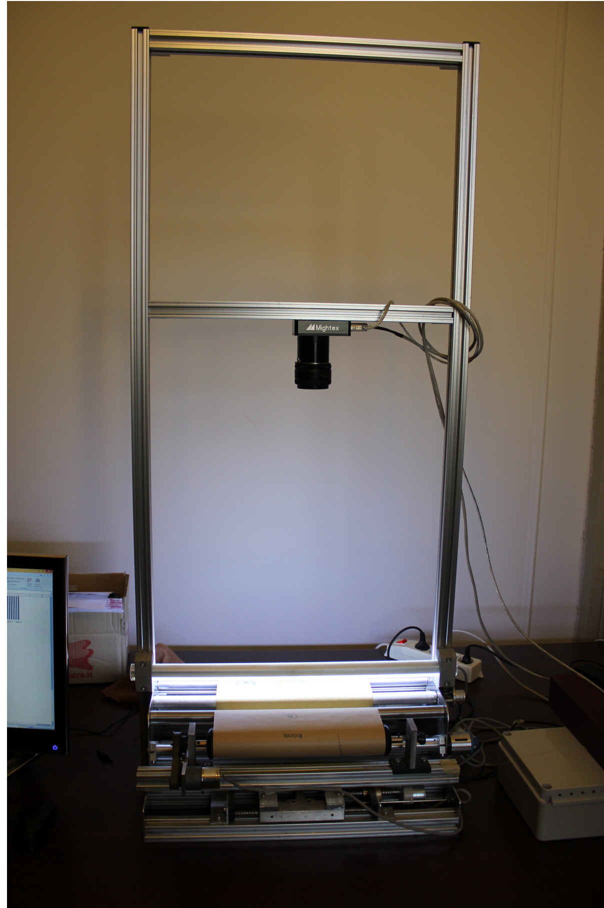

Figure 2.3: *The apparatus built by AMMILAB*

Regardless of the approach used to obtain the scanned image, the process of conversion is straightforward. The first step is detecting all the punches and recording their position one column at a time. Since the columns are relatively well spread this is not hard to do and the results are the so-called Raw files. If one only needs to copy the original rolls, these Raw files are usually more than enough even though the new punched roll might inherit some inaccuracies and they are not suited to be played electronically. The harder part is to assign each

punch to its row, a step commonly referred to as Punch Matrix Recovery. This is considerably more challenging because in the original roll, holes might overlap with one another, the rows may be skewed or the row spacing may be uneven. Interestingly enough, if this step is performed with care the obtained files known as Punch files can be used to create copies of the original roll with perfectly accurate rows and columns. There are currently two software used for Punch Matrix Recovery, one by Warren Trachtman and the other by Wayne Stahnke. The former, while easier to use is also inaccurate and thus completely unusable so there is no choice unless one decides to do it from scratch. Stahnke's software is also prone to making mistakes and there is often the need for a visual inspection for manual correction. At this point, the next step involves offsetting events in the scanned image to correspond with the periods when they pass over the tracker bar ports to generate Bar files. These Bar files can be played by retrofitted player pianos as a cheaper alternative to original rolls and emulated MIDI files may be played on digital synthesized pianos or modern solenoid-operated pianos such as the aforementioned Yamaha Disklavier.

Chapter 3

Choice of instruments

3.1 C# and AvaloniaUI

At the time of writing, there are various digital archives of piano rolls[22] and a handful of projects similar to the one presented on these pages, at various degrees of usability.
Why then start a new project from scratch instead of working on the software still employed by AMMILAB? The main reason is that the original code is very hard to maintain and change, let alone build upon. Instead, we aim to offer a strong foundation for future releases and even though we are going to ship the software with far fewer functionalities than the AMMILAB software at first, it should be considerably easier to add more along the line. How exactly we are going to achieve just that will be explained in the next section.
Going back to the language of choice, for this specific project, we feel that C# has its own set of advantages over Python, the programming language AMMILAB software is written with. Being a compiled language means faster execution time and better performance, making C# a solid choice in spite of Python's ever-growing pool of useful libraries that make writing passable code easier. Python is usually favored for its concise syntax, versatility, ready-to-use libraries and cross platform compatibility but in switching to C#, we aim to avoid incurring in dependencies issues that are common in working with small-scale Python projects or having to rely on third-party libraries that may or may not be maintained in the future.
It is of note that this issue is inherent to the black box programming approach and that every developer has to deal with it to some degree. As software developers, we are used to exploiting someone else's code at our heart's content,

C#	**Python**
Developed by Microsoft. Comes with the license.	Open-source development and distribution.
Based on OOP concepts.	Supports multi-paradigm programming (OOP, procedural).
Supports work on .NET framework.	Can be integrated with Java (JVM), .NET, C and JavaScript.
Dependency injection is easy out of the box.	No concept of DI.
Because of the Common Language Infrastructure (CLI) framework, C# is faster and offers better performance.	Lackluster performance.
Multi-threading is quite easy using the .NET framework.	Because of the Global Interpreter Lock (GIL), multithreading requires multiple processes.

Table 3.1: *C# versus Python.*

thinking of their classes and methods as nothing more than a box with preconditions and postconditions. But what if that code gets deprecated and we suddenly need to do the heavy lifting ourselves? We may or may not be able to fix new issues as they arise. That's why it is imperative when starting a new project to be very wary of importing libraries and packages to our code in order to keep dependencies to a minimum.

The only library we really need is OpenCV[12], which does not support C# out of the box. To tackle this issue, we are going to make extensive use of an open-source wrapper, EmguCV[6], to make it work with our C# code. At first glance, the need to use a wrapper may seem a significant downside, but OpenCV was originally written in C and C++ and only works with Python code out of the box thanks to its own officially supported wrapper. While not official, EmguCV is both open source and well-maintained since its first release in 2008 and it's thankfully not overly hard to set up the first time. The community is also quite active and fast to solve issues on Github making EmguCV a safe choice to get the best of both worlds: a fast compiled language and a powerful library to take advantage of without having to worry too much about it getting deprecated quickly.
The second reason for switching to C# is portability. Thanks to the .NET framework and ecosystem it is quite easy to port code from different architectures with

minimal modifications to the source code and we have been extensively using containers paired with a simple CI/CD pipeline to ensure painless debugging, especially in the early stages of development. While this is by no means a novel concept and not unique to C#, far from it[9], it is nonetheless a useful best practice that can ease the burden of adding more functionalities on top of preexisting work.
Regarding portability in particular, another limitation of the AMMILAB software is the use of Qt for its UI module. Qt is primarily a C++ framework and while it can work with Python thanks to its PyQt binding, it is far from ideal and relegates the project to the Microsoft Windows ecosystem. Since we are switching to C# it has been decided to make use of the common ground provided by the .NET framework and change UI technology to Avalonia.
Avalonia is open source, lightweight, and easy to set up and run and it's also an UI tool that has a strong focus on building cross-platform desktop applications. It also supports the MVVM pattern design approach thanks to the powerful ReactiveUI framework which we favored over CommunityToolkit.Mvvm for its focus on easy multithreading and concurrent programming which can be crucial for building a responsive application, something that Python is amazing at despite GIL preventing true parallelism and that we felt the need to somehow replace.

The common ground in choosing Avalonia and C# is that both have a strong emphasis on performance and in trying to trim anything that might encumber the software the hope is to pave the way for a modular approach to development and support future releases down the line while favoring a platform-agnostic development experience.

3.2 The MVVM design pattern

In this section, we aim to provide some context to our claim that in choosing Avalonia/ReactiveUI we would be able to improve the software capabilities of getting more functionalities down the line. The need to properly separate UI and underlying code logically and semantically for testing and maintaining purposes is indeed significant and properly addressing them can be a monumental task[20].
Developers often structure their code without a specific design pattern in mind, writing out line after line of code, and eventually their project reaches a critical point where the complexity and intricacies of classes and methods overwhelm them and they need to refactor their work, if possible. Structuring the project

Strengths of Qt	**Strengths of Avalonia**
C++ framework with a wide range of features and tools for GUI applications.	Open-source framework for building cross-platform desktop applications in C#.
Strong integration with C++ and support for a wide variety of platforms, including Windows, Linux, macOS, mobile, and embedded systems.	Designed specifically for C# developers bused upon a XAML-based markup language.
Wide selection of customizable UI elements and themes.	Supports MVVM (Model-View-ViewModel) architecture.
Qt Creator IDE provides its own development environment	Supports Visual Studio and other C# development environments, making it more accessible
Well-documented with a large and active community	Open-source and community-driven with growing support and an expanding ecosystem.
QML provides a powerful way to create fluid and responsive user interfaces using a declarative language.	Offers a similar approach with XAML for building cross-platform UIs with a focus on flexibility and performance.
Extensive capabilities for application deployment and distribution.	Cross-platform support for Windows, Linux, and macOS

Table 3.2: *Qt versus Avalonia.*

with a specific design pattern in mind form the very beginning can ease this issue to the point of never feeling it at all.
In the case of UI-creation technologies, the mistake to avoid is letting the UI handle more than it should, without a clear separation of responsibilities from the code-behind layer. This is especially bad since code referenced in the UI layer of an application is very hard to test and debug without running the application or launching scripts to automate the task. It also makes it more difficult to refactor the code, as the logic to perform a business function may be copied or cross-referenced anywhere in the code [4].
Between 2004 and 2006, Jean-Paul Boodhoo, John Gossman and Martin Fowler each presented their own version of something quite similar, a design pattern to promote a clear separation of scope between these two layers, the View Layer and the View-Model layer that is between the UI and the code-behind. The gist of it is that the code-behind does not need to see the view to be effective. The view binds to properties on a ViewModel, which, in turn, exposes data contained in model objects and other state-specific to the view. It is of paramount importance that the model and the view-model are unaware of the UI layer. By binding properties of a view to a ViewModel, you get loose coupling between the two and entirely remove the need for writing code in a ViewModel that directly updates a view. It should be noted that this is no mere incapsulation: separating the business logic from the UI overhead let us test and debug our code without even being aware of our prospect users which in turn helps reducing the effort of mantaining and upgrading our code.

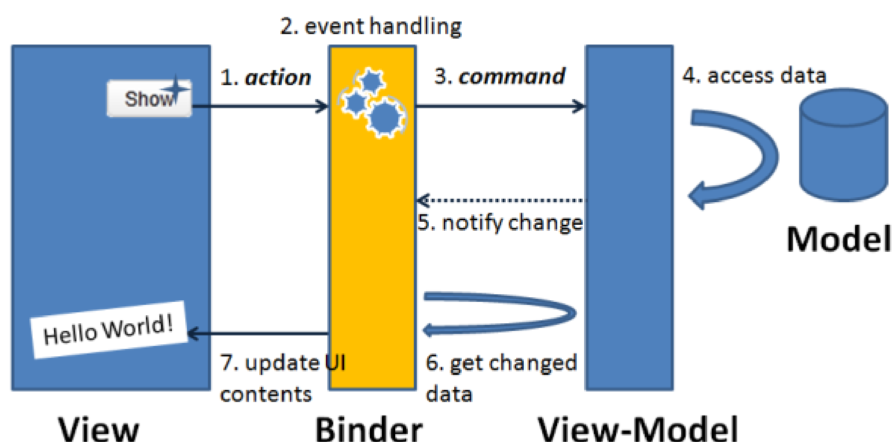

Figure 3.1: *The logic behind the MVVM design pattern*

In addition, the app UI can also be redesigned without touching the view model and model code, provided that the view is implemented entirely in XAML or C#. Therefore, a new version of the view should always work with the existing view model without even touching a single line of code behind it. While this is not our case study, it is also of note that when tackling a bigger project, designers and developers can theoretically work independently and concurrently on their components during development. Designers can focus on the view, while developers can work on the view model and model components.
The MVVM pattern is well established in .NET, and the community has created many frameworks that help ease this development. While it is not necessary to use these frameworks, they can speed up and standardize mantaining the code, even though they do slow initial releases. Let us focus on each of the three components of the MVVM design pattern, then. For a more in-depth coverage of this topic please refer to the excellent [23].
The view is responsible for defining the structure, layout, and appearance of what the user sees on screen. Ideally, each view is defined in XAML (or in the case of Avalonia, AXAML) with a limited code-behind that does not contain business logic. There are several options for executing code on the view model in response to interactions on the view, such as a button click or item selection. If a control supports commands, the control's Command property can be data-bound to an ICommand property on the view model and we shall see that this will actually be the case with Avalonia. Behaviors can be attached to an object in the view and can listen for either a command to be invoked or the event to be raised.
The view model implements properties and commands to which the view can data bind to and notifies the view of any state changes through change notification events. The properties and commands that the view model provides define the functionality to be offered by the UI, but the view determines how that functionality is to be displayed. Multithreading happens here, with frameworks and languages capable of parallelism using asynchronous functions to improve performance.
Model classes are non-visual classes that encapsulate the app's data. Therefore, the model can be thought of as representing the app's domain model, which usually includes a data model along with business and validation logic.
Once we do have the UI up and running though we need it to react to users' input, and to help mantaining the code in the future we want to avoid hardcoding where possible. ReactiveUI[16] is right tool for the job. It is a composable, cross-platform model-view-view model framework for all .NET platforms, that is inspired by functional reactive programming, which is a way of coding with

asynchronous data streams[21]. Buses or click events are really an asynchronous event stream on which you can observe and do some side effects. In this case, loading the configuration files and then processing the image with the user intervention before the conversion into MIDI files. We capture these emitted events only asynchronously, by defining a function that will execute when a value is emitted, another function when an error is emitted, and another function when 'completed' is emitted. Reactive Programming raises the level of abstraction of the code so we can focus on the interdependence of events that define the business logic, rather than having to constantly fiddle with a large amount of implementation details. In truth, the benefit is more evident in modern webapps and mobile apps that are highly interactive with a multitude of UI events related to data events. Implementing ReactiveUI in this particular project should help in maintaining the code if and when the software grows in functionalities and needs to handle interaction with the users better. At this point in time it has been added to the code only in minimal part at the Model View layer.
Clearly, this approach needs to merge view and models at runtime if it wants to succeed. There are two ways in which to tackle the challenge: a view first composition or a model first composition. For this project, we decided to follow a view first composition where we follow the visual structure of the app since we deemed it easier than the alternative which was to follow code at runtime to better understand its inner workings. By following the principles outlined here, view models can be tested in isolation, therefore reducing the likelihood of software defects by limiting scope. Similarly, view models can be created declaratively or programmatically. We decided to build them declaratively, as it feels natural to instantiate a view model object when the corresponding view gets constructed.

Chapter 4

Discussion of the code

The majority of the work done here is just reaping rewards from the careful planning done in the previous sections. At this point, the software can and should be developed in watertight compartments, ideally with team members tackling different parts of the project according to their strengths. Thankfully, the MVVM design pattern let us do just that. We can now work on the Model layer where we started tackling the computer vision part of the project and the Util layer where we developed a useful Logger that should prove beneficial when it is time to add more functionalities.
The structure of our project looks like this:

```
C:.
|   App.axaml //Avalonia XAML
|   App.axaml.cs
|   app.manifest
|   PianoRollMIDIConverter.csproj //config files for dependencies
|   Program.cs
|   README.md //check it out on GitHub
|   tree.txt //this file
|   ViewLocator.cs
|
+---.vscode
|       launch.json
|       tasks.json
|       //cache, user-specific settings
|
+---Assets
```

```
|       //... assets for UI
|
+---bin //binaries
+---Models
|       ConfigHandler.cs
|       ImageHandler.cs
| //...
|
+---obj
+---Tests //a scripting folder for debugging
|   |   main.csx
|   |   omnisharp.json
|   |   Sandbox.cs
|   |
|   \---.vscode //do not commit this folder, a .gitignore file should be added
|           launch.json
|
+---ViewModels
|       MainWindowViewModel.cs
|       ViewModelBase.cs
|       // ...
|
\---Views
        MainWindow.axaml
        MainWindow.axaml.cs
       // ...
```

4.1 Handling .rcf configuration files

The MIDI conversion software developed by AMMILAB uses .rcf files to handle configuration. Before attempting a conversion, the user has to manually select the correct configuration. If the conversion is not successful, the software will crash. There are several such files, each tailored for a specific brand of perforated piano rolls. For specific rolls, there are even custom configurations. Inside these files, there are a number of parameters ranging from the holes ratio to the roll velocity. While the number of parameters does vary, to our knowledge the .rcf file will always start with the string "[ROLL_CONFIG]". In order to streamline

this part of the software, it was decided to move from RCF/SHC files to simple JSON objects that can be manipulated later on. We proceeded to write a simple C# class to convert those .rcf files to the JSON object this software will use to handle configuration. For the serialization/deserialization of the JSON objects we decided to use Json.NET, a third-party library often employed for its highly efficient algorithms that should help in trimming any overhead. The code presented here only considers relevant code that needs commenting, for a better look at the whole project please refer to the repository on GitHub.

```
//... environment ...

namespace PianoRollMIDIConverter.Models
{
    public class ConfigHandler
    {
        private readonly string filepath;

        public ConfigHandler(string filepath)
        {
            this.filepath = filepath;
        }

        protected async Task<Dictionary<string, string>> ConfigParser(
        string filepath
        )
        {
            var config = new Dictionary<string, object>();
            string fileDump = "42"; //magic number for csc
            try
            {
                fileDump = await File.ReadAllTextAsync(this.filepath);
            }
            catch (Exception e)
            {
            // ...
            }
            string[] lines = fileDump.Split('\n');
            lines = lines.Select(line => line.Trim()).ToArray();
            JObject jsonObject = new JObject();
```

```
            foreach (string line in lines)
            {
                // ignore empty lines or lines starting with '['
                if (line.StartsWith("[") || string.IsNullOrWhiteSpace(line))
                    continue;
                // split key/value pair
                string[] parts = line.Split('=');
                if (parts.Length == 2)
                {
                    string key = parts[0].Trim();
                    string value = parts[1].Trim();
                    // add pair to JSON
                    jsonObject[key] = JToken.FromObject(value);
                }
            }
            // optional: from JSON to Dictionary
            config = jsonObject.ToObject<Dictionary<string, object>>();
            return config;
        }
    }
}
```

While the code itself is very easy to follow, there are still a couple of points that would benefit from a brief explanation. First of all, the filepath string has the keyword readonly attached to it not for security purposes but only to strengthen the code by enforcing constraints on variable mutability. Following the same logic, ConfigParser is a protected method.
We defined ConfigParser as an async method and we shall use it for every part of the code that might be called from the UI. The reason for that is to guarantee that the UI stays responsive and to better utilize resources. While this is by no means a problem right now, the software employed by AMMILAB is a bit slow, and computer vision algorithms are especially resource-intensive. The most important feature of asynchronous programming is that it supports cancellation tokens and they should be used to end stuck tasks gracefully when incurring into a suspiciously long runtime. Also, if in the future part of the features of this software get delegated to API calls, asynchronous programming is very useful.

4.2 Why is it necessary to write a Logger

One of the best ways to ensure painless debugging is to create a log service to register anything that might go wrong in our code. One could argue that we do not need this, that the stack trace is more than enough to assist developers in debugging code, and while with the right IDE it is somewhat possible to do so, redundancy when debugging is very welcome. This is especially important since this project will probably be continued with the assistance of other people, and we may need to work on each other's code. A log goes a long way in helping set up the right workspace. It is also the industry standard to have such things in any decent medium-sized project. To create the Logger, we decided to use a Singleton class. In object oriented programming, a singleton class is a class that can have only one instance of the class at a time. After the first time, if we try to instantiate the Singleton classes, the new variable also points to the first instance created. So whatever modifications we do to any variable inside the class through any instance, affects the variable of the single instance created and is visible if we access that variable through any variable of that class type defined. After that, we only need to reference the singleton class instance in our Model layer classes.

```
//... environment ...


namespace PianoRollMIDIConverter.Models.Utils
{
public sealed class LoggerSingleton
{
    //this is the only instance of our Logger
    private static LoggerSingleton _instance = null;
    //a block to guarantee one single instance made available
    private static readonly object lockObject = new();
    //Serilog instance, it will access the Console and the Sink for us
    private ILogger _logger;

    //please note the private constructor
    private LoggerSingleton()
    {
        //desired configurations, rollingInterval is up to debate
        _logger = new LoggerConfiguration()
            .MinimumLevel.Debug()
```

```
            .WriteTo.Console()
            .WriteTo.File("log.txt", rollingInterval: RollingInterval.Day)
            .CreateLogger();
    }

    public static LoggerSingleton Instance
    {
        get
        {
            //it guarantess that just one thread creates the instance
            lock (lockObject)
            {
                _instance ??= new LoggerSingleton();
                return _instance;
            }
        }
    }
    //gives us the Serilog instance
    public ILogger Logger => _logger;
}
}
```

Serilog is an open-source .NET library for structured logging in applications and what it actually does here is to access Console for us. The code is again very easy to follow, now that we have a logging service it's very easy to call the logging instance and to dump message on a log file. If and when trouble arises, we can just first of all access the file and check what went wrong. This is especially useful when we do not actually know what went wrong in the first place. Now, whenever we do something in our code that might throw an exception or even better that can silently fail we just need to dump the result in our shared logger:

```
public class ConfigHandler
    {
        private readonly string filepath;
        private readonly ILogger _logger;

        public ConfigHandler(string filepath)
        {
            this.filepath = filepath;
```

```
            _logger = LoggerSingleton.Instance.Logger;
        }
    ...
        protected async Task<Dictionary<string, object>> ConfigParser
        (
        string filepath
        )
        {
            ...
            try
            {
                fileDump = await File.ReadAllTextAsync(this.filepath);
                _logger.Information
                (
                "File di configurazione caricato correttamente."
                );
            }
            ...
```

4.3 Contour detection using EmguCV

In order to convert into MIDI files the perforated piano roll scans, we need to be able to recognize two different patterns in our images: the dynamic line and the holes representing musical notes. As long as we focus on perforated piano roll scans taken with the optical scanner lights on, we can do both with OpenCV findContours algorithms.
Let us focus on the recognition of the dynamic line. While at this point the software does not recognize the holes yet, a very similar routine should prove sufficient to detect them in the future.
The strategy is as follows: first of all if the scan is full color, we convert it into greyscale. The reason is that findContours algorithms work better at pattern recognition if the image is greyscale. Then we employ a mix of blur filters to reduce noise and smooth the edges of prospect contours. Since we are interested in detecting prominent shapes, blurring helps in making our processing routine more robust to small variations in light or color. The last part of the preprocessing routine is contrast enhancement with a negative shift.

At this point, launching EmguCV shape detection algorithm will not yield

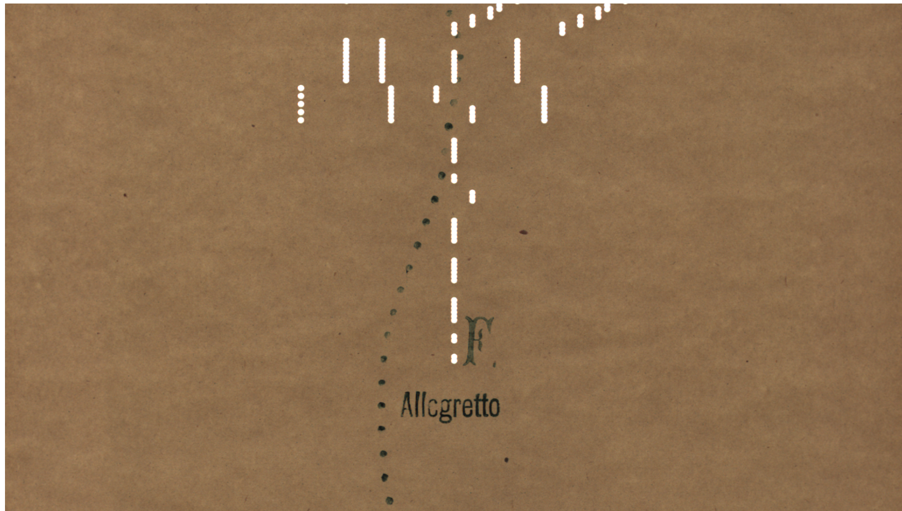

Figure 4.1: *A section of perforated piano roll before processing*

acceptable results. As we can see in the next image, there are spurious contours detected all around the scan. The paper is old and likely to have stains or tears so it's not surprising. What's more worrying is that findContours will detect comments, for instance about musical tempo such as "Allegretto" in this section. It's clear that we need to filter out spurious contours and we did just that in two passages. First of all, we can filter out shapes that are not akin to circles. Please note that in the case of holes, we would merely filter out shapes not akin to pseudo ellipsis so yet again the strategy stays the same. The second filter is by distance. Contours too far from other contours are clearly spurious. Now that we managed to filtered out all the spurious contours, a simple linear interpolation will return the processed dynamic line. Please refer to the next images to see the results of the various passages.

The relevant code is too long to discuss here in its entirety but there are still two passages that we'd like to highlight:

```
...
// Area and shape filter
double areaThreshold = 0.8;
double shapeThreshold = 0.9;
List<List<Point>> filteredContours = new List<List<Point>>();
foreach (var contour in contours)
{
```

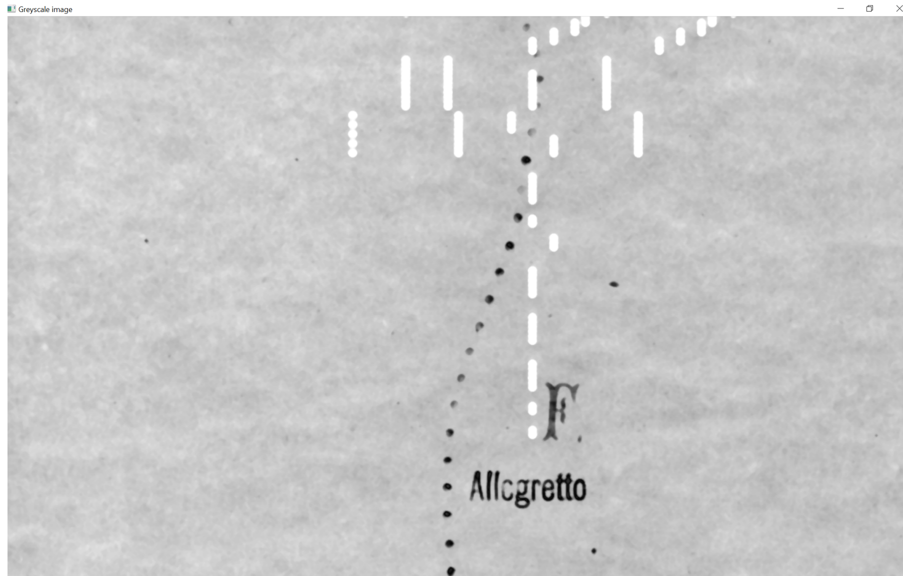

Figure 4.2: *Greyscale and blurred section of a perforeted piano roll*

```
double contourArea = CvInvoke.ContourArea
(
new VectorOfPoint(contour.ToArray())
);
double perimeter = CvInvoke.ArcLength
(
new VectorOfPoint(contour.ToArray()),
true
);
double expectedCircleArea = (perimeter * perimeter) / (4 * Math.PI);
VectorOfPoint contourVector = new VectorOfPoint(contour.ToArray());
RotatedRect boundingBox = CvInvoke.MinAreaRect(contourVector);
double width = boundingBox.Size.Width;
double height = boundingBox.Size.Height;
double aspectRatio = Math.Max(width / height, height / width);
if
(
(contourArea / expectedCircleArea) > areaThreshold
&& aspectRatio > shapeThreshold
)
```

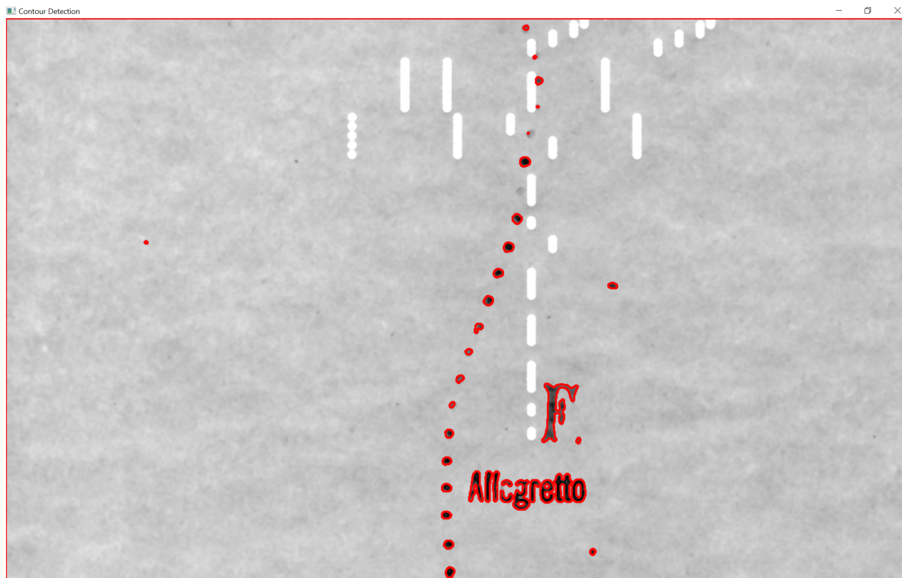

Figure 4.3: *The result of the findContour algorithm*

```
        {
            filteredContours.Add(contour);
        }
    }
    ...
```

In this part of the code, we try to detect which contours are good candidates for actually being part of the dynamic line. The parameters chosen to make such a distinction are completely heuristic. We found that a bigger areaThreshold had too big a chance of deleting non-spurious contours. In the next code snippet, we calculate the center of the centroids of the surviving candidates to further filter based on relative distance. It is of note that we decided not to filter based on Euclidean distance: a sudden shift on the x-axis is more likely to be a sign of a spurious contour while a jump on the y-axis might just be caused by a blotch on the paper or a hole covering a single point of the dynamic line. In other words, the distance doesn't have the same weight on both axes and so needs to be processed separately.

```
List<PointF> centroids = new List<PointF>();
foreach (var contour in filteredContours)
{
```

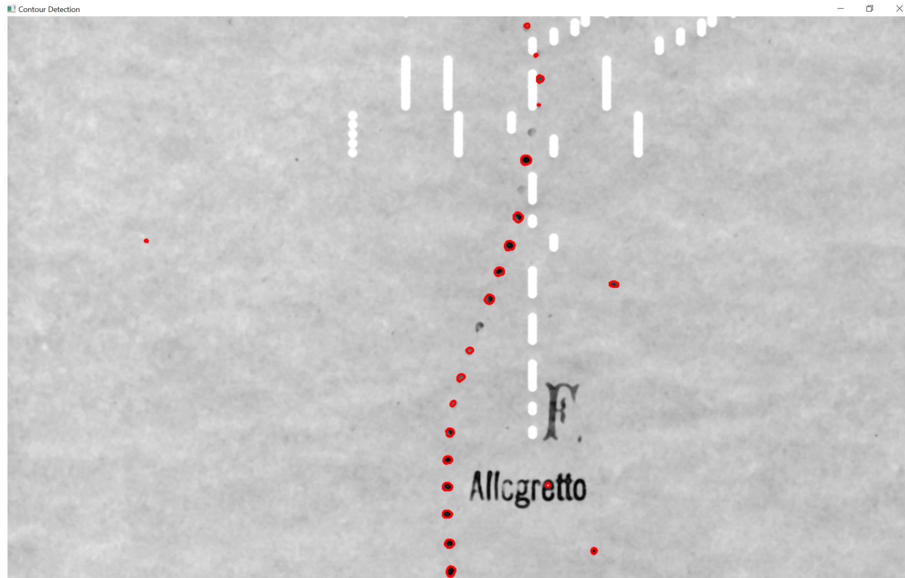

Figure 4.4: *findContour with a filter based on shape*

```
float centerX = 0;
float centerY = 0;
foreach (var point in contour)
{
    centerX += point.X;
    centerY += point.Y;
}
if (contour.Count > 0)
{
    centerX /= contour.Count;
    centerY /= contour.Count;
}
centroids.Add(new PointF(centerX, centerY));
}
```

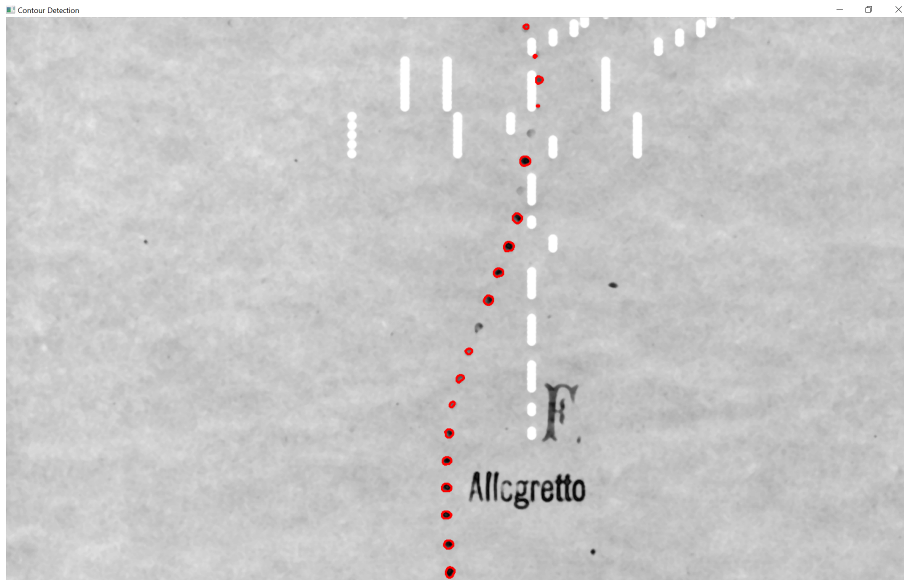

Figure 4.5: *findContour with a filter based on shape and distance*

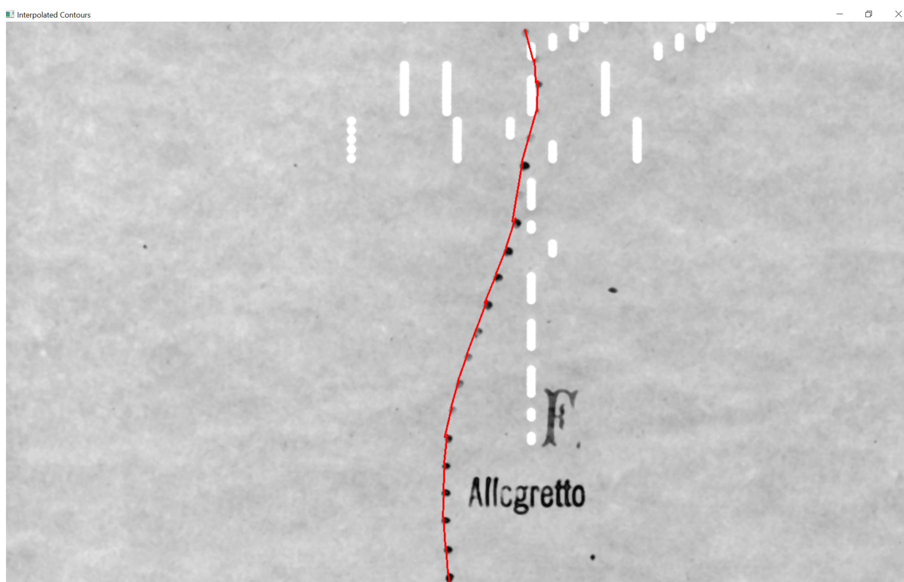

Figure 4.6: *The filtered contours interpolated to follow the dynamic line*

Conclusions

Perforated piano rolls are a curious oddity that, in spite of their historical significance, has been set aside solely due to the inherent difficulties in finding a refurbished player piano and the degrading conditions of the rolls themselves. In recent years, great efforts have been made to at least digitize the available paper rolls, in constant danger of tears. The scans can then be employed to print new rolls or to convert them into MIDI files with the aid of computer vision software.
In these pages we outlined the tools used to start the development of one such software, similar in scope to the one realized by AMMILAB but easier to upgrade and maintain.
We set up EmguCV, AvaloniaUI, and Reactive UI and managed to realize the preprocessing routine and the computer vision tools to correctly detect the dynamic line of perforated piano rolls. While at this point the software is not yet able to detect holes, a tweaked similar strategy should theoretically be able to detect the holes' contours.
The configuration files employed by AMMILAB are also correctly recognized and processed by the new software thus assuring painless switching to the new software as soon as it gets released.
At this point, the main limit in our approach is the assumption about the optical scans being realized with lights on since the contour detection routine needs a significant difference in color between holes and dynamic line.
The next step in the development involves tackling the holes recognition and the actual conversion into MIDI files while setting up the UI for user intervention.

Appendixes

Appendix A

A brief overview of the MIDI protocol

A.1 A bridge between music and computers

Visually, there is not much difference between a modern piano roll MIDI data visualization and the scan of the perforated piano roll itself, both represent musical data that can be interpreted by a musical instrument, be it a player piano or a personal computer. Clearly, we are being purposefully lax with the term "musical instrument" but the parallelism should be evident. What was lacking during the emergence of computer technology was a bridge between music and computers, a standardized means of communication between electronic musical instruments, computers, and related equipment.

Several attempts were made between the 50s and the early 1980s, both regarding electronically generated music and communication between electronic devices meant to produce music. In 1957, the Columbia-Princeton Electronic Music Center produced the first electronically programmable synthesizer, Herbert Belar's so-called Mark II[3]. At the same time, Max Mathews, an engineer working at Bell Laboratories, created the first computer program that was able to synthesize a sequence of tones into a short song lasting only 17 seconds. While initially the software only employed one single waveform and had no control over the dynamics of the generated sound it was later refined into GROOVE (Generated Realtime Output Operations on Voltage-Controlled Equipment) which was able to store musical information played by a musician on an external synthesizer[10] Later on, with the growing availability of home computers, enterprising engineers managed to put into musicians' hands the research efforts to widespread

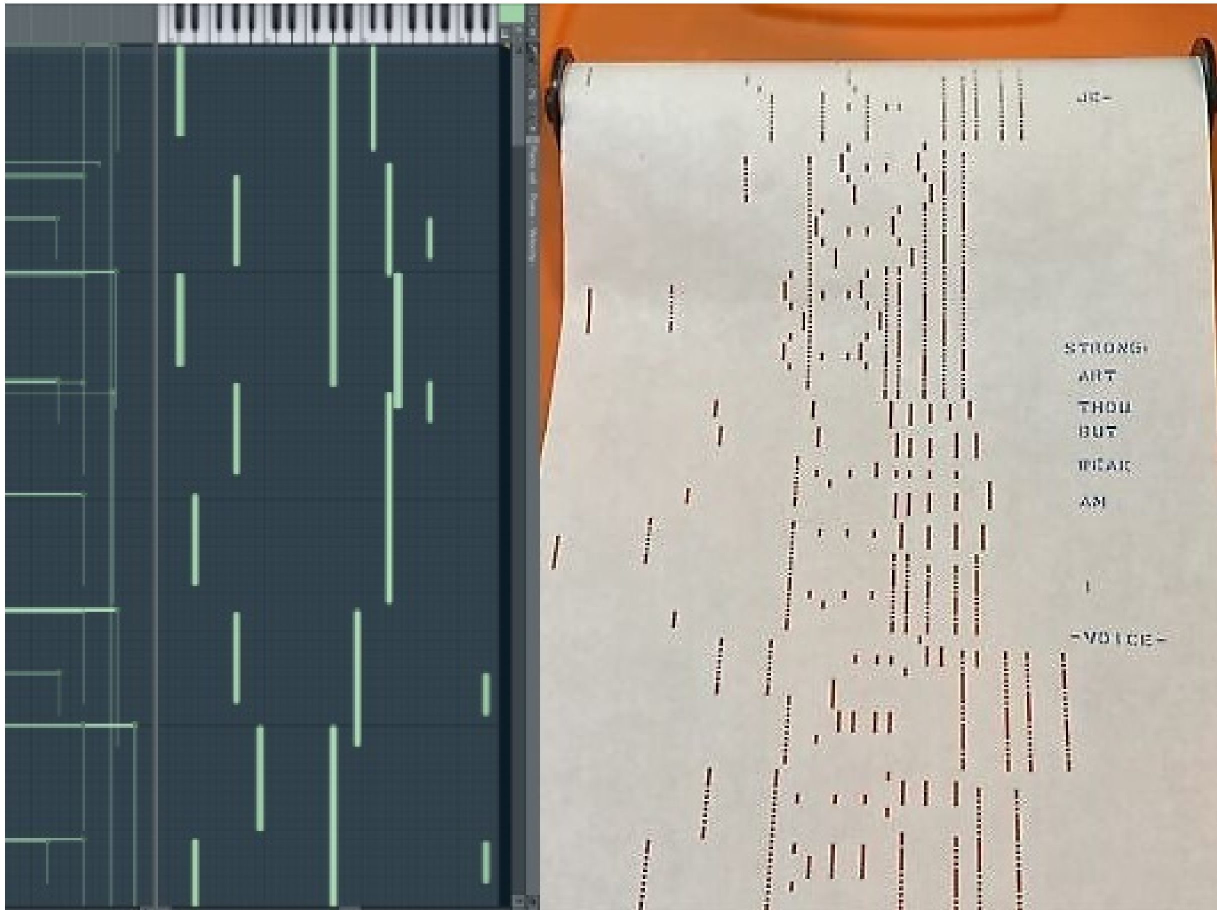

Figure A.1: *A perforated piano roll and a DAW digital workstation*

popularity. A particularly interesting product from this point of view is the Minimoog, a monophonic analog synthesizer introduced to the market in 1970 by Moog Music, headed by Robert Moog[15].
Composers and musicians could then begin to explore the timbral possibilities of synthesizers. A milestone in this regard is the album "Switched-On Bach," created in the studio by the American composer Wendy Carlos and released in 1968. It contains some of Johann Sebastian Bach's most famous pieces, including the entire Brandenburg Concerto No. 3, performed using a Moog modular system.
It is at this point in time, with the growing availability of different synthesizers and software that the need for a unified protocol starts to arise. Market leaders start to worry that the lack of compatibility between brands and models might

cripple the synthesizer technology altogether. Sure, both Yamaha and Roland made great efforts to make their own products compatible with one another but it wasn't enough. So, the equivalent of the Buffalo Convention for perforated piano rolls was needed, one that hopefully could do more than define, for instance, the data for pitch and note duration.
Some progress was made during the AES Convention held in 1981. On this occasion, Dave Smith and Chet Wood defined a standardized industrial-level interface to provide synthesizers, sequencers, and personal computers with. This communication protocol involves the use of standard 1/4-inch jacks and a transmission speed of 19.2 kbps.
The project, further developed in cooperation between different companies, made its debut at the NAMM trade show in Los Angeles in 1983. During the event, an analog synthesizer, the Prophet-600 by Sequential Circuits, demonstrated its ability to communicate with a Roland Jupiter 6 keyboard through a 5-pin cable. The initial specifications of the protocol were published under the title "MIDI 1.0 Specification"[2] in August 1983 by the International MIDI Association (IMA), also known as MIDI Users Group (MUG) or International MIDI User Group (IMUG) to widespread acclaim.
In the next decades, the MIDI protocol gets recognized as the standard for managing music in digital form, receives changes and improvements, and more documentation is made available. In 2020, MMA presented the MIDI 2.0 specification project, a leap toward bidirectional communication between devices without losing retrocompatibility with MIDI 1.0.

A.2 The fundamentals of MIDI protocol

The MIDI protocol defines how control messages between devices interact with one another and it assures both compatibility between brands as long as they conform to the protocol and expandability, meaning that messages sent by devices using older versions of the protocol should be correctly interpreted by newer models.
Not every MIDI device can answer every MIDI possible MIDI message, and it wouldn't make sense for the protocol to require so. For instance, controllers and digital synthesizers don't need to answer a TUNE REQUEST message, usually reserved for recalibrating the oscillators of analogical syntethizers. The MIDI specifications contain messages that encode functions or commands to which a compatible device may or may not respond. This behavior is perfectly acceptable,

Figure A.2: *A Minimoog, in its 2016 rerelease*

provided that the device is designed to ignore commands it doesn't understand and doesn't attempt to interpret them. The protocol also reserves undefined bit messages, foreseeing future releases. When defined, older releases would merely discard such messages[1].
Commands are typically sent in the form of messages, which are usually quite short and consist of a variable number of bytes, typically ranging from 1 to 3 bytes. The language's structure is simple, and the variety of messages needed to encode a performance and control MIDI devices is relatively small.
One of the reasons for which the MIDI protocol was readily accepted and in time became the industry standard is that adopting it was quite cheap. Adding the MIDI protocol to a digital synthesizer only cost 5$ per device, a small price to pay to assure compatibility between brands and future releases. Quite clearly, some compromises had to be made to contain cost, for example, the use of 5-pin DIN connectors and the limited data transmission speed allowed for the transfer of MIDI data[18]
According to the original specifications, the transmission of MIDI messages from sender to receiver so that it may parse them happens at a capped 31.25 Kbit/s speed which at the time seemed reasonable. When the original MIDI specification's data rate of 31.25 Kbit/s became insufficient for certain applications, var-

ious solutions were developed to accommodate higher data throughput. One common approach is the use of multiple MIDI ports or channels to transmit parallel streams of MIDI data. Some newish MIDI protocols, such as MIDI over USB or Ethernet (e.g., RTP-MIDI), can achieve higher data rates and offer more flexibility than the original 5-pin DIN MIDI protocol.
Since MIDI was designed to transmit data about musical performance, it must provide sufficiently accurate timing to preserve its rhythmic integrity: one serious limitation is that it is not possible to send or receive multiple MIDI messages simultaneously, for example, messages generated by the simultaneous pressing of multiple keys forming a chord would be sent over the transmission channel one after the other. Many MIDI devices include a "MIDI Thru" port, which allows you to daisy-chain multiple MIDI devices together. This way, the MIDI data received by one device can be passed through to others, allowing them to respond to different aspects of the data, including simultaneous note events. Anyway, most modern MIDI instruments, especially synthesizers and digital keyboards, are designed to be polyphonic. This means they can play multiple notes simultaneously. Each note being played generates its own MIDI note-on and note-off messages, which are sent over the MIDI channel. This allows for the representation of chords and complex musical passages.

Figure A.3: *A Minimoog, in its 2016 rerelease*

A MIDI port is made up of three components for the exchange of data that enable communication between MIDI-compatible devices, facilitating the creation and control of music. MIDI IN is where a device receives incoming MIDI data. It serves as the entry point for MIDI messages, allowing external controllers like keyboards to send musical commands, such as NOTE ON/OFF, modulation, and more, to the receiving device. Unsurprisingly, MIDI OUT is responsible for send-

ing out musical information generated by the device, making it possible to control and synchronize other MIDI instruments or route the generated sound to external sound modules or computers for further processing. MIDI THRU acts as a parallel output to MIDI IN, replicating the incoming MIDI data without altering it. This feature allows for daisy-chaining multiple MIDI devices, ensuring that the same MIDI data can be sent to several devices in the chain, simplifying complex MIDI networks and setups. There are also multi-port MIDI devices, for instance, the Yamaha YME-8 expander, equipped with 2 MIDI IN and 8 MIDI OUT/THRU.

A.3 MIDI messages

MIDI is a music description language in binary form. It was designed for use with keyboard-based musical instruments, so the message structure is oriented to performance events, such as picking a note and then striking it, or setting typical parameters available on electronic keyboards. Every message is made up of one or more bytes, with two extra bits for flagging the start and end of the message. The first byte is called status byte and it serves the purpose of identifying the kind of sent message to the receiver. The rest of the message is made of one or more data bytes, and they function as the payload of the communication.
MIDI files also have a header chunk at the beginning that contains information about the MIDI file itself, such as the file format, the number of tracks, and the division (timing) format. This header chunk is not part of the individual MIDI messages contained within the file.
Meta events are how the MIDI files convey this layer of information that MIDI devices need to be able to understand in order to reconstruct the piece of music correctly like tempo changes or time signature changes.

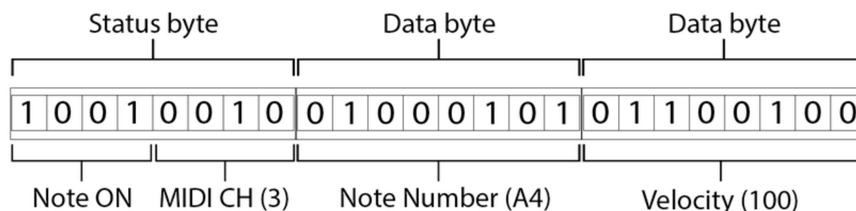

Figure A.4: *Format of a NOTE ON message*

The most common MIDI messages are quite clearly the NOTE ON and NOTE OFF messages. In a NOTE-ON message, the status byte appears in the form

1001nnnn, with the first bit set to 1 representing the byte as a status byte and the three bits 001 identifying a NOTE ON message. After this status byte, the payload is made of two bytes, respectively for pitch and velocity of the note. For both data bytes, the most significant bit is set to 0, therefore, the pitch and velocity values occupy the remaining 7 bits. This allows for a total of 128 values or levels in the range [0, 127]. The NOTE OFF message has the same format, albeit with code 000. Interstingly enough, there are two ways to switch off a note. The first one is with a NOTE OFF message targeting a note with a set pitch and value. The second is with a NOTE ON message targeting a note with set pitch and value 0. This second alternative does not allow specifying a release velocity, though.

Bibliography

[1] L. L. A., *Midi una guida al protocollo, alle estensioni e alla programmazione*, Milano University Press, 2021, pp. 9–24. $https://doi.org/10.13130/milanoup.27$.

[2] I. M. ASSOCIATION, *Midi 1.0 detailed specification*. $https://www.midi.org/specifications/midi1-specifications/midi-1-0-core-specifications/midi-1-0-detailed-specification-2$ - Accessed: 9/16/23.

[3] BELAR, HERBERT, *RCA Mark I and Mark II Synthesizers*. URL: $https://ethw.org/RCA_Mark_I_and_Mark_II_Synthesizers$ - Accessed: 9/16/23.

[4] J.-P. BOODHOO, *Design patterns - model view presenter*, 2006.

[5] M. CAULFIELD, *1908 buffalo convention piano roll specifications*, Mechanical Music Digest, (2008). URL: $https://www.mmdigest.com/Gallery/Tech/1908Buffalo.html$ - Accessed: 9/15/23.

[6] EMGU CV, *EmguCV GitHub*. $https://github.com/emgucv/emgucv$ - Accessed: 9/15/23.

[7] C. B. FOWLER, *The museum of music: A history of mechanical instruments*, Music Educators Journal, 54 (1967). DOI: $doi/10.2307/3391092$.

[8] S. HAROLD, *The Great Pianists*, New York: Simon and Schuster, 1963.

[9] M. KROMER, *Basics of ci/cd and pipeline scheduling*, in Mapping Data Flows in Azure Data Factory, Apress, 2022, pp. 139–154. DOI: $https://doi.org/10.1007/978-1-4842-8612-8_9$.

[10] M. V. MATHEWS, *Groove—a program to compose, store, and edit functions of time*, Communications of the ACM, 13 (1970), pp. 715–721.

[11] M. NIXON, *Feature Extraction and Image Processing for Computer Vision*, Academic Press, 2019. DOI: $https://doi.org/10.1016/C2017-0-02153-5$.

[12] OPENCV, *OpenCV GitHub.* $https://github.com/opencv/opencv$ - Accessed: 9/15/23.

[13] P. PHILLIPS, *Piano Rolls and Contemporary Player Pianos: The Catalogues, Technologies, Archiving and Accessibility*, 2017. URL: $https://ses.library.usyd.edu.au/handle/2123/16939$.

[14] ——, *History of piano roll digitization*, Australian Collectors of Mechanical Musical Instruments, (2022).

[15] T. PINCH, *In the moog*, in Journées d'Informatique Musicale, 2011.

[16] REACTIVEUI, *ReactveUI documentation*. URL: $https://www.reactiveui.net/docs/$ - Accessed: 9/15/23.

[17] A. ROBINSON, *An introduction to piano roll scanning*, (2023).

[18] J. ROTHSTEIN, *MIDI: A comprehensive introduction*, vol. 7, AR Editions, Inc., 1992.

[19] Z. SHI, K. ARUL, AND J. O. SMITH III, *Modeling and digitizing reproducing piano rolls.*, in ISMIR, 2017, pp. 197–203.

[20] J. SMITH, *Patterns - wpf apps with the model-view-viewmodel design pattern*, 2009. URL: $https://learn.microsoft.com/en-us/archive/msdn-magazine/2009/february/patterns-wpf-apps-with-the-model-view-viewmodel-design-pattern$ - Accessed: 9/15/23.

[21] STALTZ, ANDRÉ, *The introduction to Reactive Programming you've been missing*. URL: $https://gist.github.com/staltz/868e7e9bc2a7b8c1f754$ - Accessed: 9/15/23.

[22] STANFORD UNIVERSITY PIANO ROLL ARCHIVE (SUPRA), *Introducing Stanford's Digital Piano Roll Archive.* $https://exhibits.stanford.edu/supra$ - Accessed: 9/15/23.

[23] M. STONIS, *Enterprise Application Patterns Using .NET MAUI*, Microsoft Developer Division, .NET, and Visual Studio product teams, 2022.