Università
degli Studi
di Padova

Dipartimento
di Ingegneria
dell'Informazione

DEPARTMENT OF INFORMATION ENGINEERING

MASTER DEGREE IN
ICT FOR INTERNET AND MULTIMEDIA

TECHNOLOGICAL MIGRATION FOR INTERACTIVE MULTIMEDIA ARTWORK CASE STUDY: REACTIVATION OF "IL CAOS DELLE SFERE" BY CARLO DE PIRRO

Supervisor: Prof. Sergio Canazza

Co-supervisor: Dott. Alessandro Fiordelmondo

Candidate: Luca Zecchinato

ACADEMIC YEAR 2021-2022
DATE 28/11/2022

Abstract

Preservation of Interactive Multimedia Artworks has become a particularly active research area over the last years. Starting from the end of the last century, the role played by multimediality within the performing arts has become more and more important, thanks to the adoption of increasingly more sophisticated technologies. Such analogical and digital artworks rely on underlying working computer systems; however, since technology advances with a relevant pace, hardware and software involved in the installations eventually become obsolete. There is therefore the need for preserving the experience and more importantly the interactions and the artistic thinking via digital tools and preservation methods. While dealing with the preservation of standard document formats is well established, very few detailed approaches about preserving multimedia objects and pieces of electronic art have been defined. In this context, this work presents a multilevel preservation approach developed in the *Centro di Sonologia Computazionale* (CSC) group of the Department of Information Engineering at the University of Padova. The application of such system is considered for the reactivation of the "Il Caos delle Sfere", a 1999 installation by Carlo De Pirro. Furthermore, since an accurate documentation of the original artwork's setup was missing, it has been produced during the reactivation. On top of the methodological multilevel analysis, a corresponding technological migration for the "Il Caos delle Sfere" has been completed at the CSC laboratories: the experimental results of the operations carried out and all the hardware and software migrations are also reported. From the point of view of the preservation, the overall migration process has aimed at preserving more the identity behind the artwork rather than a simpler replication of the original performance, also due to the lack of documentation for it. This is also why, in this work, several future additional improvements required in the proposed technological reactivation are finally analysed.

Contents

List of Figures

Introduction

With the development of new technologies over the last century, society went through a process of radical change. Everyday life started to be more and more altered in its fundamental activities. The way people lived within it, the way their behaviour was impacted by the advent of new tools and media affected also the artistic production. It gave raise to the possibility of having an interaction between the public and an artwork or an installation, opening up to a greater involvement in the artistic thinking. In this context, *Interactive Multimedia Artworks* became a crucial factor: they are artworks composed by a heterogeneous set of media and in which human interaction is a key point. The role played by multimediality within the performing arts increased over the years also thanks to a fast technological development. Due to the relevant pace, progress caused the hardware and software involved in the installations to eventually become obsolete. In this framework, preservation and reactivation of interactive multimedia artworks are fundamental practices to take care of artworks but above all to allow the transmission of past and nowadays forms of art to future societies. The topic of preservation and reactivation of interactive multimedia artworks has gained therefore a particularly active role in research. However, it is still far from a standardization and shared methodology to deal with. This work outlines a model for preservation of interactive multimedia artworks developed recently at the *Centro di Sonologia Computazionale* (CSC). It's a *multilevel* preservation model, i.e. it establishes a way to organize the various components of artworks (both hardware and software) adopting a structured pattern. The model essence is to aim at identifying the authenticity of an artwork, reinterpreting it and by consequence updating the technological settings. The model considers interactive multimedia artworks through their process of transformation, which is what makes them different from traditional artistic manifestation. It also provides a scheme to be used to overcome the problem of fast obsolescence and negligence affecting the less recent interactive artworks in the preservation field. In particular, it sets out the so called *Digital Preservation Objects* (DPOs), the fundamental digital units that represent the evolution of the artwork and enable a complete digitalization for it. The multilevel preservation model is the result of the CSC experience in the context of preservation of multimedia artworks. The model proposed was modified over the years: the version reported was recently applied for *The time consumes* reactivation, a *videoloop* artwork by Michele Sambin. This work proposes the application also for the reactivation of *Il Caos delle Sfere: Become a Pianist with 500 Italian Lire* artwork (1999). The installation was one of the various artistic expressions of the composer Carlo De Pirro in collaboration with technical partners Nicola Orio and Paolo Cogo, at that time members of CSC. The concept of the artwork is to use an electronic pinball, a common gaming machine, to control an automatic performance played over a Disklavier. Up to the 2004 (and with some later exhibition in 2012 and 2014), the artwork was constantly presented in different festivals and scientific disseminations. However, its hardware and software components went through a very rapid process obsolescence over the years. Therefore, a reactivation for that had to be executed in order to preserve one of the most important artistic production developed in collaboration with the CSC group by the composer Carlo De Pirro, which unfortunately died in 2008. The occasion of a technological migration occurred for the *Science4All* dissemination of 2022 in Padua. The reactivation, technological migration and the DPOs organization was carried out by Alessandro Fiordelmondo, Mattia Pizzato and the author of this work at the CSC laboratory. Starting from the multilevel preservation model, the research team developed the hardware, software and documentation that made it possible to present again to the public the artwork. The path followed was to preserve more the identity rather than the technological assets.

The thesis has the objective to describe in detail not only the *Il Caos delle Sfere* artwork as a whole but also to provide a documentation of its main features that was never developed before. In this framework, apart from the technological migration, the way the preservation model was applied to the artwork represents also a possible improvement in the organization of DPOs for future case studies. The approach consists on digitalizing records composed by *items* which are documented by using a modified version of the *Dublin Core* scheme, a metadata *schema* designed to enable descriptions of any resource. The research of the best way to create such DPOs is still in progress, modifications will be introduced to make the metadata scheme more suitable for the preservation and reactivation of interactive multimedia artworks. The application of the metadata scheme as well as the implementation of the technological migration were carried out by the author of this work during the internship at the CSC laboratory. The main focus in the reactivation was more on the engineering part, on testing how the connection with a Parallel Port or a MIDI port could be handled with modern microcontroller and constructing new hardware. The work aimed at reactivating the artwork rather than making it efficient. At the end of the activities, however, the result was satisfying. *Il Caos delle Sfere* collected a large attention on the public during the *Science4All* dissemination, making this artistic production of Carlo De Pirro living again after years.

The exposition is organized as follows. *Part I* introduces the key aspect of the multilevel preservation approach developed at the CSC with a deep insight on the definition of interactive multimedia artworks and how preservation and reactivation should be performed for them. *Part II* instead is about the application of the model to the *Il Caos delle Sfere* and it is composed of three chapter. *Chapter 2* introduces all the features of the original installation: the hardware involved, the pinball, the Disklavier and the acquisition board are treated in detail. A focus is also provided on the interfaces connecting the hardware, the Parallel Port and the MIDI port. *Chapter 3* concerns the algorithm and the source code that were developed to make the installation work. A brief discussion regarding how the code was written together with the description of the algorithm route are also examined. *Chapter 4* describes the majority of the results that were obtained during the reactivation process. Firstly, the DPOs development and the digitalization approach are introduced in relation to the evolution of *Il Caos delle Sfere* over the exhibitions. Secondly, a discussion about what technological migration was followed for the reactivation is unfolded. Finally, the modified and updated code developed for *Science4All* dissemination together with a look for future improvements is considered.

Part I

Preserving Interactive Multimedia Artworks

Chapter 1

The multilevel preservation approach of the CSC research group

The topic of preservation and reactivation of interactive multimedia artworks, despite being well-known by many institutional entities and characterized by many projects around the world, is still far from a standardization and shared methodology to deal with. In this context, this chapter presents the *Centro di Sonologia Computazionale* (CSC) multilevel preservation model, which defines a way to organize the heterogeneous components of artwork (its constituent elements, its description and its appearance) into a set of so called *Digital Preservation Objects* (DPOs). The rationale of this approach is to define artworks through a process of transformation rather than the identification of a unique and fixed manifestation of authenticity. In addition to that, the digital migration of technologies and the problems and advantages of this approach are presented. Although it cannot represent a solution suitable for any art form, digital migration can be an effective option to reactivate and preserve artworks. It leads to the identification of the authenticity of an artwork and therefore allows to reinterpret it and update technological settings, overcoming the problem of fast obsolescence increasing with years. In the technological migration the collaboration with the original authors (artist, technicians, performers) who are responsible for the identity of the artwork is also fundamental. In addition to that, digital migration opens up the possibility for other important perspectives. It can simplify the remote transmission of digitized artworks, which can be supplemented with virtualization for more complex works.

1.1 Interactive multimedia artworks and preservation

Starting from the second half of the 20th century, forms of art were characterized by a radical transformation process as a result of the western social changes after the war. Artists started to focus on the material prospects of social and political events making the public and participation in artistic production more important [24]. The interaction acquired a central role in the artistic works, both in the performance (performer interaction) and in the installation (audience interaction). Human-art interaction was no longer considered as a specific ability of a skilled performer. In addition to this, early forms of technology started to enter everyday life radically altering society and, by consequence, gaining also a central role in artistic productions. Such evolution of the art production gave a very deep focus on the society, revealing its identity, how people live within it and how they think and interact with technologies and media. Such aspect become a significant component of the cultural heritage available nowadays. The interaction affecting the artistic production became a crucial factor in the so called *Interactive Multimedia Artworks*, which can be defined as artworks composed by a heterogeneous set of media and in which human interaction is central. Both interactive installations and performances fall under this definition. In this context, preservation and reactivation of interactive multimedia artworks are fundamental practices. They enable the study and the transmission of past and nowadays forms of art through contemporary and future societies. However, interactive multimedia artworks are characterized by preservation and restoration practices different from the traditional ones. The motivation is the rooted interconnection with technology, very relevant in terms of the fast obsolescence of hardware and software, which may soon become an irreversible loss. These artworks are completely different from analogue fixed ones, such as paintings, sculptures or architecture, which remain rather durably over years [5]. In fact, features including variability, reproduction, performance, interaction are incorporated in many works. Media art is not a static, unique and solid object, but often a set of components, hardware, and software which create a process- and time- based experience [26]. In addition to that, multimedia artworks are often the result of the collaboration and participation of multiple artists, technicians, curators, performers, and audiences (as in the case of interactive installations) and with a strong relationship with the original surrounding environment.

On top of the cultural field, even technological development and the economy take advantage of preservation and reactivation strategies. In the creation of multimedia artworks, artists keep driving technological development with their ideas and request new tools to make the current solution more advanced. The art industry represent a valuable role in technological development and therefore it is important to have it supported. Creativity is in fact gaining more and more significance in industrial contexts. Especially today, with the forecast of a new industrial form, this quality acquires a central role in the productivity. In particular with Industry 5.0[1], the human workers will come back to be the factory floor and, together with machines and smart systems, participate in productivity with their creativity and brainpower [18]. In this framework, interactive multimedia art is an ideal case of human-machine coexistence in which humans dominate machines through their own creativity. Apart from the technological aspect, the preservation of these new forms of art can be seen as a concrete practice to build the future. Besides being material knowledge, available for all societies, the preserved art provides an exemplary model for a society increasingly involved in a technological landscape. This is especially important for interactive multimedia installations, which offers original systems of human-based creativity for the technological development. The loss of contemporary and recent-past artworks would translate into slowing down the maturation of the central and creative role of the human in the future society. Therefore preservation is necessary as new artworks have a generally short life expectancy and there is an high risk of losing important expressions of society. Finally, artistic reactivations can bring an economic benefit. Reactivating artworks would result in more exhibitions, more audience, more cultural dissemination with direct and indirect economic income and positive impact on the art.

Nowadays, there are many preservation strategies and examples of reactivated multimedia artworks. The same characterizes archives which are growing by collecting contemporary artworks through original and different approaches. From the 2000s onward, the archival community, as well as universities, museums, and artists and cultural ministries have become more aware of the problem of multimedia art preservation. Institutional entities are aware of the benefits derived by the circulation and enhancement of reactivated artistic works, therefore they are increasingly supporting innovative preservation projects. In Europe, for example, there are many projects involved in the preservation of new multimedia artworks. An Italian example is the *Protocollo per l'Autenticità, la Cura e la Tutela dell'Arte contemporanea* (PACTA - protocol for authenticity, care and protection of contemporary art) of the cultural ministry

[1]Industry 5.0 is a vision of industry that aims beyond efficiency and productivity as the unique goals, and reinforces the role and the contribution of industry to society `www.research-and-innovation.ec.europa.eu/research-area/industrial-research-and-innovation/industry-50_en` (accessed 29 October 2022)

of Italy (*Ministero per i Beni e le Attività Culturali e per il Turismo* - MiBACT), developed in 2017. It is a document assessing a set of guidelines and principles for the protection and valorization of artist archives. It is particularly focused on installation and multimedia artworks. Despite many experiences in this field, it is still difficult to establish shared and standardized practices of preservation and reactivation of such art forms. That is because traditional preservation strategies are not suitable for contemporary art, so there is the need to develop new preservation paradigms. The issue is that the study of these practices is still too related to traditional ones and therefore it has yet to develop properly. In this framework, a contradiction often emerges: on the one hand multimedia artworks are considered as time- and process-based objects, on the other, new preservation practices still aim to capture the artwork as a fixed object with a unique and unaltered authenticity.

A specific preservation and reactivation strategy has been developed by the CSC at the University of Padua, it is called the *Multilevel Preservation* model. It has undergone further development from the original definition defined in [7] and in [8], based on the results obtained from the case studies in which it has been applied. This model aims to record different reproduction phases of the artwork and to define it as a process rather than a fixed object. The layers that make up the model are such that it is possible to move from the detailed characteristics of a single exhibition (performance or installation) to the relationships of all exhibitions of the artwork. With such a stratification it is possible to record the dynamic authenticity of the artwork.

1.2 The challenge of preserving and reactivating

As described in the previous section, preservation is fundamental to slow down the process of degradation and obsolescence and then to ensure the permanent availability of the artistic heritage [13]. Preservation can be considered as the sum total of the steps necessary to ensure the permanent accessibility, forever, of documentary heritage [8]. To transmit the artwork to future societies as faithfully as possible, the act of preservation must consider the concept of authenticity. This feature can reveal the fundamental properties which are necessary to restore the identity of an artwork. Generally authenticity is closely linked to the concept of physical integrity of the artwork. However, art preservation cannot be granted by a simple maintaining of physical integrity due to the increasing use of ephemeral and heterogeneous material. Interactive multimedia artworks are made up of multiple instances, such as events, assemblages and experiences, which must be remixed with new or partially new material, equipment and human interaction [10]. But since each artwork has completely different and unique properties, it is difficult to standardize a definitive methodology. The solution is to determine a research approach that must be applied individually for a single artwork and an administrative model with which all the parts that determine the work should be organized. In this aim authors, technicians and performers should be involved in the preservation and reactivation processes, through technical collaborations, interviews and case studies. With them, the fundamental properties of an artwork can be established as well as the authenticity principles of it. Another fundamental aspect to consider is the evolution of the work over time, from exhibition to exhibition to evaluate authenticity. Art preservation strategies should not only consider the initial exhibition but rather all the exhibitions because they represent the development of the work over time. However, since interactive multimedia artworks have a dynamic nature, the act of reinterpretation should be also considered. In fact, every reactivation of an artwork should be considered as a new reinterpretation (whether or not it is reactivated by the authors). It's important to notice that in preservation of artworks, the replacement of abstract or concrete components is to be taken into account. In turn, the replacement of such parts always introduces a reinterpretation of the work's essential system. Reinterpretation may seem like a dangerous act at first glance but it is a powerful operation to assess and to spot the fundamental properties of authenticity and then identity. Finally, to ensure the integrity of every component involved in the artwork (rather than a simple physical integrity), a documentation of all the preservation and reactivation processes must be compiled [23].

1.3 The multilevel preservation model

The model described in this section was developed as a part of the preservation operations of a selection of interactive multimedia installations by the composer Carlo De Pirro. Subsequently, it was applied for *Medea*, an opera-video by Adriano Guarnieri, which was then reactivated as a multimedia installation by Alvise Vidolin and recently for *The time consumes*, a *videoloop* artwork by Michele Sambin. The model belongs to the solid experience of the CSC in the computing preservation field.

The multilevel preservation model aims to preserve multimedia interactive artworks with a focus on their various exhibitions and thus as a process or a dynamic object rather than a fixed one. This model was developed as an expansion of the methodology for the preservation of audio documents defined in [6]. Behind the preservation method there is a main concept which is the *preservation copy*. For any audiovisual document, the preservation copy can be described as the artefact designed to be stored and maintained as the preservation master [17]. It consists of an organized dataset collecting all the information represented by the original document, accompanied by the metadata, i.e. a set of data that describes and gives information about other data, and by the documentation about the preservation process. In the proposed multilevel preservation model establishes this results into the concept of *Digital Preservation Object* (DPO). The artwork's DPO is a digital file that encapsulates a set of digital and analogue inter-related and inter-connected items, coordinated according to a logical architecture with the aim to represent a single exhibition of an artwork (the first exhibition or any reactivation characterizing the installation). Therefore, the goal of the multilevel model is to group and connect all the DPOs of a single artwork to represent it as a process rather than a single fixed work.

The overall architecture of the model is based on the *General Instruction Standard for Archival Description* ISAD(G) [14], an international framework to register archival documents produced by corporations, persons and families and providing guidelines for creating descriptions of archival materials. The proposed multilevel approach model defines three levels arranged in a hierarchical order to obtain a representation of the artwork from the general to the specific. The highest level of the model represents the artwork and internally it groups all the exhibitions or reactivations the artwork went through. Given the ISAD(G) scheme, the artwork can be represented via the *series*, i.e. documents arranged in accordance with a filing system or maintained as a unit because they result from the same accumulation, filing process or the same activity [14]. Each exhibition is a presentation of the artwork's identity, which in turn constitutes the unity of the series. The intermediate level instead represents the single exhibition represented as a DPO. The DPO is a container in which all the items of an artwork's exhibition can be collected. Finally, the lowest level represents the single items of an individual artwork's exhibition. The items are all the analogue and digital elements that compose the artwork, its distinct features and any other kind of documentation that describes the experience of an exhibition. This level is defined as the *item* level, which in the ISAD(G) terminology is the "smallest intellectually indivisible archival unit" [14]. However, in the proposed model, three different kinds of items with distinct functions for the artwork and thus different roles are considered. They are classified as bit, data and experience:

- **Bit**: it consists on all those parts of an artwork that can be directly preserved, both analogue and digital items (for example hardware and software, performative objects, fixed-media files like video or audio used in the exhibition, etc.). It concerns the data that are kept in the original for-

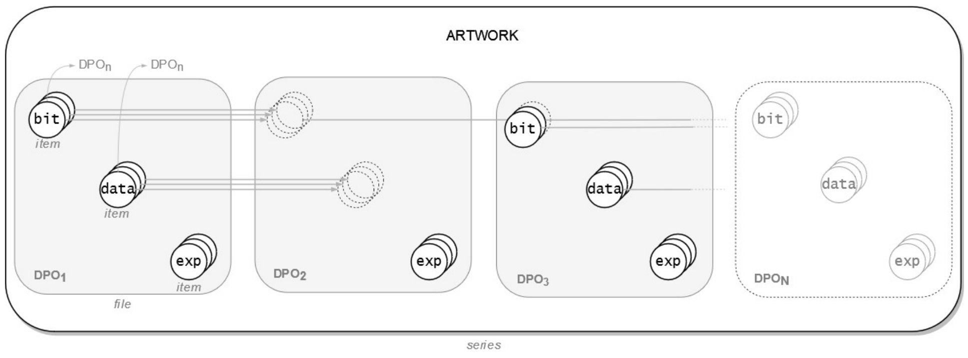


Figure 1.1: *Graphical representation of the multilevel preservation model.*

mat for the artwork (the problem of their interpretation is a matter aside), and the risk of introducing alterations for them must be avoided. In order to digitally store all the useful information about each object with an organized set of metadata and eventual further documentation produced in the digitalization processes a DPO is provided. Each digital object will be accompanied by its representation information, preservation description information (i.e. reference, context, provenance and fixity) and descriptive information;

- **Data**: this type refers to all that useful information about the realization of the artwork. Data can be represented by operating instructions, score (in the case of music), scripts, technical notes and comments about the artwork and high-level descriptions of algorithms and computational models adopted. No special attention is paid to presentation and context. This type of items is affected by a dynamic form of preservation, as it could be necessary to use new design languages, possibly developed on purpose for the installation [8];
- **Experience**: it represents any document that gives witness to some aspect of the installation. In this type interviews, audio/video recordings, usability tests of the original system, information about people (artists, performers, technicians) involved in the artwork and their roles are considered. The experience document type includes also any documentation about reactivation and preservation processes (description of approaches, used methodologies, used software etc.). The type refers more to a museum-like approach, that aims at keeping track of the history of the artwork.

In previous versions of the model, there was an additional kind of item defined as *record*. It was about any element that was modified or updated in respect of original installation, including reinterpretation of the patches and information about the context [8]. This concept has been merged in the definition of the DPO for every artwork exhibition which implicitly underlines every modification made in the evolution process.

Starting from the multilevel model presented, a graphical representation of it can be defined as in figure 1.1. It can be noticed that bit and data type items can belong to multiple DPOs. Furthermore, if some parts (or even all of them) of the original or previous exhibition are reused in an ongoing reactivation, those will also be registered as elements of the new DPO. The *multiple belongingness* of items is an important feature of the presented multilevel approach, which diverges from the general structure of the ISAD(G) in which each archival unit (named *item*) only belongs to a single file. However, the multiple belongingness can be applied only to bit or data type items due to the fact that experience-type ones are designed to document the ongoing exhibition and therefore must characterize only a unique exhibition or reactivation's DPO. This property allows the artwork to be represented not only as a group of delimited entities but rather as a dynamic evolving object. From the graphical representation, it can be noted that the order of DPOs is not given (e.g., chronological order). Although it presents a well-defined vertical structure, the model doesn't establish any kind of association between DPOs in order to avoid pre-determined discursive formulation and interpretation. For instance, the physical integrity and/or the definition of a fixed object is not denied by this model. From the evolution of a process, if multiple exhibitions are characterized by the same physical properties, the physical persistence of artwork can be deduced. The model aims to promote a high degree of freedom in the examination and arrangement of information.

1.4 Application of the model to *The time consumes* by Michele Sambin

A case study that as allowed the improvement of the model while maintaining the fundamental properties is on the *The time consumes* by Michele Sambin, a recent experience of the CSC laboratory in archiving and reactivation of an audio-video performance of the Seventies. Similar to other artworks invented by Sambin, *The time consumes* is an example of application of the *videoloop* technique, a circular system created by two video recorders in which a closed ring tape passes. The closed ring tape is created by the conjunction of both the tape extremities and it dragged by the video recorders' engines and so it spins. The first video recorder works as a recorder, while the second works as a player. The camera placed in front of the monitor is attached to the first video recorder, and the second video recorder is attached to the monitor. The preservation and reactivation processes for the artwork took place between May 2021 and May 2022. The first step was to assemble the DPOs of the first two performances (in 1979 and in 1980) by collecting bits, data and experience documents according to the preservation model presented in the previous section. In this case, DPO's set of bits remain partially incomplete because the original technological tools involved (analogue video recorders, cameras, etc.) are missing since they were not owned by the artist. For what concerns the DPOs data items, they are the documentation of Sambin's works consisting on sketches and scores (sometimes even three-dimensional paper models) with appropriate comments and instructions the has always drafted carefully. The artist also produced audio/video recordings and photos of the performance that report both the original appearance, the actions that occurred during the performance, the setting for the tools and the use of them. All these documents are considered DPO's experience items. The process of reactivation was conducted by the authors together with the artist.

For what concerns the reactivation, due to the impossibility to recover the original obsolete technologies, e.g. the video recorder, tape and analogue camera, researchers at CSC decide to transfer the entire performative system into the digital domain by implementing a migration approach. The hardware composing the original system has been substituted by modern devices: the cathode-ray tube screen has been replaced by a ultra HD LED screen; the old camera based on the cathode-ray tube has been replaced by a modern portable 4K camera; the audio system of the original performance – camera's built-in microphone and monitor's speakers – has been replaced by a pair of dedicated speakers and a single cardioid condenser microphone; the whole system formed by the video-recorder and tape has been replaced by a computer with sound and graphic cards. The *videoloop* technique has been reactivated thanks to the development of original software. All the new hardware and software items represent the bits of a new DPO record. With them, it's possible to construct a collection of data (relationship between hardware and software, elements role, operating instructions, etc.) and experience (audio/video recordings of performance in the digital domain, new interviews, etc.) documents. In the case of bit-type items, it was not possible to apply the multiple belongingness property because the artwork underwent a migration approach. Although some parts of the data (e.g., general structure and system usage) had to be replaced, the multiple belongingness property can be appled to almost all performance actions, which are always of data type.

The digital reactivation has been performed three times in 2022: at the Cas-

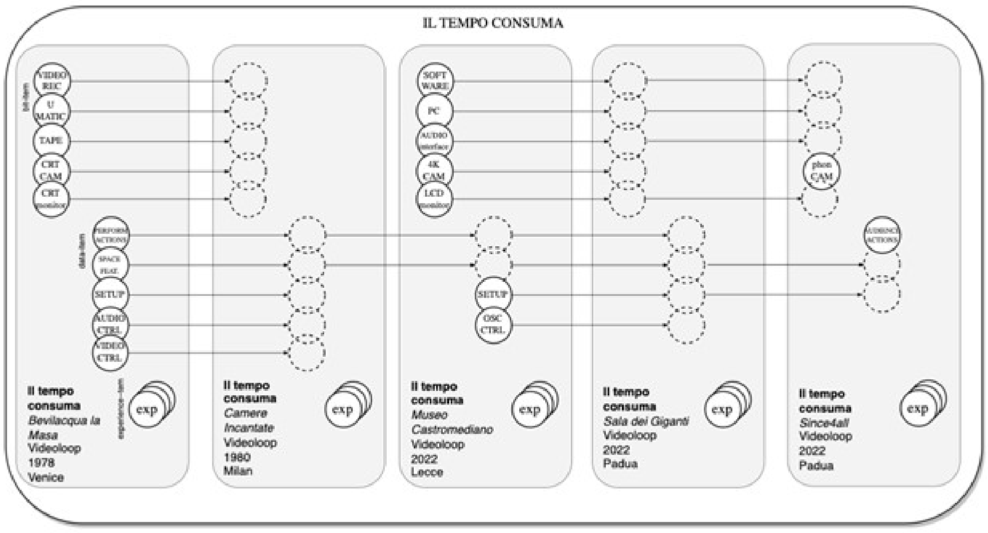


Figure 1.2: *Approximate chronological representation of The time consumes through the multilevel preservation model.*

tromediano museum of Lecce on the 19th of February, during the 800th anniversary of the University of Padua at the Sala Dei Giganti and during *Science4All*, a scientific dissemination festival of the University of Padua held on the 30th of September. In the latest reactivation, the *videoloop* has been presented as interactive installation, with some technological changes. All the preservation and reactivation works can be described via an approximate chronological representation through the presented multilevel preservation model, as can be seen in figure 1.2.

Part II

A case study: *Il Caos delle Sfere* by Carlo De Pirro

Chapter 2

The original setup

The model proposed for the preservation of interactive multimedia artworks presented in part I finds a direct application on the reactivation of an interactive music installation, *Il Caos delle Sfere: Become a Pianist with 500 Italian Lire*. The main scientific and technical partners developing the artwork were Nicola Orio and Paolo Cogo, at that time members of CSC, while Carlo De Pirro was the involved artist. It was presented for the first time at the *Giovani Artisti di Europa e del Mediterraneo* (*Biennal of the Young Artists of Europe and Mediterraneo*) in Rome in 1999. Afterwards the exhibition toured in other artistic manifestations until year 2004. Although it didn't have scientific aims, it has been based on the results of a joint research work on music interaction called "Controlled Refractions" [19] on the interaction between a pianist and a computer through a music performance.

The concept of the artwork is to use an electronic pinball, a common gaming machine, to control an automatic performance played on an automatic motorized piano named *Disklavier*, represented in figure 2.1. The main feature of it is that the performance and the generated sounds are related to the type of interaction involved in the gameplay experience. Users generally have only a loose control on the ball and the corresponding game's degree of unpredictability is very high. However, despite that normally all the electronic pinballs give auditory feedback to the player, the basic idea of the composer was to avoid a simple one-to-one mapping between the objects hit by the ball and the generated sound. According to the amount of player-pinball interaction in the evolution of the game, different note sequences are played by the Disklavier. The game starts with some pre-written sequences; when the player completes more and more in-game tasks, some automatically generated sequences start to play but the user partially controls them depending on the kind of targets he is hitting. For every new objective fulfilled by the player the style of automatic sequences changes, so it does the way the user can control

Figure 2.1: *The installation Il Caos delle Sfere during its realization at Carlo De Pirro's home in 1999. From left to right: Veniero Rizzardi, composer Carlo De Pirro, the scientific partners from CSC, Paolo Cogo and Nicola Orio. The Disklavier can be observed on the left, next to the monitor of the computer, and to the pinball machine.*

them [8]. The artistic idea behind the installation was to make a player able to govern the *chaos*, i.e. the fact that a pinball game can be very random, so that it can reach a state where the game is more controlled and predictable: this is reflected in the way the musical performance evolves.

In this chapter the technical description of the original setup is presented, in particular it's possible to identify three *nodes* making up the artwork.

2.1 Technical description: the nodes characterizing the artwork

In order to provide an overview of the elements composing the multimedia installation, it's better to divide them into *nodes*, i.e. subsystems where two or more elements contribute, cooperate and interact to perform a given task. For what concerns the considered case study, three main nodes are involved as depicted in the scheme of figure 2.2. They are:

- ***Interaction*** node: it's the node where in-game data signals are generated for the selection of the sequences that are going to be reproduced;

- ***Communication*** node: it's the node in which starting from the data produced in *Interaction* node, the sequences to be played are selected (or

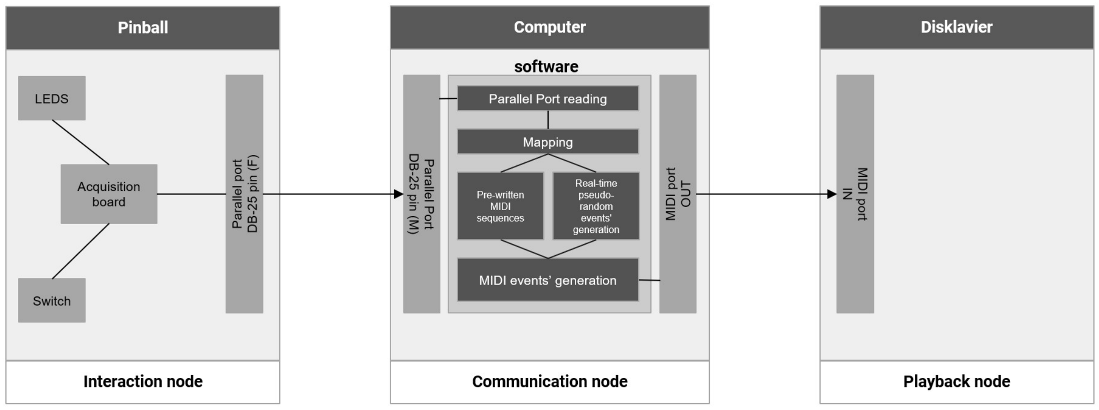


Figure 2.2: *Schematic representation of the nodes composing Il Caos delle Sfere.*

created in real-time) and corresponding events, named *MIDI events*, are generated;

- ***Playback* node**: it's the node where MIDI events generated in the *Communication* node are used to move Disklavier's keys.

The way the nodes interact between each other is done through two interfaces: the *DB25 Parallel Port* interface where *Interaction* node's data are injected and made available for the *Communication* node; and the *MIDI* interface in which the MIDI events generated in the *Communication* node arrive at the *Playback* node.

2.1.1 *Interaction* node

As its name says, the *Interaction* node is the part of the artwork where a user can be involved and interact with the system in order to determine the final reproduced melody. This node is centred around the *Creature from the Black Lagoon* pinball machine over which a user can play and indirectly generate the useful data to be elaborated in the *Communication* node. All data are acquired by an ad hoc designed acquisition board which in turns makes them available to the *Communication* node through the DB25 Parallel Port interface.

2.1.1.1 *Creature from the Black Lagoon* pinball

The main role in the *Interaction* node is played by a popular electronic pinball machine named *The Creature from the Black Lagoon* (figure 2.3). Based on the movie of the same name, it was released in December 1992 by Midway Games Inc [16].

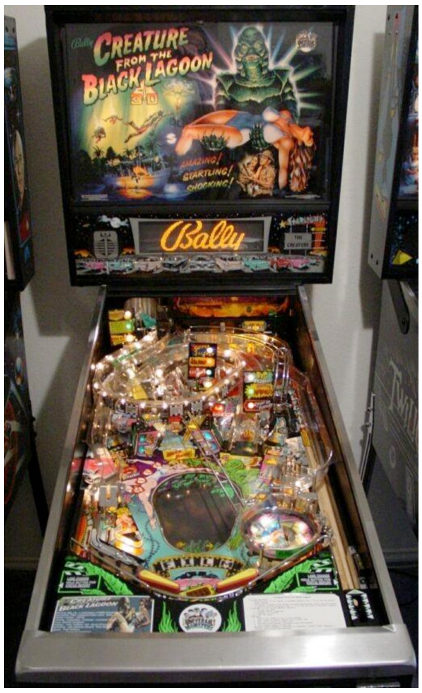


Figure 2.3: *Creature from the Black Lagoon pinball*

This game machine was one of the first pinballs to introduce in a game the idea of a story and different levels that the player can achieve by progressively fulfilling a certain number of goals. This is a key feature which in fact was what influenced the technical and artistic choice of the installation [8].

From the gameplay point of view, the way the artwork generates the performance is strictly related to the main objective of a game which is to activate four letters "F-I-L-M". The activation of them is associated with combinations of targets hit by the ball and it's necessary for the player to progress with the story. In fact, once the letters are all on, the game can enter the *Extra-Ball* (EX) mode in which multiple balls are in play. It's also worth to mention three in-game actions during the multiball mode that impact the final performance of the artwork. The player has to *Search* (CS) the Creature in one of three positions, after that it has to *Rescue* (RE) the girl and finally he can score a *Jackpot* (JK) by finding the correct scoop in the table where the Creature hides the girl [16].

All the aforementioned events contribute to create an high level of unpredictability which reflects on the final obtained musical execution. The way the gameplay evolves allows also to reward good players: the more goals are hit, the more complex and interesting is the generated performance on the Disklavier.

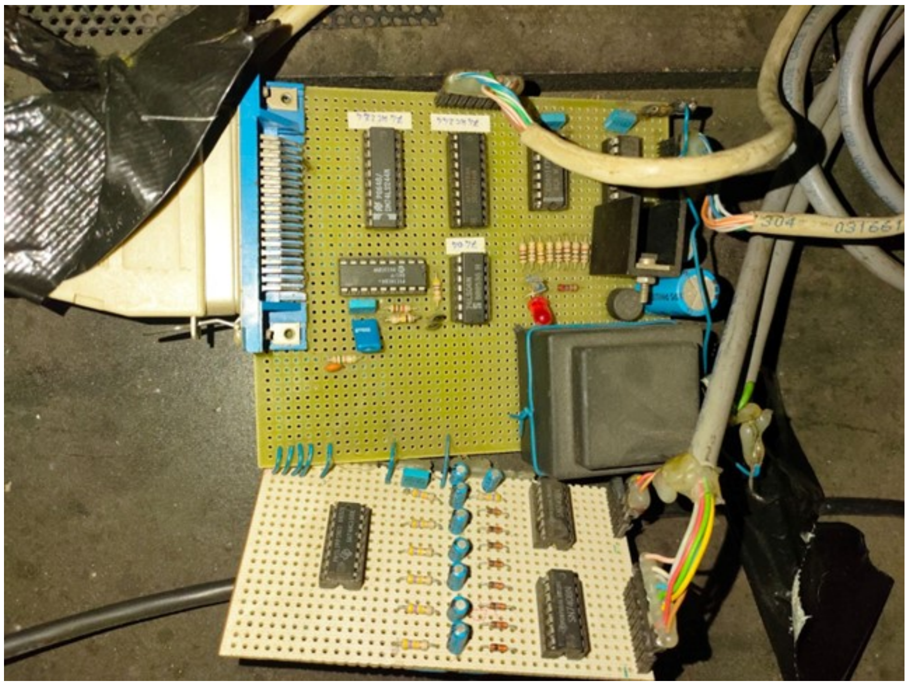


Figure 2.4: *The acquisition board*

2.1.1.2 The ad hoc designed acquisition board

The main technical issue the developing team had to face was how to monitor the game minimizing the need to interface with the pre-existing electronic inside the pinball [8]. To do so, it has been chosen to split the signal coming from the pinball switches to track the targets hit by the ball and the lights associated with the letters F I L M and with events EX CS RE JK as described in the previous subsection. It can be noted that in this way it is only possible to estimate the level of the in-game story and that some of the features (i.e. the actual number of points gained) have been neglected so that they are not useful for the final result. The acquisition was made through an electronic circuit (figure 2.4), designed ad hoc by Paolo Cogo, which is contained inside the pinball machine.

With reference to the figure, it's possible to see the presence of several multiplexers, among which there is a Quad 2 line to 1 line multiplexer like the 74LS157. However, due to the lack of documentation for the original project, a schematic representation of the ad hoc designed circuit was not found so a more detailed components' description cannot be provided. Despite that, by inspecting the way the pinball communicates with the *Communication* node, the routine through which data are made available for the latter can be estimated. Firstly, it consists on the processing of the voltage signals (characterized by TTL logic levels) of the lights associated with the letters (or events) which are then passed to the input of one of the multiplexers. In addition to that, the assessment of the signal identifying the switch corresponding to the last hit target is performed thanks to the presence of 8 lines of TTL logic levels voltages. Each of these 8 lines defines a bit and the overall byte can be used to produce a mapping between an active switch and a byte value, as what can be found in the original pinball data-sheet. Such a value is written in the lines and it is taken from the output of a switch matrix circuit contained within the pinball circuit board: a microprocessor constantly strobes to determine what switch is on by doing a *row by column* type of check [16]. All the data acquired and passed to the board's multiplexers are finally ready to be sent to the *Communication* node via the DB25 Parallel Port interface. The way the data transfer is done over it will be described later.

2.1.2 *Communication* node

The role of processing the data coming from the pinball machine in order to generate a sequence of sounds to be then reproduced for the users is given to the *Communication* node. It can be also seen as that part of the artwork which makes the pinball machine (the interface through which a user can interact with the artwork) and the Disklavier (the instrument which reproduces the sounds generated over the gameplay evolution) communicate. This subsystem of the artwork is entirely represented by the original computer machine where the project developers created several executables, among which there is the one in charge of testing the communication with the pinball machine, the one in charge of testing the communication with the Disklavier and, more importantly, the main software dedicated to process, generate, and modify melodic sequences according to the indication provided by the composer. In particular, the latter rely on the *MidiShare* environment to generate the MIDI messages and events. The data acquired from the pinball machine is accessed via the Parallel Port while the sounds generated are sent to the Disklavier in the form of MIDI events via the MIDI port.

2.1.2.1 The original computer machine

The original computer machine over which the software required to make the artwork perform runs is a Windows 95 machine, in use at the CSC group also for several other projects and artworks in the early 2000s. In order to implement the communication with the ad hoc designed board inside the pinball machine, they developed also a dedicated board with a male DB25 connector

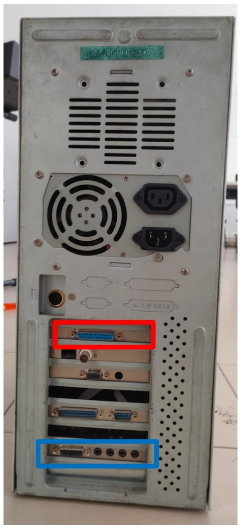


Figure 2.5: *The Parallel Port (highlighted in the red rectangle) and the sound card (highlighted in the blue rectangle) of the original computer machine.*

for the Parallel Port interface (with reference to figure 2.5, it's indicated by the red rectangle). The motivation for that is the absence of such type of connector in most of the old PC machines, where only the female version of it is present. The developers also equipped the machine with a sound card (with reference to figure 2.5, it's indicated by the blue rectangle) in order to provide a MIDI port for the transfer of MIDI events to the Disklavier.

The PC also contains several versions of the aforementioned software and the corresponding source codes that were progressively modified for the different exhibitions the artwork went through. On top of this, the PC provides also a set of *.wri (*Microsoft Write Documents*) files over which many sequence of MIDI events are listed. They were generated starting from the melodic sequences composed by Carlo De Pirro just for the artwork and exported in WRI format by using the *Finale* program (not included in the PC). The files are then inputted in the software in order to generate the MIDI events to be sent to the Disklavier.

2.1.2.2 *MidiShare* usage for the reproduction of MIDI events

As previously mentioned, all the pre-written sequences that are then used in the software are contained in *.wri files. Each of the files contains a set of MIDI events (whose structure will be described in the subsection 2.2.3.3) that are what needs to be passed to the Disklavier in order to generate sounds in the very end. The process by which such MIDI events are sent is handled by the

MidiShare environment which enables the communication via MIDI interface and MIDI port.

MidiShare is an open-source software developed by Grame in 1989 in order to provide a development kit to build real-time music software. It is a real-time multitasks music operating system specially devised for the developing of musical applications. The main features it provides are [1]:

- **High level musical events handling**: it's a fully structured and time stamped with a millisecond resolution MIDI events manager;
- **Efficient scheduling**: it's in charge of delivering events at their falling dates through a scheduling algorithm which ensures a very low and constant time overhead per event;
- **Enabling inter-applications communication**: it's equipped with a communication manager which routes the events to the client applications, according to the connection set between them;
- **Real-time tasks managing**: it provides the control the real-time behavior of an application, in particular for user-defined function calls that can be scheduled for future need.

The conceptual template of a typical *MidiShare* routine can be found in the figure 2.6.

The reason why the development team decided to rely on the *MidiShare* environment is because, as presented in the features, it results to be very precise when dealing with MIDI events that are highly time-sensitive as will be later described. Most of the functions developed for the algorithm, in particular the ones to output pre-written or newly generated sequences, are in fact called and scheduled via the libraries' functions that *MidiShare* provides. This features suits exactly with the requirement of the algorithm to schedule MIDI events at given future time instants, one after the other, to generate a continuous flow of notes. For what concerns the *MidiShare* functions' calls, they require three parameters mainly: the date at which a call is scheduled for, the address of the function to be called and the reference number of the *MidiShare* instance set at the beginning of the process. Given this, it can be understood why *MidiShare* is a fundamental tool to enable the transmission of MIDI events via the MIDI port contained in the ad-hoc audio board designed for the original PC.

Starting from the *MidiShare*'s library, a custom made function was coded for the events transmission task: a copy of the MIDI event is sent to the *MidiShare* instance's destination according to a date which specifies when the destination will actually receive the event. The *MidiShare* instance then transmits the event

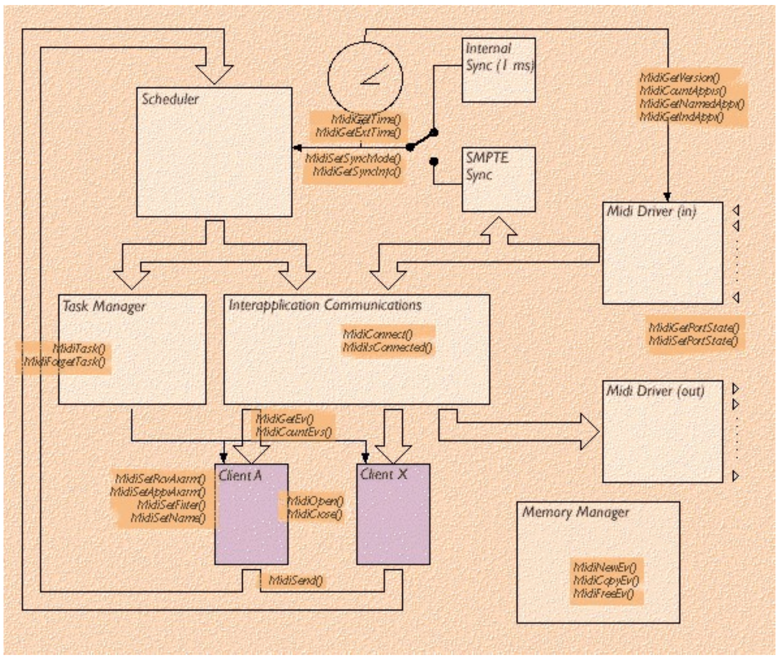


Figure 2.6: *Conceptual template of a typical MidiShare routine as presented in the original documentation [1].*

to the MIDI port using the correct MIDI port number in the operating system provided by the user. In addition to that, the environment comes up with the possibility also to flush and eliminate all waiting function calls in the *MidiShare* instance list. This feature is fundamental in order to stop the current play sequence of MIDI events when, for example, a new task is completed during the gameplay performance.

2.1.3 *Playback* node

The final node composing the *Il Caos delle Sfere* artwork is the *Playback* node. It's the subsystem dedicated to the reproduction of the musical performance as it was programmed by the algorithm and the gameplay evolution. Here the MIDI events coming from the *Communication* node via the MIDI port are translated into audible sounds to let the gamer hear the result of its game experience. The *Playback* node is the only sound source of the artwork and it consists on either a grand or a upright piano depending on the place of the installation. For the exhibitions in which the artwork was exposed the Disklavier piano is the reproducing machine they choose.

2.1.3.1 Disklavier

As previously defined, the playback machine associated with the installation is the *Disklavier*, a type of acoustic pianos manufactured by *Yamaha Corporation* starting from 1987 [27]. The typical Disklavier is an acoustic piano integrated with electronic sensors for recording and electromechanical solenoids for piano-style playback, i.e. it can be described as an automatic motorized piano. It can be either used to record a performance thanks to sensors which registers the movements of the keys, hammers and pedals; the performance data can be then saved according to many formats, mainly according to the MIDI protocol. It can be also used to perform a playback of a musical performance: solenoids move the keys and pedals to reproduce the specific performance. Generally, non-contact optical sensors detect the movement of the keys and pedals to get a high level of fidelity, delivering a performance almost indistinguishable from that of a real pianist [27]. On top of this, modern Disklaviers typically include a large set of electronic features, such as a built-in tone generator , speakers and MIDI connectivity that supports communication with computing devices and external MIDI instruments. Its usage spans from a didactic approach, where a student can record daily practice sessions to be then checked by instructors, to a more engineering approach where a variety of devices can be adopted to control or operate the instrument, including infrared handheld controllers, handheld wi-fi controllers, applications running on portable devices, etc. Disklaviers are manufactured in the form of upright, baby grand, and grand piano styles.

2.2 Nodes' interfaces

The main issue the developing team had to face was to enable the inter-node communication, in particular the one between Interaction and *Communication* nodes. They had to find a reliable way to transfer data asynchronously from the ad hoc designed board inside the pinball towards the PC, on one side to move all the computational complexity to the PC's CPU and to make the acquisition board design as simple as possible in terms of electronic components. On the other side to provide a sufficiently high bitrate transmission to avoid delays in the data acquisition process that would eventually lead to a stop of the sound flow in the end. To comply these requirements, they decided to adopt the Parallel Port interface also motivated by the fact that Windows 95 Os gave the possibility to access directly the serial and Parallel Ports.

For what concerns the *Communication-Playback* nodes's connection, they decided to go with the MIDI interface. That was (and still represents) the most

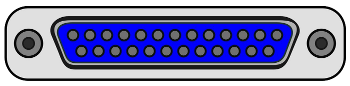


Figure 2.7: *Representation of a Parallel Port's 25-pin female D-sub connector.*

suitable communication protocol that connects several electronic musical instruments, computers, and related audio devices for controlling, playing, editing and recording music. The choice was also dictated by the simplicity and the robustness that such a protocol provides.

2.2.1 DB25 Parallel Port interface

Despite being used in practice for several years, the Parallel Port interface was fully standardized in 1994 under the *IEEE 1284* standard [2]. The main content of it is the definition of a signalling method for asynchronous, fully interlocked, bidirectional parallel communications between hosts and printers or other peripherals [2]. This means that here the devices don't share a common clock, and the timing of events is defined in relation of one event to another. In addition to that, every control signal is acknowledged with an answering control signal, ensuring that the transmitting device sends data only when the receiving device is ready [15]. The interface described in the standard is a bidirectional extension of the already existing PC parallel interface, developed in the 70's by *Centronics* and originally designed as a printer port.

The original PC's Parallel Port had eight outputs, five inputs and four bidirectional lines but only in [2] IEEE introduced a number of distinct communication modes for it, using the already-used signals in the Centronics' interface making the interpretation of such signals depending on the implemented mode.

About the hardware, most Parallel Ports use the 25-contact D-sub connector (standardized as the *IEEE 1284-A connector*) represented in figure 2.7. For Parallel Ports there is also the possibility to use the 36-pin connectors, i.e. the *IEEE 1284-B* and the *IEEE 1284-C*. For the purpose of the project, only the 25-contact connector will be considered.

The input and output of the Parallel Port are characterized by TTL logic levels. The current that can be sunk and sourced varies from port to port. Most Parallel Ports are implemented in *ASIC* (Application Specific Integrated Circuit) and can sink and source around 12 mA [20].

Pin:D-sub	Signal	Function	Register		Hardware
			Name	Bit #	Inverted
1	nStrobe	Strobe D0-D7	Control	$\overline{C0}$	Yes
2	D0	Data Bit 0	Data	D0	No
3	D1	Data Bit 1	Data	D1	No
4	D2	Data Bit 2	Data	D2	No
5	D3	Data Bit 3	Data	D3	No
6	D4	Data Bit 4	Data	D4	No
7	D5	Data Bit 5	Data	D5	No
8	D6	Data Bit 6	Data	D6	No
9	D7	Data Bit 7	Data	D7	No
10	nAck	Acknowledge	Status	S6	No
11	Busy	Printer Busy	Status	$\overline{S7}$	Yes
12	PaperEnd	Paper end, empty (out of paper)	Status	S5	No
13	Select	Printer selected	Status	S4	Yes
14	nAutoLF	Generate automatic line feeds	Control	$\overline{C1}$	Yes
15	nError (nFault)	Error	Status	S3	No
16	nInit	Initialize printer (Reset)	Control	C2	No
17	nSelectIn	Select printer (Place on line)	Control	$\overline{C3}$	Yes
18	Gnd	Ground return for nStrobe, D0			
19	Gnd	Ground return for D1, D2			
20	Gnd	Ground return for D3, D4			
21	Gnd	Ground return for D5, D6			
22	Gnd	Ground return for D7, nAck			
23	Gnd	Ground return for nSelectIn			
24	Gnd	Ground return for Busy			
25	Gnd	Ground return for nInit			

Figure 2.8: *Pin Assignments of the D-Type 25 pin Parallel Port Connector.*

2.2.1.1 The signals involved

Most of the signals used in the Parallel Port and the functions of each one of the 25 contacts are named according to a convention established by *Centronics* when developing the original Parallel Port for dot-matrix printers. Therefore, all ports' names reflect that use [15]. The characterization of each pin can be found in the figure 2.8. In such a table, the letter "n" is used in front of the signal name in order to denote that the signal is active LOW and "Hardware inverted", i. e. the signal is inverted by the Parallel Port's hardware. If +5 V (Logic 1) is applied to such pins, they would return back a 0 in their corresponding bit.

The standard Parallel Port is composed by three 8-bit port registers: a PC can accesses the Parallel-Port signals by reading and writing to them. Registers are defined as the *Data*, *Status* and *Control* registers. The Data register, whose bits are denoted as (D0-D7), is about the byte written to the data outputs or, in case of bidirectional communication, the byte read at the connector's data pins. The Status register holds the logic states of five inputs, denoted as S3 through $\overline{S7}$ (the bar indicates that the corresponding signal is Hardware inverted). Instead, bits defined as S0–S2 don't appear at the connector. The Control register collects the states of four bits, named as $\overline{C0}$ through $\overline{C3}$ while bits named C4 to C7 don't appear at the connector. The characterization of each pin, in their conventional use for printers, can be found in [20].

The signalling method standardized in [2] provides five modes of host-peripheral communication, each ones consisting of one or more phases. Additional phases may be defined in order to cover initialization and transitions between communication modes. The modes are the *Compatibility Mode*, the *Byte Mode*, the *Nibble Mode*, the *Extended Capabilities Port* mode and the *Enhanced Parallel Port* mode.

2.2.2 Nibble Mode overview

By inspection of the project's original code, it is possible to realize how the communication mode between the pinball and the PC is the Nibble Mode. Therefore, only this specific mode will be treated in details.

The Nibble mode is one of the modes capable of providing peripheral-to-host data transfers. It's also the preferred way of reading 8 bits of data without using the data lines [20]. In fact it implements the transmission of 4 bits (a *nibble*) at a time. In this mode status signals are used to read each nibble while the data bits are not in general considered, even if there is the possibility for the host to write or read on them. For what concerns the control lines, IEEE 1284 defines only the usage of pin 14 (bit $\overline{C1}$).

Nibble mode's functionality consists of two parts or phases: the *Data transfer* phase which includes the writing of a byte from the peripheral to the host and the *Idle phase* that defines the signal states for the port when data transfer is not occurring [15].

The Data transfer phase is made by the following operations [9]:

1. The host (PC) indicates that it is ready to receive the first nibble by setting the ($\overline{C1}$) pin (defined also as *HostBusy*) LOW;

2. The peripheral places the first nibble on the status lines S3, S4, S5 and $\overline{S7}$;

3. The peripheral indicates that the data is valid on the status line by setting pin (S6) (defined also as*PtrClk*) LOW;

4. The host reads from the status lines and sets *HostBusy* HIGH to indicate that it has received the nibble, but it is not yet ready for another transmission;

5. The peripheral sets *PtrClk* HIGH as an acknowledgement to the host;

6. Repeat steps 1-5 for the second nibble.

After it receives a byte, the host may bring *HostBusy* LOW and wait for more data or it may leave *HostBusy* HIGH to prevent the peripheral from sending another nibble.

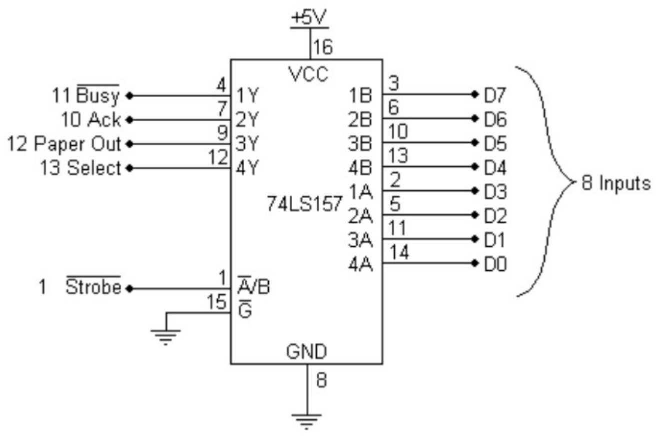


Figure 2.9: *Multiplexer representation of the Nibble Mode functionality [20].*

In order to construct a byte from the two nibbles, software can then be used. Despite being simpler than other modes, this technique can be proven to be slow. This is because of the additional instructions needed to read and compose the byte that make the aforementioned operations software-intensive [9].

2.2.2.1 Implementation of the Nibble Mode

Since Nibble mode reads data through the port without setting it in the reverse mode, i.e. without using the data lines to receive the data, a possible implementation of its functionality can be done by applying inside the peripheral a Quad 2 line to 1 line multiplexer like the 74LS157 (depicted in figure 2.9) to read a nibble of data at a time and then the following nibbles sequentially.

In details, the Quad 2 line to 1 line multiplexer acts as four switches. With references to figure 2.9, when the A/B input (represented in the hardware by the Strobe line $\overline{C0}$ which is Hardware inverted) is LOW, the A inputs are selected, e.g. 1A passes through to 1Y, 2A passes through to 2Y. When the A/B is HIGH, the B inputs are selected. The Y outputs are connected up to the Parallel Port's status port, in such a manner that it represents the most significant nibble of the status register. The overall exchange of data works as follows: when A/B is LOW (thus Bit 0 of the Control register must be set to 1 to get a LOW on the Strobe pin) the host reads the least significant nibble. It then sets A/B to HIGH in order to read the most significant nibble. Finally the two nibbles are composed together via software to make a byte (whose bits corresponding to the Busy line, $\overline{S7}$, have to be inverted). It may be also necessary to add delays in the process, if the incorrect results are being returned.

2.2.2.2 Understanding the pinball to PC communication

In order to understand how the communication between the PC and the pinball works, it's fundamental to analyse the implementation of the electric circuit inside the original installation and to inspect the corresponding code developed for the project. As described in subsection 2.1.1.2, only two features of the pinball are tracked. Firstly, the signals associated to the pinball lights are monitored in order to determine the level of the game. Secondly, the pinball switches are observed to track the targets hit by the ball and the overall assessment is made through the ad hoc designed acquisition board.

To acknowledge the operational mode associated with the Parallel Port communication, an inspection of the software developed to process the information acquired (found on the PC used to run the overall interactive music installation) reveals how the operational mode chosen was the *Nibble Mode*. More specifically, a multiplexer implementation as described in subsection 2.2.2.1 was developed inside the ad hoc designed circuit for the pinball.

2.2.2.3 Testing the communication

By inspecting the software, it was also possible to find an executable appropriately developed to test the communication between the pinball and the PC and what piece of information is transmitted trough the lines depicted in the scheme of figure 2.9. Overall, the procedures involved in the communication are the following:

1. The PC acquires the first nibble (information about lights "F", "I", "L" and "M");
2. The PC sets the pin 1 ($\overline{C0}$) to HIGH to perform the switch in the multiplexer;
3. The PC acquires the second nibble (information about lights "EX", "JK", "RE" and "CS");
4. The PC read the data lines (D0-D7) to acquire the value corresponding to the last target hit by the ball;
5. The PC sets the pin 1 ($\overline{C0}$) to LOW to perform the switch in the multiplexer (for the acquisition of the following first nibble).

It's worth to notice how the data lines, despite not being involved in the standard Nibble mode operations, are here used in order to communicate some values (between 0 and 127) which denote the last target hit by the ball while

Port bit	Function
S4	in first read checks if "F" light is on and in second read checks if "EX" light is on
S5	in first read checks if "I" light is on and in second read checks if "JK" light is on
S6	in first read checks if "L" light is on and in second read checks if "RE" light is on
$\overline{S7}$	in first read checks if "M" light is on and in second read checks if "CS" light is on
$\overline{C0}$	used to swap between the read of the MSnibble and the LS nibble
D0 - D7	used to read the value of the last target hit by the ball

Figure 2.10: *Pinball signals and their function.*

the status lines are used to communicate the condition of the considered lights ("F", "I", "L", "M", "EX", "JK", "RE" and "CS"). The table in figure 2.10 summarizes all the pins involved and their functions. The inspection of the aforementioned software allows also to understand the way the "checks" of the lights are made. In the original code written in C, some bitwise operations like AND, OR, XOR and RIGHT-SHIFT (and some other functions specific for the I/O operations with Parallel Ports) were performed to control if each of the status pins S4-S6 was set to LOW and if the status pin $\overline{S7}$ was set to HIGH (due to the fact it's Hardware inverted). If a pin verifies its own condition, then it means that the corresponding light is on.

2.2.3 MIDI interface

As previously described, the PC-Disklavier communication is enabled thanks to the transmission of MIDI *events* through the MIDI port interface. In order to understand what they are, it's necessary to specify what the MIDI technical standard is. MIDI technology was standardized in 1983 by the MIDI Manufacturers Association (MMA) and is still nowadays the most widely used protocol allowing the connection between computers, audio devices and musical equipment. The main feature of the MIDI technology is to provide an *operational representation* of music. Music is highly used in multimedia applications therefore a media type for music is necessary to allow analysis, processing, storage and transmission of it. Music can be represented by audio samples and encoded losslessly or lossy according to standards like WAV or MPEG-1 or it can be described given its structural representations where there is information about the internal structure of the music. In this framework, it's possible to distinguish two kinds of music representation: the *Operational* one, which defines the exact timings for music and physical descriptions of the sounds to be produced and the *Symbolic* one which uses descriptive symbolism to describe the form of the music and allow great freedom in the interpretation [11]. MIDI is an example of the first type.

The original MIDI standard specifies a protocol for a digital information interchange for different types of devices in multimedia applications. In addition to that it establishes the physical connector and the hardware as well as the message format for connecting devices and controlling them in real time. It sets up the electronic circuit for the MIDI interface (the MIDI *port*), the bitrate of the information flow and the specification of the byte instructions composing the protocol. The most important part of MIDI is the *Message specification* (or MIDI *Protocol*). In fact the so called *MIDI messages* are used inside any device to generate music and the MIDI protocol allows their transmission between different musical instruments or PC through the MIDI interface. The precise meaning of MIDI messages is finally determined by the *General MIDI* (GM), a standardized specification for electronic musical instruments generating or receiving such messages. While the MIDI protocol provides mostly the interface for the communication, General MIDI requires that all compliant MIDI instruments meet a given minimal set of characteristics and it sets out specific interpretations for many parameters and messages (mainly the control ones) which are left unspecified in the MIDI protocol. For example, GM can univocally determine a musical stamp.

The final part of MIDI is the *Standard MIDI File* (SMF) which represents a format specifying and determining how MIDI messages can be stored in permanent supports. It is used to distribute music playable on MIDI players via the *MIDI files* (*.mid), containing one or more sequences of MIDI messages together with their timing information and all the instructions needed to execute them in an instrument. For the purpose of the work, the SMF won't be treated.

2.2.3.1 MIDI hardware

From the hardware point of view, the MIDI interface consists on an serial interface of asynchronous type over which data is sent as a bit sequence, enabling the connection of a group of MIDI devices together. This means that bits are transmitted via MIDI port only when they are available at the device generating them. The bitrate that is set through the interface is 31 250 bit s^{-1} and, since MIDI messages are made by 10-bits words, 320 µs are required for the single word transmission. The connectors involved in the MIDI interface are of 5 pin DIN types, as represented in figure 2.11, whose winding is specifically used only for the MIDI standard. For what concerns the MIDI ports, the connectors can be distinguished in: *MIDI In* ports receiving data as input, *MIDI Out* ports generating data as output and *MIDI Thru* ports, dedicated to the transmission of a copy of data received in input.

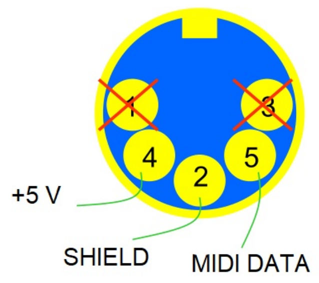


Figure 2.11: *5 pin DIN connectors for MIDI ports*

2.2.3.2 MIDI messages

MIDI messages are the carrier used by MIDI devices to communicate with each other. They are not representing the evolution of a waveform but they are control information composing a one-way connection from the MIDI Out connector of the sending device to the MIDI In connector of the receiving device. Each such message represents a common musical performance event or gesture like picking a note and then striking it or setting typical parameters available on electronic keyboards. As previously anticipated, MIDI messages are composed of 10-bit words (known as *bytes*) whose structure is 1 start bit, 8 data bits, and 1 stop bit. The majority of MIDI messages is characterized by only three bytes (meaning that only about a thousand messages per second can be transmitted) but in general a MIDI message can consist of from one to several thousand bytes of data. The receiving device knows how many bytes to expect by analysing the value of the first byte of it. This byte is known as the *Status byte*, specifying the meaning of the following ones, called *Data bytes*. Status bytes always have the most significant bit equal to 1 (as opposed to the 0 of the Data bytes) and it is used to inform the receiver as to what to do with incoming data. All messages include in the corresponding Status Byte information about the *channel number*. The channels can be described as ”paths for the communication” in the message flow and they are used to separate ”voices” or ”instruments”. The MIDI protocol specifies 16 possible channels and it grants the ability to multiplex 16 channels onto a single wire. This makes it possible to control several instruments at once using a single MIDI connection. When a MIDI instrument is capable of producing several independent sounds simultaneously (a multi-timbral instrument), MIDI channels are used to address these sections independently. In addition to that, according to the status byte, MIDI messages can be classified in *System Messages*, sent to all channels and received by all the devices connected through the MIDI connection, or *Channel Messages* which apply to a specific channel (included in their status byte) and received only by devices listening to that channel. With reference to the figure 2.12, in

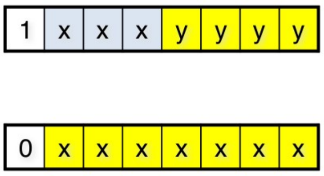


Figure 2.12: *Status and Data bytes' structure.*

the Status bytes bits ”x” denote the type of the message while bits ”y” indicate the destination channel ($2^4 = 16$ possible choices) in Channel messages while in System messages they specify the sub-type of System message represented. Instead in Data bytes, ”x” bits denote the value of the parameter contained in it (for a total of $2^7 = 127$ possible choices).

Channel Messages can be further classified as being either *Channel Voice Messages*, or *Channel Mode Messages*. Channel Voice Messages carry musical performance data, and these messages comprise most of the traffic in a typical MIDI stream. The messages in this category are the Note On, Note Off, Polyphonic Key Pressure, Channel Pressure, Pitch Bend Change, Program Change and the Control Change messages. In the specific MIDI flow for the *Il Caos delle Sfere* artwork, only three types of Channel message are used. They are:

- **Note on** and **Note off**: in MIDI systems, the activation of a note and the release of the same note are considered as two separate events. When a key is pressed on a MIDI keyboard instrument or keyboard controller, a Note On message is generated and outputted on the MIDI Out port. Its Status byte contains the value 1 in the ”x” bits and the selected Channel number c in the ”y” bits. The Note On status byte is followed by two data bytes, which specify note number (indicating which note was pressed) and key velocity (how hard the key was pressed). The note number is used in the receiving device to select which note should be played, and the velocity is normally used to control the amplitude of the note. The note numbers start with 0 representing the lowest C while middle C is note number 60. A tone with note number (also known as *MIDI pitch*) p has frequency

$$f = 440 \times 2^{\frac{p-69}{12}}\,\text{Hz} = 440 \times s^{p-69}\,\text{Hz} \tag{2.1}$$

  and a note with frequency f Hz has MIDI pitch

$$p = 69 + 12 \log_2 \frac{f}{440} \tag{2.2}$$

where $s = 12\sqrt{2} \approx 1.059$ is the semitone frequency ratio. When the key is released, the keyboard instrument or controller will send a Note Off message (0 in the "x" bits). The Note Off message also includes data bytes for the note number and for the velocity with which the key was released. The Note Off velocity information is normally ignored. Note Off is actually not used very much. Instead, MIDI allows for a shorthand, known as running status: a MIDI message can be sent without its Status byte (i.g. just its data bytes are sent) as long as the previous, transmitted message had the same Status. In the case of Note on, this shorthand is interpreted as Note off. This allows to save on average 33% of traffic over the communication [11];

- **Control Changes**: with Status byte containing the value 11 in the "x" bits, this message is used to set a particular controller's value. A controller is any switch, slider, knob that implements some function (usually) other than sounding or stopping notes. Each command has two parts, defining which control to change and what to change it to. Controllers are numbered from 0 to 121, and some of them have defined purposes (e.g. control n°1 is Modulation wheel, n°7 the volume, etc.) while values 122-127 are reserved for special mode messages which affect the way a synthesizer responds to MIDI data. The controller number is specified in a first Data byte. In addition to that there is a second Data byte whose value can be between 0 and 127 for controllers with continuous values while for switches it can be 0 (OFF) or 1 (ON).

Channel Mode messages instead affect the way a receiving instrument will respond to the Channel Voice messages.

On the other side System Messages apply to all machines and carry information that is not channel specific, such as timing signal for synchronization, positioning information in pre-recorded MIDI sequences, and detailed setup information for the destination device.

2.2.3.3 MIDI events

All the previous description about MIDI messages didn't considered the presence of *timing*. Timing in music and therefore in the MIDI protocol is necessary: it specifies when a MIDI message should be sent through a MIDI interface to be then used to produce sounds.

In music, the fundamental time unit is the *beat*, also defined as the way a musician counts the note to make them stay in synch with one another. The *tempo* is instead defined as the speed of the sequence of beats in a musical performance and it is measured in BPM (*beats per minute*). In general notes' durations can be specified in a beat-based manner: a quarter note is normally one beat long, an half note is 2 beats long while a full note is 4 beats long. When dealing with timing in MIDI, musical timing is defined in fractions of a musical beat: the timeline is therefore split into chunks of a beat. Each tiny fraction of a beat is called a *tick* (the smallest unit of time in MIDI) and the number of ticks per beat can be changed. This piece of information is what is considered for MIDI messages composing a SMF: each MIDI message is equipped with a time field which tells how many ticks have passed since the last message. Instead, when dealing with MIDI messages as the ones contained in the pre-written (or real-time-generated) sequences which don't compose a SMF, timing is specified in microseconds. Each message is therefore associated with an additional field which specifies that it should be played after a given amount of microseconds (named MIDI *delta time*) after the previous one [11]. The contribution of delta time and MIDI message composes the MIDI *event*, which is the basic building block connecting the gameplay experience to the audible sounds in the *Il Caos delle Sfere*. From this argument, as anticipated in subsection 2.1.2.2, it can be understood how the *MidiShare* environment is suitable for the MIDI events handling due to the high time sensitivity it grants.

All the events characterizing the pre-written sequences contained in *.wri files can be decomposed into four fields: the delta time, the type of message (for the installation only Channel message are concerned), the pitch (or the controller number for Control change messages) and the velocity (or the value for the control in Control change messages). In fact it is possible to see how all the sequences in *.wri files are made by rows of 4 integers representing exactly the 4 aforementioned fields. The generation of those field was made thanks to the Finale tool starting from the compositions developed by the composer Carlo De Pirro for the artwork.

The way *MidiShare* treats the events is the following: functions contained in the library named *MidiTask* and *MidiDTask* allow to schedule the transmission of an event via the MIDI port spacing the events by a quantity given by the delta time field which, as all the 3 other fields, is an argument passed to those functions. The two are then used inside custom made functions developed for the artwork algorithm, to make it possible to read and play both pre-written and real-time generated-sequences.

Chapter 3

The algorithm and the C code for the original installation

The core of the artwork is the software that enables the sound generation as the gameplay evolves. It processes the raw information coming from the pinball machine and outputs a sequence of MIDI events to be reproduced by the Disklavier. In the aim of minimizing the presence of digital media, the developers decided to provide the software with almost no graphical user interface [8].

Also in this case, due to the lack of documentation, it's not possible to retrieve a detail analysis of how the software works and how it produces an output. This is due to the fact it was developed in direct collaboration with the composer with a *trial-and-error* approach and, by consequence, it has been left mostly undocumented. In fact several variations to the code were carried out and adjusted for the different exhibitions the artwork participated in. Therefore, in order to understand its working process, an analysis of the source C code associated with the software is needed. The problem is that, by inspecting the PC where the installation was originally running, more than ten different versions of the same C code can be found, some with minor changes but in general very different with respect to another. This is what makes the reactivation of *Il Caos delle Sfere* complicated from the engineering point of view. This will be discussed in chapter 4.

Despite several versions of the same code, the artistic concept behind all of them is that the gameplay evolution and the targets and in-game events characterizing it are what is used in order to create a musical form. To each gameplay action (corresponding to the ″F″ ″I″ ″L″ ″M″ ″JK″ ″RE″ ″EX″ ″CS″ power on) there corresponds a level for the artwork algorithm characterized by a different type of melody and sounds, more and more interesting and complex, to be heard by the player (the levels defined inside the algorithm have to be not confused with the in-game story levels defined within the *Creature from the Black Lagoon* pinball gameplay). This makes the player's ability fundamental to have a complete and advanced experience of the artwork performance.

The way the MIDI events are generated in output is handled according to two approaches. The software makes use of pre-written sequences, contained in WRI type of files, generated from Carlo De Pirro's melodic sequences and associated to different levels for the game. Secondly, it generates also real-time events called *Refractions*: they are automatically generated sequences whose notes' pitches and durations depends on the actions completed in game and on the targets hit by the ball. The word Refractions (or Controlled refractions) refers to a set of musical multiplications of the musician's gestures performed in real-time by an installation as the *Il Caos delle Sfere* case study [19]. The refractions are represented by: *Trillo*[1], *Aeolian harp*[2], *Bordone*[3], *Canon*[4], *Swing (ventata)*, *Chords*[5] and *Harmonic accents* [6].

The algorithm and the source code/software analysed in this chapter are the ones used most likely in the 2012 and 2014's exhibition, the last ones prior to the proposed reactivation. The source code consists on two header files, `arpa.h` and `flipper.h` and a main *.cpp file, `arpa.cpp`. The first one contains the initialization of all variables present in `arpa.cpp` as well as all the functions' definitions. For example, here there are the names associated to the different sequences to be outputted at every new level as well as their characteristics (a flag indicating if a sequence is active, `SQNZON`, their length, `lenSQNZ`, an indicator about the levels the sequences are played at, `tipiSQNZ`, etc.).

```
// Gestione altre sequenze
char SQNZON;
const lenSQNZ=6000;
const numSQNZ=43;
int aSQNZ;
int tipiSQNZ[5][2]={{0,10},{10,8},{18,10},{28,10},{38,5}};
char *SQNZName[numSQNZ]={"campbag6.wri","campbag8.wri","catacc1.
  wri","catacc5.wri"," catacc6.wri","sofcamp1.wri","soff5.wri","
  trill.wri","trilcam4.wri","vent4.wri",

```

[1] A trillo is a rapid alternation of few played notes

[2] In this context, aeolian harp refers to the sounds generated by an aeolian harp instrument, a stringed instrument producing sounds when a current of air passes through it

[3] Bordone is an harmonic or monophonic effect where a note is continuously sounded throughout most or all of a performance

[4] A canon is a counterpoint-based compositional technique that plays a melody with one or more imitations of the melody played after a given duration

[5] A chord is an aggregate of pitches sounded simultaneously

[6] An harmonic accent is an emphasis or stress on a particular note or set of notes or chords

```
  "campla2.wri","flus1.wri","flus2.wri","schiso1.wri", "schiso15.
    wri","schiso9.wri","tril3.wri","trilca2.wri",

  "camplla1.wri","palc4.wri","rit10.wri","rit12.wri", "schiso14.wri"
    ,"schiso8.wri","soff3.wri","tril5.wri", "vent2.wri","vent4.wri",

  "rib4.wri","rit14.wri","rit20.wri","rit27.wri", "rit28.wri","rit31
    .wri","rit32.wri","rit4.wri", "rit9.wri","schi1.wri",

  "rit11.wri","rit13.wri","rit29.wri","schi5.wri","sofgra2.wri"};
...
```

The second header file, `flipper.h`, presents instead the initialization of variables regarding the addresses of the Parallel Port registers and the definition of two fundamental functions, `leggiPorta` and `scriviPorta`. In addition to that, proceprocessor statements are used in order to define the integers corresponding to each switch inside the pinball to track the evolution of the gameplay.

```
/* Definizione degli elementi interattivi */
#define BASE 0x378
void pascal leggiPorta(unsigned long,short,long,long,long);
void pascal scriviPorta(unsigned long,short,long,long,long);
int portaAttiva;
const cport=BASE+2;
const sport=BASE+1;
const dport=BASE;
int lastV;

#define blex 15
#define p1 25
#define p2 26
#define p3 27
#define p4 28
...
```

All the functions' implementations and the main part of the source code is contained on `arpa.cpp`. In order to include both header files, include guards are adopted in the first part of the code. As it can be seen in row 3 of the following piece of code, two other header files are used, `mshare.h` and `refract.h`. The first one refers to the *MidiShare* environment described in subsection 2.1.2.2: it includes all the library functions to be used in the custom-made functions for the installation. The second one is instead included to incorporate constants that are need to compile and generate the final software.

```
1 #include <stdlib.h>
2 #ifndef __MidiShare__
3 #include "mshare.h"
```

```
#endif

#include "flipper.h"
#include "refract.h"
#include "arpa.h"

// Flag principale per attivare il suono

// Flag per le rifrazioni
int fHarm,fBeat,fBordo;
// Gestione del tempo
unsigned long tmpTime,lastTime,elapsTime;
unsigned long pedTime;
bool normTime;
unsigned long pauseTime=0,playTime=10000;
int voci;
unsigned long rit[7];
unsigned long faCanone=0;
unsigned long smettiCanone=0;
int canoneOn=0,vai=1;

struct Seq
{ int nRow;
int nCol;
int* data;
};

...
```

The initial part of `arpa.cpp` suggests the structure behind the way the source code was designed. The developing team decided to mainly instantiate global variables rather than using local variables for each function or object created. Most of the variables are used by several types of objects so they need to be accessible from almost any place in the code. They could go with that strategy also because there was no memory constraint imposed by the PC used for the installation. Even if this results in a less efficient code, it simplifies most of the actions to be performed by the code functions. The second main feature of the code is a massive usage of object-oriented programming. Each type of sequence was implemented with a dedicated object, both structure (e.g `Seq` in the proposed snippet) and classes. The main features of each type of sequence is saved in the data members of the corresponding class while the actions (e.g. set up a sequence, load a sequence, send out MIDI events for that sequence) are implemented in form of function members. For example, in case of *Bordone* type of sequence, a class named `bordone` has been created. As all the other classes implemented, it contains a constructor which instantiates the Bordone's features as its type, its length and its melodic direction (here represented by the interv array), a set function to update its features and a perform function to send its MIDI events via MIDI port.

```

// -------
// Bordone
// -------
const NUM_BORDONI=6;
const prSB=9;
const plSB=4;
int pSB[prSB][plSB]={{12,5,11,X},{5,1,10,X},{7,5,8,X},
  {3,3,3,X},{9,0,10,X},{0,0,8,X},
  {6,5,5,X},{7,0,8,X},{7,0,6,X}};

class bordone
{  int row;
  int interv[NUM_BORDONI];
  waitSeq* rSeq;
  int memo[128];
  set(int);
  public:
  bordone(int);
  perform(short,MidiEvPtr);
};
```

3.1 The algorithm

The routine the algorithm follows can be summarized in the procedure depicted in figure 3.1. All the actions are executed within a loop between two functions, leggiPorta and scriviPorta. After a first initialization of all the objects and variables involved in the code, they are in charge of reading data from the acquisition board's output and for every new target hit by the ball they trigger a sequence playback. This depends on the current level of the algorithm and on the number of balls in game. Sequences can be either pre-written (and this is the case for the ones played as soon as a game starts) or automatically generated, starting from some pseudorandom parameters determined by the game and the algorithm.

As it can be seen in the following snippet, the function scriviPorta executes three operations: it reads from the Parallel Port, it writes over it and it uses the function MidiDTask to schedule a call to leggiPorta. It implements the first two steps of the procedure presented in subsection 2.2.2.3 to perform the acquisition of information about which in-game lights are turned on. The function inportb reads from sport (a variable containing the Status register address) an integer and it performs a bit-by-bit AND logical operation

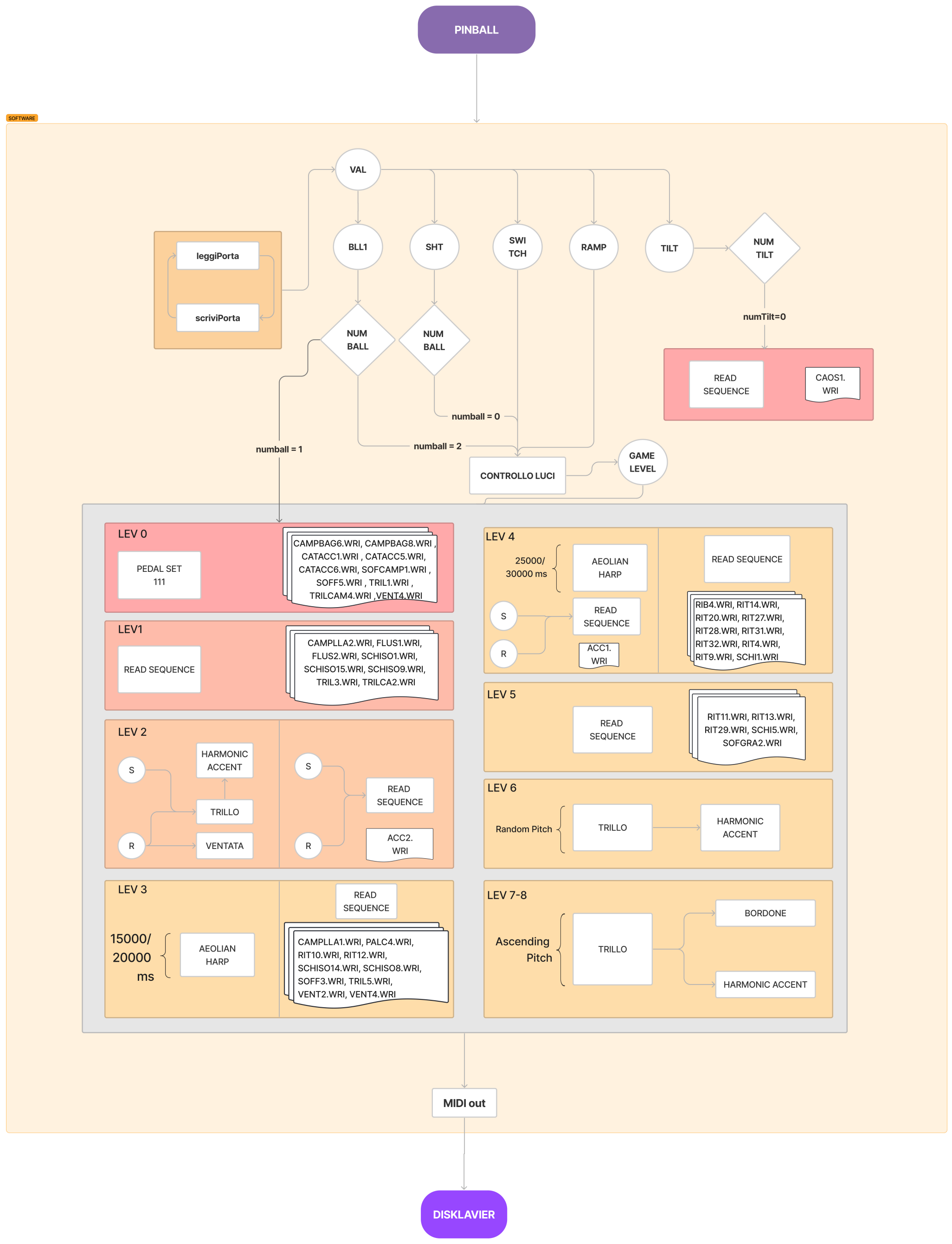


Figure 3.1: *The algorithm routine*

with $0xf0$ to only retain the four bits corresponding to the pins denoted as Y in the multiplexer representation of figure 2.9. The corresponding integer representing if F-I-L-M are on is stored on the variable `luci`. On the contrary, `outportb` writes over `cport` (a variable containing the Control register address) the hexadecimal number $0x21$ to to perform the switch in the multiplexer used in the acquisition board for reading data. Finally `MidiDTask` is called with arguments the function `leggiPorta`, `dt+5` and `ref`. As anticipated in subsections 2.1.2.2 and 2.2.3.2, `MidiDTask` is a *MidiShare*'s library function which can schedule function calls in a given instant of time over a given *MidiShare* instance. In this case it is used to call `leggiPorta` at the absolute time instant `dt+5`, where `dt` is a a variable initialized at the beginning of the routine with the exact time instant (in milliseconds) at which the software starts running. Therefore, 5 ms have to pass between the execution of the two functions. The final input parameter is `ref`, which is the reference number of the *MidiShare* instance controlling the scheduling.

```c
/* ----------------------------- */
/* Lettura dalla porta parallela */
/* ----------------------------- */
void pascal scriviPorta(unsigned long dt,short ref,long a1,long a2,
    long a3)
{  unsigned char azz;
  azz = (inportb(sport) & 0xf0);
  luci=(int) azz;
  outportb(cport, 0x21);
  MidiDTask(leggiPorta,dt+5,ref,0,0,0);
}
void pascal leggiPorta(unsigned long dt,short ref,long a1,long a2,
    long a3)
{  unsigned char azz;
  int val;

  luci=luci>>4;
  azz = (inportb(sport) & 0xf0);
  luci=luci|((int) azz);
  luci^=0x88;
  val=inportb(dport);

  ...
  outportb(cport, 0x20);
  ...

  if(portaAttiva)
  MidiDTask(scriviPorta,dt+300,ref,0,0,0);
  }
    }
  }
```

The function `leggiPorta` executes the remaining 3 steps in the acquisition process. It firstly performs a right-shift by 4-bit-positions of `luci` to be then merged with the second nibble (information about lights ”EX”, ”JK”, ”RE” and ”CS”) read by `inportb` as before. The merging is done via a logical OR bit-by-bit operation. In addition to that, a logical XOR with $0x88$ is required to invert the bit corresponding to pin 11 $\overline{S7}$ which is hardware-inverted. Another reading operation is then performed via `inportb` over `dport` (a variable containing the Data register address) to acquire the value corresponding to the last target hit by the ball which is stored in `val`. Then the multiplexing is switched via another `outportb` (with $0x20$) operation to prepare for the acquisition of the following first nibble in the next loop. If the Parallel Port input is still active, `MidiDTask` is called with input the function `scriviPorta` to restart the loop. For what concerns the time interval, in this case the delay is of 300 ms to let the pinball internal processor write the new light bit-values over the input of the multiplexer. Having such a large time horizon also gives the possibility for *MidiShare* to schedule several tasks in the meanwhile, especially the MIDI events transmission.

```
...
if(val==bll1 && lastVal==bll1)
  { ++countSW;
    if(countSW>20)
    { cambioLivello(0,ref,dt);
      numBall=0;
      countSW=0;
    }
  }

if(val!=lastVal)
  { switch(val)
    { case sw1: case sw2: case sw3: case sw4: case sw5: case sw6:
      case sw7: case sw8: case sn1: case sn2: case sn3: case sn4:
      case p1: case p2: case p3: case p4: case blex:
      SWITCH=1;
      countDown=0;
      STATUS2=0;
      if(numBall==0)
        numBall=1;
      if(gameLev==0)
        cambioLivello(1,ref,dt);
      break;
      case trp1: case trp2: case rpsi: case rpso: case rpsu:
      case rpdi: case rpdo: case rpci: case rpco:
      RAMP=1;
      countDown=0;
      STATUS2=0;
```

```
        if(numBall==0)
          numBall=1;
        if(gameLev==0)
          cambioLivello(1,ref,dt);
        break;
        case bll:
        if(STATUS1==0)
          STATUS1=1;
        break;
        case bll1:
          if(STATUS1==1)
          { STATUS1=0;
            if(numBall==2)
            cambioLivello(1,ref,dt);
            if(numBall>0)
            --numBall;
            if(numBall==0)
            countDown=dt;
            }
        break;
        case sht:
        countDown=0;
        if(STATUS2==0)
          { STATUS2=1;
            if(numBall<2)
              ++numBall;
            if(gameLev==0)
              cambioLivello(1,ref,dt);
          }
        break;
        case tilt:
          --numTilt;
          if(numTilt==0)
          { numTilt=10;
            faiTilt(ref,dt);
          }
        break;
        default:
        SWITCH=0;
        RAMP=0;
        break;
        }
        lastVal=val;
        }
...
```

Prior to scheduling `scriviPorta` again, `leggiPorta` performs a control about which is the last target hit by the ball in order to then progress with the algorithm level-check. The control is done if and only if the value read (`val`) is different from the one read in the previous iteration of the loop (`lastVal`). If the ball has hit a ramp or a different type of target (which the developers defined as *switch* with an abuse of notation) then some flags are activated to indicate that such an event happened and the algorithm level is changed to 1 if the game just started. In case the ball has exited the game table (value read corresponding to `bll`) a flag indicating that is activated as in the case of a switch or a ramp. On the contrary, the situation where the value read is `bll1` is more complex. It corresponds to the ball being on the so called *left trough*, which is tunnel where a ball passes through or is blocked before being sent to the shooting ramp. What discriminates different events in this case is the variable `numBall`, representing the number of balls currently active in game. If `numBall` is equal to 2 then this corresponds to a ball going out of play during a multiball event and therefore one ball still in play. The routine to follow is then to set the algorithm level down to 1 and then to decrease `numBall` by one unit. If instead `numBall` is equal to 1 then this corresponds to a ball exiting the pinball table and nothing is triggered. If `bll1` is read consecutively for at least 21 times, then the algorithm interprets such an event as the end of the game and reset its level down to 0. The case when `val` is equal to `sht` happens whenever the ball is on the shooting ramp. Apart from some flag activations, reading `sht` translates into an increase of `numBall` by 1 unit and setting the algorithm level to 1 if the game is about to start. The remaining case that could happen is when `val` coincides to `tilt`, i.e. the pinball has been hit by the player in a violent manner. The only action triggered here is at the point when the tilt event has happened for at least 10 times. Only at that point the algorithm invokes the function `faiTilt` which reproduces a MIDI sequence contained in the file `CAOS1.wri`.

```
...
if(gameLev>0)
{ controllaLuci(ref,dt);
  if(countDown>0 && (dt-countDown)>10000)
  { cambioLivello(0,ref,dt);
    numBall=0;
  }
}
...
}

void controllaLuci(short ref,unsigned long dt)
  {  char n;
    int m=1;

    // Controlla se le luci sono stabili
    n=luci^0xff;
    if(n==lastn)
    { ++countn;
      if(countn>7)
    {  countn=0;
      if(n & 16)  ++m;
      if(n & 32)  ++m;
      if(n & 64)  ++m;
      if(n & 128) ++m;
      if(m==1)
      { if(n & 1) m=6;
        if(n & 4) m=7;
        if(n & 2) m=8;
      }
      if(gameLev==6 && m==1)
      m=6;
      if(m!=gameLev)
      cambioLivello(m,ref,dt);
    }
    }
    else
      { countn=0;
        lastn=n;
      }
}
```

The final main action `scriviPorta` implements is the light check starting from the value written in `luci`. If the level is not 0 and the game has not ended, the function `controllaLuci` is called. The latter takes `luci` and checks if the value of the variable has been equal for 8 times consecutively using a counter `countn`. Only when `countn` reaches 8, i.e. the lights' signal has been stable for a sufficient amount of time, the algorithm level will be checked. This analysis is needed because during the gameplay evolution there are targets such that hitting them causes all the lights to blink. Such a blink may cause an erroneous and misleading reading of the lights so to differentiate this by a proper light read, counting the same light value several consecutive time helps to reach that goal. In particular, since each read action is performed roughly every 305 ms, it means that `luci` has to remain the same for nearly 2 s. This is a much larger duration that the blink duration which is about less than a second so the events can be discriminated. In order to determine the algorithm level, an auxiliary variable `m` is adopted. By doing logical AND bit-by-bit operations between`luci` and multiples of 2 and checking if the result is non-zero, `m` is incremented by one unit every time or set to 6, 7 or 8 whenever the first 4 checks fails. The latter situation means that F, I, L and M are off so the algorithm level is neither 1, 2, 3 ,4 or 5 but 6 or 7 or 8 which corresponds to EX, RE or JK/CS being on respectively. The final action is to change the algorithm level based on the one estimated (`m`) in `controllaLuci`: this is done by calling the function `cambioLivello` given (`m`) as input.

3.1.1 Evolution over the levels

The function `cambioLivello` is what defines the output of the algorithm, i.e. the MIDI events to be sent to the Disklavier, at every iteration of the loop. As it can be seen in the snippet of code, an action is performed only if the level of the algorithm is changing with respect to the current one determined previously stored in the variable `gameLev`.

```
void cambioLivello(int lev,short ref,unsigned long dt)
{
  if(lev!=gameLev)
  { ARPAON=0;
    ARPAON2=0;
    venON=0;
    venON2=0;
    SQNZON=0;
    TrilloON=0;
    gameLev=lev;
    MidiFlushDTasks(ref);
    ...
```

If the level has to change, the routine to follow consists firstly on resetting any flag associated to the status of what was generated or outputted in the previous level. For example `ARPAON`$=0$ indicates that any Aeolian Harp sequence played at level 3 has to be stopped in order to let events at the new level to happen. In addition to that the algorithm's level is updated with the current one. The main action of the function is however to invoke the `MidiFlushDTasks`, defined in the *MidiShare* environment. According to the library's documentation, it flushes all the waiting tasks, both function calling and MIDI reproduction, that were scheduled in the *MidiShare* instance (denoted by the address `ref`) with previous calls of `MidiDTask`. All the old level's actions to be performed simultaneously with the subsequent progression of the routine are eliminated in order to schedule activities related to the new level.

The main feature of `cambioLivello` is the activation of flags, variable setting and function calling and scheduling associated to each new level via a swith-case statement.

Level 0

```
case 0:
  numBall=0;
  aSQNZ=0;
```

```
ped->set(ref,dt,111);
loadSQNZ();
sqnzPosSQNZ=0;
timeLev0=dt+30000;
break;
```

Case 0 is the algorithm level set by the routine before the game starts which is however immediately changed as soon as the ball hit a target (a ramp or all the other types) as can be seen in the `scriviPorta`'s snippet of code. The action considered here are the setting of the variable `numBall` to 0 to establish that no ball has been still thrown into the game yet. In addition to that, the same is executed for the variable `aSQNZ`. It refers to an integer which indicates which set of sequences out of the ones stored in the `SQNZName` array has to be considered for the MIDI events at each level, in this case the level 0. This is the reason why `aSQNZ` is given the value 0. The function `loadSQNZ` is then in charge of randomly picking one sequence out of `campbag6.wri`, `campbag8.wri`, `catacc1.wri`, `catacc5.wri`, `catacc6.wri`, `sofcamp1.wri`, `soff5.wri`, `tril1.wri`, `trilcam4.wri` and `vent4.wri` files. In particular it loads the MIDI events in a set of arrays, one for each MIDI message field, that are initialized at the beginning of the routine and are re-written for each sequence to upload. Apart from `sqnzPosSQNZ` which is an integer to save the reading position over the sequence and `timeLev0` that is a variable to store a limit of time over which the game can remain in level 0, the main action executed at level 0 is a pedal set. The function calls the `set` function member of the `pedal` object. In musical terminology, a pedal is a long-lasting note (or group of notes), almost always in the low register. The term comes from the pedal board, a device contained within pianos to reach that effect. The `pedal` object is instantiated exactly to guide and set the main parameters for the Disklavier pedal board. The function `set` takes as argument 111, which is the integer that will be the second Data Byte of a MIDI Control Change message, i.e. it indicates how much a MIDI controller is to be pressed down. As the name of the function suggests, the MIDI controller of interest is the Disklavier pedal board. Therefore, the main role of this calling is to send to the MIDI port a MIDI Control Change message to press down (or to move up in other cases) the pedal. In particular, the Control Change message is made by two Data Bytes: the first one with value 64 to indicate the pedal MIDI controller and the second one with value 111 to specify the intensity of the pressing.

By analysing the loop between `scriviPorta` and `leggiPorta`, it can be noticed that the sequence uploaded at level 0 is almost never sent out, this is because it would require the algorithm level to stay at 0 for a sufficiently long amount of time. However, such an event cannot happens because even if the ball remains on the shooting ramp and is not thrown, the level is always set to

1.

Level 1

```
case 1:
  numBall=1;
  aSQNZ=1;
  SQNZON=1;
  loadSQNZ();
  sqnzPosSQNZ=0;
  MidiDTask(playSQNZ,dt+100,ref,0,0,0);
  break;
```

After the ball has been shot for the first time and the game has just started, whenever the ball hits one switch, the algorithm level is switched to level 1. Apart from setting the variable `numBall` to 1 to indicate the game is started, the main result of this level update is the playback of a random sequence out of the level 1 set of them. In order to do so, firstly `aSQNZ` is updated with 1 and then the sequence is picked and its event are loaded with `loadSQNZ`. Secondly, after having reset the reading sequence index (`sqnzPosSQNZ`= 0), the function `playSQNZ` is scheduled via `MidiDTask` after 100 ms since the `cambioLivello` call (occurred at the absolute time instant `dt`). `playSQNZ` is a function developed with the purpose of sending MIDI events to the MIDI port to be then played at the Disklavier. In addition to that, if the level is not changing and a sequence has been read completely, `playSQNZ` calls again `loadSQNZ` to then play another level 1 sequence. The set of potential sequences consists of `campl1a2.wri`, `flus1.wri`, `flus2.wri`, `schiso1.wri`, `schiso15.wri`, `schiso9.wri`, `tril3.wri` and `trilca2.wri` files.

Level 2

```
case 2:
  numBall=1;
  aACC2=0;
  TRI2ON=1;
  numTotTri=15;
  tri->set(0);
  break;
```

In case the level is 2 (which corresponds to have just one of the letters F I L M on) the routine again puts 1 in `numBall` to indicate that still the gameplay has not entered multiball mode. On top of that, the main action is the activation of the flag `TRI2ON`, indicating that trillos sequences randomly generated in real-time will be outputted. The presence of 2 in the variable name suggests that these trillo's MIDI events will be specific of level 2, to distinguish them from trillo's MIDI events characterizing other levels. The command `numTotTri=` 15 sets the maximum number of trillo's sequences to be played to 15 while `tri− >set(0)` initializes the characteristics of the trillo events (established on the trillo object), following the configuration number 0. In particular, in this level a trillo sequence is generated per each switch or ramp hit respectively. This can be seen in the following snippet of code which is taken from `leggiPorta`.

```
...
if(gameLev==2)
  {   if(SWITCH || RAMP)
    {   SWITCH=0;
      RAMP=0;
      if(TRI2ON)
      {  if(--numTotTri>0)
      {   numTri2=10+random(20);
      MidiDTask(playTrillo,dt,ref,0,0,0);
  }
  else
  { TRI2ON=0;
  }
  }
  else
  { if(ACC2ON==0)
    ACC2ON=1;
    initACC2();
    MidiDTask(playACC2,dt,ref,0,0,0);
  }
```

After each ramp or "switch" is hit, `numTotTri` is decreased by one. `numTri2` represents instead how many repetitions characterize each sequence and such number can vary from 10 to 30. Finally `playTrillo` is scheduled and immediately executed (no delay is set since its time reference is `dt`, the same one as the `leggiPorta` call). It sends trillo's MIDI events in output together with harmonic accents events, generated with the function member `perform` of the object `accent`. In addition to trillo's sequences, `playTrillo` generates swings MIDI events (invoking the `playVentate` function) if a ramp is hit by the ball. Once `numTotTri` reaches 0, for each switch or ramp a part of a sequence of chords is outputted. This is obtain by setting `ACC2ON` to 0. It is a variable suggesting that parts of a sequence of chords specific for level 2 is outputted. Such sequence is loaded via `initACC2()` from the file `ACC2.wri` and then outputted thanks to `playACC2`.

Level 3

```
case 3:
  numBall=1;
  aSQNZ=2;
  loadSQNZ();
  ARPAON=1;
  venON=1;
  arpaFine=dt+15000+50*random(100);
  MidiDTask(playArpa,dt+100,ref,0,0,0);
break;
```

For level 3 (corresponding to two letters on) the main action is to output an Aeolian harp sequence of MIDI events randomly generated in real time. Such a task is activated by setting the flag `ARPAON` to 1 and implemented by the function `playArpa`, which is delayed by 100 ms. Harp MIDI events are equipped with six-voice canon events (generated with the `playCanone` routine inside `playArpa`), swings, bordone (generated with the `perform` function member of a bordone object named `bordoV`) and harmonic accents. The overall total duration of the Harp sequence is set randomly whenever the level changes and its duration varies from 15 to 20 s as the `arpaFine` variable suggests. After its end, a new pre-written sequence associated with level 3 is uploaded and played if the level stays the same. The sequence is loaded with `loadSQNZ` and chosen via `aSQNZ`= 2 among `campl1a1.wri`, `palc4.wri`, `rit10.wri`, `rit12.wri`, `schiso14.wri`, `schiso8.wri`, `soff3.wri`, `tril5.wri`, `vent2.wri` and `vent4.wri` files.

Level 4

```
case 4:
  numBall=1;
  aSQNZ=3;
  loadSQNZ();
  ARPAON=1;
  ARPAON2=1;
  venON=1;
  arpaFine=dt+25000+50*random(100);
  MidiDTask(playArpa,dt+100,ref,0,0,0);
break;
```

If the level is equal to 4 (corresponding to three letters on) the same routine as level 3 is performed with some differences. First of all, the overall harp duration varies from 25 to 30 s as the `arpaFine` variable indicates. On contrary with respect to the previous level, harp MIDI events are associated to parts of a sequence of chords specific for level 4, played for each switch or ramp hit. Such sequence is loaded via `initACC1()` from the file `ACC1.wri` and then outputted thanks to `playACC1`. The sequences considered in level 4 are loaded with `loadSQNZ` and chosen via `aSQNZ`= 3 among `rib4.wri`, `rit14.wri`, `rit20.wri`, `rit27.wri`, `rit28.wri`, `rit31.wri`, `rit32.wri`, `rit4.wri`, `rit9.wri` and `schi1.wri` files.

Level 5

```c
case 5:
  numBall=1;
  aSQNZ=4;
  loadSQNZ();
  MidiDTask(playSQNZ,dt+100,ref,0,0,0);
  break;
```

Whenever all 4 letters of the FILM set are o, the algorithm enters level 5. The only action the routine executes is to upload one or more sequences specific for this level. As before, `loadSQNZ` is called for this purpose after having set `aSQNZ`$=$ 4. The playback is performed by the function `playSQNZ` which chooses among `rit11.wri`,`rit13.wri`,`rit29.wri`,`schi5.wri` and `sofgra2.wri` files.

Level 6

```c
case 6:
  numBall=2;
  TrilloON=1;
  tri->set(0);
  MidiDTask(playTrillo,dt+100,ref,0,0,0);
  break;
```

When the EX (extra ball) event occurs then the algorithm enters level 6. As it can be seen from the code snippet, this translates into setting `numBall`$=$ 2. At this level the routine generates a continuous sequence of trills, differently from level 2 in which trills correspond to a switch or a ramp being hit by the ball. An additional feature of level 6 trillos is that their centroid is moved around notes Sol#6 and Sol#7. In order to distinguish level 6 trillo's activation from the one at level 2, another flag, named `TrilloON`, is used and set to 1. The trillo MIDI events generated are then outputted by scheduling `playTrillo` after 100 ms. If either one of the two ball exits the game table, the level is immediately reset to 1.

Level 7 and Level 8

```c
case 7: case 8:
  numBall=2;
  TrilloON=1;
  tri->set(1);
  MidiDTask(playTrillo,dt+100,ref,0,0,0);
  break;
```

Even if level 7 is entered when the light RE is on and level 8 is entered when either one of CS or JK is activated, the algorithm follows the same routine for both cases. It consists of trill sequences that are continuously sent in output, with variations in a upward shift of the centroid with semitone or tone intervals for each target hit. In addition to that, for these levels there is also the possibility to have a double transposition for every trill together with the bordone (in the configurations upper-lower-upper + lower given that some pseudo-random conditions are verified) or no drone at all. Similarly to level 6, `numBall` is given the value 2 because when RE, CS or JK are activated the game must be in multiball mode and the flag `TrilloON` is set to true. The main difference with respect to the previous level is that the member function `set` of the object `trillo` is called with parameter 1 instead of 0. This setting allows the `playTrillo` function to possibly add bordone (generated with the `perform` function member of either the bordone object of type `bordoV` and `bordoL`)

3.2 Transmission of MIDI messages

```
void VidiSend(short ref,MidiEvPtr e)
{ float v;

  if(EvType(e)==typeKeyOn)
  { v=Vel(e)/1.6;
    Vel(e)=(int) v;
    if((numBall>0 && STATUS2==0) || (gameLev==0 && SQNZON))
      MidiSend(ref,MidiCopyEv(e));
    }
    else
    {
    if((numBall>0 && STATUS2==0) || (gameLev==0 && SQNZON))
      MidiSend(ref,MidiCopyEv(e));
  }
}
```

Every developed function to send MIDI messages to the MIDI port starting from MIDI events contained in the pre-written sequences or the ones generated in real time rely on `VidiSend`. Apart from the reference to the *MidiShare* instance, it takes as input a variable of type `MidiEvPtr`, a pointer to a *MidiShare* event. A *MidiShare* event is a container defined in the *MidiShare* environment to wrap a MIDI event object. In particular, it is composed by several data members corresponding to the characteristic of a generic MIDI event, for example the type of message yield, the date, the channel, the pitch, etc. The main goal of `VidiSend` is to check whether the input MIDI event passed by pointer is of type Note ON. If this condition is verified, it scales the velocity of the event by a factor 1.6. The motivation behind this action is to reduce how fast a key is pressed on the Disklavier in order to not overload the internal mechanics of the instrument. In any case, `VidiSend` calls a *MidiShare* library function named `MidiSend`, which sends an event to the MIDI port via the *MidiShare* instance address.

Chapter 4

Preservation of the human-machine interaction: Technological migration

Starting from the multilevel preservation approach that was presented in part I, a technological migration for the *Il Caos delle Sfere* artwork has been completed recently by Alessandro Fiordelmondo, Mattia Pizzato and the author of this thesis at the CSC laboratories. *Technology migration* is part of the "adapted/updated technology approach" of conservation [26], i.e. it refers to the reactivation of an artwork with new modern technologies. The objective of the migration was mainly to preserve the identity of the artwork, especially on the human-machine interaction, rather than a simpler preservation and reproduction of the sound performance for the artwork. This was also motivated by the fact that, as described in the previous chapters, it was not possible to identify the exact source code that was used in the different exhibitions. In addition to that, the technological migration was needed because of the obsolescence starting to affect the old original computer used for the installation. Apart late exhibitions held in 2012 and 2014, *Il Caos delle Sfere* has never been consistently presented to the public since the 2004. The original PC has also never been updated or even used over the years at the CSC laboratories due to fast advent of better technology. This made the PC obsolete and very unreliable to be presented in a public exhibitions. As a consequence a technological migration of it had to be taken into consideration.

4.1 *Il Caos delle Sfere* DPO records

The preservation and reactivation processes took place between March 2022 and September 2022. The first step was to analyse every document or element that could help into define a possible strategy for the reactivation, i.e. to

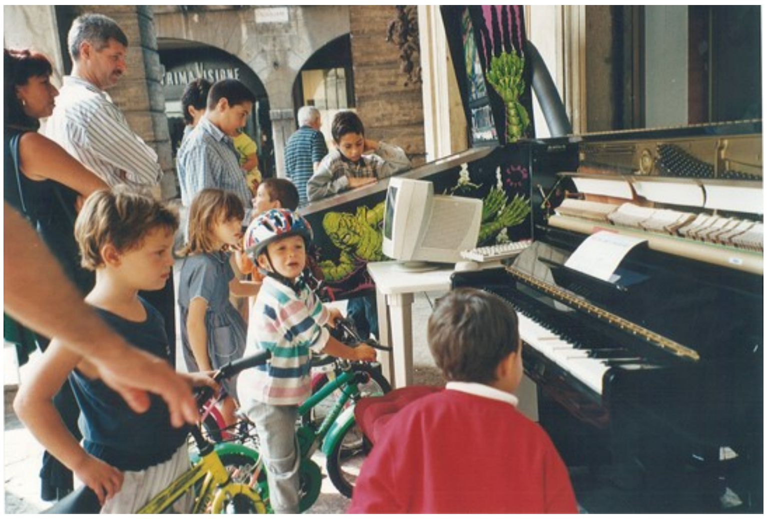


Figure 4.1: *Il Caos delle Sfere during the Rassegna Finestre sul Novecento at the Civic museum in Treviso.*

assemble the DPOs of the first performances by collecting bits, data and experience documents. For what concern the first type of item (bit), the original performative system was still available. Starting from the *Biennal of the Young Artists of Europe and Mediterraneo* in 1999, the exhibition toured in other artistic manifestations until year 2004, including at the *Rassegna Macchine musicali* (1999) in Zagarolo, at the *Rassegna Finestre sul Novecento* at the Civic museum in Treviso (figure 4.1) (1999), in Rome, Milan, Venice, Monselice and Bolzano. The final exhibitions prior to the reactivation were *Visioni del Suono. Musica elettronica all'Università di Padova* in Padua (2012) and *Tilt — Disklavier: calcolare il pianoforte* in Turin (2014). Almost all components, both hardware and software, remained the same in all the exhibitions the artwork went through, with modifications only on the source code, as discussed in chapter 3, and the pre-written sequences stored in *.wri files. The original technological tools involved as the acquisition board, the computer and the pinball have been stored in the CSC laboratories over the years while the Disklaviers, at least the ones considered in 2012 and 2014 exhibitions, have been used in the *Conservatorio Pollini* in Padua for didactic usage. In addition to that, the *Creature from the Black Lagoon* pinball has undergone a process of physical restoration. Conservative copies of all the software, the music sequences to be played at different levels and an overall copy of the hard disk were all generated. In this case, the DPO's set of bits is almost complete. On the contrary, the collection of both the data and experience documents was more difficult to execute. Both the algorithm and the source code were developed in direct collaboration with the composer Carlo De Pirro with a *trial-and-error* approach for every exhibitions, however all the versions of them have been left mostly undocumented. No digitalization of the electric scheme of the acquisition board was found even if it was carried out over the years [8]. The same goes with the notes of the composer about the installation and the melodic sequences it composed for it. All these documents are considered DPO's data items but the set of them is totally incomplete. For what concerns the experience documentation, several photos of the exhibitions that report both the original appearance, the actions that occurred during the performance, the tools placement and the use of technologies have been collected by the *Carlo De Pirro association* and they are available online[1]. No video or audio recording were found apart from the very last exhibition of 2014. On the other side, in the set of DPO's experience items the software testing the pinball-PC communication has to be inserted as well as several scientific publications describing both the artwork and the multilevel preservation approach as [8] and [19].

The process of reactivation was conducted by the authors without the artist, that unfortunately died in 2008. Therefore every choice about what direction the reactivation had to take was decided by the research group. In the first step of the reactivation, the performative system's rehabilitation has been questioned, fielding the issues of the technological migration and recovery of analogue tools. In particular, the recovery of original technologies was possible, as described previously. The main part of the entire performative system, the algorithm and the source code for it, were the part that underwent a migration approach. In particular, they have been rewritten, modified and migrated from the original PC running Windows 95 and *MidiShare* to an *Arduino Mega* board. This means that, given the high level conceptual description of the three nodes presented in chapter 2, the *Communication* node is then characterized by a complete change. The decision to perform this migration was because of the aforementioned unreliability of the original PC while the Arduino board was selected among a much larger set of potential boards for reasons that will be described in section 4.2.1. After several tests, it resulted to be the best one in terms of meeting the technological requirements for the installation. The remaining hardware characterizing the original performative system was not altered: the pinball, the acquisition board and the Disklavier piano were not considered in the migration. This implies that the *Interaction* node and the *Playback* node persist in the proposed reactivation. The motivation for this is the complexity that a potential improvement in technology for those components would require. However, the next step in the reactivation

[1] *Associazione Carlo De Pirro*, Il Caos delle Sfere, `http://csc.dei.unipd.it/depirro/opere/composizioni/caos-delle-sfere` (accessed 1 November 2022)

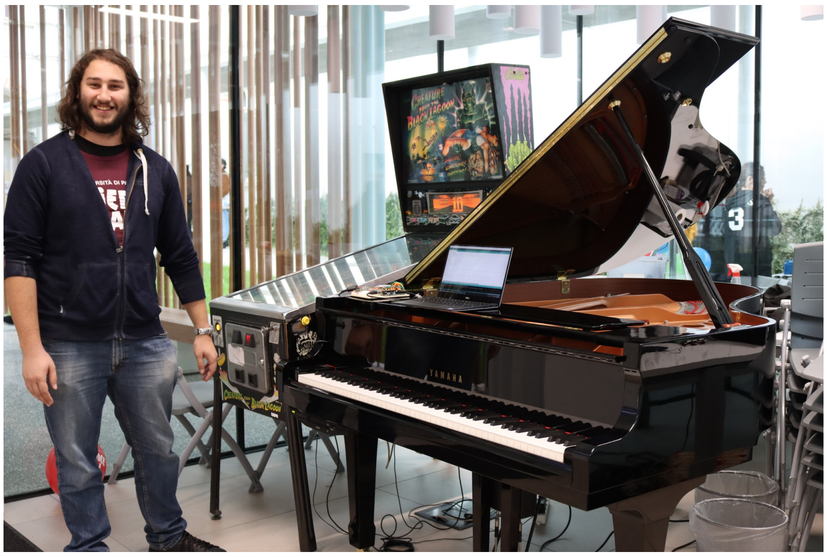


Figure 4.2: *Il Caos delle Sfere installation and Mattia Pizzato during the Science4All scientific dissemination.*

of the *Il Caos delle Sfere* necessarily needs to consider a migration especially for the acquisition board, as will be analysed in later sections. The algorithm and the software have been reactivated through the development of an Arduino sketch written in Arduino programming language. The Arduino Mega and the updated version of the algorithm and the source code, together with the remaining old hardware and software items represent the bits of a new DPO record. With them, there is also the collection of data (relationship between hardware and software and documentation of the new algorithm, the source code and the switches mapping) and experience (audio recordings of performance in the digital domain, new photos and articles) documents. In the case of bit-type items, it was possible to apply the multiple belongingness property (described in section 1.3) to the pinball-acquisition board system because they were not involved in any migration.

The reactivation of *Il Caos delle Sfere* has been exhibited during *Science4All*, a scientific dissemination festival of the University of Padua held on the 30th of September (figure 4.2). The new DPO record for the Science4All' exhibition together with the DPO record involving the first exhibition, the one in Zagarolo (1999), the one in Padua (2012) and the one in Turin (2014) are depicted in figure 4.3. The chronological representation helps to show how the main part of the hardware (pinball and the acquisition board) have remained unchanged from 1999 to the present proposed reactivation. For those DPOs bit-items the multiple belongingness property holds. On the contrary, changes have been made for the PC which has been replaced in the final reactivation with an Arduino Mega board, a Parallel Port and a MIDI port. For what concerns the WRI files containing the pre-written MIDI sequences, different versions of them were generated with *Finale*. However, starting from the Zagarolo exhibitions, they were not changed: this is underlined with v1 in the proposed scheme to distinguish them from their original versions made for the first installation (and some subsequent ones), named v0. This also the reason why in the scheme, only the DPO record for Zagarolo is depicted. Almost certainly several other sequences were adopted in previous exhibitions but the ones in use in 2012 and 2014 were exactly developed for that festival. For the *Science4all* dissemination, they have been stored instead in three *.txt files saved in a MicroSD card connected to the Arduino board. Another item which has changed over the DPOs is the Disklavier: according to the availability, a different type of Disklavier was adopted for the artwork. No indication of the specific type was found therefore in the scheme only the word Disklavier is present. Finally, the items associated to the algorithm and the source code are what has been changed for the largest part over the exhibitions. While it's possible to determine that more than 10 versions of the same source code were developed, it's not possible to associate each version to a specific exhibition. That's why in the scheme in figure 4.3, items for algorithm and source code are named with a generic suffix v0, v1 and v2. Only for the last two artwork exhibitions, it's possible to determine that the same version (v2) was taken in consideration. In the reactivation, an Arduino IDE sketch running the modified and updated algorithm is considered. Due to the lack of documentation, DPOs data item set has been instead defined from scratch only for the *Science4all* DPO record. It includes both hardware and software (both algorithm and sketch) documentation, the setup's description, i.e. how to interconnect all the components for an exhibition and an updated version of the mapping between switches, lights and the corresponding byte value that can be read via the Parallel Port. DPO's experience-items instead contain several photos of the exhibitions until 2012 taken from the archive of the *Carlo De Pirro association*. For the 2014 festival, instead, videos and journal articles have been found over the internet. Finally, for *Science4All*'s dissemination, videos, photos and a scientific article about the reactivation (still to be completed) are considered.

4.1.1 DPO records organization

All the DPO for all the items are contained in a GitLab repository (figure 4.4), in which DPOs for the 1999, 2012, 2014 and 2022 exhibitions can be found inside the "DPO" folder. "BIT", "DATA" and "EXPERIENCE" folders are character-

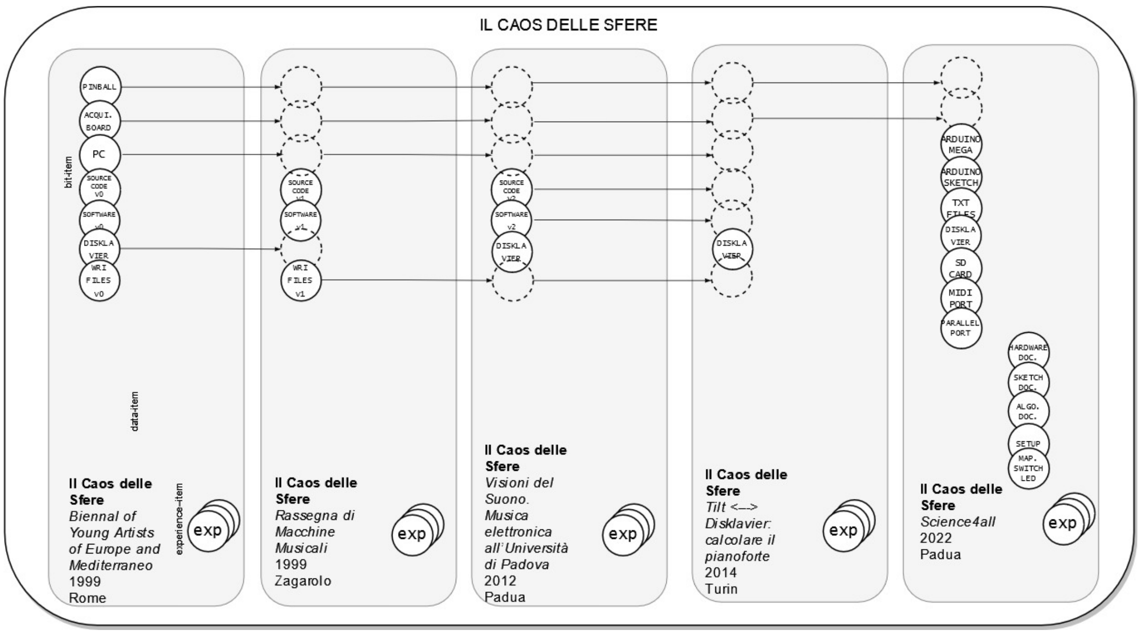
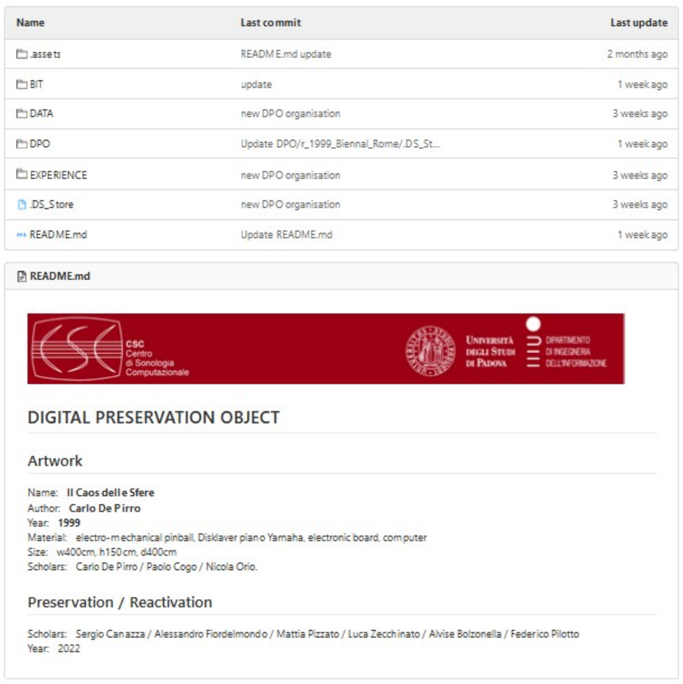


Figure 4.3: *Approximate chronological representation of Il Caos delle Sfere through the multilevel preservation model.*

ized by all items of corresponding type for both records. The creation of DPOs is a process still being in development. No prior indication about how Digital Preservation Obejcts should be created from the engineering point of view has been set. It needs to be digital but no specific format was developed. Over the reactivation of artwork, the decision the research group decided to take was to create DPOs and the corresponding items digitally in the *Dublin core* format, a metadata *schema* designed to enable description of any resource [21] together with any type of file related to the resource. *Resource* is defined as "potentially informative object", something about which a statement is being made. The statement contains the *description* of the resource, what needs to be specified about an object in a subjective manner, i.e. the nature of the object. A statement about a potentially informative object is called *metadata*. In the context of the preservation model, the resource refers to any item composing a DPO record and the DPO record itself. A metadata *schema* can be referred as a set of rules about what sorts of statements (composed by *subject-predicate-object*) can be made about a resource [21]. For a metadata schema, any predicate is called *element*. An element in a metadata schema is a category of statement that can be made about a resource; an element names an attribute of a resource. The data that is then assigned to an element is named *value*. Together, an element-value pair constitutes the totality of a single statement about a resource. If metadata



Name	Last commit	Last update
.assets	READM E.md update	2 months ago
BIT	update	1 week ago
DATA	new DPO organisation	3 weeks ago
DPO	Update DPO/r_1999_Biennal_Rome/.DS_St...	1 week ago
EXPERIENCE	new DPO organisation	3 weeks ago
.DS_Store	new DPO organisation	3 weeks ago
README.md	Update README.md	1 week ago

README.md

CSC Centro di Sonologia Computazionale — Università degli Studi di Padova — Dipartimento di Ingegneria dell'Informazione

DIGITAL PRESERVATION OBJECT

Artwork

Name: **Il Caos delle Sfere**
Author: **Carlo De Pirro**
Year: **1999**
Material: electro-mechanical pinball, Disklaver piano Yamaha, electronic board, computer
Size: w400cm, h150cm, d400cm
Scholars: Carlo De Pirro / Paolo Cogo / Nicola Orio.

Preservation / Reactivation

Scholars: Sergio Canazza / Alessandro Fiordelmondo / Mattia Pizzato / Luca Zecchinato / Alvise Bolzonella / Federico Pilotto
Year: 2022

Figure 4.4: *Organization for Il Caos delle Sfere DPO's items.*

is statements about a potentially informative object, the element-value pair is the irreducible unity of metadata [21]. The *Dublin Core* was developed in 1996 as a metadata *schema* with the intention of making it widely adopted and ubiquitous for eveety resource on the web [21]. The authors wanted to develop a *core* set of descriptive metadata elements that could be applied to any and all resources on the Internet. In the end 15 elements emerged in the core [21]:

- **Contributor**: it refers to an entity responsible for making contributions to the resource;
- **Coverage**: it refers to the spatial or temporal topic of the resource, the spatial applicability of the resource, or the jurisdiction under which the resource is relevant;
- **Creator**: it refers to an entity primarily responsible for making the resource;
- **Date**: it refers to a point or period of time associated with an event in the life cycle of the resource:
- **Description**: it refers to an account of the resource;

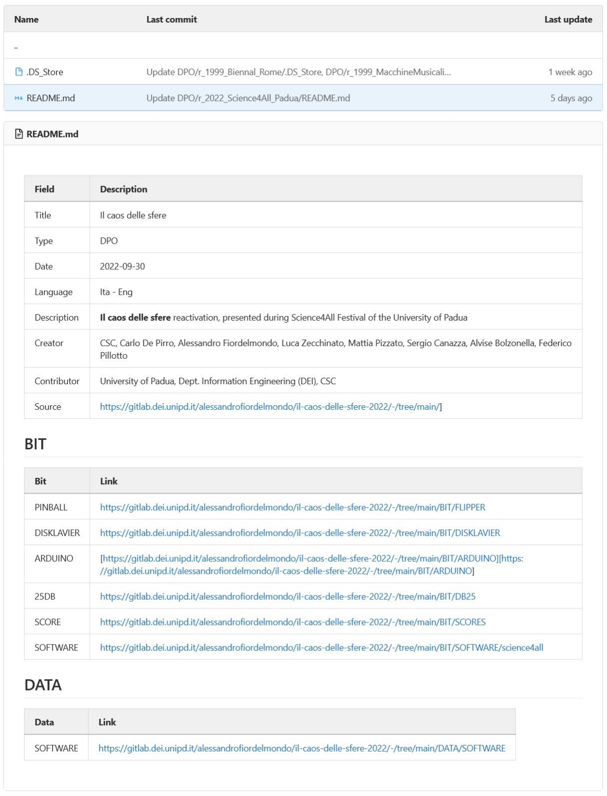
- **Format**: it refers to the file format, physical medium, or dimensions of the resource;
- **Identifier**: it refers to an unambiguous reference to the resource within a given context;
- **Language**: it refers to a language of the resource;
- **Publisher**: it refers to an entity responsible for making the resource available;
- **Relation**: it refers to a related resource;
- **Rights**: it refers to a piece of information about rights held in and over the resource;
- **Source**: it refers to a related resource from which the described resource is derived;
- **Subject**: it refers to the topic of the resource;
- **Title**: it refers to a name given to the resource;
- **Type**: it refers to the nature or genre of the resource.

A modified scheme consisting only a part of the elements of the *Dublin Core* model is adopted to create each DPO record and the corresponding items respectively. For example figure 4.5a represents the metadata description for *Science4all* DPO, consisting on Title, Type, Date, Language, Description, Creator, Contributor and Source elements and the corresponding data.

The figure also shows two other tables named DATA and BIT where link to the items composing the DPO are contained. By clicking on the links one can access the metadata *schema* for the items. For example, figure 4.5b contains all the statement description for the item associated with the *Creature from the Black Lagoon* pinball machine and a PDF file with its data-sheet. The way of implementing a DPO record proposed is still under study and several variations will be considered, especially in the set of elements composing the metadata scheme description

4.2 Reactivation of the *Il Caos delle Sfere*

The following sections describe all the main components of the reactivation and the technological migration together with a deep insight on the algorithm



Name	Last commit	Last update
..		
.DS_Store	Update DPO/r_1999_Biennal_Rome/.DS_Store, DPO/r_1999_MacchineMusicali...	1 week ago
README.md	Update DPO/r_2022_Science4All_Padua/README.md	5 days ago

README.md

Field	Description
Title	Il caos delle sfere
Type	DPO
Date	2022-09-30
Language	Ita - Eng
Description	**Il caos delle sfere** reactivation, presented during Science4All Festival of the University of Padua
Creator	CSC, Carlo De Pirro, Alessandro Fiordelmondo, Luca Zecchinato, Mattia Pizzato, Sergio Canazza, Alvise Bolzonella, Federico Pillotto
Contributor	University of Padua, Dept. Information Engineering (DEI), CSC
Source	https://gitlab.dei.unipd.it/alessandrofiordelmondo/il-caos-delle-sfere-2022/-/tree/main/]

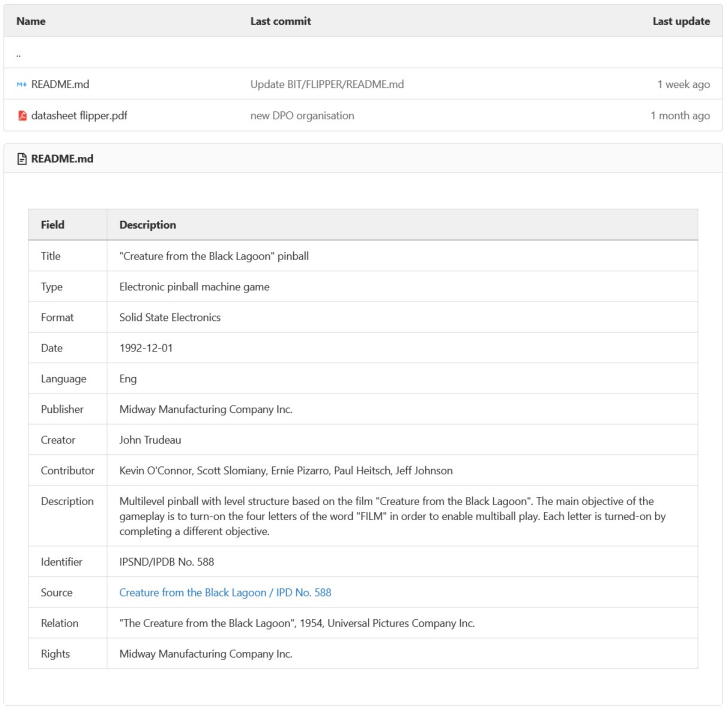
BIT

Bit	Link
PINBALL	https://gitlab.dei.unipd.it/alessandrofiordelmondo/il-caos-delle-sfere-2022/-/tree/main/BIT/FLIPPER
DISKLAVIER	https://gitlab.dei.unipd.it/alessandrofiordelmondo/il-caos-delle-sfere-2022/-/tree/main/BIT/DISKLAVIER
ARDUINO	[https://gitlab.dei.unipd.it/alessandrofiordelmondo/il-caos-delle-sfere-2022/-/tree/main/BIT/ARDUINO][https://gitlab.dei.unipd.it/alessandrofiordelmondo/il-caos-delle-sfere-2022/-/tree/main/BIT/ARDUINO]
25DB	https://gitlab.dei.unipd.it/alessandrofiordelmondo/il-caos-delle-sfere-2022/-/tree/main/BIT/DB25
SCORE	https://gitlab.dei.unipd.it/alessandrofiordelmondo/il-caos-delle-sfere-2022/-/tree/main/BIT/SCORES
SOFTWARE	https://gitlab.dei.unipd.it/alessandrofiordelmondo/il-caos-delle-sfere-2022/-/tree/main/BIT/SOFTWARE/science4all

DATA

Data	Link
SOFTWARE	https://gitlab.dei.unipd.it/alessandrofiordelmondo/il-caos-delle-sfere-2022/-/tree/main/DATA/SOFTWARE

(a)



Name	Last commit	Last update
..		
README.md	Update BIT/FLIPPER/README.md	1 week ago
datasheet flipper.pdf	new DPO organisation	1 month ago

README.md

Field	Description
Title	"Creature from the Black Lagoon" pinball
Type	Electronic pinball machine game
Format	Solid State Electronics
Date	1992-12-01
Language	Eng
Publisher	Midway Manufacturing Company Inc.
Creator	John Trudeau
Contributor	Kevin O'Connor, Scott Slomiany, Ernie Pizarro, Paul Heitsch, Jeff Johnson
Description	Multilevel pinball with level structure based on the film "Creature from the Black Lagoon". The main objective of the gameplay is to turn-on the four letters of the word "FILM" in order to enable multiball play. Each letter is turned-on by completing a different objective.
Identifier	IPSND/IPDB No. 588
Source	Creature from the Black Lagoon / IPD No. 588
Relation	"The Creature from the Black Lagoon", 1954, Universal Pictures Company Inc.
Rights	Midway Manufacturing Company Inc.

(b)

Figure 4.5: *Modified Dublin Core scheme examples .*

and the source code developed for the *Science4All* dissemination. In addition to that, a presentation on the research for future installation of the artwork will be presented.

4.2.1 The Arduino Mega board

The core of the reactivation is the *Arduino* board, in the *Mega* version. *Arduino* is an open-source electronics platform developed since 2005 by the Arduino company that designs and manufactures single-board microcontrollers and microcontroller kits for building digital devices [4]. Arduino boards are able to read inputs via sets of digital and analog input/output (I/O) pins that may be interfaced to various expansion boards or breadboards. They also provide serial communication including USB on some models, which is also used for loading programs, i.e. sending a set of instructions to the microcontroller on the board. The microcontrollers can be programmed using the Arduino programming language (based on Wiring) using the Arduino Software (IDE), inspired by the *Processing* language and used with a modified version of the Processing IDE. The Arduino project provides an integrated development environment (IDE) and a command line tool developed in the programming language *Go*.

The Arduino boards are characterized by offering a simplification in the process of working with microcontrollers mainly for didactic usage. In fact they are inexpensive compared to other microcontroller platforms and they are equipped with a simple programming environment, easy-to-use for beginners, yet flexible enough for advanced users to take advantage of as well. In addition to that Arduino boards are provided with open source and extensible software, available for extension by experienced programmers: the language can be expanded through C++ libraries. Finally the hardware is open source and extensible too: the plans of the Arduino boards are published under a Creative Commons license so experienced circuit designers can make their own version of the module, extending it and improving it [4].

For the reactivation of the installation, the *Arduino Mega 2560* board (figure 4.6) is considered. It was developed for applications that require large number of input and output pins, which is the case for the proposed technological migration, and the use cases which need high processing power. The Arduino Mega accommodates the ATmega2560 microcontroller, which operates at a frequency of 16 MHz. The board also contains 54 digital input/output pins, 16 analog inputs, 4 UARTs (hardware serial ports), a USB connection, a power jack, an ICSP header and a reset button [4]. From the computational point of view, the primary processor of Arduino Mega 2560 board is the ATmega2560

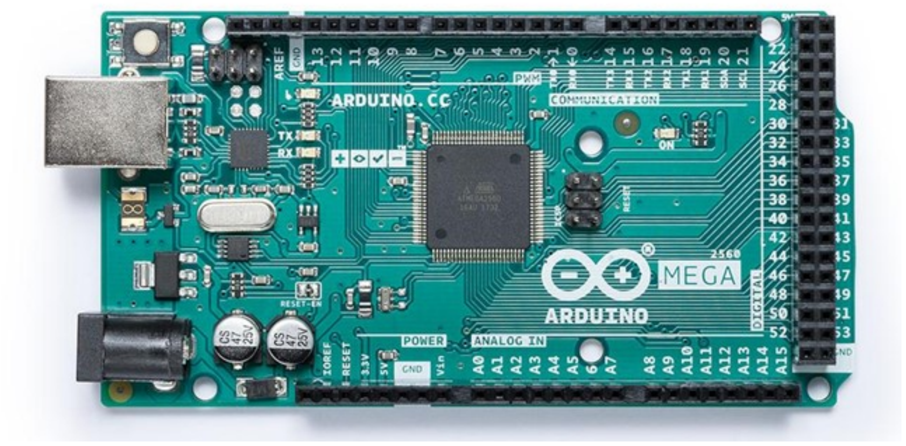


Figure 4.6: *Arduino Mega 2560*

chip: thanks to the large number of input and output lines it operates with, it gives the provision of interfacing many external devices. At the same time the operations and processing is not slowed due to its significantly larger RAM than the processors concerning the other Arduino boards. The board also features a USB serial processor ATmega16U2 which acts an interface between the USB input signals and the main processor. This increases the flexibility of interfacing and connecting peripherals to it.

The choice of this board for the technological migration was motivated by the constraints involved in the way the reactivation has been performed. The *Interaction* node has been left untouched, i. e. no change has been implemented in the acquisition board inside the pinball and therefore in the Parallel Port interface through which data is outputted. In this context, considering that the DB25 port has TTL-logic-levels pins, there is therefore the need to use either a board whose input/output pins work in the range 0 to 5 V or a *logic-level shifter* circuit (also known as voltage level translator) to translate signals from one *voltage domain* to another. The latter allows the compatibility between the DB25 pins and integrated circuits with different voltage requirements, as the *Raspberry Pi* boards or the *ESP32* microcontrollers working with 3.3 V voltage domain pins. Originally the reactivation project considered the usage of a Raspberry Pi 3 board, a small single-board computer (SBC) providing a set of GPIO (general purpose input/output) pins, together with a DB25 to USB connector to acquire data from the acquisition board via Parallel Port interface. Despite the USB input was correctly detected by the Raspberry Pi, no data coming from the *Interaction* node could be received successfully. In fact several tests conducted using the *pyUSB* and *pySerial* libraries (allowing USB and Serial access) for Python failed in this scope. The motivation for this is the way the aforementioned converter was designed for: it grants USB communication for old printers whose output port is the Parallel one only, according to the modes contained in the IEEE 1284 standard as described in the subsection 2.2.1.1. However, the Nibble Mode considered for the *Il Caos delle Sfere* installation is a modified version of the standard one causing the impossibility to translate data correctly for any Parallel port-to-USB converter. In addition to that, starting from a DB25 Male Solder Connector inserted in the Female DB25 connector outputted by the acquisition board, an attempt to interface its pins and the GPIO pins of the Raspberry PI via a logic-level shifter failed too. That was motivated by problems on the current values sourced by the Parallel Port that were not sufficient to successfully change the Voltage level on the output side (the one towards the Raspberry) of the voltage level translator. Even if a signal coming from the acquisition board ramped down from 0 to 5 V, the corresponding 3.3 V signal was not doing the same. The exact same problem occurred while trying to interface the Parallel Port pins with the GPIO pins of a *ESP32* microcontroller, a low-power system on a chip with integrated Wi-Fi and dual-mode Bluetooth. The only board allowing a correct reading (and writing) of the Parallel Port pins was the Arduino one. In particular the choice was to use a Mega version, both due to the high number of digital input/output pins available (54, as can be seen in figure 4.7) compared with the 25 required by the DB25 port and due to the computational capability the ATmega2560 processor provides.

4.2.2 The replication of the pinball data acquisition

Given the considerations presented in the previous section, a replica of the communication test between pinball and PC can be performed by acting on each of the pins of the 25-pin D-sub connector separately. This has been done by using a DB25 Male D-Sub solder type Connector connected to the parallel cable of the pinball on one side and to an Arduino Mega's breadboard on the other side.

To perform the communication test, similar operations with respect to the one computed by the PC in the original installation are implemented. In details they are:

1. The board reads the state of the pins connected to the status lines of the port and inverts the one associated to $\overline{S7}$ (first nibble read);
2. The board sets all the pins connected to the control lines of the port to LOW (except for $\overline{C0}$ which is set to HIGH) and waits in an idle state for 5 ms;

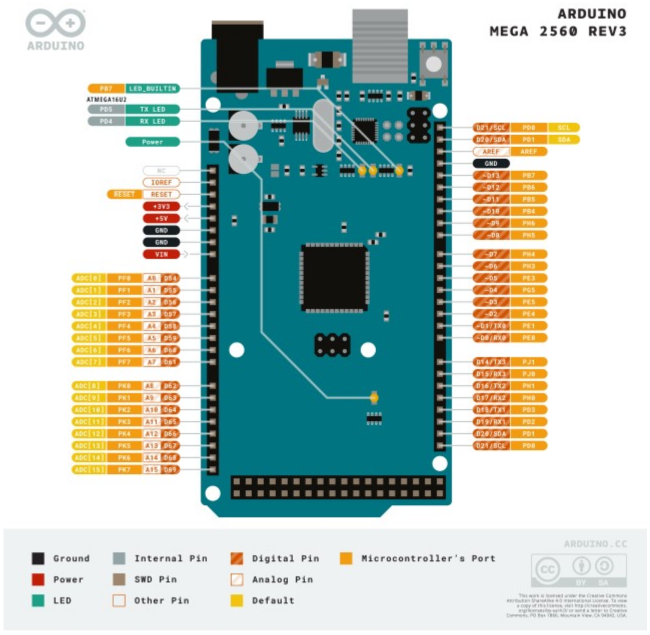


Figure 4.7: *Arduino Mega 2560 pinout.*

3. The board reads again the state of the pins connected to the status lines of the port and inverts the one associated to $\overline{S7}$ (second nibble read);
4. The board reads the state of the pins connected to the data lines of the port and compose an integer out of them;
5. The board sets all the pins connected to the control lines of the port to LOW ;
6. The board performs the check of the states of the status lines (over both reading phases) and computes the integer associated to the last hit target and the name of the lights that are on; it finally waits in an idle state for 3 ms.

4.2.3 The hardware of the proposed reactivation

As previously anticipated, the hardware considered in the reactivation consists on an Arduino Mega board which acquires raw data coming from the pinball via the Parallel Port. Arduino's pins are interfaced pin-by-pin to the ones of a DB25 male connector. Since the input and output of the Parallel Port are in TTL logic levels, it is enough to use just the ground and the digital pins of the Arduino Mega board to implement the connection. Figure 4.8 shows this connec-

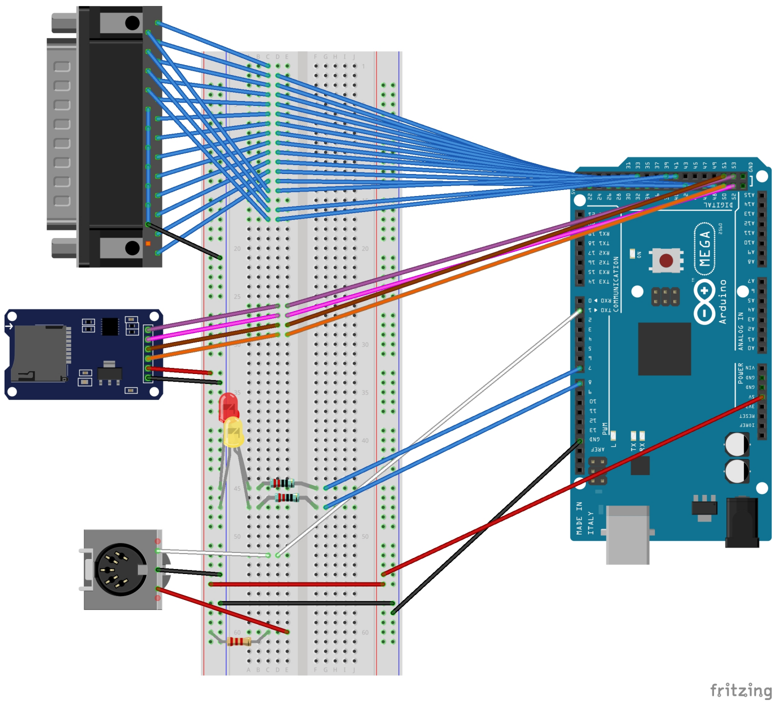


Figure 4.8: *Schematic representation of the hardware composing the installation.*

tion. All the ground pins of the DB25 connector have been short-circuited via Breadboard, a solder-less construction base used for developing an electronic circuit and wiring for projects with microcontroller boards, to a ground pin of the Arduino board. With reference to the Parallel Port and Arduino respectively, the pin association is 1 ($\overline{\text{C0}}$)-31, 2(D0)-22, 3(D1)-24, 4(D2)-26, 5(D3)-28, 6(D4)-30, 7(D5)-32, 8(D6)-34, 9(D7)-36, 10(S6)-46, 11($\overline{\text{S7}}$)-48, 12(S5)-44, 13(S4)-42, 14($\overline{\text{C1}}$)-33, 16(C2)-41 and 17($\overline{\text{C3}}$)-39. Pin 15 is not considered in the process. The hardware also consists on a microSD card reader, in which a microSD card containing three file *.txt for pre-written sequences can be inserted, and a MIDI port through which MIDI events generated by the Arduino board can be sent via MIDI cables to the Disklavier.

The choice of storing sequences on an external memory device as a microSD card is because of the size of those MIDI messages' sequences. Most part of them contains over 6000 messages made by 4 integers so even using the most efficient way to store all fields makes it impossible to store sequences in the limited internal Arduino memory. The Arduino Mega board has 256 kB of flash memory for storing code: saving all MIDI events as global variables (i.e. arrays of integers) would cause a memory overflow. The only way to access such sequences is to store them in an external source as a microSD card. In order for the Arduino board to read from the card, an SD card module is used. The module interfaces with the board in the *Serial Peripheral Interface* SPI protocol. The SPI bus is a synchronous serial interface data bus with full duplex, few signal lines and fast transmission speed [22]. It consists on a *master-slave* communication mode, i.e. the SPI specifies that the communication between two devices must be controlled by the master device. A master device can control multiple slave devices by providing a clock and Slave Select for the slave device. The SPI protocol also stipulates that the slave device's clock is provided by the master device to the slave device through a pin named SCK. In particular, master and the slave devices are connected by four signal lines, named SCK, MOSI, MISO, CS. SCK (*Synchronous clock signal*) is used to synchronize data transmission between master and slave. The host controls the output while the slave receives and transmits data on the edge of SCK. MOSI (*master output, slave input* signal) is the line where the master sends data to the slave and the slave receives the data through it. On the contrary, MISO (*master input, slave output* signal) is used for the slave to transmit data and for the master to receive it. Finally, the CS (*Chip Select*) is a signal used by the master device to select the slave device to communicate with [22]. This connection has been implemented via the following SD module- Arduino pins interface via the breadboard. The pin association is CS-53, SCK-52, MOSI-51, MISO-50 and both VCC and ground of the module have been short-circuited with the Arduino ones. The choice of these specific pins of Arduino is because they are the ones designed to implement the SPI protocol.

For what concerns the transmission of MIDI events to the Disklavier, a MIDI connector has been considered. As described in section 2.2.3, MIDI is a serial protocol that operates at $31\,250\,\mathrm{bit\,s^{-1}}$ while the transmission occurs via 5 poles DIN connectors and cables. In order to make this communication between Arduino and any MIDI controller possible, the board built-in serial port and a $31\,250\,\mathrm{bit\,s^{-1}}$ baud rate have to be used. However, since MIDI events are just sent by the Arduino, only the pin for serial transmission has to be considered, named pin 1 (TX). With reference to figure 2.11, the board is wired to the MIDI port (via breadboard) according to the following structure: MIDI pin 5 (MIDI data) is connected to pin 1 of Arduino through a $220\,\Omega$ resistor, MIDI pin 2 (Shield) is connected to ground and MIDI jack pin 4 is connected to +5V generator pin of the Mega board.

As can be seen in figure 4.8, the last component of the hardware is a pair of leds coupled with two $220\,\Omega$ resistors connected to pins 7 and 8 of the Arduino board. While they have no specific role in the final performance for the *Il Caos delle Sfere* reactivation, they have been placed to perform a control. The control is about a problem which has been showed up during the reactivation process, i.e. it consists on some switch values' errors. The first problem is that the value read by the Arduino via Parallel Port is not the one corresponding to the last target hit by the ball but the same with a decrease of 2. In order to assess if this problem is occurring, one led turns on if when the ball is on the shooting ramp the value read is 64 rather than the correct one, 66. Such a led is activated by setting the voltage level on the pin 8 "HIGH". The same led is used to reveal if the ball is on the trough, i.e. the tunnel through which a ball passes prior to be prepared on the shooting ramp. Therefore the led lights up if the value read is the one associated to the trough switch, number 58. The second problem is the fact that the value 27 is continuously read by the Arduino. The cause of this error is the circuity inside the pinball, which sets that the switch of the letter "I" in the "PAID" group active even of it's not activated by the ball. To observe if this situation happens, the second led activates as soon as the value 27 is read. The same led is also used to evaluate whether the value associated to the ball being on the shooting ramp is 66.

4.2.4 Multithreading requirement in MIDI sequences generation

The main problem in the technological migration is the way MIDI events can be outputted with the right timing in order to make the playback temporal distance between MIDI messages (either contained in the pre-written sequences or generated in real-time) to be equal to the delta time field composing the corresponding MIDI events. In the original source code, as anticipated in the chapter 3, such task was accomplished by the *MidiShare* environment using the functions *MidiTask* and *MidiDTask* which allows to send an event over the Parallel Port. The characteristic of these and the other library's functions is that they can schedule such a transmission in any given instant of time, fulfilling the requirement of getting a perfect playback temporal distance. This was possible because all the events' transmission tasks are scheduled and are executed on a separated *thread* with respect to the main one where the main part of the source code runs, i.e. all the functions to be called are executed.

In programming terminology, an activity potentially executed concurrently with other activities is called a *task*. Instead, a *thread* can be described as the system-level representation of a computer's facilities for executing a task. A thread is an abstraction of the computer hardware's notion of a computation. In most programming languages, included the C one used by the *MidiShare* environment, there are the so called *library threads*, intended to map one-to-one with the operating system's threads. Threads are considered when several tasks in a program need to progress concurrently. The point is that, on a system with several processing units (known as *cores*), threads allows us to use those units [25]. For what concerns the source code, there is a main thread, that acts as a manager of the other thread, dedicated to the execution of all the main routine of the algorithm and the scheduling of functions and MIDI events transmissions to the *MidiShare* instance's destination which in turns schedules the execution of them in a secondary more important thread. The latter is in charge of either perform the transmission of MIDI events at the correct timing or to execute the scheduled functions or to schedule new events transmission or function calls. This is possible only if a given processor is *multicore*, i.e. it can support multiple threads running at the same time in parallel, and that is the case of the PC used for the original installation and the *MidiShare* environment. For Arduino the previous statement doesn't hold. Arduino, especially in its Mega version, cannot support multi-threading because the ATmega 2560 microcontroller has only one core and it is only capable of executing one instruction at a time. There is still the need to manage work on only a single core, but still to do multiple things at once as calling and scheduling functions and transmit MIDI events. In order to accomplish this concurrency requirement, needed to make the technological migration work as the original installation, an approach close to multiple threads can be implemented with some software though.

The routine flow characterizing the source code for the algorithm of the installation is an example of *Event-Driven programming*, a programming paradigm in which the flow of the program is determined by events such as messages passing from other programs or threads, user actions or sensor outputs. It is a common programming model for memory-constrained embedded systems, as sensor networks. In an event-driven application, there is generally a main loop that listens and waits for events and then triggers a callback function when one of those events is detected or occurs [12]. For the *Il Caos delle Sfere* case study, the events correspond to the outputting of a MIDI event or the execution of a function that was scheduled previously. Event-driven models does not support a blocking wait abstraction. Therefore, programmers of such systems frequently need to use state machines, i.e. programming architectures that allow dynamic flow to depend on values from previous states or user inputs, to implement control flow for high-level logic that cannot be expressed as a single event handler. In general the need for explicit state machines to manage control flow makes event-driven programming complex [12]. A solution to it is to adopt a programming abstraction called *Protothreads* that makes it possible to write event-driven programs in a thread-like style. Protothreads simplify programming by providing a conditional blocking wait operation, thereby reducing the need for explicit state machines. They provide sequential flow of control without complex state machines or full multi-threading [12].

For what concerns Arduino, there are different implementations of protothreading, different ways of performing what would normally be a multitasking operation. In general, it consists on breaking work down by the 'loops' or 'lines' of code being executed by a sketch. The concept is, if more stuff is to be done, Arduino loops would take longer, so each task will have vastly different 'loops per second' durations. In addition to that, each loop there is no other work to do, some less-demanding or less-frequent work is performed in the main loop (or nothing at all). When there is no main routine work to be done, then the routine checks to see if it's time to do one of those other pieces of work yet. If so, it branches off and goes do it. It's important to notice that tasks that are "blocking", meaning they have to be completed all at once without interruption, will still block other protothreads from occurring "on time", but for simple things like two loops executed simultaneously and performing quick actions like variable changes or changing output values, this approach is very suitable [3]. The library allowing this simple implementation of protothreading is the *TimedAction* one. The idea is to create global variable for objects of type TimedAction to be initialized with the *TimedAction()* function. The latter takes as parameters a time value, corresponding to how many seconds have to pass between a function call and the subsequent one and the name of the function itself. In the loop, the attribute .check() will be then called in each of the allocated TimedAction objects to see whether the corresponding task has to be executed [3].

A second way to implement and simulate multi-threading with an Arduino is to use something similar to a state machine. Most of the times the processes to be executed require a lot of different actions and a given amount of time to wait between 2 actions. In this context, it's better to separate this process into several small functions and create a state machine in the main program to call them one by one, when needed. This allows to also execute any other part of the program between these functions of the process. This is more suitable than protothreading in cases where the execution time of almost all the functions is very short and using explicit state machines to manage the flow of actions is not complex. This is the case for the *Il Caos delle Sfere* artwork, where tasks to be performed are reading and writing over arrays, generation of random numbers and transmission of MIDI messages. For this reason, in order to handle multithreading in the technological reactivation the choice is to use state machines rather than protothreading. Even if the number of functions and objects to be considered in the algorithm is large, the number of states machines to use is small. The states involved in *Il Caos delle Sfere* are about verifying if a real-time generated or pre-written sequence is playing or not so introducing a limited amount of state machines is not increasing the overall complexity. The rationale behind the approach is that the code should be such that Arduino doesn't hang to wait for something. If the algorithm is listening to an input, for example a text message over serial communication, then it means that it doesn't control when this event will happen. The trivial way to get the input is to wait for it and then continue the execution of the program as data arrives. However this is not efficient and, more importantly, destroys the objective of simulating multiplexing and multithreading. There should be ways to keep external communication non-blocking for the rest of the algorithm. For the reactivation this translates into the need to find a way to keep the execution of the routine of the algorithm while continue on reading or generating sequences and outputting MIDI events with the right timing.

The implementation of state machines in the code has been done via the presence of multiple status variables, defining if a sequence is currently reproduces, i.e. its MIDI events are being reproduced, and, more importantly, via some time tracking techniques to trigger an action when it's the right time. For example, a MIDI event is sent to the MIDI port only if the time passed since the transmission of the previous event is larger or equal to the delta time field characterizing it. The key tool to enable this possibility is to call the *millis()* function for every iteration of the loop function in the Arduino. Millis() returns the number of milliseconds passed since the Arduino board began running the current program loaded in it. This is then used to compute the time difference between a current instant of time and the last time an action was triggered and if the duration is greater or equal than the interval required the task can be completed. If not, a block has to be inserted to not make the execution of an action possible before the time interval has not passed. Such a control requires the comparison between a delta-time, which is repeatedly updated at every loop iteration using the millis() function, and the period required; as soon as a sufficient amount of time has passed, the delta-time variable need to be reset down to 0 in order to then wait for the next time period to be completed. In the meanwhile some other actions can be done by the Arduino: provided that all the operations are fast from a computational point of view and the code is properly divided into small blocks or functions, this strategy results to simulate very well the multithreading required, apart from some negligible delays that for the *Il Caos delle Sfere* reactivation do not change the final performance. It's important to note that the *delay()* function cannot be used. This function completely blocks a program and destroy any attempt to simulate multi-tasking.

4.2.5 The algorithm and the Arduino sketch

In order to replicate the same musical performance that the original installation considers, the algorithm presented in section 3.1 has to be adapted to follow the characteristics that the Arduino board provides. The main difficulties to assess are on one side the absence of any multithreading capability and on the other side the limited amount of memory available in the board. For the first issues, as presented in the previous subsection, using multiple status variables leads to an efficient simulation of multitasking. Not so many tasks have to be executed therefore the result is almost as accurate as with the usage of *MidiShare* in the original computer. Regarding the strict Arduino memory requirement, every MIDI events sequence to be uploaded cannot be saved inside the board as anticipated in subsection 4.2.3. Therefore there is the need to consider an external memory device as a microSD card to interface with the board. Every *.wri file in the original computer is to be saved inside the microSD. However, to simplify the reading process, three *.txt files are generated out of all the WRI files. All the sequences characterizing the different levels are contained continuously in a unique file, named `SQNZ.txt` while the chord sequences are contained in `acc1.txt` and `acc2.txt` respectively. On the contrary, the Tilt sequence has not been implemented in the reactivation. Merging every sequence in a *.txt file firstly allows a fast reading over the board because no large amount of overhead is contained within such type of text file. Secondly, it is the most suitable way to store sequences according to the manner sequences are read in the proposed source code. Another constraint involving the memory is the need to keep the amount of global variables the smallest possible to not overload every task that the Arduino board executes. Therefore, in the modification of the algorithm for the reactivation, the major part of variables considered in the original code were dropped. Most of them is about libraries that cannot be imported in the board so no inefficiency was introduced in the modification of the code.

Every part of the code, excluded the MIDI sequences files, are contained in one Arduino sketch named `Science4All.ino` to be then uploaded in the board. The approach that was followed in modifying the original source code was to use still an object-orienting programming approach but much more simplified especially in the sequences handling. This translates into only two classes or structures developed (named `Seq` and `waitSeq`) that provide the basis to generate any other not pre-written MIDI sequence. For all the other types no object was implemented and functions to initialize, set and update the parameters and to send the MIDI events to the MIDI port were developed. In this way the code gets simplified conceptually but it gets very inefficient from the coding point of view. Despite that, the code proved to work and not to overload the Arduino board tasks execution.

```
//--------------------
// SD MANAGEMENT
//--------------------

#include <SPI.h>
#include <SD.h>

...
```

Inside the sketch, only two external libraries are considered: `SPI.h` which enables the Arduino board to communicate according to the SPI protocol with the Micro SD reader and `SD.h` which makes available all the functions to access the card, read and write data over it. The main part of the sketch consist on global variables and function definitions. Since the code was not developed using a massive object-oriented approach, several variables have to be accessed by multiple functions and all over the `loop` section of the sketch. That's why the majority of them is defined at the beginning of the file.

```
/******************************/
/*     IL CAOS DELLE SFERE   */
/******************************/

#define blex 15
#define p1 27
#define p2 26
#define p3 27
#define p4 30

...

//--------------------
// Arduino active pins
//--------------------

int pin_data_bit0 = 22;  //pin 2
int pin_data_bit1 = 24;  // pin 3
int pin_data_bit2 = 26;  // pin 4
int pin_data_bit3 = 28;  //pin 5
int pin_data_bit4 = 30;  // pin 6

...

int led_27 = 7;
int led_64 = 8;

...
```

In the snippet reported, it's worth to underline what enables the board to interface with the Parallel Port. As in the original code, it's necessary to specify the integer value corresponding to the targets hit by the ball. However by implementing a pin-by-pin connection and due to errors in the acquisition board inside the pinball, the resulting integers present some mismatch with respect to the mapping that was used in the original code and the one contained in the data-sheet. A new mapping was therefore drafted in order to make the new modified version of the algorithm follow the original routine for what concerns the musical performance. All the updated associations are included in the code as pre-processor statements for efficiency reason with the keyword `define`. On the other side, the Parallel Port-to-Arduino connection has to be implemented according to the scheme present in figure 4.8. At software level, this translates into setting an integer variable containing the Arduino pin used to read or write data into a Parallel Port pin. For example, with reference to line 17 of the previous snippet of code, the pin designed to interact with pin number 2 of the DB25 connector is the number 22 and this is saved in a variable named `pin_data_bit0` to suggest that pin 22 is the one which will read from bit 0 of the Data register. `led_27` and `led_64` are instead variables that specify the pins that light on the leds according to the problem presented in subsection 4.2.3.

```
...
//--------------------
// GESTIONE MIDI
//--------------------

// MIDI

#define noteON  (uint8_t)144
#define noteOFF  (uint8_t)131
#define allNoteOFF  (uint8_t)123
#define controlChange  (uint8_t)176
#define ctrlPedal  (uint8_t)64

...

// MIDI ARRAY
int               a_time[986];        // MILLISENCONDS
byte              a_note[986];        // MIDI NOTE
byte              a_velo[986];        // MIDI VELOCITY
int               W_IDX = 0;          // IDX MIDI ARRAYS (WRITE)
int               R_IDX = 0;          // IDX MIDI ARRAYS (READ)

...
```

One of the most important set of variables defined at the beginning of the sketch is made by MIDI events handling variables. Exactly as the source code presented in section 3.1, some integers are used to specify the type of MIDI message considered. For example, 144 is the Status byte indicating a Note ON message while 123 is the value of the second Data byte of a Control Change message of type all Note OFF. The latter is a fundamental type of control for the proposed reactivation. It turns off all notes that were turned on by received Note ON messages, and which haven't yet been turned off by respective Note OFF messages. Without relying on the *MidiShare* environment, all Note OFF is the only instruction which can flush all previous Note ON MIDI messages and stop the playback of every sound in the Disklavier. Arrays of integers as `a_time`, `a_note` and `a_velo` are instead structures defined to load MIDI events' fields (delta time, key and velocity) to be then sent to the MIDI port. Every pre-written sequence, both for the levels or for the chords, relies on those arrays to be loaded and sent out. The size of the arrays is 986, which is much less than the average length of MIDI sequences that is around 6000 events. However this is a design choice: instead of loading every event of the MIDI sequence, only 986 are considered at a time. The motivation under this choice is to avoid a full upload of every set of events, both for memory and time requirements. Loading 6000 events may take a large amount of milliseconds and may overload the limited storage capability of the board. In addition to that, the specific value of 986 was estimated by analysing several *.wri files. For efficiency reason, it's more efficient to load events which are spaced apart by 100 ms one with respect to the next one because they will be sent in output almost continuously. Thus avoiding that the board keeps on waiting the right time for them to be played. Therefore the task of sending them to the MIDI port will be completed in few loop iterations. The average number of consecutive MIDI events spaced by 100 ms is exactly 986. This strategy helps also for the final musical performance which results to be as if all the MIDI events in a sequence were processed consecutively. `W_IDX` and `R_IDX` are instead indexes to keep track of the position inside the three arrays for uploading events and for playing events respectively.

```
...

unsigned long        sqnz_pos_idx[45] = {0, 5231, 16405, 20670,
    24204, 31954, 42555, 52729, 60959, 72708, 85857, 103721, 126219,
    140139, 160672, 192948, 235794, 261305, 10792, 318987, 336964,
    367230, 427277, 436946, 447776, 467912, 488943, 500688, 513837,
    548104, 559242, 579711, 606732, 652785, 682766, 696332, 724519,
    748100, 788763, 829426, 848125, 873006, 923437, 961662,988399 };
int                  acc1_pos_idx[13] = {0, 306, 510, 869, 1236,
    1396, 2029, 3867, 4172, 4712, 5388, 6366, 6717};
int                  acc2_pos_idx[21] = {0,156, 2085, 2479, 3279,
    6004, 8675, 14764, 16617, 19953, 22511, 26740, 32908, 36564,
    37202, 38845, 40072, 42077, 44381, 46211, 52115};

...
```

Having all the level's sequences in one *.txt file raises another problem, which is how to read one specific sequence. In the original code there was no such issue: all sequences were indexed with their filename; this setup instead makes it impossible. However, even the *.wri files, each sequence ends with a row of four zeros in each MIDI event field (considering also the type of message), acting as a delimiter. This feature has also been considered when merging all the set in one file. In addition to that, the SD handling library included as header provides an API for reading files stored in the card byte by byte. Using such a function, it's possible to find out the position (intended as in number of bytes) of the first element of a row immediately after one ending row for a sequence. In other terms it's possible to find the starting point (in byte) for every sequence. Such positions were calculated and saved in an array named `sqnz_pos_idx`. The reading function and the loading function use `sqnz_pos_idx` to move across the sequence files. For the two chords sequences (named `acc1.txt` and `acc2.txt`) something similar was done to determine each part of the sequences to be played alone.

```
...

//--------------------
// TIME VARIABLES
//--------------------

unsigned long   t;                    // TIME FROM LOOP BEGINNING -
    milliseconds
int             t0 = 0;               // TIME -> LAST MIDI CMD
int             te = 5;               // ERROR TIME
unsigned long   tp = 0;               // TIME PORT READING
unsigned long   tluci = 0;

int             t0s;                  // lAST TARGET CONTROL

int             t0t;                  // LAST MIDI CMD FOR TRILLO
int             t0v;                  // LST MIDI CMD FOR VENTATA
int             t0p;                  // t0 pedale
unsigned long   checkFine;            // counter per fine gioco
...
```

Time handling to simulate multithreading is the fundamental aspect of the code that makes it possible to hear a musical performance in the end. As described in section 4.2.4, the simplest way to implement multitasking is to use state variables and to keep track of the amount of time passed since the last activity. Variables that contain such time instants are instantiated for the different tasks to execute. For example `t` is a global time variable which is updated at every loop iteration to count how many milliseconds passed since the Arduino board was turned on. It's the time variable each function refers to when deciding to change a state variable to enable an action. `t` is an unsigned long integer to account for the fact that, if the installation is presented in an exhibition, probably it has to stay active for a large amount of time. This implies that the number of milliseconds passed since the activation of the board quickly diverges and assumes very large values. Variables like `t0t` are instead relative time quantities. It represents the instant of time at which the playback of the last trillo sequence occurred.

```
...

/*FLAGS*/

bool                trilON = false;    // TRILLO IS PLAYING
bool                SWITCH = false;    // HIT A "SWITCH" TARGET
bool                RAMP = false;      // HIT A "RAMP" TARGET
bool                venON = false;     // PLAY VENTATA DURING ARPA
bool                venON2 = false;    // VENTATA IS RUNNING
bool                playSQNZ = false;  // PLAY A SEQUENCE
bool                arpaON = false;    // ARPA IS RUNNING
bool                arpaON2 = false;   // CHORDS DURING ARPA
bool                flagVal = false;   // Se luci sono stabili
bool                startcountFine = false; //flag per inizio count fine
bool                game = false;      // flag inizio gioco
...
```

To implement state-machine programming, several flags are considered as can be found in the snippet of code above. Depending on the type of sequence to play, flags like `trilON` or `arpaON` indicate whether the corresponding events are active or not. As will be discussed later, they are fundamental both to activate a specific task only at a given level but, above all, to avoid that the routine of the algorithm calls functions that require a time check to trigger (or not) any action. In other terms, they act as blocking variables for the functions that should be excluded from multi-threading simulation. This reduces the complexity involved in using state-machine programming by a large amount.

```
...
/*--------------
Gestione Trillo
--------------*/
uint8_t myNote;         // nota del trillo
uint8_t maxTri;         // massimo numero di trilli (da 3 a 7)
uint8_t actTri;         // ennesimo trillo ()
uint8_t actTri2;        // ennesimo trillo (numTri2)
```

```
char varTri;          // variazione pitch (note)
uint8_t velTri;          // velocity trillo

...
```

The rest of the global variables saved in Arduino memory are secondary variable used for example to save the level, the value read over the Parallel Port, etc. However, there is a large amount of variables used to represent sequence characteristics that in the original code were data members of ad hoc developed classes to wrap every type of sequence. In the snippet of code, for example, `myNote` is a global variable storing the note to be played about a trillo sequence. Having a lot of global variables like that is an inefficiency of the code but it's not a big issue because the amount of memory they occupy is very limited. Most of them can be saved as a `uint8_t` integer, i.e. a integer value taking just 8 bits of storage.

4.2.5.1 Reading the Parallel Port

```
...
void readPort(){
  if (port_phase == 0 and t-tp>10){
    tp = t;
    bit_0 = digitalRead(pin_status_bit4);
    bit_1 = digitalRead(pin_status_bit5);
    bit_2 = digitalRead(pin_status_bit6);
    bit_3 = (1-digitalRead(pin_status_bit7_n));

    digitalWrite(pin_control_bit0,HIGH);
    digitalWrite(pin_control_bit1,LOW);
    digitalWrite(pin_control_bit2,LOW);
    digitalWrite(pin_control_bit3,LOW);

    port_phase = 1;
    } else if (port_phase == 1 & t-tp>10){
    tp = t;

    bit_4 = digitalRead(pin_status_bit4);
    bit_5 = digitalRead(pin_status_bit5);
    bit_6 = digitalRead(pin_status_bit6);
    bit_7 = (1-digitalRead(pin_status_bit7_n));

    val = digitalRead(pin_data_bit0)*1 +digitalRead(pin_data_bit1)*2
     + digitalRead(pin_data_bit2)*4 +digitalRead(pin_data_bit3)*8 +
    digitalRead(pin_data_bit4)*16 + digitalRead(pin_data_bit5)*32 +
    digitalRead(pin_data_bit6)*64 + digitalRead(pin_data_bit7)*128;

    if(val==27 or val==66){
      digitalWrite(led_27,HIGH);
    } else {
      digitalWrite(led_27,LOW);
    }

    if(val==64 or val==58){
      digitalWrite(led_64,HIGH);
    } else {
      digitalWrite(led_64,LOW);
    }

    digitalWrite(pin_control_bit0,LOW);
    digitalWrite(pin_control_bit1,LOW);
    digitalWrite(pin_control_bit2,LOW);
    digitalWrite(pin_control_bit3,LOW);

    if (val!=last_switch_val){
      switch(val){
      case sw1: case sw2: case sw3: case sw4:
      case sw8: case sn1: case sn2: case sn4:
      case p1: case p2: case p4: case blex:
        SWITCH = true;
        t0s=t;
        last_switch_val = val;
        if(lev==0){
        gameLev=1;
        cambioLivello();
        }
      break;
      case trp1: case rpsi: case rpsu:
      case rpdi: case rpci: case rpco:
        RAMP = true;
        t0s=t;
        last_switch_val = val;
        if(lev==0){
        gameLev=1;
        cambioLivello();
        }
      break;
      case 58:                // Quando la pallina cade
        if(numBall==2){
        last_switch_val = 100;      // 100 non esiste come valore
        }else{
        last_switch_val = val;
        }
        if(numBall>0){
        numBall--;          // Diminuisci di uno il numero di
   palline
        gameLev = numBall;  // Se hai multiball torni al livello 1,
      se hai
75          }
76          cambioLivello();
77
78        default:
79        break;
80        }
81        }
82      port_phase = 0;
83    }
84  }
```

In the code presented in section 3 reading and writing over the Parallel Port was executed through scheduling of `scriviPorta` and `leggiPorta` with a time difference of 300 ms. `scriviPorta` reads the first nibble of data and switches the multiplexer input, while `leggiPorta` reads the second nibble, the integer value for the target and resets the multiplexer. In the proposed reactivation's code, the two phases are contained within the same function, `readPort`. It represents the first example of state machine programming, in order to move in time between one phase and the other. Phases are distinguished with values 0 and 1 (saved in the variable `port_phase`) while `tp` is the time instant at which the last phase was executed and entered. As it can be seen in lines 3 and 16 of the above code, the if statements are true only for one phase and if a sufficient amount of time since the last phase has passed. The routine can't enter the if statements for the wrong phase and if the difference in time is not larger than 10 ms. If conditions are verified then `tp` is set to `t`, the global time of the routine. The reason why the time to wait between one phase and the other is much smaller than the one considered for the original code is that Parallel Port-Arduino communication and lights checks are decoupled in time. The light check to estimate the algorithm level is executed less frequently, to avoid situations in which a possible blinking may destroy the correctness of the light reading and, consequently, the algorithm level. From the snippet of code it's possible to notice how the Arduino board sets voltage levels (HIGH or LOW) to every Parallel Port pin connected to it. Functions `digitalRead` and `digitalWrite` fulfil the accomplishment. The result of this action is to save the value read for the lights (0 if a light is OFF, 1 if on) in variables named `bit_0`, `bit_1`, etc. The integer corresponding to the last target hit is instead written in `val`, composed starting from the voltage levels read over the Parallel Port's Data register. Lines 27 and 32 are instead the routines followed to asses if some problem in the reading occurs. Finally in the second phase, a check for `val` is executed, similar to what is done in `leggiPorta`. Firstly `val` has to be different from its previous value saved in `last_switch_val`. Secondly, according to if `val` indicates a ramp, the ball being on the left trough, etc. the number of balls is decreased or a new level is estimated and saved in `gameLev`. The function `cambioLivello` then assesses if the algorithm level has to change according to the estimated one.

4.2.5.2 Loading and Playing a pre-written sequence

```
/*** READ TEXT FROM TXT FILES ***/

void loadSQNZ(){
  SQNZ = SD.open("SQNZ.txt");
  is_reading = true;

  if (pos){
    SQNZ.seek(pos);
    }
    while (is_reading){
      String s = SQNZ.readStringUntil('\n');
      splitString(s);
      if (arr[0]<100 or W_IDX==0){
        if (arr[1]!=3){
          a_time[W_IDX] = arr[0];
          a_note[W_IDX] = arr[2];
          a_velo[W_IDX] = arr[3];
          W_IDX++;
        } else {
          a_time[W_IDX] = arr[0];
          a_note[W_IDX] = X;
          a_velo[W_IDX] = arr[3];
          W_IDX++;
    }
      pos = SQNZ.position();
  } else {
    is_reading = false;
    SQNZ.close();
  }
  }
}
```

In order to upload a set of MIDI events from `SQNZ.txt`, one spaced from the successive one by 100 ms, the function `loadSQNZ` is considered. It opens the file from the SD reader via the API `open` of the library. After having sought the current reading position `pos` (always expressed in number of bytes) and having activated the flag `is_reading` to indicate that the reading is on, it starts reading the txt file line by line starting from `pos`. It derives the 3 integer fields of the MIDI event contained in each line and stores them inside arrays `a_time`, `a_note` and `a_velo` distinguishing if they are of type Note ON /

Note OFF or Control change (this indication is saved on the second element composing each line of the file containing the sequence as indicated in line 14). Reading and writing over the arrays continues only up to when a MIDI event is spaced from the successive by more than 100 ms. At this point `is_reading` is set as false and the file can be closed, according to `SQNZ.close()`. Note that throughout the entire reading, the position inside the file `pos` is constantly updated. In this way, in the successive reading activity it's possible to restart from the point where the current reading action stopped. `W_IDX` is instead a reading index that is incremented by every unit per reading but its main role is to preserve the length of the part of the sequence uploaded when it will be played.

```
/*** READ MIDI SEQUENCE ***/

void readSQNZ(){
  int deltT = t - t0;
  while(deltT > a_time[R_IDX]-te and a_note[R_IDX]!=0){
    if(a_note[R_IDX]==X){
      pedalSet(a_velo[R_IDX]);
    } else{
      outMIDI(noteON, a_note[R_IDX], a_velo[R_IDX]);
    }
    R_IDX++;
    t0 = t;
    deltT = t - t0;
  }
  if (a_note[R_IDX]==0) {

    endSQNZ = true;
  }
}
```

To send to the MIDI port the events stored in the arrays, `readSQNZ` has to be considered. It represents the best example of dealing with state machines to implement multi-threading simulation. Given that `t0` is the time instant when the function is actually called, another variable named `deltT` is calculated from the difference between `t` and `t0`. It represents the time elapsed since either the first function call or the last MIDI event transmission. Using the reading index `R_IDX` (initialized at the beginning with 0) a while cycle is used to span all the events saved and transmit them only if the time elapsed between the last transmission and the current time is greater than the delta-time field of each corresponding MIDI event. This translates into the checking of `deltT>` `a_time[R_IDX]−te`. In this context `te` is a tolerance quantity used in order to anticipate a little bit the reproduction of an event. In the same while cycle, it's also verified if the current event is corresponding to the row which delimit the sequence to be played, i.e. if a_note[R_IDX] is not 0. If the time check doesn't hold the function ends; if instead an event can be sent to the MIDI port and if it's not carrying a Control change about the Pedal, outMIDI is called. outMIDI is in charge of transmitting the MIDI event to the MIDI Port, given its velocity, its type and its note. In addition to that, every time the transmission action is executed, R_IDX is incremented, t0 is updated to the current time value and deltT is reset to start another time counter for the subsequent transmission.

4.2.5.3 Midi Event transmission

```
/*** SEND MIDI MESSAGE ***/

void outMIDI(int cmd, int n, int vel) {
  if(cmd == noteON){
  vel = (int) vel/1.6;
  }

  Serial.write(cmd);              // send Note ON, note OFF or
    control change command (Status Byte)
  Serial.write(n);                // send pitch (Data byte 1)
  Serial.write(vel);              // send velocity (Data byte 2)
}
```

Starting from the MIDI events, MIDI messages have to be sent via the MIDI port to the Disklavier for the final musical performance. The Arduino board can communicate with the MIDI port only using the Serial communication protocol given the baud rate of $31\,250\,\mathrm{bit\,s^{-1}}$. Therefore any MIDI data has necessarily to be sent out via pin 1, designed in the Arduino Mega board to enable Serial transmission. The function that is called every time MIDI transmission is required is outMIDI. It takes as input parameters three integers cmd, n and vel. They are parameters defining univocally a MIDI message: cmd represents the type, either Note ON, Note OFF or a Control change, n represents the pitch in a Note ON/OFF message and the type of control in a Control change message and vel is the velocity for a Note ON/OFF message and the intensity of the control in Control change message. outMIDI transmits such integers using the serial communication and for this purpose the Serial.write API. Whenever the message is either Note ON or Note OFF, an additional rescaling by a factor 1.6 is applied over the velocity exactly as what is done on VidiSend for the original code.

4.2.5.4 The main routine

Exactly as the algorithm for the original installation, even in the code for the proposed reactivation the routine is based on a loop generated by the `loop` function inside the Arduino sketch. The path the routine follows can be found in figure 4.9.

```
loop() {
...
t = millis();
readPort();
if (game){
  if(t-tluci > 300){
    if(gameLev>0){
      controlloLuci();
    }
  if(flagVal){
    cambioLivello();
  }
  tluci = t;
  }
...
```

The loop executes operations starting from two main tasks: the control about the pinball game being off and the `readPort` function which enables the same tasks the original algorithm goes through. In order to maintain the multi-threading capability for the board, at every iteration the time reference variable `t` is updated with the current amount of millisecond that passed since the activation of the board. After the time update, `readPort`, as previously anticipated, executes the steps to acquire information about lights and the target hit every 10 ms and implements the level switch based on the available estimation. As opposite to the function `leggiPorta` in the original installation, the light check is mismatched with respect to the acquisition of the target value information. Checks are delayed in time by 300 ms and to block any additional light check a blocking if statement is considered. Using `tluci`, a variable that stores the instant of time of the previous check, the routine disables any action if the time difference between the actual instant of time for the loop `t` and `tluci` is smaller than the threshold. If the condition is verified the lights are controlled to estimate the new level and if `flagVal` is true then the level is actually updated via `cambioLivello`. `flagVal` is a indicator that is set to true if the lights are stable, i.e. if their value stays constant over more than 7 checks. It follows the same rationale behind `controllaLuci` function as seen in section 3.1. The choice to perform the light check less frequently than the target value reading is to protect the routine from misleading reading. As anticipated, blinking of all the lights inside the pinball occurs whenever some

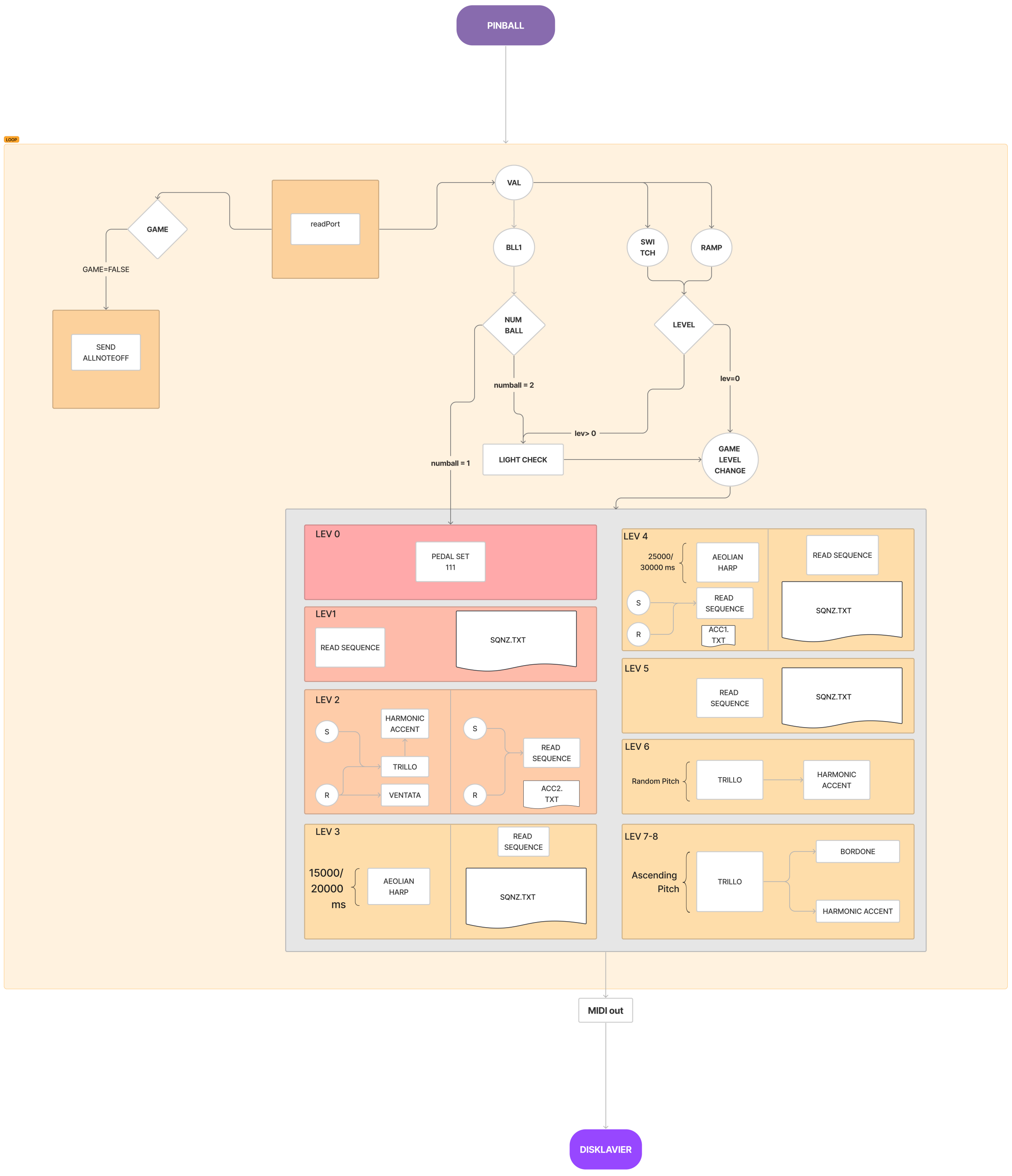


Figure 4.9: *The modified algorithm routine*

specific targets are hit in game. Potentially this could result in a wrong estimate of the algorithm level. To avoid that `flagVal` is set to true, the lights are controlled every 300 ms and this cause them to not be stable for more than seven readings since a blinking lasts around 1 or 2 seconds. After the reading `tluci` is updated with `t`.

```
void cambioLivello(){
  if(lev!=gameLev) {
    venON=false;
    venON2=false;
    trilON = false;
    endSQNZ = true;
    playSQNZ = false;
    arpaON=false;
    arpaON2=false;

    lev=gameLev;

...
```

The following snippet of code shows the function `cambioLivello` which is in charge of changing the level if the estimation of it is different from the current one. The actions to be performed are the same as the original function. The first action to take is to set to false any flag or status variable to stop any event reproduction or MIDI events generation. Secondly it updates the algorithm level `lev` to the estimated one `gameLev`. Thirdly it enters a switch case statement to set flags and executes the same actions as the original routine. The actions are not going to be treated in details. It's worth to described instead some common tasks the routine goes through for every level. The main task in levels that require a MIDI event playback starting from pre-written sequences is to set the values of `i_min` and `i_max`. They are indexes that are used in the loop function to indicate a set of byte indexes in the array `sqnz_pos_idx`. In other terms, `i_min` and `i_max` are used to select a set of MIDI sequences in the `SQNZ.txt` file to then randomly pick any one. `i_min` and `i_max` are different from level to level following the MIDI sequences distribution over the five indicated levels in the original routine.

```
...
    switch(lev){
      case 0:
        i_min = 0;
        i_max = 10;
        if(game){
          pedalSet(111);
          outMIDI(controlChange, allNoteOFF, 0);
          pedalSet(111);
        }

        numBall = 0;
        break;
      case 1:
        t0 = t;
        i_min = 10;
        i_max = 18;
        playSQNZ = true;

        numBall = 1;
        outMIDI(controlChange, allNoteOFF, 0);
        break;
      case 2:
        trilON = true;
        setTrillo(0);
        t0t = t - 75;
        numTotTri = 15;
        trilSQNZ = false;
        acc2_i = 0;
        numBall = 1;
        outMIDI(controlChange, allNoteOFF, 0);
        break;
      case 3:
        i_min = 18;
        i_max = 28;

        arpaInitialize();
        arpaON = true;
        arpaFine=t+15000+50*random(100);

        venON = true;
        numBall = 1;
        outMIDI(controlChange, allNoteOFF, 0);
        break;
      case 4:
        i_min = 28;
        i_max = 38;

        arpaInitialize();
        arpaON = true;
        arpaFine=t+25000+50*random(100);

        arpaON2 = true;

        venON = true;

        numBall = 1;
        outMIDI(controlChange, allNoteOFF, 0);
        break;
      case 5:
        i_min = 38;
        i_max = 43;
        playSQNZ = true;

        numBall = 1;
        outMIDI(controlChange, allNoteOFF, 0);
        break;
      case 6:
        numBall = 2;
        t0t = t - 75;
        trilON = true;
        setTrillo(0);

        outMIDI(controlChange, allNoteOFF, 0);
        break;
      case 7: case 8:
        numBall = 2;
        t0t = t - 75;
        trilON = true;
        setTrillo(1);

        outMIDI(controlChange, allNoteOFF, 0);
        break;
        default:
        break;
    }
  }
}
```

Each level setting is also characterized by `outMIDI` function call. It's executed to end any key pressing that possibly may be dangling while changing level. An allNoteOFF type of control change MIDI event is sent invoking the `outMIDI` function. Apart from that, levels over which trillos sequences are generated require also settings for their time reference variable as `t0t=t`−75. In this way the trillos can start immediately as soon as a switch or a ramp or a generic target is hit. Associated to each case there is an additional flag activation task according to what has to be reproduced at the level. For example at level 1, one out of the MIDI sequences for it has to be loaded, played and sent out to the Disklavier. For that purpose, a flag named `playSQNZ` is set true to enable the execution of the subroutine to send out MIDI sequence events inside the main `loop` function of the sketch.

```
...
/*******************************/
// SQNZ
/*******************************/
if (playSQNZ){
 6
 7   // PLAY SQNZ
 8
 9   if (endSQNZ){
10
11     // OR INITIALIZE OR END SQNZ
12
13     W_IDX=0;
14     R_IDX=0;
15     int i = random(i_min, i_max);
16     pos = sqnz_pos_idx[i];
17     loadSQNZ();
18     endSQNZ = false;
19   } else {
20   if (W_IDX>R_IDX){
21     readSQNZ();
22     }
23   else {
24     W_IDX = 0;
25     R_IDX = 0;
26     loadSQNZ();
27     }
28   }
29 }
30 ...
```

The piece of code reported above shows all the actions executed whenever a sequence has to be played provided that `playSQNZ` is activated. If a previous sequence MIDI events transmission ended (`endSQNZ`=true) both reading and writing indexes `W_IDX` and `R_IDX` are reset to restart all the procedures. Then a byte index is randomly picked, given the proper interval and saved in the variable `pos` that will be then used to upload the correct part of `SQNZ.txt` inside `loadSQNZ`. In the next loop cycle, the if statement just described wont' be entered because of `endSQNZ` set to false which denotes how the sequence-reading has to start. Every set of MIDI events (up to a total of 986) that was loaded with `loadSQNZ` is gonna be transmitted in output at the correct time instant following `readSQNZ`. As was previously presented, if the correct amount of time between two consecutive MIDI events playback has not elapsed, `readSQNZ` stays silent and no action is performed. The second if statement in line 20 of the code snippet is therefore entered but no action is performed indeed. Only when the entire part of MIDI events is reproduced, `W_IDX` and `R_IDX` are reset and the routine proceeds with another `loadSQNZ` call to upload the subsequent part of events composing the sequence to be played at that given time instants.

Similar routines are followed for trillos sequence and aeolian harps generation and transmission. Each type of newly-generated MIDI event sequence presents the same characteristics as the one generated in the original code. So bordone, chords, harmonic accents and canone are generated every time they are required. As it can be seen in figure 4.9, the sketch for the reactivation doesn't include any Tilt sequence. Its absence is only due to a practical reason: there is a very low probability that a player smashes the pinball so hard to activate the tilt indicator. This was in fact verified during the *Science4all* dissemination.

4.2.5.5 The ending of the game

One major issue that was encountered during the development of the code was that some MIDI events were sent out even if no game was on. The problem was that some blocking variables had to be established to avoid that reading of values by the Arduino board resulted in MIDI event generation whenever no ball was in game. To solve this issue, as anticipated in subsection 4.2.5.4, the `controllaLuci` function and any other action are enabled only if a flag named `game` is true. As its name suggests, it indicates whether the actual game in the pinball is on or not. Determining this condition is rather complex to be implemented in software. The following piece of code is a working solution for this issue.

```
...
    if (numBall==0){

    if (startcountFine){

      if (t-checkFine>9000){
        game = false;
        startcountFine = false;
        outMIDI(controlChange, allNoteOFF, 0);
        }

      if (val == 66){
        startcountFine = false;
        checkFine = 0;
      }
    } else {
  // FIRST 58
        if (val==58){
          checkFine = t;
          startcountFine = true;
          }
        }
        }
    } else {
```

```
25       outMIDI(controlChange, allNoteOFF, 0);
26 
27     if (startcountFine){
28       if (val!=66){
29         startcountFine = false;
30   }
31 
32       if (t-checkFine>200){
33         game = true;
34         startcountFine = false;
35       }
36       } else {
37         if(val == 66){
38           checkFine = t;
39           startcountFine = true;
40         }
41       }
42 
43     }
44   }
45 ...
```

Whenever `numBall` is set to 0, i.e. the ball goes out from the board, the algorithm has to check whether this event is the end of the game or if a new ball is gonna be shot over the shooting ramp and the game goes on. As soon as the first value 58 is read, the routine initializes a time variable named `checkFine` with the current time reference `t` and starts counting the amount of milliseconds passed since a new value 66 is gonna be read. In other terms, the time elapsed from the first time instant the ball enters the left trough to the first time instant the ball enters the shooting ramp. `startcountFine` is a flag that indicates the start of the count and it's set to true for this purpose. In the successive loop iterations, if the value 66 is verified then the counter is immediately stopped and `checkFine` is reset. Instead, according to line 6, if 9 seconds passed since the beginning of the count and no value 66 is read over the successive loop iterations then the game is considered as finished. The variable `game` is set to false and it inhibits all the functions routine since no light checks will be performed. The level stays 0. Once this situation is established, the action to be performed is just the transmission of allNoteOFF message at every loop iteration. Even if some MIDI events are erroneously transmitted in output, no sound will be heard. The only action that reactivate the game so the light check and the level change is when the value 66 is read consecutively in input for more than 200 ms. This helps discriminate a proper game restart (a complete new game or a new ball injected after one exited the board) from a blink of that value which lasts for fewer seconds. The entire game check routine is executed only whether `numBall` is 0 which implicitly excludes every cas in which during a multiball event one ball exits the pinball table.

4.3 Possible future improvements for the technological migration

The proposed technological migration is just a first step into a complete reactivation of the *Il Caos delle Sfere* artwork. Several improvements can be set up in order to have a more reliable and efficient reactivation. The main feature to be upgraded is the acquisition board, developed originally for the project. All the connections from the pinball switches and from the switch matrix circuit contained within the pinball circuit board to the acquisition circuity should be controlled. The aim has to be to check whether the resulting 8-bit-value associated to a generic switch corresponds to the value that can be found in the pinball data sheet. In addition to that, the action the acquisition board executes should be moved from an acquisition task only to acquisition and processing of data coming from the pinball. This means that the acquisition board should be integrated with a microprocessor which runs an algorithm exactly as what the Arduino Mega used in the proposed reactivation performs. The DB25 Parallel Port, being an old standard not present in any modern device, has to be removed in order to avoid possible problems in the pin-by-pin interfacing (as in the proposed hardware) with the port. The board should also contain a MIDI port: the microprocessor should generate MIDI events to be then transmitted via MIDI cable to the *Playback* node. In this way the Interaction and *Communication* nodes would be both contained within the *Creature from the Black Lagoon* pinball, eliminating any external hardware required for the installation. In the current setting there is the Parallel Port cable exiting the pinball, in the future it should be replaced with a MIDI OUT cable.

Apart from hardware substitution which may require a large amount of modifications, the very next improvement should be to make the Arduino sketch more efficient than what has been developed for the *Science4all* dissemination. The coding approach doesn't follow the object-oriented programming paradigm completely. Only two structures are considered an no class was developed for every type of sequence. In this way there is also the possibility to reduce the amount of global variables contained within the sketch. For that purpose, recently at CSC a fully object-oriented programming code was written by Alvise Bolzonella and Federico Pilotto. It considers objects not only for the type of sequences and the corresponding attributes or functions but also for any event-handling variable or function. For example, the following snippet of code is about a time object whose data members are the absolute time variable `to`, the time associated to Parallel Port phases `tp` and the one for light checks `tluci`. Only a member function is part of the class, `updt` which updates the absolute time instant `t` via the function `millis`.

```
#include "Arduino.h"
#include "Variabili.h"

Time::Time(){
t = millis();
t0 = t;
tp = t;
tluci = t;
}

/* Aggiorna il valore della variabile t all'inizio di ogni loop
Input: void
Output: void
*/
void Time::updt(){
t = millis();
}
```

A similar class was developed for the MIDI event transmission.

```
#include "Arduino.h"
#include "Midi.h"
#include "Variabili.h"

Midi::Midi(){}

/* Invio di un messaggio MIDI attraverso la porta seriale
Input:
cmd  =   MIDI Status Byte
n    =   MIDI Data byte 1
vel  =   MIDI Data byte 2
Output: void
*/
void Midi::outMIDI(int cmd, int n, int vel){
  if(cmd == noteON){
    vel = (int) vel/1.6;
  }
  Serial.write(cmd);              // invia il comando note on, note off o control change  (status byte)
  Serial.write(n);                // invio tono della nota da suonare (Data byte 1)
  Serial.write(vel);              // invio velocita' della nota (Data byte 2)
}
```

Another improvement should be to migrate and modify all the remaining versions of the source code and to implement them. Together with a more accurate research about every exhibitions the artwork went through, it could help in not only assessing the evolution of the installation from the artistic point of view but also to define in detail every DPO record introduced in section 4.1.

Conclusion

The thesis has addressed the topic of preservation of interactive multimedia artworks with a focus on the multilevel preservation model developed by the CSC group. Its importance in preservation and reactivation is that it considers artworks through a process of transformation rather than a fixed object. It defines a way to organize the multiple and heterogeneous components of artwork into a set of digital preservation objects. An approach to construct DPOs from the elements of an existing artwork has been presented for the case study of *Il Caos delle Sfere*. The artwork was the result of the artistic production of Carlo De Pirro collaborating with the CSC researchers in 1999. The multilevel preservation model has been applied for the reactivation of this artwork that was not presented in public since 2014. The setting of the original installation has been analysed in detail. The main hardware components and the software were all stored in the CSC laboratories for years. However, due to the obsolescence in the original components and the lack of a detailed documentation, a technological migration became necessary. The reactivation was carried out for the first exhibition after years of the artwork at the *Science4All* dissemination of 2022 in Padua. In this context, the thesis has discussed the main steps executed in the reactivation. The main difficulty was to interface the acquisition board inside the pinball machine via Parallel Port to a modern microcontroller board, the *Arduino Mega*, to replicate the same installation's acquisition routine. All the reasons why the Arduino board was selected among multiple potential boards to replace the old PC had been considered. A deep focus has been given also to the code and the software modification for the Arduino capability adaptation. From a very object-oriented programming relying on a external environment named *MidiShare* – a MIDI event time handler – all the algorithm has been redesigned to run over a microcontroller board. The thesis has also underlined how memory and multithreading issues arose over the research for the reactivation together with solutions that in the end allowed an efficient event handling. Both the algorithm defined for the original installation and for the reactivation have been analysed, especially in how the in-game actions translate into different algorithm levels and a different final musical performance played on the Disklavier. In addition to a more practical description, the thesis has also considered a new way to implement digital preservation objects for *Il Caos delle Sfere* records still in research that will be refined for future reactivations. It involves a metadata *schema* as the *Dublin Core* which needs however to be refined for the reactivation of interactive multimedia artworks.

Lots of work still needs to be executed, especially to make the reactivation of the artwork truly complete. A new electronic board should be interfaced with the pinball circuity. As it was explained in the thesis, a new acquisition board should be developed to perform pinball data collection, algorithm processing and MIDI events reproduction. Overall it should be included inside the pinball so that the only external piece of instrument would be the Disklavier. Only after this step, this brilliant artistic production by Carlo De Pirro would be finally at its real potential.

Bibliography

[1] *Midishare: A real-time operating system for musical applications.* www://midishare.sourceforge.net/. Online, accessed: 2022-08-16.

[2] *Ieee standard signaling method for a bidirectional parallel peripheral interface for personal computers*, IEEE Std 1284-1994, (1994).

[3] D. ALDEN, *How to "multithread" an arduino (protothreading tutorial).* www.create.arduino.cc/projecthub/reanimationxp/how-to-multithread-an-arduino-protothreading-tutorial-dd2c37, 2016. Online, accessed: 2022-10-23.

[4] ARDUINO S.R.L., *Arduino MEGA 2560 Rev3 Product Reference Manual*, 2022.

[5] H. BESSER, *Longevity of electronic art.*, in ICHIM (1), 2001, pp. 263–275.

[6] F. BRESSAN AND S. CANAZZA, *A systemic approach to the preservation of audio documents: Methodology and software tools*, Journal of Electrical and Computer Engineering, 2013 (2013).

[7] ——, *The challenge of preserving interactive sound art: a multi-level approach*, International Journal of Arts and Technology, 7 (2014), pp. 294–315.

[8] F. BRESSAN, S. CANAZZA, A.RODÀ, AND N. ORIO, *Preserving today for tomorrow: A case study of an archive of interactive music installations*, Proceedings of WEMIS-Workshop on Exploring Musical Information Spaces, (2009).

[9] W. J. BUCHANAN, *Parallel Port*, Springer US, 2004, pp. 641–664.

[10] B. E. CASTRIOTA, *Securing a futurity: artwork identity and authenticity in the conservation of contemporary art*, PhD thesis, University of Glasgow, 2019.

[11] G. DE POLI, *Standards for audio and music representation*, 2020, ch. 10, pp. 196–205.

[12] A. Dunkels, O. Schmidt, T. Voigt, and M. Ali, *Protothreads: Simplifying event-driven programming of memory-constrained embedded systems*, in Proceedings of the 4th International Conference on Embedded Networked Sensor Systems, SenSys '06, New York, NY, USA, 2006, Association for Computing Machinery, p. 29–42.

[13] R. Edmonson, *Memory of the world: general guidelines to safeguard documentary heritage*, 2002.

[14] International Council on Archives (ICA), *ISAD(G): General International Standard Archival Description*, 2000.

[15] J.Axelson, *Parallel Port Complete: Programming, Interfacing & Using the PC's Parallel Printer Port*, Lakeview Research, 1996.

[16] Midway Manufacturing Compamy, *Creature From The Black Lagoon*, 1 1993.

[17] M. Miliano, *The IASA cataloguing rules: a manual for the description of sound recordings and related audiovisual media*, no. 5, International Association of Sound and Audiovisual Archives, 1999.

[18] S. Nahavandi, *Industry 5.0—a human-centric solution*, Sustainability, 11 (2019), p. 4371.

[19] N. Orio and C. De Pirro, *Controlled refractions: A two-levels coding of musical gestures for interactive live performances.*, in ICMC, 1998.

[20] C. Peacock, *Interfacing the standard parallel port*, (1998).

[21] J. Pomerantz, *Metadata*, MIT Press, 2015.

[22] J. Qiang, Y. Gu, and G. Chen, *Fpga implementation of spi bus communication based on state machine method*, Journal of Physics: Conference Series, 1449 (2020), p. 012027.

[23] C. G. Saba, *Media art and the digital archive*, Preserving and Exhibiting Media Art, (2013).

[24] K. Stiles, *Performance art*, Oxford University Press, 2014.

[25] B. Stroustrup, *The C++ Programming Language*, Addison-Wesley Professional, 4th ed., 2013, pp. 1209–1210.

[26] G. WIJERS, *7.4 obsolete equipment: Ethics and practices of media art conservation. preserving and exhibiting media art.*, Preserving and exhibiting media art, (2014).

[27] YAMAHA CORPORATION, *Diskalvier ENSPIRE Owner's manual*, 2016.