Vigilada Mineducación

TÍTULO DEL PROYECTO

AdLibitum: A Non-Traditional Music Notation Software For Web Browsers

NOMBRES Y APELLIDOS COMPLETOS DE LOS AUTORES

Sebastián Arango Muñoz

TESIS DE MAESTRÍA

DOCENTE ASESOR

Juan Guillermo Lalinde Pulido

UNIVERSIDAD EAFIT

Escuela de Ingeniería

Maestría en Ingeniería

Medellín

2021

AdLibitum: A Non-Traditional Music Notation Software For Web Browsers

Abstract

AdLibitum is a web-based software tool that enables both traditional and non-traditional music writing. It aims to be straightforward and simple to use while still providing a vast range of features for a flexible music composition process. By running over web browsers, it also accounts for a large portability and universal access since these are widely supported across multiple platforms (e.g. desktop, mobile). One main feature at the core of AdLibitum is its support for non-traditional, artist-concept music notation through the use of tools for graphical creation and edition based upon Scalable Vector Graphics (SVG) object manipulation. Moreover, AdLibitum aims at preserving both notational and musical semantics of every compositional object to the extent possible, even for some of the non-traditional, author-provided objects and symbols for the purpose of reproducibility. In this article, technical generalities of the application are discussed under the light of software engineering and user experience perspectives. Additionally, strategies for tackling musical semantic preservation and reproducibility are presented. Furthermore, some relevant use cases and future directions of the software are outlined.

Keywords: Music software, web browser, non-traditional music notation, music semantics, graphical editor.

Introduction

While a large stake of computer music has turned towards digital signal processing and its ever-growing set of applications, symbolic representation of musical data at a digital level seems to be a matter of interest to a few theorists and software crafters only.

In a general sense, theoretical advances on semantic representation of musical objects serve software makers at creating more intuitive and accurate tools for music notation and composition, as it has been the case of such systems as OpenMusic (Agon, 1998), Finale, MuseScore (Schweer, 2011), GUIDO (Hoos, 1998) and Sibelius, which have been designed to primarily support some largely traditional, western composition styles. However, with the rise of other compositional variants such as experimental and electroacoustic music from the 20th century, traditional-only notation systems have been increasingly left behind in favor of more flexible writing schemas closer to performance than theoretical design, as it is the case of sequencers and digital audio workstations (DAWs) modules and plug-ins for real-time or instrumental aided-composition.

To certain extent, this extension of traditional notation with new compositional styles has been addressed by specialized software systems such as the Bach Project for Max (Agostini, 2015), Finale plug-ins and OOTMN (Eales, 1998). Nevertheless, all of the aforementioned pose several reproducibility and usability issues. While OOTMN proposes a very detailed object-oriented structuring to represent musical symbols (including non-contextual relationships typical from non-traditional writing), it is written purely in C++, which does provide high runtime speed and type safety yet makes it difficult to run in environments other than desktop applications. Whereas the Bach Project for Max (hereafter, just "Bach") is also written in C++, preserving the same technical qualities and constraints as of OOTMN, it suffers from additional issues. First, it runs strictly over the Max platform, which is a paid, desktop-only music creation environment. Hence, reproducing the artwork made with it is subject to the purchase of a license and the desktop software installation/configuration or, in other cases, to interchanging with other software platforms through cross-platform standards for music representation such as MusicXML (Good, 2001) or MIDI (Smith, 1981) which may derive in musical information loss. Moreover, given its graphical programming paradigm, adding custom, non trivial data processing functions becomes difficult (Agostini, 2015). Finale extensions for flexible composition also aim

to ease the graphical edition process within the core software platform but are often tied to the musical semantics management of the Finale core platform itself, so there are no means to preserve information regarding non-traditional objects. This results in a vendor lock-in in which it becomes necessary to pay for accessing the platform to take advantage of its plug-ins. Again, relying on representation standards for cross-platform reproducibility in this case might imply a loss of information due to their inability to represent non-contextual relationships between the graphical and semantic components of a non-standard composition.

Another challenging aspect of existing music notation software tools is their lack of built-in capabilities for graphical edition apart from musical symbology. Modern scores often require the use of free drawing and graphics manipulation in order to introduce author-created symbols and ornaments (Sabbe*, 1975). However, as mentioned above, these are rarely supported since platforms generally provide graphical support for pre-defined musical symbols only and there is no representation standard to preserve the semantic definition of author-specific objects. Due to this, some modern composers have switched to using merely graphics manipulation tools as Illustrator or Inkscape instead, which lack musical semantics but allow great flexibility at graphical design and edition.

AdLibitum surges as an alternative software platform for addressing the aforementioned challenges and constraints of existing tools. It offers a web-browser based platform for augmented music writing, bridging both standard notation and author-defined symbols in an easy-to-use interface accessible from several types of devices like laptops, desktop computers and mobile devices. It provides additional tools for free drawing and graphical edition based upon SVG manipulation, abstracting technical details from end users.

Software Tools For Music Notation

With the surge of several programming languages for computers towards the end of the 20th century, knowledge domains (including music) increasingly started to use them in order to automate some of their tasks and/or ease their otherwise manual processes. Given that advances in computer sciences were initially aimed at leveraging scientific and numerical computation, areas like digital signal processing (DSP) saw an early growth and, hence, DSP-oriented music software proliferated.

Nevertheless, adding domain context (particularly notation features) to music software took longer and was conducted by a few academic institutions and companies. Some of these tools are to be discussed in this chapter in a broad sense, since commercial products usually keep their technical principles out of public access.

Finale

Finale is a licensed desktop-based music notation package written in C++ created by Coda Music Technologies, now MakeMusic, in 1988. Initially supported by Windows OS only, now can be run over Macintosh OS too and is currently in its version 26.3. It allows the creation of plug-in software modules that can be seamlessly integrated with the core platform, enabling developers to add extensions according to their needs. Among its more relevant features are the adoption of the Garritan Personal Orchestra sound library for a more real playback, interoperability through semantic representation over MusicXML and MIDI, and the use of domain context to provide user-friendly behaviors such as proper stem direction, spatial alignment of multiple rhythmic values, enharmonic spelling and established conventions for positioning notes.

Sibelius

Sibelius is a licensed desktop-based music notation package first written in Assembly and then ported to C++ and originally developed by Sibelius Software Ltd, now part of Avid Technologies, in 1993. It supports a broad set of standard music notations and also runs over Windows and Macintosh operating systems only. Sibelius has a proven track of being used as a tool for democratising musical education (Nixon, 2011), and currently offers academic licensing. Like Finale, it allows interoperability with other platforms through formats like MusicXML, PDF and MIDI.

Bach Project

Bach is a package containing several pre-compiled modules for the Max music library, written in C/C++. It is oriented to real-time interaction at a graphical level, offering a seamless integration with other instruments and processor devices controlled by Max (Agostini, 2015). Bach stands upon the idea that music scores should be treated as collections of arbitrarily long sequential collections of data on various hierarchical levels instead of the primitive data structuring performed by Max, but taking advantage of its robust graphical environment. Among its restrictions are the need of having Max running, a licensed platform which only runs over Windows and Macintosh.

Smoosic

Smoosic is an open-source, web-based music writing tool written in TypeScript. Despite being at an early stage, it is capable of creating complete scores using a standard symbology. For the purpose of near-real-time rendering, it uses its own rendering engine called SMO which is in turn based upon VexFlow, a JavaScript library for rendering music notation. It is also compatible with the MusicXML and MIDI data formats, and provides an

Attribute	Finale	Sibelius	Bach	MuseScore	Smoosic	AdLibitum
Type	Application	Application	Extensions	Application	Application	Application
Distribution	Licensed	Licensed	Free download, Max requires license	Open source	Open source	Licensed
Language	C++	Assembler/C++	C/C++	C++	TypeScript	JavaScript
Platform	Desktop	Desktop	Max	Desktop	Browser	Browser
Operating System	Windows, Macintosh	Windows, Macintosh	Windows, Macintosh	Windows, Macintosh, GNU/Linux	All	All
Reproducibility	MusicXML, MIDI, PDF	MusicXML, MIDI, PDF	MusicXML, MIDI, PDF	MusicXML, MIDI, PDF	MusicXML, MIDI, Inter. API,	ADLB, MusicXML, PDF
Non-traditional support	Non-contextual graphics edition	No	Context preserving, GUI manipulation	No	No	Partially-contextual graphics edition

Table 1. Overall benchmark of music notation platforms on some features of interest for the AdLibitum design process.

interactive API to allow programmatic music writing as well. Among its usability features are MIDI playback, transposition, dynamic figure alteration and many more.

Web Browsers as Platforms For Music Writing

Created in 1990, the web browser emerged as a software application aimed at retrieving resources from the World Wide Web (WWW), an information system where content can be uniquely indexed by Universal Resource Locators (URL) using hypertext interlinking (Berners-Lee, 2004). Since then, their support for diverse hypermedia content such as text, audio, video and image has led them to become the de-facto way to retrieve multimedia from web servers. Even further, several actors such as the Web Audio Working Group of the W3C, Google and the Mozilla Foundation have worked on incorporating advanced music media formats like MIDI, enabling features like audio playback and contextual note transposition, among others (Wyse, 2013).

As of 2021, the dominant web browsers are Google Chrome, Microsoft Edge, Mozilla Firefox and Safari, yet there are other commercial options available. These applications can often be installed across several platforms and operating systems including Windows, Macintosh, GNU/Linux, Android and iOS, running on workstations, laptops, mobile phones and tablets, providing a huge cross-platform interoperability.

It is worth noting that web browsers can run standalone applications without the

need of an external server. This is mostly due to their support for JavaScript (JS) built-in functions. JavaScript is an interpreted programming language capable of natively running in browsers over their JavaScript and rendering engines. The former allows the execution of JS functions in the browser, and the latter manages the graphical processes running at the interface. This graphical management enables displaying images and vectors on the screen, which is a must-have for notation software.

Historically, software packages for music writing have been designed to work well as standalone desktop applications over specific operating systems, mostly Windows and Macintosh. However, given the increasing adoption of web browsers into all types of computing devices and their broad range of built-in features for multimedia processing make them a suitable option for running music writing software. For instance, Songsterr offers a web application for guitar tab writing that relies on SVG for graphics. Flat.io on its side includes a traditional notation suite with MIDI playback capabilities. Plus, entire JS libraries for music instruments creation in web browsers have been released (Roberts *et al*, 2015).

Representing Musical Objects In Software

Bridging music notation and software, while preserving as much contextual relationships as possible, has been a subject of research for decades (Dannenberg, 1993). It is to be recalled that music itself lies on multiple levels of abstractionism (Katayose, 1989), from audio to visual symbolism, and it is common for some use cases to have to move across these levels, which makes it harder to standardize the process of computational music representation.

One common approach to this relies on the use of hierarchies, allowing each core

object to have their behavioral information nested (Buxton, 1985). Nonetheless, given the complex relationships between objects and their contextual details, hierarchies may end up to be considerably deep (Dannenberg, 1993). This can become troublesome since searching across deeply nested data structures is proven to reduce efficiency. Samely, non-hierarchical approaches may be used too, although these may yield a very simplistic structure, limited in its semantic relationships.

A natural strategy from the software engineering context for mapping an object's properties to software is the object-oriented programming (OOP) paradigm, which has been largely used within the scope of music notation software as the founding principle of tools like OOTMN (Eales, 1998) and MuseScore. It allows to model real-life objects (e.g. a musical note) as programmatic entities consisting of data (often referred to as *attributes*) and actions (commonly called *methods* or *functions*).

Treating graphical components as SVG objects has proven to be one of the most appropriate decisions, as these objects are fully-compatible with today's web browsers rendering engines and provide manipulation capabilities at different levels. Unlike other formats such as PNG or JPEG, these are highly editable while preserving appearance ratios and pixel quality, and can also be embedded inside other instance, a base class named *Note* (1) can be inherited from the classes *WholeNote* and *QuarterNote*, without having to re-define the common attributes of a note for each type.

In addition to object representation in software, computer music programs must also consider the underlying time and structure conditions proper of compositions, understanding that the objects and relationships in real-time writing environments are not static and, therefore, need to keep track of them at a semantic level. Even though (Honing, 1993) stated the benefits of using an entirely declarative basis in order to preserve meaningful contexts in music representation of time, an adequate management of object data under

Note

- duration: float
- pitch; float
- hasAccents: bool
- SVG: str
- positionX: int
- positionY: int

+ getPitch(): float
+ setPitch(float): void
+ getDuration(): float
+ setDuration(float): void
+ getSVG(): str
+ setSVG(str): void
+ getPositionX(): int
+ setPositionX(int): void
+ getPositionY(): int
+ setPositionY(int): void

Figure 1. Example of a musical object defined under the object-oriented programming paradigm, with some of its inner attributes (up) and actions (down). Note that its inner properties are extensible to several types of musical notes.

OOP should account for a semantic preservation of objects and their relations.

Adding Non-Traditional Notation

More challenging is representing non-standard symbols, a common use case for modern composition. Typically, there is the need of adding new symbols for stating specific instrument aspects, and/or expanding the music notation model to take in modern music as well. The addition of new symbols means having to add fonts, relationships among other symbols, execution and formatting rules. For example, the addition of a symbol modeling a special figure (such as a cloud of notes) implies a deep change in the music notation model, whereas the addition of a new marker for notes can be performed by adding positioning rules and fonts, since the relationship with the note is already established (Bellini, 2003).

As seen in the previous section, musical objects can be treated as software objects too, which can in turn store their inner data. Hence, in theory, a software representation of an

object might be able to keep both its graphical and contextual information in its attributes. However, there are two main issues with this:

1. If the graphical attributes of an object are eventually modified by the user, those changes apply for that specific object only. This implies that these changes would need to be applied for every intended object (i.e. no replicability), and would also allow for multiple objects in the score, with different graphical attributes, to have the same musical semantics, which might yield interpretability inconsistencies and redundancy.

2. For the case when a user attempts to create a new graphical element (e.g. using the Figure tool), even being able to somehow store it, there is no way to assign musical context to it other than the rules and behaviors that other elements already pose, since musical rules can not be guessed from the artist's intention and, therefore, are to be previously defined from a traditional framework.

Despite the flaws of issue 1, modifying existing visual objects might enhance the compositional freedom of the user at a graphical layer. Therefore, this is thought to be a fully enabled feature in AdLibitum. Moreover, under the light of the above, it is also clear that in-place creation of brand new graphical symbols with an uniquely-defined musical semantic (issue 2) is not feasible under the scope of AdLibitum. Nonetheless, external and user-defined symbols with no implicit music context are to be allowed within the platform through an Actions menu which may include, but will not be limited in the future to options such as:

- Free-drawing
  - Pencil

    - Brush
    - Eraser
- Insertion
    - Basic Figure
        * Ellipse
        * Rectangle
        * Triangle
    - Image
        * SVG
        * JPEG
        * PNG
    - Symbol
        * SVG
        * Unicode

Hence, a reasonable way to handle both semantic-aware and non-aware object attributes may be decoupling musical and graphical information, although the same element should, at any moment, be able to refer to both sets of attributes through sharing an unique object identifier.

AdLibitum

AdLibitum is a web-browser-based application written in JavaScript, under the ECMAScript 6 standard. It relies on the ReactJS library to organize the front-end components

in an encapsulated and reproducible manner, and the FabricJS library to seamlessly interact with graphical objects at a high level. The use of these tools obeys the need of operating the application via web browsers, since JavaScript is a natural choice for this environment due to its engine's support. Besides, it heavily uses HTML5 for hypertext markup and CSS3 for visual styling of the web interface.

Being the general purpose programming language with the largest number of external/contributed libraries, JavaScript provides tools for handling several standard data formats including XML, JSON and MIDI. Also, it has the ability to read and parse those data structures from diverse sources such as files, databases or APIs, which enables integration with other applications that use these same data formats (e.g. Finale, MuseScore, Sibelius).

Software Design Principles

As aforementioned, AdLibitum relies on ReactJS as the core library for setting up its user interface and underlying logic. Thus, Presentational and Container patterns are followed in order to decouple visual and operating aspects of the software. As shown in fig. 2, the presentational layer holds the application components related to visual interfacing (i.e. what the user sees) and the container layer encapsulates the operating and data models (i.e. how the application works). This deliberated split between presentational and operational components increases control over the application's state and prevents undesired inconsistencies, since only the container layer can update stateful information whereas the presentational components can only receive it via parameters or ReactJS *props* without being able to modify states in any way.

Handling application-level states has no default support in ReactJS. In certain cases, when components hold parent-child relationships, passing information from one another

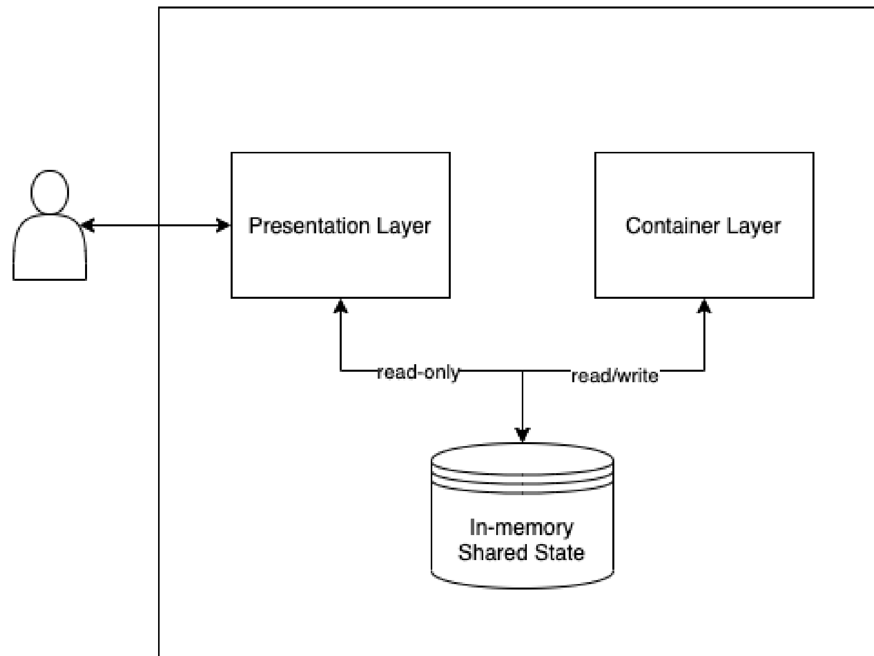

Figure 2. Presentation and container split for application components.

can be done through sharing ReactJS *props*. This becomes harder, though, when components do not hold such relationships, and it is often the case, so preserving a global state across isolated components requires additional efforts. However, AdLibitum requires to share the information of musical objects - initially loaded in-memory - with all the components in which these are to be presented and manipulated. This leads to the use of such tools as Redux, a state container for JavaScript applications. With it, persisting global states and traversal information becomes easier and safer.

Another design strategy enabled by the use of ReactJS is conditional rendering. Since AdLibitum aims to provide several distinct behaviors for its musical objects depending on the actions selected for them, before-rendering state validation becomes relevant at avoiding the creation and rendering of unnecessary components. For instance, when the "Move" mode is on, an object should become selectable and draggable across the canvas when clicked whereas, if the current mode is "Delete", that same click event should erase the selected object from the score (both visually and logically). Therefore, having these

state validations prior to rendering allows for more efficiency at component instantiation.

I/O Management

One of the main priorities for the software is accounting for interoperability with both AdLibitum instances and other music composing tools, specially at retrieving musical data based upon various formats. Given that JavaScript supports the reading of multiple format files, directly importing musical information can be straightforward. However, for the sake of lightness and simplicity, AdLibitum internally structures musical data using the XML and JSON formats, due to JavaScript's ease to treat them as native objects. Thus, this implies the need of having two main considerations: first, composition reproducibility across AdLibitum instances must be standardised upon a sigle file format and, second, it might prove useful to eventually have a conversion step where external formats (e.g. MusicJSON, MIDI, and so on) can be parsed to the AdLibitum data model.

As per the first consideration, the ADLB file format is proposed as in the Fig. 3. It consists of a lossless ZIP-compressed object with the *.adlb* extension, containing two files in its interior: *score.mxml* file, which carries all the musical and semantic information for every object within the composition in the MusicXML format; and a *graphics.json* file, which indicates explicit graphical alterations to be assigned to specific objects within the score, using a pre-defined key-value JSON schema. Once loaded into the software execution runtime, AdLibitum should be able to recover the last state of the composition in both musical and graphical contexts.

Regarding the second issue, and as it will be explained in further detail, AdLibitum proposes an internal translation engine in charge of parsing external information such as imported files or user actions into the appropriate data structures expected by the application data stores. Initially, this engine focuses on translating information retrieved

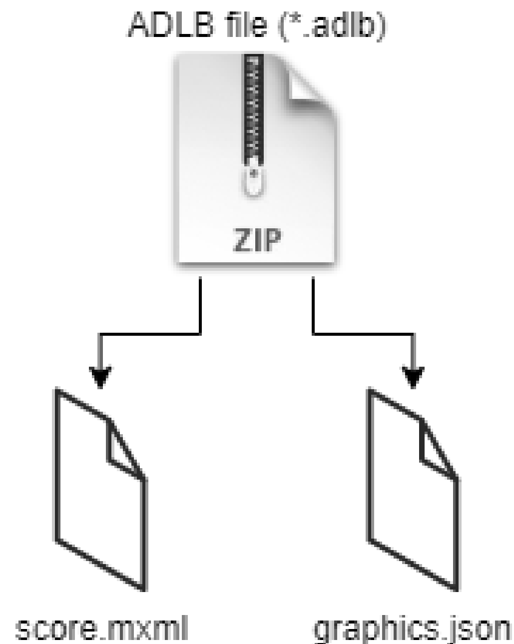

Figure 3. Constitution of the ADLB file, intended for complete portability across AdLibitum instances.

from user actions or ADLB files only (i.e. AdLibitum-to-AdLibitum portability only), but it can be eventually extended to support other formats.

Data Management

AdLibitum requires two main data stores. One is the definition for all the supported objects, which includes their graphical properties and musical semantics. This acts as an object reference and is thereby static and immutable at execution. On the other hand is the composition data model, which progressively stores all the created and modified elements. This needs to be constantly updated in runtime by interacting with the canvas. Since both of these represent light and structured information, both the JSON and XML formats are used to model them due to its native compatibility with JavaScript and web environments.

However, it is important to note that these models are not currently stored in database

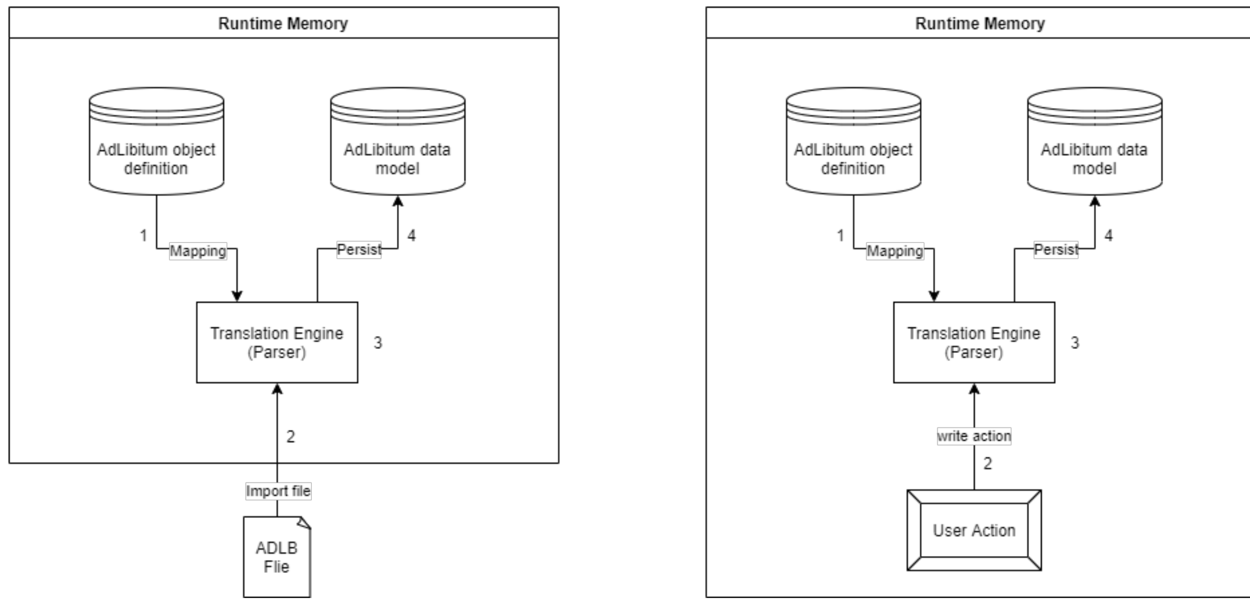

Figure 4. Process of adapting external music data (left) and writing music actions (right) to the internal data model.

systems, but are preserved in-memory. This way, the data models live at application execution only. For the case of the object mapping model, the base data is pre-defined and loaded into memory at runtime start and, for the composition data model, a multi-file (ADLB file) export is implemented so the current state can be downloaded and posteriorly recovered. Using this I/O-based strategy reduces the need of having external database services consuming compute and memory resources, and also allows reproducibility, since every running instance of AdLibitum should be able to recover from another instance's exported state only by importing the corresponding ADLB file.

In order to isolate the data stores from user actions, preventing them from facing undesired operations over a well-defined interface, AdLibitum implements a translation engine. Its aim is to establish a mapping between the incoming music data and the established musical semantics. Taking advantage of existing alternatives for this task, AdLibitum's translation engine is built upon the Smoosic's SMO engine (Newman, 2021), wrapping its core functionalities and extending them with a few application-specific

features.

As per the flow shown in Fig. 4, musical information first starts by being either uploaded from the ADLB file into the web interface or generated from user actions, and its contents are placed in cache so the corresponding translation engine (parser) can access them. The parser then retrieves the object definitions and proceeds to establish a semantic map between them and the contents of the uploaded file. After mapping corresponding fields, the parser writes the composition data model with the formatted information and, at last, this is the one to be served through the user interface by the rendering engine.

User Interfacing

As it is common for web-based applications, the user interface for AdLibitum needs to be rendered and displayed by the browser's built-in rendering engine. What it does is basically grouping the hypertext markup templates from HTML5 files, the style sheets from CSS3 files and the JS scripts to be processed by the browser's JavaScript engine, and translate that collection into visual components whose primary behaviour relies on the definitions stated in the JS scripts.

Nonetheless, adapting graphical information to be rendered by the browser the way it is stored requires some additional efforts. Given that the browser's JavaScript engine is only capable of displaying valid hypertext markup language tags and resources, the graphical metadata stored in the AdLibitum data model needs to be adapted to that. For this purpose, an internal rendering engine is incorporated to the software so the browser is able to understand the graphical information from the compositional data model. This one acts as an interface between the data model and the visual user interface so they can integrate and communicate seamlessly in spite of the structural complexity of the way the graphical attributes are stored and the formatting constraints of the browser's rendering

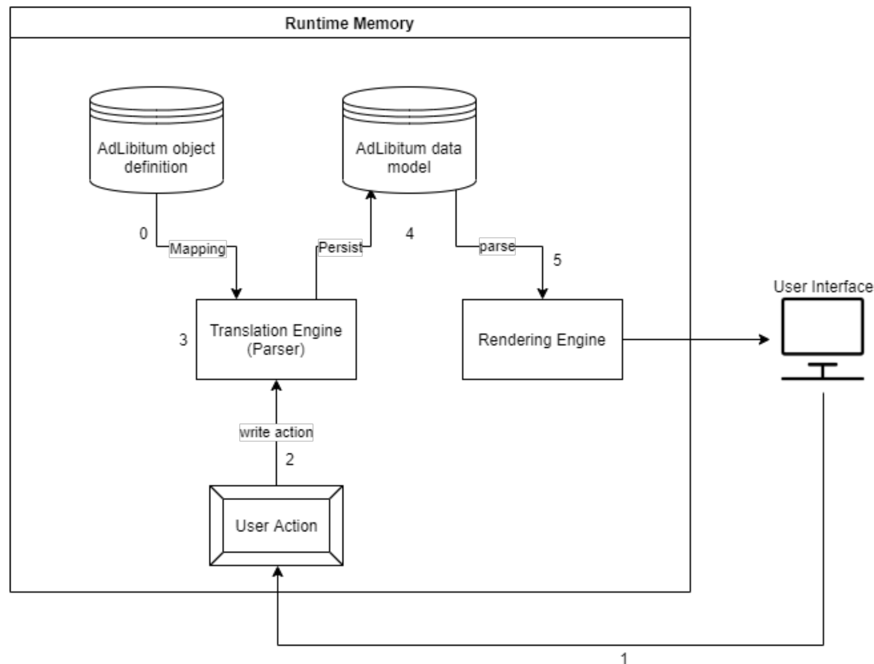

Figure 5. Interaction flow between the user and the AdLibitum memory management strategy.

engine. As evident in Fig. 5, the user never gets to directly manipulate the data models but, instead, the user interface sends both writing and reading actions to the translation and rendering engines respectively and these ensure that the requested information be adequately processed at the data stores.

Similarly to the translation engine, the rendering engine is also built upon the Smoosic's rendering engine, which is in turn based on a fork of the VexFlow project, an open-source JavaScript rendering engine for music notation on web browsers. With it, converting well-established (e.g. traditional) musical symbols into HTML-embedded SVG graphics becomes much easier and standard than implementing so from scratch. This allows for a more practical and modular implementation, yet has the issue of creating dependencies on third-party tools.

Additionally, in terms of the interface structuring, the ReactJS library incorporates a set of suggested practices for organising and managing templates, stylesheets and scripts.

A common scaffolding for these resources consists of organising them on a component basis. This is also the convention used in our application for the purposes of order and modularity, specially facing eventual incoming modules such as login and collaborative workspaces.

The user interface design aims to cover several usability needs. The core visual space of the application is the composition canvas, which is a sheet-like blank space placed in the middle of the page. This space must remain clear from other menus and tools since it is intended for adding symbols and traces only. However, musical fonts and symbols must also be easily accessible for the user. Thus, a vertical selection menu is placed to the left of the page so symbols can be easily searched through it in a hierarchical fashion. There, each symbol can be selected by clicking its corresponding button and this action automatically adds one instance of it to the central canvas. Another relevant usability aspect is enabling different behaviors and actions for the symbols in the canvas, such as movement, edition, deletion, persistence and so forth. All these actions can be centralised in a menu, which for our case is placed to the right side of the web page. A glance of the user interface can be seen in Fig. 6.

As of now, the layout and tool distribution of the AdLibitum's interface is based on the Smoosic visual appearance. This is due to its convenient proposed strategy to arrange menus and options in an accessible fashion with respect to the composition canvas. Besides, it allows to keep track of all the tools that Smoosic's SMO and API enable by being used as a basis for traditional notation management.

In addition to using the proposed menus, some keyboard shortcuts have also been added in order to improve usability. Actions such as copying, pasting and deleting symbols use their respective industry standard shortcuts (i.e. CTRL + X for cutting, CTRL + C for

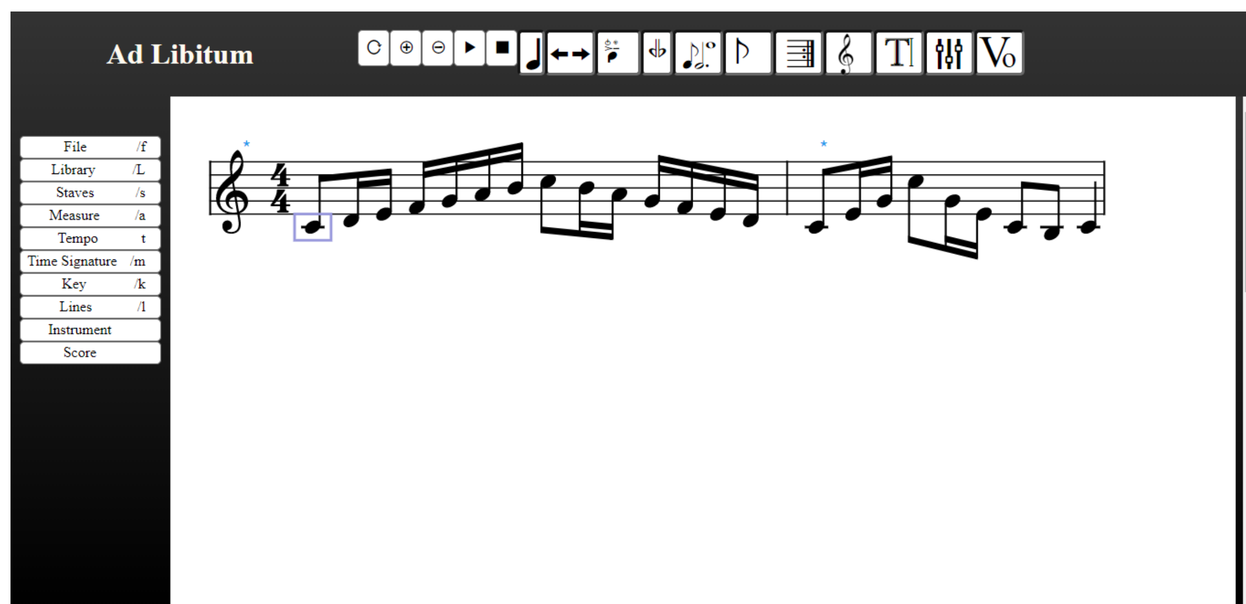

Figure 6. Current user interface for AdLibitum. This is closely based on the Smoosic's one, taking advantage of some pre-existing features.

copying, CTRL + V for pasting, and so forth).

Conclusions

Despite the continuous efforts, AdLibitum is still at a very early stage. Hence, our work so far has focused on developing core features, which involve adding traditional notation support in the first place and eventually increasing the subset of non-traditional capabilities. We have found that bridging the gap between graphical and musical information in order to preserve and reproduce it as a whole has proven to be more than difficult and we deem it might be still an open problem in music representation for digital environments, for which our data models might still need some extra refining.

Choosing web browsers as a base platform has proven to be useful in many aspects. Portability across operating systems and devices has not been problematic at all, even

mitigating installation issues. Support for JavaScript and its general purpose features makes it easier to add new functionalities that involve not only notation but also file parsing or user experience actions, which otherwise might be very complicated to add if using music writing specific languages. ReactJS has been very useful to abstract a lot of development details, but it might impose a steep learning curve for new users.

Treating graphical components as SVG objects has proven to be one of the most appropriate decisions, as these objects are fully-compatible with today's web browsers rendering engines and provide manipulation capabilities at different levels. Unlike other formats such as PNG or JPEG, these are highly editable while preserving appearance ratios and pixel quality, and can also be embedded inside other HTML components.

Regarding the user experience, there are still many aspects to be improved. For instance, testing users have stated that having to search for the symbols and actions in the side menus becomes increasingly tiring with the number of searches needed to complete a score, and so it would prove useful to add shortcuts and central access to some of the sub-menus from the canvas itself rather than getting out of it. Additionally, there are still some missing menus and actions to be implemented.

Future Directions

Given the short run the software has been through, there are several future steps to be addressed. So far, support for traditional notation has been the main focus but non-traditional features other than adding free-drawing, SVG edition and removing placement restrictions need to be defined and implemented according to current standards for modern composition.

In the near future there must be a thorough work on refining and standardising data models, so reproducibility and inter-operation with other music software tools becomes more straightforward. Since, at this moment, only AdLibitum-to-AdLibitum portability has been under active development, adding compatibility robustness through supporting other formats such as MIDI, MusicJSON and so forth is clearly in the development roadmap. Besides, the heavy dependence on the Smoosic's control engines form translation and rendering might provoke an undesirable lock-in phenomenon, which highlight the importance of building more decoupled engines for AdLibitum.

Regarding the user interfacing, there is still a lot of pending work. As of now, the visual interface looks very similar to the Smoosic's one since it provides convenient menus to interact with its internal composition API, but usability and layout distribution must be eventually improved according to feedback received from testing users. Moreover, the graphical edition menus are to be completed and set apart from the musical ones.

The business model for the software is still under administrative consideration. As it might work under license agreements, authorisation and authentication schemes may need to be considered, as well as deployment strategies that involve application hosting and serving.

References

An Object-Oriented Toolkit for Music Notation. 1998. Andrew Eales.

The Viability of the Web Browser as a Computer Music Platform. 2013. Lonce Wyse, Srikumar Subramanian.

Modeling Music Notation in the Internet Multimedia Age. 2003. P. Bellini, P. Nesi.

On the translation of languages from left to right. 1965. Donald Knuth.

Music Notation in the Twentieth Century. 1980. Kurt Stone.

MusicXML for notation and analysis. 2001. M. Good.

Music Notation: A Manual of Modern Practice. 1966. Paul A. Pisk, Gardner Read.

Designing musical instruments for the browser. 2015. Charles Roberts, Graham Wakefield, Matthew Wright, Joann Kuchera-Morin.

ChucK: A Strongly Timed Computer Music Language. 2015. Ge Wang, Perry R. Cook Spencer Salazar.

Syntactical and semantical aspects of Faust. 2004. Y. Orlarey, D. Fober, S. Letz.

Max at seventeen. 2002. Miller Puckette.

Kronos: A Declarative Metaprogramming Language for Digital Signal Processing. 2015. Vesa Norilo.

Languages for Computer Music. 2018. Roger B. Dannenberg.

Formal semantics for music notation control flow. 2013. Z. Jin, R. Dannenberg.

An Approach to an Artificial Music Expert. 1989. Haruhiro Katayose, Hirokazu Kato.

Diminuendo al bottom - Clarifying the semantics of music notation by re-modeling.

2019. Markus Lepper, Michael Oehler, Hartmuth Kinzler, Baltasar Trancon y Widemann.

GUIDO Music Notation - Towards an Adequate Representation of Score-level Music. 1998. Holger H. Hoos, Keith A. Hamel, Kai Flade, Jurgen Kilian.

Symbolic Music Representation. 2006. Pierfrancesco Bellini, Paolo Nesi, Maurizio Campanai, Giorgio Zoia.

Symbolic Music Representation in the MUSITECH Project. 2011. Tillman Weyde.

Music Programs: An Approach to Music Theory through Computational Linguistics. 1976. Stephen W. Smoliar.

The Concept of Musical Grammar. 1983. Mario Baroni, Simon Maguire, William Drabkin.

Visualization in comparative music research. 2006. P. Toiviainen, T. Eerola

On some computational models of music theory. 1980. J. Rahn.

Issues on the representation of time and structure in music. 1993. H. Honing.

Music Representation Issues, Techniques, and Systems. 1993. Roger B. Dannenberg

A Max Library for Musical Notation and Computer-Aided Composition. 2015. Andrea Agostini, Daniele Ghisi.

Computation and visualization of musical structures in chord-based simplicial complexes. 2013. Louis Bigo, Moreno Andreatta, Jean Louis Giavitto, Olivier Michel, Antoine Spicher.

Geometric representation and algebraic formalization of musical structures. 2018. A Papadopoulos, L. Pernazza, M. Andreatta, S. Cannas.

International Conference on New Musical Notation Report - General Categories. 1975. Sabbe *et al.*

Smoosic - A JavaScript library for editing and rendering music notation. Copyright (c). 2021. A. D. Newman.