Bitte deutlich leserlich ausfüllen!

Deckblatt einer

wissenschaftlichen Bachelorarbeit

Vor- und Familienname	Matrikelnummer
Aron Rocco Petritz	12008635
Studienrichtung	**Studienkennzahl**
Elektrotechnik-Toningeineur	

Thema der Arbeit:

gemuPG - A Generative Music Poly* Grid

Angefertigt in der Lehrveranstaltung: SE Computermusik und Medienkunst
(Name der Lehrveranstaltung)

Vorgelegt am: 05.03.2025
(Datum)

Beurteilt durch: DI Johannes Zmölnig
(Leiter*in der Lehrveranstaltung)

Abstract

This thesis proposes the term *poly** for inherently polyrhythmic, polymetric, microtonal music software and lists current available software approaches for this category. A new music environment titled *gemuPG* was developed to fit this description and its features and technical implementation outlined. Possible further improvements are reviewed and musical usability is analysed based on select examples, showcasing its capabilities as inherently poly* music software, but also laying out the software's shortcomings.

Zusammenfassung

Diese Abschlussarbeit schlägt den Begriff *poly** für Musiksoftware mit zugrunde liegenden polyrhythmischen, polymetrischen und mikrotonalen Herangehensweisen vor und listet aktuell verfügbare Softwareansätze dieser Kategorie auf. Eine neue Musikumgebung namens *gemuPG* wurde entwickelt, um dieser Beschreibung zu entsprechen. Ihre Funktionen, sowie deren technische Umsetzung, werden behandelt. Ein Überblick über mögliche weitere Verbesserungen wird dargeboten und die musikalische Nutzbarkeit wird anhand ausgewählter Beispiele analysiert, wobei einerseits die Fähigkeiten als poly*-fokussierte Musiksoftware, aber auch die Mängel des Programms aufgezeigt werden.

Contents

1 Introduction

Music notation software like *MuseScore* naturally adheres to traditional western music notation. Similarly, digital audio workstations (DAWs) which have grown from recording and mixing tools to complete compositional environments, inherit the same limitations. A DAW's piano roll holds the same notes as a music sheet's staff lines, all instruments or tracks share the same measure length at any time and subdivisions beyond multiples of two are a specialty, as made evident by their notation. These constraints are often circumvented, but still form the foundation of most tools. This thesis showcases gemuPG, a new music environment with the aspiration to eliminate these constraints in order to allow new approaches to musical exploration and composition.

Chapter 2 first examines existing alternatives to conventional music creation and outlines the design philosophy of gemuPG. The following chapter presents implemented features, but also covers valuable possible improvements to the software. After this, Chapter 5 deals with the technical aspects of implementations as well as current bugs. The last chapter showcases example projects to improve comprehension of workflow and details possible use cases for the software.

2 Poly* Music Software

Computers do not need to read sheet music. We can therefore decouple the process of modern music-making from established standards. The following examples show select approaches to integral poly-metric, poly-rhythmic and microtonal audio composition. I suggest *poly** (pronounced 'poly asterisk', or simply 'poly') as an umbrella term, which does not suit the microtonal aspect semantically but encapsulates the concept adequately. Furthermore, I picked the term generative to describe procedural music as popularised by Brian Eno: "Generative music [...] specifies a set of rules and then lets them make the thing." [1] Many of the following examples are often referred to as 'nonlinear' music software. Although a lot of nonlinear software can also be considered poly* and vice versa, the term poly* differentiates itself by focussing on forms of more granular pitch and time subdivision rather than breaking away from linear structural norms.

2.1 Existing Software

2.1.1 Skeuomorphism

Skeuomorphism describes the software emulation of physical interfaces. This design philosophy aids in bridging the gap between analog and digital interaction and is very prominent in audio software [2]. The most notable group of poly* audio software here is virtual modular synthesis racks like *VCV Rack* [3]. These emulations are flexible in complexity

and friendly to musicians coming from their hardware equivalents, but are often met by initial difficulties for newcomers and do not leverage software interfaces effectively.

2.1.2 Node-Based Approaches

Node-based audio software, such as *Pure Data* [4] or *Bitwig*'s *The Grid* [5], could be considered the real digital derivative of modular synthesisers, offering adaptive interfaces and more suitable controls for computer interaction. Nodes work on different abstraction layers. *Pure Data* offers many low level audio manipulation functions, while *The Grid* focuses on higher level abstractions that ease the initial usage complexity. Thus, node-based tools show an interplay of granularity of control and difficulty of use.

2.1.3 Audio Programming Languages

This category includes *SuperCollider* [6], *ChucK* [7] and their older sibling *Csound* [8]. They are programming languages with the goal of bringing the flexibility of coding to musicianship. These tools are primarily catered towards a certain group of users with prior programming experience wanting to make creative use of their existing skills. Once again, these tools need in-depth preparation to receive satisfying results.

2.1.4 Graphical Sound

Another notable, albeit small, group of software takes the approach of drawn sound, taking inspiration from the ANS Synthesiser and the works of Arseny Avraamov. Software in this field mostly makes use of the spectrogram, allowing users to draw freely within the audible frequency range [9]. While this allows for unrestricted non-discretised placement of notes, current implementations are limited in scope, merely allowing short audio snippets to be created, without multitrack capabilities. Programs like *Photosounder* [10] or *Virtual ANS* [11] are therefore more to be seen as tools or instruments, rather than modern environments for creating unconstrained music.

2.1.5 Other Approaches

Various other approaches have emerged over the years. Notably *Electroplankton*, a 2005 *Nintendo DS* game featuring various different sequencing options, allowing for genuinely poly* approaches to music [12]. *Midinous*, on the other hand, is a sequencer where you connect note blocks on a 2D grid [13]. The distances to one another determine the delay and additional randomisation options and conditional statements allow for complex generative poly* music creation. Lastly, I would like to mention *Blockhead*; a grid-less sample-focused DAW with advanced pitch manipulation and poly-metric and -rhythmic

functionality baked in. Its innovative twist on the traditional DAW layout allows for quick entry and vast sonic experimentation [14].

2.2 A New Environment: gemuPG

2.2.1 The Goal

I desired to create an accessible music environment that is generative and poly* at its core. Additionally it should not mimic physical hardware but instead offer a software-centric interface, that is easy to achieve results with. An approach differing from the ones previously mentioned was sought after. Following some initial brainstorming I landed on the concept of transforming the linearity of tracks in a DAW, which could be considered one-dimensional, into two-dimensional areas; an instrument track usually consists of one or more sound generators (often called virtual instruments) and optional effects. This instrument track needs to be passed a note sequence to output sound. Thus, we can establish the distinction between callers, which are the notes of the sequence, and callee, which is the instrument consisting of generators and effects.

2.2.2 Theory of Operation

This thought process lead me to the concept of a grid-based system, where areas can be defined in various shapes and sizes by connecting several area fields. Within those areas lie our callees. Generator and effect blocks can be placed here to build a virtual instrument. Conversely, the callers, are to be placed around the sides of the areas, which creates a note sequence that loops at a length equivalent to the area's circumference. This forms the foundation of my software, which I named *gemuPG*. The name is an abbreviation of 'GEnerative MUsic Poly* Grid', a self-explanatory name for the software.

Concept art is shown in Figure 1, which shows two areas with surrounding sequence blocks, working as callees. The sequences are read out by stepping around the sides of the areas in counter-clockwise direction at a preset tempo. If there is a sequencer block attached to the current side, the virtual instrument within the area is played at the given pitch(es). If there is no sequencer block however, the instrument is paused. The left area's virtual instrument consists of generator blocks (blue), an effect block (green) and a modulator block (red), while the right one only holds one generator block and two modulator blocks. Modulator blocks can affect settings of adjacent blocks based on fixed waveforms or patterns, but also be influenced by the current sequence value.

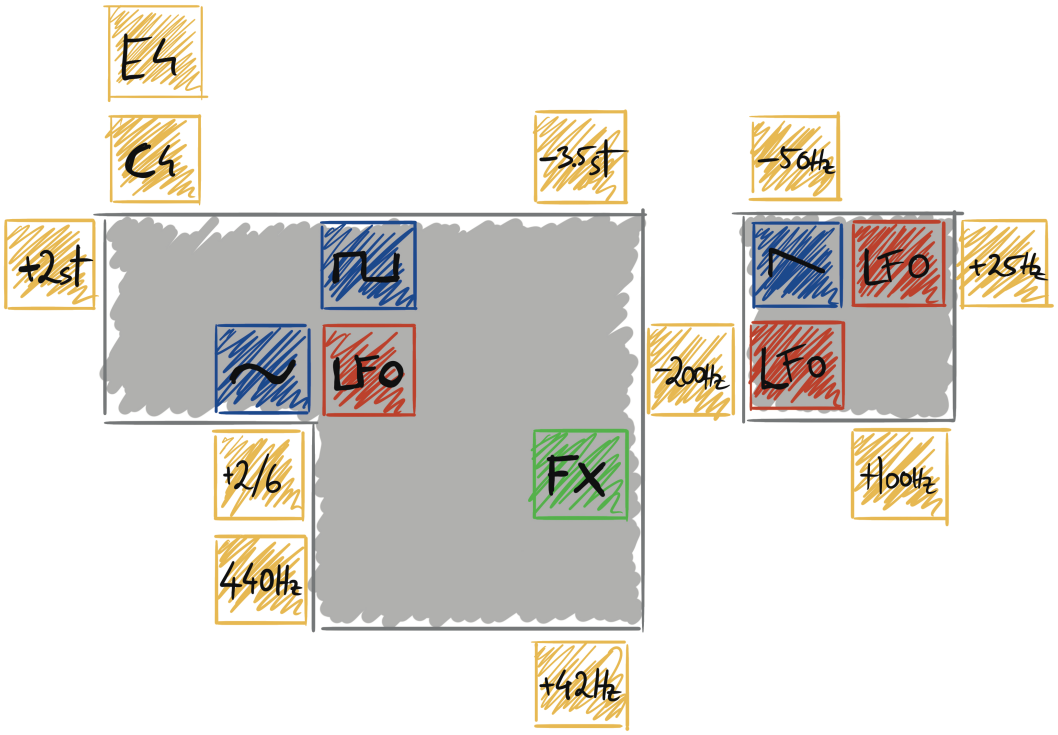


Figure 1: Concept art for gemuPG.

3 Features of gemuPG

When I first set out to create gemuPG I had envisioned a much larger scope. Unfortunately, a lot of ideas had to be dropped - mostly due to time constraints, rather than technical ones. Below you will find a quick rundown of features, separated into those that made it into the final release and those that I considered to be valuable additions but lacked the time to implement.

3.1 Implemented Features

3.1.1 The Grid

Naturally, for the implementation of most other aspects of the software, the grid needed to be implemented first. I also added support for panning and zoom to simplify navigation.

3.1.2 User Interface and Shortcuts

Visualisations and toolbars were added, next to shortcuts, to improve usability. Keyboard shortcuts boost interaction speed, allowing for quicker creative results. The toolbars, on the other hand, enhance accessability by ensuring that interaction with the keyboard is not mandatory.

Figure 2 shows an empty gemuPG project. The vertical toolbar on the left allows for switching between the placement of areas, generator blocks and sequencer blocks. Areas are grey, generators blue and sequencers yellow. In the left corner above, the current block selection is shown. Moving on to the bottom of the screen we find another toolbar, offering play/pause/stop functionality, as well as volume and tempo controls in beats per minute (BPM), which default to 0.5 and 60.0 respectively. The view in the bottom right shows the software's audio output signal. Additionally, blocks and areas have their own settings windows, which will be shown and explained along with their settings in their respective subchapters. All windows show their block's position in their title bar.

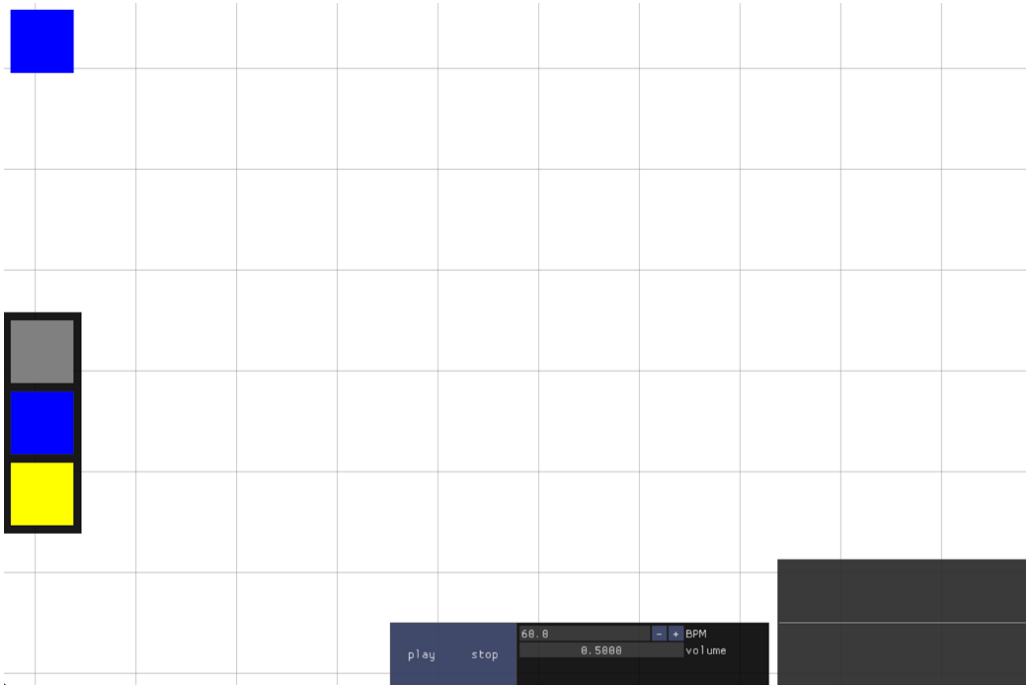


Figure 2: An empty gemuPG project.

3.1.3 Areas

As previously established, areas need to hold virtual instruments and be surrounded by sequence blocks to output sound. Automatic merging and splitting needed to occur if connecting areas were added or removed. Furthermore, beat subdivisions (supporting all whole numbers from 1 to 32) were implemented, allowing for complex rhythmic experimentation, as well as an area-wide amplitude setting to control the volume of all contained generator blocks at once. To improve sound sculpting capabilities, settings for glissandi between notes, as well as for attack and release times were added. These are controlled as a percentage of note length. The window shown in Figure 3 displays the default settings for areas. With a default global BPM of 60.0 and a subdivision of '1/4' we get a note length of 0.25 s. If we consider the default attack and release values of 25%, this leads to attack and release times of $250 \text{ ms} \cdot 25\% = 62.5 \text{ ms}$. Hence, the amplitude reaches its

maximum value 62.5 ms after the note begins playing and starts to be reduced 62.5 ms before the next note starts. The glissando works in a similar fashion to the attack time, starting at the beginning of a note and sliding down to the new frequency in a variable percentage of the note length's time.

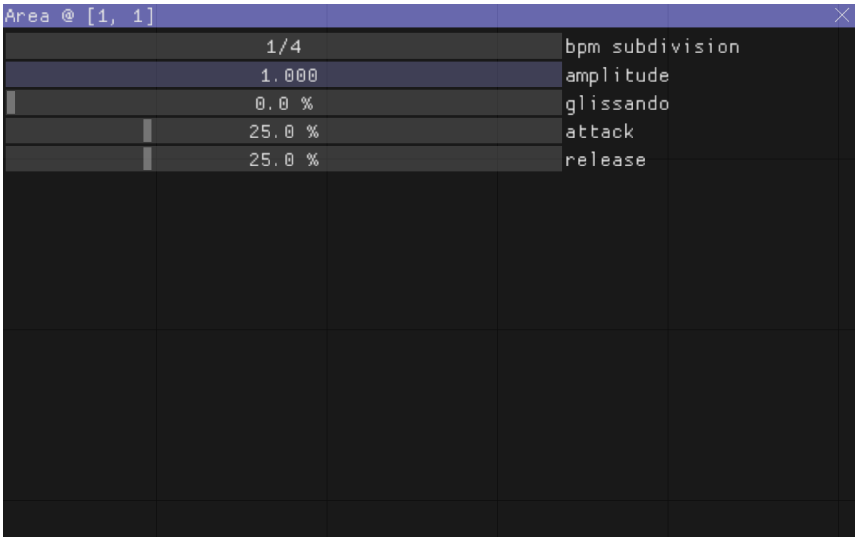
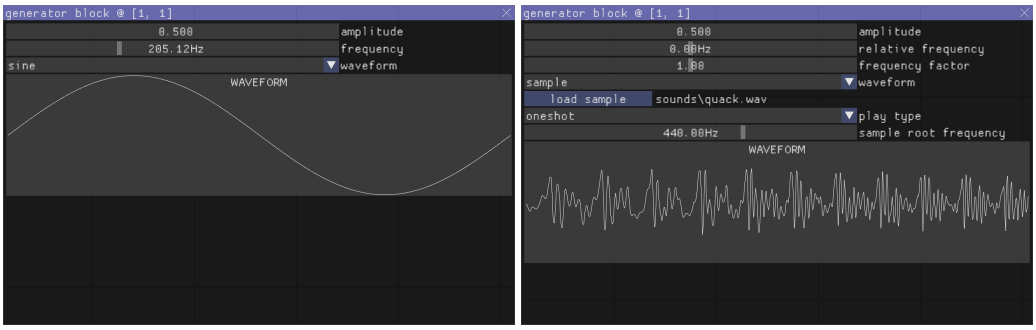


Figure 3: The settings window of an area.

3.1.4 Generators

Moving on to the sound generators, I decided not to limit their placement to areas, but instead also allow for global blocks that play a continuous note. These global blocks only show options for setting the frequency, amplitude and waveform, as shown in Figure 4a. Frequency and waveform are set to be randomised upon placement. Generator blocks within areas, however, replace the fixed frequency setting with 'relative frequency' and 'frequency factor' controls which affect the output frequency according to $f_{out} = (f_{sequence} + f_{relative}) \cdot factor$. This enables additive synthesis using multiple generator blocks within areas. The waveform selection supports basic waveforms, more precisely: sine, square, triangle and saw waves. Moreover, sample loading with support for WAV files of various formats was added. Selecting the 'sample' option enables controls for loading a file, as well as a toggle for selecting whether the sample should be played once ('oneshot') or looped ('repeat'). The previously mentioned randomisation only chooses one of the preset basic waveforms however. Lastly, the loaded sample can be tuned using the 'sample root frequency' setting. Successive sequencer blocks that result in the same frequency can be used to let the sample play out instead of retriggering it, if the sample's length exceeds the note length. The current file name is also shown next to the 'load sample' button. The settings window for a generator block within an area, with its waveform set to a sample named 'quack.wav' is shown in Figure 4b.



(a) A global sine generator block. (b) A sample generator block in an area.

Figure 4: The settings windows of generators blocks.

Quick readability of block settings was important to me, which is why rendering of relevant information directly onto the blocks on the grid was implemented. The type of waveform is shown in large, while frequency information is shown on top. Figure 5 shows two generators blocks, as rendered on the grid. The left one holds a sample and is placed in an area. It shows a frequency factor of 1.75 and a relative frequency of 6.75 Hz. The right generator holds a square wave and is placed globally, showing an absolute frequency of 220.0 Hz. The font used is *Sono* by Tyler Finck [15].

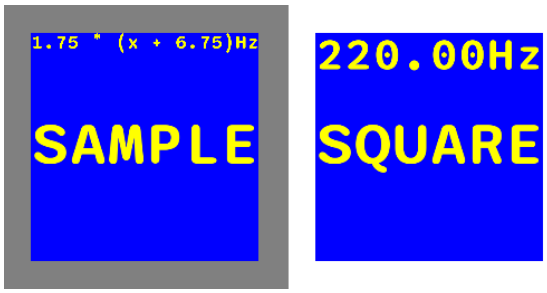


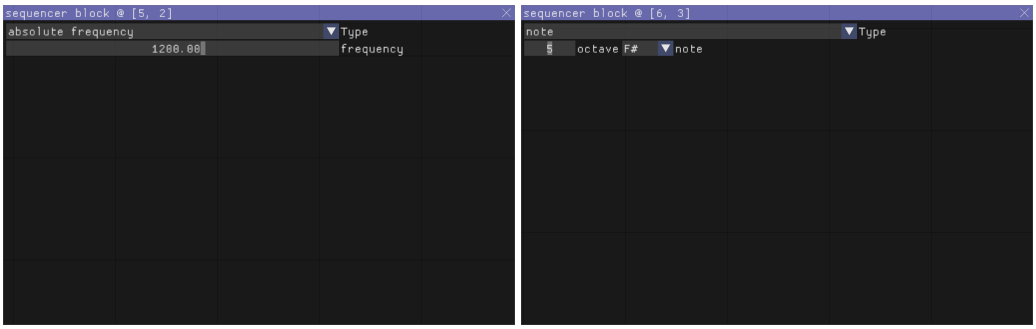
Figure 5: Display of generator blocks in the grid.

3.1.5 Sequencers

The sequencer blocks' placement is limited to positions orthogonally adjacent to the outer area positions. Additionally to this, automatic removal of sequencer blocks, when the supporting area is removed, was implemented. The sequencer blocks come in four different types:

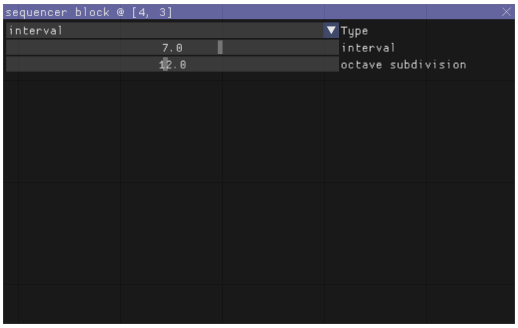
- 'absolute frequency'
- 'relative frequency'
- 'note'
- 'interval'

Absolute and relative frequency sequencers are self-explanatory. The window for a sequencer block with the 'absolute frequency' type selected shows a single slider for frequency selection underneath the 'type' drop-down menu, an example of which can be



(a) An 'absolute frequency' sequencer.

(b) A 'note' sequencer.



(c) An 'interval' sequencer.

Figure 6: The settings windows of sequencer blocks.

seen in Figure 6a. Relative frequency blocks similarly show a single frequency slider, albeit with different ranges to accommodate for negative frequency changes. Sequencer blocks of type 'note' allow for selecting notes from the conventional twelve-tone scale in octaves 0 through 10. An example window for F#5 is shown in Figure 6b. The fourth sequencing option is 'interval', which offers two control sliders, as seen in Figure 6c. The 'octave subdivision' input determines the amount of equal temperament frequency steps per octave. The other input, named 'interval', determines how many of these equal temperament steps should be moved up or down. This equates to $f = f_{prev} \cdot 2^{\frac{interval}{octave\ subdivision}}$. Importantly with relative sequencer blocks (as in 'relative frequency' or 'interval' sequencer blocks), the frequency changes were implemented in a way that the frequency range wraps from 20 Hz to 20 kHz. This was added to better support sequences with exclusive use of relative sequencer blocks.

The type and value(s) of the sequencer blocks are randomised within preset ranges when first placed. Direct rendering of the sequencer blocks' settings is also given, similar to generator blocks. Figure 7 shows all available block types and their on-block renderings. The interval block is the currently active one, as signalised by its enlarged block size. Sequencers are stepped in counter-clockwise direction for each side of an area. Because the sequence length is not equal to the amount of possible sequencer block positions around an area but rather the amount of sides of the area shape, it causes sequencers placed in a corner of an area to play twice and sequencers in the gap of a U-shaped area three times. If a side is left empty it counts as a pause. Illustrations visualising the sequence

order can be found in Chapter 4.2.

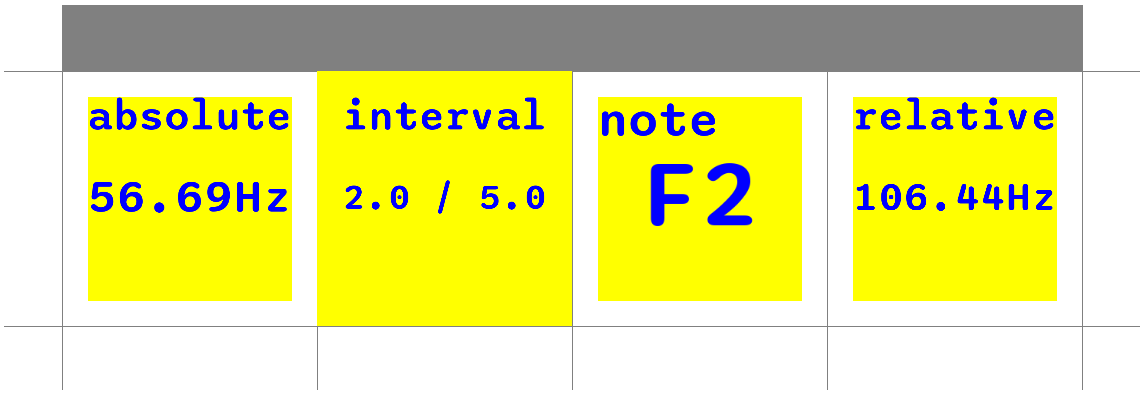


Figure 7: Randomised sequencer blocks of all available types.

3.1.6 Other Functionality

Saving and loading projects was implemented, a substantial addition for making the use of gemuPG worthwhile. The project files are stored in JSON format with a '.gemupg' extension, which allows for direct manipulation of save files thanks to the format's human-readability. On a related note, project loading via launch parameters is supported, allowing operating systems to set the project files to directly open with gemuPG. The play, pause and stop functionality is another valuable addition. The stop button halts sequencer progression and resets each area's sequence to its initial position. The initial position is the sequencer block left of the top-left-most area block, meaning the left-most area block of the top-most row of the area. The last addition, which improved usability drastically, was the ability to copy and paste blocks. This allows for very fast sequence programming by copying sequencer types with specific settings.

3.2 Omitted Features

The first obvious omission from the project is the effect blocks mentioned in Chapter 2.2. An earlier version, before rewriting the software from scratch, included some backend support for this. They did not make it back into the thesis release version, because I failed to find a satisfying translation for serial processing to the 2D space created by gemuPG. Generator blocks did not pose a problem, since they do not need to adhere to an order, seeing as they effectively run in parallel. In contrast, a way of sorting the order of effects processing would need to be found for placed effect blocks. From left to right and top to bottom did not seem intuitive enough given the otherwise more circular nature of signal flow. If this hindrance were to be solved, however, it was conceived that various effects would be available, which would affect their respective areas as a whole. Furthermore,

lua scripting capabilities for effect blocks were considered to allow for easy expandability of effect types.

Another feature that failed to make it into gemuPG but was previously mentioned and seen in Figure 1 is modulator blocks. These were meant to modify variables of adjacent generator and effect blocks using different patterns or waveforms. Their frequency could either be fixed or relative to the current sequencer block's value. Additionally, a modulator block's variables could be controlled by another modulator block, allowing for frequency or ring modulation. Modulator blocks would reinforce the two-dimensional design philosophy. Every modulator block could affect up to four adjacent blocks, in theory allowing for complex synthesis structures.

The concept art also shows an instance of two sequencer blocks stacked on one another (C4 and E4). These parallel sequencer blocks would allow for chord building, once again advancing the possibilities of composition within the environment. The calculation order for relative sequencer blocks would intuitively start from the side of the area, going outwards.

With the aim of enhancing and actively supporting live playability of gemuPG, mute groups were considered. I wanted to add the ability to toggle bypassing of certain areas or blocks using the number keys, if previously mapped in their respective options. A performer could effortlessly switch between various different poly* patterns or change the underlying instruments with this.

Next, conditionals would allow generator or sequencer blocks to activate randomly or by a set of rules. For example a generator block could be set to only play if the given frequency is above a certain value. Additionally, they are meant to support sequence modification, such as skipping a certain amount of sequencing steps if a requirement is met or changing direction of the sequence. This addition would greatly improve possible compositional complexity.

The dropped features I mentioned thus far would be the main additions I would make to consider gemuPG complete. However, contemplation was given to other features as well, such as relative BPM adjustments for areas, audio panning, more advanced ADSR capabilities, sample loop markers and in-software audio recording for exporting, as well as sample creation, to list a few. Audio plugin support for generators and effects would also immensely broaden the software's capabilities, due to the enormous amounts of software found in this area. In reality, of course, there are nearly infinite ways of improving creative software and I am but one person with a questionable understanding of time.

4 gemuPG in Use

4.1 Controls

The controls for gemuPG were chosen to be intuitive. The number keys 1 through 3 select the tool for placement, removal or editing. The numbers are mapped to areas, generators, and sequencers respectively, which mimics the order of the left toolbar. The selected area or block can be placed using the left mouse button, or removed using the right mouse button. A block's menu opens when it is left clicked, which allows for double clicking an empty space to create and immediately edit a block. Camera movement is possible through dragging the screen using the middle mouse button and additionally, for those on touchpads or without mice with a third button, using the left mouse button while holding down the Ctrl key. Zooming is achieved through the scroll wheel. Sliders in block menus can be left clicked while holding the Ctrl key to enter values with the keyboard. Since I noticed that block windows can accumulate quite quickly when making changes, the Q and Escape keys were mapped to to closing all open windows. Ctrl + C and Ctrl + V are used for copying and pasting blocks respectively and the D key can be used toggle the audio output view. F11 toggles fullscreen mode. The conventional keyboard shortcuts for saving and loading have also been assigned as seen in Table 1.

Shortcut	Function
1	Select Area Blocks
2	Select Generator Blocks
3	Select Sequencer Blocks
Left Mouse Click	Place Block
Left Mouse Click	Edit Block
Right Mouse Click	Remove Block
Middle Mouse Drag	Pan Camera
Ctrl + Left Mouse Drag	Pan Camera
Scroll Wheel	Zoom Camera
Ctrl + C	Copy Block
Ctrl + V	Paste Block
Ctrl + S	Save Project
Ctrl + Shift + S	Save Project As...
Ctrl + O	Open Project
Ctrl + N	New Project (Warning: No Save Prompt!)
Q	Close All Block Windows
Escape	Close All Block Windows
D	Toggle Audio Output View
F11	Fullscreen
Ctrl + Left Mouse Click on Sliders	Enter Value via Keyboard

Table 1: Keyboard shortcuts and mouse controls for gemuPG.

4.2 Examples

The following examples demonstrate the usage of gemuPG. All examples are available in the project's Git repository.

4.2.1 How Sequences Work

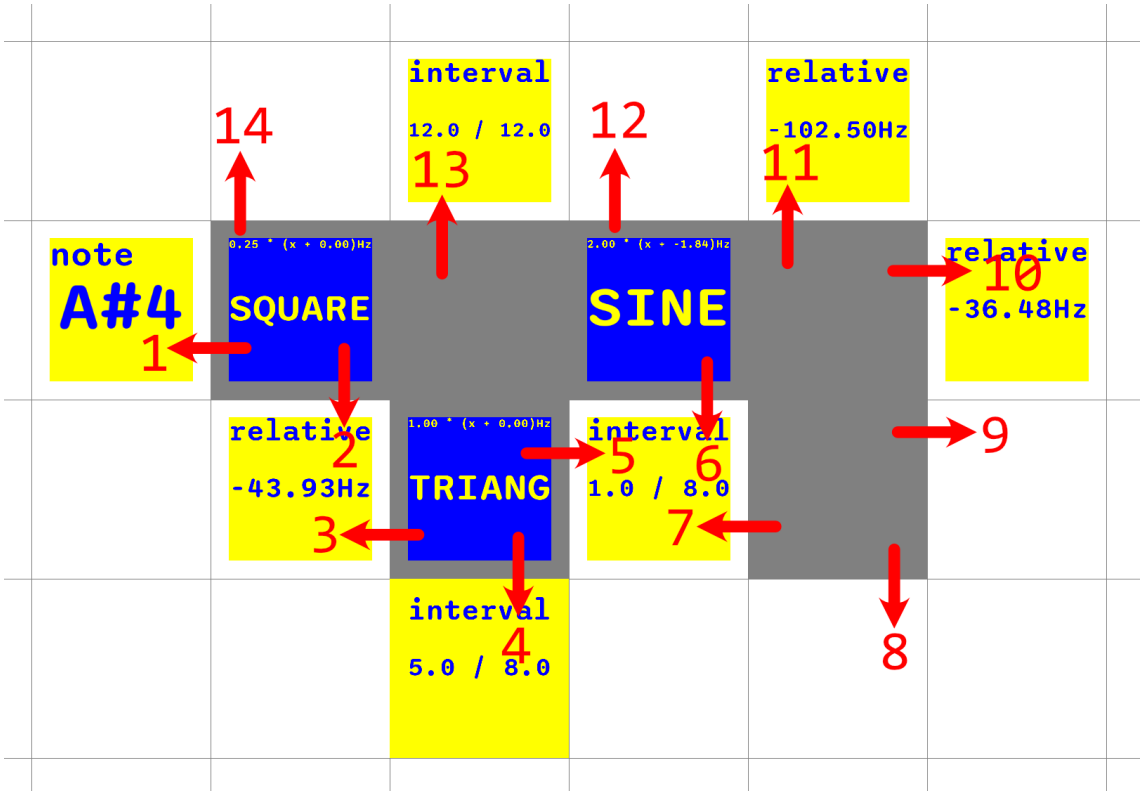


Figure 8: Example 01, showing an area with a visualisation of the sequencing order.

To further illustrate the sequencing order, I will now explain how the area shown in Figure 8 plays out. The sequence starts at the 'note' block that defines A#4 as its pitch, which internally gets translated to a frequency of 466.24 Hz. This is followed by a 'relative' sequencer block playing twice because it is placed in a corner of the area and thus being activated by two sides, changing the pitch to 422.31 Hz and 378.38 Hz. The interval block on the bottom multiplies the frequency by $2^{\frac{5}{8}}$ resulting in 583.54 Hz. Being adjacent to three sides of the area, the next sequencer block is being played three times. Please refer to Table 2 for the remaining frequency values. Following the three $1.0 \,/\, 8.0$ interval increments are two pauses, as shown by a lack of sequencers. The remaining five sides of the area are evaluated in the same fashion, after which the sequence repeats from the start. This results in a repeating pattern of length 14, which triggers the generator blocks within the area. The generators shown provide us with a square wave at a quarter of the given frequency, a triangle wave without any relative changes to pitch, as well as a sine wave with a frequency offset of -1.84 Hz and a multiplication factor of 2. The amplitudes

of the generator blocks and the area settings cannot be interpreted from this screenshot, but can be examined in this example's project file in the gemuPG code repository.

n	sequencer type	value	frequency	note
1	note	A#4	466.24 Hz	A#4
2	relative	−43.93 Hz	422.31 Hz	G#4 + 29ct
3	relative	−43.93 Hz	378.38 Hz	F#4 + 39ct
4	interval	5.0 / 8.0	583.54 Hz	D5 - 11ct
5	interval	1.0 / 8.0	636.35 Hz	D#5 + 39ct
6	interval	1.0 / 8.0	693.95 Hz	F5 - 11ct
7	interval	1.0 / 8.0	756.76 Hz	F#5 + 39ct
8	-	-	pause	-
9	-	-	pause	-
10	relative	−36.48 Hz	720.28 Hz	F#5 - 47ct
11	relative	−102.50 Hz	617.78 Hz	D#5 - 12ct
12	-	-	pause	-
13	interval	12.0 / 12.0	1235.56 Hz	D#6 - 12ct
14	-	-	pause	-

Table 2: Note sequence for example 01 with corresponding 12-tone scale note values.

4.2.2 Polymetric Composition and Shared Sequencer Blocks

Polymetric composition is natural and straightforward in gemuPG, since each area can be thought of following its own time signature. If the circumference of one area is not the multiple of another, a polymetre is formed. Take for example an interplay of $\frac{10}{8}$ and $\frac{6}{4}$ metres. This example's convergence period, meaning the time it takes for the combined pattern to repeat, is twelve quarter beats or two iterations of the $\frac{6}{4}$ pattern. We can create a 1x2 area, which has six sides with a default subdivision value of '1/4', and an area with ten sides, with its subdivision changed to '1/8'. For this example an L shape was chosen. Figure 10 shows one possible realisation, which also makes use of shared sequencer blocks in the middle. Interestingly, the shared multi-block subsequence of one area is always the reverse of the same subsequence of another area. The shown example intentionally makes exclusive use of the twelve-tone scale to allow for easy transcription to conventional staff notation, seen in Figure 9.

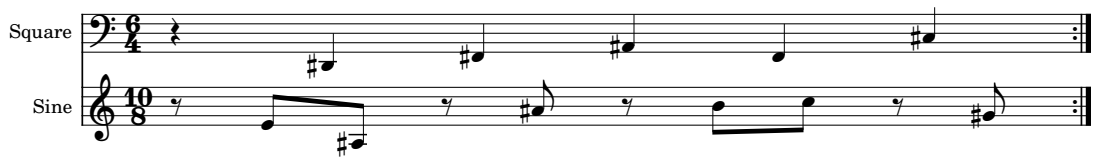


Figure 9: A transcription of example 02.

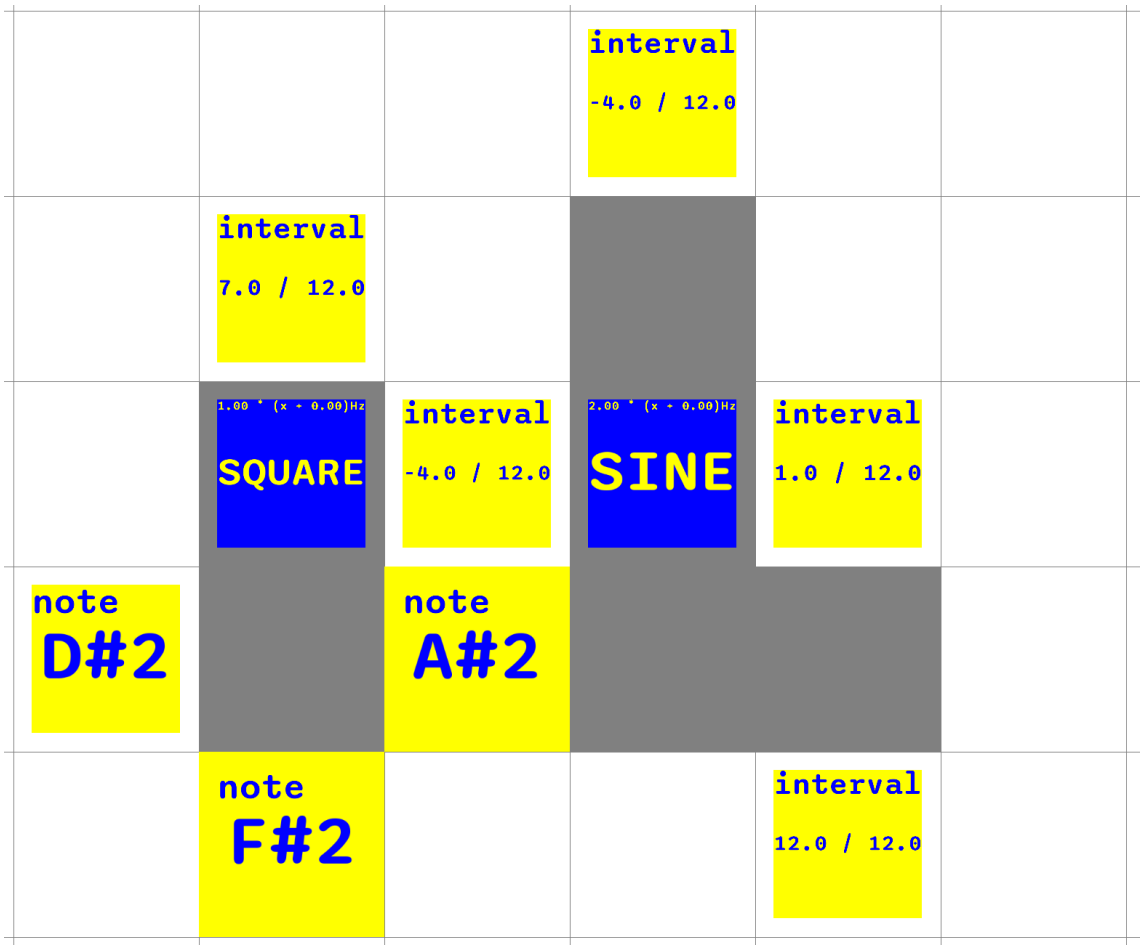


Figure 10: Example 02, showing the realisation of a polymetre.

4.2.3 Polyrhythmic Composition

Polyrhythmic composition is similarly easy to achieve using the area's subdivision setting. The example in Figure 12 shows a 4:5 (four over five) polyrhythm, realised by setting the left area's subdivision setting to '1/4' and the right one's to '1/5'. The combined pattern repeats after four iterations of the left area, or five of the right one. The transcription is shown in Figure 11.

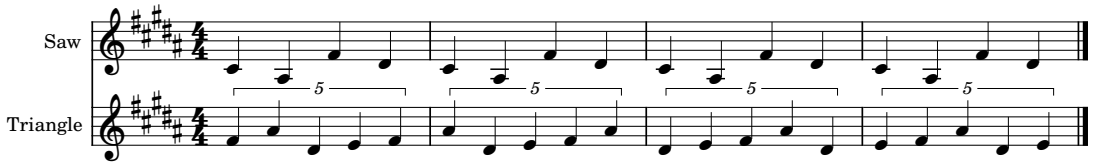


Figure 11: A transcription of example 03.

Figure 12: Example 03, showing the realisation of a polyrhythm.

4.2.4 Generative Music

The previous examples covered different aspects of poly* music making. While polymetres of a sufficiently sized convergence period already approximate generative music, I will now illustrate how gemuPG can also be used for other rudimentary generative composition using the example shown in Figure 13. This project shows two areas, one with exclusive use of interval sequencer blocks, the other only with sequencer blocks of type 'relative frequency'. Considering the frequency value wrap within a range of $[20\text{ Hz}, 20\text{ kHz}]$ these are two infinite sequences with different starting notes for each iteration. I will start with the left sequence which, excluding pauses, has nine sequencing steps, each changing the pitch by a factor of $2^{\frac{x}{16}}$:

$$f = f_{last} \cdot \prod_{i=1}^{9} 2^{\frac{x_i}{16}} = f_{last} \cdot 2^{\frac{\sum_{i=1}^{9} x_i}{16}} = f_{last} \cdot 2^{\frac{-6}{16}}$$

Each iteration of the left area causes a pitch shift by a factor of $2^{\frac{-6}{16}}$ and therefore transposes the next iteration down six steps on a 16-tone equal temperament scale.

In contrast, the right area uses relative frequencies, the pitch difference after each iteration thus equates to:

$$f - f_{last} = +200.00\text{ Hz} + 25.00\text{ Hz} - 2 \cdot 300.00\text{ Hz} - 50.00\text{ Hz} = -425.00\text{ Hz}$$

Naturally, using interval sequencers results in exponential pitch changes, while the relative frequency sequencers work linearly, yielding dissimilar outcomes.

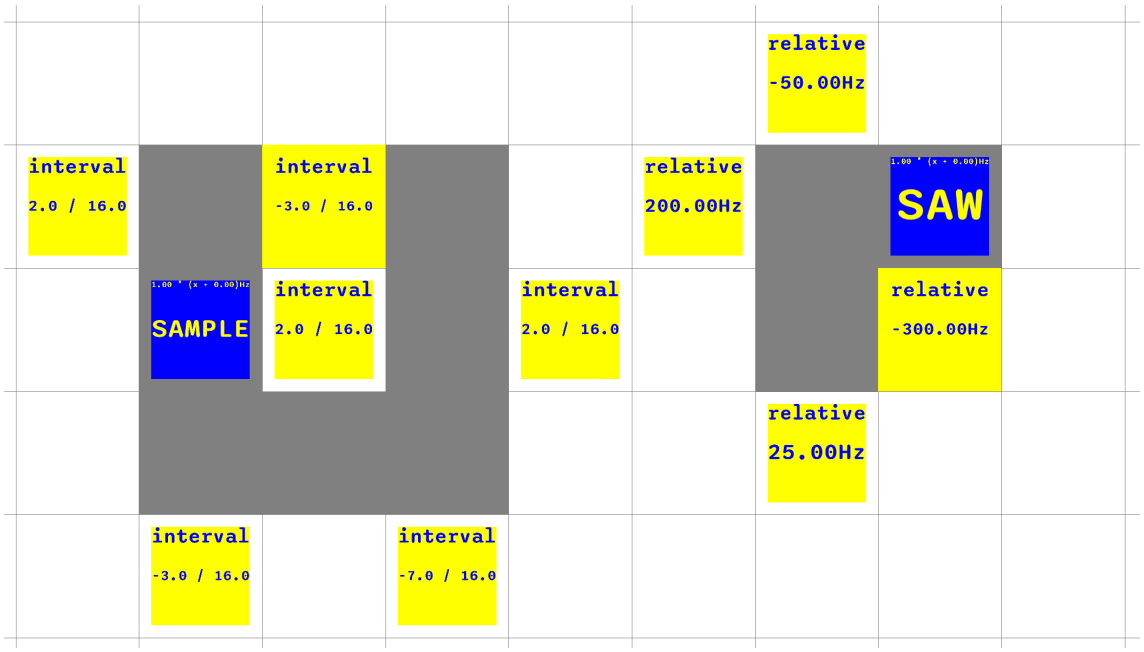


Figure 13: Example 04, showing generative music.

4.3 Instrumental Idioms

Instrumental idioms have been described by David Huron and Jonathan Berec as follows: ”In the case of instrumental idiomaticism, we noted that idiomaticism might be defined as the degree to which a given means of achieving a certain musical goal is significantly easier than other hypothetical means” [16]. This leads to the question of what idiomatic opportunities and limitations gemuPG provides. I have therefore summarised the easy to achieve musical structures into Chapter 4.3.1 and the difficult ones into Chapter 4.3.2. I chose the tiles *Affordances* and *Constraints* loosely based on their definitions in Don Norman’s *The Design of Everyday Things* [17].

4.3.1 Affordances

Through testing the software during development several patterns became evident. As intended, polymetric music was typically created unless the areas’ circumferences were intentionally matched. The added complexity of different shapes of the same area position count naturally resulting in different circumferences supports this further. Contrary to this, The polyrhythmic aspect, being ’hidden’ in a menu, is perceived as an option rather than a natural occurrence. Furthermore, the randomisation of blocks encourages quick experimentation and a ’seeing what sticks’ approach. This, in combination with the occasional unpredictability of results from area shapes - regarding sequence length and sequencer block repetition in corners - can lead to happy accidents, which arguably set the cornerstone of gemuPG. The looping structure of gemuPG leads to a strong tendency for repetitive music, even if not fully repeating due to the underlying polymetric design. This

lends itself to composition of minimal music, sound scapes and similar forms without strong musical motion. Alternatively, gemuPG can also be used as an instrument, rather than a compositional tool, where the output is recorded and combined with other tools in a DAW.

4.3.2 Constraints

The central role of loops limits support for any advanced musical forms. Although sufficiently large areas that allow for these forms could be declared, it would directly oppose gemuPG's usability, making traditional music software the better alternative. In practice I often resorted to heavy use of note and interval sequencer blocks, a tendency probably influenced by traditional notation. Considerations were therefore made to revise the 12-tone note selection to conform to gemuPG's poly* design philosophy: the octave selection would remain, but the note selection setting would be replaced by options to choose the octave subdivision and the step of that octave.

Another, albeit smaller, constraint is the fact that adding or removing area blocks always causes a change of area circumference of ± 2 or 0. In turn, this infers the need to double the sequence for uneven sequence lengths.

Unfortunately, in part due to the lack of conditionals and modulator blocks, the generative music aspect of gemuPG falls short. Unless manually interacted with, iterations vary by transposition at most, strongly limiting complexity of projects.

5 Technical Implementation

5.1 Language, Framework and Toolchain

As for any software project the programming language first needed to be chosen. I opted for C++ due to familiarity with the language, as well as its widespread use in the field of audio programming. Next, I needed a framework to allow cross-platform interaction with input, audio and graphics hardware. Here, my choice was split between *JUCE* [18], a widely used framework that is explicitly made for audio programming, or *Simple DirectMedia Layer* (SDL) [19], an even more widespread but also lower level framework than JUCE. Both of these frameworks feature an open source license. Ultimately I settled on using SDL, because I valued its wider usability higher than JUCE's suitability for this use case. Version 3 of SDL (SDL3) was available as a stable preview when I started this project and made its official stable release with version 3.2.0 a couple of weeks before writing this. SDL3 was used for gemuPG as the new version added significant improvements to its audio system. Additionally, I used the SDL external *SDL_ttf* [20] for rendering the on-block information and the user interface library *Dear ImGui* [21] for

implementing the toolbars and settings windows. Lastly, Niels Lohmann's single-header JSON library was used for implementing the save and load functionality. [22]

I programmed and tested my software in Windows 11 using GCC 14.2.0 via MSYS2's UCRT64 environment and CMake 3.31.5. The minimum C++ standard library version has been set to C++20, due to active use of `std::format`. This also makes GCC version 13 the minimum requirement for compilation. The minimum CMake version has been set to 3.25. The software has also been compiled and tested under Ubuntu 24.04.2 LTS. Compilation using MSVC 17.12 was also successful but the resulting executable was prone to crashing. Compilation with clang is untested. The source code of gemuPG is available under the GNU AGPLv3 license in the Git repository from the *Institute of Electronic Music and Acoustics*: https://git.iem.at/aronpetritz/gemupg/-/releases/Thesis

5.2 Software Structure

SDL3 added so-called 'main callbacks' splitting functionality into the following main functions:

- `SDL_AppInit` is called at the start of the program
- `SDL_AppQuit` is called upon closing the program
- `SDL_AppEvent` is called asynchronously for each registered event
- `SDL_AppIterate` is the main update loop

`SDL_AppInit` and `SDL_AppQuit` were naturally used for initialisation and memory clean-up. Here I also initialised an `AudioEngine` singleton class among other things, which in turn starts the audio stream and grabs data from the assigned audio callback function. I will detail this in the Chapter 5.2.3.

The event callback forwards most events to an `Input` singleton class, where interaction with Dear ImGui windows and other classes is handled. Controls for panning and zooming are sent to the `Camera` singleton for example, which is later accessed by the rendering code.

The core software loop `SDL_AppIterate` separates code for sequence stepping and updating the screen using a `Clock` singleton, as shown in the code snippet below.

```
SDL_AppResult SDL_AppIterate(void *appstate) {
 if (Clock::getInstance().shouldStep())
  Interface::getInstance().getGrid().stepSequence();

 if (Clock::getInstance().shouldDraw())
  Interface::getInstance().draw();

 return SDL_APP_CONTINUE;
}
```

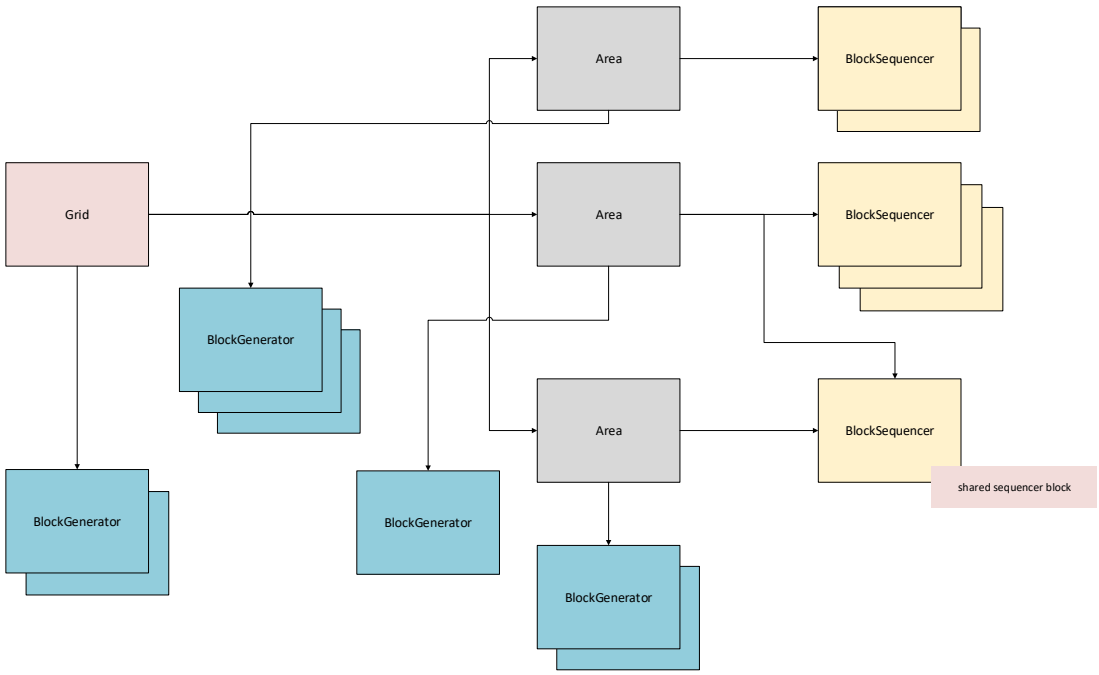


Figure 14: A rough illustration of gemuPG's class relationships.

This separation should prioritise sequence stepping to graphics rendering in the hopes of maintaining tempo stability.

Figure 14 illustrates the class relationships of objects within the grid. It shows that each `Area` object can hold several `BlockGenerator` objects, but each `BlockGenerator` object can only be held by one `Area`. Additionally, the `Grid` object can also directly store `BlockGenerators`. In contrast, `BlockSequencer` objects can be referenced by multiple areas, as illustrated by the bottom `BlockSequencer`.

5.2.1 Area Merging and Splitting

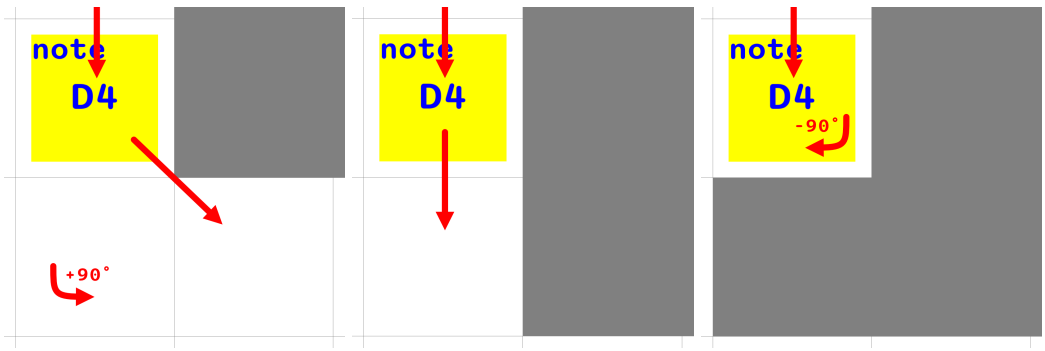
Areas need to automatically split and merge. Hence, when an area is placed, it first checks all adjacent positions for other areas. The newly placed area is then added to the first discovered area. This area is marked and all positions and blocks held by further areas are moved over to the marked area, after which the other areas are removed from the grid. If there are no adjacent areas, a new one is created. Conversely, splitting areas was implemented via a flood fill algorithm that checks which fields of an area are still connected after a position is removed.

5.2.2 Sequencer Stepping

Areas store pointers to sequencer blocks and pauses are marked by `nullptr`. Sequences need to be sorted and updated when the area shape changes or sequencer blocks are added or removed. Modifying an area in these ways therefore calls an `updateSequence()` function, containing the following instructions:

1. Move to the starting position and set direction to 'DOWN'.
2. Add reference to `BlockSequencer` or alternatively `nullptr` if there is no sequencer block.
3. Set the next sequence position:
   (a) If no area exists at the next diagonal inwards position, move to that position and rotate the direction by $90°$.
   (b) Otherwise, if no area exists at the position straight ahead, move to that position.
   (c) If both positions are occupied, rotate the direction by $-90°$.
4. Repeat steps 2 and 3 until starting position is reached again.

Step 3, determination of the next sequence position is shown in Figures 15a, 15b and 15c respectively. Each `stepSequence()` call, areas acquire their next sequence block's frequency and apply it to all generator blocks within. If the next sequence position shows no sequencer block, the frequency is set to 0.0 Hz.



(a) Diagonal movement. (b) Straight movement. (c) Rotation only.

Figure 15: Determining the next sequence position.

5.2.3 Audio Engine

Audio processing is handled via the `AudioEngine` singleton class, which upon starting gemuPG initialises the SDL audio backend and starts the main output stream. It also initialises wavetables for the basic waveforms using fourier addition with different amounts of harmonics to minimise aliasing. The wavetables are stored in the `wavetables_` map, which contains `std::pair<WAVE_FORMS, int>`s as its keys, where the `int` value sets the amount of harmonics, and `std::array<float, WAVE_SIZE>`s as their val-

ues. `WAVE_FORMS` is an enum holding the selectable waveforms and `WAVE_SIZE` is a `constexpr` variable defining the size of a wavetable. This pre-initialisation is important for quick writing to the audio stream. The main audio callback function calls `SDL_GetAudioStreamData(...)` on all generator blocks, which pulls samples from their individual audio callbacks and sums them up. Additionally, a simple low-pass filter was implemented using a canonical second-order structure [23], which is applied at the end of the function. The streams contain 32-bit float values.

If a basic waveform is selected, the respective generator's audio callback reads from `AudioEngine::getWaveTable(WAVE_FORMS, pitch_t)` that returns a band-limited wavetable based on a given frequency. If a sample is selected, it reads from the array stored within the block's `Sample` object instead. The audio data is converted to the output's sample rate and into 32-bit float values upon loading. Sample repitching is implemented via simple playback speed variation. The array index is determined via `getPhase()`, which is implemented as $\phi = \phi + \frac{f}{f_s}$ for preset waveforms and as $\phi = \phi + \frac{f}{f_{root} \cdot sample\_length}$ for loaded samples, both wrapping around $[0.0, 1.0]$. The phase is then multiplied with the respective size of the read array to receive a floating point index. An `interpTable(...)` function linearly interpolates values between samples. For samples, a change of pitch additionally resets the phase to 0.0 to preserve transients.

5.3 Performance Limitations and Bugs

Very late I realised that the main update loop `SDL_AppIterate` is called at the refresh rate of the display. This limits the precision of the implemented clock and thus the accuracy of fast sequence stepping. The clock would have to preferably run on a separate thread to fix this. Moreover, since the main callback structure of SDL3 is inherently multithreaded, along with the audio callbacks, variable locking is important for avoiding race conditions. This, however, was not implemented due to concerns of blocking the audio thread, which leads to occasional crashes.

Lastly, audible artifacts exist for samples played at high frequencies, which exposes the insufficiency of the implemented filter. A steeper filter structure is needed here.

6 Conclusion

In comparison to other music software in the generative, polymetric, polyrhythmic and microtonal music space, gemuPG offers several novel ideas that lead to new compositional approaches. The current state of the software provides a functional foundation and successfully delivers a natural poly*-centric environment for music creation. However, the absence of many mentioned features, the implementation of which would allow for greater musical complexity, hinders the desired big leap in sonic exploration. Future development should prioritize integrating these omissions, particularly modulator blocks and conditional statements, which would allow for more nuanced generative variation. Furthermore, an overhaul of the 'note'-type sequencer that avoids the twelve-tone scale as a standard would align more with the microtonal aspect of the design philosophy and encourage stronger deviation from traditional composition. Altogether, gemuPG can be seen as a playground for new forms of musical expression with a lot of room for improvement.

References

[1] Brian Eno. *"Evolving metaphors, in my opinion, is what artists do."* 1996. URL: https://www.inmotionmagazine.com/eno1.html (visited on 02/22/2025).

[2] John Lagomarsino. *Why are there so many knobs in GarageBand?* 2017. URL: https://theoutline.com/post/2157/why-are-there-so-many-knobs-in-garage-band (visited on 02/21/2025).

[3] *VCV Rack*. URL: https://vcvrack.com/ (visited on 02/28/2025).

[4] *Pure Data*. URL: https://puredata.info/ (visited on 02/28/2025).

[5] *Bitwig's The Grid*. URL: https://www.bitwig.com/the-grid/ (visited on 02/28/2025).

[6] *SuperCollider*. URL: https://supercollider.github.io/ (visited on 02/28/2025).

[7] *ChucK*. URL: https://chuck.cs.princeton.edu/ (visited on 02/28/2025).

[8] *Csound*. URL: https://csound.com/ (visited on 02/28/2025).

[9] Lavoslava Benčić. "GRAPHICAL SOUND - FROM INCEPTION UP TO THE MASTERPIECES". In: *Academia Letters* (2021). URL: https://doi.org/10.20935/AL1108 (visited on 02/22/2025).

[10] Michel Rouzic. *Photosounder*. URL: https://photosounder.com/ (visited on 02/28/2025).

[11] Alexander Zolotov. *Virtual ANS*. URL: https://warmplace.ru/soft/ans/ (visited on 02/28/2025).

[12] Craig Harris. *Electroplankton*. 2006. URL: https://www.ign.com/articles/2006/01/11/electroplankton (visited on 02/28/2025).

[13] *Midinous*. URL: https://midinous.com/ (visited on 02/28/2025).

[14] Matt Mullen. *This 'experimental' new DAW has a ton of innovative features that we haven't seen anywhere else: "This is something that should have existed for years"*. 2023. URL: https://www.musicradar.com/news/experimental-daw-blockhead (visited on 02/28/2025).

[15] Tyler Finck. *Sono*. 2021. URL: https://www.dafont.com/de/sono.font (visited on 02/23/2025).

[16] David Huron and Jonathan Berec. "Characterizing Idiomatic Organization in Music: A Theory and Case Study of Musical Affordances". In: *Empirical Musicology Review* (2009). URL: https://doi.org/10.18061/1811/44531 (visited on 02/25/2025).

[17] Don Norman. “The Design of Everyday Things”. In: Basic Books, 2013, pp. 72–73.

[18] *JUCE*. URL: https://juce.com/ (visited on 02/27/2025).

[19] *Simple DirectMedia Layer*. URL: https://libsdl.org/ (visited on 02/27/2025).

[20] *SDL_ttf*. URL: https://github.com/libsdl-org/SDL_ttf (visited on 02/27/2025).

[21] Omar Cornut. *DearImgui*. URL: https://github.com/ocornut/imgui (visited on 02/27/2025).

[22] Niels Lohmann. *JSON for Modern C++*. URL: https://json.nlohmann.me/ (visited on 02/27/2025).

[23] “DAFX: Digital Audio Effects”. In: ed. by Udo Zölzer. John Wiley & Sons, Inc, 2002, p. 34.

Appendix: Use of AI

The AI *Perplexity* helped with the initial setup of the toolchain and was used to find definitions of certain concepts. Additionally, *GitHub Copilot* was active during programming, but mainly used for quick completion of getter and setter functions, as well as function parameters.

Aron Rocco Petritz
(Name in Blockbuchstaben)

12008635
(Matrikelnummer)

Ehrenwörtliche Erklärung

Mit meiner Unterschrift bestätige ich, dass mir der *Leitfaden für schriftliche Arbeiten an der KUG* bekannt ist und ich die darin enthaltenen Bestimmungen eingehalten habe. Ich erkläre ehrenwörtlich, dass ich die vorliegende Arbeit selbständig verfasst, andere als die angegebenen Quellen nicht verwendet und die wörtlich oder inhaltlich entnommenen Stellen als solche kenntlich gemacht habe. Sofern fremde Hilfe (z.B. Korrekturlesen durch Native Speaker) und/oder KI-Dienste verwendet wurden (z.B. internetbasierte Übersetzungstools und/oder KI-basierte Sprachmodelle) habe ich dies ausgewiesen.

Graz, den 05.03.2025

Unterschrift der Verfasserin*des Verfassers

SIGNATURINFORMATIONEN	**Signatory**	Aron Rocco Petritz
	Date/Time-UTC	2025-03-05T16:55:41+0100
	Verification	Information about the verification of the electronic signature can be found at: https://www.signaturpruefung.gv.at
Note	This document is signed with a qualified electronic signature. According to Art. 25 para. 2 of the Regulation (EU) No 910/2014 of 23. July 2014 ("eIDAS-Regulation") it shall have the equivalent legal effect of a handwritten signature.	