Automatic Beatmap Creation for The Video Game *osu!* Through The Use of Abstract Syntax Trees to Analyze Musical Rhythms

Arlow Emmanuel Hergara - 13523161[1]
Program Studi Teknik Informatika
Sekolah Teknik Elektro dan Informatika
Institut Teknologi Bandung, Jl. Ganesha 10 Bandung 40132, Indonesia
[1]arlow5761@gmail.com, 13523161@std.stei.itb.ac.id

***Abstract*—The Abstract Syntax Tree is an interesting data structure that is mainly used in the field of programming language design. While its use in that specific field has been found to be irreplaceable, it still hasn't found any major uses outside of that field. This is odd considering the concept of formal language that the AST operates on is not limited in uses to that field. Thus this paper was created as a means to explore the possibly untapped potential of the AST by employing for beatmap creation in the video game *osu!*. Through the creation of a hypothetical software, this paper explores how AST could be used to complete a task that it is not commonly used for.**

***Keywords*—Abstract Syntax Trees, osu!, pattern generation**

I. Introduction

The Abstract Syntax Tree (AST) is a well-known structure in the world of modern programming language design, commonly used in compilers and interpreters for code analysis, optimization, and compilation. While ASTs have been found to play an important role in modern programming language design, their application in other fields are still limited. This opens up an interesting question of what are the unexplored possibilities of using ASTs in fields outside of programming language design. As such this paper is an attempt at exploring those possibilities by uniquely using ASTs to solve a niche problem, that is the creation of *osu! beatmaps*.

This paper explores the conceptual design and implementation of a hypothetical software solution aimed at generating *osu!* beatmaps directly from MIDI files. The core premise is to investigate the feasibility of automating beatmap creation by leveraging the structured nature of MIDI data and the hierarchical representation capabilities of ASTs. By treating beatmap generation as a problem analogous to code compilation, the proposed software envisions MIDI files as the "source code" and *osu!* beatmaps as the "executable program." The translation process involves parsing the musical elements encoded in MIDI, such as notes, timing, and dynamics, and converting them into corresponding rhythmic and spatial patterns suitable for *osu!*'s gameplay.

This paper consists of several sections: introduction, background, methodology, discussion, and conclusion. The introduction section—that is the current section—provides an introduction to the topic of the paper and the reasons behind the creation of the paper. The background section explains the necessary background knowledge needed to understand the more domain-specific information of the paper. The methodology section explains how the process of creating a beatmap creation software would go and how such a software would work in the first place. The discussion section explains the possible strengths and weaknesses of using ASTs in such a software were it to be actually implemented and any interesting observations from the method proposed to create such a software. The conclusion section summarizes the information within this paper and the key takeaways to be had.

II. Background

A. *osu!*

Fig. 1. osu! main menu.

osu! is a free-to-play rhythm game created by Dean "peppy" Herbert in 2007. While being almost two decades old as of writing this paper, the game is still actively maintained and updated. The game has several modes of gameplay that the players can choose from: standard, mania, taiko, and catch the beat. These modes change the gameplay drastically, each feeling like an entirely different game. The most popular mode and the one that is used in this paper is the standard mode.

osu!'s standard mode is a mode where the gameplay mainly consists of clicking circles on the screen in sync with the rhythm of a level. This gameplay mode was the very first mode ever added to the game and is very similar to *osu!*'s original source of inspiration, a Nintendo DS rhythm game going by the name *Osu! Tatakae Ouendan!*. Levels (or beatmaps as the game calls them) in the standard mode are comprised of three elements: circles, sliders, and spinners. These three elements combine with

one another inside of a beatmap to create interesting patterns and exciting gameplay challenges.

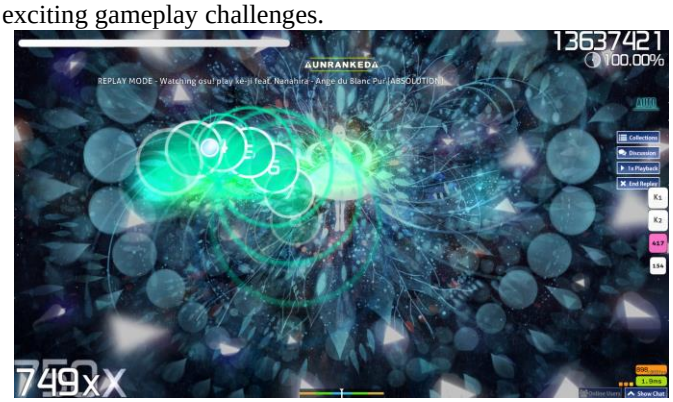

Fig. 2. osu! standard mode circles.

Circles in *osu!* standard are the most basic objects in a beatmap. As the name suggests, these objects take the form of circles when they appear in a beatmap. The player must hit these circles by clicking them at the correct time and in the correct order. Failure to do so results in a miss which negatively impacts the player's results.

Fig. 3. osu! standard mode slider.

Sliders in *osu!* standard are a variation of the circle objects. They consist of three main parts: the sliderhead, the sliderbody, and the slidertail. The sliderhead acts like a normal circle that needs to be hit at the start of the slider. After hitting the sliderhead, a sliderball will appear that travels from the sliderhead to the slidertail following the path of the sliderbody. While the sliderball is present, the user must hold down their click button and follow the position of the sliderball. When the sliderball reaches the slidertail, it disappears, registers the slider as successfully being hit, and allowing the user to let go of the click.

Fig. 4. osu! standard mode spinner.

The spinner object is the most different object compared to circles and sliders. When a spinner appears in a beatmap, it fills up the entire screen, prompting the user to click and drag the cursor while making a spinning movement. The spinner stays on the screen for a specified amount of time giving players a chance to spin as fast as they can in order to gain points. The faster a player spins, the more points that they can get from a spinner. When time specified for the spinner has run out, the spinner disappears. At that time, if the player did not spin fast enough, they get a miss.

Fig. 5. osu! beatmap clear screen.

As any other rhythm game, one of the main goals of playing beatmaps in *osu!* is to improve and get better results. That reason is the source of motivation for many players to keep playing the game. The game displays these results as a combination of multiple metrics that describes a player's performance.

First, there is health, a performance metric that is shown through the player's health bar. The player's health decreases constantly while playing eventually reducing it to zero. Hitting objects grants health for the player while missing objects reduces health even further. When the player's health reaches zero (except from the constant health drain), the player loses and cannot complete the rest of the beatmap.

The second metric for measuring the player's performance is their accuracy. The accuracy measures how accurate the player is at hitting objects in the beatmap. The closer the player hit objects to their specified timings, the higher the accuracy will be. Higher accuracy leads to higher ratings at the end of playing the beatmap.

The third metric for measuring performance is the score. When players hit an object, they gain score and combo. The total

score that is received from that hit is calculated by the score from the hit and the total combo that the player has. When a player misses an object, they lose their combo.

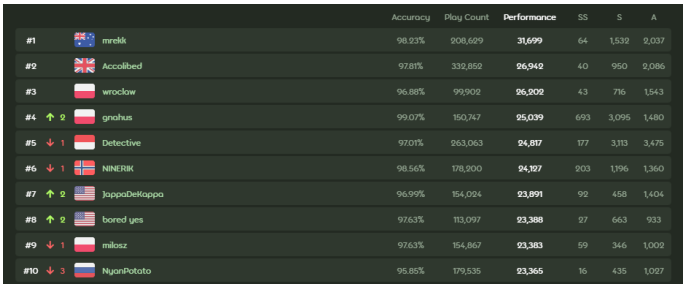

Fig. 6. osu! global leaderboard.

For certain beatmaps that have been marked as ranked, there is another performance metric that is calculated by the game called performance points (PP). PP is considered to be the most accurate measure of a player's performance. The PP that a player gains from playing a beatmap is calculated with a complex algorithm that is frequently updated to ensure a fair and accurate rating of player performance. As PP is considered a very accurate measurement of performance, it is the used as the main method of ranking players in the game's leaderboard.

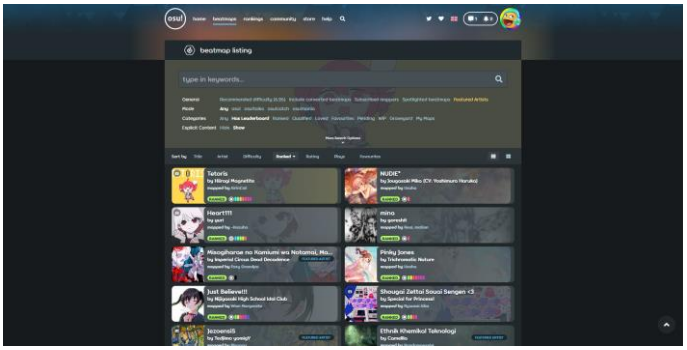

Fig. 7. osu! ranked beatmaps listing.

One of the most interesting parts of *osu!* that makes it stand out amongst other rhythm games is the community generated content. Unlike other rhythm games where the levels are created by the developer, *osu!* beatmaps are created by the players themselves. Regular players can try creating a beatmap by using the editor function inside the game and they can also publish their beatmap to the public if they want. While anyone can publish beatmaps, most beatmaps end up as unranked, meaning they do not generate PP for players playing them and they also do not have a leaderboard (except for beatmaps in the loved category). For beatmaps to be ranked, they must go through a quality assurance process that is also done by members in the *osu!* community with special roles. Players that are regularly able to make beatmaps that end up being ranked are called mappers in the community.

B. Abstract Syntax Tree

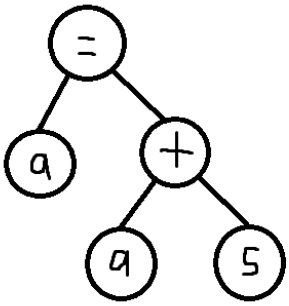

Fig. 8. A small AST.

An Abstract Syntax Tree (AST) is a data structure that is commonly used in source code compilation. The AST provides a way to store the structured data from text written in a formal language with a specific grammar in a hierarchical way. In source code compilation, the AST is used as a way to represent a program after parsing its source code. The use of ASTs to represent programs during compilation makes it easier to do semantic analysis on it, allowing the compiler to understand what the program does and even find ways to optimize it.

As the name suggests, the Abstract Syntax Tree stores information with an abstract syntax. The data structure does not store unnecessary details of the language like formatting or punctuation. Instead, it focuses on representing the structure of the input such that each node in the tree represents a specific construct of some sort. This abstraction of the unnecessary language details is what makes AST desirable for code compilation.

While Abstract Syntax Trees primarily have a use in programming language design, the data structure itself is not bound to only do tasks for that specific field. The AST can represent any text with a formal language and grammar. In fact, there are uses of AST outside of programming language design. Examples of these uses include natural language processing where ASTs are used to represent real natural sentences, identifying malicious code in the field of cybersecurity, and many more.

In the case of this paper, the Abstract Syntax Tree is used to create a hierarchical representation of musical data. If there exists a language where the words are notes and the grammar is different patterns in music, then an AST of musical data can be created. An AST representing musical data can then reveal the structures of the musical piece.

C. MIDI Specification

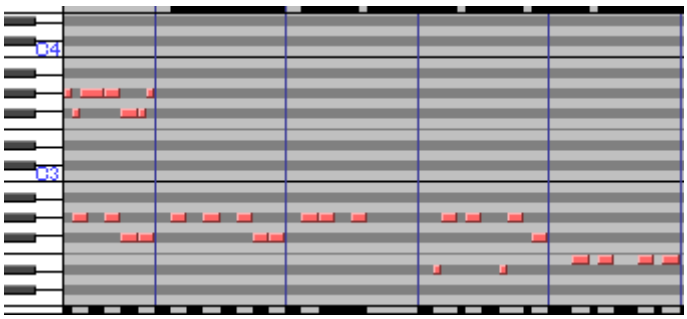

Fig. 8. A MIDI track in audacity.

The Musical Instrument Digital Interface (MIDI) Specification defines the protocol, interface, and file format used for the exchange of musical data between different devices. Unlike standard audio formats, MIDI signals do not actually transmit audio signals. Instead, the MIDI signals contain event messages, which are messages that notify when an event occurred on the source device of the MIDI signal. This difference between standard audio files and MIDI files. While standard audio files are used to store the sound produced by an instrument or device, MIDI files are used to store the different states of an instrument or device when being used.

MIDI files are stored using the .mid or .midi extensions. A typical MIDI file contains a header chunk followed by one or more track chunk. The header chunk contains metadata information about the MIDI file that is useful when trying to read the data contained in the file. The track chunks contain the actual MIDI event data that make up the MIDI signal.

The header chunk consists of a header chunk identifier, the length of the header chunk, the format of the MIDI file, the number of track chunks in the file and the division value. The identifier and length are used to verify the that the chunk is indeed a header chunk. The format is used to determine how the track chunks should be read. A format value of 0 means that the file contains a single track, a format value of 1 means that the file contains multiple tracks and a format value of 2 means that the file contains multiple song. MIDI formats 0 and 1 are common while format 2 is seen less frequently. The number of tracks in the file is used for the reading program to determine how much data should be read by the program before stopping. The division value of the header corresponds to the number of ticks (time points) that are in a beat.

The track chunks consist of a track chunk identifier, the length of the track event sequence, and a sequence of track events that make up the track chunk. As with the header chunk, the identifier is used to verify that the data is indeed a track chunk. Each track event is composed of the timestamp of the event and the event itself. Events in a MIDI file can be one of three types: midi events, meta events, and system exclusive events.

MIDI events are events that happen on specific MIDI channels and usually corresponds with how notes are played. Common MIDI events that are typically used are note on and note off messages. As the name implies the note on and note off messages specify when a certain note in a channel is turned on and off. The note on and note off messages can also contain velocity information which represents how hard the instrument was struck when the note was played.

Meta events are events that don't affect the sound of notes and are not specific to one channel but to the entire track. While most meta events are optional, all tracks need at least one meta event signaling the end of a track. Having meta events and being able to process those events can also result in a better experience as some of meta events contain information that can be useful for playback such as the tempo, name of the instrument used, and the time signature of the track. With such information, a more accurate playback can be created using a MIDI file.

System exclusive (sysex) events are events that are exclusive to a particular system. As their name suggests, sysex events can store any information that the system wants it to hold. Typically, sysex events contain the id of the manufacturer of the device used to produce the event so devices can determine whether or not they can read that specific sysex. If the device recognizes the id, it will process the sysex but if not then the sysex is ignored.

MIDI files are used in the context of this paper as they provide an easy way of decomposing musical notes. As standard audio file formats store the combined audio wave signal of all sounds, it is difficult to decompose the combined wave into its building blocks. Whereas with MIDI files, the stored data is the notes being played themselves. Thus, such a format is perfect for representing musical data in a way that can be accessed easily.

III. METHODOLOGY

In order to construct a program that can create *osu!* standard mode beatmaps from MIDI files using Abstract Syntax Trees, a formal language must first be defined. To do this, the text that will be represented as an AST is determined beforehand. For this paper, the information that needs to be represented is the musical information, primarily when notes start and stop. Thus, all note on and off events within the MIDI file needs to be extracted, appended with their track number information, and combined into one single note event sequence. This single note event sequence that contains information about the track, channel, note number, and note event will be used as the text. Because of this, it can be determined that the words in the formal language will consist of all the possible timestamps, note numbers, event types, and channel numbers.

After determining the words of the formal language, the formal grammar of the language is constructed. There are many ways to create the grammar of the language. Different grammars will result in different ASTs, representing different structures and concepts in music. However, all grammar must at the very least contain a rule that defines a note. As previously, a note was decomposed into consisting of different words, it must now be reconstructed using a grammar rule. Ideally, a note should combine both note on and note off events as to keep its representation as simple as possible. As the end goal of this process is to create *osu!* beatmaps, a grammar rule should also be added to determine if a note should become a circle or a slider. A simple grammar rule that can be used for this is that notes that last longer than a certain threshold become sliders while the rest become circles. However, a different grammar rule can be used to produce different and more interesting results. After assigning notes to be either circles and sliders, another grammar rule must be added to determine which circles and sliders stay when two or more are present at the same time. This rule is required as traditionally, *osu!* standard beatmaps do not require more than 1 button at a time be pressed.

After applying the previous grammar rules to obtain a string of circles and sliders representing the beatmap's rhythm, it is time to encode location information to each circle and slider. Just as previously done to generate the overall rhythm of the beatmap, generation of object location will be done by applying grammar rules to the current language. The simplest way to add location information to each object is to create several grammar rules that map specific object rhythmic patterns to specific object spatial patterns. Doing so results in objects being grouped

into specific patterns in the playfield. However, the resulting patterns would not create a fun gameplay experience as the same patterns would map to the exact same locations. This problem can be fixed by adding more grammar rules that group these patterns into larger patterns that could create different patterns in the playfield such as by offsetting each pattern in the larger pattern by a certain amount to create a sense of progression between patterns. The processing of adding new grammar rules to create larger and larger patterns can be repeated to create more interesting and complex level designs.

After designing the language with all the necessary words and grammar rules, the AST can be successfully constructed to represent the beatmap structure from the MIDI file. After creating the AST, all that's left to do is to actually construct the beatmap. The beatmap can be constructed by creating objects for every circle and slider that appears in the hierarchical structure. Then for every object, the path that it takes from the root node to the node representing the object is determined. This path will go through all the patterns that determine the position of the object allowing the calculation of the object's final position. The end result should be a beatmap that is created completely automatically.

IV. Discussion

In the methodology section, a detailed process for creating a program capable of generating osu! beatmaps from MIDI files through the use of Abstract Syntax Trees (ASTs) was described. This method demonstrated that ASTs could theoretically serve as a tool for translating musical information into beatmaps in a structured and systematic manner. The approach relies on the existence of a formal language for beatmap creation, where the rules and patterns of a beatmap are encoded as grammar rules. While this provides a clear and logical framework for generation, the creation of such a language presents significant challenges. Specifically, the encoding of various beatmap patterns as grammar rules is a labor-intensive task, as each pattern must be explicitly defined. Furthermore, the complexity of the grammar increases with the desire for more intricate or diverse patterns, potentially leading to diminishing returns in terms of effort versus output quality.

An intriguing observation arising from this method is the distinct separation between the rhythmic and spatial components of beatmap generation. The rhythmic patterns, which are directly derived from the timing and structure of the MIDI file, are essentially independent of the spatial placement of objects on the screen. This separation highlights the possibility of splitting the beatmap creation process into two distinct steps: rhythm generation and spatial arrangement. Such a modular approach not only simplifies the problem but also opens the door for combining different methods or algorithms tailored to each aspect. For instance, rhythm generation could be achieved through deterministic rules based on musical structure, while spatial arrangement could leverage machine learning or probabilistic methods to create visually appealing and varied patterns. This dual-process methodology suggests that a hybrid approach might lead to better outcomes by leveraging the strengths of multiple techniques.

Despite the promise shown by the described method, it also reveals certain limitations and areas for improvement in using ASTs for beatmap creation. One significant limitation is the deterministic nature of the generated patterns due to the fixed grammar rules employed. While deterministic patterns ensure consistency and adherence to the defined rules, they can result in beatmaps that lack variety, potentially making them repetitive and uninspiring for players. To address this, introducing mechanisms for variability within the AST framework could enhance the diversity and creativity of the generated beatmaps. As an example, adaptive grammar rules could be incorporated to create more dynamic and unpredictable patterns. Additionally, integrating user-defined preferences or adaptive algorithms could further enhance the process, allowing for greater customization and alignment with player expectations.

V. Conclusion

This paper has shown the possibility of an interesting use case for the Abstract Syntax Tree as data structure used to generate *osu!* standard mode beatmaps from MIDI files. This shows that the Abstract Syntax Tree still has untapped potential and could have more use cases that have still not been discovered. As with the specific topic of this paper itself, while the method for creating a beatmap generation software has been described, a functional implementation of it has not yet been attempted. As such, the next step forward should be the creation of such a software to prove the feasibility of using Abstract Syntax Trees for more unconventional use cases.

VI. Acknowledgment

The author would like to acknowledge their professor for teaching a subject that would inevitably lead them to research such an interesting topic. The author would also like to acknowledge the help of a long-time friend who inspired the possibility of such an interesting idea. The author would also like to acknowledge their other friends who kept the author motivated to continue doing research on this paper's topic. Lastly, the author would like to thank their parents for their unconditional support with this paper and with everything else that the author has faced.

References

[1] D. Thain, *Introduction to Compilers and Language Design*, 2nd ed. University of Notre Dame, 2023. [Online]. Available: https://www3.nd.edu/~dthain/compilerbook/compilerbook.pdf [Accessed: Jan 8, 2025]

[2] "osu! wiki main page," osu! wiki. [Online]. Available: https://osu.ppy.sh/wiki/en/Main_page. [Accessed: Jan. 8, 2025].

[3] "Standard MIDI File Format," McGill University. [Online]. Available: https://www.music.mcgill.ca/~ich/classes/mumt306/StandardMIDIfileformat.html. [Accessed: Jan. 8, 2025].

PERNYATAAN

Dengan ini saya menyatakan bahwa makalah yang saya tulis ini adalah tulisan saya sendiri, bukan saduran, atau terjemahan dari makalah orang lain, dan bukan plagiasi.

Bandung, 8 Januari 2025

Arlow Emmanuel Hergara
13523161