Tachyon: Multiplatform Rhythm Game with Automatic Beatmap Generation

Ganendra Afrasya Salsabilla
Department of Informatics, Faculty of Intelligent Electrical and Informatics Technology
Institut Teknologi Sepuluh Nopember
Surabaya, Indonesia
ganendra16@mhs.if.its.ac.id

Hadziq Fabroyir
Department of Informatics, Faculty of Intelligent Electrical and Informatics Technology
Institut Teknologi Sepuluh Nopember
Surabaya, Indonesia
hadziq@its.ac.id

Darlis Herumurti
Department of Informatics, Faculty of Intelligent Electrical and Informatics Technology
Institut Teknologi Sepuluh Nopember
Surabaya, Indonesia
darlis@if.its.ac.id

Imam Kuswardayan
Department of Informatics, Faculty of Intelligent Electrical and Informatics Technology
Institut Teknologi Sepuluh Nopember
Surabaya, Indonesia
imam@its.ac.id

Shintami Chusnul Hidayati
Department of Informatics, Faculty of Intelligent Electrical and Informatics Technology
Institut Teknologi Sepuluh Nopember
Surabaya, Indonesia
shintami@its.ac.id

***Abstract*—In this research, Tachyon was developed as a horizontal scrolling rhythm game with an automatic beatmap generator. This game was built on osu!framework and BASS Audio Library. It was designed as multi-platform: It runs on both desktop (Windows, Linux, and macOS) utilizing keyboard as the input interface and mobile (Android and iOS) employing touch as the input interface. As a result, Tachyon succeeded in becoming the proof-of-concept of multi-platform rhythm game with automatic beatmap generation. The game was tested on various devices. The test results showed that the game could run up to 240 frames-per-second (fps) on the desktop PCs and 40 fps on the mobile devices. Its automatic beatmap generation yielded a ratio of 90 percent and 93 percent for consistency and accuracy performances. In summary, its generated beatmap was acceptable and playable.**

***Keywords*— automatic beatmap, game framework, desktop game, mobile game, osu!framework**

I. Introduction

Rhythm game is a game that sharpens the reflex of the player through the notes of song or music. The popularity of rhythm games has skyrocketed since the emergence of Parappa, the Rapper in late 1996, that became a pioneer of the early rhythm-based game [1]. The popularity has got even higher after the introduction of Guitar Hero and Rock Band, rhythm games that utilize consoles mimicking real guitars, in 2005. Generally, rhythm games focus on simulating musical instruments and directing players to do some action like pressing certain keys at the right time. Players then gaining score that can be marked as a challenge to play it better. [12]

Rhythm game types are quite diverse. They are based on several aspects, like their inputs (touchscreen, keyboard, dedicated controller, and motion detection), their platforms (PC, mobile, console, and arcade), and their mechanics. However, rhythm games have the same basic and common structure consisting of three main components: audio file, note or hit objects, and accuracy. Audio file or known as chart or beatmap is the main component of this game genre. It represents music, beat, rhythm, and timing. Note or hit object is a component that represents melody, beat, and musical instrument. Accuracy is the main objective of this genre. It also represents players' consistency during the play.

There are a lot of rhythm games nowadays. Nevertheless, most of them are playable only on a single platform since their game mechanics usually require a specific console or specific device to make it more challenging and fun [13]. For example, rhythm game that requires motion is commonly available only on desktop as an arcade, while game that requires touch commonly available only on mobile. Therefore, through the existence of a new game framework called osu!framework covering both mobile and desktop platforms, a multiplatform rhythm game called Tachyon was ideated in this research.

Beatmap generation has been an interesting topic in the context of both game engineering and audio processing. There are several proposed solutions and research in this field. In their research, Pei-Pei Chen et al [10] proposed a real-time percussion identification by classifying a frame with only 2 spectrum-based features. Y. Liang et al [14] proposed solution using C-BLSTM method for timestamp generation method that used for creating osu!mania beatmap. But in this research, we aimed a simpler automatic beatmap generation that will fit into our multiplatform model, yet still generate good beatmaps that satisfy the players.

II. System Overview

Tachyon was built on top of osu!framework version 2020.131.0 local checkout. The main reason for using the local checkout rather than the official NuGet package was to ease the addition or modification on framework's source code while applying it directly to Tachyon. The changes applied to local checkout were focused on performance improvement.

Tachyon's subgenre was horizontal scrolling in which hit objects were moving along horizontal lane. There were 2 lanes: upper and lower lanes in Tachyon. Each lane comprised one judgement circle determining whether player action was correct and on the right timing or not. Both lanes utilized different action to trigger: On the desktop platform players needed to press defined keys while on the mobile platform players were required to touch the lane.

As mentioned previously, there were two types of input mechanism: Players could play with four keys (2 keys each lane) on the desktop and players were only required to touch their respective lane on the mobile. Aside from the input type, there was no difference between desktop and mobile

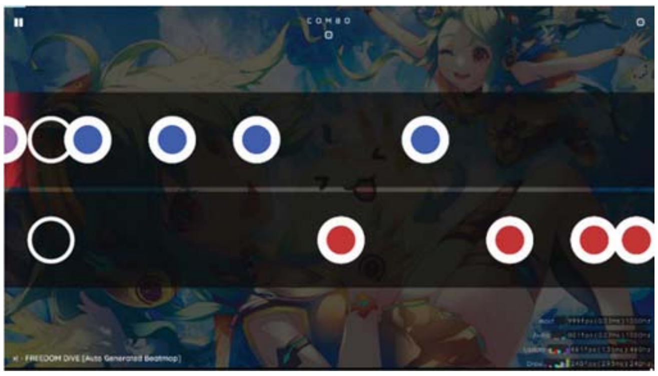

Fig. 1. Tachyon gameplay screenshot

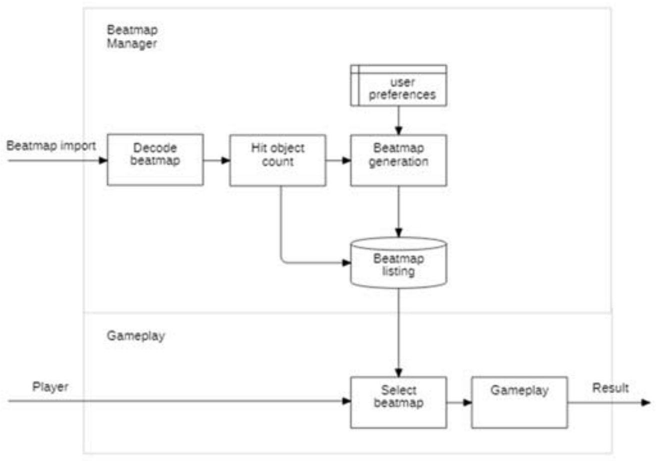

Fig. 2. Tachyon system overview

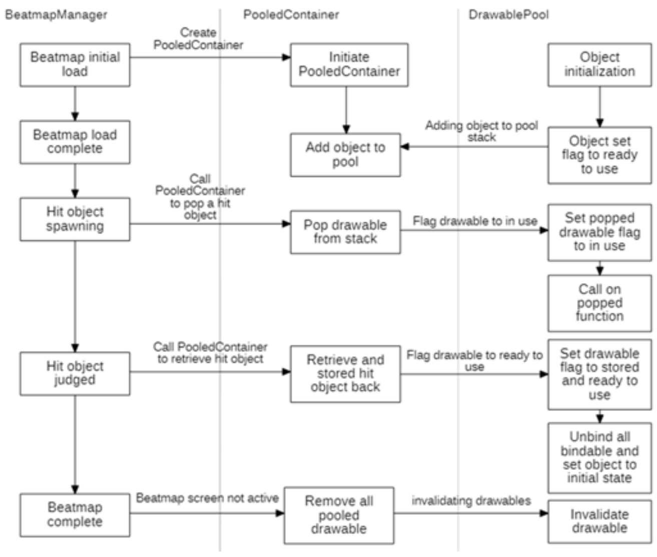
BeatmapManager
PooledContainer
DrawablePool
Create PooledContainer
Beatmap initial load
Initiate PooledContainer
Object initialization
Beatmap load complete
Adding object to pool stack
Add object to pool
Object set flag to ready to use
Call PooledContainer to pop a hit object
Hit object spawning
Pop drawable from stack
Flag drawable to in use
Set popped drawable flag to in use
Call on popped function
Call PooledContainer to retrieve hit object
Hit object judged
Retrieve and stored hit object back
Flag drawable to ready to use
Set drawable flag to stored and ready to use
Unbind all bindable and set object to initial state
Beatmap screen not active
Beatmap complete
Remove all pooled drawable
invalidating drawables
Invalidate drawable

Fig. 3. Drawable pooling implementation on hit object

platform. Both desktop and mobile version of this game shared the same user interfaces and features. The reason is to increase the playability of the game by fixating the gameplay screen, so it makes players easier to understand the playfield and the information (score and combo) placement on screen [2].

Figure 1 shows the screenshot of Tachyon during gameplay (playing a beatmap). The 2 lanes where hit objects are moving provide both Tachyon was divided into modules: Two common modules that handled resources (image assets, system audio and samples); and one main module that handled all logics and game mechanics. Additionally, there were platform-specific modules that handled platform related logics and platform related tests. Figure 2 shows the system overview of the game. A player could add any beatmap files into the game. Imported beatmap could be a fully mapped beatmap (beatmap that already has hit objects and timing) or an empty beatmap. If the beatmap is empty, then game's beatmap manager would generate beatmap automatically based on user preferences. There were two user preferences related to auto beatmap generation: method and beat divisor values.

III. Methods

A. Drawable pooling

Since Tachyon aimed the same game experiences on both desktop and mobile platforms, a few adjustments were necessary to be added. One of primary adjustments was object pooling for better memory allocations (primary for drawables). Beatmap might have hundreds of hit objects depending on its length. Their drawable were could not be reused. As a result, the hit object allocations grew larger according to the beatmap's length.

The drawable had its specific lifetime. When its life reached the end, the drawable was exposed to the garbage collector. However, as for the use case of hit objects, there was a problem since the games could have a very large memory allocation depending on how long the beatmap is and how many the hit objects are on the beatmap. For example, a beatmap with a long duration (commonly was referred to a marathon beatmap) could have a duration of over seven minutes and could have more than one thousand hit objects [3-4]. Hence, each hit object was generated along with its memory allocation resulting in a degraded performance issue.

To overcome the issue, it was necessary to create a system where the drawables could be stored in a pool and reused. In this system, two components were created: *DrawablePool* and *PooledContainer*. The *DrawablePool* was an abstract implemented by other drawables (such as the hit object drawable), which indicates that this drawable could be added to a *PooledContainer*. This *PooledContainer* later managed the drawable's creation (asynchronously), storage (to a pool stack for later use), assignation (on a pooled drawable from the pool stack), invalidation, and disposal (of pool stack contents when *PooledContainer* was no longer used or invalid). With pooling for hit objects, memory allocation had decreased significantly and system performance while playing marathon beatmaps had increased (characterized by stable frame rate and low lag spiking during gameplay). Figure 3 shows how game-side pooling was used. Hit object had implemented *DrawablePool,* which indicated that it could be added to *PooledContainer* pool stack.

B. Initial texture upload and queueing flow

Tachyon had a component called loader that preloaded every texture that was used during gameplay. This loader helped game to reduce in-game loading since each texture was loaded at single time prior to the gameplay. As a matter of fact, this technique required some improvements on how to handle texture upload and queue. Initially, our GL wrapper only uploaded single texture or item per frame. When this

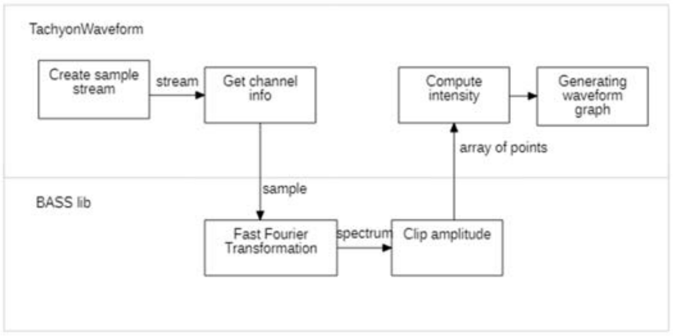

Fig. 4. TachyonWaveform generation

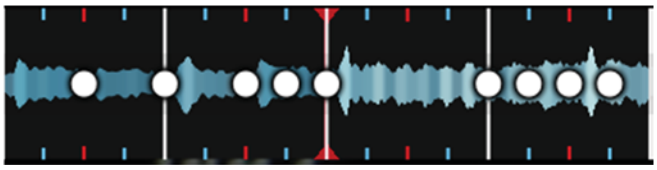

Fig. 5 Generated hit objects on ¼ beat snap

was fine for high-spec desktop, high initial memory usage was considered as a serious problem for mobile and low-spec desktop.

To tackle this performance issue, there were some paths taken. First was to make sure each queue was aware of the texture uploading and executed operation This was done by making our GL wrapper's expensive operation queue, controllable by the user through full control over how many operations were executed per frame to the game host. Through this change, the initial memory usage was reduced by two hundred megabytes.

Another issue that was found when uploading texture was redundancy in which texture might be uploaded more than once. So as the second path, a flag was added to indicate that the upload operation was undergoing, so it was not uploaded on next frame. Additionally, as the third path expensive queue was separated into two queues: One for texture upload and another for shader queue. The separation was to avoid interference across the operations. As a result, the game then took another reduction on RAM usage around two hundred megabytes for the initial texture uploads.

C. Waveform generation

Waveform called *TachyonWaveform* was generated as a component inside the system. This component broke audio file components such as samples, channel count, amplitude, and intensity, which were mostly categorized as song features differing the difficulty level on the same audio file [5]. Figure 4 shows how *TachyonWaveform* were created.

After the audio file was collected from the beatmap file, it was decoded into a stream. Afterward, this audio sample was passed to BASS lib to get its Fast Fourier Transformation data. As for the number of bins generated by the Fast Fourier Transformation (FFT 1024) [6], 512 *fft_bins* were used. The generation results were clipped later since BASS might provide unclipped results.

Respectively, after having an array of points consisting of channel info and amplitude, an intensity [9] was computed per millisecond. There were three ranges that were computed: Low intensity (20-100Hz), midrange intensity (100-2kHz), and high intensity (2kHz-12kHz). This array was then passed to the beatmap manager as a decoupled clock-seeking measurement. This clock seeking was based on the number of beat divisor and the result of the waveform.

Algorithm 1 Hit object decision

```
if ampl[0] >= ampl_threshold and ampl[1] >= ampl_threshold
  if mid_intensity >= mid_threshold
    if high_intensity >= low_intensity
      if ampl_average > 0.5
        upper note at current time
      else
        lower note at current time
    else
      if ampl_average > 0.75
        upper note at current time
      else
        lower note at current time
  else
    no hit object at current time
else
  no hit object at current time
end
```

D. Beatmap generation

There were two beatmap generation methods that players could choose: A random method and another method that could read audio features. The generation was done under beatmap manager component. The component had a decoupled game clock that was used later for both methods to ensure hit objects were placed on the correct timing based on their beats per minute (BPM) and beat divisor.

Specifically, the random method did not rely on audio features; it only relied on audio's BPM. The random method randomized generated pattern at a certain time. There were two different patterns that could be generated: Triplet (three notes on a single section) and quintuplet (five notes on a single section) [3, 8]. To provide various patterns, the hit object's type was also randomized [9], so that the generated pattern was not repeated only on a single lane.

The method that read audio features, namely waveform-based note placement, was coupled highly to *TachyonWaveform*. This method's component held audio features such as amplitude and intensity. Unlike the random method, this method read amplitude and intensity before determining whether there was a hit object and what kind of hit object should be generated at that time.

Several thresholds were used to guarantee suitable conditions on the addition of hit object at that time. If the amplitude was below the specified threshold or it had small value of mid intensity, then the method classified the result as no hit object and the generator seek into the next section, respectively. This classification is to make sure that there was no ghost hit object, an object that does not relate to any beat on the music.

Afterward, in case of the method was suitable for placing a hit object, it would read which intensity that stands the most. The reading was to make sure that the generated hit object was not repetitive and was able to represent the current sample most noticeable percussion.

TABLE I. DRAW THREAD TEST ON VARIABLE CONTROL DEVICE

Device OS	Draw Thread Condition		
	Idle	Gameplay	Minimized
Windows 10	300 fps (2 ms)	240 fps (4 ms)	60 fps (3 ms)
macOS 10.15.3	150 fps (8 ms)	120 fps (10 ms)	60 fps (9 ms)
Ubuntu 20.04	200 fps (5 ms)	120 fps (8 ms)	60 fps (8 ms)
Android 10	58 fps (15 ms)	40 fps (28 ms)	-
iOS 13.4.1	24 fps (28 ms)	15 fps (38 ms)	-

TABLE II. UPDATE THREAD TEST ON VARIABLE CONTROL DEVICE

Device OS	Update Thread Condition		
	Idle	Gameplay	Minimized
Windows 10	300 fps (2 ms)	240 fps (4 ms)	60 fps (3 ms)
macOS 10.15.3	150 fps (8 ms)	120 fps (10 ms)	60 fps (9 ms)
Ubuntu 20.04	200 fps (5 ms)	120 fps (8 ms)	60 fps (8 ms)
Android 10	58 fps (15 ms)	40 fps (28 ms)	-
iOS 13.4.1	24 fps (28 ms)	15 fps (38 ms)	-

There were cases that the generated patterns had the same hit object type on long section. The case occurred mostly because the samples on that section had the same percussion for long period (mostly found on the end section of music). Figure 5 shows the example of one forth beat divisor and the generated hit object.

IV. Evaluation

A. *Performance test*

The performance test was conducted to measure the system's or the game's stability. It evaluated four devices as dependent variable representing all platforms that were compatible with the game.

The test was administered in three scenarios: Idle, gameplay, and minimized. The results were recorded as framerate in draw and update thread under multithreading

condition. According to the results, the test on the desktop environments yielded higher framerate and had more stable framerate compared to mobile platforms. Overall, the system secured an acceptable framerate for a rhythm game and had a small amount of *lag spiking*. The details of the results are shown on Table 1, 2, 3, and 4.

B. *Auto generated beatmap test*

The auto generated beatmap test mainly focused on the robustness of the generated beatmap under the method that read audio features and on the user feedback regarding challenge level, intuitiveness, and uniqueness [11] of the generated beatmap. This test was conducted on twenty players who had already had experience in making beatmaps or playing horizontal scrolling rhythm games. The test measured the beatmap difficulty, timing distribution, pattern

TABLE III. DRAW THREAD TEST RESULT ON ADDITIONAL TEST DEVICES

Device OS	Condition		
	Idle	Gameplay	Minimized
Windows 10	1100 fps (0.8 ms)	1000 fps (0.3 ms)	60 fps (1 ms)
macOS 10.15.3	150 fps (8 ms)	120 fps (10 ms)	60 fps (9 ms)
Windows 10	1800 fps (0.6 ms)	1500 fps (0.8 ms)	60 fps (1 ms)

TABLE IV. UPDATE THREAD TEST RESULT ON ADDITIONAL TEST DEVICES

Device OS	Condition		
	Idle	Gameplay	Minimized
Windows 10	6000 fps (0.1 ms)	2500 fps (0.4 ms)	60 fps (1 ms)
macOS 10.15.3	1200 fps (2 ms)	600 fps (2 ms)	60 fps (1 ms)
Windows 10	8000 fps (0.1 ms)	2100 fps (0.5 ms)	60 fps (1 ms)

TABLE V. BEATMAP AUTO-GENERATION TEST RESULT

Tests point	Result (%)
The difficulty level of the generated beatmap	86%
The accuracy of the timing of the hit object to the audio	77%
The resulting level of pattern variation	90%

variation, and beat-snapping correctness. There were several beatmap tested with different genres (drum and bass, dance, electronic, rock, J-pop, hardcore). According to the results,

the overall beatmap difficulty that was generated was acceptable for casual rhythm game players.

Pattern variation that was generated also had good results due to the threshold that was applied during hit object decision. On the other hand, however, the threshold also made some hit objects not representing the correct percussion mostly on music with lyrics. For the timing of the hit objects, there were some misplaced hit objects (ghost hit objects) [3]. This was mostly happened on the music that had diverse percussion and had very noticeable low BPM music. Nonetheless, it gave good results on the music that had noticeable percussions like drum and bass (DnB) and rock genre. The results of this test are shown on Table 5.

C. *osu! beatmap comparison*

Figure 6 shows an example of osu! ranked beatmap (human-made) compared to auto generated on the same music with genre electronic. We can find a lot of similarities in hit object placement. Although the generated beatmap cannot replicate denser pattern, the timing and hit object placement are similar. But only comparing the timing and hit object placement is not enough to estimate how playable or how decent the generated beatmap.

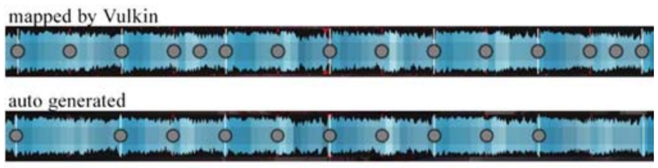

Fig. 6. Compare of ranked osu! beatmap and auto generated beatmap

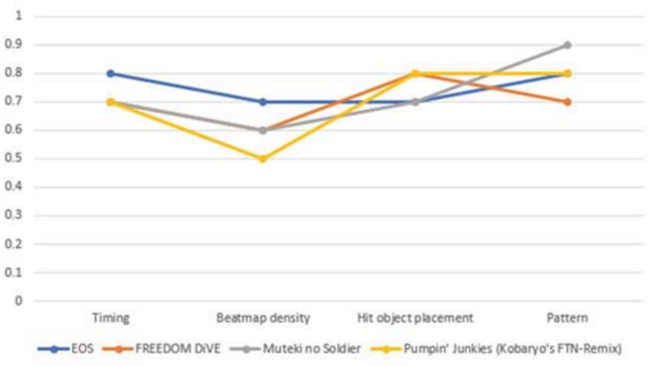

Fig. 7. Mapper score on four generated beatmap compared to osu! beatmap

The comparison test was also conducted to five mappers to compare human-made beatmap with the auto generated beatmap. There are four points assessed in this test: timing, beatmap density, hit object placement, and the generated Overall, it shows that the timing, hit object placement, and the generated pattern of the beatmaps are acceptable and good enough compared to osu! beatmap with the same music. For the beatmap density, the generated beatmap with high BPM (250 BPM and above) has a low score compared to the beatmaps with slower BPM. Figure 7 shows the results of this test.

V. Discussion

osu!framework is a framework that still in high pace development cycle. There are a lot of performance improvements made by its developer. However, it still has a lot of holes to cover. One of them is the object pooling as mentioned in the previous section. When Tachyon (and osu!framework in general)'s performances were solid on most desktop devices, it still had some performance issues on mobile platforms and some low-end desktop devices. One of the proposed methods is to tackle this issue is by adding object pooling on framework side. The results yielded by the performance evaluation reached an acceptable frame rate on the mobile and stable performance on the desktop platform.

In general, auto generated beatmap secured an acceptable quality as playable beatmap. Some issues were mostly found on audio with lyrics since it could make hit object decision hard to check on the overlapped percussion and vocal amplitude and intensity. Nevertheless, the results were quite good for other audio genres like drum, bass, electronics, and instrumental, since their instruments had a distinct value of intensity and amplitude (but not every audio are having a distinct value). Having only intensity and amplitude to decide hit object type was a good early step but cannot cover every song genre and instrument. So, it needs to expand more by not only reading the amplitude and intensity but other features, like audio spectral energy [10] to make percussion's identification better.

The proposed method also has some limitations. The current method does not work well with music or track that have various BPM change since beat divisor is not aware of BPM change thus will provide bad timing for hit object placement. The proposed method also does not provide a break section on the beatmap. The break section is a section on the beatmap that does not contain any hit object, commonly placed when music or track is on the slow part. When break section is not a must in a beatmap, some song has a very distinct section that indicates it is suitable for break section. This can be done by improving the proposed method to not only check the current position but also checking the whole section based on the BPM and or based on the beat divisor.

VI. Conclusion

In this research, a new yet promising game framework called osu!framework that have robust and good performance for the multiplatform game was introduced. Additionally, a game called "Tachyon" that provides a unique game mechanics that can be played on both desktop and mobile platform was developed as a proof-of-concept. The game allowed players to play their own favorites music without any worry about the beatmap creation. This game could generate beatmap with good hit objects timing and acceptable beatmap difficulties. This feature that was combined with osu!framework performance made this game unique and worth to try on mobile or desktop.

Our evaluation results show that by improving framework's performance during gameplay Tachyon could reach up to around 240 frames-per-second (fps) on the desktop and about 40 frame-per-second (fps) on the mobile. The evaluation also resulted in good and playable generated beatmap with 90% variation on the generated patterns and 93% accuracy on the beat snapping. All in all, these results and auto generated beatmap innovation imply Tachyon's unique and playable quality for casual rhythm game.

References

[1] A. Webster, "Roots of rhythm: a brief history of the music game genre," 2009. [Online]. Available: https://arstechnica.com/gaming/2009/03/ne-music-game-feature/. [Accessed 2020].

[2] C. H. C. a. C. S. Lo, "The Development of a Music Rhythm Game with a Higher Level of Playability," 2016.

[3] osu!, "Beatmapping · wiki · help," [Online]. Available: https://osu.ppy.sh/help/wiki/Beatmapping. [Accessed 3 November 2019].

[4] osu!, "Hit Objects · wiki · help," [Online]. Available: https://osu.ppy.sh/help/wiki/Hit_Objects. [Accessed 3 November 2019].

[5] R. Y. a. Y. Y. Yudai Tsujino, "Characteristics Study of Dance-charts," 2019.

[6] M. Goto, "An Audio-based Real-time Beat Tracking System for Music With or Without Drum-sounds," *Journal of New Music Research,* vol. 30, 2002.

[7] L. D. S. A. C. D. M. D. a. Juan Pablo Bello, "A Tutorial on Onset Detection in Music Signals," *IEEE TRANSACTIONS ON SPEECH AND AUDIO PROCESSING,* vol. 13, 2005.

[8] Tasha, "Taiko Mapping Basics / Guide to making a good Taiko Mapset," [Online]. Available:

https://docs.google.com/document/d/1STnZVwB-RN6glJbSlOiTEiUxu7XmgMInXrG15s2pfbQ/. [Accessed 10 06 2020].

[9] D. B. P. a. N. U. Maulidevi, "Beatmap Generator for Osu Game Using Machine," 2015.

[10] T.-C. Y. J. R. J. a. W. L. Pei-Pei Chen, "AutoRhythm: A music game with automatic hit-time generation and percussion identification," in *IEEE International Conference on Multimedia and Expo (ICME)*, Turin, 2015.

[11] J. Hoysniemi, "International survey on the Dance Dance Revolution game," *Computers in Entertainment,* April 2006.

[12] F. K. Martin Pichlmair, "Levels of Sound: On the Principles of Interactivity in Music Video Games," in *DiGRA Conference*, 2007.

[13] P. W. Penelope Sweetser, "GameFlow: a model for evaluating player enjoyment in games," *Computers in Entertainment,* July 2005.

[14] Y. Liang, W. Li and K. Ikeda, "Procedural Content Generation of Rhythm Games Using Deep Learning Methods," *ICEC-JCSG,* 2019.