M22 - A Modern Visual Novel Framework

Samuel Lynch
Creative Technology
Bournemouth University, UK
i7420737@bournemouth.ac.uk

Fred Charles
Creative Technology
Bournemouth University, UK
fcharles@bournemouth.ac.uk

Karsten Pedersen
Creative Technology
Bournemouth University, UK
pedersenk@bournemouth.ac.uk

Charlie Hargood
Creative Technology
Bournemouth University, UK
chargood@bournemouth.ac.uk

ABSTRACT

This paper presents a modern, open-source game engine/framework for the visual novel genre of interactive narrative. It takes the insights from the engines of visual novel games and the products made with them to produce a free engine that contains all the features and components required of a standard visual novel, and demonstrates its capabilities with a demo artefact. Visual novels provide authors with a powerful way of presenting their fiction and narratives, yet they are often considered less viable due to the costs required against the profit in sales, or because of their technical requirements to use. The M22 engine aims to address both these issues.

ACM Reference format:
Samuel Lynch, Fred Charles, Karsten Pedersen, and Charlie Hargood. 2019. M22 - A Modern Visual Novel Framework. In *Proceedings of 8th International Workshop on Narrative and Hypertext, Hof, Germany, September 17, 2019 (NHT'19),* 5 pages.
https://doi.org/10.1145/3345511.3349284

1 INTRODUCTION

Visual novels are a type of interactive narrative, typically used by game developers in Asia to present a non-linear, interactive story. However, the genre is gaining popularity amongst independent developers in the West due to the relatively low budget required for their development. There are a variety of features typical of a visual novel game, which will be detailed further on, however these features are generally minimalistic and therefore the majority of the effort in creating such a game is in the writing and scripting. There exists open source software development kits for creating visual novels, however they are limited in number, each with their own strengths and weaknesses. And these weaknesses are generally common amongst each other. Chiefly, these are their accessibility to developers, their platform flexibility, and script/game extensibility. By leveraging a minimalistic scripting language, the Unity Engine [3] as a base, and script language extensions via C#, the M22 engine aims to resolve these weaknesses, whilst maintaining the expected feature set of a visual novel engine.

Visual novel games— while holding a very minor position in western markets (there are approximately 1625 titles tagged "Visual Novel" on Steam, out of 30,000 as of 2019 [19])— are still financially viable representations of prose in eastern markets, particularly Japan's. However, these are relatively monopolised by large companies producing the games together with a popular franchise (for example, a television series or a film) or popular *seiyuus* (voice-over artists). As such their engines receive investments of time and money, and are made proprietary to protect this. There exists a market of independent game developers in Japan, self-publishing games (*dōjinshi*) in small quantities who have limited options in the way of free, open-source solutions to producing visual novels. These options— limited as they are— are still viable, but may deter certain individuals without a background in programming. Generally, interactive narrative authors will have had programming experience in some form creating these narratives, however prose authors wishing to experiment with interactive narrative via visual novels would likely find difficulty in the setup of these existing visual novel solutions.

Our aims are to produce an original, open-source visual novel engine that uses more modern, more familiar technology than that of other engines; and to make the engine's ease-of-use and barrier-to-entry as optimised and minimal as possible. By making the engine easy to use and setup, the hope is that narrative authors with little to no programming knowledge can hit the ground running, whilst also not making the engine limiting to those with existing knowledge of programming or visual novels creation.

2 RELATED WORKS

The visual novel games chosen for their feature sets were Clannad [14], Katawa Shoujo [8], Steins;Gate [11], NEW GAME! -The Challenge Stage- [10], Clannad: Side Stories [15], and Re:Zero -DEATH OR KISS- [9]. These games were picked for their "typicality" as visual novels, regarding features. While NEW GAME! and Re:Zero are modern examples of visual novels. Clannad: Side Stories is an example in minimalism (i.e. minimal features) and is a kinetic novel rather than a visual novel (i.e. no branching decisions). Clannad, Katawa Shoujo, and Steins;Gate are in the top 100 most popular visual novels of all time, as ranked by the Visual Novel Database [5].

Figure 1 shows that the three most popular visual novels in the selected games lacked gameplay elements (any player interaction

NHT'19, September 17, 2019, Hof, Germany

ACM ISBN 978-1-4503-6901-5/19/09…$15.00
https://doi.org/10.1145/3345511.3349284

	Visual Novel games						
Feature	Clannad	Katawa Shoujo	Steins;Gate	Clannad: Side Stories	NEW GAME: THE CHALLENGE STAGE	Re:Zero – DEATH OR KISS	Planned for project engine?
Background graphics	Y	Y	Y	Y	Y	Y	Y
Character graphics	Y	Y	Y	N	Y	Y	Y
Character facial animations	N	N	N	N/A	Y	Y	Y
Audio (music, SFX)	Y	Y	Y	Y	Y	Y	Y
Branching story	Y	Y	Y	N	Y	Y	Y
Gameplay elements	N	N	N	N	Y	N	N/A
"CG" Scenes	Y	Y	Y	Y	Y	Y	Y
Character voiceover	Y	N	Y	Y	Y	Y	N
Saving/loading	Y	Y	Y	N	Y	Y	Y
Transition effects	Y	Y	Y	Y	Y	Y	Y
Video playback	N	Y	N	N	N	N	Y
Multiplatform	N	N	N	N	Y	Y	Y
Total:	8	8	8	5	11	10	
First Released	2004-04-28	2012-01-04	2009-10-15	2010-06-03	2017-01-26	2017-03-30	
Genre	Slice of Life, Drama	Slice of Life, Drama	Sci-Fi, Mystery	Slice of Life	Comedy, Slice of Life	Fantasy, Mystery	

Figure 1: Comparison of existing visual novels.

with game-like elements that change the story), character facial animations (where the character's mouth moves with the text onscreen, or the eyes blink), and multiplatform capability (at launch). This indicates that these features are not a necessity for developing a successful visual novel. Additionally, the presence of multiplatform capability in the two newer titles indicates that multiplatform was a difficulty in the past (Unity, for example, offers one-button building to multiple platforms with minimal setup required). However, these features should be implemented regardless, albeit on low-priority.

Due to the popularity of the first and third titles in Figure 1, their engines are available as reverse-engineered. This enables the ability to see how the features of these games were implemented on a technical level, in addition to the core engine components (e.g. script language/compiler). The engine used by Katawa Shoujo is the Ren'Py engine [2]; a freely available, open-source engine written in Python. It's touted as relatively easy to use, despite having what is called [13] a text-based narrative authoring system; arguably the least accessible authoring system, in addition to narrative being written in a programming language. However, it could also be said that Python contributes to its ease of use, as non-programmers could quickly learn enough to write interactive narrative. However, it faces the limitation of the language in performance; Python maintains small code sizes but is generally regarded as ten to one hundred times less performant than a compiled language such as C [18], but has the benefit of being slightly easier to port to other platforms due to it being an interpreted language.

Steins;Gate, New Game!, and Re:Zero all use a variation of the same engine, of which its older versions are reverse-engineered [7]. libnpengine takes a much more programmatic approach to authoring the narrative than Ren'Py, though it is written in C, which allows it to be far more performant. More difficult to port to other platforms, but if ported would allow the games to run on lower-spec devices.

Previous attempted iterations of the project engine, as previously mentioned, were C++ and Lua. Both versions used the same custom script language for writing the narrative, but the C++ version was more performant than the Lua version (in both memory and CPU usage), while the Lua version was more easily ported (PS Vita [6] to PC took one day). Additionally, the C++ version was difficult to extend/maintain as it was much lower-level than the Lua iteration. rlvm [4] is a free reimplementation of the VisualArt's RealLive engine/visual novel script interpreter (used in Clannad), which does not contain a compiler nor decompiler for the scripts, as the engine is simply an interpreter that expects a specific file, and as such is only useful for seeing how to implement certain visual novel features. Written in C++, it may be useful for seeing performant code to present certain features (e.g. character animation).

The primary outcomes from this is that text-based narrative authoring systems can still be successful engines, so long as their feature set is varied enough, and is accessible in other aspects. It is also important to be accessible to the user; by using an interpreted language like C in an engine like Unity, the resulting software should be performant enough to manage anything that a visual novel author would require.

3 SYSTEM OVERVIEW

When creating the Unity version, the engine structure was kept similar to the style of the original C++ version, with modifications made where C#/Unity could run more efficiently. However, the main focus was that the system was designed to be as easy to setup and use as possible. The assumption is that developers using the system have an elementary working knowledge of Unity— only to the extent of being able to setup a new Unity project. From a default, blank Unity scene, the user can simply attach the M22 master C# script to their main camera, and everything barring content/assets will already be configured appropriately. From the technical side, this master script initializes the visual novel/narrative system thusly:

- Initialize required submodules (chiefly script, character/background, audio, and text/narrative handling)
- Initialize the required 2D canvases, to which the narrative/assets will be rendered
- Load, compile, and (optionally) execute the entry-point script

In Figure 2, the M22 Master Script is the start of the "waterfall" flow of the engine described above. By attaching the main script to a camera in a Unity scene, the script handles the instantiation of required components (chiefly the main interpreter script, ScriptMaster), as well as the required objects in the Unity scene (i.e. the canvases for backgrounds/characters, audio sources, the visual novel textbox). This method of setting up the framework for the developer was designed for ease of use (only required step is to drag the script onto the camera), but also so that the framework could be instantiated as needed. This makes the framework viable for conveying narrative in Unity games that aren't visual novels by simply adding the script to the camera and turning off autostart for the entrypoint script. This is complementary to the goal of making the framework viable for "power-users", or experienced visual novel developers.

The rest of engine is self-explanatory from their component names; audio is handled by the AudioMaster, input is handled by the InputWrapper, and visual novel elements such as backgrounds, characters, and text rendering are handled by the VNHandler. "Function scripts" corresponds to the individual visual novel functions in the script language, which are stored as C# classes with uniform syntax, and queried by the ScriptMaster when the corresponding

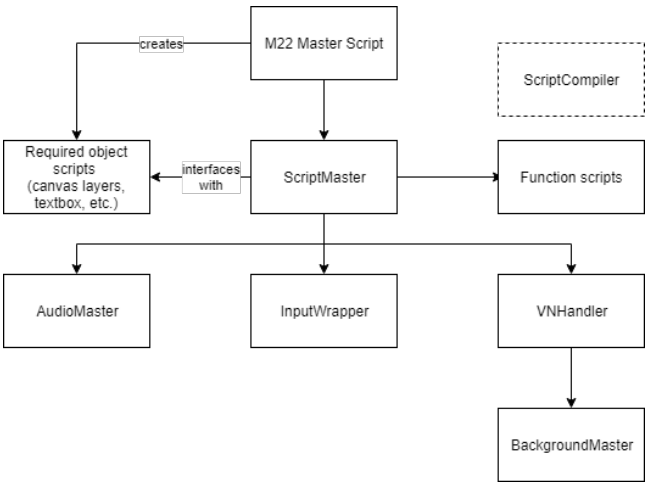

Figure 2: A top-level diagram of engine composition for the Unity iteration. *Note: ScriptCompiler is a static class.*

function is called in code. The ScriptCompiler is written in pure-C# (hence the necessity for the UnityWrapper class; for wrapping functionality such as loading text files) and is a static class. This was done to have the compiler be portable C# code that could be exported as a library or simply transplanted into future projects, for re-usability. Having previously written the compiler in C++, Lua, and JavaScript, this was seen as a good move for the long-term health of the project.

Character/background rendering is handled by a parent class (their respective handlers) which simply spawn instances of characters and backgrounds— based on the parameters— into the scene, leveraging Unity's prefab system. Eventually, a character or background is just a Unity 2D Sprite drawn onto a canvas layer, with methods for manipulating it via game script (e.g. adjusting position with/without animation, transitioning between different forms of the same sprite, etc.). The reason why a character handler does not appear in Figure 2 while the BackgroundMaster script does is because characters are streamed assets (as opposed to pre-loaded) and therefore handle themselves independently. The audio was implemented very simply, also leveraging Unity's audio source and listener functionality, which enabled such a simple implementation. Video was implemented similarly to the audio; just passing a Unity-compliant video file to the Unity video API.

The aforementioned canvas layers are separated by what is contained within them; there are multiple canvases for both backgrounds and characters, to be able to layer characters on top of other characters, etc. There are extra layers marked for effects— both foreground and background— which are not used by default but can be interfaced with via script extensions.

3.1 The M22 Scripting Language

The compiler simply takes an input script file and compiles an object in as memory-friendly a way as possible. It recognises keywords to construct tightly-packed function payloads; data objects that tell the script interpreter what function to execute, with what parameters. Dialogue/narrative is simply the fallback if no keyword

```
label this_is_a_chapter:

anything written like this
will be ignored by the parser

window show

script functions such as above and below

play music music fadein 1.0

"I am the narrator speaking"

char "This is a character speaking, as indicated by the this prefix"

"Back to narrator!"
```

```
--this_is_a_chapter

// anything written like this
// will be ignored by the parser

ShowWindow

// script functions such as above and below

PlayMusic music 1.0

"I am the narrator speaking"

char "This is a character speaking, as indicated by the this prefix"

"Back to narrator!"
```

Figure 3: Ren'Py (left) and M22 (right) script language syntax

was detected, which makes the script more error-prone to user-error (such as a typo), but also makes non-dialogue text look more natural when writing. For most intents and purposes, a character's name is treated as a keyword, however this is checked after failing to detect a function keyword, and before defaulting to narrative text. This could be seen as inefficient, but this process only takes place at script compilation during runtime; it does not affect the performance after this point (i.e. during gameplay). This is also because the compiler hashes the function keywords/character names so that at runtime, the interpreter can more quickly lookup and execute the appropriate functionality. Because the compiler simply compiles a script file into a standardized data object, it is not infeasible to create parsers/compilers for pre-existing script languages (for example, Ren'Py's form of Python) that are interpreted the same way, and this has been experimented with before in the RPY-eBook project [16], which was capable of receiving *.rpy*, *SEEN.txt* (from visual novels made by KEY [1]), or the *.m22* format of script, and produce an eBook-friendly, readable output. However, the current focus is on refining the custom language before supporting others.

The script language (Figure 3) is not dissimilar to Ren'Py, with only a few key differences. This was deliberate, as Ren'Py has a script language that many people see as easy-to-use (or at the very least, easy to get started), as well as making the M22 language approachable to existing/former Ren'Py developers, so there were quite some syntax decisions taken from the Ren'Py language. For example, functions were abstracted into single keywords rather than chains of keywords like in Ren'Py. "*play music track fadein 1.0*" becomes "*PlayMusic track 1.0*", with the fade-in aspect implicitly declared by the absence of fade speeds (the third parameter). Another example would be the handling of comments— or any text in the script file that isn't explicitly narrative. In Ren'Py, text that isn't explicitly declared as a string/as narrative is ignored (i.e. narrative declaration is explicit), however in M22, narrative declaration is implicit. In addition, M22 has no requirements on indentation, whereas Ren'Py relies on indentation for code blocks in the same way Python does. This makes M22's script language look cleaner and closer to traditional narrative, at the cost of the "failing loudly" paradigm that Ren'Py has (the approach of 'if it can fail, then at least make it fail loudly so it can be caught quicker').

3.2 Extending the Functionality of the Engine

The script extension system involves using C# to extend script language functionality by associating new keywords with a C# class. From there, an advanced user can manipulate the Unity engine from the script language. An example used in the demo software was a function that would utilise the background effect layer to hold a particle emitter for leaves blowing in the wind, and by using this with a background that has transparency, the resulting effect

creates a sense of separation between the protagonist's microcosm and the outside world, by animating the exterior while keeping the interior static.

The more complex features required a more manual implementation. Perhaps the most complicated of these was the transition effects. A key, subtle part of visual novel games is how they handle the transitions between settings, environments, characters, etc. One could think of them similarly like slide transitions on PowerPoint; no one would use a star-wipe transition during a presentation on, say, charity fundraisers for life-threatening diseases. And the same is true for transitions in narrative; during a particularly intense scene, one requires a particularly intense transition. For a dream sequence, one might use a swirling fade-to-black transition. As this is such a crucial element, great care was taken in developing the feature. After reversing older visual novels such as Snow Sakura [12] and Katawa Shoujo, it was evident that the primary methodology used for this feature was using greyscale images— images with pixels of values between 0 and 255— to define the pattern and speed at which the transition occurs. Previous iterations of the M22 framework (barring the JavaScript iteration) transitioned between backgrounds or characters via per-pixel CPU operations; an intensive operation that scales linearly with larger resolutions. With the Unity and JavaScript versions, this process is handled by shaders; leveraging hardware acceleration to the point where the operation is trivial. An author can define how quickly the transition occurs, and what to transition into (keywords such as "black" and "white" exist, but the author can also transition to any other image). The shader was first developed for backgrounds but is used in a similar capacity for transition operations on character sprites.

4 DISCUSSIONS

For testing engine features and overall stability, a specific game was ported as a test/demo software; Katawa Shoujo [8]. As it is a Ren'Py game, porting it to the M22 framework would also be a testament to the ease-of-use for the porting process. Additionally, the game makes full use of the minimum expected feature set of a visual novel outlined in Figure 1, meaning that— once created— the demo software could function as a feature implementation test; by running through the software and noting anything that behaves as if it shouldn't according to the game as if it were running on Ren'Py. For example, there is a scene in Katawa Shoujo that contains chained character sprite changes, inline functionality (i.e. script functions run in the middle of text as it appears), an animated background, and branching narrative based on a decision made shortly before. This scene functions as an excellent stress test for the character/background systems, as well as core script language functionality.

This test proved to be extremely helpful in the development of the engine for highlighting non-working features, and also resulted in a demo artefact for the engine. There was a tool made in the past [17] that could compile Ren'Py script to M22-Lua format, and said tool would need little modification to do so, however it was felt that porting the Ren'Py script by hand would give better understanding of how .rpy script is written. The process of converting the scripts was painless, although the knowledge of the M22 framework would

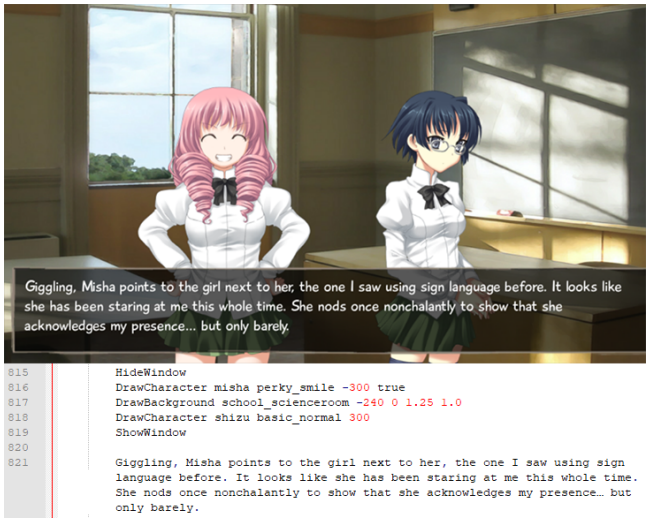

Figure 4: A scene in Katawa Shoujo running on M22, with a script snippet of how the scene is created

bias this. Nevertheless, another look at Figure 3 shows great similarity in the two languages; likely the most difficult part of the process was the mapping of Ren'Py-specific functions to M22, as well as writing the script language extensions for functionality that the Katawa Shoujo developers implemented specifically for their game.

4.1 Features Implemented

Overall, our work sufficiently implements most planned features outlined in Figure 1. Saving and loading of game progress is unfortunately not implemented yet, but since games with short stories (e.g. Clannad Side Stories) do not implement the feature— as well as it not being necessary for demo software— it was considered unnecessary for a minimum-viable-product or demo artefact. Previous iterations possessed save-load functionality, although they were all flawed in some way. The C++ version possessed the most straightforward implementation, storing the current state as a binary file and simply restoring the state later, but this also was flawed since the point the game restores at may not require all the assets of the script, as the player may have passed the last point those assets were utilized, yet they are still loaded. The Lua version stored save data as an executable Lua file that initialized the required variables to restore the state, which was a security concern; this would not be permitted on any closed platforms such as the PlayStation 4. The JavaScript implementation exported the state to a JSON-format file and restores it later, which is also a security concern as the file is in plaintext and easily editable.

There were a number of difficulties encountered during development; issues that arose as a result of design choices, or issues that forced redesigns. There were the issues encountered in previous iterations of the engine under C++, Lua, and JavaScript as aforementioned, but the the Unity version had its own problems during development. The first issue was a result of attempting to incorporate the "drag-and-drop" approach of the previous engines

(i.e. a user could simply drop assets in one folder, scripts in another, and run the game). Unity utilizes two key methods of loading assets from code (as opposed to using the Unity systems), which is via the *Resources* folder and the *StreamingAssets* folder. The key difference between the two is that the Resources folder is packaged, compressed, and bundled with the rest of the application, while StreamingAssets are files that are not changed in any way between building the game and playing it (i.e. they are not compressed or bundled into archives). To maintain the drag-and-drop approach, StreamingAssets makes the most sense, as it allows developers to place new scripts without recompiling the entire software. However, because it does not do any compile/build-time processing, it can result in large or sub-optimal games. In addition, to have script files loaded from StreamingAssets could be a security concern as it means anyone with filesystem access to inject malicious content with minimal difficulty (made worse when considering secure platforms such as Nintendo Switch). To that end, it was necessary to support both approaches; StreamingAssets for development/authoring, and Resources for production. An improvement that could be made would be to automatically change assets from StreamingAssets to Resources when compiling a production build. This would solve issues that could arise from simply forgetting this step when shipping production versions of software.

Another issue encountered during development was the overall project/engine structure/architecture; the Unity iteration was not originally intended to be the final iteration, but just another of the iterative steps. Once decided that the Unity version would in fact be the definitive iteration of the engine, substantial refactoring of the codebase was required. The first working prototype of the engine did not have the ease of setup that the final version possessed. While the final version requires a prospective developer to attach a single script to the camera, the prototypical version required up to five scripts to be attached *and* configured on the main camera, as well as attaching the relevant canvases to which the visual novel would be rendered. This was eased by having a Unity prefab of the main camera with all these scripts pre-attached, however they would still require configuration, and this setup was not ideal for developers who already had a game setup and merely required the M22 framework for narrative. Therefore, it was necessary to implement the more straightforward approach of a single script, which automatically sets up the entire framework on the camera that it is attached to; the only requirement beyond this is actually supplying the game scripts and assets for it to interpret.

5 CONCLUSIONS

We have created a visual novel engine with modern technology, taking full advantage of Unity functionality; from a top-level such as playing videos using their video system, to the language-level of structuring the engine as Unity-compliant C# code. It uses shader capabilities to serve transition effects, as well as making full use of the 2D UI/canvas systems that Unity offers, which results in a highly optimal visual novel engine that is fully capable of interpreting a large visual novel script (1000+ lines).

The next step will be to conduct appropriate and extensive user evaluation, though the engine bears striking similarities to the Ren'Py syntax, assumedly making the language approachable to existing developers of visual novels via Ren'Py. However, for users who have not written visual novels before— at the least, not with one of similar syntax— it would require user evaluation before being able to conclusively state how effective the syntax-styling of the M22 language is for these users. However, what can be said is that due to the nature of the development of the accompanying demo software (Katawa Shoujo Act 1), porting Ren'Py games to M22 is relatively easy. Most common functionality was simply in need of re-working to the syntax of M22, with the more in-depth functionality replicated using the script extension system (for example, *snow_effect* for creating a snow particle effect).

Multiplatform capabilities was achieved through the use of Unity; the developer can forego configuration of each platform (i.e. use the default settings) and compile the same visual novel for any platform with a touch-screen and/or any device with widescreen aspect ratios without any required changes (as the engine and Unity handle all of this). Because of this, theoretically less time is spent on these compatibility issues— lowering costs— and the increased variety of target platforms increases profits due to a wider reach to the audience.

REFERENCES

[1] 1998. KEY - A Japanese visual novel studio. http://key.visualarts.gr.jp/. (1998). Accessed: 2019-07-23.
[2] 2004-2019. Ren'Py - Python Visual Novel Interpreter. https://renpy.org. (2004-2019). Accessed: 2019-05-05.
[3] 2005-2019. Unity. https://unity.com/. (2005-2019).
[4] 2006. rlvm. https://github.com/eglaysher/rlvm. (2006). Accessed: 2018-12-01.
[5] 2007-2019. VNDB - The Visual Novel Database. https://vndb.org. (2007-2019). Accessed: 2019-05-05.
[6] 2011. Sony PlayStation Vita. https://www.playstation.com/en-gb/explore/ps-vita/. (2011). Accessed: 2019-07-23.
[7] 2013. libnpengine. http://dev.pulsir.eu/krofna. (2013). Accessed: 2018-12-01.
[8] 4LeafStudios. 2012. Katawa Shoujo. https://www.katawa-shoujo.com/. (2012). Accessed: 2019-05-05.
[9] 5pb. Games. 2017. Re:ゼロから始める異世界生活 -DEATH OR KISS- . PlayStation 4, PlayStation Vita. (2017).
[10] 5pb. Games. 2017. New Game! -The Challenge Stage!-. PlayStation 4, PlayStation Vita. (2017).
[11] 5pb., Nitroplus. 2009. Steins;Gate. [CD-ROM]. (2009).
[12] D.O. 2003. 雪桜 (Snow Sakura). https://vndb.org/v71. (2003). Accessed: 2019-05-05.
[13] Daniel Green, Charlie Hargood, and Fred Charles. 2018. Contemporary Issues in Interactive Storytelling Authoring Systems. *Interactive Storytelling Lecture Notes in Computer Science* (2018), 501–513. https://doi.org/10.1007/978-3-030-04028-4_59
[14] VisualArts KEY. 2004. Clannad. [CD-ROM]. (2004).
[15] KEY, VisualArts, Prototype. 2010. Clannad - 光見守る坂道で. [CD-ROM]. (2010).
[16] Sam Lynch. 2016. A C++ application to convert Ren'Py script to eBook-compliant narrative. https://github.com/Slynchy/RPY-eBook/. (2016).
[17] Sam Lynch. 2016. A C++ application to convert Ren'Py script to M22-Lua. https://github.com/Slynchy/March22-Lua/tree/master/sdk/RPYtoLua/. (2016).
[18] Jean Francois Puget and IBM. 2005-2019. A speed comparison of C, Julia, Python, Number and Cython on LU Factorization. https://www.ibm.com/developerworks/community/blogs/jfp/entry/A_Comparison_Of_C_Julia_Python_Numba_Cython_Scipy_and_BLAS_on_LU_Factorization. (2005-2019). Accessed: 2019-05-05.
[19] RockPaperShotgun. 2019. Half of all games on Steam came out since 2017. https://www.rockpapershotgun.com/2019/01/15/how-many-games-are-on-steam/. (2019). Accessed: 2019-05-05.