**RadIceCream: A RISC-V Assembly Game**

Nirva Neves de Macedo

Rodrigo Rafik

Mariana Simion

Department of Computer Science at University of Brasília

**A screenshot of a video game

Description automatically generated**

***Figure 1:*** *Introductory Image (concept arts, title, among other things)*

**0 Abstract**

This scientific paper describes the development process of the game RadIceCream, created by Computer Science undergraduates Nirva Neves de Macedo, Rodrigo Rafik Menezes de Moraes and Mariana Simion dos Santos, inspired by the previously Flash and nowadays HTML5 game BadIceCream by publisher and developer Nitrome. The game was developed in Assembly Language for RISC-V processors. More specifically, it was developed utilising the RISC-V Assembly Runtime Simulator (From this point on, referred to simply as “RARS”) as an Integrated Development Environment, in conjunction with the tool FPGRARS – a far less glitchy version of RARS itself that can run assembly code much more quickly. Together, these utilities made the project possible.

**1 Introduction**

BadIceCream is a browser game released in 2010 by the publisher Nitrome. While it is a very simple game, recreating this game in assembly language proved itself a quite extreme challenge. Much of the game had to be simplified to fit the schedule, however this does not mean there is no polish in the game’s assembly implementation. In fact, despite the enormous codebase, when well analysed most of the assembly “functions” in RadIceCream are indeed simple and are made to appear, as best as it could be done, organised.

RadIceCream is thus not an exact recreation – one could say it only recreates the original game’s mechanics, but in no way its art and style. In some areas, simpler art and style proved itself easier to implement. As such, this game attempts to subvert the original’s concept in a certain way, stylized much more ironically and presenting subtle satire, with the goal of simply adding to the fun factor. Moreover, the more complex enemies had to be disregarded in favour of possibly entirely new enemies (albeit not *that* exciting).

Beyond just the aforementioned tools, it should be mentioned that a couple of other key tools – which could have been alternatives – were vital for a much quicker development process. Mainly, paint.net, which is an image manipulation program that was widely used for the creation of sprites, backgrounds and other graphical elements; bmp2oac3.exe, which is a tool for conversion of bitmap image files into data files that can be accessed by the two runtime simulators used through a rendering function; Notepad++, which was used extensively as a text editor for data matrices and other data files with structures that were purpose-built for the game, given that it is a far more sophisticated editor than the simplistic one that comes built into RARS. Finally, GitHub and Git were used at all times after the first few functions were written and settled to keep track of changes and allow easier contribution by the project’s authors.

The file structure of the project is rather simple, being mainly broken down into three parts, merely for organisation. The main code is contained within the main folder, as well as the changelog and some of the tools used for development (not to mention this document itself also shares that space). Within the main folder are also subfolders, which are all accessed by the program during runtime. They are two: The sprites folder, which is self-explanatory, and the maps folder, which contains the matrices for each level, their backgrounds and other backgrounds used throughout the game, as well as level information file. All of these will be explained in detail throughout the Methodology section.

**2 Methodology**

This section will describe most of the development process. While a chronological description of that process would be most interesting, the authors have elected that, ultimately, the section should be categorised according to the problems that were faced during development and their subsequent solutions, as that provides much deeper insight as to how the game performs certain things such as loading a level or rendering something on the screen. As such, a few preliminary principles should be presented and explained first as they were followed throughout development and were important for the project’s conclusion.

**2.1 Modularity**

The first, and perhaps most important aspect of the codebase is modularity. Modular here defined as something that is easily modified and incremented. The code is written in such a way that adding a new function does not break others (or, if it breaks them, it’s still a very easily fixed problem). To achieve this, the order in which functions appear, the way loops are built and the way the data section is organised were thoroughly studied. Modularity was specifically important for the continued addition of features up to completion, as well as the subsequent changing of features that required changing for other features to work. In summary, it is an attempt to make the program as linear as possible – algorithm followed by algorithm such that one causes as little interference as possible in the other.

**2.2 Simplification**

Considering the game by Nitrome, RadIceCream is obviously far simpler. These simplifications should be clarified in this subsection. Firstly, one of the most obvious simplifications is the static map. Whereas in the original game, the camera moves and adjusts itself to the player’s position, such an effect would add enormous complexity to the game’s code. With that in mind, the approach taken was to present every level in the game statically – only the objects in the screen will move, not the screen itself. Another simplification, despite subtler, played a significant role in the differentiation of the games and adaptation to the game’s runtime environment: lower resolutions and overall simpler art. This is not to say that the art in the game had less effort put into it, but that the art style was adjusted to fit 16x16 cells in order to facilitate rendering. Tricks such as transparency are extremely tricky to implement, so breakable blocks with collectibles inside them were treated as a single universal entity which, once broken, would turn into whatever collectible is the current objective. Another obvious simplification is the animation of movement between cells – or rather, the lack thereof. Since in the game, the player’s movement is limited to certain spaces (the player cannot *stay* in a space in between two cells), we chose to render the game according to a matrix. One of the limitations posed by this choice was that it would be extremely difficult to render intermediary states, like the player or any enemy walking from one cell to the next. While, given enough time, this animation could have been implemented, the time was not enough. The last of the most significant simplifications are the menu layouts. Regarding these, a lot of liberty was taken to make the design usable without taking into account the original game’s design at all. Other minor simplifications are seen in the number of levels (which was a limit imposed by time constraints) as well as in music and sound effects (the MIDI system forced by RARS is very restrictive when it comes to sound design and sequencing.

**2.3 Documentation**

This project attempts its best to be as well documented as possible. Of course, code documentation within the project’s own files is rather limited and made possible mostly by comments explaining actions. Ideally, for code in assembly language, one should comment every few instructions such that it is clear what was just done. The approach for RadIceCream deviated slightly from that general rule – as it eventually becomes exhaustive to document every single line of code. Repeated code was usually not explained more than once and blocks of code with a singular purpose, if the registers used in which did not affect other blocks of code, were given a general explanation either above or below. Similarly, simpler blocks of code that update or reset a value were likewise given a general explanation. While this adds to the effort of reading and understanding the provided program,

**3 Obtained Results**

**4 Conclusion**

First and foremost, it should be noted that designing a game within the severe limitations of assembly language, as well as the limitations posed by the runtime simulator itself among other things, is not perhaps the best of ideas unless for scientific or entertainment purposes. Full games written in assembly nowadays are mostly projects for older hardware – mostly consoles. Games that *had* been written in assembly before all fall into one of those situations. Artistic expression is, perhaps, another applicability of creating a game using low-level mechanisms.

With that in mind, for educational purposes as was the objective of this project, the game proved itself a mind-tangling challenge, perhaps impossible to truly finish for all those who attempted it. It can easily be assumed that all participants and contributors had a learning experience that certainly builds upon the concepts learned in the Introduction to Computational Systems class and, moreover, refines that knowledge to whole new levels of understanding of the intricacies concerning the functionality of a computer system.

However, this was not the only area that was built upon throughout the project. All around, it involved concepts of game design and forced the authors to learn their way around concepts that, to a naive viewer who simply played the game, seem very simple, but are indeed quite complex. Manipulating matrices, navigating around data structures that the authors themselves designed shows that this was not only an exercise in assembly language itself, but an exercise in mathematics, logic, game design, arts and the real-world application of these subjects.

**5 References**