Using Multithreading in a 2D Physics Engine in the Context of Games

Calvin Barnes
UCF CECS Undergraduate Student
University of Central Florida
Orlando, Florida
ca435366@ucf.edu

Jacob Pacheco
UCF CECS Undergraduate Student
University of Central Florida
Orlando, Florida
ja566743@ucf.edu

Jacob Peach
UCF CECS Undergraduate Student
University of Central Florida
Orlando, Florida
ja389006@ucf.edu

Leo Salazar
UCF CECS Undergraduate Student
University of Central Florida
Orlando, Florida
le373424@ucf.edu

Pavan Tamma
UCF CECS Undergraduate Student
University of Central Florida
Orlando, Florida
line 5: email address or ORCID

Abstract - The purpose of this project is to achieve a calculation-heavy implementation of a game utilizing the Raylib library. These will be physics calculations, acting on objects within the game affected by the player. These calculations will be handled with concurrent programming, utilizing simple multithreading concepts, to achieve a highly optimized 2-dimensional physics engine. This engine will handle many different entities at the same time, which would dramatically affect the performance of a singly-threaded implementation of this game.

I. Background

Raylib is a c++ library designed with the purpose of making small game development simple, and approachable. It works based off of a game loop that continually calculates the game code, draws the output, then iterates to the next frame. It is in this context where the idea for our project lives; raylib's lightweight framework allows for programming games where large sections of code are able to be compiled completely independent of any functions within the raylib library. Some objects implemented with raylib functions may have their values accessed in these situations, but none of the raylib drawing functions (which sends data to the graphics pipeline) are utilized. This has implications for how concurrency is achieved in the implementation phase of the project.

II. Project Value

A. BSingle-Threading Issues

Raylib is a simple library with relatively easy to understand functions and classes. C++'s speed, coupled with the simple implementation of most raylib games, make for a process that is generally easy to run, making for performant games.

It is the contrast of this performance and the calculation heavy simulation of this game where a hypothetical bottleneck can be identified; every other portion of this program will be held back by the physics calculations, given enough game objects for physics to act upon.

B. Target and Benefits of Concurrency

The raylib library utilizes OpenGL for its graphics rendering, which will make multithreading this portion of the program unfeasible (it is technically possible, but would require technical workarounds involving multiple OpenGL contexts). Additionally, as the bottleneck of the program, this physics implementation is the place where the concurrency will give us the greatest (or any) benefit. As such, the target of this multithreading is clear, as is the benefit of it; at a high enough number of objects, in which the effect on the number of calculation can have a value greater than the sum of its parts, the theoretical benefits of multithreading (per Ahmdal's Rule) would have a desirable effect on the performance of the program, by many metrics, and in many contexts.

III. Implementation

We have decided to program two games as separate entities to the main program, for the purpose of making "proof of concept" builds that illustrate the likely performance increase we'll see in the finished product. The benefits of this process are as follows:

- Not all of us are familiar with Raylib, and needed some "practice projects" in order to get ourselves used to the functions that the library provides. We decided that, if this were necessary, it would be best to practice with projects whose concepts/goals were close to that of our ultimate goal. The Rain program would be used for practicing gravity concepts, and the Particle Accelerator program would be used for practicing collision concepts.
- These two programs also proved the benefits of concurrent programming to us. The simplicity of these programs allowed us to quickly gauge the

prospective value of this project, by testing multithreading on multiple components of a physics engine in a vacuum

These Two programs, both with a single threaded and multi-threaded version, are our Particle Accelerator, and our Rain simulator.

A. *Particle Accelerator*

The Particle Accelerator program (Depicted in Fig. 1) consists of ten thousand differently colored particles that bounce around the screen. The main function of this simulation is window bound detection and collision, specifically in the context of reflecting speed values along specific axes, to maintain realistic angles.

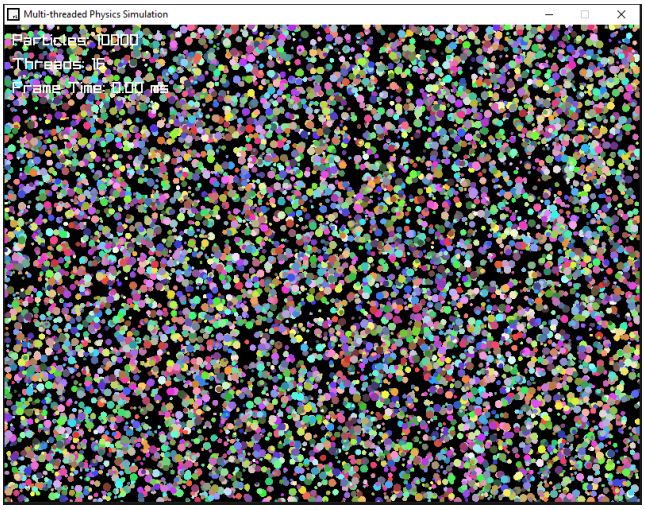

Fig. 1. The Particle Accelerator Program. Metrics such as frames per second (FPS), frame time, and particle count can be seen in the corner.

The structure of this project mirrors that of the final goal; the standard Raylib gameloop consists of the setup in the main function, and game loop setup, where the window is drawn and redrawn as fast as directed by the FPS target, until the program ends.

The point of concurrency is in the calculation of the particle's collision with the screen bounds, which are detected by continually checking each particle's position, every single frame. This is designed to be taxing to system performance at such a high number of particles. Multithreading for this program occurs on a per-particle basis, where one thread computes the direction of one particle, and moves on to the next one.

(Need paragraph for performance statistics in the Particle Accelerator Program)

B. *Rain*

The Rain Program (Depicted in Fig. 2) consists of two thousand five hundred rain particles that spawn at the top of the screen and fall down to the bottom, as a function of gravity. The main function of this program is to accurately simulate and multithread a realistic looking gravity, which pulls each raindrop down to the bottom of the screen, often at differing speeds.

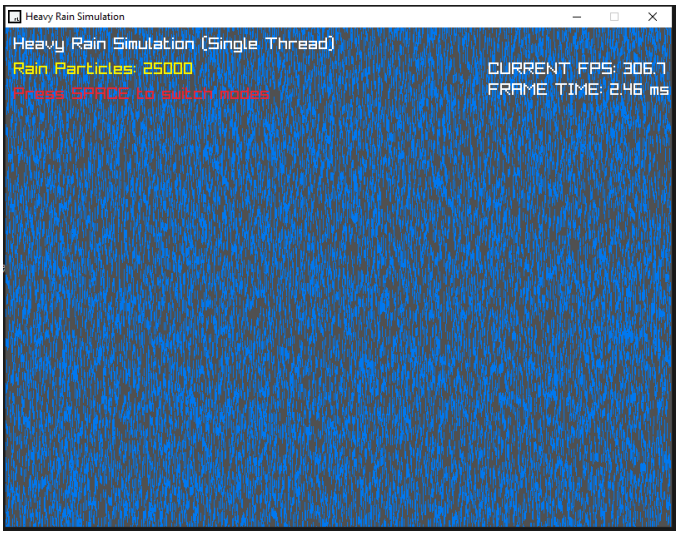

Fig. 2. The Rain Program. Metrics such as frames per second (FPS), frame time, and raindrop count can be seen at the top.

This program, too, is of a similar structure to the main game, where the components that end up being multithreaded are easily identifiable and have their own context, and thus not subject to some of the quirks that come with trying to multithread Raylib functions directly.

(Need paragraph for performance statistics in the Rain Program)

IV. Performance Interpretation

Many people familiar with video games understand the concept of frame rate, or frames per second. Generally, for titles that the average person plays, the better the game runs (i.e. the more performant it is), the more FPS is achieved. Perhaps one of the most important parts of keeping a smooth experience is making sure the game always runs at the same speed, regardless of frame rate. When this is achieved, we say that the game speed is "not tied" to the frame rate. In essence, the game progresses just as fast no matter how frames per second the program puts out.

The simulated examples depicted above are not so sophisticated; they are implemented in a way such that there is an amount of logic done per frame, rather than there being an amount of frames rendered per specific amount of logic (The smarter implementations that aim to decouple game logic speed from frame rate do so by taking into account the time taken between each rendered frame (i.e. frame time), thereby preserving game speed even when performance dips affect frame rate).

What this means is, with the previous two examples, the most obvious way to determine that the game is performing better in a multi-threaded implementation is by seeing if all the particles are physically moving faster across the screen.

V. Main Program

The main program (depicted in Fig. 3), titled 2D Physics, combines the functions we practiced and verified in the previous two examples.

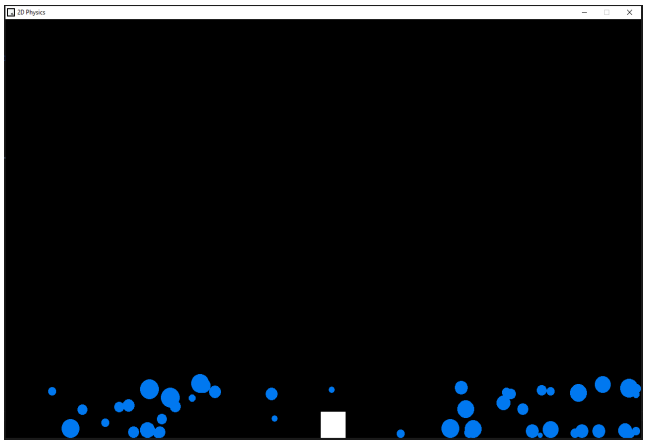

Fig. 3. The 2D Physics Program. Blue balls (The particles) have physics events imparted onto them by the white box (The player).

Each particle in this demo is defined in a vector array, in a similar manner to the previous examples in order to make the multithreading process simple. The bounds-checking collision shown in the particle accelerator is employed here, as well as a rudimentary gravity constant reminiscent of the rain demo.

As added functionality, the player character can be controlled with the left or right arrow keys, gaining a speed that is, upon collision with one of the two particles, imparted onto it uniquely. All of the programming that goes into calculating a single particle's speed is done completely separate from other particles. This is done not only to make multithreading possible, but also to facilitate the addition of interpersonal physics calculations so that the particles can affect each other.

A. *Multithreaded vs Single Threaded Performance Comparison*

Frame time, graphs, etc., possibly at multiple predetermined instances to standardize performance metrics (think performance at 1,000 objects, performance at 10,000 objects).

VI. Project Retrospective

Answer high level questions regarding the "value" of concurrency, by showing how the performance was enhanced by the multithreaded process. What kind of performance gain should be expected? Is there a point of diminishing returns for multithreading when taking into account other potential performance bottlenecks of the engine?

TABLE I. Table Type Styles

Table Head	**Table Column Head**		
	Table column subhead	***Subhead***	***Subhead***
copy	More table copy[a]		

[a] Sample of a Table footnote. (*Table footnote*)

Fig. 1. Example of a figure caption. (*figure caption*)

Acknowledgment *(Heading 5)*

The preferred spelling of the word "acknowledgment" in America is without an "e" after the "g". Avoid the stilted expression "one of us (R. B. G.) thanks ...". Instead, try "R. B. G. thanks...". Put sponsor acknowledgments in the unnumbered footnote on the first page.

References

The template will number citations consecutively within brackets [1]. The sentence punctuation follows the bracket [2]. Refer simply to the reference number, as in [3]—do not use "Ref. [3]" or "reference [3]" except at the beginning of a sentence: "Reference [3] was the first ..."

Number footnotes separately in superscripts. Place the actual footnote at the bottom of the column in which it was cited. Do not put footnotes in the abstract or reference list. Use letters for table footnotes.

Unless there are six authors or more give all authors' names; do not use "et al.". Papers that have not been published, even if they have been submitted for publication, should be cited as "unpublished" [4]. Papers that have been accepted for publication should be cited as "in press" [5]. Capitalize only the first word in a paper title, except for proper nouns and element symbols.

For papers published in translation journals, please give the English citation first, followed by the original foreign-language citation [6].

[1] G. Eason, B. Noble, and I. N. Sneddon, "On certain integrals of Lipschitz-Hankel type involving products of Bessel functions," Phil. Trans. Roy. Soc. London, vol. A247, pp. 529–551, April 1955. *(references)*

[2] J. Clerk Maxwell, A Treatise on Electricity and Magnetism, 3rd ed., vol. 2. Oxford: Clarendon, 1892, pp.68–73.

[3] I. S. Jacobs and C. P. Bean, "Fine particles, thin films and exchange anisotropy," in Magnetism, vol. III, G. T. Rado and H. Suhl, Eds. New York: Academic, 1963, pp. 271–350.

[4] K. Elissa, "Title of paper if known," unpublished.

[5] R. Nicole, "Title of paper with only first word capitalized," J. Name Stand. Abbrev., in press.

[6] Y. Yorozu, M. Hirano, K. Oka, and Y. Tagawa, "Electron spectroscopy studies on magneto-optical media and plastic substrate interface," IEEE Transl. J. Magn. Japan, vol. 2, pp. 740–741, August 1987 [Digests 9th Annual Conf. Magnetics Japan, p. 301, 1982].

[7] M. Young, The Technical Writer's Handbook. Mill Valley, CA: University Science, 1989.

[8] K. Eves and J. Valasek, "Adaptive control for singularly perturbed systems examples," Code Ocean, Aug. 2023. [Online]. Available: https://codeocean.com/capsule/4989235/tree

[9] D. P. Kingma and M. Welling, "Auto-encoding variational Bayes," 2013, arXiv:1312.6114. [Online]. Available: https://arxiv.org/abs/1312.6114

[10] S. Liu, "Wi-Fi Energy Detection Testbed (12MTC)," 2023, gitHub repository. [Online]. Available: https://github.com/liustone99/Wi-Fi-Energy-Detection-Testbed-12MTC

[11] "Treatment episode data set: discharges (TEDS-D): concatenated, 2006 to 2009." U.S. Department of Health and Human Services, Substance Abuse and Mental Health Services Administration, Office of Applied Studies, August, 2013, DOI:10.3886/ICPSR30122.v2