Individual Project: N-Body

Alex Janninck
ajanninc@purdue.edu
LC5-12

Introduction

My program is an n-body gravitational physics simulation. I chose to make this because it's something that I've wanted to make for a long time, but I've never taken the time to actually do it. It is able to simulate the physics between any number of bodies/particles (assuming enough computational power).

Features *v0.5.4*:

- Partially parallelized physics simulation
- Adaptive simulation steps per frame
- Automatic deletion of particles that have drifted far away from main star (optional)
- Dynamic particle radius based on mass
- Particle collision with conservation of momentum and mass
- Settings configuration file (optional)
- 2 different random particle distribution methods w/ parameters editable in ".ron" config files
- Automatic file picker refreshing
- Configuration file cache
- Graphical User Interface
    - Light and Dark mode
    - Editing settings
    - Selecting configuration files
    - Viewing configuration files
    - Viewing live stats on simulation
        - Moveable
        - Collapsible
        - FPS and frame time displayed using a moving average
- Able to be downloaded as a standalone .exe file
    - Able to generate its own config files
- Rock solid user input and config file checking
- Impossible to crash (baring issues with system calls like reading files)
- 100% memory safe (no memory leaks)

Function Descriptions

Since there are 128 things (94 functions, 8 structures, 3 enums, 16 modules, 7 constants), I will not discuss how they each work in this section. All[1] of them have been explained with doc comments (triple slashes) which can be read on GitHub (link in the Code Appendix) or in the auto-generated documentation.

General Overview

main::main
The window is configured, and it calls the start_screen::start_screen function which shows the splash screen and configuration screens. Then the particles are randomly generated and passed to the game loop

start_screen::config_screen
Creates a new start_screen::param_edit::Persistence struct to maintain state between each frame of the UI.

particle::Particle
The structure that represents a particle. Many functions are implemented for it to do various things.

particle::Particle::grav
Takes 2 RefCell<&mut Particle> as references. RefCell's are used to appease the rust borrow checker. RefCell's enforce rust's borrowing rules at runtime instead of compile time. RefCell's allow the value stored within them to be mutable even if the RefCell itself isn't. This function calculates the acceleration due to gravity and adds it to each particle's acceleration field. It does NOT move the particle or change velocity (that is done by particle::Particle::step).

particle::Particle::step
This moves a Particle according to its velocity, acceleration, and delta-t passed in as a parameter. Once this is done, acceleration is set to 0.

particle::Particle::check_collision
This uses the radius of the 2 Particle's to determine if they are overlapping (and thus collided). It uses RefCell's just like particle::Particle::grav. It ignores collisions if either of the Particle's have already collided. Otherwise, if there is a collision it marks Particle B as having collided, adds its mass to

[1] Some functions that just implement traits haven't been documented since the trait explains what it does. I also haven't documented unit tests.

Particle A, and recomputes the radius of Particle A. Velocity of Particle A is also adjusted such that momentum is conserved (inelastic collision).

particle::Particle::RandomParticleGen
(aka particle::Particle::random_particle_gen::RandomParticleGen)
A trait to define what functions must be implemented by a struct to be a RandomParticleGen. This allows us to have variables that could be either a particle::Particle::PlainRandomGen or particle::Particle::BeltRandomGen, or any other future struct that implements this trait. This makes it *relatively* easy to add more random distribution methods in the future. I couldn't make Serialize / Deserialize a required trait for this trait, so I created config::DistributionMethod. (I don't believe it is possible to make them required since the Serde crate needs to what type it is deserializing)

config::DistributionMethod
(aka config::DM)
Allows us to serialize / deserialize either a BeltRandomGen or a PlainRandomGen from a file (or really any string or stream of bytes). Since a RandomParticleGen trait object doesn't implement Serialize / Deserialize[2], I needed to make this enum to wrap around it. A couple of the functions implemented on this enum return a Box<dyn RandomParticleGen>. The dyn keyword tells Rust that RandomParticleGen is a trait object and we want to use dynamic dispatch for function calls on it. A Box (smart pointer) stores a value that would normally be on the stack in the heap. The Box itself has a known size at compile time, but it allows the contained value to have an unknown size at compile time (e.g, if using a trait object).

game_loop::game_loop
This function does all the physics and UI stuff after the settings and config stuff is done. Before starting the simulation, it eliminates overlapping particles. Each graphical frame, the screen is cleared, camera zoom controls are checked, aspect ratio is corrected, and the physics loop is run. After the physics loop for the current graphical frame is complete, particles that are too far from the star are deleted if the kill distance option is enabled. After that, all the particles are drawn and the fps and frame time moving averages are adjusted, and the UI is drawn. Then, the game_loop::adaptive_simrate function is called.

---

[2] Although both of the structures which implement the RandomParticleGen trait also implement Serialize / Deserialize, it is possible to implement the RandomParticleGen trait without Serialize / Deserialize.

game_loop::adaptive_simrate
This function increases/decreases the number of physics frames per graphical frame to maintain a certain FPS. It only adjusts if the current framerate is a certain amount greater/less than the goal.

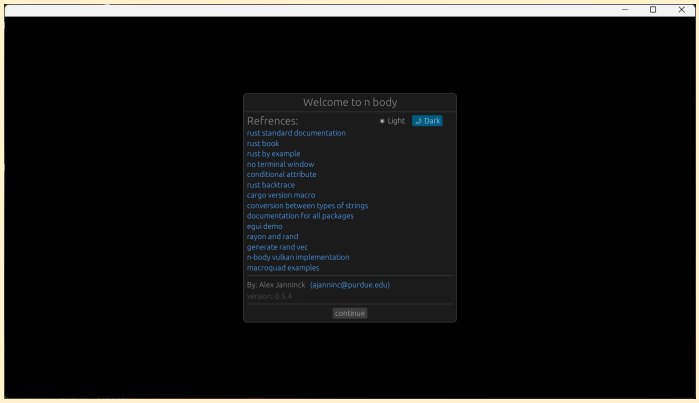
physics::physics_loop
Apply gravity, move particles, check for collisions, and delete collided bodies for each physics frame.

User Manual

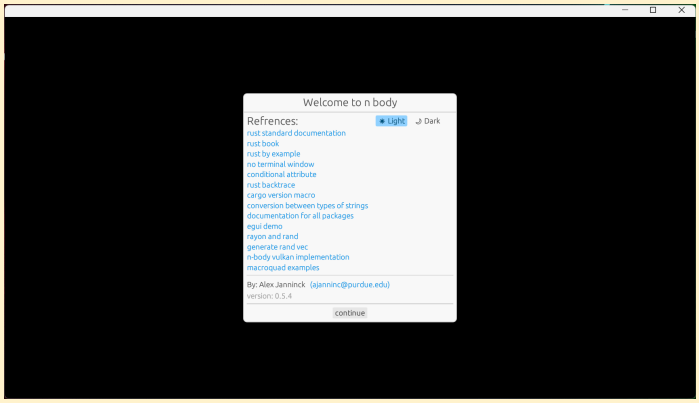
Welcome Screen

Upon launching the app, you will be greeted by the welcome screen. There are a pair of buttons in the top of the dialog that allow you to switch between dark and light mode. Below are links to various resources I used to assist me in programming this app. Below is my name, a link to email me, the version of the app, and the continue button. When you are ready to continue, press the continue button.

Figure 1



welcome screen (dark)



welcome screen (light)

Settings Screen

In the settings screen the light/dark mode buttons are still present. In the top portion of the dialog is where the settings can be tweaked to your liking. The setting scan be edited by clicking on them and typing in a new value and then pressing enter, escape, or clicking off of it. All of the settings with the exception of “#of particles” can also be edited by clicking on it and dragging your mouse left or right.

Description of Settings

- time multiplier: A multiplier to increase or decrease how fast everything moves.
- simulations per frame: Number of physics simulations done per frame shown on screen initially. This value will change due to adaptive sim rate.
- # of particles: Number of particles to be randomly generated. NOTE: any overlapping particles will collide into 1 particle

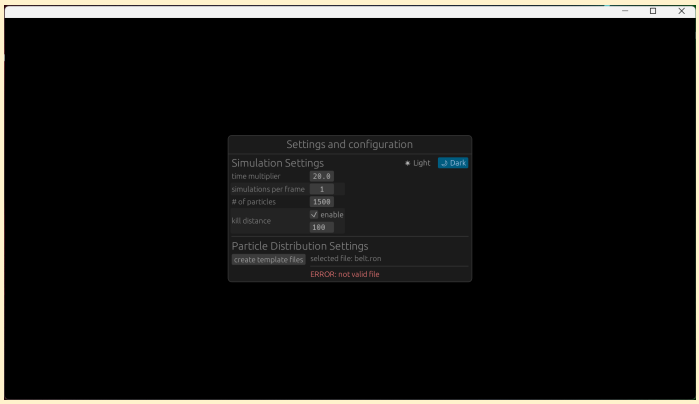
- kill distance (checkbox): allows you to enable or disable kill distance
- kill distance (input): only shows up if you enable kill distance. Kill distance is the maximum distance a particle can drift from the star before it is considered adrift and is deleted.

Particle Distribution Settings

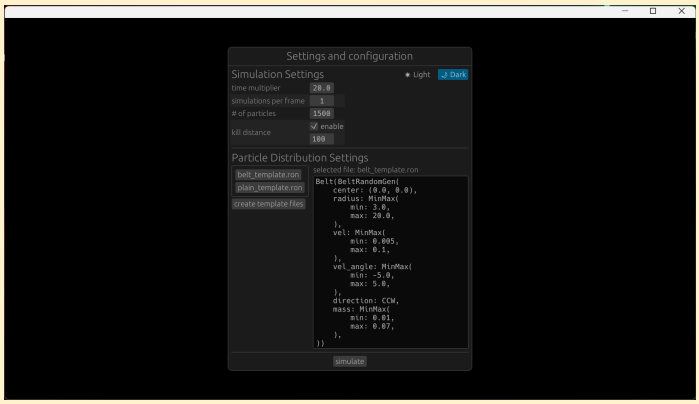
Located in the lower portion of the dialog, there is the particle distribution settings. It is split in half. On the left you have the file picker and the “create template files” button. Only .ron files that aren’t settings.ron will show up in the file picker (files are refreshed every 0.25 to 1.66 seconds). To select a file, simply click it. If you would like to create template files, click the button to do so. Once a valid file is selected it will show up on the lower right half of the dialog; if the file is invalid, it will display an error message instead. To adjust the random distribution configuration, simply edit the .ron file in a text editor of your choice.

Once you are happy with the settings, hit the simulate button. The simulate button is only shown when there is no issue with any of the settings.

Figure 2



no .ron files



a valid .ron file is selected and displayed

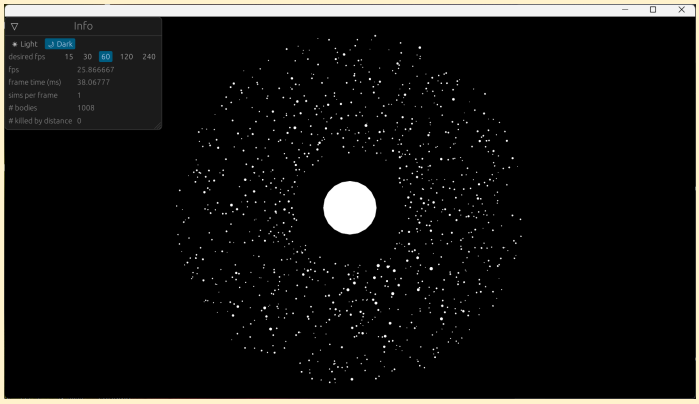
Simulation

The simulation screen has an info box in the top left which can be collapsed and moved around to wherever you want on screen. The light/dark mode buttons still remain, followed by the adaptive FPS goal buttons. You can use these buttons to set what FPS you want the adaptive sim rate to try to achieve. It will adjust the

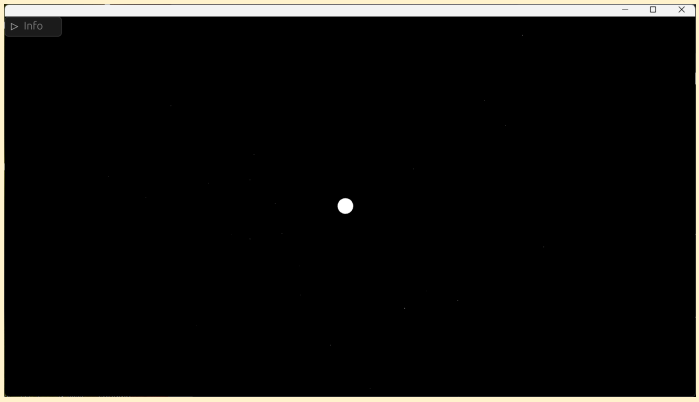
number of simulations per frame (also shown) to the best of its ability to achieve the goal.

The view of the simulation can be zoomed in/out using the +/- keys respectively.

Figure 3



screenshot of simulation



screenshot of simulation w/ info box collapsed

Code Appendix

Pasting all 1,500 lines (exactly) of code in here takes up 25 pages with a miniscule font, broken tabbing, and wrapped lines. It's hard to read because of that. Instead, here is the GitHub link. All the code is in the src folder.