- Problem description

For this project, I wanted to create a Mandelbrot platformer. I have always liked exploring the Mandelbrot set, but the only visualizations that exist of it are either long zooms on a chosen point or random pan/zooms in a web browser/standalone application. While these are sufficient, I wanted to create a demo of something a lot more interactive.

Mandelwalk is a unique Mandelbrot platformer where you play as a cursor that can traverse the Mandelbrot boundary. The cursor can zoom in and out from the Mandelbrot set, and can even jump off of it onto other parts of the Mandelbrot fractal. All assets for the project (music, sound effects, shaders, code) were produced by me.

- Language/library/etc. used and why GPU used

The main challenge of developing Mandelwalk comes from how it utilizes an intersection of many different languages/libraries. The main engine runs in Godot 4.2, which provides the main renderer, a Pythonic scripting language, and a scene editor to organize the code within.

Mandelwalk defers all of its rendering instructions down into Rust, using a custom Node class called the MandelbrotImageBuilder. This class creates the image data (the Mandelbrot set visual) and terrain data (edge detection based off of the image data) for use in both visual and gameplay.

The Rust code at this layer uses an OpenCL crate to further defer the rendering instructions into kernels supplemented by the OpenCL library.

Links to each language/library are provided here:

- Godot 4.2: https://godotengine.org/download/archive/4.2-stable/
- Godot Rust: https://godot-rust.github.io/
- Rust OpenCL: https://github.com/cogciprocate/ocl

- How GPU used, in detail

The OpenCL library used for the project exposes a similar interface for creating and executing GPU kernels similar to CUDA. The kernels are located in rust/src/lib.rs as raw strings, and are composed using Rust OCL's standard interface.

Rust OCL has a fantastic abstraction on top of the OCL API, allowing it to automatically manage the platform context, device management, command queue, and the majority of the kernel execution functions. A good reference for how it abstracts all of the instructions is available here:
https://github.com/cogciprocate/ocl/blob/master/ocl/examples/trivial.rs

- How work is evaluated

In my initial proposal, I wanted my primary evaluation to be whether or not the game is functional. I still consider this a fantastic goal, but I also wanted to define some finer performance benchmarks to demonstrate that the project itself could not be functional by itself within base Godot.

I chose to benchmark the following functions to perform and compare between Godot and Rust+OCL calls:

1. Populating a PackedByteArray with 255
    - A PackedByteArray is the packed byte array type for Godot.
    - The array size is WIDTH * HEIGHT * 3.
    - This serves as a control for the other tests.
2. Generating the Mandelbrot set in a PackedByteArray
    - This uses the Escape time algorithm described here: https://en.wikipedia.org/wiki/Plotting_algorithms_for_the_Mandelbrot_set
    - This algorithm populates a WIDTH * HEIGHT * 3 sized array to generate RGB color data.
3. Generating the Mandelbrot terrain in a PackedByteArray
    - The terrain calculation populates a WIDTH * HEIGHT sized array with “normals” for each pixel on the image data.
    - The point normal is calculated by taking the 8 neighbors of a given point, determining if they were at max iteration, and encoding the present neighbors within a byte.

Each of these benchmarks occur across four different screen resolutions, with one or three bytes used per pixel:

1. 288x162 (this is 1/4th the game viewport, and is used in-game)
2. 576x324
3. 1152x648
4. 2304x1296

- Results of work and evaluation

I’ve uploaded a video demo of the project here:
https://www.youtube.com/watch?v=mVrwSeKbk4M

Test benchmarks are below.

CPU: Intel i7-10700F CPU @ 2.90GHz
GPU: NVIDIA GeForce RTX 2070 SUPER

Test 1 (control): populating a PackedByteArray with 255, size WIDTH x HEIGHT x 3
These were ran and averaged with 100 iterations.

Resolution	Godot time	Rust+OCL time
288x162	0.015 sec	0.117 sec
576x324	0.060 sec	0.167 sec
1152x648	0.242 sec	0.348 sec
2304x1296	0.961 sec	1.074 sec

The Rust+OCL methods take approximately 0.1 seconds longer to populate these arrays than the Godot code. This makes sense, as the Rust+OCL code uses the exact same function calls to populate the PackedByteArray as the Godot code, and has the added the overhead of populating an array in OCL.

This is a useful control test, as it demonstrates that the minimum overhead for deferring calculations to Rust+OCL is 0.1 seconds.

Test 2: Generating the Mandelbrot set in a PackedByteArray, 256 MAX_ITERATION, size WIDTH x HEIGHT x 3 (RGB image with color data passed into the functions)
These were ran and averaged with 10 iterations.

Resolution	Godot time	Rust+OCL time
288x162	0.675 sec	0.128 sec
576x324	2.665 sec	0.169 sec
1152x648	10.761 sec	0.348 sec
2304x1296	43.848 sec	1.083 sec

Here we see the performance gains of the Rust+OCL code are extremely present. At the in-game resolution, it is around 5 times faster to render the set in Rust+OCL than in Godot. The function also scales much better for higher resolutions as well, suggesting that I could have ran the game at much higher resolutions as well (though I didn't since I

preferred the aesthetic). Nevertheless, this is a sizable enough improvement that I believe demonstrates that the game heavily benefits from the presence of GPU calculations – reducing the lag between zooms and re-renders helps keep the game feel stable and fun.

Test 3: Generating the Mandelbrot terrain in a PackedByteArray, size WIDTH x HEIGHT
These were ran and averaged with 10 iterations.

Resolution	Godot time	Rust+OCL time
288x162	0.026 sec	0.140 sec
576x324	0.104 sec	0.211 sec
1152x648	0.421 sec	0.526 sec
2304x1296	1.71 sec	1.705 sec

Surprisingly, Godot beats Rust+OCL times across the board here.

I believe this is because the actual terrain algorithm here is far less intensive and instruction-lighter than the Godot code. Comparatively, the times are similar to the control test. So, Rust+OCL turns out to not be the most optimal way to calculate the terrain set for this project due to the kernel overhead alone. However, it does scale better at higher resolutions, implying that Godot only succeeds at the game resolution here because of the kernel overhead.

The algorithms could have been further analyzed, benchmarked and optimized, but I did not have the time to polish them much further.