CS 488 Final Project

Term: Fall 2023

Name: Devin Leamy

UW ID: 20872933

UW User ID: dleamy

Eva

A WebGPU Real-time Ray Tracer Written in Rust

Contents

1. Overview

Eva is a real-time ray tracer built in Rust using WebGPU, with an integrated Python3 scripting API. Unlike WALL-E (A4) which took hours to render a nice image, Eva is hip and modern. Eva can render at 850x850 resolution with 16 samples per pixel at 60FPS on an M1 Max Macbook Pro. Eva can also go big, rendering images with over 1000 samples and hundreds of reflections per pixel in only a couple of minutes!

2. Features

2.1. Texture Mapping

Any material can be assigned a texture. Textures are sourced from: `/assets/textures`.

```
# Load a texture.
texture_handle = Eva.add_texture("texture.png")

# Add the texture to a material.
textured_material = Material(
  1.0,
  0.0,
  (1.0, 1.0, 1.0),
  texture=texture_handle
)

# Add the material to some geometry.
box = Box()
box.set_material(textured_material)
```

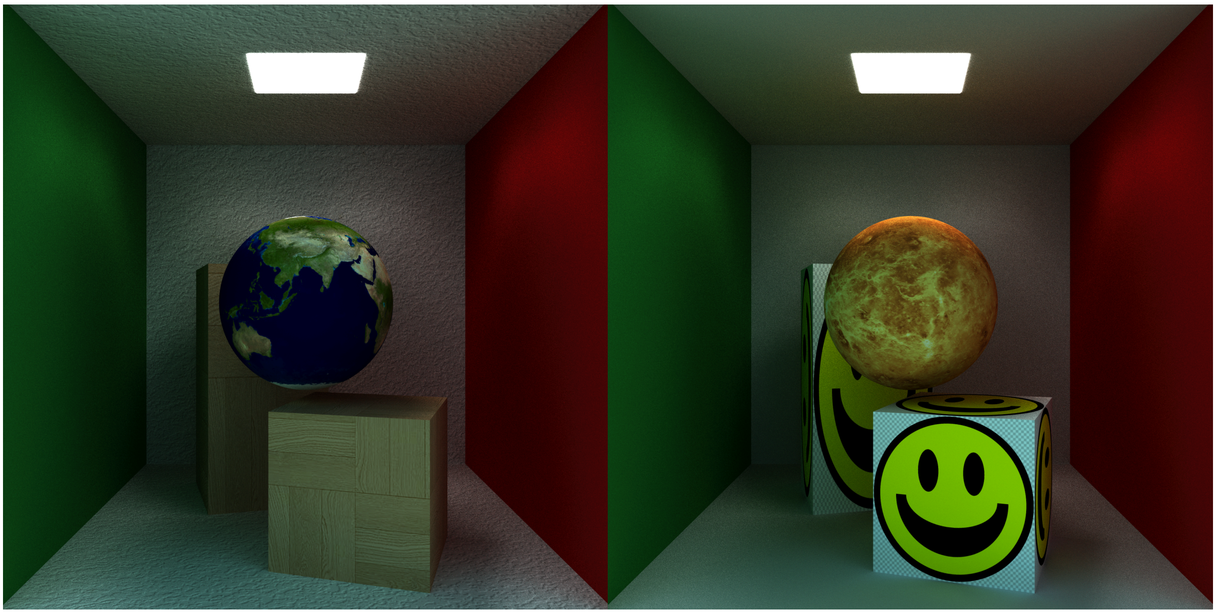

Figure 1: Textured Materials

2.2. Skyboxes

Scenes can optionally set a skybox. Skyboxes are sourced from: `/eva/skybox`. Skyboxes are defined by six images, listed in the order: [“x”, “-x”, “y”, “-y”, “z”, “-z”], defining the six faces of a cube.

```
Eva.add_skybox([
    "clouds/x.png",
    "clouds/-x.png",
    "clouds/y.png",
    "clouds/-y.png",
    "clouds/z.png",
    "clouds/-z.png",
])
```

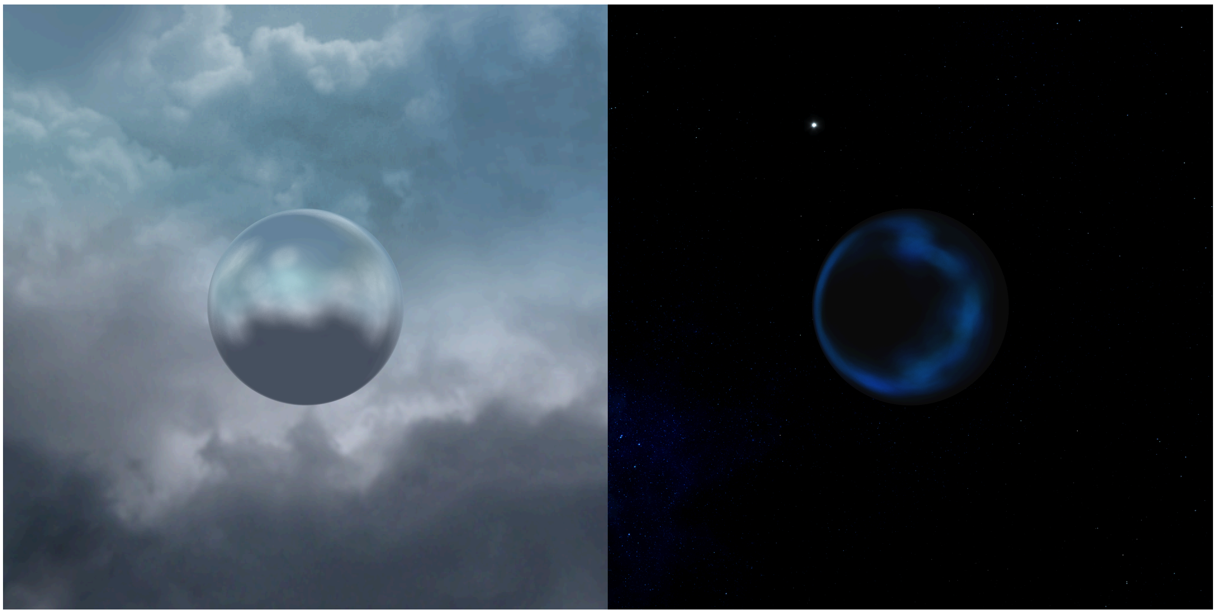

Figure 2: Day and Night Skyboxes

2.3. Phong Shading

Eva can render `.obj` meshes with triangular faces. If the mesh has vertex normals, Phong Shading is applied.

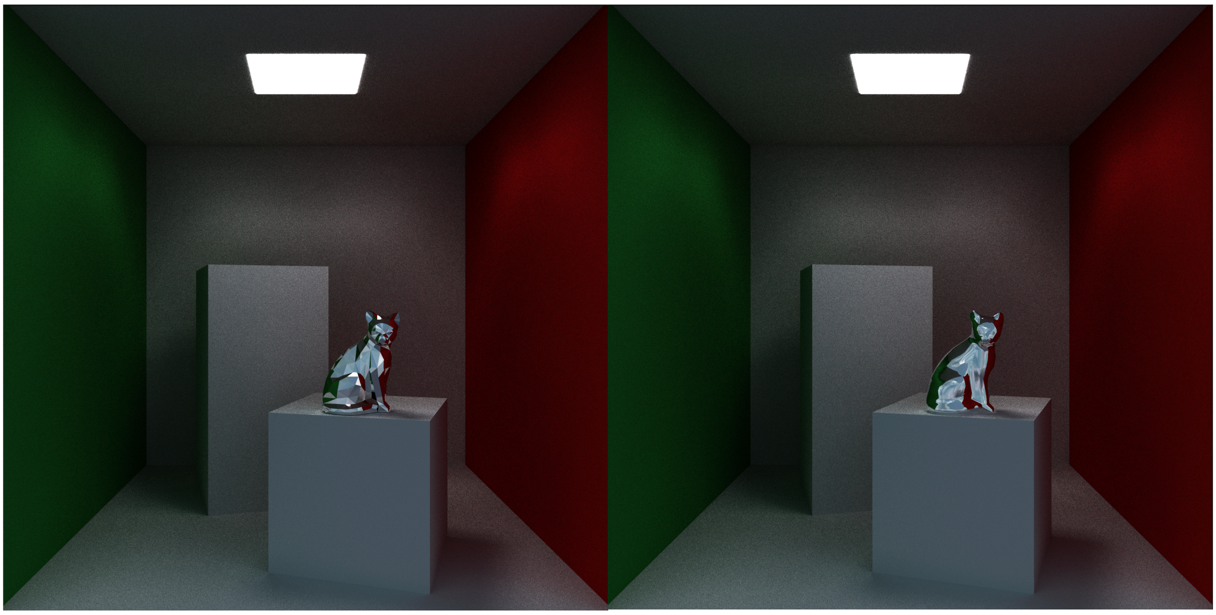

Figure 3: Cat Phong Shading

2.4. Real-time Ray Tracing

Eva supports two render modes `RenderStatic` and `RenderDynamic`. Implementing `RenderDynamic` makes your application real-time, and provides `update` and `handle_input` methods.

```
class Realtime(RenderDynamic):
  def __init__(self):
    super().__init__()

    self.cube = Box()
    self.add_geometry(cube)
```

```python
  def update(self):
    self.cube.rotate_x(1)

  def handle_input(self, key, state):
    # Move the camera left and right in response to input.
    if state == "Pressed" and key == "A":
      self.camera.translate(-1, 0, 0)
    if state == "Pressed" and key == "D":
      self.camera.translate(1, 0, 0)
```

2.5. Reflections

Rays can reflect off of metallic surfaces. The maximum ray reflections can be configured:

```python
Eva.set_max_reflections(100)
```

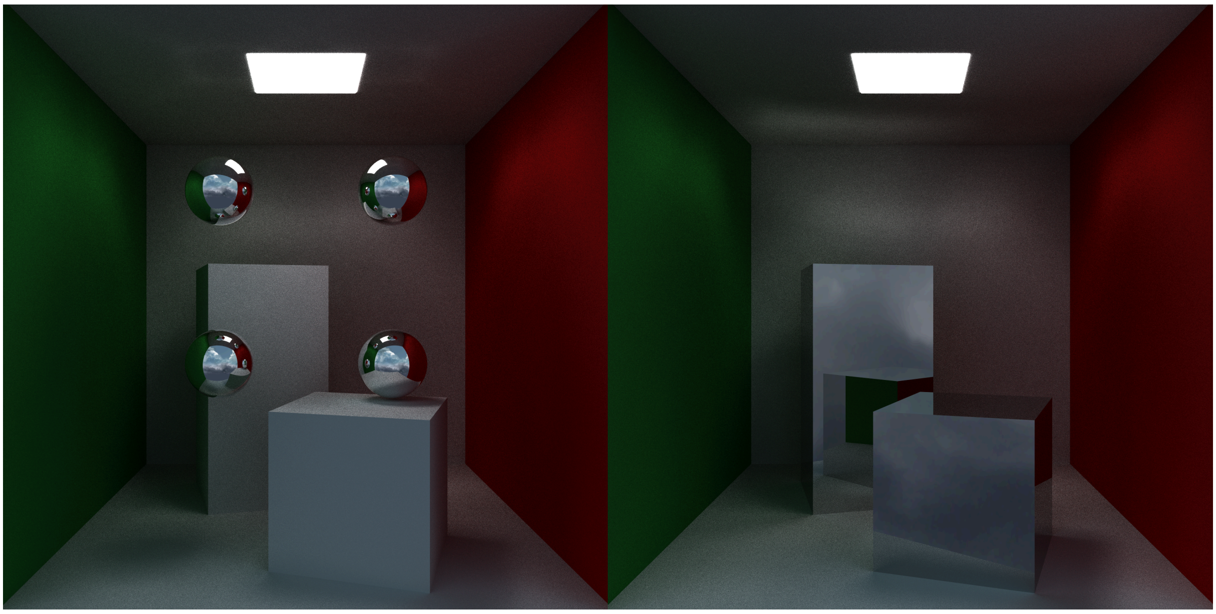

Figure 4: Reflective Surfaces

2.6. Python Scripting

Eva is divided into two core components: `/eva` and `/eva-py`. `/eva-py` defines a scripting API for the `/eva` renderer. Scripts are sourced from `/scripts`.

Scripts can be run using the utilities `run.sh` and `debug.sh`. `debug.sh` will display build logs.

To run a script, `my-scene.py` execute:

```
./debug.sh my-scene
```

2.7. Materials

A material is defined by a `roughness`, `metallicness`, `color`, an optional `texture`, and an optional `emissiveness`.

```python
ruby = Material(0.1, 1.0, (1, 0.1, 0.1))
blue_light = Material(0, 0, (0, 0, 1), light=(0.0, 0.0, 1.0))

earth_handle = Eva.add_texture("earth.jpeg")
earth = Material(0, 0, (1, 1, 1), texture=earth_handle)
```

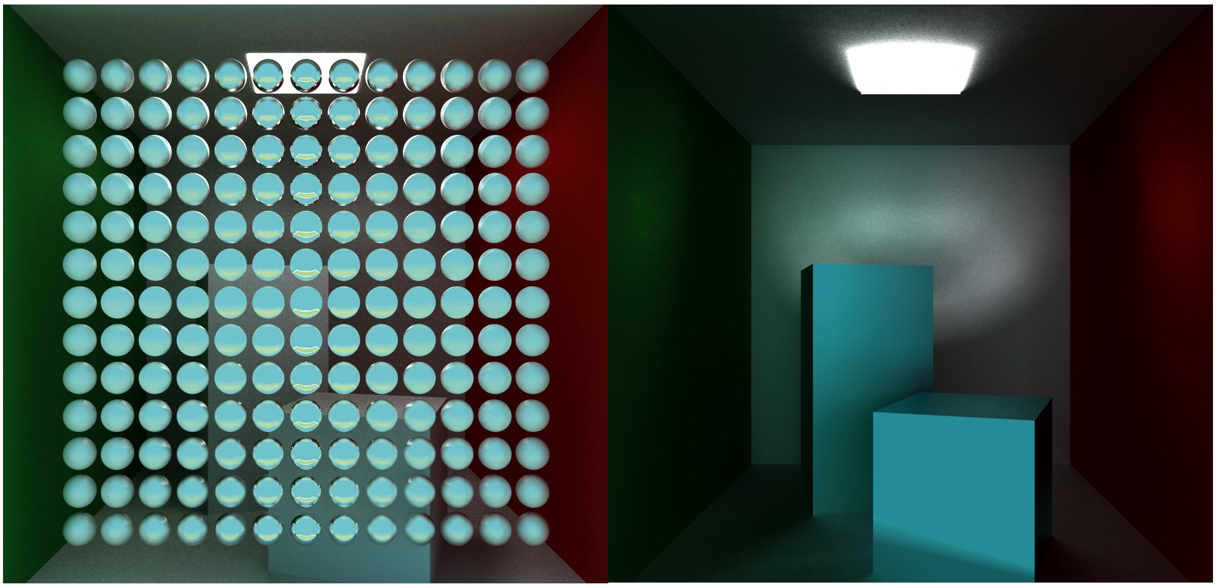

Figure 5: Different Kinds of Materials

3. Technical Overview

3.1. Project Structure

- `/eva`: Core renderer.
- `/eva-macros`: Marcos for the core renderer.
- `/eva-py`: Python3 scripting API.
- `/eva-py/python/eva_py`: Pure Python3 wrapper on the Rust bindings.
- `/eva-py-macros`: Macros for the Python3 scripting API.
- `/scripts`: Python3 scripts.

Note: In Rust macros must be put into a separate crate.

3.2. Ray Tracer

The ray tracer lives in `/eva`. It's written in 100% safe Rust and WGSL (WebGPU Shading Language).

- `/eva/src`
  - `/shader`: Types that can be loaded into the WGSL shaders.
  - `/scene`: Scene definition.
  - `/renderer`: The machinery that creates the WebGPU primitives, loads GPU data, and runs the shaders.
- `/eva/shaders`
  - `display.wgsl`: Fragment and vertex shader.
  - `ray_tracer.wgsl`: Ray tracer compute shader.

Eva uses `winit` as the windowing API. It's cross-platform and the defacto API in the Rust ecosystem. After creating a `winit::Window`, to use the ray tracer, you first create a `StaticRenderContext` which contains information about your scene that will not change. This includes the loaded textures, materials, and some parameters for the ray tracer. Using the `StaticRenderContext` and the `Window` you can build a `Renderer` using the `RendererBuilder`.

```
let window: winit::Window = todo!("create a window");
let static_context: StaticRenderContext = todo!("create a static context");
let mut renderer = RendererBuilder::new(window, static_context).build();
```

Eva uses wgpu to access WebGPU. The RendererBuilder will create the WebGPU buffers, bind group layouts, bind groups (where possible), pipelines, textures, shader modules, and create all of the required WebGPU core API components including the Device, Queue, Surface, and Adapter. build() then loads this state into a Renderer.

The Renderer has one function, render, which takes in a DynamicRenderContext. The DynamicRenderContext contains things that *will change*. This includes the positions of objects in your Scene, and the position and orientation of your Camera.

```
let mut renderer: Renderer = RendererBuilder::new(window, static_context).build();
let mut dynamic: DynamicRenderContext = todo!();
loop {
  renderer.render(dynamic);
  update(dynamic);
}
```

On render, all of the data is loaded into the shaders and the render commands are encoded using a WebGPU CommandEncoder. There are two* render passes, a ComputePass and a RenderPass. The ComputerPass, which uses ray_tracer.wgsl, will run ray_tracer.wgsl once for each pixel in the screen using "working groups" (a collection of runs of a compute shader). Each run will compute the color of the pixel and store it in a texture. The RenderPass uses display.wgsl which is a simple shader that is used to compute the UVs for sampling from the texture written to by the ComputePass. The screen buffers are then swapped and the new frame is displayed.

The Renderer *does not* have any notion of a render loop. Updates can be handled in whatever way you want so long as you can provide the Renderer with a DynamicRenderContext. This made it significantly easier to enable runtime update()s from Python.

**There's a third render pass for Multisample Anti Aliasing and a fourth render pass to take a screenshot.*

3.3. Lighting

Objects are lit in a physically based way. Rays are shot into the scene, objects are hit, and the rays are reflected based on the material properties on the impacted surface. Diffuse surfaces will randomly scatter the rays, while metallic surfaces will perfectly reflect the rays. Materials can also be partially diffuse and partially metallic, in practice meaning the reflected ray is a weighted average of a perfect reflection and a random reflection. There is no ambient lighting.

Initially, Eva used Blinn-Phong lighting. Then PBR as described in Learn OpenGL PBR. The lighting was then changed to how it is now. The motivation for this was that I didn't want the real-time renders to look like something you could easily achieve with a raster engine (i.e. clever approximations of real lighting). I wanted it to be apparent that the images were ray traced.

3.4. Scripting Bindings

The scripting bindings live in /eva-py. Eva uses pyo3 to generate bindings and maturin to create the Python3 package eva_py. The raw pyo3 bindings are not very easy to work with directly due to Rust's ownership rules (e.g. once I "give" an object to Rust it can no longer be updated from Python), and type restrictions (e.g. cannot build a robust class hierarchy). For those reasons, a pure Python layer - /eva-py/python/eva_py - was added to make the scripting API easier to work with.

The eva_py package can be built as follows:

```
cd eva-py
# create a virtual environment
python3 -m venv .env
source .env/bin/activate
```

```
# install maturin
pip install maturin
# start maturin
maturin develop
python3 "script-that-uses-eva.py"
```

3.5. Scripting API

Non-comprehensive overview of the scripting API.

```
# import the module
from eva_py import *

# create a skybox (from /assets/skybox)
Eva.add_skybox(["x", "-x", "y", "-y", "z", "-z"])

# load a texture (from /assets/textures)
wood_handle = Eva.add_texture("wood.png")

rock_material = Material(
  0.9, # roughness
  0.3, # metalicness
  (1, 0, 0), # rgb colour
  # texture=wood_handle (optional) texture handle
  # light=(1, 1, 1) (optional) light emissiveness
)
rock_handle = Eva.add_material(rock_material)

# create a mesh (from /assets/meshes)
suzanne = Mesh("suzanne.obj").translate(1, 0, 0) \
                             .set_material(rock_handle)
```

For a more comprehensive look at how it can be used, check out the `/scripts/flappy-bird` for a dynamic example and `/scripts/nonhier` for a static example.

4. Games and Images

Screenshots can be found in `/assets`. Eva comes with two games, “Flappy Bird” and “Pong”.

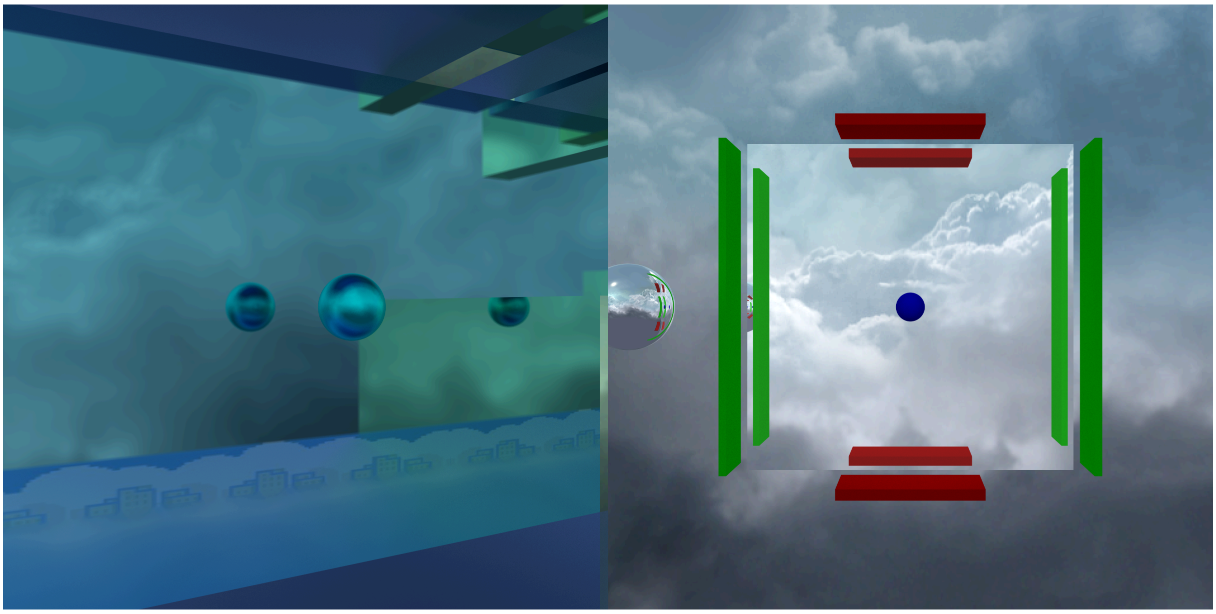

Figure 6: Real-time Flappy Bird and Pong

Flappy Bird
- `[Space]` Jump

```
./debug.sh flappy-bird/flappy-bird
```

Pong
- `[A]` Move bottom paddle left
- `[D]` Move bottom paddle right
- `[J]` Move top paddle left
- `[L]` Move top paddle right

```
./debug.sh pong
```

5. Notes

5.1. Objectives

Objectives I did not complete:

3. Photon mapping (emit and trace photons).
4. Photon mapping (estimate irradiance using stored light information).
7. Port the application to the web.
8. Modelling of the game objects.

(If you think my project warrants bonus marks for MSAA, Materials, the Python API, Games, or anything else it would make my heart sing)

5.2. Screenshots

From my experience, WebGPU will occasionally stop rendering before a frame is finished if it's been taking a long time, resulting in images where part of the image is "empty". It's not supposed to happen - `Device::poll(MaintainBase::Wait)` is supposed to *wait* for the GPU to finish running - but it doesn't always. This is particularly relevant when you're rendering a complicated scene in the static rendering mode. I've found that if you do things on your machine while Eva is rendering, this is significantly more likely to happen.

5.3. Realtime Rendering

The dynamic scenes, `pong` and `flappy-bird`, run at 60FPS (real-time) on my M1 Max Macbook Pro. They do not run smoothly on the school Linux machines. Rendering speeds can be improved by reducing the number of samples per pixel and the maximum number of reflections per ray. Another solution is to reduce the resolution (in `eva-py/src/eva_main.rs`) down from `850x850`.

5.4. Static Rendering

All the images showcased in this document took less than 3 minutes to render, most taking less than 30 seconds. The number of samples per pixel varied from 300 to 1500. They take significantly longer to render on the school Linux machines.

5.5. Compile Times

Compiling Eva on the school Linux machines takes *several minutes*. If it's taking a long time, it's *just taking a long time*. Reducing compile times drastically reduced performance and, therefore, was not something I wanted to do.

6. Post Mortem

The biggest trump card for this project is that I didn't know what I wanted to do until I started building Eva. I looked at my output images and considered what would make them better and I let that, more than my core objectives, drive my development process. Perhaps a little more research at the proposal phase could have helped to avoid some of this, but I couldn't *see* what my project would look like at that point so it was hard to look forward and know what I'd want to add.

The second biggest trump card was that virtually every feature was harder to add and debug because the routine that determines the color of each pixel runs in a compute shader on the GPU. All data structures need to be passed into the GPU, textures need to be encoded in a GPU-compatible format, and there is no such thing as a print statement.

Two things were harder than I thought they would have been:

6.1. Porting Eva to the Web

Yes, `wgpu` compiles to WASM but the work extends far beyond that.

- i) Getting the Python scripting to work on the web was very tricky and something I ultimately sided against doing. To make real-time updates work, my renderer requires exclusive access to the Python3 Global Interpreter Lock (GIL) to run the Python code and fetch the updated values. On the web, this is hard because WASM is single-threaded. So, although I could have gotten the scripts to compile to WASM, I couldn't have run them in my browser without significantly modifying how I handle the updates and adding a lot of `#[cfg(target = "wasm")]` annotations for conditional rendering.
- ii) Assets. Loading assets on the web requires making requests. This is fine, but it requires an asynchronous runtime, like NodeJS, to poll the `Futures` (promises in JavaScript) to see if the asset is ready. Threads cannot block. Eva and Eva-py fetch texture assets, mesh assets, and create screenshots. To make these asynchronous, I needed to either move all asset loading into Rust and add an async runtime for Rust that was WASM compatible, or add an async runtime for both Rust and Python. And for Python runtimes, I needed it to be compatible with `pyodide` so it could run in the browser.

For those reasons, I decided to not port the application to the web.

6.2. GPU Compatibility

Not all GPUs have the same features. The WebGPU `Adapter` is used to ask for a `Device` and `Queue` supporting a certain set of features, if they're available. The WebGPU `Instance`, a wrapper on your native GPU, allows you to create a `Surface` (a fancy texture) given a `winit::Window` and can tell you what the capabilities of that `Surface` are. Because I work at home on my local machine, I found out rather late into this assignment that the features and capabilities of the `Device` and `Surface` of my local machine (an M1 Max Macbook Pro) are different than what is available on the school Linux machines. My project would not run. Fixing this required changing texture formats, storage types for texture, and some other shader-specific logic. This was a non-trivial diff I was not expecting.

6.3. Dependencies

- `wgpu`: Rust implementation of the WebGPU specification.
- `pyo3`: Rust to Python3 bindings.
- `maturin`: Python3 package builder for pyo3 generated bindings.
- `nalgebra`: Rust linear algebra.
- `winit`: Rust cross-platform windowing.
- `pollster`: Rust crate for statically resolving `Futures`.
- `bytemuck`: Rust crate for converting data types into bytes.
- `image`: Rust crate for encoding and decoding images.
- `encase`: Rust crate for byte-aligning structures for use in WebGPU shaders.
- `obj`: Rust crate for loading `.obj` meshes.
- `half`: Rust crate for handling 16-bit floating float numbers.