3D graphics in Rust
The missing introduction

John Nagle

Animats

Version of January, 2024

Introduction

3D graphics in Rust isn’t all that complicated. But the documentation is nonexistent. Here’s an introduction to getting started. This assumes a knowledge of 3D graphics at the OpenGL level or similar. Things are far enough along that 3D graphics programs can and have been built. It will be a bumpy ride, because this is not a mature ecosystem.

The main alternatives that don’t involve directly using C/C++ packages are Bevy, the game engine, and calling WGPU directly. Bevy isn’t bad, but you have to do things Bevy’s way. If you’re not writing a game, it’s a bad fit. Calling WGPU directly means creating something to do the job Rend3 does.

This is a honest statement of what it looks like from the perspective of someone who has used this stack extensively.

The graphics stack

Here’s the stack of crates needed to do 3D:

- The Rend3 framework.
- Egui – 2D crate for dialogs, menus, etc.
- Rend3 – 3D graphics with a high level API
- WGPU – Cross-platform 3D graphics, low-level API.
- Winit – Cross platform window and event handling
- Vulkan/Metal//Direct-X/OpenGLAndroid – Talks to various GPUs and graphics devices

Application developers need to be familiar with the APIs of Egui, Rend3, and Winit. WPGU and the Vulkan, etc. layer are used internally but not called directly.

CITY OF NEW BABBAGE
CLOCKHAVEN MILITIA CENTRE

A typical scene

Rend3

Rend3 is a retained mode graphics library. The application creates graphics objects. Rend3 puts them into the GPU and displays them. The application doesn't draw objects itself. Rend3 and WGPU handle all the details. It's physically based rendering, with shadows, but without reflections. The model follows the glTF spec.

Rend3's main structs are straightforward.

- **Meshes of triangles**. These have vertices, normals, UV values, and triangles, like everybody else's meshes.
- **2D textures**. These are images, and are in a format used by the Rust "image" crate.
- **Materials**. These pull together various textures – albedo, roughness, metalness, normals, and emission, which behave the way the glTF spec says they should.
- **Transforms**. Standard 4X4 affine matrices which transform an object to world space.
- **Skeletons**. Rigged meshes are supported. Each vertex can have up to four weights tying it to a skeleton bone.
- **Objects**. An object is a material plus a mesh and a transform. Creating an object puts it on screen.
- **Lights.** About what you'd expect.
- **Camera**. About what you'd expect.

The Rend3 creation functions for each of these things return an opaque handle. Handles are reference-counted Rust Arc structs. When all the users of a handle have dropped it, its struct goes away. This releases any lower-level resources involved, down to the GPU level. Dropping an Object makes it disappear from the scene. Meshes, 2D textures, and materials can be shared by Objects.

Creating meshes and textures is relatively expensive. The other operations are all cheap.

Any thread can create and release these structs. For high performance, the render thread should mostly just render, with other threads updating content.

All this is straightforward to use. There's even a loader for glTF content. See the "scene-viewer" example.

This is a safe API. You don't need "unsafe" in your code. If you have no unsafe code, and the stack crashes, it's not your fault. File a bug report.

The Rend3 framework

Most of these crates are libraries – the application calls them – but atop it all sits the Rend3 Framework. This owns the event loop. Using it requires a few hundred lines of boilerplate code. Each new version of Rend3 seems to involve major changes to the framework, requiring a rewrite of that part of the application. None of this is well-documented. So do as little in the actual framework function code as possible. Put your own code in your own functions and call them from the framework.

To understand the framework, read the Rend3 examples "cube", "egui", and "scene-viewer". Don't try to understand them, just follow what they do.

The framework has three main functions.

setup

Called once at startup.The boilerplate for starting things up. Just go look at the Rend3 examples and copy.

handle_event

Called on each window event other than redraw. Handles all window events other than redrawing. Copy and paste as required.

handle_redraw

Called over and over to redraw. Application must do a lot of boilerplate here. Again, copy and paste as required.

egui

Egui is a 2D graphics library for menus, dialogs, and such. Unlike Rend3, it's not retained mode – everything is drawn on every frame. The claim is that it only needs about 1% of the frame time. This is true if you don't overdo it.

Egui is straightforward, but it takes a lot of code to do a dialog box or menu. There’s been talk of some kind of generator to crank out dialog boxes and such, but so far, it’s custom Rust code for each GUI item.

Cross-platform

Amazingly, all this works across multiple platforms. Windows, Linux, and MacOS mostly work. You can even cross-compile for other platforms and have it work. Linux → Windows works. Linux → MacOS has linker problems because it needs some proprietary Apple stuff. You can still recompile the identical code on a Mac. The program ui-mock is set up to work on all three desktop platforms.

Targeting Android and WebGPU works, too, but isn’t as portable. Web land doesn’t have full Rust threading. Android land has some wierdness. Despite this, the Rend3 examples work cross platform. It’s only when you go for high performance with multi-threading that you hit the limits.

Trouble spots

Version hell

All of this stuff advances in version lockstep. Each of these crates needs some specific version of the others.

Rend3, being unfinished, doesn’t have versions, just Github rev numbers. You pick some rev from the trunk and refer to it directly in Cargo.toml. There is a version of Rend3 on crates.io, but it’s two years old and way out of date. Read the Rend3 discord to keep up with changes.

Running out of GPU resources

You can fill up the GPU. There’s no way to know if you’re out of resources until you get an error return when creating some Rend3 struct. This isn’t a fatal error. Applications have to be able to detect overflow and then take steps to reduce their resource consumption, probably by drawing less stuff. Design for this.

Panics

Rend3 still crashes occasionally. The application should catch panics and display or log a backtrace. True crashes, as “segmentation faults”, are rare. Usually you get a clean Rust backtrace. Submit issues on Github.

Major missing features

- Selection of objects via mouse/touch
- Environmental shaders
- Efficient directional lights

You can write and use your own shaders, but how to do that is beyond this note.

Resources

- Rust game development on Reddit: https://www.reddit.com/r/rust_gamedev/
- Rend3 Github: https://github.com/BVE-Reborn/rend3
- Rend3 Discord: https://discord.com/invite/mjxXTVzaDg
- WGPU main site: https://wgpu.rs/
- ui-mock, a cross-platform dummy game: https://github.com/John-Nagle/ui-mock

Conclusion

This is a a reasonably good graphics stack. It’s far less of a headache than writing directly to Vulkan or WGPU, because you get automatic resource management and Rust safety. It’s not a finished graphics stack, but it’s getting close. It needs more users to push it to a mature state.