

---

## Granite

Granite is my personal Vulkan renderer project.

### Why release this?

The most interesting part of this project compared to the other open-source Vulkan renderers so far is probably the render graph implementation.

The project is on GitHub in the hope it might be useful as-is for learning purposes or generating implementation ideas.

### Disclaimer

**Do not expect any support or help. Pull requests will likely be ignored or dismissed.**

### License

The code is licensed under MIT. Feel free to use it for whatever purpose.

### High-level documentation

See [OVERVIEW.md](#).

### Low-level rendering backend

The rendering backend focuses entirely on Vulkan, so it reuses Vulkan enums and data structures where appropriate. However, the API greatly simplifies the more painful points of writing straight Vulkan. It's not designed to be the fastest renderer ever made, it's likely a happy middle ground between "perfect" Vulkan and OpenGL/D3D11 w.r.t. CPU overhead.

- Memory manager
- Deferred destruction and release of API objects and memory
- Automatic descriptor set management
- Linear allocators for vertex/index/uniform/staging data
- Automatic pipeline creation
- Command buffer tracks state similar to older APIs

- 
- Uses TRANSFER-queue on desktop to upload linear-allocated vertex/index/uniform data in bulk
  - Vulkan GLSL for shaders, shaders are compiled in runtime with shaderc
  - Pipeline cache save-to-disk and reload
  - Warm up internal hashmaps with Fossilize
  - Easier-to-use fences and semaphores

Missing bits: - Multithreaded rendering - Precompile all shaders to optimized SPIR-V

Implementation is found in [vulkan/](#).

## High-level rendering backend

A basic scene graph, component system and other higher-level scaffolding lives in [renderer/](#). This is probably the most unoptimized and naive part.

## PBR renderer

Pretty barebones, half-assed PBR renderer. Very simplified IBL support. Fancy rendering is not the real motivation behind this project.

## Post-AA

Fairly straight forward FXAA, SMAA and TAA (no true velocity buffer though).

## Automatic shader recompile and texture reload (Linux/Android only)

Immediately when shaders are modified or textures are changed, the resources are automatically reloaded. The implementation uses inotify to do this, so it's exclusive to Linux unless a backend is implemented on Windows (no).

## Network VFS

For Linux host and Android device, assets and shaders can be pulled over TCP (via ADB port-forwarding) with [network/netfs\\_server.cpp](#). Quite convenient.

## Validation

In debug build, LunarG validation layers are enabled. Granite is squeaky clean.

---

## Render graph

[renderer/render\\_graph.hpp](#) and [renderer/render\\_graph.cpp](#) contains a fairly complete render graph. It supports:

- Automatic layout transitions
- Automatic loadOp/storeOp usage
- Automatic scaled loadOp for simple lower-res game -> high-res UI rendering scenarios
- Uses async compute queues automatically
- Optimal barrier placement, signals as early as possible, waits as late as possible VkEvent is used for in-queue resources, VkSemaphore for cross-queue resources
- Basic render target aliasing
- Can merge two or more passes into multiple subpasses for efficient rendering on tile-based architectures
- Automatic mip-mapping if requested
- Uses transient attachments automatically to save memory on tile-based architectures
- Render target history, read previous frame's results in next frame for feedback
- Conditional render passes, can preserve render passes if necessary
- Render passes are reordered for optimal (?) overlap in execution
- Automatic, optimal multisampled resolve with pResolveAttachments

I have written up a longer blog post about its implementation [here](#).

The default application scene renderer in [application/application.cpp](#) sets up a render graph which does: - Conditionally renders a shadow map covering entire scene - Renders a close shadow map - Automatically pulls in reflection/refraction render passes if present in the scene graph - Renders scene G-Buffer with deferred - Lighting pass (merged with G-Buffer pass into a single render pass) - Bloom threshold pass - Bloom pyramid downsampling - Async compute is kicked off to get average luminance of scene, adjusts exposure - Two upsampling steps to complete blurring in parallel with async - Tonemap (HDR + Bloom) rendered to backbuffer (sRGB) - (Potentially UI can be rendered on top with merged subpasses)

## Scene format

glTF 2.0 with PBR materials is mostly supported. A custom JSON format is also added in order to plug multiple glTF files together for rapid prototyping of test scenes.

---

## Texture formats

- PNG, JPG, TGA, HDR (via stb)
- GTX (Granite Texture Format, custom texture format for compressed formats)

ASTC, ETC2 and BCn/DXTn compressed formats are supported.

## glTF-repacker

There's a tool to repack glTF models. Textures can be compressed to ASTC or BC using ISPC Texture Compressor. zeux's meshoptimizer library can also optimize meshes. The glTF emitted uses some Granite specific extras to be more optimal, so it's mostly for internal use.

## Compilers

Tested on GCC, Clang, and MSVC 2017.

## Platforms

- SDL3 (Linux / Windows)
- [VK\\_KHR\\_display](#) (headless Linux w/ basic keyboard, mouse, gamepad support)
- libretro Vulkan HW interface
- Headless (benchmarking)
- Custom surface plugin
- Android

## Vulkan implementations tested

- AMD Linux (Mesa, AMDVLK)
- Intel Linux (Mesa)
- AMD Windows
- nVidia Linux
- Arm Mali (Galaxy S7/S8/S9)
- Pixel C tablet (Tegra X1)

---

## Build

Plain CMake. Remember to check out submodules with `git submodule update --init`.

```
1 mkdir build
2 cd build
3 cmake .. -DCMAKE_BUILD_TYPE=Release -G Ninja
4 ninja -j16 # YMMV :3
```

For MSVC, it should work to use the appropriate `-G` flag. There aren't any real samples yet, so not much to do unless you use Granite as a submodule.

`viewer/gltf-viewer` is a basic glTF viewer used as my sandbox for more complex testing. Try some models from `glTF-Sample-Models`.

## Android

Something ala:

```
1 cd viewer
2 gradle build
```

Assets used in the default `gltf-viewer` target are pulled from `viewer/assets`.

## Third party software

These are pulled in as submodules.

- SDL3
- glslang
- rapidjson
- shaderc
- SPIRV-Cross
- SPIRV-Headers
- SPIRV-Tools
- stb
- volk
- meshoptimizer
- Fossilize
- muFFT
- MikkTSpace (inlined into `third_party/mikktspace`)