Linux Graphics 101
Converting bits to Triangles
Disclaimer

- I am not (yet) an experienced Graphics developer
  - Take my words with a grain of salt
  - Please correct me if I’m wrong
What is this talk about?

- This presentation is about
  - Providing a overview of the Linux Open Source Graphics stack

- This presentation is not about
  - Teaching you how to develop a GPU driver
  - Teaching you how to use Graphics APIs (OpenGL/Vulkan/D3D)
  - Explaining what GPUs are and how they work
The Linux Graphics Stack
The Big Picture

Application

Graphics API

Userspace Drivers

OpenGL GLX EGL Direct3D Vulkan WSI

Mesa3D

DRM

freedreno panfrost etnaviv i915 ...

Kernel

Open First
The Graphics API
The Graphics API: What are they?

- Entry points for Graphics Apps/Libs
- Abstract the GPU pipeline configuration/manipulation
- You might have the choice
  - OpenGL/OpenGLES: Well established, well supported and widely used
  - Vulkan: Modern API, this is the future, but not everyone uses/supports it yet
  - Direct3D: Windows Graphics API (version 12 of the API resembles the Vulkan API)
The Graphics API: Shaders

- Part of the pipeline is programmable
  - Separate Programming Language: GLSL or HLSL
  - Programs are passed as part of the pipeline configuration...
  - ... and compiled by drivers to generate hardware-specific bytecode
The Graphics API: OpenGL(ES) vs Vulkan

- Two philosophies:
  - OpenGL tries to hide as much as possible the GPU internals
  - Vulkan provides fine grained control
  - Vulkan provides a way to record operations and replay them
  - More work for the developer, less work for the CPU

- Vulkan applications are more verbose, but
  - Vulkan verbosity can be leveraged by higher-level APIs
  - Drivers are simpler
  - Improved perfs on CPU-bound workloads
The Kernel/Userspace Driver Separation

- GPUs are complex beasts → drivers are complex too:
  - We don’t want to put all the complexity kernel side
  - Not all code needs to run in a privileged context
  - Debugging in userspace is much easier
  - Licensing issues (for closed source drivers)
Kernel Drivers
Kernel Drivers

- Kernel drivers deal with
  - Memory
  - Command Stream submission/scheduling
  - Interrupts and Signaling
- Kernel drivers interfaces with open-source userspace drivers live in Linus’ tree: drivers/gpu/drm/
- Kernel drivers interfacing with closed-source userspace drivers are out-of-tree
Kernel Drivers: Memory Management

- Two Frameworks
  - GEM: Graphics Execution Manager
  - TTM: Translation Table Manager

- Some Terminologies
  - Buffer Object - A region of memory to upload GPU Data (Textures, Vertexes, etc)
  - ioctl - the most common way for applications to interface with device drivers.
  - cmdstream - A set of commands compromising a full job on the GPU.
Kernel Drivers: Memory Management

- GPU drivers using GEM
  - Should provide an `ioctl()` to allocate Buffer Objects (BOs)
  - Releasing BOs is done through a generic `ioctl()`
  - Might provide a way to do cache maintenance operations on a BO
  - Should guarantee that BOs referenced by a submitted Command Stream are properly mapped GPU-side

```c
#define DRM_PANFROST_WAIT_BO              0x01
#define DRM_PANFROST_CREATE_BO            0x02
#define DRM_PANFROST_MMAP_BO              0x03
#define DRM_PANFROST_GET_PARAM            0x04
#define DRM_PANFROST_GET_BO_OFFSET        0x05
```
Kernel Drivers: Scheduling

- Submission != Immediate execution
  - Several processes might be using the GPU in parallel
  - The GPU might already be busy when the request comes in
- Submission == Queue the cmdstream
- Each driver has its own ioctl() for that
- Userspace driver knows inter-cmdstream dependencies
- Scheduler needs to know about those constraints too
- DRM provides a generic scheduling framework: drm_sched
Userspace/Kernel Driver Synchronization

- Userspace driver needs to know when the GPU is done executing a cmdstream
- Hardware reports that through an interrupt
- Information has to be propagated to userspace
- Here come fences: objects allowing one to wait on job completion
- Exposed as syncobjs objects to userspace
- Fences can also be placed on BOs
Userspace Drivers
Userspace Driver: Roles

- Exposing one or several Graphics API
  - Maintaining the API specific state machine (if any)
  - Managing off-screen rendering contexts (if any)
  - Compiling shaders into hardware specific bytecode
  - Creating, populating and submitting command streams

- Interacting with the Windowing System
  - Managing on-screen rendering contexts
  - Binding/unbinding render buffers
  - Synchronizing on render operations
Mesa: Open Source Userspace Drivers

- 2 Graphics APIs 2 different approaches:
  - GL:
    - Mesa provides a frontend for GL APIs (libGL(ES))
    - GL Drivers implement the DRI driver interface
    - Modern drivers make use of the Gallium state tracker within mesa
    - Drivers are shared libs loaded on demand
  - Vulkan:
    - Khronos has its own driver loader (Open Source)
    - Mesa just provides Vulkan drivers
    - No abstraction for Vulkan drivers, code sharing through libs
Mesa State Tracking

(Pipeline Configuration)
Mesa State Tracking: Pre-Gallium
Mesa State Tracking: Gallium

Application

Graphics API

Driver Interface

Kernel

OpenGL
GLX
EGL
Direct3D

GL Dispatcher
DRI

dd_function_table/DRI
State Trackers

Gallium3D

panfrost
etnaviv
...
freedreno

Mesa3D

panfrost
etnaviv
...
msm

DRM
Mesa State Tracking: Vulkan

Application

Graphics API

Kernel

OpenGL, GLX, EGL, Direct3D, Vulkan, WSI

GL Dispatcher, DRI

dd_function_table / dri, nine

State Trackers

Gallium3D

Driver Interface

panfrost, etnaviv, freedreno

Mesa3D

panfrost, etnaviv, ..., msm, i915

DRM
Mesa Shader Compilation

(Pipeline Manipulation)
Mesa: Shader Compilation

- Compilation is a crucial aspect
- Compilation usually follows the following steps
  - Shader Programming Language -> Generic Intermediate Representation (IR)
  - Optimization in the generic IR space
  - Generic IR -> GPU specific IR
  - Optimization in the GPU specific IR space
  - Byte code generation
- Note that you can have several layers of generic IR
Mesa: Shader Compilation

- Shader Programming Languages
  - GLSL
  - HLSL
- Intermediate Representations
  - GLSL IR
  - SPIR-V
  - TGSi
  - NIR
- Driver Compilers
  - NV50 IR (Nouveau IR)
  - MIR (Midgard IR)
  - IR3 (Adreno IR)
Debugging Tips
Tips and Tricks

- GDB is your friend, get comfortable with it
  - _mesa_error() to trap Mesa errors
  - _mesa_foo entry points for glFoo functions
  - Turn on asserts with -Db_ndebug=false

- Set MESA_DEBUG for error messages to stdout

- Every driver has its own debugging variables
  - Check https://docs.mesa3d.org/envvars.html for complete list

- Piglit
  - https://gitlab.freedesktop.org/mesa/piglit/
  - Comprehensive way of understanding a particular feature or gl call.
Conclusion
Nice overview, but what’s next?

- The GPU topic is quite vast
- Start small
  - Choose a driver
  - Find a feature that’s missing or buggy
  - Stick to it until you get it working
- Getting a grasp on GPU concepts/implementation takes time
- Don’t give up
Useful readings

- Understanding how GPUs work is fundamental:
  - [A trip through the Graphics Pipeline 2011](#)
  - [How a GPU Works](#)
  - Search "how GPUs work" on Google ;-)  
- [Khronos OpenGL Wiki](#)
  - [OpenGL Objects](#)
  - [Rendering Pipeline](#)
- Mesa source tree is sometimes hard to follow, refer to the [doc](#)
- And the [DRM kernel docs](#) can be useful too  
  - [Fences](#)
- [Open Source Graphics 101: Getting Started - Boris Brezillon, Collabora](#)
Q & A
Thank you!