

Open Source Graphics 101: Getting Started

Boris Brezillon
ELCE 2019

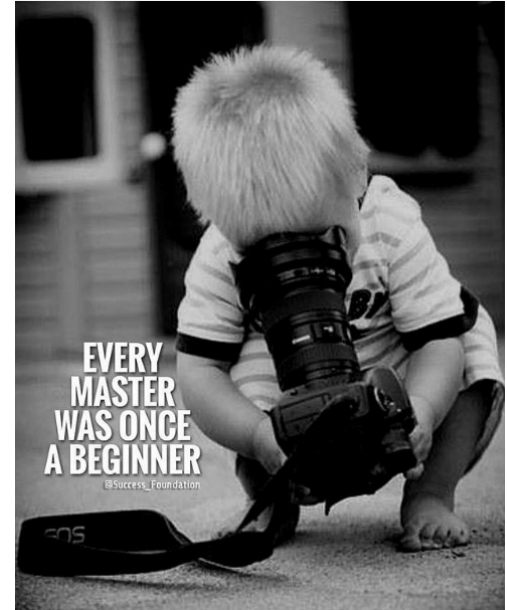


COLLABORA

Open First

Disclaimer

- I am not (yet) an experienced Graphics developer
 - Take my words with a grain of salt
 - Please correct me if I'm wrong



Source: <https://me.me/i/every-master-was-once-a-beginner-success-foundation-well-said-16284942>



COLLABORA

Open First

What is this talk about?

- This presentation is about
 - Explaining what GPUs are and how they work
 - Providing a brief 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)

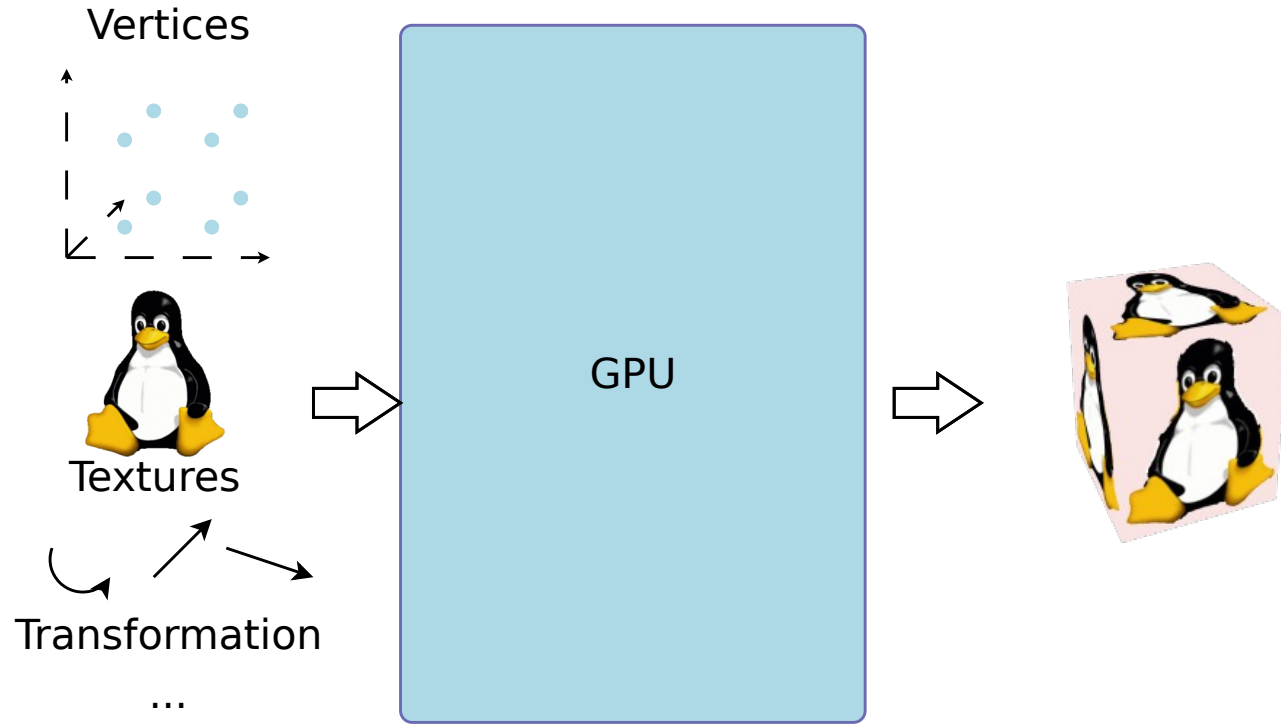




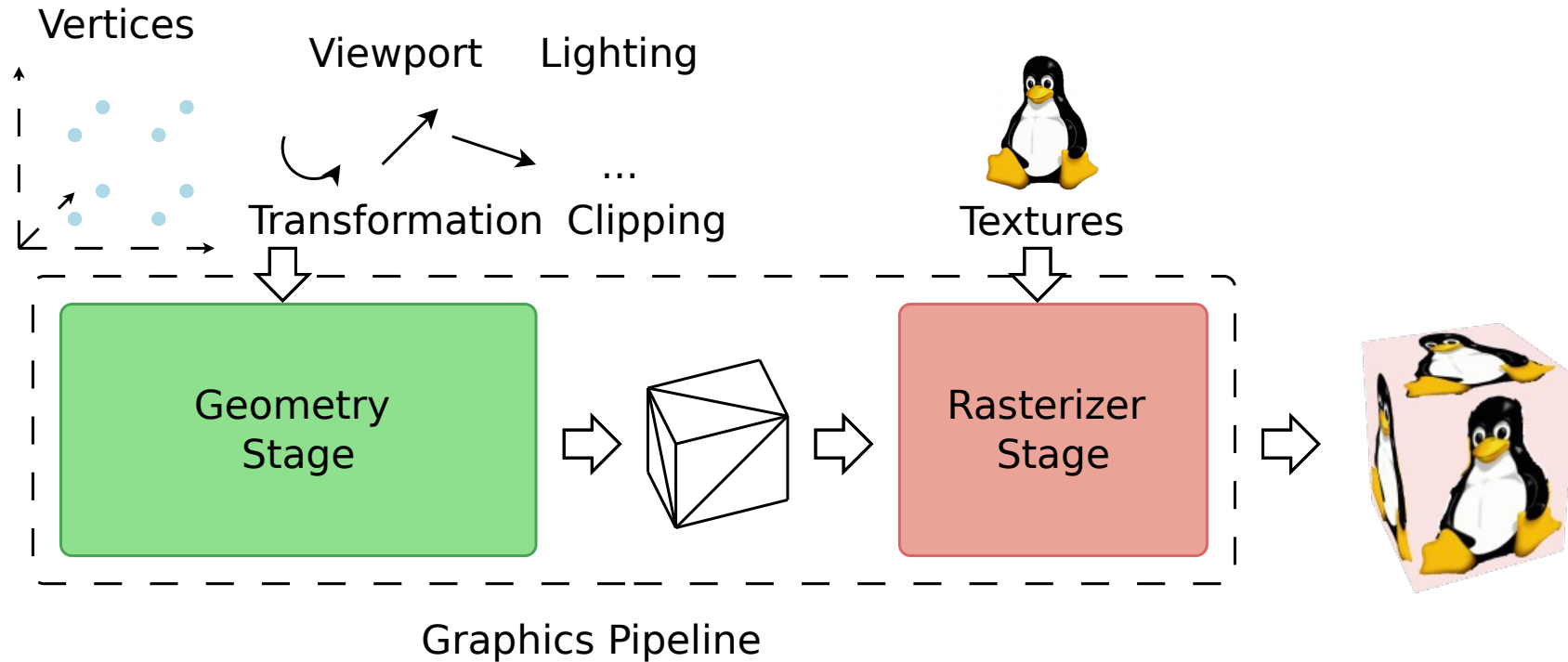
COLLABORA

The Graphics Pipeline

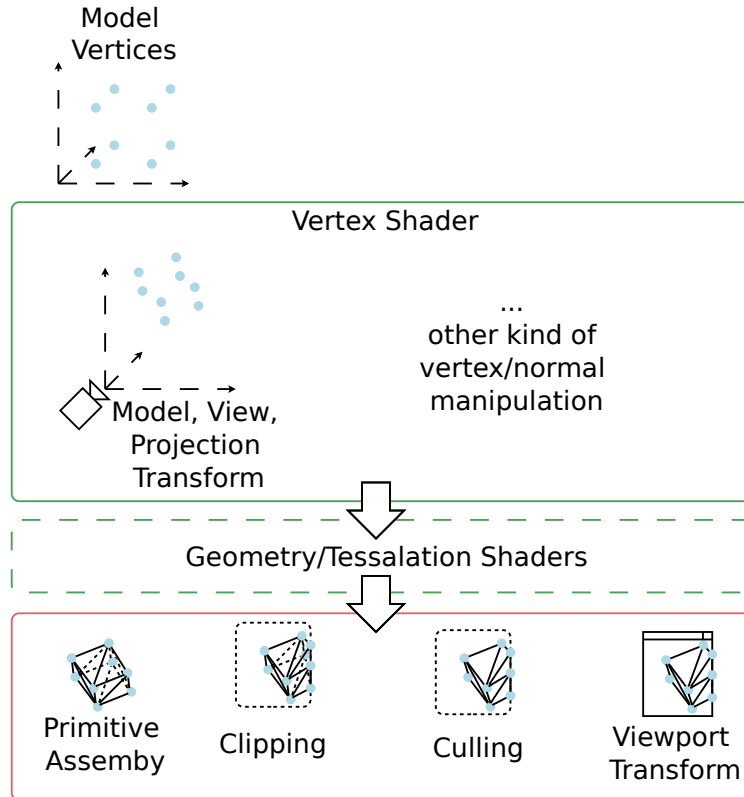
The Graphics Pipeline



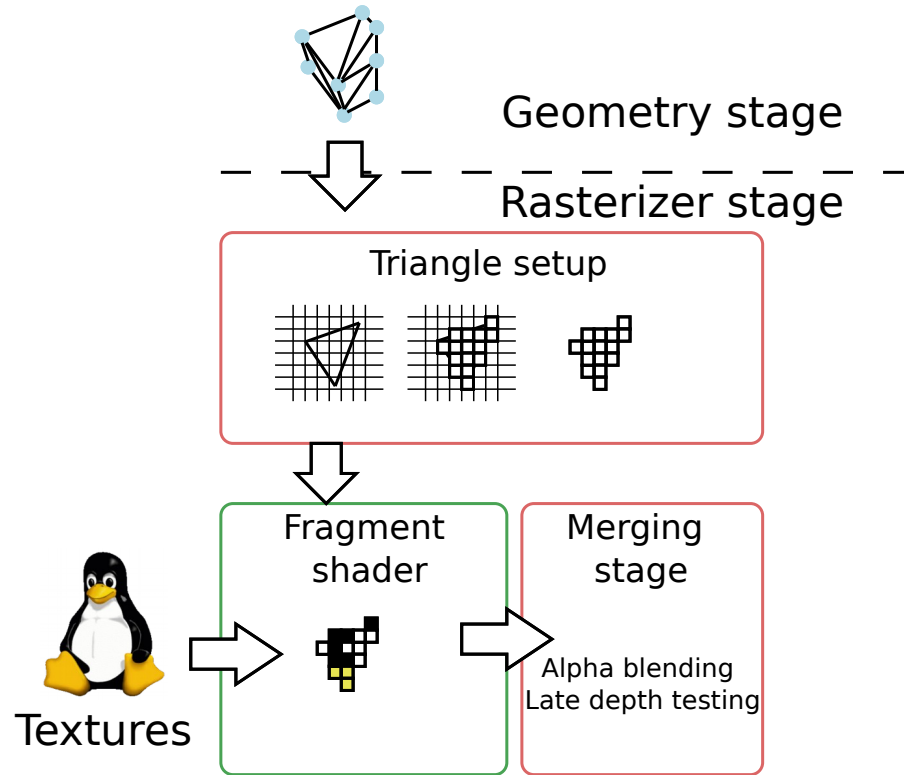
The Graphics Pipeline



The Geometry Stage



The Rasterizer Stage

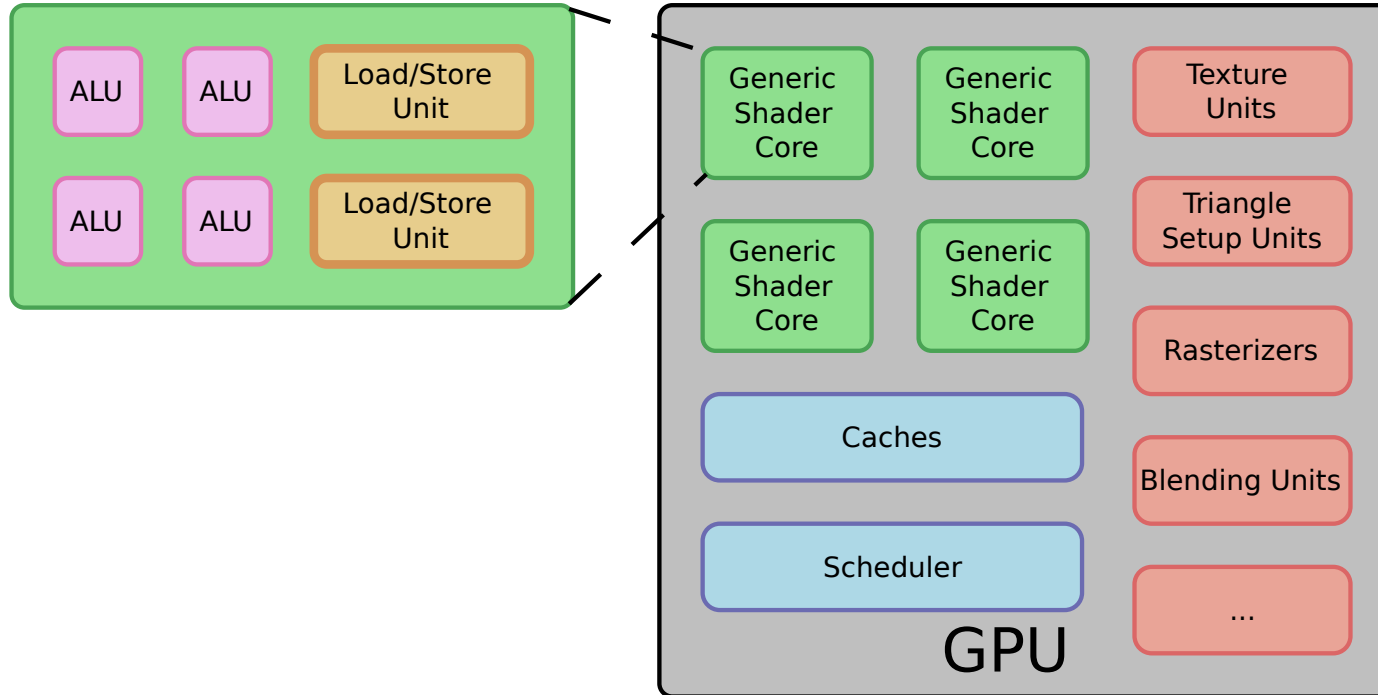




COLLABORA

GPU Internals

GPU Internals



Let's go massively parallel!

- Why?

- Vertices, normals, fragments can be processed independently
- We have a lot of them (complex scene, complex models, high resolution)
- We want real-time rendering

- How?

- SIMD (Single Instruction Multiple Data)
- Shared dedicated units for complex/specialized tasks
- No fancy CPU stuff like out-of-order control logic, smart pre-fetcher, branch predictors, ...



Parallization, how hard can it be?

SIMD + lot of cores: we're done, right?



Parallization, how hard can it be?

Multithreaded programming



Source: <http://devhumor.com/media/multithreaded-programming>



COLLABORA

Open First

Parallization, how hard can it be?

- Stalls caused by memory access
 - Add caches
 - Multi-threading
- SIMD: try to get all ALUs busy
 - Avoid conditional branches
 - Try to pack similar operation together





COLLABORA

Interaction with your GPU

CPU: Hey GPU, listen/talk to me please!

- The CPU is in charge of all apps running on a machine, including graphics apps
- The CPU needs a way to send requests to/get results from the GPU
- Huge amount of data needs to be exchanged (vertices, framebuffers, textures, ...)

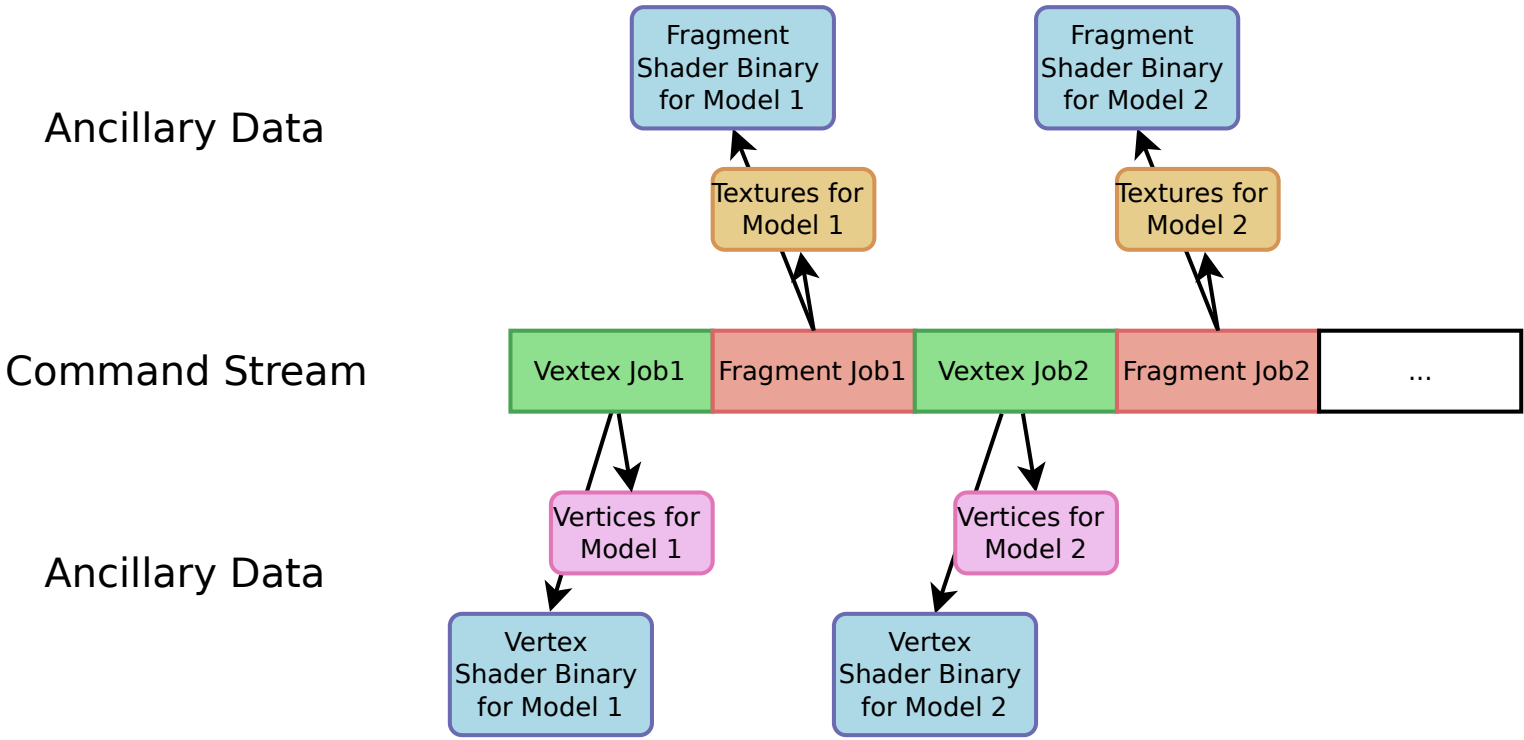


CPU: Hey GPU, listen/talk to me please!

- How?
 - Put everything in memory
 - Set of operations to execute is also stored in memory (frequently called command stream)
 - Once everything is in memory, ask the GPU to execute what we prepared
 - Let the GPU inform us when it's done



GPU Command Stream

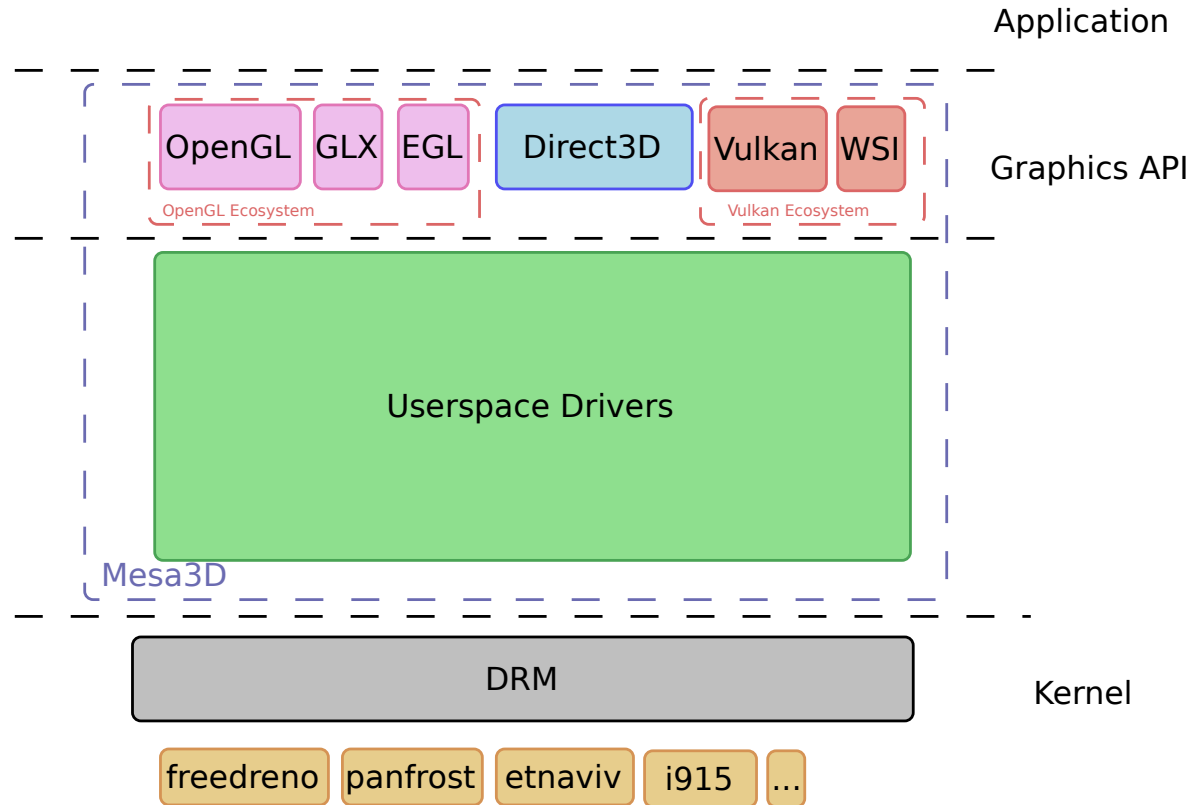




COLLABORA

The Linux Graphics Stack

The Big Picture



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 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

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

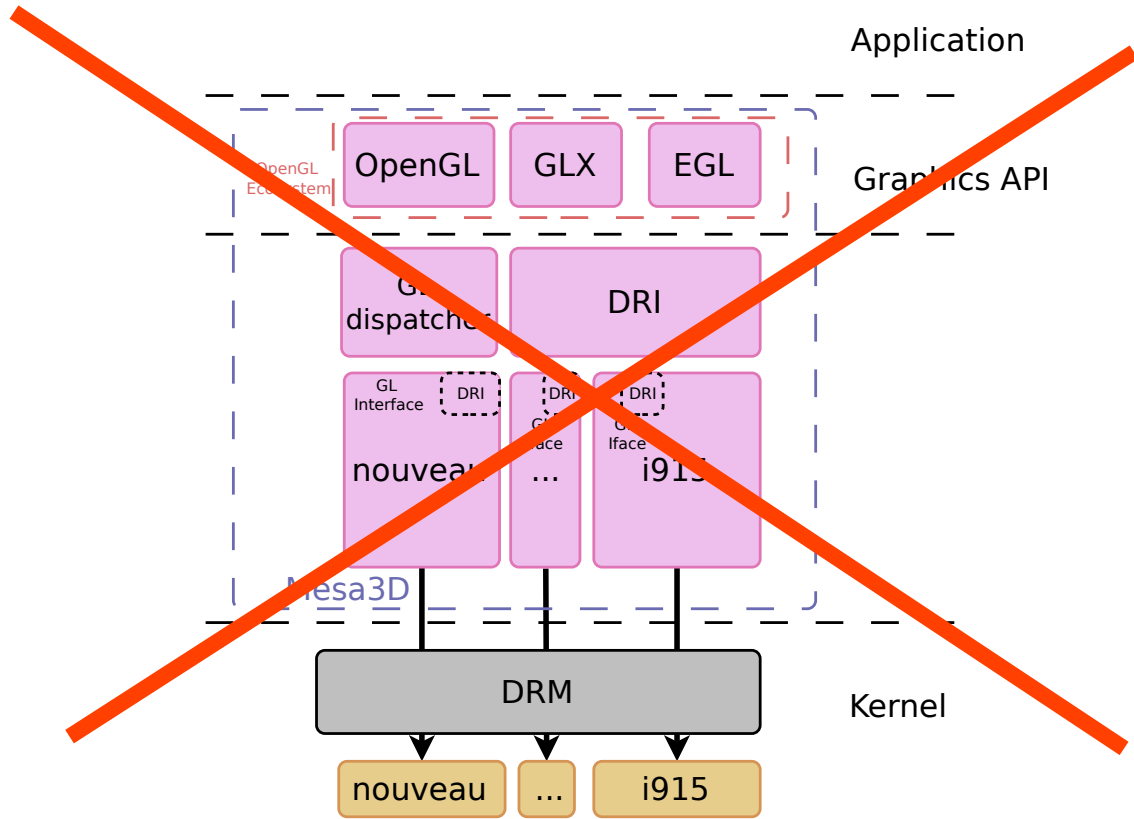


Mesa: Open Source Userspace Drivers

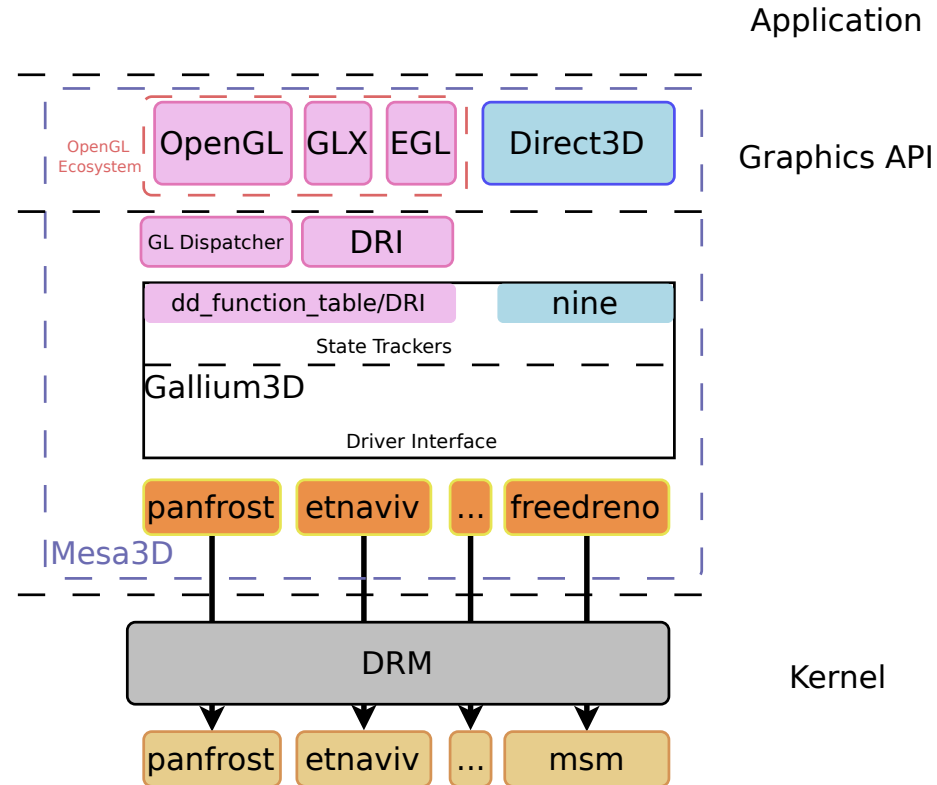
- 2 Graphics APIs 2 different approaches:
- GL:
 - Mesa provides a frontend for GL APIs (libGL(ES))
 - GL Drivers implement the DRI driver interface
 - Drivers are shared libs loaded on demand
- Vulkan:
 - Khronos has its own driver loader (Open Source)
 - Mesa just provides Vulkan drivers
 - No abstraction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



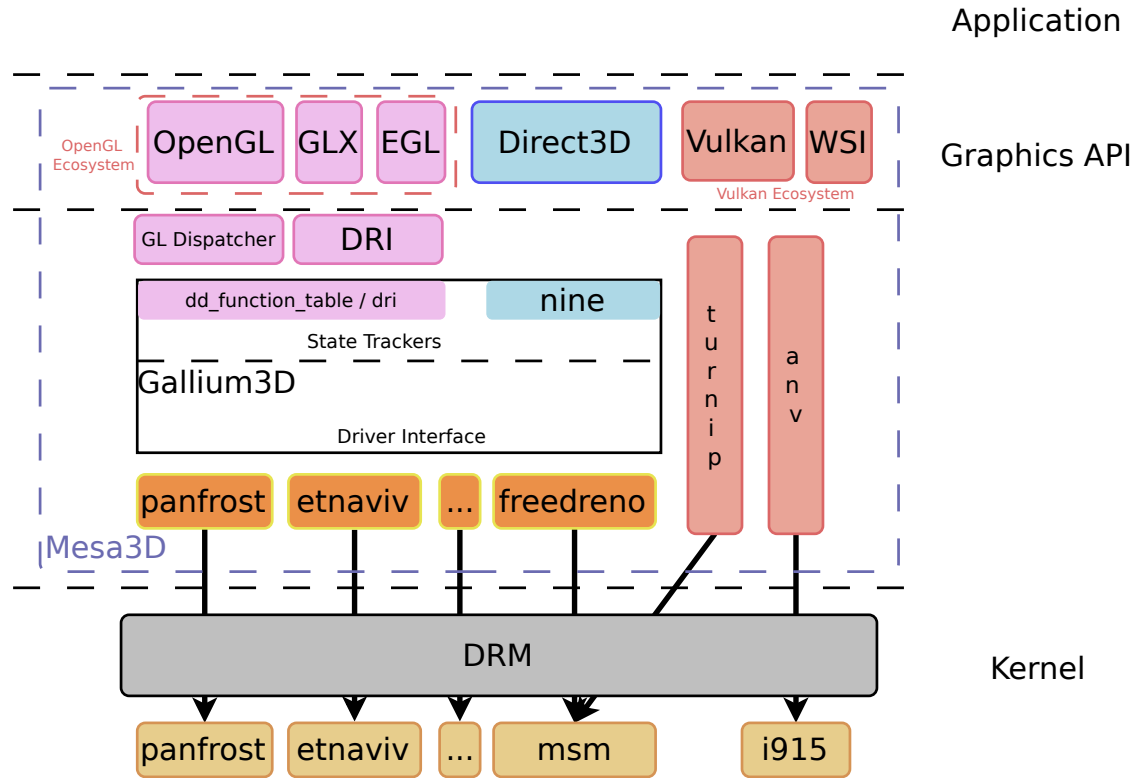
Mesa State Tracking: Pre-Gallium



Mesa State Tracking: Gallium



Mesa State Tracking: Vulkan





COLLABORA

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:
 - <https://fgiesen.wordpress.com/2011/07/09/a-trip-through-the-graphics-pipeline-2011-index/>
 - https://www.cs.cmu.edu/afs/cs/academic/class/15462-f11/www/lec_slides/lec19.pdf
 - Search "how GPUs work" on Google ;-)
- Mesa source tree is sometimes hard to follow, refer to the doc: <https://mesa-docs.readthedocs.io/en/latest/sourcetree.html>
- And the DRM kernel doc can be useful too: <https://01.org/linuxgraphics/gfx-docs/drm/gpu/index.html>





Q & A
Thank you!
(Psst, we're hiring!)

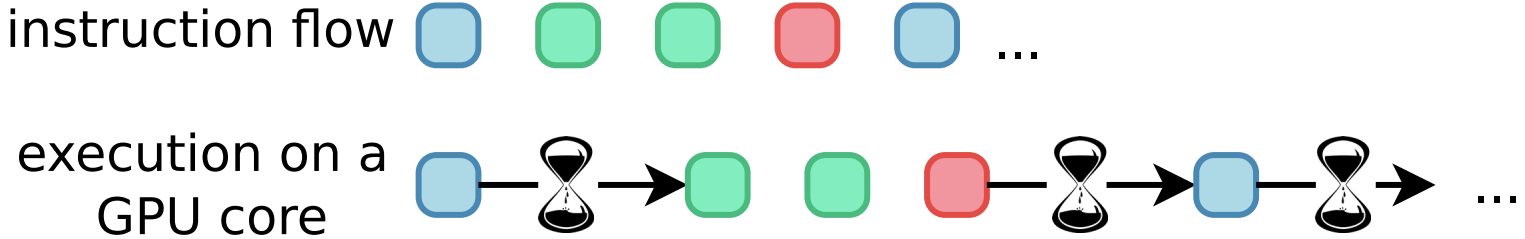







COLLABORA

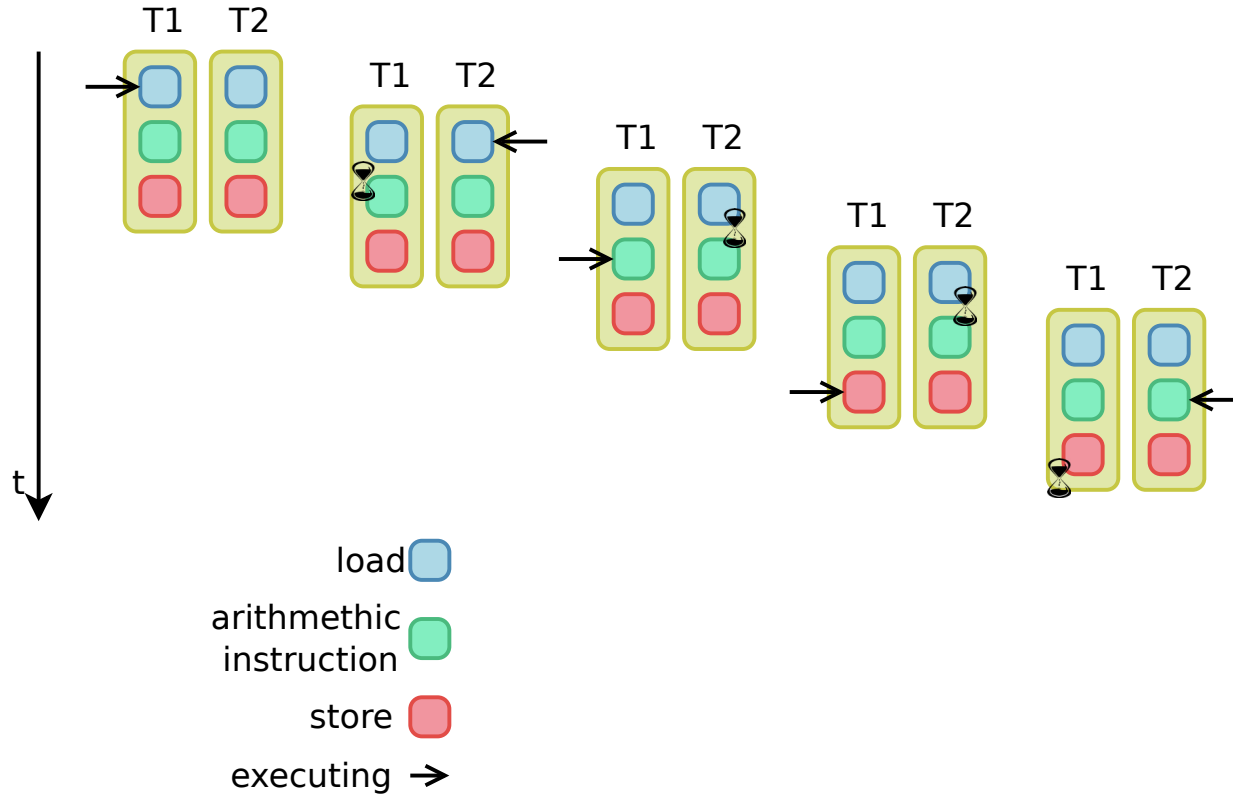
Backup Slides

Stalls on Memory Accesses

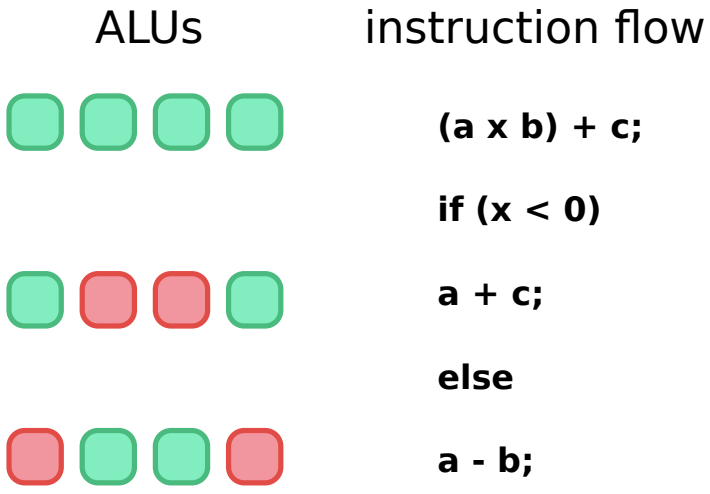
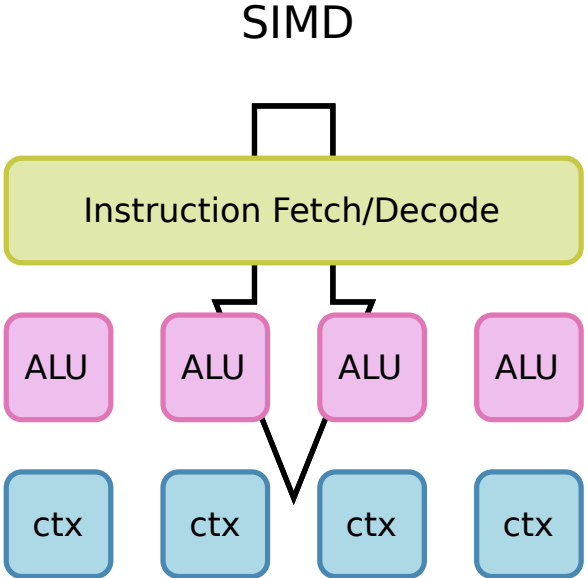


- load 
- arithmetic instruction 
- store 

Avoid Id/st stalls: Multi-threading



SIMD & Conditional branches: Ouch!



Kernel Drivers: Memory Management

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



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



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

