Introduction to Vulkan Layers

Baldur Karlsson

Vulkan Layers

- Allows additional functionality to be added above what a driver implements.

Vulkan Layers

- Allows additional functionality to be added above what a driver implements.
- Layers can add, modify or examine all aspects of the API. Layers can even implement extensions that are not supported by the driver.

Vulkan Layers

- Allows additional functionality to be added above what a driver implements.
- Layers can add, modify or examine all aspects of the API. Layers can even implement extensions that are not supported by the driver.
- Pay only for what you use, deactivated layers have zero cost.

Vulkan Layers

- Allows additional functionality to be added above what a driver implements.
- Layers can add, modify or examine all aspects of the API. Layers can even implement extensions that are not supported by the driver.
- Pay only for what you use, deactivated layers have zero cost.
- Primarily for during development, but some layers are used with released products like steam overlay.

Dispatchable Objects

- The loader is the the vulkan dispatch library. vulkan-1.dll or libvulkan1.so.

Dispatchable Objects

- The loader is the the vulkan dispatch library. vulkan-1.dll or libvulkan1.so.
- It doesn’t implement anything itself, it is responsible for enumerating which drivers are installed and calling into them.

Dispatchable Objects

- The loader is the the vulkan dispatch library. vulkan-1.dll or libvulkan1.so.
- It doesn’t implement anything itself, it is responsible for enumerating which drivers are installed and calling into them.
- A VkInstance can have several VkPhysicalDevices - each with a different driver.

Dispatchable Objects

- The loader is the the vulkan dispatch library. vulkan-1.dll or libvulkan1.so.
- It doesn’t implement anything itself, it is responsible for enumerating which drivers are installed and calling into them.
- A VkInstance can have several VkPhysicalDevices - each with a different driver.
- When you call a function like vkCmdDraw() - which driver does it go to?

Dispatchable Objects

- The loader is the the vulkan dispatch library. vulkan-1.dll or libvulkan1.so.
- It doesn’t implement anything itself, it is responsible for enumerating which drivers are installed and calling into them.
- A VkInstance can have several VkPhysicalDevices - each with a different driver.
- When you call a function like vkCmdDraw() - which driver does it go to?
- Answer - it uses whichever driver the objects are set to **dispatch** to.

Dispatchable Objects

- The loader is the the vulkan dispatch library. vulkan-1.dll or libvulkan1.so.
- It doesn’t implement anything itself, it is responsible for enumerating which drivers are installed and calling into them.
- A VkInstance can have several VkPhysicalDevices - each with a different driver.
- When you call a function like vkCmdDraw() - which driver does it go to?
- Answer - it uses whichever driver the objects are set to **dispatch** to.
- Vulkan objects are **dispatchable** or **non-dispatchable**. Dispatchable objects are:
    - VkInstance, VkPhysicalDevice, VkDevice, VkQueue, VkCommandBuffer

Dispatchable Objects

- Dispatchable objects are always pointer-sized.

Dispatchable Objects

- Dispatchable objects are always pointer-sized.
- Look again at the list. It’s no coincidence that **every** vulkan function call takes a dispatchable object as its first parameter:
  - VkInstance, VkPhysicalDevice, VkDevice, VkQueue, VkCommandBuffer

Dispatchable Objects

- Dispatchable objects are always pointer-sized.
- Look again at the list. It’s no coincidence that **every** vulkan function call takes a dispatchable object as its first parameter:
  - VkInstance, VkPhysicalDevice, VkDevice, VkQueue, VkCommandBuffer
- Each object is created from a parent - command buffers created from a device, so at each point we know which driver this object is associated with.

Dispatch tables

- Dispatchable objects are pointer-sized and the memory they point to is mostly driver-defined, as with non-dispatchable objects.

Dispatch tables

- Dispatchable objects are pointer-sized and the memory they point to is mostly driver-defined, as with non-dispatchable objects.
- The loader-layer-driver interface defines that the pointed-to memory contains at its beginning a pointer to a **dispatch table**.

Dispatch tables

- Dispatchable objects are pointer-sized and the memory they point to is mostly driver-defined, as with non-dispatchable objects.
- The loader-layer-driver interface defines that the pointed-to memory contains at its beginning a pointer to a **dispatch table**.
- A dispatch table is a big list of function pointers, with one for every function call in the vulkan API.

Dispatch tables

- Dispatchable objects are pointer-sized and the memory they point to is mostly driver-defined, as with non-dispatchable objects.
- The loader-layer-driver interface defines that the pointed-to memory contains at its beginning a pointer to a **dispatch table**.
- A dispatch table is a big list of function pointers, with one for every function call in the vulkan API.
- Inside vkCmdDraw in the loader, it looks up this table to find which function pointer to call - into the right driver for this object.

Dispatch tables

- Dispatchable objects are pointer-sized and the memory they point to is mostly driver-defined, as with non-dispatchable objects.
- The loader-layer-driver interface defines that the pointed-to memory contains at its beginning a pointer to a **dispatch table**.
- A dispatch table is a big list of function pointers, with one for every function call in the vulkan API.
- Inside vkCmdDraw in the loader, it looks up this table to find which function pointer to call - into the right driver for this object.
- The loader automatically populates these tables behind the scenes. There are two different tables - one for VkInstance objects (driver independent), one for VkDevice objects (driver dependent).

Dispatch chains

- This solves the problem of how to get vkCmdDraw() to the right driver - but how do layers fit in?

Dispatch chains

- This solves the problem of how to get vkCmdDraw() to the right driver - but how do layers fit in?
- There’s no reason that this table has to go directly to the driver. It could call a function in a layer library first which then knows to go to the driver:
  - Loader -> Layer -> Driver

Dispatch chains

- This solves the problem of how to get vkCmdDraw() to the right driver - but how do layers fit in?
- There’s no reason that this table has to go directly to the driver. It could call a function in a layer library first which then knows to go to the driver:
  - Loader -> Layer -> Driver
- Instead of the table calling directly to the driver, you can call one or more layers in a chain of calls before ending up at the driver:
  - Loader -> Layer A -> Layer B -> Layer C -> Driver

Dispatch chains

- This solves the problem of how to get vkCmdDraw() to the right driver - but how do layers fit in?
- There's no reason that this table has to go directly to the driver. It could call a function in a layer library first which then knows to go to the driver:
  - Loader -> Layer -> Driver
- Instead of the table calling directly to the driver, you can call one or more layers in a chain of calls before ending up at the driver:
  - Loader -> Layer A -> Layer B -> Layer C -> Driver
- For every function in the vulkan API, we then have a chain like this - the application calls into the loader, which calls along all the layers, then finally to the driver.

Implementing a layer

- A simple layer is quite easy to implement.

Implementing a layer

- A simple layer is quite easy to implement.
- First you need a JSON manifest registered in a specific location to tell the loader that your layer is available, and where the library file is.

Implementing a layer

- A simple layer is quite easy to implement.
- First you need a JSON manifest registered in a specific location to tell the loader that your layer is available, and where the library file is.
- Then your layer must implement a few entry points, the most important of which are vkGetInstanceProcAddr and vkGetDeviceProcAddr.

Implementing a layer

- A simple layer is quite easy to implement.
- First you need a JSON manifest registered in a specific location to tell the loader that your layer is available, and where the library file is.
- Then your layer must implement a few entry points, the most important of which are vkGetInstanceProcAddr and vkGetDeviceProcAddr.
- The loader uses this to find out which functions your layer implements using these functions as normal.

Implementing a layer

- A simple layer is quite easy to implement.
- First you need a JSON manifest registered in a specific location to tell the loader that your layer is available, and where the library file is.
- Then your layer must implement a few entry points, the most important of which are vkGetInstanceProcAddr and vkGetDeviceProcAddr.
- The loader uses this to find out which functions your layer implements using these functions as normal.
- Inside vkCreateInstance or vkCreateDevice, your layer can find the vkGetInstanceProcAddr and vkGetDeviceProcAddr for the next element in the chain (layer or driver) through the pNext extension chain.

Storing Layer Data

- Each link in the dispatch chain has a unique dispatch table, containing the functions of the next link in the chain where it needs to call.

Storing Layer Data

- Each link in the dispatch chain has a unique dispatch table, containing the functions of the next link in the chain where it needs to call.
- But the dispatch table stored inside the dispatchable object is reserved for the loader, so that it can start the chain. Layers cannot store their own table here.

Storing Layer Data

- Each link in the dispatch chain has a unique dispatch table, containing the functions of the next link in the chain where it needs to call.
- But the dispatch table stored inside the dispatchable object is reserved for the loader, so that it can start the chain. Layers cannot store their own table here.
- There are two primary ways to store layer-specific data:
  1. Map Lookups
  2. Object Wrapping

Storing Layer Data - Map Lookups

- The loader’s device or instance table is unique for each VkInstance chain or VkDevice chain (newly malloc’d each time).

Storing Layer Data - Map Lookups

- The loader’s device or instance table is unique for each VkInstance chain or VkDevice chain (newly malloc’d each time).
- You can use a std::map<> or similar to look up your own dispatch table using the loader’s table as a key.

Storing Layer Data - Map Lookups

- The loader's device or instance table is unique for each VkInstance chain or VkDevice chain (newly malloc'd each time).
- You can use a std::map<> or similar to look up your own dispatch table using the loader's table as a key.
- Similarly you can use the object handle itself as a key, and use a std::map<> to store object-specific data you may need to look up.

Storing Layer Data - Object Wrapping

- Layers can ‘wrap’ all of the vulkan objects.

Storing Layer Data - Object Wrapping

- Layers can ‘wrap’ all of the vulkan objects.
- When a new object - e.g. VkCommandBuffer - is created. The layer allocates new memory, stores the loader’s dispatch table at the start, the real object handle next, and after that as much data as it wants

Storing Layer Data - Object Wrapping

- Layers can ‘wrap’ all of the vulkan objects.
- When a new object - e.g. VkCommandBuffer - is created. The layer allocates new memory, stores the loader’s dispatch table at the start, the real object handle next, and after that as much data as it wants:
  - An ID
  - Its dispatch table
  - A pointer to cold data, etc

Storing Layer Data - Object Wrapping

- Layers can ‘wrap’ all of the vulkan objects.
- When a new object - e.g. VkCommandBuffer - is created. The layer allocates new memory, stores the loader’s dispatch table at the start, the real object handle next, and after that as much data as it wants:
  - An ID
  - Its dispatch table
  - A pointer to cold data, etc
- Any time an object is passed down through the API, the layer ‘unwraps’ the object by looking up the real handle and replacing it for further down the chain.

Storing Layer Data - Summary

Map Lookups

- Simple to implement.

Storing Layer Data - Summary

Map Lookups

- Simple to implement.
- Non-modifying - only requires lookups.

Storing Layer Data - Summary

Map Lookups

- Simple to implement.
- Non-modifying - only requires lookups.
- Thread-unsafe - requires read-write lock for object creation & destruction.

Storing Layer Data - Summary

Map Lookups

- Simple to implement.
- Non-modifying - only requires lookups.
- Thread-unsafe - requires read-write lock for object creation & destruction.
- Per-object map can have problems with duplicate keys (GOOGLE_unique_objects layer).

Storing Layer Data - Summary

Map Lookups

- Simple to implement.
- Non-modifying - only requires lookups.
- Thread-unsafe - requires read-write lock for object creation & destruction.
- Per-object map can have problems with duplicate keys (GOOGLE_unique_objects layer).

Object Wrapping

- Complex to implement.

Storing Layer Data - Summary

Map Lookups

- Simple to implement.
- Non-modifying - only requires lookups.
- Thread-unsafe - requires read-write lock for object creation & destruction.
- Per-object map can have problems with duplicate keys (GOOGLE_unique_objects layer).

Object Wrapping

- Complex to implement.
- Modifying - layer must intercept all entry points to unwrap.

Storing Layer Data - Summary

Map Lookups

- Simple to implement.
- Non-modifying - only requires lookups.
- Thread-unsafe - requires read-write lock for object creation & destruction.
- Per-object map can have problems with duplicate keys (GOOGLE_unique_objects layer).

Object Wrapping

- Complex to implement.
- Modifying - layer must intercept all entry points to unwrap.
- Thread-safe - wrapping happens in-place, no locks required.

Storing Layer Data - Summary

Map Lookups

- Simple to implement.
- Non-modifying - only requires lookups.
- Thread-unsafe - requires read-write lock for object creation & destruction.
- Per-object map can have problems with duplicate keys (GOOGLE_unique_objects layer).

Object Wrapping

- Complex to implement.
- Modifying - layer must intercept all entry points to unwrap.
- Thread-safe - wrapping happens in-place, no locks required.
- Supports duplicate object handles.

Conclusion

- Writing a small layer only takes an hour or two.

Conclusion

- Writing a small layer only takes an hour or two.
- Once you wrap your head around dispatch tables/chains, the hard part is done.

Conclusion

- Writing a small layer only takes an hour or two.
- Once you wrap your head around dispatch tables/chains, the hard part is done.
- Layers can do all sorts of fun things - full extensibility of the API in your hands.

Conclusion

- Writing a small layer only takes an hour or two.
- Once you wrap your head around dispatch tables/chains, the hard part is done.
- Layers can do all sorts of fun things - full extensibility of the API in your hands.
- Questions?

Appendix

- Full article: https://renderdoc.org/vulkan-layer-guide.html
- Sample code: https://github.com/baldurk/sample_layer

Thanks to Matthäus Chajdas, Jasper Bekkers for checking slides :).