Karl Schultz
LunarG, Inc

Vulkan Validation Layers Deep Dive

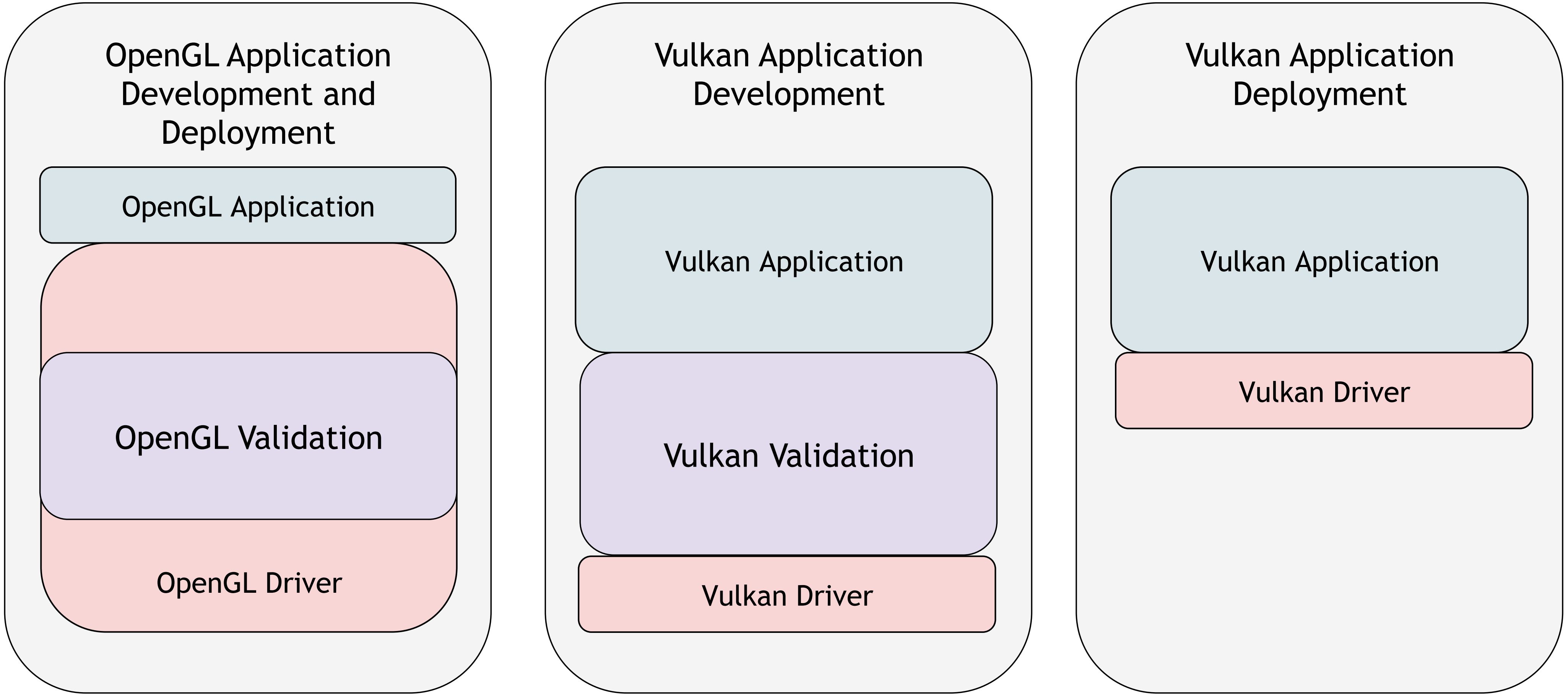
Sept 27, 2016

Agenda

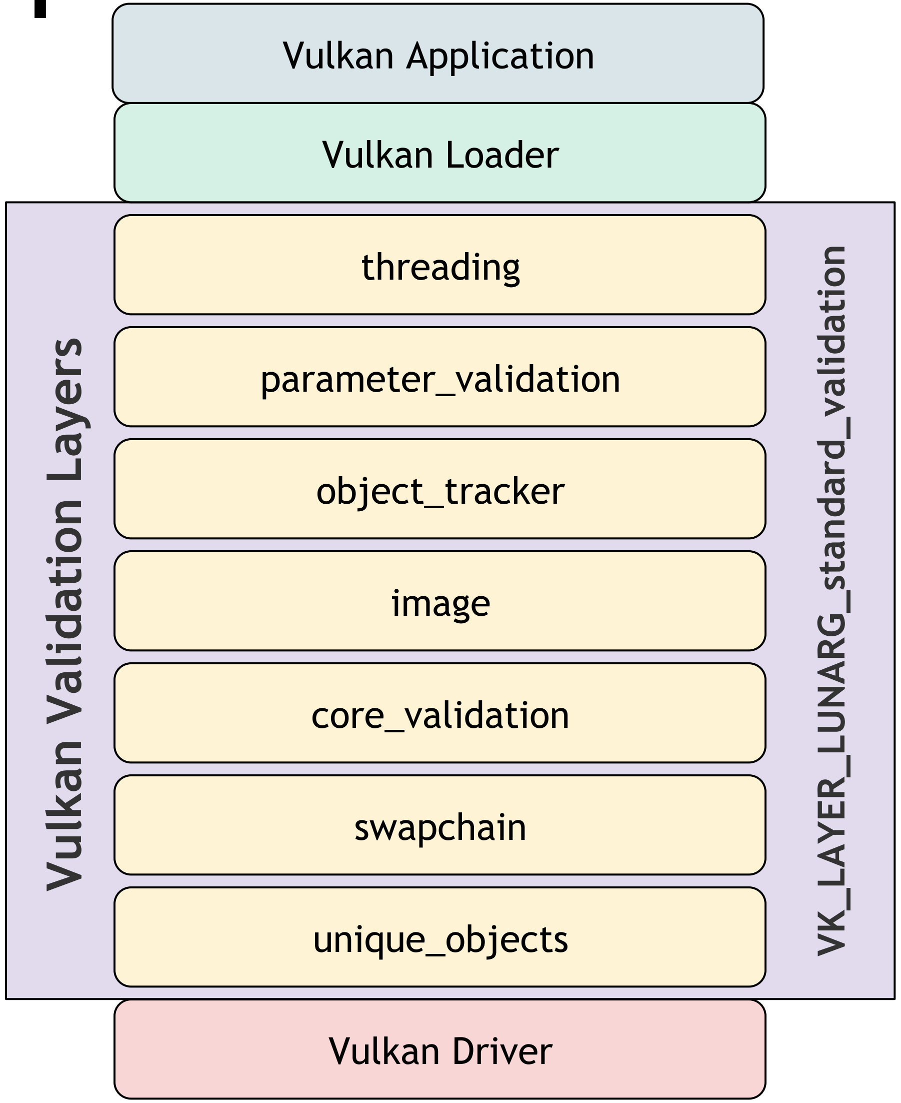
- Basic Vulkan Application Architecture
- Why Use Validation Layers ?
- How to Use Validation Layers
- Detailed Examination of Each Validation Layer
- The Future

Basic Vulkan Application Architecture

OpenGL and Vulkan Application Stacks



Vulkan Application Stack – a Closer Look



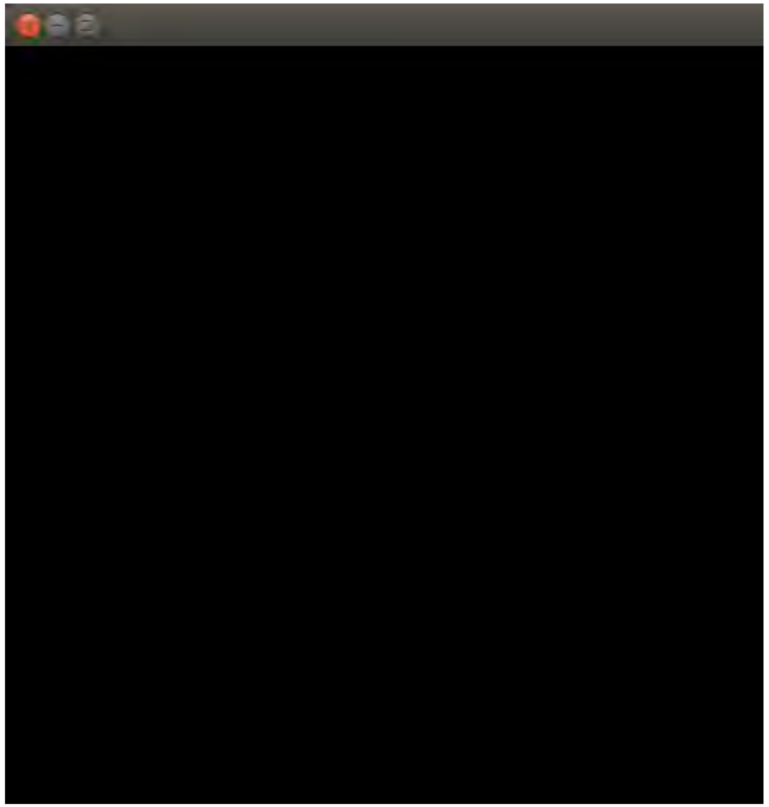
Why Use Validation Layers???

Reasons to Use Validation Layers

- Correctness
- Portability
- Efficiency

Correctness

- Sometimes it is hard to figure out what is wrong!
- No error message mechanism in core Vulkan.
  - No “glGetError”
  - We do have vkResult, but not always useful.
- No messages.
- VkResult all OK.
- What the heck is wrong?
- Validation layers help you follow the Vulkan specification and can help you with debugging.

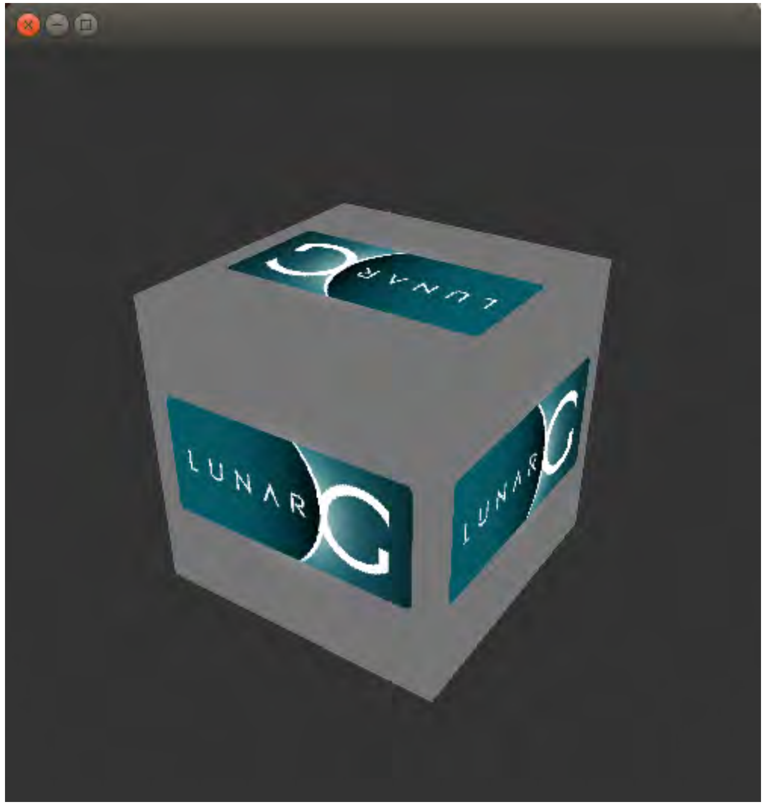


Correctness

- Turn on validation layers and see:

```
ERROR: [DS] Code 31 : You must call
vkEndCommandBuffer() on CB 0x97b8e0 before this
call to vkQueueSubmit()!
```

- Ooops! But easy to fix with this clue.



Portability

- **The Vulkan API is considered to be portable.**
  - Vulkan drivers can be implemented on a wide variety of platforms (e.g., workstations, mobile devices).
  - But this aspect of “portability” does not easily carry over to Vulkan applications. In fact, quite the opposite.

- **Validation layers help you write applications that run correctly on many devices.**
  - They help you catch things that won’t work on devices other than the one you are working on.
  - Your application may run fine on your device, even if it has validation errors.
  - But that may not be the case on a different device.

Efficiency

- **Validation layers themselves don't make the application more efficient at run-time.**
  - Some layers do report performance warnings.
- **But the main gain is getting the validation work out of the way during development and deployment.**
  - Validation can be performed when needed during application testing.
  - Can choose which layers/checks to run.
- **You can be smart about when to run/re-run validation**
  - Validation doesn't always need to be on all the time.
  - Validation can be used during development and taken out of they way as it progresses.

The One Take-Away

If you go away with nothing else from this presentation, then take at least this:

Vulkan Validation Layers are to Vulkan Applications

As

The Vulkan Conformance Test Suite is to Vulkan Drivers

- **Use CTS to validate drivers, and Validation Layers to validate applications.**

You wouldn’t want to use a Vulkan Driver that didn’t pass the CTS, would you?

How to Use Validation Layers

How to use Validation Layers

Topics for the next few slides:

- **Standard validation meta-layer**
- **Two ways to enable layers:**
  - Environment Variables
  - Programmatically at Create Instance time
- **Controlling Output and Validation Layer Actions**
  - vk_layer_settings.txt file
  - Debug Report callback
- **Turning on only some validation layers**

Standard Validation Layer

- **The `VK_LAYER_LUNARG_standard_validation` layer enables the following layers, in this order:**
  - Threading
  - Parameter Checking
  - Object Tracker
  - Image
  - Core Validation
  - Swapchain
  - Unique Object
- **This meta-layer is known to the Vulkan Loader, which simply loads the above layers in the same manner as if they were individually loaded in that order.**
  - It can be thought of as a convenience layer for loading these layers in a configuration that is known to work well.
  - Also protects against the changes in the layer list.

Activating Layers with Environment Variables

- **Easy to do - no code changes**
- **All layers in the standard meta-layer send their messages to stdout by default**
- **To LOCATE the layers on Windows:**
  - If you have the LunarG SDK installed, the Vulkan Loader already knows how to find the validation layers via registry entries.
- **To LOCATE the layers on Linux:**
  - Install the SDK and set the environment variable:

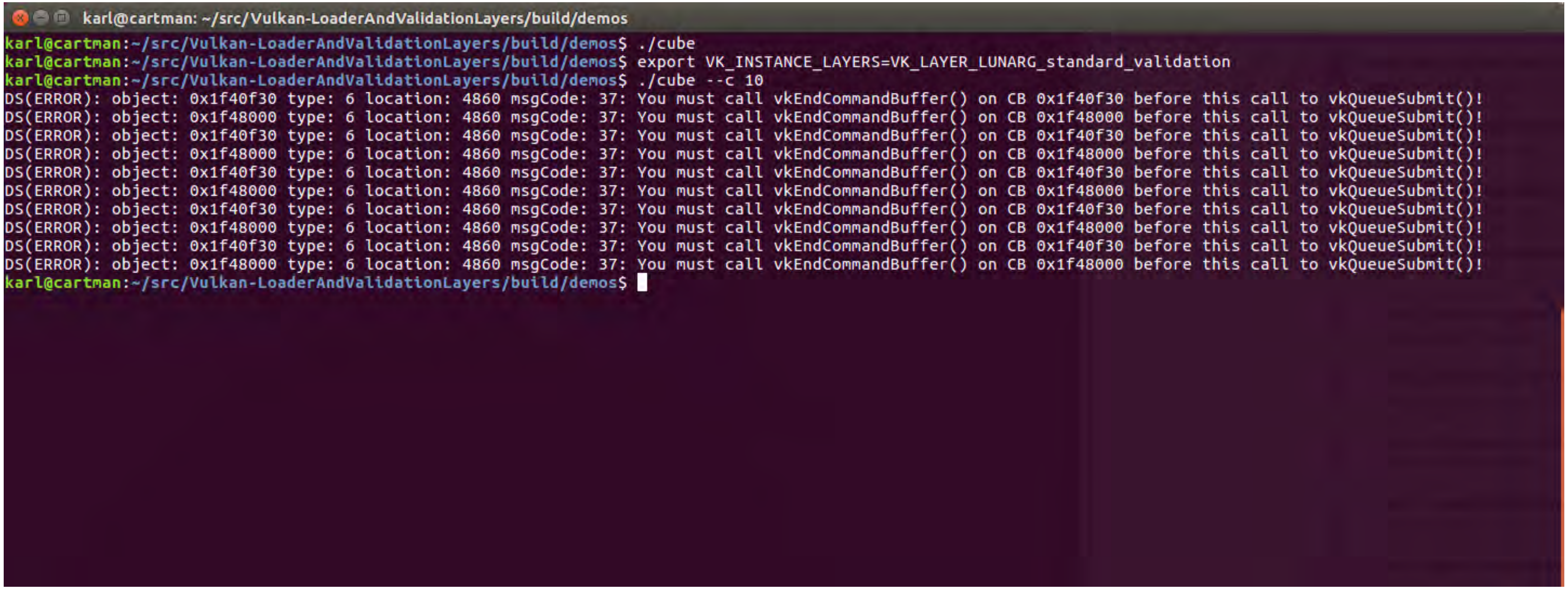
    ```
    VK_LAYER_PATH=<SDK install location>/x86_64/etc/explicit_layer.d.
    ```
  - See the Linux Getting Started Guide in the SDK or LunarXChange website for more detail

- **To ACTIVATE the layers, set the environment variable:**

  ```
  VK_INSTANCE_LAYERS=VK_LAYER_LUNARG_standard_validation
  ```

Layer Activation Example:

No messages seen at first, but turn on validation to see them.
(“—c 10” says to draw 10 times)



karl@cartman: ~/src/Vulkan-LoaderAndValidationLayers/build/demos

```
karl@cartman:~/src/Vulkan-LoaderAndValidationLayers/build/demos$ ./cube
karl@cartman:~/src/Vulkan-LoaderAndValidationLayers/build/demos$ export VK_INSTANCE_LAYERS=VK_LAYER_LUNARG_standard_validation
karl@cartman:~/src/Vulkan-LoaderAndValidationLayers/build/demos$ ./cube --c 10
DS(ERROR): object: 0x1f40f30 type: 6 location: 4860 msgCode: 37: You must call vkEndCommandBuffer() on CB 0x1f40f30 before this call to vkQueueSubmit()!
DS(ERROR): object: 0x1f48000 type: 6 location: 4860 msgCode: 37: You must call vkEndCommandBuffer() on CB 0x1f48000 before this call to vkQueueSubmit()!
DS(ERROR): object: 0x1f40f30 type: 6 location: 4860 msgCode: 37: You must call vkEndCommandBuffer() on CB 0x1f40f30 before this call to vkQueueSubmit()!
DS(ERROR): object: 0x1f48000 type: 6 location: 4860 msgCode: 37: You must call vkEndCommandBuffer() on CB 0x1f48000 before this call to vkQueueSubmit()!
DS(ERROR): object: 0x1f40f30 type: 6 location: 4860 msgCode: 37: You must call vkEndCommandBuffer() on CB 0x1f40f30 before this call to vkQueueSubmit()!
DS(ERROR): object: 0x1f48000 type: 6 location: 4860 msgCode: 37: You must call vkEndCommandBuffer() on CB 0x1f48000 before this call to vkQueueSubmit()!
DS(ERROR): object: 0x1f40f30 type: 6 location: 4860 msgCode: 37: You must call vkEndCommandBuffer() on CB 0x1f40f30 before this call to vkQueueSubmit()!
DS(ERROR): object: 0x1f48000 type: 6 location: 4860 msgCode: 37: You must call vkEndCommandBuffer() on CB 0x1f48000 before this call to vkQueueSubmit()!
DS(ERROR): object: 0x1f40f30 type: 6 location: 4860 msgCode: 37: You must call vkEndCommandBuffer() on CB 0x1f40f30 before this call to vkQueueSubmit()!
DS(ERROR): object: 0x1f48000 type: 6 location: 4860 msgCode: 37: You must call vkEndCommandBuffer() on CB 0x1f48000 before this call to vkQueueSubmit()!
karl@cartman:~/src/Vulkan-LoaderAndValidationLayers/build/demos$
```

Activating Layers Programmatically

- **When you create your instance, you fill in this structure:**

```
typedef struct VkInstanceCreateInfo {
    VkStructureType             sType;
    const void*                 pNext;
    VkInstanceCreateFlags       flags;
    const VkApplicationInfo*    pApplicationInfo;
    uint32_t                    enabledLayerCount;
    const char* const*          ppEnabledLayerNames;
    uint32_t                    enabledExtensionCount;
    const char* const*          ppEnabledExtensionNames;
} VkInstanceCreateInfo;
```

- **Set up your layer list:**

```
 char *instance_validation_layers[] = {
        "VK_LAYER_LUNARG_standard_validation"
    };
```

- **Then:**

```
info->enabledLayerCount = 1;
info->ppEnabledLayerNames = instance_validation_layers;
```

Activating Layers Programmatically - a Tip

- **If you expect to turn validation on/off often, consider an application program option that controls enabling validation layers at Create Instance time.**
  - The cube demo does this. See the –-validate option.
  - This may make it easier on users so that they don't have to set environment variables or hack the source.

The Layer Settings File

- You can use this layer settings file to control what the layers do:
  - Action to take when a validation violation occurs.
  - Where to direct the message output.
- Find the vk_layer_settings.txt file:
  - SDK config directory
- Default settings for one of the validation layers looks like:

```
# VK_LAYER_LUNARG_core_validation Settings
lunarg_core_validation.debug_action = VK_DBG_LAYER_ACTION_LOG_MSG
lunarg_core_validation.report_flags = error,warn,perf
lunarg_core_validation.log_filename = stdout
```

- For example, to send output to a file:

```
lunarg_core_validation.log_filename = core.txt
```

- Put your modified settings file in the “current” directory.
- The settings file contains documentation for using other settings.

Debug Report Callback

- **Each layer uses its own built-in debug report callback to act on a validation violation, according to the settings in the vk_layer_settings file.**
- **You can supply your own callback to take additional actions.**
- **Your callback can:**
  - Examine incoming error to determine its type, severity, message text, etc.
  - Execute any code in response.
  - Notify the user/operator with any message or mechanism.
  - Suppress reporting of known issues, perhaps due to work-in-progress.
  - Signal the validation layer to either:
    - Skip the call to the driver by returning true.
    - Call the driver anyway by returning false.

Example Debug Report Callback

```
VKAPI_ATTR VkBool32 VKAPI_CALL
dbgFunc(VkFlags msgFlags, VkDebugReportObjectTypeEXT objType, uint64_t srcObject, size_t location,
        int32_t msgCode, const char *pLayerPrefix, const char *pMsg, void *pUserData)
{
    char *message = (char *)malloc(strlen(pMsg) + 100);
    if (msgFlags & VK_DEBUG_REPORT_ERROR_BIT_EXT) {
        sprintf(message, "ERROR: [%s] Code %d : %s", pLayerPrefix, msgCode, pMsg);
    } else if (msgFlags & VK_DEBUG_REPORT_WARNING_BIT_EXT) {
        // We know that we're submitting queues without fences, ignore this warning
        if (strstr(pMsg, "vkQueueSubmit parameter, VkFence fence, is null pointer")) {
            return false;
        }
        sprintf(message, "WARNING: [%s] Code %d : %s", pLayerPrefix, msgCode, pMsg);
    } else {
        return false;
    }

#ifdef _WIN32
    MessageBox(NULL, message, "Alert", MB_OK);
#else
    printf("%s\n", message);
    fflush(stdout);
#endif
    free(message);
    return false;
}
```

Activating Individual Validation Layers

- This is easy to do. For example, using environment variables:

```
VK_INSTANCE_LAYERS=VK_LAYER_GOOGLE_threading:VK_LAYER_LUNARG_core_validation
```

Or, using the programmatic approach:

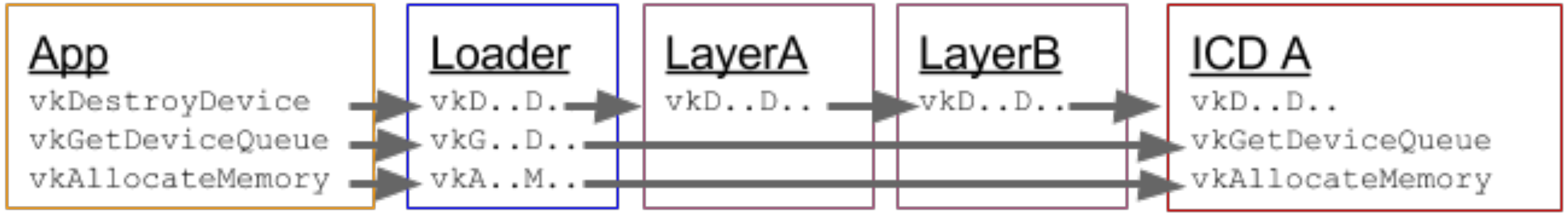
```
char *instance_validation_layers[] = {
        "VK_LAYER_GOOGLE_threading",
        "VK_LAYER_LUNARG_core_validation"
    };
```

- After we finish the deep-dive into each layer, you’ll understand why you might want to do this.

Detailed Examination of Each Validation Layer

How (All) Layers Work, in general

- Layers “hook” Vulkan API calls.
- The Loader places them between the App/Loader and the driver (ICD).
- Layers may hook only a subset of the API.
  - Especially true for some validation layers.



Where Does the Source Code Come From?

- SDK
  - source/layers directory
- GitHub KhronosGroup/Vulkan-LoaderAndValidationLayers repository
  - layers directory

- Don’t worry, we’ll have some URLs for you at the end.
- Note that a lot of code is generated with python scripts.
  - The SDK includes these generated files.
  - You’ll have to build the GitHub repo to see the generated files.

Threading Validation Layer

Threading Validation Layer

- Checks that actions that are using objects are not violating threading restrictions
- These objects are often identified in the Vulkan specification with:

"Host access to *object* <u>must</u> be externally synchronized"

- Taking the VkCommandPool object as an example:
  - We know that Vulkan allows you to record command buffers using multiple threads.
  - But you cannot have more than one thread recording to command buffers from the same VkCommandPool at once.
- If the application is not multithreaded, this layer can be deactivated.

Let's look at some code…

This code comes from `<build_dir>/layers/thread_check.h`
This file is auto-generated from Vulkan specification representations in an XML file.
The comments (in **bold**) below indicate the spec content motivating this check.

```
VKAPI_ATTR void VKAPI_CALL CmdSetViewport(
    VkCommandBuffer                                 commandBuffer,
    uint32_t                                        firstViewport,
    uint32_t                                        viewportCount,
    const VkViewport*                               pViewports)
{
    dispatch_key key = get_dispatch_key(commandBuffer);
    layer_data *my_data = get_my_data_ptr(key, layer_data_map);
    VkLayerDispatchTable *pTable = my_data->device_dispatch_table;
    bool threadChecks = startMultiThread();
    if (threadChecks) {
        startWriteObject(my_data, commandBuffer);
        // Host access to commandBuffer must be externally synchronized
    }
    pTable->CmdSetViewport(commandBuffer,firstViewport,viewportCount,pViewports);
    if (threadChecks) {
        finishWriteObject(my_data, commandBuffer);
        // Host access to commandBuffer must be externally synchronized
    } else {
        finishMultiThread();
    }
}
```

```
VKAPI_ATTR void VKAPI_CALL CmdSetViewport(
    VkCommandBuffer                             commandBuffer,
    uint32_t                                    firstViewport,
    uint32_t                                    viewportCount,
    const VkViewport*                           pViewports)
{
    dispatch_key key = get_dispatch_key(commandBuffer);
    layer_data *my_data = get_my_data_ptr(key, layer_data_map);
    VkLayerDispatchTable *pTable = my_data->device_dispatch_table;
    bool threadChecks = startMultiThread();
```

The (**bold**) code above obtains access to the layer’s tracking data.

```
VKAPI_ATTR void VKAPI_CALL CmdSetViewport(
    VkCommandBuffer                                 commandBuffer,
    uint32_t                                        firstViewport,
    uint32_t                                        viewportCount,
    const VkViewport*                               pViewports)
{
    dispatch_key key = get_dispatch_key(commandBuffer);
    layer_data *my_data = get_my_data_ptr(key, layer_data_map);
    VkLayerDispatchTable *pTable = my_data->device_dispatch_table;
    bool threadChecks = startMultiThread();
```

`startMultiThread()` detects multithreading.

Multithreading Detection

In `startMultiThread()`:

- **Basically uses two flags visible to all threads**
- **First thread turns on a flag (static volatile) that indicates "In API"**
  - The same flag gets turned off by the thread in `finishMultiThread()` when leaving `CmdSetViewport()`
- **If second thread enters and sees the "In API" flag turned on, it sets a "multithreading present" (static volatile) flag and returns true**
  - This "multithreading present" flag is "sticky" and never goes off
- **If the "multithreading present" flag is on when `startMultiThread()` is entered, it just returns true.**
  - This is a fast path for the case there multithreading has already been determined to be present.

```
VKAPI_ATTR void VKAPI_CALL CmdSetViewport(
    VkCommandBuffer                                 commandBuffer,
    uint32_t                                        firstViewport,
    uint32_t                                        viewportCount,
    const VkViewport*                               pViewports)
{
    dispatch_key key = get_dispatch_key(commandBuffer);
    layer_data *my_data = get_my_data_ptr(key, layer_data_map);
    VkLayerDispatchTable *pTable = my_data->device_dispatch_table;
    bool threadChecks = startMultiThread();
    if (threadChecks) {
        startWriteObject(my_data, commandBuffer);
```

`startWriteObject()` updates reader and writer counts for an object.

In this case, it finds the `VkCommandPool` that contains the commandBuffer and updates the counts for that object.

If the numbers of concurrent readers and writers is not allowed by the spec, then it emits a validation error.

```
VKAPI_ATTR void VKAPI_CALL CmdSetViewport(
    VkCommandBuffer                                commandBuffer,
    uint32_t                                       firstViewport,
    uint32_t                                       viewportCount,
    const VkViewport*                              pViewports)
{
    dispatch_key key = get_dispatch_key(commandBuffer);
    layer_data *my_data = get_my_data_ptr(key, layer_data_map);
    VkLayerDispatchTable *pTable = my_data->device_dispatch_table;
    bool threadChecks = startMultiThread();
    if (threadChecks) {
        startWriteObject(my_data, commandBuffer);
        // Host access to commandBuffer must be externally synchronized
    }
    pTable->CmdSetViewport(commandBuffer,firstViewport,viewportCount,pViewports);
```

Call the next layer in the chain.

```
VKAPI_ATTR void VKAPI_CALL CmdSetViewport(
    VkCommandBuffer                            commandBuffer,
    uint32_t                                   firstViewport,
    uint32_t                                   viewportCount,
    const VkViewport*                          pViewports)
{
    dispatch_key key = get_dispatch_key(commandBuffer);
    layer_data *my_data = get_my_data_ptr(key, layer_data_map);
    VkLayerDispatchTable *pTable = my_data->device_dispatch_table;
    bool threadChecks = startMultiThread();
    if (threadChecks) {
        startWriteObject(my_data, commandBuffer);
        // Host access to commandBuffer must be externally synchronized
    }
    pTable->CmdSetViewport(commandBuffer,firstViewport,viewportCount,pViewports);
    if (threadChecks) {
        finishWriteObject(my_data, commandBuffer);
        // Host access to commandBuffer must be externally synchronized
    } else {
        finishMultiThread();
    }
}
```

Update the reader and writer counts as a result of leaving and turn off “in API” flag.

Parameter Validation Layer

Parameter Validation Layer

- **This layer performs stateless API parameter validation.**
  - valid usage
  - device limits
- **This layer is near the top.**
  - Good to check early for NULL pointers and fundamental problems that will certainly cause trouble (crashes) later.
  - Subsequent layers don’t need to re-check these things. And they don’t!
- **Any errors reported by this layer should probably be fixed with high priority.**
  - Likely to be simple, yet potentially catastrophic errors.
  - Will allow later layers (and the driver!) to function properly.

Code please …

This code comes from `layers/parameter_validation.cpp`

```
VKAPI_ATTR void VKAPI_CALL
CmdSetViewport(VkCommandBuffer commandBuffer, uint32_t firstViewport,
               uint32_t viewportCount, const VkViewport *pViewports) {
    bool skip_call = false;
    layer_data *my_data =
        get_my_data_ptr(get_dispatch_key(commandBuffer), layer_data_map);
    assert(my_data != NULL);

    skip_call |=
        parameter_validation_vkCmdSetViewport(my_data->report_data,
                                              firstViewport, viewportCount,
                                              pViewports);


    if (!skip_call) {
        get_dispatch_table(pc_device_table_map, commandBuffer)
            ->CmdSetViewport(commandBuffer, firstViewport,
                             viewportCount, pViewports);
    }
}
```

```
VKAPI_ATTR void VKAPI_CALL
CmdSetViewport(VkCommandBuffer commandBuffer, uint32_t firstViewport,
               uint32_t viewportCount, const VkViewport *pViewports) {
    bool skip_call = false;
    layer_data *my_data =
        get_my_data_ptr(get_dispatch_key(commandBuffer), layer_data_map);
    assert(my_data != NULL);
```

Code to get the layer’s tracking data.

```
VKAPI_ATTR void VKAPI_CALL
CmdSetViewport(VkCommandBuffer commandBuffer, uint32_t firstViewport,
               uint32_t viewportCount, const VkViewport *pViewports) {
    bool skip_call = false;
    layer_data *my_data =
        get_my_data_ptr(get_dispatch_key(commandBuffer), layer_data_map);
    assert(my_data != NULL);

    skip_call |=
        parameter_validation_vkCmdSetViewport(my_data->report_data,
                                              firstViewport, viewportCount,
                                              pViewports);
```

This is a call to a helper function that performs the actual checking.

We'll take a look at the function soon, but note that it could set `skip_call` to indicate that the API call should not be passed down the layer/driver chain. This is a common pattern in the layers.

```
VKAPI_ATTR void VKAPI_CALL
CmdSetViewport(VkCommandBuffer commandBuffer, uint32_t firstViewport,
               uint32_t viewportCount, const VkViewport *pViewports) {
    bool skip_call = false;
    layer_data *my_data =
        get_my_data_ptr(get_dispatch_key(commandBuffer), layer_data_map);
    assert(my_data != NULL);

    skip_call |=
        parameter_validation_vkCmdSetViewport(my_data->report_data,
                                              firstViewport, viewportCount,
                                              pViewports);

    if (!skip_call) {
        get_dispatch_table(pc_device_table_map, commandBuffer)
            ->CmdSetViewport(commandBuffer, firstViewport,
                             viewportCount, pViewports);
    }
}
```

Finally, this code just calls down the chain if it is OK to continue.

Let's look at `parameter_validation_vkCmdSetViewport()`
This code comes from `<build_dir>/layers/parameter_validation.h`
This file is auto-generated from Vulkan specification representations in an XML file.

```
static bool parameter_validation_vkCmdSetViewport(
    debug_report_data*                           report_data,
    uint32_t                                     firstViewport,
    uint32_t                                     viewportCount,
    const VkViewport*                            pViewports)
{
    UNUSED_PARAMETER(firstViewport);

    bool skipCall = false;

    skipCall |= validate_array(report_data, "vkCmdSetViewport",
                               "viewportCount", "pViewports",
                               viewportCount, pViewports, true, true);

    return skipCall;
}
```

This code just calls another utility…

This code comes from `layers/parameter_validation_utils.h`
Part of `validate_array()` looks like:

```
    // Count parameters not tagged as optional cannot be 0
    if ((count == 0) && countRequired) {
        skip_call |= log_msg(report_data, VK_DEBUG_REPORT_ERROR_BIT_EXT,
                             VK_DEBUG_REPORT_OBJECT_TYPE_UNKNOWN_EXT, 0,
                             __LINE__, REQUIRED_PARAMETER, LayerName,
                             "%s: parameter %s must be greater than 0",
                             apiName,
                             countName.get_name().c_str());
    }
```

The specification says that `vkCmdSetViewport()` cannot be called with a viewport count of 0. The caller of `validate_array()` sets `countRequired` to true, causing this code to log a validation error if count is 0.

The Debug Report Callback (reached via log_msg()) is ultimately responsible for deciding whether to skip or not.

Valid Usage Coverage - How are we doing?

- **Valid Usage for vkCmdSetViewport (1.0.27 spec)**
  - ☑ commandBuffer must be a valid VkCommandBuffer handle
  - ☑ pViewports must be a pointer to an array of viewportCount valid VkViewport structures
  - ☒ commandBuffer must be in the recording state
  - ☒ The VkCommandPool that commandBuffer was allocated from must support graphics operations
  - ☑ viewportCount must be greater than 0
  - ☒ The currently bound graphics pipeline must have been created with the VK_DYNAMIC_STATE_VIEWPORT dynamic state enabled
  - ☒ firstViewport must be less than VkPhysicalDeviceLimits::maxViewports
  - ☒ The sum of firstViewport and viewportCount must be between 1 and vkPhysicalDeviceLimits::maxViewports, inclusive
- **Some of the static parameter checking is covered.**
  - The last two should be checked by this layer, but the implementation isn't complete.
- **Other unchecked items could be checked in core_validation (coming up).**
- **Note that the `VkViewport` structure contents are not checked yet either.**
- **Eventual goal is to cover all of the above Valid Usage items.**

Object Tracker Validation Layer

Object Tracker Validation Layer

- **Validates objects for correctness, proper creation, and lifetime.**
- **The use case for `vkCmdSetViewport` is pretty boring because the only thing to check is that the CommandBuffer is valid.**
  - Same goes for all `vkCmd*` functions.
- **Let's look at `vkDestroyBuffer()`.**
  - Need to make sure that the device and buffer are valid objects.
  - But also would be good to check if the application tries to destroy a buffer twice.
- **Object Tracker keeps a record of objects as they are created and destroyed.**
  - It then knows if the application tries to use an object that does not exist.

This code comes from `layers/object_tracker.cpp`

```
VKAPI_ATTR void VKAPI_CALL DestroyBuffer(VkDevice device, VkBuffer buffer,
                             const VkAllocationCallbacks *pAllocator) {
    bool skip_call = false;
    {
        std::lock_guard<std::mutex> lock(global_lock);
        skip_call |= ValidateNonDispatchableObject(device, buffer,
            VK_DEBUG_REPORT_OBJECT_TYPE_BUFFER_EXT, false);
        skip_call |= ValidateDispatchableObject(device, device,
            VK_DEBUG_REPORT_OBJECT_TYPE_DEVICE_EXT, false);
    }
    if (skip_call) {
        return;
    }
    {
        std::lock_guard<std::mutex> lock(global_lock);
        DestroyNonDispatchableObject(device, buffer,
            VK_DEBUG_REPORT_OBJECT_TYPE_BUFFER_EXT);
    }
    get_dispatch_table(ot_device_table_map, device)->DestroyBuffer(device,
        buffer, pAllocator);
}
```

```
VKAPI_ATTR void VKAPI_CALL DestroyBuffer(VkDevice device, VkBuffer buffer,
                                 const VkAllocationCallbacks *pAllocator) {
    bool skip_call = false;
    {
        std::lock_guard<std::mutex> lock(global_lock);
        skip_call |= ValidateNonDispatchableObject(device, buffer,
            VK_DEBUG_REPORT_OBJECT_TYPE_BUFFER_EXT, false);
        skip_call |= ValidateDispatchableObject(device, device,
            VK_DEBUG_REPORT_OBJECT_TYPE_DEVICE_EXT, false);
    }
    if (skip_call) {
        return;
    }
```

The above Validate* calls make sure that the device and buffer are valid.
If either is not valid, these calls log an ERROR message and proceed by returning false.
Since skip_call is stays false, we continue on…

```
VKAPI_ATTR void VKAPI_CALL DestroyBuffer(VkDevice device, VkBuffer buffer,
                                         const VkAllocationCallbacks *pAllocator) {
…
    {
        std::lock_guard<std::mutex> lock(global_lock);
        DestroyNonDispatchableObject(device, buffer,
            VK_DEBUG_REPORT_OBJECT_TYPE_BUFFER_EXT);
    }
    get_dispatch_table(ot_device_table_map, device)->DestroyBuffer(device,
        buffer, pAllocator);
}
```

This function destroys the tracking data for the Object Tracker layer by calling `DestroyNonDispatchableObject()`.
If the object doesn't exist, this function issues an error.
A double-destroy of a Buffer object might then generate these validation errors:

```
ERROR: [ObjectTracker] Code 4 : Invalid Buffer Object 0x11
ERROR: [ObjectTracker] Code 1 : Unable to remove Buffer obj 0x11. Was it created? Has it already been
destroyed?
```

Finally, the function calls down the chain. The driver *may* crash.

Image Validation Layer

Image Validation Layer

- Validates items related to image generation and usage.
- Good News! We're not going to look at any code here!
- This layer performs the same type of checking as core_validation, but with an emphasis on image objects.
- Over time, core_validation added more and more image checking.
- We expect, soon, that the image layer will be merged into core_validation.

Core Validation Layer

Core Validation Layer

- Main layer for validation
  - Draw_state
  - Shader_checker
  - Mem_tracker
  - Device_limits
- Many checks for stateful consistency and correctness across Vulkan objects.
- Let’s look at our old friend `vkCmdSetViewport()` in core_validation:

This code comes from `layers/core_validation.cpp`

```
KAPI_ATTR void VKAPI_CALL CmdSetViewport(VkCommandBuffer commandBuffer,
        uint32_t firstViewport, uint32_t viewportCount,
        const VkViewport *pViewports) {
    bool skip_call = false;
    layer_data *dev_data =
        get_my_data_ptr(get_dispatch_key(commandBuffer), layer_data_map);
    std::unique_lock<std::mutex> lock(global_lock);
    GLOBAL_CB_NODE *pCB = getCBNode(dev_data, commandBuffer);
    if (pCB) {
        skip_call |= addCmd(dev_data, pCB, CMD_SETVIEWPORTSTATE,
                                        "vkCmdSetViewport()");
        pCB->status |= CBSTATUS_VIEWPORT_SET;
        pCB->viewportMask |= ((1u<<viewportCount) - 1u) << firstViewport;
    }
    lock.unlock();
    if (!skip_call)
        dev_data->device_dispatch_table->CmdSetViewport(commandBuffer,
            firstViewport, viewportCount, pViewports);
}
```

Really, the only interesting thing going on here is the call to `addCmd()`, which checks Command Buffer info …

Core Validation for CmdSetViewport

- Here's an update on Valid Usage coverage for `vkCmdSetViewport()`:
  - ☑ commandBuffer must be a valid VkCommandBuffer handle
  - ☑ pViewports must be a pointer to an array of viewportCount valid VkViewport structures
  - ☑ commandBuffer must be in the recording state
  - ☑ The VkCommandPool that commandBuffer was allocated from must support graphics operations
  - ☑ viewportCount must be greater than 0
  - ☒ The currently bound graphics pipeline must have been created with the VK_DYNAMIC_STATE_VIEWPORT dynamic state enabled
  - ☒ firstViewport must be less than VkPhysicalDeviceLimits::maxViewports
  - ☒ The sum of firstViewport and viewportCount must be between 1 and vkPhysicalDeviceLimits::maxViewports, inclusive
- The call to `addCmd()` makes two more checks, which are indicated above.
- The dynamic state check looks like it should be done as a Draw State check in core validation.
- The last two are potential TODOs for the parameter validation layer.
- Our list in filling in, but there are three missing checks.

Draw State Example

- **Snippet from `CmdSetLineWidth()` in** `layers/core_validation.cpp`

```
PIPELINE_NODE *pPipeTrav =
    pCB->lastBound[VK_PIPELINE_BIND_POINT_GRAPHICS].pipeline_node;
if (pPipeTrav != NULL && !isDynamic(pPipeTrav, VK_DYNAMIC_STATE_LINE_WIDTH)) {
    skip_call |= log_msg(dev_data->report_data,
                         VK_DEBUG_REPORT_WARNING_BIT_EXT,
                         (VkDebugReportObjectTypeEXT)0,
                         reinterpret_cast<uint64_t &>(commandBuffer),
                         __LINE__, DRAWSTATE_INVALID_SET, "DS",
                         "vkCmdSetLineWidth called but pipeline was created "
                         "without VK_DYNAMIC_STATE_LINE_WIDTH "
                         "flag.  This is undefined behavior and could be "
                         " ignored.");
}
```

This is a Draw State check to make sure that the pipeline has its dynamic state flag set for line width. Similar code could be added to `CmdSetViewport()`. It is just missing at the moment.

Shader Checker Example

- **Snippet from `validate_push_constant_block_against_pipeline()` in** `layers/core_validation.cpp:`

```
 if ((range.stageFlags & stage) == 0) {
     if (log_msg(report_data, VK_DEBUG_REPORT_ERROR_BIT_EXT,
                 VkDebugReportObjectTypeEXT(0), 0,
                 __LINE__,
                 SHADER_CHECKER_PUSH_CONSTANT_NOT_ACCESSIBLE_FROM_STAGE,
                 "SC",
                 "Push constant range covering variable starting at "
                 "offset %u not accessible from stage %s",
                 offset, string_VkShaderStageFlagBits(stage))) {
        pass = false;
    }
}
```

This checks to see if the current stage (stage) has its bit set in the pipeline's push constant range flags.

Memory Tracker Example - Leak Detection

- **Snippet from `DestroyDevice()` in** `layers/core_validation.cpp:`

For each tracked memory object do:
Set pInfo to memory object’s tracking info, then:

```
if (pInfo->alloc_info.allocationSize != 0) {
    // Valid Usage: All child objects created on device must
    // have been destroyed prior to destroying device
    skip_call |= log_msg(dev_data->report_data, VK_DEBUG_REPORT_ERROR_BIT_EXT,
        VK_DEBUG_REPORT_OBJECT_TYPE_DEVICE_MEMORY_EXT,
        (uint64_t)pInfo->mem, __LINE__, MEMTRACK_MEMORY_LEAK, "MEM",
        "Mem Object 0x%" PRIx64 " has not been freed. "
        "You should clean up this memory by calling "
        "vkFreeMemory(0x%" PRIx64 ") prior to vkDestroyDevice().",
        (uint64_t)(pInfo->mem), (uint64_t)(pInfo->mem));
}
```

The Memory Tracker layer creates an internal tracking object for each memory allocation.
Memory Tracker deletes these objects when the application frees the memory allocations.
At DestroyDevice time, any existing tracking memory objects constitute leaks.

Device Limits Example

- **Snippet from `ValidateFramebufferCreateInfo()` in** `layers/core_validation.cpp:`

```
// Verify FB dimensions are within physical device limits
if ((pCreateInfo->height > physLimits->maxFramebufferHeight) ||
    (pCreateInfo->width > physLimits->maxFramebufferWidth) ||
    (pCreateInfo->layers > physLimits->maxFramebufferLayers)) {
        skip_call |= log_msg(dev_data->report_data,
            VK_DEBUG_REPORT_ERROR_BIT_EXT, VkDebugReportObjectTypeEXT(0), 0,__LINE__,
            DRAWSTATE_INVALID_FRAMEBUFFER_CREATE_INFO, "DS",
            "vkCreateFramebuffer(): Requested VkFramebufferCreateInfo "
            "dimensions exceed physical device limits. "
            "Here are the respective dimensions: requested, device max:\n"
            "width: %u, %u\n"
            "height: %u, %u\n"
            "layerCount: %u, %u\n",
            pCreateInfo->width, physLimits->maxFramebufferWidth,
            pCreateInfo->height, physLimits->maxFramebufferHeight,
            pCreateInfo->layers, physLimits->maxFramebufferLayers);
}
```

Fairly straightforward device limits check.   Part of Draw State checking.

Swapchain Validation Layer

Swapchain Validation Layer

- Validates swapchain-related API calls
- Good News! We’re not going to look at any code here!
- This layer performs the same type of checking as core_validation, but with an emphasis on swapchain objects
- Over time, core_validation added more and more swapchain checking
- We expect, soon, that the swapchain layer will be merged into core_validation
  - Actually, expected within the next couple of weeks!

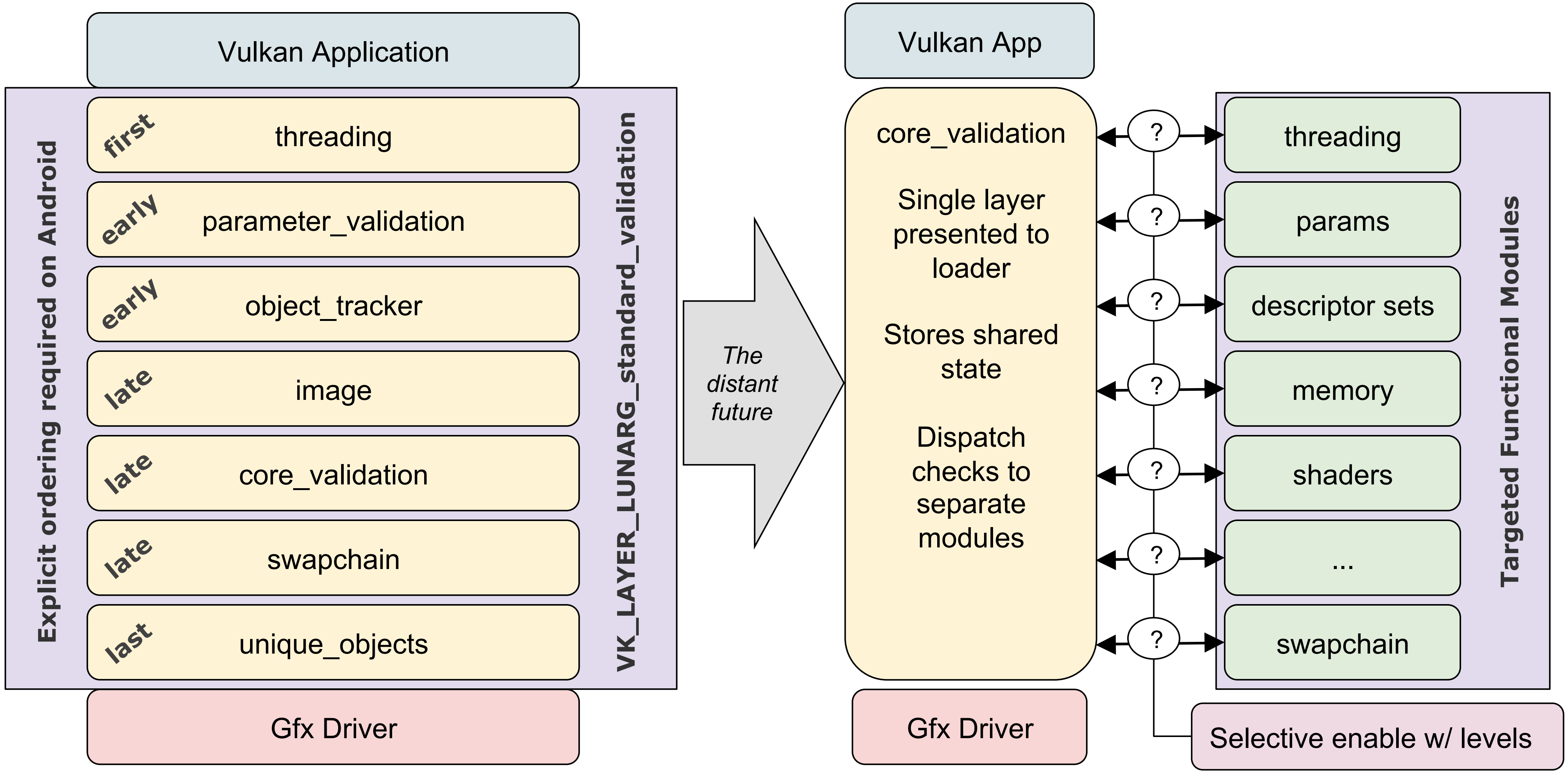
Unique Objects Validation Layer

Unique Objects Layer

- **This layer really doesn't perform any validation checks.**
  - It is more of a utility, but implemented as a layer.
- **Problem: Some drivers return non-unique handles for objects.**
  - Objects of different classes can have the same handle (e.g., vkImage and vkBuffer).
  - Drivers can also reuse a handle after its object has been destroyed.
- **To address this, the Unique Objects layer assigns its own unique handles to objects, replacing those assigned by the driver.**
  - Handle returned to the application not the same as the handle returned by the driver.
  - Application will "see" different handles when layers are enabled.
- **Not supposed to be a problem for applications because handles are opaque.**
- **Layers "unwrap" the handles before passing back to the driver.**
- **Can be tricky for Vulkan extensions.**
  - Beyond the scope of this discussion, but being worked on.

The Future for Validation Layers

Validation Layers - Future



Questions and Credits

- We're about done for today!

- Mark Lobodzinski is up next to help answer any questions you've been submitting.

- Lots of people working on the validation layers!
  - Engineers from LunarG and Google
  - Check out the GitHub repo!

Call to Action

- Use the Validation Layers during application development!!!!
- File issues on the GitHub Vulkan-LoaderAndValidationLayers repository
- Pull requests appreciated!

References / Links:

- LunarG LunarXChange website (for SDK and docs):
  - https://vulkan.lunarg.com
- GitHub Vulkan-LoaderAndValidationLayers repository
  - https://github.com/KhronosGroup/Vulkan-LoaderAndValidationLayers

Happy Validating!!!!