PowerVR Framework

Development Guide

Filename	:	PowerVR Framework.Development Guide
Version	:	PowerVR SDK REL_18.1@5080009 External Issue
Issue Date	:	31 May 2018
Author	:	Imagination Technologies Limited

Contents

List of Figures

1. Overview of the PowerVR Framework

The PowerVR Framework, also referred to as the Framework, is a collection of libraries that is intended to serve as the basis for a graphical application. It is made up of code files, header files and several platforms' project files that group those into modules, also referred to as libraries.

The PowerVR SDK aims to:

- Facilitate development using low level graphics APIs (OpenGL ES, Vulkan)
- Promote best practices using these APIs
- Show and encourage optimal API use patterns, tips and tricks for writing multi-platform code, while also ensuring optimal behaviour for the PowerVR platforms
- Demonstrate variations of rendering techniques that function optimally on PowerVR platforms.

The purpose of the PowerVR Framework is to find the perfect balance between raw code and engine code. In other words:

- It is fast and easy to get going with - for example, default parameters on all Vulkan objects
- For Vulkan, it thinly wraps the raw objects to make things easier and more convenient. Whilst the raw Vulkan must provide a C interface, the Framework through PVRVk provides a C++ interface.
  This allows reference counting and STL objects, and in general makes coding much easier and shorter.
- It is obvious what the code does to someone used to the raw APIs from looking at any example
- Any differences from the raw APIs such as Vulkan lifecycle management are documented.

Note: This document has been written assuming the reader has a general familiarity with the 3D graphics programming pipeline, and some knowledge of OpenGL ES (version 2 onwards) and/or Vulkan.

1.1. Libraries

All the PowerVR Framework libraries are provided as source code, and compiled by default as static libraries. A developer could theoretically configure these as dynamic libraries (.dll/.so and so on) to allow dynamic binding, but no such attempt has been done in the SDK.

We use CMake to build our demos and framework modules. The demos depend on the Framework, so a developer can always just use CMake on the demo they are interested in, and it will build everything required. We also provide top-level CMakeLists that build the entire SDK as it does not take that long to build – generally a few minutes.

The Framework is a high-level C++ project, so no sterilised C APIs exist. This means that the libraries and the final executable should always be compiled with the same compiler make/version with compatible parameters to ensure that C++ rules such as the One Definition Rule (ODR) is observed. This is one of the reasons we provide a common .cmake used by both the examples and the Framework. The compilers must use the same C++ name mangling rules and other details, otherwise the behaviour may be unexpected.

1.1.1. Default library locations

The provided CMake files place Framework library files into:

```
[SDKROOT]/framework/bin/[PLATFORM,CONFIG…]/
```

For example:

```
c:\Imagination\PowerVR_SDK\Framework\bin\Windows_x86_64\Debug\PVRVk.lib
```

or:

```
//home/myself/PowerVR_SDK/Framework/bin/Linux_x86_64/Debug_X11/libPVRVk.a
```

1.2. Additional header files

The PowerVR Framework contains some useful header files that solve some of the most problematic OpenGL problems, and bring Vulkan closer to the C++ world.

Those header files have no dependencies and are not part of any module. They can be used exactly as they are, and each one functions individually. They can all be found in `[SDKROOT]/include`.

It is highly recommended that developers read about and use these header files as they are very beneficial.

1.2.1. DynamicGles.h/DynamicEGL.h

`DynamicGles.h` and `DynamicEgl.h` are based upon similar principles - they provide a convenient single header file solution that loads OpenGL ES/EGL. This includes all supported extensions, dynamically and without statically linking to anything, by using advanced C++ features.

To use them, drop the files somewhere where the compiler will find the header file which is generally wherever library header files are usually stored. Write "`#include DynamicGLES.h`" at the top of the code file, and everything will work – the code will have OpenGL ES functions available.

There is no need to link against EGL, or define function pointers.

It is still necessary to test for extension support, and there is a function for that in EGL. However, everything else is automatic including loading the extension function pointers.

The libraries (`libGLESv2.lib`/`so`, `libEGL.lib`/`so`) do not need to be linked to as they are loaded at runtime. However, they do need to be present on the platform where the application runs.

`DynamicGles.h` can limit the compile-time OpenGL ES version, minimum 2. This can be done by defining `DYNAMICGLES_GLES2`, `DYNAMICGLES_GLES3`, `DYNAMICGLES_GLES31` or `DYNAMICGLES_GLES32`. The default is always the highest supported.

These are all replacements for the corresponding `gl2.h`/`gl3.h`/`gl2ext.h`/`egl.h`, so do not include those directly.

Whenever possible, namespaces are used to group symbols and keep the global namespace as clear as possible.

- `DynamicGles.h` places functions in `gl::` (so `glGenBuffers` becomes `gl::GenBuffers`) unless `DYNAMICGLES_NO_NAMESPACE` is defined before including the file.
- `DynamicEGL.h` places functions in `egl::` (so `eglSwapBuffers` becomes `egl::SwapBuffers`) unless `DYNAMICEGL_NO_NAMESPACE` is defined before including the file.

1.2.2. vk_bindings.h/vk_bindings_helper.h

In Vulkan, each driver or device or Vulkan installation in the developer's system is represented as Vulkan underlying objects. For example, there is the `VkInstance`, the `VkPhysicalDevice`, `VkDisplay` and others. Vulkan is called by calling the `vkXXXXXXX` globally and passing the corresponding object - for example call `vkCreateBuffer(myVulkanDevice…)`

The problem is that objects that belong to different `VkPhysicalDevices` need to dispatch to different functions as they use different drivers to implement the functions. The Vulkan Loader (the `vulkan-1.dll` that is linked against) will need to do this dispatching, causing a needless level of indirection. This is because the global function is called, which is in the loader. It then determines which device it needs to be dispatched to, and calls that function, and then returns the result.

The Vulkan recommended way to tackle this is simple. Do not use the global functions for functions dispatched to a VkDevice – these are functions that get a VkDevice as their first parameter. Instead, get function pointers for the functions specific to that device, so that the dispatching can be skipped.

`vk_bindings.h` does exactly this. It is an auto-generated file that provides all the function pointer definitions and loading code to get per-instance and per-device function pointers.

1.3. Overview of the PowerVR Framework Modules

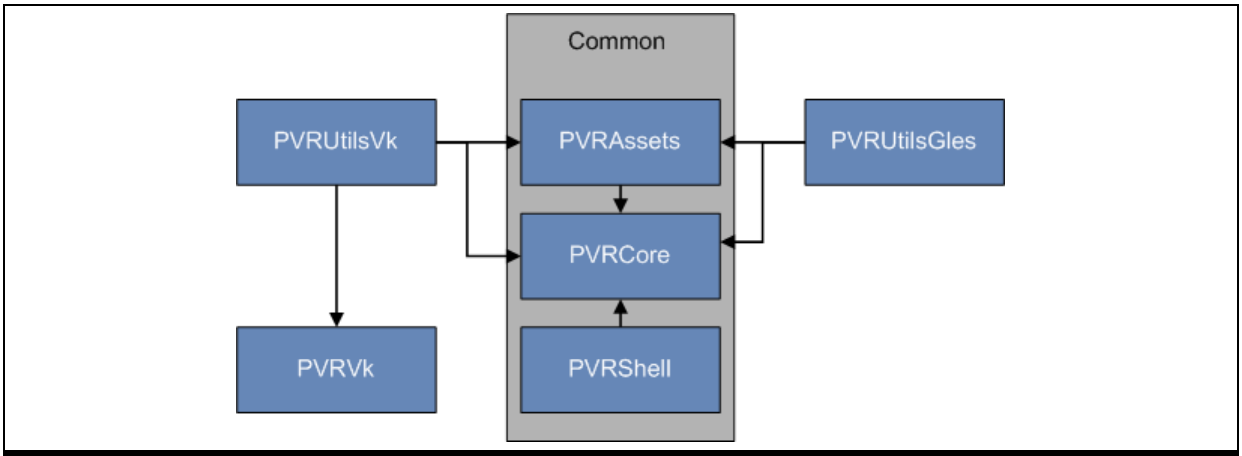

Figure 1. Framework Structure

1.3.1. PVRShell

About PVRShell

PVRShell**,** and especially the `pvr::Shell` class is the scaffolding of an application. It implements the entry point (`main`) of the application and provides convenient places to immediately start coding application logic.

It is intended for the main structure of an application to be a class implementing `pvr::Shell` as seen in The Skeleton of a Typical Framework Application. This way, all of the per-platform initialisation (creating the window, reading command line parameters, calling a function at initialisation, every frame, and teardown) is taken care of. Every conceivable platform operation such as loading files from device-specific storages is provided.

Its public contents can easily be accessed from inside the application class itself. The application class normally derives from the `pvr::Shell` class, and is powered by its callbacks. So in most IDEs, after writing `this->` somewhere in the main application class, autocomplete should give all the information needed to be able to use PVRShell**.**

PVRShell handles everything up to the level of display/window creation. Higher levels - for instance GPU contexts/devices/API calls/surfaces including any and all API objects - are not handled by the Shell. These should be dealt with from the application, normally using PVRUtilsVK / PVRUtilsGles.

In summary, PVRShell:

- abstracts away the platform - display, input, filesystem, window and so on
- contains `main()` or any other platform-specific entry point of the application
- is the ticking clock that provides all the events that structure the application
- provides utilities for reading command line parameters, loading and saving files, displaying the fps and many others
- catches `std::runtime_error` exceptions. All the exceptions used in the Framework derive from `std::runtime_error`. The message is displayed in a platform-specific way, usually a dialog window.

How to use PVRShell

PVRShell uses PVRCore and therefore requires it to be linked in the application.

- Include `PVRShell/PVRShell.h`, in the main app class file

- Create a class that derives from `pvr::Shell` as described in The Skeleton of a Typical Framework 5.x Application to begin using the shell. The library file will be named `PVRShell.lib` or equivalent; `libPVRShell.a` and so on.

1.3.2. PVRVk

About PVRVk

PVRVk is an independent module providing a convenient, advanced, yet still extremely close to the original Vulkan abstraction. It offers a sweet spot combination of simplicity, ease of use, minimal overhead and respect to the specifications.

The main features are:

- C++ classes that wrap the Vulkan objects with their conceptual functionalities
- automatic reference counted smart pointers for all Vulkan objects/object lifecycle management
- Command buffers know what objects await execution into them and keep them alive
- Descriptor sets actually contain references to objects they contain
- Default parameters for all parameter objects and functions, where suited
- Structs initialised to sensible defaults
- Strongly typed enums wrapping Vulkan enums

Developers who have used the Vulkan spec should find it very familiar without any other external references. All developer-facing functionality can be found in the `pvrvk::` namespace.

To make sure PVRVk can be used by as wide an audience as possible, it is completely independent from any other Framework module. This includes PVRCore, so this is the reason there is a little duplication of code between PVRVk and PVRCore, especially error logging.

How to use PVRVk

PVRVk can be used by just following the Vulkan specification and getting a handle on the obvious conventions:

- Enums are type safe (`enum class`) and their members lose the prefix. Instead `e_` is added so that members like `VK_FORMAT_2D` can be defined as `e_2D`)
- Vulkan functions become methods of their first parameter's class. This means that any function that takes a command buffer as its first argument becomes a member function of the `CommandBuffer` class.
- A few other obvious rules such as Resource Acquisition Is Initialisation (RAII) objects. This means release whatever is not wanted any more by null-ing or resetting its handle, or just letting it go out of scope.

PVRVk has no Framework dependencies and uses `vulkan_bindings.h`. The compiled library file is named PVRVk, therefore the files are `PVRVk.lib` and `libPVRVK.a`. The library will need to be linked to be used.

`PVRVk/PVRVk.h` will need to be included in order to make available the symbols required for PVRVk, but PVRUtilsVk will always include the PVRVk headers anyway. Therefore when using PVRUtilsVk there is no need to `#include "PVRVk/PVRVk.h"`

For developers familiar with Vulkan, the sections Using PVRVk and Tips and Tricks may also give some useful Vulkan tips.

PVRUtilsVk uses PVRVk. All of the PowerVR SDK Vulkan examples except for *HelloAPI* (which is completely raw code) are nearly all PVRVk/PVRUtilsVk code.

1.3.3. PVRAssets

PVRAssets is used to work directly with the CPU-side of the authored parts of an application, for example models, meshes, cameras, lights, textures and so on. It is used when dealing with application logic for things like animation and in general scene management.

PVRAssets does not contain any code that is related to the Graphics API and API objects. A mesh defined in `pvr::assets` contains raw vertex data loaded in CPU-side memory. This may be decorated by metadata such as datatypes, meaning (semantics) and all the data needed to create a Vertex Buffer Object (VBO), but not the VBO itself. This area is covered by PVRUtils or the application.

PVRAssets is the recommended way to load and handle a multitude of assets. These include but are not limited to all PowerVR formats like POD (models), PVR (textures, fonts) and PFX (effects). The PVRAssets classes map very well to these, but can easily be used by other formats as well.

This would normally be accomplished by extending the `AssetReader` class for other formats, so it should work well with the rest of the Framework. For instance, an `AssetReader<Texture>` for JPEG, or an `AssetReader<Model>` for Wavefront OBJ would be simple to write.

How to use PVRAssets

PVRAssets requires PVRCore. PVRAssets is required by PVRUtilsVk and PVRUtilsGles. It is necessary to link against the PVRAssets library if using its functionality or PVRUtils(Vk/ES). Include "`PVRAssets/PVRAssets.h`" to include all normally required functionality of PVRAssets.

Start from the `pvr::assets::Model` class to become familiar with PVRAssets.

1.3.4. PVRCore

About PVRCore

PVRCore contains low-level supporting C++ code. This includes, but is not limited to:

- data structures
- code helper functions
- math such as frustum culling and cameras
- string helpers such as Unicode and formatting

PVRCore has several fully realised classes that can be used in their own right, such as the `RefCountedResource` and `Stream`. These can be used on their own if required. Look into the `pvr`, `pvr::strings` and `pvr::maths` namespaces for other useful functionality.

How to use PVRCore

PVRCore should be linked into applications that use PVRShell, PVRAssets or PVRUtils**.** It is required by all modules (except PVRVk) and requires none of the other PVR modules.

PVRCore requires and includes the external library GLM for vectors and matrices, a modified version of `moodyCamel::ConcurrentQueue` for multithreading, and pugixml for reading XML data.

Include `PVRCore/PVRCore.h` to include all common functionality of PVRCore.

1.3.5. PVRUtils

About PVRUtils

PVRUtils is another central part of developer-facing Framework code. Where PVRShell abstracts and provides the platform, PVRUtils provides tools and facilitates working with the rendering API, automating and assisting common initialisation and rendering tasks.

It provides higher level utilities and helpers for tedious tasks such as context creation, vertex configuration based on models, and texture loading. These extend right up to very high level complex areas like the UIRenderer which is a full-fledged 2D renderer itself, threading and access to the hardware camera and so on.

There are two versions available covering Vulkan and OpenGL ES - these are PVRUtilsVk and PVRUtilsGles. They provide a similar but not identical API, as they have several differences in order to optimise better each for their underlying API.

The most typical functionality in PVRUtils (either version) is boilerplate removal. Tasks such as creating contexts, surfaces, queues and devices can be reduced to one line of code. There is also support for creating VBOs from a model, loading textures from disk and much more.

For Vulkan, this reduced the usually hundreds of lines of code to create physical devices, surfaces, devices, queues, and backbuffers to around ten lines. Loading textures or buffers with an allocator becomes a single line each.

For both OpenGL ES and Vulkan, `StructuredBufferView` is very important, as it is a tool to determine the std140 layout of buffers. It makes mapping and setting members more straightforward, without needing to go into complicated packing/padding calculations.

Importantly, it also contains the UIRenderer. This is a very powerful library, which provides the capability of rendering 2D objects in a 3D environment, especially for text and images. For text rendering, font textures can be generated in a few seconds with PVRTexTool. An Arial font is provided and loaded by default.

Several methods of layout and positioning are provided, such as:

- anchoring
- custom matrices
- hierarchical grouping with inherited transformations

Other functionality provided by PVRUtils includes:

- asynchronous operations via a texture loading class that loads textures in the background
- the Vulkan RenderManager. This is a class that can completely automate rendering by using the PowerVR POD and PFX formats for a complete scene description.

How to use PVRUtils

PVRUtilsVk is built on top of PVRVk, and PVRUtilsGles is built on top of raw OpenGL ES 2.0+. Their main header files need to be included with `#include "PVRUtils/PVRUtilsVk.h"` or `#include "PVRUtils/PVRUtilsGles.h"`

Later on in this document is more information on the basic use of UIRenderer.

All the SDK examples use PVRUtils for all kinds of tasks. They use UIRenderer to display titles and logos. The exceptions are *HelloApi* and *IntroducingPVRShell*.

IntroducingUIRenderer and *ExampleUI* are both examples of more complex UIRenderer usage. *Multithreading* showcases the Asynchronous API for Vulkan.

1.3.6. PVRCamera

PVRCamera provides an abstraction for the hardware camera provided by Android and iOS. Currently it is only implemented for OpenGL ES (not Vulkan) as the camera texture is provided as an OpenGL ES texture. In Windows/Linux, a dummy implementation displaying a static image instead of the camera stream is provided to assist development on desktop machines.

Using PVRCamera

Include `PVRCamera/PVRCamera.h`.

The SDK example *IntroducingPVRCamera* shows how to use this module.

1.3.7. Changes from older modules

PVRPlatformGlue

PVRPlatformGlue (EGL/EAGL) context creation functionality has been moved into PVRUtilsGles and called explicitly in application code.

PVRPlatformGlue (Vk) is no longer separate and is covered by PVRVk, with its helper functionality into PVRUtilsVk.

PVRApi

PVRApi (OpenGL ES) has been removed. No OpenGL ES abstraction is provided any more, and the high level functionality moved into PVRUtilsGles as helpers and support classes.

PVRApi (Vulkan) has been reimagined and streamlined into PVRVk, with its high-level functionality extracted into PVRUtilsVk.

PVRUIRenderer

PVRUIRenderer is now a part of PVRUtils, and has been split into two platform specific parts:

- OpenGL ES has been reimplemented with tweaks for more natural use with raw OpenGL ES code, and moved into PVRUtilsGles. The biggest difference is that it executes GL commands inline instead of recording into command buffers.
- Vulkan is largely the same as in versions 4.+

1.4. Platform Independence

All platform specific code is abstracted away from the application. Apart from obvious issues, such as different compilers/toolchains, project files and so on, this also means areas such as the file system and the window/surface itself. This platform independence is largely provided from PVRShell.

1.4.1. Supported platforms

The PowerVR Framework is publicly supported for Windows, Linux, OSX, Android, iOS, and QNX.

1.4.2. Build system

All platforms use CMake for all platforms. Each framework module and example have their own `CMakeLists.txt`, and there is additionally a `CMakeLists.txt` file at the root of the SDK.

For Android, we are using gradle for the java part and putting the apk together, and CMake (the same CMake as all the other platforms) for the native part. CMake is called internally by the Android build system. Each framework module and example has their gradle build scripts in a build-android folder.

We are also providing a number of cross-compilation toolchains. These are found in the `cmake/toolchains` folder of the SDK. These are:

- iOS cross-compilation toolchain
- Linux cross-compilation with gcc (x86_32,x86_64,armv7, armv7hf, armv8, mips32,mips64)
- QNX cross-compilation with qcc (x86_32,x86_64,aarch64le, armlet-v7)

Changing the compiler to something else is achieved by changing/copying the CMake toolchains.

1.4.3. File system (streams)

The file system is abstracted through the PVRCore and PVRShell.

In PVRCore, the abstract class `pvr::Stream` contains several implementations:

- `pvr::FileStream`, provides code for files
- `pvr::AndroidAssetStream`, provides code for Android Assets
- `pvr::WindowsResourceStream` provides code for Windows Resource files
- `pvr::BufferStream` provides code for raw memory

Any Framework functionality requiring raw data will require a `pvr::Stream` object, so that files, raw memory, android assets or windows resources can be used interchangeably.

PVRShell puts everything together. The `pvr::Shell` class provides a `getAssetStream(…)` method which will try all applicable methods to get a `pvr::Stream` to a filename provided. It initially looks for a file with a specified name, and if it fails it will then attempt other platform specific streams such as Android Assets or Windows Resources. Linux by default only supports files, and iOS accesses its bundles as files. It is important to check the returned smart pointer for `NULL` to make sure that a stream was actually created.

1.4.4. Windowing system

The windowing system is abstracted through PVRShell. No access is required, or given to the windowing system, except for generic `OSWindow` and `OSDisplay` objects used for context/surface creation by PVRUtils. Window creation parameters such as window size, full screen mode,

framebuffer formats and similar are accessed by several `setXXXXXXX` PVRShell functions that the developer can call in the initialisation phase of the application.

1.5. Supported APIs

Version 5.0 and beyond of the PowerVR SDK will no longer provide API independence. PVRUtils is now separate libraries which sometimes differ in both API and implementation.

This was not a decision taken lightly. However, it was felt that whilst in the early Vulkan days an experimental cross-API implementation would be very valuable, as Vulkan matured the value became less. Vulkan is intended to be cross-platform, so a cross-API with OpenGL ES is no longer necessary. In the interests of Vulkan SDK flexibility and expressiveness, and the educational value of OpenGL ES SDK, it seemed sensible to separate the APIs so they no longer trip over each other.

Some benefits to Vulkan are immediately obvious:

- Vulkan queues have been unleashed. Any queue configuration allowed by the specification can be produced. In previous versions, queues were internally handled.
- Any synchronisation scheme may be realised. The previous version featured automatic synchronisation that was convenient, but limited.
- Multi surface cases and similar can be done. In the previous versions, surfaces were handled internally, making multi-surface impossible.
- In general it is now much easier to do custom, even unusual solutions without performing unnecessary or complex operations.

The main benefit for OpenGL ES is a return to its educational values.

The obvious change that this causes is that all API objects such as contexts, devices, and surfaces are explicitly created in application code, by the developer. PVRUtils makes this easy, but it is still completely obvious what is happening, and it is simple to follow the code.

Additionally, by incorporating PVRShell command line configuration, initialisation becomes a handful of lines at most. For example, PVRUtilsGles can one-line-create an EGL/EAGL context supporting the highest available OpenGL ES version that is supported at runtime, with a command-line developer-specified resolution and backbuffer format.

1.6. The Skeleton of a Typical Framework 5.x Application

1.6.1. PowerVR SDK examples structure

PowerVR SDK examples follow the following structure:

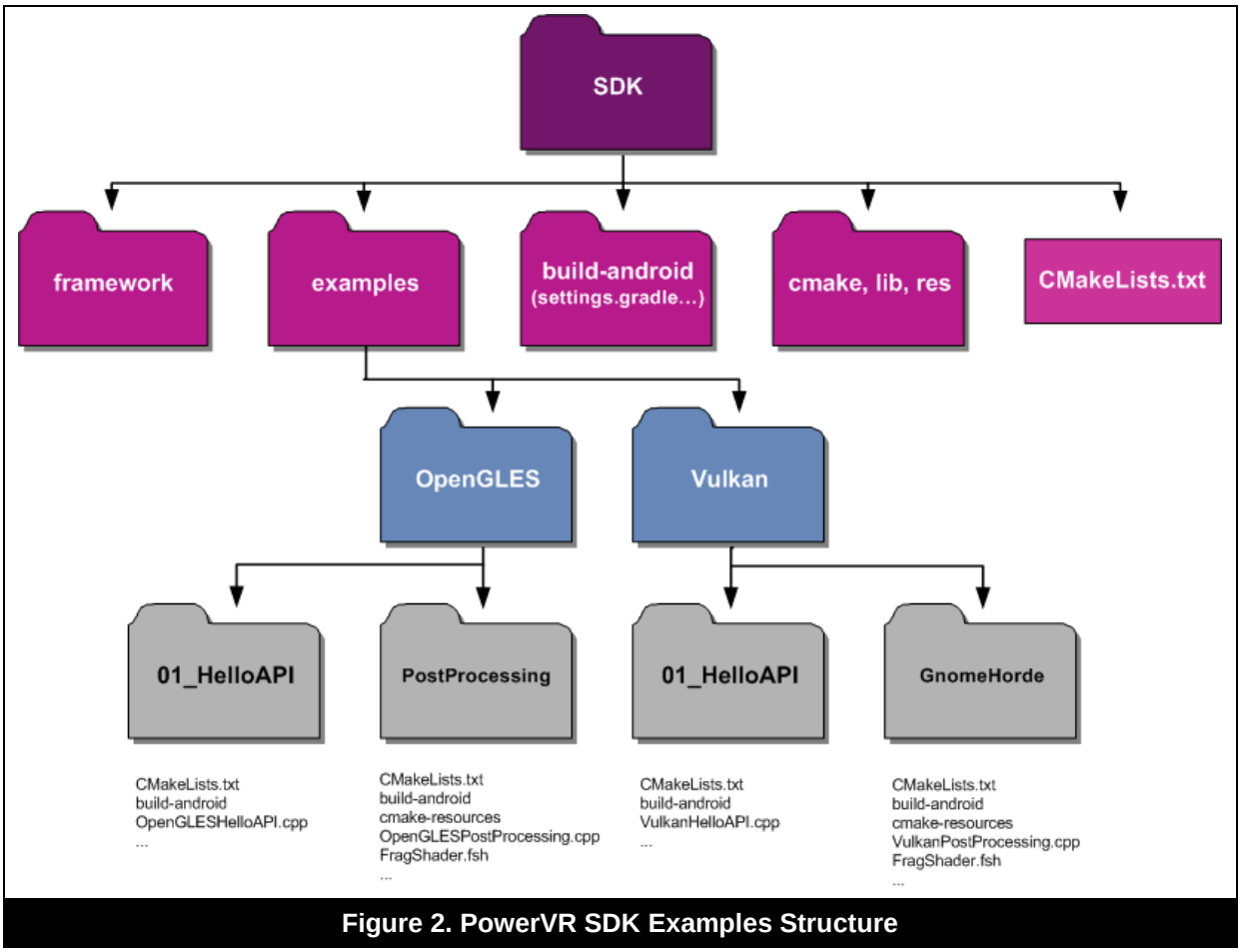

Figure 2. PowerVR SDK Examples Structure

1.6.2. The minimum application skeleton

The easiest way to create a new application is to copy an existing one - for example *IntroducingPVRUtils*.

- Add the sdk folders `framework/` and `include/` as include folders
- Link against PVRShell, PVRCore, PVRAssets, and either PVRUtilsVk (`PVRUtilsVk.lib`) or PVRUtilsGles (`PVRUtilsGles.lib`)
  - If using Vulkan, the application should also link against PVRVk (`PVRVk.lib`)
  - If using OpenGL ES, there is no need to link against OpenGL ES libraries. `DynamicGles.h` takes care of loading the functions at runtime.
- Include `PVRShell.h`, and `PVRUtilsVk.h` or `PVRUtilsGles.h` in the file where the application class is
- Create the application class, inheriting from `pvr::Shell`, implementing the five mandatory callbacks as follows:

```
class MyApp : public pvr::Shell
{
    //...Your class members here...
    pvr::Result::initApplications();
    pvr::Result::initView();
```

```
    pvr::Result::renderFrame();
    pvr::Result::releaseView();
    pvr::Result::quitApplication();
}
```

- Create a free-standing `newDemo()` function implementation with the signature `std::unique_ptr<pvr::Shell> newDemo()` that instantiates the application. The Shell uses this to create the Application. Use default compiler options such as calling conventions for it.

```
std::unique ptr<pvr::Shell> newDemo()
{
    return std::unique_ptr(new MyApp());
}
```

1.6.3. Using PVRVk

PVRVk follows the Vulkan spec, and all operations normally need to be explicitly performed. The calls themselves are considerably shortened due to the constructors and default values, while reference counting again dramatically reduces the bookkeeping code required.

For simplifying the operations and tasks themselves see PVRUtils.

Note that with PVRVk, the optimised, per-device function pointers are always called as devices hold their function pointer tables internally. Function pointers do not need to be retrieved with `VkGetDeviceProcaddress` and similar, as this happens automatically at the construction of the device.

There are obvious usage changes, due to the Object Oriented paradigm followed. In general:

- Vulkan functions whose first parameter is a Vulkan object become member functions of the class of that object. For example, `VkCreateBuffer` becomes a member function of `pvrvk::Device`, so developers will need to use `myDevice->createBuffer()`.
- Functions without a dispatchable object as an input parameter object remain global functions in the `pvrvk::` namespace. For example, `pvrvk::CreateInstance(…)`.
- Simple structs like Offset2D and so on are shadowed in the `pvrvk::` namespace.
- `VkXXXCreateInfo` objects get shadowed by `pvrvk::` equivalents with default parameters and potentially setters. Obvious simplifications/automations are done, for instance `VK_STRUCTURE_TYPE` is never required as it is autopopulated.
- Vulkan Objects are wrapped in C++ reference-counted classes providing them with a proper C++ interface. Usage remains the same and the Vk… prefix is dropped. For example `VkBuffer` becomes `pvrvk::Buffer`.
- All enums are shadowed by C++ scoped enums (enum class `TypeName`)
    - The `VK_ENUM_`*`TYPE_NAME_`* prefix of enum members is dropped and replaced by `e_` In many cases, for example after dropping `VK_ FORMAT_2D`, this would become `pvrvk::Format::2D` which is illegal as it starts with a number.
    - Flags/Bitfields are used like every other enum as bitwise operators are defined for them. `VkCreateBufferFlags` and `VkCreateBufferFlagBits` become just pvrvk::`CreateBufferFlags` and are directly passed to corresponding functions.

1.6.4. Using PVRUtils

PVRUtilsVk

Even using PVRVk, the amount of boilerplate code required by Vulkan for many common tasks can be daunting due to operations complexity.

This is particularly obvious in the simplest tasks that require CPU-GPU transfers, especially loading textures. Even this requires staging buffers and special synchronisation, and in general looks daunting at first.

For even just initialising, the list is as follows:

- creating an instance

- getting device function pointers
- enabling extensions
- creating a surface
- enumerating the physical devices
- selecting a physical device
- querying queue capabilities
- creating a logical device
- querying swap chain capabilities
- deciding a configuration
- creating a swapchain

Each of these steps has a few tens of lines in raw Vulkan, and with quite involved logic.

Initially, the PVRUtilsVk helpers can help automate most, if not all of these tasks, without taking away any of the power. Additionally, if a fringe case cannot be done with the helpers, they only need to be used as required. They don't introduce unnecessary intermediate objects, so mixing and matching as needed is fine.

It is recommended to look at any SDK example, and then if more information is required, follow the utility code. The helpers are usually reasonably simple functions, though in some cases they can get longer the more complex the task.

One of the more important helpers is texture loading, which is automated. The particulars of it can be daunting – for instance:

- determining the exact formats
- array members
- mipmaps
- calculating required memory type and size
- allocating yet more memory for staging buffers
- copying data into it
- kicking transfers to the final texture
- waiting for the results

Or, alternatively, use the texture upload functions, and follow the code to see how it is done.

To learn more about the particulars such as staging buffers, determining formats, read the implementation of the texture upload function or look at the *HelloAPI* example.

Many more such helpers can be found in `pvr::utils`.

Note: During initialisation, some PVRUtilsVk utility functions will use the `DisplayAttributes` class. The Shell auto populates this from defaults, command line arguments and the `setXXXXX()` functions. This is a means of communication from the System to Vulkan. If this functionality is not required, for example if the Shell is not being used, then the `DisplayAttributes` can be populated manually as it is just a raw struct object. This still would take advantage of its sensible defaults. The UIRenderer will be described on its own.

PVRUtilsEs

PVRUtilsEs is similar in use to PVRUtilsVk, but tailored for the OpenGL ES 2.0, 3.x APIs.

The EGL/EAGL context creation is simplified, again with PVRShell command line arguments being automatically used when passed by the developer.

Read any PowerVR SDK OpenGL ES example (except *HelloAPI* or *IntroducingShell*) to see how it is used. The context creation can normally be found in `initView(): createEglContext()`, then `EglContext::init(...)`. Additionally, several helpers, including loading/uploading textures, exist in `pvr::utils`.

1.6.5. Using the UIRenderer

UIRenderer (both the OpenGL ES and Vulkan versions) is a library for laying out and rendering 2D objects in a 2D or 3D scene. The main class of the library is `pvr::ui::UIRenderer`.

The UIRenderer is part of the PVRUtils library.

Initialisation

In Vulkan, UIRenderer refers to and is therefore compatible with a specific `RenderPass`, and UI Rendering commands are packaged and recorded into a `pvrvk::CommandBuffer`.

In OpenGL ES, the OpenGL state is recorded, rendering commands are executed inline, and then the OpenGL ES state is restored.

During initialisation, the rendering surface of the UIRenderer is configured. Note that it is not implied that this surface is the entire canvas, and it will not cull the rendering. It is only a coordinate system transformation from pixels to normalised coordinates and back.

In both OpenGL ES and Vulkan, `beginRendering(...)` is normally used on the UIRenderer objects, either to push and configure the OpenGL state, or to open the Vulkan command buffer. Any sprites required are then rendered, and finally `endRendering(...)` follows to either signify closing command buffers, or pop the GL state.

Sprites

The class that will most commonly be accessed using the UIRenderer is the `pvr::ui::Sprite`. In short, a sprite is an object that can be laid out and rendered using UIRenderer. The sprite is aware of and references a specific `pvr::UIRenderer`. The sprite allows the developer to `render()` it and set things like its colour, rendering mode and so on.

The layout itself and the positioning are not done by the sprite interface class. Instead, the next level of the hierarchy will normally provide methods to lay it out depending on its specific capabilities.

There are two main categories of layout: the 2DComponent and the MatrixComponent.

2D components provide methods to lay the component out in a screen aligned rectangle, provide methods for anchoring to the corners, offsetting by X/Y pixels, rotations and scales and so on.

Matrix components directly take a Matrix for 3D positioning of the component. All predefined primitive sprites (`pvr::ui::Text`, `pvr::ui::Image`) are 2D components.

Complex layouts can be achieved by using a group. Groups are hierarchical containers of other components including other groups. So for example, there could be a 2D group representing a panel with components. Text sprites and an image could be added and positioned to the group, which is then added into a MatrixGroup. Finally, the group could be transformed with a projection matrix to display it in a Star Wars intro scrolling marquee way. This use case is shown in the *IntroducingUIRenderer* example.

An Example of Layout

In order to better understand this, here is a simple example. The developer wishes to print a scrolling marquee, Star Wars-intro like, with some icons at the corners of the marquee, scrolling text, and some text in the corners of the screen. This needs to start from being a single point in the centre of the screen and blow up to take up the entire screen.

The steps would be as follows:

1. Create text for the lines marquee, image for the icons, and put those in a `MatrixGroup`
2. Put this `MatrixGroup` in a `PixelGroup`
3. Set the `anchor` of the marquee texts to `Centre`, and calculate the fixed `PixelOffset` of each text based on line spacing. Each frame, add a number to each text's `PixelOffset.y` to scroll them.
4. `Anchor` the `TopLeft` of the top left symbol to the `TopLeft` of the matrix group. `Anchor` the `BottomRight` of the bottom right symbol to the `BottomRight` of the `MatrixGroup` and so on.
5. Calculate a suitable transformation with a projection to nicely display the marquee relative to its containing group. The marquee and symbols will move as one item here.

6. Anchor the corner text with the same logic, potentially offsetting it so it does not touch the borders.
7. Finally, centre the `PixelGroup`, and set its scale to a very small number. As it increases frame by frame, the group takes up the whole screen when it reaches the value of one.

Preparation (Create Fonts)

If any other font besides the default (Arial TrueType font) is required, use PVRTexTool's Create Font tool to create a `.pvr` file. This is actually a `.pvr` texture file that contains metadata to be used as a font by `UIRenderer`, or the old Print3D used in legacy versions of the SDK.

Usage Example

Here is how to use sprites:

1. Create any sprites that are to be used using `UIRenderer->createFont(…)`, `createImage(…)`, `createText(…)`, `createMatrixGroup(…)` and so on
2. Set up sprites with `setColour`, `setText`
3. Create hierarchies by adding the sprites to groups
4. Lay out sprites/groups using `setPosition()` and so on
5. Call `render()` on the top level sprite. Never call `render()` on sprites that are themselves in containers - only call render on the item containing other items. Otherwise, any sprite on which `render()` is called will be rendered relative to the top level (screen), not taking into account any containers to which it may belong.

For reference:

- `Position` refers to the position of a sprite relative to the component it is added to. For example, the top-left corner of the screen
- `Anchor` refers to the point on the sprite that position is calculated from. For example, the top-left corner of the sprite
- `Rotation` is the angle of the sprite. 0 is horizontal. `Rotation` happens around the `Anchor`
- `Scale` is the sprite's size compared to its natural size. `Scale` happens pinned to the `Anchor`
- `Offset` is a number of pixels to move the final position by, relative to the parent container, and relatively to the finally scaled sprite

To Render the Sprites

If any changes are made to text or matrices or similar, call the `commitUpdates()` function on the sprite on which it will actually be rendered.

There is no need, although it will still execute and cause some overhead, to call `commitUpdates()` on every single sprite. This is only necessary if a sprite needs to be rendered both on its own and inside a container. Otherwise, only call commit updates on the container (the object on which `render()` is called), and the container itself will take care of correctly preparing its children items for rendering.

Note: It is perfectly legal to render a sprite from more than one container. For instance, create the text "Hello" and then render it from five different containers, one spinning, one still, one raw and so on. Then call `commitUpdates()` and `render()` on each of those.

To actually render with UIRenderer:

1. Call `uiRenderer->beginRendering(…)` In Vulkan, this takes a `SecondaryCommandBuffer` where the rendering commands will be recorded.
2. Call the `render()` method on all top level sprites that will be rendered - the containers, not the contents. Again, do not call render on a component that is contained in another component, as the result is not what would be expected. It will be rendered as if it was not a

part of the other component. So if there are two images and a group that contains ten texts and another two images, only three `render()` calls are necessary.

3. Call the `uiRenderer->endRendering()` method.

For OpenGL ES, that is all, the rendering of the objects is complete. The state should be as it was before the `beginRendering()` command, so any state changes between the `beginRendering()` and `endRendering()` commands will be lost.

For Vulkan, The Secondary Command Buffer passed as a parameter in the `beginRendering()` command will now contain the rendering commands such as bind pipelines, buffers, descriptor sets, draw commands, and so on for the UI. It must be submitted inside the renderpass and subpass which were used to initialise the `UIRenderer`, or a compatible renderpass.

Recommendations:

- For Vulkan, reuse the command buffer of the sprite. Even if the text or the image is changed, it is unnecessary to re-record the command buffer unless the actual sprite objects rendered changed. UIRenderer uses indirect drawing commands to make it possible to even change the text without re-recording.
  - For example, if the colour and position of a sprite is changed, the command buffers do **not** need to be re-recorded
  - If the actual text of a `Text` element is changed, the command buffers do **not** need to be re-recorded
  - If new text or remove a text from a container is added, the command buffers **do** need to be re-recorded
- Only call `commitUpdates()` when all changes to a sprite are done. In some cases, especially if text length increases, this operation can become expensive. For example, VBOs may need to be regenerated and so on.

Important:

- If the command buffer is open (`beginRecording()` has been called) when `uiRenderer->beginRendering()` is called, the commands will be appended. In the end, the command buffer **will not** be closed when `endRendering()` is called.
- On the other hand, if the command buffer is closed (`beginRecording()` has **not** been called) when `uiRenderer->beginRendering()` is called, the command buffer will be reset and opened. Then, when finished rendering with `uiRenderer->endRendering()`, the command buffer **will** be closed, `endRecording()` will be called on it.

1.6.6. A simple application using PVRVk/PVRUtilsVk/PVRUtilsEs

initApplication (VK/ES)

The `initApplication` function will always be called once, and only once, before `initView`, and before any kind of window/surface/API initialisation.

Do any non-API specific application initialisation code, such as:

- Loading objects that will be persistent throughout the application, but do not create API objects for them. For example, models may be loaded from file here, or PVRAssets may be used freely here.
- Do not use PVRVk**/**PVRUtilsVk**/**PVRUtilsES at all yet. The underlying window has not yet been initialised/created so most functions would fail or crash, and the shell variables used for context creation (like `OSDisplay` and `OSWindow`) are not yet initialised.
- Most importantly, if any application settings need to be changed from their defaults, they must be defined here. These are settings such as window size, window surface format, specific API versions, vsync or other application customisations. The `setXXXXX()` shell functions give access to exactly this kind of customisation. Many of those settings may potentially be read

from the command line as well. Keep in mind that setting them manually will override the corresponding command line arguments.

initView(Generic)

`initView` will be called once when the window has been created and initialised. If the application loses the window/API/surface or enters a restart loop, `initView` will be called again after `releaseView`.

In `initView`, the window has been created. The convention is to initialise the API here. All PowerVR SDK examples use PVRUtils to create the context (if OpenGL ES) or instance, device surface and swapchain here (if Vulkan).

initView(OpenGL ES)

Create the EGL/EAGL Context here using `pvr::eglCreateContext`. It needs to be initialised with the display and window handles returned by the `getDisplay()` and `getWindow()` functions of the Shell, as well as some further parameters.

After that, it is ready to go. Create any OpenGL ES Shader Programs and other OpenGL Objects that are required. Set up any default OpenGL ES states that would be persistent throughout the program or any other OpenGL ES initialisation that may be needed.

Remember to use the `pvr::utils` namespace to simplify/automate tasks like texture uploading. If needed, jump into the functions to see their implementations. Not all are simple, but will help to point developers in the right direction.

initView (Vulkan)

Create/get the basic Vulkan objects here - instance, physical device, device, surface, swapchain, and Depth buffer. Unless doing a specific exercise, use the PVRUtilsVk helpers, otherwise the two to three lines of code can explode well into the hundreds.

After initialising the API, this place can be used for one-shot initialisation of other API-specific code. In simple applications, this might be all actual objects used. In more complex applications with streaming assets and so on, this may be the resource managers and similar classes.

Usually, the `initView` in our demos is structured in Vulkan as follows:

Initial Setup

1. Create the instance, surface, swapchain, device and queues with our helpers. These can be created with various levels of detail with different helpers, but usually use `pvr::utils::createInstance(…)`, `pvr::utils::createSurface(…)`, `pvr::utils::createLogicalDeviceAndQueues(…)` and `pvr::utils::createSwapchainAndDepthStencilImageView(…)`
2. Create `DescriptorSetLayout` objects depending on the app: `pvrvk::LogicalDevice::createDescriptorSetLayout(…)`
3. Create `DescriptorSet` objects based on the layouts: `pvrvk::LogicalDevice::createDescriptorSet(…)`
4. Create `PipelineLayout` objects using the descriptor layouts: `pvrvk::LogicalDevice::createPipelineLayout(…)`
5. Configure `PipelineCreateInfo` objects (amongst others using those shader strings) and create the pipelines
6. Configure the `VertexAttributes` and `VertexBindings` usually using helper utilities such as `pvr::utils::CreateInputAssemblyFromXXXXX(…)` This is in order to automatically populate the `VertexInput` area of the pipeline based on our models
7. Create the pipelines: `myLogicalDevice->createPipeline(…)`

Textures and Buffers

1. Create a Vulkan Memory Allocator: `pvr::utils::vma::createAllocator(…)`.Use it for all memory objects by passing it as a parameter to all the functions that support it
2. Create the memory objects such as buffers, images and samplers

3. Get streams to the texture data on disk: `getAssetStream(…)`
4. To access CPU-side texture data, load the images into `pvr::assets::Texture` objects using `pvr::assets::textureLoad()`, then upload them to the GPU as `pvrvk::Image` and `pvrvk::ImageView` using `pvr::utils::uploadImageAndView`
   Otherwise, merge the two steps using `pvr::utils::loadAndUploadImageAndView`.
5. For a proper method to do this asynchronously in a multithreaded environment, see the *Multithreading* example
6. Use the shortcut utilities `pvr:: createBuffer()` for creating UBOs or SSBOs.
7. To automatically layout buffers that will have a shader representation (UBOs or SSBOs), we use `StructuredBufferView`. This is an incredibly useful class. The UBO/SSBO configuration needs to be described, and it will then automatically calculate all sizes and offsets based on STD140 rules. This includes array members, nested structs and so on, automatically allowing the developer to both determine size, and set individual elements or block values. Connect it to the actual buffer with `StructuredBufferView-> pointToMappedMemory()`.

Objects

1. Update the descriptor sets with the actual objects. This might sometimes need to be done in `renderframe` for streaming resources.
2. Create command buffers, synchronisation objects and other app-specific objects. Normally one is needed per backbuffer image. Use `getSwapChainLength()` and `logicalDevice -> createCommandBufferOnDefaultPool()` for this.
   In a multithreaded environment at least one command pool per thread should be used. Use `context -> createCommandPool()` and then `commandPool -> allocateCommandBuffer()` on that thread.
   Do not use a command pool object from multiple threads, create one per thread.
   `pvrvk::CommandBuffer` objects track their command pools and are automatically reclaimed. One caveat here is that normally these should be released on the thread their pool belongs to, or externally synchronise their release with their pool access.
3. Use a loop to fill them up as follows: (very simple case)

- For each `swapChainImage`, for the CommandBuffer that corresponds to that swap image: (get the index with `getSwapchainIndex()`)
  - `beginRecording()`
  - `beginRenderPass()` – pass the FBO that wraps the backbuffer image corresponding to this index to the index of this command buffer
  - For each material/object type:
    - `bindPipeline()` - pass the pipeline object
    - `bindDescriptorSets()` – for any per-material descriptor sets such as textures
    - For each object:
      - `bindDescriptorSets()` – for any per-object descriptor sets, for example worldMatrix
      - `bindVertexBuffer()` - pass the VBO
      - `bindIndexBuffer()` - pass the IBO
      - `draw`*`XXXXXX`*`()` - for instance `draw()`, `drawIndexed()`
  - `endRenderPass()`
  - `endRecording()`

renderFrame

This function gets executed by the shell once for each frame. This is where any logic updates will happen. In the most common, recommended scenarios, this will end up being updated values of uniforms such as transformation matrices, updated animation bones and so on.

renderFrame (ES)

In OpenGL ES, it behaves as expected. Run app logic and OpenGL ES commands as with any application with a main, per-frame loop. Remember to call `eglContext->swapBuffers()` when rendering is finished, or swap the buffers manually if not using PVRUtilsGles.

renderFrame (Vulkan)

It is normal to sometimes generate command buffers in `renderFrame`. However, it should be considered whether it is better to offload as much of this work as possible to either `initView` so it is only done once, or to separate threads. It is very much desirable to generate CommandBuffers in other threads - for example as objects move in and out of view. See the *GnomeHorde* example for this.

In all cases, it is highly recommended for Command buffers to be submitted here, in the main thread. Otherwise, the synchronisation can quickly get unmanageable. There is nothing to be gained by offloading submissions to other threads, unless they are submissions to different queues than the one used for on screen display.

Remember to submit the correct command buffer that corresponds to the current swap chain image using `this->getSwapChainIndex()`.

Usually, `renderFrame` will be home to rather complex synchronisation - this is expected and normal with Vulkan development. See the SDK examples for the typical recommended basic synchronisation scheme.

If application logic determines it is time to signal an exit, return `pvr::Result::ExitRenderFrame` instead of `pvr::Result::Success`. This is virtually the same as calling `exitShell()`.

releaseView

`releaseView` will be called once when the window is going to be torn down, but before this actually happens. For example, if the application is about to lose the window/API/surface or enters a restart loop, `releaseView` will be called before `initView` is called again.

For OpenGL ES just follow normal OpenGL ES rules for deleting objects.

For PVRVk, leverage RAII: Release any API objects specifically created, by releasing any references that are held. Release can be achieved by calling `reset()` on the smart pointer itself, or by deleting the enclosing object and so on.

Do not attempt to `release` the underlying object. If such an API exists, it will probably mean something else and behave as expected. Access the `reset()` function with dot `'.'`(not arrow `'->'`) operator on a PVRVk object. Releasing should only be attempted on the references that the developer is holding. There is never a need to try and specifically delete an API object. All objects are deleted when no references to them are held.

When cleaning up, it is recommended to use `device->waitIdle()` before deleting objects that might still be executing. As this takes place during teardown, objects are being released, and there is no performance worry.

Note that RAII deletion of PVRVk and other framework objects is deterministic. Objects are always deleted exactly when the last reference to them is released. Also, note that objects may hold references to other objects that they require, and these object chains are also deleted when the last reference is released. For example, command buffers hold references to their corresponding pools, descriptor sets hold references to any objects they contain. This is one of the most important features of PVRVk.

It does not impose any particular required deletion order, so the developer does not need to be concerned. The developer should keep any objects needed to reference, and release them when no longer required.

In general, it is recommended that C++ destructors are used. This is the way it is done in the Framework examples.

A struct/class is kept that contains all the PVRVk objects. It is allocated in `initView` and deleted in `releaseView`. Ideally, a `std::unique_ptr` is used. Then as objects are deleted (unless a circular dependency was created) then the dependency graph unfolds itself as it has to, and destructors are called in the correct order.

quitApplication

`quitApplication` will be called once, before the application exits. In reality, as the application is about to exit, little needs to be done here except release non-automatic objects, such as file handles potentially held, database connections and similar. Some still consider it best practice to tear down any leftover resources held here, even if they will normally automatically be freed by the operating system.

1.6.7. Rendering without PVRUtils

The Framework is modular, and libraries can commonly be used separately from the others. There are a lot of options that can be used:

- Everything – obviously, that's our official recommendation. Derive the application from `pvr::Shell`, use PVRUtils and PVRVk for rendering, load and use the assets with PVRAssets and use the PVRUtils for rendering 2D elements and multithreading. Most PowerVR SDK examples use this approach, leveraging all the power of the PowerVR Framework.
- Forgo using the top-level libraries (PVRUtils, PVRCamera) because a different solution is needed, or the functionality is not required. Be warned that the boilerplate may become unbearable while the overhead is minimal.
- Completely raw Vulkan, without even PVRVk. This is not recommended, it is better to use PVRVk or another Vulkan library to help. Check out the Vulkan *IntroducingPVRShell* or even *HelloAPI* examples to get a taste of just how much code is needed to get even a triangle on screen. Then check out any other example to see how much lifetime management, sensible defaults and in general a modern language can help. These gains are practically for free.
- Forgo using even PVRAssets, in which case, the only thing that is being used from the framework is PVRShell. In this scenario, code will need to be written for everything except the platform abstraction. Loading assets will now become non-trivial, and support classes will need to be written for every single bit of functionality. This is done in *IntroducingPVRShell*.
- Less than that, it's not using the Framework at all.

1.7. Synchronisation in PVRVk (and Vulkan in general)

The Vulkan API, and therefore PVRVk, has a detailed synchronisation scheme.

There are three important synchronisation objects: the semaphore, the fence and the event.

- The **semaphore** is responsible for coarse-grained syncing of GPU operations, usually between queue submissions and/or presentation
- The **fence** is required to wait on GPU events on the CPU such as submissions, presentation image acquisitions, and some other cases
- The **event** is used for fine-grained control of the GPU from either the CPU or the GPU. It can also be used as part of layout transitions and dependencies

1.7.1. Semaphores

The semaphores impose order in queue submissions or queue presentations.

A command buffer imposes some order on submissions. When a command buffer is submitted to a queue, the Vulkan API allows considerable freedom to determine when the command buffer's commands will actually be executed. This is either in relation to other command buffers submitted before or after it, or compared to other command buffers submitted together in the same batch (queue submission). The typical way to order these submissions is with semaphores.

Without using semaphores, the only guarantee imposed by Vulkan is that when two command buffer submissions happen, the commands of the second submission will not finish executing before the commands in the first submission have begun. This is obviously not the strongest guarantee in the world.

The basic use of a semaphore is as follows:

- Create a semaphore
- Add it to the `SignalSemaphores` list of the command buffer submission that needs to execute first
- Add it in the `WaitSemaphores` list of the command buffer that is to be executed second
- Set the Source Mask as the operations of the first command buffer that must be completed before the Destination Mask operations of the second command buffer begin. The more precise these are, the more overlap is allowed. Avoid blanket wait-for-everything statements, as this can impose considerable overhead by starving the GPU.

For example by adding a semaphore with `FragmentShader` as the source and `GeometryShader` as the destination, this ensures that the `FragmentShader` of the first will have finished before the `GeometryShader` of the second begins. This implies that, for example, the `VertexShaders` might be executed simultaneously, or even in reverse order.

This needs to be done whenever there are multiple command buffer submissions. In any case, it is usually recommended to do one big submission per frame whenever practical. The same semaphore can be used on different sides of the same command buffer submission. For example, in the wait list of one command buffer and the signal list of another, in the same queue submission. This is quite useful for reducing the number of queue submissions.

1.7.2. Fences

Fences are simple: insert a fence on a supported operation whenever it is necessary to know (wait) on the CPU side whenever the operation is done. These operations are usually a command buffer submission, or acquiring the next backbuffer image.

This means that if a fence is waited on, any GPU commands that are submitted with the fence, and any commands dependent on them, are guaranteed to be over and done with.

1.7.3. Events

Events can be used for fine-grained control, where a specific point in time during a command buffer execution needs to wait for either a CPU or a GPU side event. This can allow very precise threading of CPU and GPU side operations, but can become really complicated fast.

It is important and powerful to remember that events can be waited on in a command buffer with the `waitEvents()` function. They can be signalled both by the CPU by calling `event->signal()`, or when a specific command buffer point is reached by calling `commandBuffer->signalEvent()`. Execution of a specific point in a command buffer can be controlled either from the CPU side, or from the GPU with another command buffer submission.

1.7.4. Recommended typical synchronisation for presenting

Even the simplest case of synchronisation such as acquire-render-present =(next)=> acquire-render-present =(next)=>… needs quite a complicated synchronisation scheme in order to ensure correctness.

This scheme is used in every PowerVR SDK example. For an *n*-buffered scenario where there are *n* presentation images, with corresponding command buffers, there will be:

- One sets of fences (one if the developer is prepared to allow command buffer simultaneous execution - not recommended)
- One set of semaphores connecting Acquire to Submit
- One set of semaphores connecting Submit to Present
- Tracking a linear progression of frames (frameId) and the swapchain image id separately.

See most PowerVR SDK examples for the implementation.

Any additional synchronisation can be inserted inside the Submit phase without complicating things too much.

1.8. Overview of Useful Namespaces

Table 1. Useful Namespaces

Namespace	Description
Namespace `::pvr`	Main namespace. Primitive types and foundation objects, such as the `RefCountedResource` and some interfaces are found here.
Namespace `::pvrvk`	All public classes of the PVRVk library are found here. This is where to find any API objects that need to be created such as buffers, (GPU) textures, CommandBuffers, GraphicsPipelines, RenderPasses and so on.
Namespace `::pvr::utils`	This extremely important namespace is the location for automations for common complex tasks. For example, matching a Model (from a `POD` file) with an Effect (from a `PFX` file) to set up a pipeline and VBOs to render with. Creating a Vulkan context.
Namespace `::pvr::assets`	All classes of the PVRAssets library are found here: Model, Mesh, Camera, Light, Texture, AssetLoader and so on.
Namespace `::gl`	OpenGL ES bindings. Only OpenGL ES function pointers are found here.
Namespace `::vk`	Vulkan bindings. Only Vulkan function pointers are found here.
Namespaces `::?::details` `::?::impl`	Namespaces for code organisation. Not required by the developer.

1.9. Debugging PowerVR Framework Applications

The first step in debugging a PowerVR Framework application should always be to examine the log output. There are assertions, warnings and error logs that should help find many common issues.

1.9.1. Exceptions

All errors that are caught by the application will raise an exception. All exceptions inherit from a common base class, itself inheriting from `std::runtime_error`.

On a debug build, if a debugger is present and supported, for instance, if executing the application from an IDE, all PowerVR exceptions will cause a debugger break (similar to a breakpoint) immediately in their constructor. This allows the developer to immediately examine the situation and call stack where the exception was thrown.

In any case, if a `std::runtime_exception` is not caught anywhere else, it will be caught by PVRShell, and the application will quit. Its message (`what()`) will be displayed as a pop up with windowing systems or a logged message with command line systems.

Additionally, API specific tools should be used to help identify a bug or other problem.

1.9.2. OpenGL ES

OpenGL ES applications are usually debugged as a combination of CPU debugging and PVRTrace or other tools in order to trace and play back commands.

1.9.3. Vulkan/PVRVk

The Vulkan implementation of the PowerVR Framework is thin and reasonably simple. Apart from CPU-side considerations like lifetime, API level debugging should be very similar to completely raw Vulkan. However, the complexity of the Vulkan API makes debugging not very easy in general.

The most important consideration here are the Vulkan Layers.

Layers

Vulkan Layers sit between the application and the Vulkan implementation, performing all kinds of validation. They can be enabled globally, for example in the registry, or locally, for example in application code. PVRUtilsVk automatically enables the standard validation ones for debug builds. It is extremely important not to start debugging without these, their information is invaluable.

Check the LunarG Vulkan SDK for the default layers; they are valuable in tracking many kinds of wrong API use. It is a very good practice to:

1. Inspect the log for any errors. This includes layers output on platforms that have them.
2. Inspect the Vulkan layers and fix any issues found, until they are clear of errors and warnings.
3. Continue with other methods of debugging such as the API dump layer, CPU or GPU debuggers, Trace or any other method suitable to the error.

2. Tips and Tricks

2.1. Frequently Asked Questions

2.1.1. Which header files should I include?

For a typical application, add `[sdkroot]/Framework` as an include folder, then:

For Vulkan:

```
#include "PVRShell/PVRShell.h"
#include "PVRUtils/PVRUtilsVk.h" //Includes everything, including PVRVk
```

Or for OpenGL ES:

```
#include "PVRShell/PVRShell.h"
#include "PVRUtils/PVRUtilsGles.h"
```

If PVRCamera is required, additionally:

```
#include PVRCamera/PVRCamera.h
```

2.1.2. Which libraries should be linked against?

Usually, the framework needed should be added through CMake, using `add_subdirectory`. To link to pre-built binaries, build the framework libraries with CMake and add the library outputs from wherever they were built. The libraries themselves are:

- `[lib]PVRCore.[ext]` e.g. `PVRCore.lib, libPVRCore.a`
- `[lib]PVRShell.[ext]` e.g. `PVRShell.lib, libPVRShell.a`
- `[lib]PVRAssets.[ext]` e.g. `PVRAssets.lib, PVRAssets.a`
- (Vulkan) `[lib]PVRVk.[ext]` e.g. `PVRVk.lib, libPVRVk.a`
- `[lib]PVRUtils[API].[ext]` e.g. `PVRUtilsGles.lib, libPVRUtilsVk.a`

As noted, the dependencies are:

- PVRCore: None
- PVRShell: PVRCore
- PVRAssets: PVRCore
- PVRVk: None
- PVRUtilsVk: PVRCore, PVRShell, PVRAssets. PVRVk
- PVRUtilsGles: PVRCore, PVRShell, PVRAssets

For Android, use the `settings.gradle` file to define any required Framework projects `build-android` folders as dependencies of the application. This is in addition to CMake.

If the PVRCamera module is required, build and include in the project the PVRCamera library, which is platform specific, not just native. See the *IntroducingPVRCamera* example for more information.

2.1.3. Does library link order matter?

For Windows/OSX/iOS, it does not matter. For Android and Linux, it does because it matters for some possible underlying compilers.

Make sure that for Linux and Android, link order is in reversed order of dependencies: dependents (high level) first, to dependencies (low level) last. So the order should be:

1. PVRUtils, PVRCamera
2. PVRShell**,** PVRAssets
3. PVRCore, PVRVk
4. System libraries (usually: m, thread for linux, android_native_app_glue for Android)

If there are undefined references to functions that appear to be present, apart from needing a library that is not included, this is a common culprit.

2.1.4. Are there any dependencies to be aware of?

In general we recommend using one of our CMakeLists.txt and/or gradle scripts as a base. To start from scratch:

- The `[SDKROOT]/include` folder must be added as an include file search path. It contains the API header files and any other headers that are used. As well as the stock Khronos headers for most APIs, it contains PowerVR SDK's own custom `DynamicGles.h`, `DynamicEgl.h`, and `vulkan_wrapper.h` bindings.
- The `[SDKROOT]/framework` folder must be added as an include file search path to access the framework's headers.
- The `[SDKROOT]/external` folder contains external libraries used by the framework, such as glm, concurrentqueue and pugixml that are all used by the Framework.
- The `[SDKROOT]/lib [PLATFORM…]` folder may contain library dependencies of the project files. For example, the PVRScope libraries are located there.
- The Framework library files will be either wherever they were built, or by default, prebuilt would be in `[SDKROOT]/framework/bin/[PLATFORM…]` . Since the PowerVR libraries do NOT have a C interface (they expose C++ classes in their API), it is strongly recommended to never use prebuilts, but to always build them through CMake with common compilation options with the application.

What about linking against OpenGL ES, or Vulkan, respectively?

It is not necessary to link against them, both are loaded with dynamic library loading (except for iOS).

DynamicGLES takes care of it by dynamically linking OpenGL ES.

PVRVk loads Vulkan function pointers the optimal way, using per-device function pointers. These are, stored into per-instance and per-device function pointer table. Static linking is unnecessary.

2.1.5. What are the strategies for Command Buffers? What about Threading?

The Vulkan multithreading model in general means that the developer is free to generate command buffers in any thread, but they should be submitted in the main thread. While it may be possible to do differently, this is normally both the optimal and the desired way, so we do not concern ourselves with cases of submissions for multiple threads.

On the other hand, command buffers can and should, if possible, be generated in other threads. See the *GnomeHorde* example for a complete start-to-finish implementation of this scenario.

Apart from that, there are numerous ways that an application is structured, but some patterns will be emerging at times.

Single command buffer submission, multiple command secondary command buffers

This strategy is a very good starting point and general case. Work is mostly generated in the form of secondary command buffers, and these secondary command buffers are gathered and recorded into a single primary command buffer, which is then submitted. Almost all examples in the PowerVR SDK use this strategy.

Multiple parallel command buffers, submitted once

This strategy means creating several command buffers, and submitting them together once. A little additional synchronisation might be needed with the acquire and the presentation engine, but there could be cases where some small gain is realised. However, it is much less common for rendering than it would be immediately apparent, as a RenderPass cannot be split to multiple submissions. This means that it is mostly operations on different render targets, especially from different frames, and Compute operations on the same queue that can be split into different command buffers.

In this scenario, all those command buffers would be independent and could be scheduled to start after the presentation engine has prepared the rendering image (backbuffer).The presentation engine would then wait for these to finish before it presents the image. This scenario is applicable if no interdependencies exist between the command buffers, or if the developer synchronises them with semaphores or events. For example, it is possible to render different objects to different targets from different command buffers. It is much more complicated to stream computed data with this strategy.

Multiple parallel command buffers, submitted multiple times

When there is no reason to do otherwise, submitting once is fine. Sometimes it is better to completely separate different command buffers into different submissions, especially if utilising different queues or even queue families and sometimes activating different hardware. In general, the developer must devise their own synchronisation scheme in this case, but usually this will be connected to the basic case described above.

2.1.6. How are PVRVk objects created?

Usually, most PVRVk objects with a Vulkan equivalent (buffers, textures, semaphores, descriptor sets, command pools and so on) are created from the device by calling a `createXXXX()` function. This completely shadows the Vulkan API, so look for the corresponding `create` function in the members of the class of the first parameter of the Vulkan create function.

Whenever possible, we have provided defaults for as many of the parameters/create info fields as is feasible.

Remember that some creations are really allocations from pool objects:

There is no `device -> createCommandBuffer()`, instead developers must call `commandPool -> allocateCommandBuffer()`. These are usually hinted by the name: instead of create/destroy for objects created on a device, we have allocate/free for objects allocated on a pool.

2.1.7. How are PVRVk Objects cleaned up? Is there anything that needs to be destroyed that the developer did not create?

Discard (exit the scope) or reset any smart pointers to objects not needed, when they are finished with. If manually resetting, do this, at the latest, in the `releaseView()` function. API objects do not have to be explicitly destroyed, only their smart pointers `reset`, as they are immediately destroyed when their reference count goes to zero.

Other objects may sometimes hold references to them. Most notably, `CommandBuffers` and `DescriptorSets` hold references to objects they are using.

2.1.8. How is a UIRenderer cleaned up?

The UIRenderer has an explicit `release()` function that releases all resources held by it. Remember to do this before destroying the device.

See the following question if a specific order of destruction is needed.

2.1.9. Do any API objects need to be manually kept alive?

Mostly, they do not.

- `CommandBuffer` objects and `DescriptorSet` objects will keep references to any objects they contain (submitted/updated into them respectively) until reset is called.
- Any nested objects will keep alive underlying objects. For example, `TextureView` objects will keep alive the `TextureStore` objects underneath, `BufferView` objects will keep alive `Buffer` objects, `Pipelines` will keep their `PipelineLayouts` alive, and so on.

There are some notable exceptions:

- `CommandPool` and `DescriptorPool` objects. These objects are not kept alive by their `CommandBuffer` and `DescriptorSet` objects. This means that the developer must keep all of them alive until no longer required. This means that Command/Descriptor pools must be destroyed after any of their objects are released.
- `CommandBuffer` objects must be kept alive as long as they are executing (rendering) which generally means when the corresponding `SwapBuffer` image is released, unless a fence is specifically waited on. This rule, in conjunction with the previous one, means that destruction must happen in the order: Frame done -> commands released -> pools released.

2.1.10. How are files/assets/resources loaded?

The PowerVR Framework uses the stream abstraction for data. There are two ways to use those:

- Directly create a `FileStream`/ `BufferStream`/ `WindowsResourceStream` to load a resource. The resource must be of the correct type, and can be platform specific. For example, a `FileStream` will work fine for Windows and Linux, but not for Android. Additionally, this method is used in order to use a stream pointing to raw memory using a `BufferStream`.
- Use `pvr::Shell::getAssetStream(…name…)` This function will look for any applicable methods depending on the platform, and attempt to create a stream with that method, until it succeeds or run out of methods. The priority from highest to lowest is:
  - `FileStream`
  - `AndroidAssetStream`
  - `WindowsResourceStream`

  The parameter is usually a relative, but can be an absolute, path, and assumes that Windows Resource names will be this path. Files will be searched for both in the current folder of the executable, and in the Assets subfolder. Functions that need some kind of data to create an object, most notably, Asset load functions, will take streams as input. The only exception is PVRVk Shaders - these are passed as raw bytes to avoid a PVRVk dependency on PVRCore.

2.1.11. How are buffers updated?

At the API level, buffers can be updated with two strategies: map/unmap for CPU-synchronous mapping, and `updateBuffer` for GPU-synchronous mapping.

Update

Update copies over the data supplied by the developer, and only transfers it into the actual buffer when the command is actually executed. In other words, the command buffer submitted, and the relevant point in the command stream reached.

Map

OpenGL ES and Vulkan behave very differently when mapping.

For OpenGL ES, mapping is similar to update, and acts as if the map command happened just after the previous command and just before the following command.

However, the developer may forego this behaviour, its guarantees and the data copies they force the driver to do, by using `GL_MAP_UNSYNCHRONIZED_BIT`. This makes the changes the developer does to the data immediately visible to the API. Of course, it also forces the developer to have their own synchronisation scheme - for instance multi-buffering, synchronised slices and others.

Vulkan by default and exclusively uses this strategy. No synchronisation is attempted, so the developer must use multi-buffering, fences, events or other synchronisation strategies to ensure everything is working as intended.

Calculating buffer layouts

When a UBO or SSBO interface block is defined in the shader, and the developer needs to fill it with data, the developer must religiously follow the STD140 (or STD430) GLSL rules to determine the actual memory layout, bit for bit, including paddings. Then the developer must translate that into a C++ layout or manually `memcpy` every bit of it into the mapped block.

This can become extremely tedious, especially when considering potential inner structs or other similar complications. Fortunately PVRUtils is able to help with this.

The StructuredBufferView

This class takes a tree-structure definition of entries, automatically calculates their offsets based on std140 rules (an std430 version is planned), and allows utilities to directly set values into mapped pointers.

The ease that this provides cannot be overstated – normally a developer would have to go through all the std140 ruleset and determine the offset manually for every case of setting a value into a buffer.

This is a code example from the *Skinning* SDK example:

GLSL

```
struct Bone {
   highp mat4 boneMatrix;
   highp mat3 boneMatrixIT;
   }; // SIZE: 4x16 + 3x16(!) = 112. Alignment: Must align to 16 bytes

 layout (std140, binding = 0) buffer BoneBlock {
 mediump int BoneCount; // OFFSET 0, size 4
 Bone bones[]; // starts at 16, then 112 bytes each element
 };
```

CPU side

We wanted to provide an easy to use interface for defining the `StructuredBufferView`. Using C++ initialiser lists, we have created a compact JSON-like constructor that allows the developer to easily express any structure.

The following code fragment shows the corresponding CPU-side code for the GLSL above:

```
// LAYOUT OF THE BUFFERVIEW
pvr::utils::StructuredMemoryDescription descBones("Ssbo", 1, // 1: The UBO itself is not array
{
      { "BoneCount", pvr::GpuDatatypes::Integer } // One integer element, name "BoneCount"
      { // One element, name "Bones", that contains...
            "Bones", 1,
            { // One mat4x4 and one mat3x3
                  {"BoneMatrix", pvr::GpuDatatypes::mat4x4},
                  {"BoneMatrixIT", pvr::GpuDatatypes::mat3x3}
            }
      }
});

// CREATING THE BUFFERVIEW
pvr::utils::StructuredBufferView ssboView;
ssboView.init(descBones); // One-shot initialisation to avoid mistakes.

// SETTING VALUES
void* bones = gl::MapBufferRange(GL_SHADER_STORAGE_BUFFER,
                                 0,
                                 ssboView.getSize(),
                                 GL MAP WRITE BIT);
int32 t boneCount = mesh.getNumBones();
ssboView.getElement(_boneCountIdx).setValue(bones, &boneCount);
auto root = ssboView.getBufferArrayBlock(0);

for (uint32 t boneId = 0; boneId < numBones; ++boneId)
{
      const auto& bone =  scene->getBoneWorldMatrix(nodeId, mesh.getBatchBone(batch, boneId));
      auto bonesArrayRoot = root.getElement(_bonesIdx, boneId);
      bonesArrayRoot.getElement(_boneMatrixIdx).setValue(bones,
                                                         glm::value ptr(bone));
      bonesArrayRoot.getElement( boneMatrixItIdx).setValue(bones,
                                                 glm::value ptr(glm::inverseTranspose(bone))));
}

gl::UnmapBuffer(GL_SHADER_STORAGE_BUFFER);
```

It is highly recommended to give the `StructuredBufferView` a try even if there is no intention to use the rest of the Framework.

2.2. Models and Effects, POD & PFX

The PVRAssets library contains very detailed, carefully crafted classes to allow handling of all kinds of assets.

2.2.1. Models, meshes, cameras, and similar

The top-level class for models is the `pvr::assets::Model` class. The model contains an entire description of a scene, including a number of:

- Meshes
- Cameras
- Lights
- Materials
- Animations
- Nodes

In general, these objects are found both in raw lists, and bound to nodes. The Node contains a reference to an item in the list of meshes that is stored in the model. The lists describe the objects that are present. Call `model->getMesh(meshIndex)` to get the list.

Nodes

Nodes are the building blocks of the scene, and describe the hierarchy of the scene. Each node is part of a tree structure, with parent nodes, and carries a transformation, and a reference to an object such as a mesh, camera or light. The transformations are applied hierarchically. The transformations,

in general, are animated and dependent on the current frame of the scene. Static scenes only have one animation.

Nodes are accessed through their indices. In order to make accessing objects easier, the nodes are sorted by object types, in the order Mesh, Camera, Light. Therefore mesh nodes have the indexes from `0` to `model->getNumMeshNodes()-1`. Light nodes have the indexes from `model -> getNumMeshNodes()` to `model -> getNumCameraNodes()-1` and so on.

Always be wary when trying to access a node or its underlying object. When trying to iterate the meshes (for example, to get VBOs, attributes, textures…), always call `getMesh(…)`. When trying to display the scene, iterate `MeshNodes()`. A Mesh is a description of a mesh, not an object in the scene; an instance of an object is a Mesh Node.

Some useful methods

- `getMesh(meshIndex)`
- `getMeshNode(nodeIndex)`
- `getMeshNode(nodeIndex)->getObjectId()`
- `getCamera(id, [output camera parameters])`
- `getLight…`

Models as mesh libraries – shared pointers between models/meshes

Sometimes a model is only used as a library, and not as a scene definition. In such a case, it is preferable to deal with the meshes as objects in their own right, and not deal or even hold, if possible, the model. For example, the position and animation of objects might come from app logic and not the model.

The Framework deals with that with the Shared Refcounting feature. Call `Model -> getMeshHandle()` to get a `RefCountedResource<Mesh>` that will be functional work for any use. Feel free to discard the pointer to the model if it is not needed - the new pointer will deal with its lifecycle management.

2.2.2. Effects

Effects are PowerVR Framework's way to automate rendering. An effect wraps all necessary objects to actually render something.

Previously (up to and including version 2.0 in August 2016), the PFX file was mainly a shader container. The third version of PFX completely overhauls it to be a rendering description. See the PFX specification and the Vulkan *Skinning* example for use of the last version of PFX.

In general, a PFX can contain enough information for a complete rendering effect, including:

- different passes
- subpasses that can be implemented by different pipelines
- conditions to select different models to be rendered for different passes/subpasses
- memory objects (uniforms and/or buffers)

On the application side, in general, the intention is for the developer to add different models to different subpasses of the PFX. For example, for a Deferred Shading implementation, there may be three subpasses - a G-Buffer geometry subpass, a Shading subpass, and a PostProcessing subpass. In this scenario, the developer would add their objects to the first subpass, the light proxies such as circles to the second pass, and a full screen quad for the third one, and kick the render.

See the PFX spec and any Framework examples implemented with PFX files (*Skinning*), the PFX spec, and the RenderManager documentation for details on how to use these.

The RenderManager can be considered a Reference implementation for how PFX might be used. It is supplied for Vulkan only.

2.3. Utilities and the RenderManager

The RenderManager is a very ambitious project that has been written specifically to support the new, improved PFX files.

It is essentially a minimum system for completely automated Vulkan rendering. As the PowerVR Framework features a very permissive licence, it can also to use it as a starting point for rendering/game engines.

In conjunction with the PFX and POD files, it makes completely automating rendering very easy, and prototyping rendering demos absolutely trivial.

2.3.1. Simplified structure of the RenderManager Render Graph

The PowerVR Framework assets model structure basically contains of a hierarchy of nodes, which connect mesh objects with material objects. So, when mentioning a node here, this is referring to a specific renderable instance of a mesh with a known material.

Effects contain passes (final render targets), which contain subpasses (intermediate draws), which contain groups (imposing order in the draws, allowing to select different pipelines), to which nodes are added, matched with specific pipeline objects suitable for rendering them. There are more than ten intermediate classes and objects to this graph, but it is important to remember that the node is the final renderable object in RenderManager. From just a reference to a (Render) node object, a developer can navigate all the information required to render something start to finish.

2.3.2. Semantics

Semantics were first introduced as part of PVRShaman (PVR Shader Manager) and the original PFX format. It is a way for a developer to signify to an implementation (for example PVRShaman, or now the RenderManager class) what kind of information is required by a shader. In other words, which data from a model needs to be uploaded to which variable in the shader.

For example, in the old PFX format the developer could annotate the attribute `myVertex` with the semantic `POSITION`, so that PVRShaman knew to funnel the Position vertex data from the POD file into this attribute and display the file. Semantics have been simplified and expanded since then.

First, the PFX format itself is completely unaware of what semantics exist, and what is valid. It is an implementation using the PFX that defines that.

So, there are two sides to semantics:

- The PFX file has semantics, which define what information is *required* to render
- The POD file has semantics, which define what information is *provided* by this model

In order to render, the two sides must match. If the exact same strings are used, this can be done automatically. Otherwise, the developer can create custom mappings to map different semantics together. For example, there could be a model with an attribute semantic called `VERTEX_POSITION` instead of the commonly provided `POSITION` that the `PODReader` class provides by default.

Vertex data, material data, and other parts of the scene provide semantics.

The PFX file can use semantics to annotate attributes, uniforms, textures, or entries into buffers. Entire buffers can also be annotated by semantics, but no automatic handling is currently done.

2.3.3. Automatic Semantics

Automatic semantics is one of the most powerful features of RenderManager - it is the way to automate an actual, moving scene.

Inputs of the PFX are matched to outputs of the application, or the model, allowing them to match precisely and automatically update. The RenderManager can generate the necessary commands to update them by reading data from the model and writing it into whatever the PFX requires. Currently, attributes are read only once while uniforms or buffer entries are expected to be updated once per frame.

In general, for a simple scenario like the *Skinning* example, the following steps are enough:

1. Definition: Define a `pvr::utils::RenderManager` object
2. Initialisation:
    1. Load a PFX from file into a `pvr::assets::Effect` object with a PFXReader
    2. Load a POD file into a `pvr::assets::Model` object with a PODReader

3. Initialise the RenderManager with the `Shell(*this)`, the swapchain and a DescriptorPool
4. Add the effect to the RenderManager:
   `int effectId = renderMgr.addEffect(myEffect, context);`
   Using `EffectId` is not necessary, as the effects get sequential ids from 0 increasing.
5. Add the model to a subpass of the effect. As a shortcut, add to all passes.
   `int modelId = renderMgr.addModelToSubpass(model, effectIdx, passIdx, subpassIdx);` (normally `0,0,0`)
6. Kick the RenderManager object processing:
   `renderMgr.buildRenderObjects("CommandBuffer")`

3. Set up automatics, if required. For whatever granularity needed, call `createAutomaticSemantics()`. This can be called for the entire RenderManager, or for a specific pipeline, or for nodes. It is recommended to call this globally, on the RenderManager. Otherwise, it can be called for effects, passes, subpasses, pipelines or nodes.
4. Get the rendering commands into a command buffer:
   1. Depending on the specific requirements, commands can again be generated for different sub-trees or sub-objects. In general, recording commands are usually called on the pass.
      `renderMgr.toPass(effectIdx, passIdx).recordRenderingCommands(myCmdBuffer,…)`
   2. The command buffer will then contain all the rendering commands to render this pass with all its subpasses.
   3. If using uniforms (uniform semantics) in the PFX, UpdateCommands must be recorded because uniforms are updated in the command buffer. This should be done usually before the rendering commands - `recordUpdateRenderingCommands`
   4. If using buffers (buffer entry semantics) in the PFX, it is normally not necessary to prepare recording commands for them; it is enough to update their memory.
5. For every frame:
   1. Calculate/advance the current frame for any or all models. Usually, `model->setCurrentFrame(XXX)`) and any other logic required.
   2. If using semantics, apart from the automatic semantics, update them with `updateSemantic()` in the corresponding objects
   3. If using automatic semantics, call the `updateAutomaticSemantics()` function. This will update all the memory with the relevant semantics. In the case of the buffer entry semantics, this is all that is needed as `map`/`unmap` has been called. In the case of uniform semantics, the command buffers must still be submitted so that the `updateUniform` commands are called.
   4. Submit any command buffers that have been prepared.

In summary, the render manager will have:

- Parsed and read the PFX
- Added the nodes of every model added to pipelines suitable to render it, creating a rather complicated render graph
- Matched the required semantics of the PFX with the semantics of the POD preparing them for automatic updating
- Created commands to render each pass/subpass/pipeline/node
- For each frame, iterating all automatic semantics and updating them

2.4. Reference Counting

The PowerVR Framework always uses automatic reference counting to reduce bookkeeping needed for its use. The `pvr::RefCountedResource` and its family of classes such as `EmbeddedRefCountedResource`, `RefCountWeakReference` and so on, is basically a smart pointer class with several interesting features. It was developed to support the PowerVR framework and can be found in PVRCore. PVRVk also carries its own version of this class to avoid dependencies.

Note: Remember that with the MIT licence the PowerVR SDK is covered under, these classes can be reused in the developer's own code, even outside the Framework.

2.4.1. Performance

The `RefCountedResource` is an optimal smart pointer implementation. If it is used as recommended in the same way the Framework does in a release build, dereferencing can be just as efficient as a raw pointer. The overhead is the size of a pointer and two 32 bit integers.

Remember to use the `construct()` method to create the underlying object whenever possible, as this puts the reference count in the same contiguous block of memory as object implementation itself. This is normally an immediate benefit as the two will usually be accessed together, and should make it as fast as accessing a raw pointer. Needless to say, the PowerVR SDK exclusively uses this path.

2.4.2. Features

Single block refcounting and object

If `ptr.construct(…)` is used, this call perfect-forwards all arguments to the underlying class's constructor, if one exists. Otherwise, if a constructor that can accommodate all the arguments cannot be found, a compilation error will be raised. Note the dot "`.`" operator - this is an operation on the pointer, not the pointed object. The arrow "`->`"operator should not be used for this.

This mechanism is not mandatory, but it is highly recommended, as it avoids memory fragmentation and promotes locality.

Example:

```
Class Foo
 {
     public:
     Foo(int a = 0);
     Foo(int a, int b);
     void DoStuff();
 }

 RefCountedResource<Foo> myFoo;
 myFoo.construct();      // Good, Calls Foo(a=0);
 myFoo.construct(42);    // Good, Calls Foo(42);
 myFoo.construct(42,1);  // Good, Calls Foo(42,1);
 myFoo.construct(myBar); // ERROR – the refcounted resource cannot find a constructor
 myFoo.construct(0,1,2); // ERROR – the refcounted resource cannot find a constructor
```

The alternative is to create the object manually with "new", and then just `reset(...)` the smart pointer to the object.

Deterministic reference count

The only overhead the developer must do for these objects is to release them. This happens automatically when they go out of scope. It also generally happens in any case where the pointer no longer points to the object, for instance calling `ptr.reset()`, or the equivalent `ptr.reset(NULL)`.

In general, it is quite safe and recommended to just allow variables to go out of scope. Otherwise, use `ptr.reset()`, when the object needs to be kept around, but not holding the reference any more, for example objects that are members of an object that will be kept around.

When an object is not pointed to by any `RefCountedResources`, its destructor will be called and its memory freed. This is not garbage collection – it is a deterministic operation. It happens immediately, in the same thread where and when the last reference of an object is released, before the call that released it or its scope exits.

```
                          // myFoo has a reference count of 1
{
    auto myFoo2 = myFoo; // Refcount 2

    myFoo.reset();       // myFoo now points to null/refcount 0, myFoo2 points to the object,
                         // and has refcount 1

    myFoo.reset();       // NOP - absolutely no effect, myfoo is still null

    myFoo = myFoo2;      // Again myFoo points to the object, refcount 2

    myFoo.reset();       // myFoo again points to null/refcount 0,
                         // myFoo2 still points to the object, and has refcount 1

    myFoo2.reset();      // Refcount 0: Before this call returns, the object's destructor
                         // is called and its memory is deleted

    myFoo2.construct();  // New object, refcount 1;

}                        // myFoo2 out of scope: Refcount 0, destructor called
                         // and the new object deleted here
```

Weak references

An important weak point of reference counting is the possibility of cyclic references. When two objects hold a reference to each other, they will keep each other alive even when the program holds no reference to either of them, causing a memory leak that can easily become a much bigger problem. The way to break this is with programmer care, and the tool that can be used is weak references - handles that point to a reference counted object but without keeping it alive.

The `RefCountedWeakReference` is exactly that. These objects are similar to the `RefCountedResource`, but additionally they will allow an object to be destroyed, and also allow the developer to check if the object still exists before dereferencing the pointer. Some PVRVk objects are accessed through such weak pointers.

As a rule of thumb, child objects keep strong references to their owners so that the owners don't get released before their children, for instance, a CommandBuffer holds a reference to a pool object. However, when a parent object needs references to child objects, these are weak references, for example, a pool needing a list of all its CommandBuffers.

This rule is not absolute though. The device object may need be removed at any point, taking all objects with it. Hence, all internal references to a device are weak.

Embedded refcounting

Some of the classes additionally use Embedded Refcounting. This is an Intrusive Refcounting scheme, and is used when an object needs to be aware of its own refcounting - most notably for its implementation to be able to generate smart pointers to itself.

For example, a CommandBuffer is generated by a CommandPool object, and during its generation, it needs to be provided with a pointer to its owning CommandPool. Therefore, the CommandPool object uses EmbeddedRefcount, so that it can generate such a pointer during its `allocateCommandBuffer` function. All objects that follow similar situations (especially Device, CommandPool, DescriptorPool and so on) use embedded refcounting.

2.4.3. Creating a smart pointer

To create a smart pointer to a class there are three paths - a slow path, a fast path, and an embedded fast path.

Slow path

The slow path uses a developer-provided pointer to an already created object. Pass this pointer to the constructor of a `RefCountedResource` of its real class. Do not pass it to a subclass or the class that it is intended to be handled through, unless the object has a virtual destructor.

- Unless the object has a virtual destructor, it must be initially wrapped into a `RefCountedResource<Actual Class>` and not one of its base types. This is so that the correct destructor is called.

- If the object has a virtual destructor, it is possible to create the initial smart pointer with the type of any of its superclasses, but not void.

In both of these cases, after the initial `RefCountedResource` is created, it can be freely converted to any compatible `RefCountedResource` type (any superclass, and `void`). Pointers to the initial object can, and should, be discarded. Cast the refcounted resources back and forth to their sub/super classes.

No matter what the developer does, when the last reference is done, the correct destructor will always be called, as it is built into the refcounted entry. There is no known way to break this.

The reason this path is called the slow path is because of the accountability of the `RefCountedResource` object. The reference counting data itself and the pointer to the object, which get accessed on any operation like copy or dereference, are stored in different parts of memory. This memory may be completely unrelated to the block of memory where the object lives, so in the general case, memory locality is reduced.

Simple dereferences of the object that do not need to access refcounting data will still be just as fast as the fast path. However, doing copies, refcounting inspections, deletes or other operations that require access to the refcounting data will normally introduce an additional cache-miss.

Fast path

The fast path is the recommended method, and used throughout the Framework. It involves creating a `RefCountedResource` of the class that needs to be created, and then calls its `construct(…)` method to initially create the object. Pass to the `construct()` call the same arguments that would have been passed to the constructor of the class that is to be created. This is analogous to the `std::make_shared` call.

The parameters will then be perfect-forwarded to the correct constructor. One memory block will be allocated to hold the object and the refcounting bookkeeping data together. This improves memory locality when refcounting operations and pointer dereferencing are performed close to each other which is very common.

Embedded refcount path

This path is very similar to the fast path, but is additionally intrusive. In order to use it, a class must be designed to be used only through a `RefCountedResource` pointer. The benefit of this path compared to the fast path is that the class is aware of and can access this bookkeeping information. It can therefore generate smart pointers to itself with the `getReference()` and `getWeakReference()` methods.

The disadvantage of this path is that the class must be designed to be used through the smart pointer, and any instances of it must necessarily only ever be instantiated and used through a `RefCountedResource`.

To design a class to be used with embedded refcounting:

- Inherit from the `EmbeddedRefCount` class, passing as template parameter the class type itself, as per the CRTP pattern
- Make all its constructors protected or private, including the default, so that it can never be instantiated directly by app code
- Add the `EmbeddedRefCount` as a template friend class to allow access to its private constructors
- To actually create instances of the class, add one static factory function to it to for each of its constructors, with the same number and types of arguments. The body should forward them to the a call to the static function `createNew()` of the `EmbeddedRefcount`.

```
class MyClass: public EmbeddedRefCount<MyClass>
{
 private:
    template<typename> friend class ::pvr::EmbeddedRefCount;
    MyClass(ParamType constructorArg){ … }

 public:
     EmbeddedRefCountedResource<MyClass> createNew(ParamType constructorArg)
     {
        // possible to use std::forward(...) here
        return EmbeddedRefCount<MyClass>::createNew(constructorArg);
     }
 }
```

Shared refcounting

Shared refcounting is an interesting method, and a convenient feature of the `RefCountedResource`. It allows the developer to use reference counting with objects that have their lifetime tied to other objects, such as specific members of an array. Consider a scenario where an array of items is allocated, and a developer would like to destroy it when no outside code references any of the object it contains. What the developer can do in this case is this:

- Make the array refcounted as normal. A `std::vector` or `std::array` would work perfectly here. A c-style array would not work here, so assume a `std::array<int>` or a `std::vector`
- Create an empty `RefCountedResource<type>` where type is the type of the object that needs to be accessed
- Call the `shareRefCountFrom` on it, passing the original ref counted resource and a pointer to the object (see code below)

```
{
     RefCountedResource<int> sharedMember0;
     RefCountedResource<int> sharedMember2;

     {
         // We use this scope to show that myArray will be alive even after it is out of scope
         RefCountedResource<std::array<int, 10> > myArray;

         // Create the array using fast path
         myArray.construct();

         (*myArray)[0]=0;
         (*myArray)[1]=10;
         (*myArray)[7]=42;

         sharedMember0.shareRefCountFrom(myArray, &(*myArray)[1]);
         sharedMember2.shareRefCountFrom(myArray, &(*myArray)[7]);

         // myArray goes out of scope here, but it is kept alive by sharedMember0 and 7
     }

     printf("%d, %d", *sharedMember0, *sharedMember2); //output : 10, 42
 }
 // sharedMember0,2 get released - NOW the actual myArray object gets destroyed
```

The PowerVR Framework uses this feature to allow developers to use the model class as a mesh container. The developer can call `getMeshHandle()` on a model, and get a perfectly working smart pointer to it. The pointer will be under the hood ensuring that the developer does not get overwhelmed with a multitude of objects they do not care about. Therefore they can load a model with (for example) five meshes, get pointers to the meshes they want, discard the original model, and not have to bother about lifetime and clean-up of the original object.

2.5. Input Handling Tips and Tricks

2.5.1. PVRShell simplified (mapped) input

Nearly all SDK Examples use Simplified Input. This is a model we use that is suitable for demo applications. No matter the platform, common actions are mapped to a handful of events:

- Action1
- Action2
- Action3
- Left
- Right
- Up
- Down
- Quit

The Shell already does this mapping. All that is required is overriding the `eventMappedInput` function of `pvr::Shell` as follows:

```
pvr::Result::Enum pvr::Shell::eventMappedInput(pvr::SimplifiedEvent::Enum evt)
```

This function will be called every time the developer performs one of the actions that map to a simplified event.

Table 2. Input Events on Desktop

INPUT EVENT	How to trigger on Desktop (Window)	How to trigger on Desktop (Console)
Action1	Space, Enter, Click centre of screen	Space, Enter
Action2	Click left 30% of screen, Key “1”	Key “1”
Action3	Click right 30% of screen, Key “2”	Key “2”
Left/Right/Up/ Down	Left/Right/Up/Down keys, Drag mouse Left/Right/Up/Down	Left/Right/Up/Down keys
Quit	Escape, Q key, close window	Escape, Q key

Table 3. Input Events on Android

INPUT EVENT	How to trigger on Android
Action1	Touch centre of screen
Action2	Touch left 30% of screen
Action3	Touch right 30% of screen
Left/Right/Up/Down	Swipe Left/Right/Up/Down
Quit	“Back” key

Table 4. Input Events on iOS

INPUT EVENT	How to trigger on iOS
Action1	Touch centre of screen
Action2	Touch left 30% of screen
Action3	Touch right 30% of screen

INPUT EVENT	How to trigger on iOS
Left/Right/Up/Down	Swipe Left/Right/Up/Down
Quit	“Home” key

2.5.2. Lower-level input

Besides this simplified input, it is possible to not use `mappedInput` and instead use the lower level input events:

- `onKeyDown`
- `onKeyUp`
- `onKeyPress`
- `onPointingDeviceDown`
- `onPointingDeviceUp`

All these functions map differently to different platforms, and may not be present everywhere, for instance `keyDown` and so on for mobile devices without keyboards. They can enable custom programming of the developer's own input scheme. These functions can be used normally by overriding them from `pvr::Shell`, exactly like `eventMappedInput`.

2.6. Renderpass/PLS strategies

The PowerVR SDK is designed to work with any conformant OpenGL ES or Vulkan implementation. Most optimisation guidance we provide is sensible for any platform, but some guidance may be critical for PowerVR Platforms. Optimisations we recommend will not normally be detrimental to the performance of other platforms, but they may not actually improve them.

This section details strategies for optimisations relating to efficiently using multi-subpass RenderPasses (Vulkan) or multi-pass rendering (OpenGL ES). All of these optimisations are suitable for any platform that supports them, but their effect on PowerVR architectures makes them crucial to use whenever possible.

These optimisations are expected to benefit any platform, or at worst be neutral and have no effect. However, tile-based architectures (which applies to some mobile), and unified memory architectures (practically all mobile) are expected to hugely benefit.

Setting the load and store ops or using invalidate/discard

In Vulkan and PVRVk, when creating a `RenderPass` object, set the `LoadOp` and the `StoreOp` to it.

The `LoadOp` means "when starting a RenderPass, what do we need to do with whatever contents the FrameBuffer where we are rendering contains?"

There are three options here:

- **Clear** actually means "forget what's in there, use this colour". This is usually the recommended operation.
- **Don't Care** means "we will render to the entire scene anyway, so it doesn't matter, don't load it"
- **Preserve** means "we are incrementally rendering, using whatever is already in the framebuffer, so we need the contents of it to be preserved."

Clear and Don't Care may sound different, but it is important to realise that their effect is practically the same as far as the important parts of performance go. They both allow the driver to ignore what is in the framebuffer. In the case of Clear, the driver will just be using the clear colour instead of the contents of the framebuffer. Don't Care is similar, but also tells the driver that no specific colour is required.

Never use Preserve unless absolutely certain it is needed as it will introduce an entire round-trip to main memory. Its performance cost on bandwidth cannot be overstated. It is recommended to double-check the application design if Preserve is actually required.

In OpenGL ES, the situation is pretty much the same. When `glClear` is called at the start of a frame, or `glInvalidate` depth/stencil before swapping, the driver may well be allowed to discard the contents of the framebuffer / depthbuffer before the next frame.

Obviously, the specific flags depend on usage, but the baseline should be as follows:

Recommendations for LoadOp

- Clear for depth/stencil, using the maximum depth value/whatever the stencil needs to be.
- Clear for colour, if any part of the screen may not be rendered.
- Ignore, if it is guaranteed that every single pixel on the screen will be rendered to. It would be almost the same to always set Clear in every case, but it does not hurt to be pedantic and set ignore if it is suitable. Never set Ignore and have pixels on screen that have not been specifically overwritten, as then there is undefined behaviour and there may be artifacts or flickering.

Conversely, for OpenGL ES:

- `glClear` both `Color` and `Depth` at the start of the frame.

StoreOp

The StoreOp is much the same, but it should be even more obvious. In nearly every case, it is necessary to:

- `Store` the colour so that it can be displayed on screen
- `Discard` the depth and stencil as their work is done

Conversely, for OpenGL ES, before calling `eglSwapBuffers`:

- Do not do anything special for colour (`EGL_PRESERVE` in EGL swap behaviour)
- `glInvalidateFrameBuffers`/`glDiscardFrameBuffers` any FBOs that are not being rendered, and all depth/stencil attachments.

In short:

- Colour usually needs to be cleared on load, unless the contents of the framebuffer need to be explicitly read. A need to load the colour is very commonly a hint that subpasses/pixel local storage should be used instead if possible
- Colour usually needs to be stored at the end of the frame, in order to be presented
- Depth and stencil almost always need to be cleared to max value at the start of the pass
- Depth and stencil almost never need to be stored at the end of the pass, as they are not required for rendering

Subpasses / Pixel Local Storage

Subpasses are one of those optimisations that applications should be designed around. Use them if at all possible, explore them if remotely possible, and rewrite applications to take advantage of them. One of the first questions that should be asked when doing a multi-pass application is: "can region-local subpasses be used with it?"

Conceptually, a subpass is a run through the graphics pipeline (from vertex shader->… -> framebuffer output) whose output will be an input for a later step. For instance, rendering the G-Buffer in deferred shading can be a subpass.

This is similar to rendering to a texture of screen size in one run, and sampling the corresponding texel at the same position as the rendered pixel on the next pass.

If this is designed properly, this allows the implementation to do a powerful optimisation on tiled architectures. The output of the fragment shader of the first subpass is not stored at all to main memory as it is known that it will not need to be displayed. Instead it is kept on very fast on-chip memory (register files) and accessed again from the fragment shader of the next subpass.

For example, in Deferred Shading, the G-Buffer contents can be kept on-chip to be used in the lighting pass. This can have great performance benefits in mobile architectures, as they are commonly bandwidth limited.

The caveat is that for this to happen, each pixel must only use the information from the corresponding input pixel. It cannot sample from arbitrary locations, and it cannot sample at all from the previous contents.

For Vulkan, in order to collapse subpasses in this way:

- Render into the images in one subpass.
- Use these images as input attachments in the other subpass.
- Use `Transient` and `Lazily Allocated` flags for those attachments.

For OpenGL ES, the same effect is done with enabling the `GL_PIXEL_LOCAL_STORAGE` extension. Additionally, the shaders must have been written to explicitly take advantage of it.

In short, use subpass folding wherever suitable. With multiple passes, see if they are suitable for subpass optimisation. For both of these cases, see the *DeferredShading* example.

3. Contact Details

For further support, visit our forum:
http://forum.imgtec.com

Or file a ticket in our support system:
https://pvrsupport.imgtec.com

To learn more about our PowerVR Graphics Tools and SDK and Insider programme, please visit:
http://www.powervrinsider.com

For general enquiries, please visit our website:
http://imgtec.com/corporate/contactus.asp