PowerVR Framework

Development Guide

Filename	:	PowerVR Framework.Development Guide
Version	:	PowerVR SDK REL_17.2@4910709 External Issue
Issue Date	:	30 Oct 2017
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. Very commonly, these will also be virtually optimal and sensible to do on most/all 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
- It is more convenient, more modern than "Raw" APIs - for example C++ classes for all Vulkan objects
- It is obvious what the code does to someone used to the "Raw" APIs - take a look at any example
- Any differences it has from the "Raw" APIs (e.g. 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. A Note on Libraries

All the PowerVR Framework libraries are by default compiled as static libraries. A developer could conceivably configure these as dynamic libraries (.dll/.so etc) to allowing dynamic binding, but no such attempt is made in the SDK.

Our provided demo solutions/projects will build any required PowerVR modules using project dependencies/makefile rules etc. as required.

The Framework is a high-level C++ project, so no "sterilised" C APIs exist. The libraries and the final executable should always be compiled with the same compiler make/version with compatible parameters to ensure that the One Definition Rule (ODR) is observed. The compilers must use the same C++ name mangling rules and other details, otherwise the behaviour may be unexpected.

1.1.1. Default Library Location

The built-in SDK project files place library files into:

```
[SDKROOT]/Framework/Bin/[PLATFORM,CONFIG…]/[prefix] [MODULE_NAME].[extension]
```

For example:

```
c:\Imagination\PowerVR_SDK\Framework\Bin\Windows2010\Release_64\PVRVk.lib
```

or:

```
//home/me/Img/PowerVR_SDK/Framework/Bin/Linux_x86_64/Debug_X11/libPVRVk.a
```

1.2. Custom 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\]/Builds/Include`.

It is highly recommended that developers read about and use these header files as they are hugely 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 linking to anything, by using advanced C++ features.

To use them, drop them somewhere where the compiler will find the header file - for example wherever library header files are usually stored. Type `#include DynamicGLES.h` at the top of the code file, and everything will work.

There is no need to link, or use function pointers. It is still necessary to test for extension support and there is a function for that on 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.

Keep in mind that 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. Vulkan_IMG.h/ Vulkan_IMGX.h

Vulkan_IMG.h is a modified **vulkan.h** file. This was created in order to change something that we felt would damage **PVRVk**'s API – C-style enums that throw symbols in the global namespace without compile-time checking.
Hence, **Vulkan_IMG.h** replaces Vulkan's enums with strongly-typed enums and, after taking that into consideration, adds a few changes so that things make sense:

- enums become enum classes.
- enum members lose the `VK_ENUM_NAME` prefix and gain a `e_` prefix. The reason for the e_ prefix is that without it, and with the preceding rule, some members would need to start with a number, which is illegal. So for example, `VK_DESCRIPTOR_TYPE_SAMPLER` becomes `VkDescriptorType::e_SAMPLER`.
- Bitfields / Flags are not separated into Flags/FlagBits, but are like any other enum (strongly typed). They have bitwise operators defined to return the correct type.
- No prototypes are defined for any functions, since function pointers must be loaded per device

1.3. Overview of the PowerVR Framework Modules

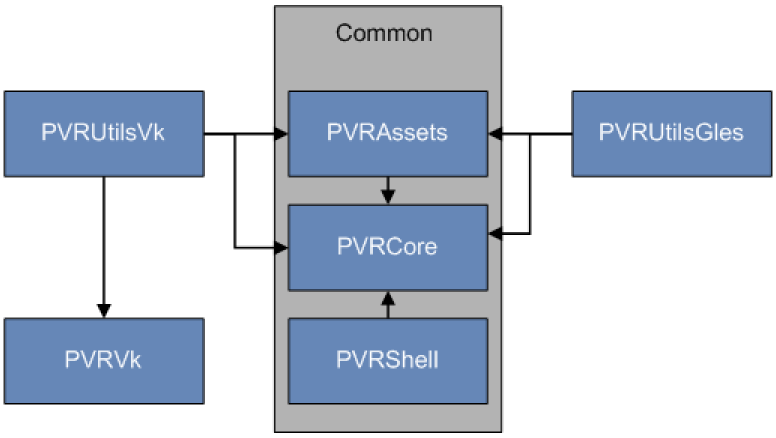

Figure 1. Framework Structure

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 add your code.

Its public contents can easily be accessed at any time from inside the application class itself. The application class normally derives from the `pvr::Shell` class (see "The Skeleton of a Typical Framework 5.x Application"), and is powered by its callbacks. So, in most IDEs, after writing `this->` somewhere in the main application class, autocomplete should tell you all you need to know to use **PVRShell**.

PVRShell handles everything up to the level of window and window creation, and provides handles to it. Higher levels - for instance GPU contexts/devices/API calls/surfaces etc including any and all API objects - are not handled by the Shell. These should be dealt with from the application, or from another library - we recommend **PVRUtilsVK**/**PVRUtilsGles**.

In summary, **PVRShell**:

- abstracts away the platform i.e. display, input, filesystem, window etc.
- 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

How to use PVRShell

PVRShell uses parts of **PVRCore** and therefore requires it. You will need to include "`PVRShell/PVRShell.h`", in your 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` etc).

1.3.2. PVRVk

About PVRVk

PVRVk is an independent module providing a convenient, very 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++, object-oriented 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 for functions, wherever suitable
- structs get initialised to sensible defaults

Developers who have used the Vulkan spec should find it very familiar without any other external references. All user-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 independently, by following the Vulkan specification and getting a handle on the obvious conventions:

- Enums are type safe (enum class) and their members lose the prefix
- Vulkan functions become methods (member functions) 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 it uses **Vulkan_IMG.h**. The compiled library file is named `PVRVk` e.g. `PVRVk.lib`, `libPVRVK.a`. Obviously 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 of course always include the **PVRVk** headers anyway.

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

PVRUtilsVk uses **PVRVk**. All of the PowerVR SDK Vulkan Examples except for **HelloAPI** (completely raw code) are 90% **PVRVk**/**PVRUtilsVk** code.

1.3.3. PVRAssets

PVRAssets is used to work directly with the CPU-side of the "authored" parts of an application, i.e. 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 essentially 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 nicely to these, but can freely and 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 easily with the rest of the framework. For instance, an `AssetReader<Texture>` for JPEG, or an `AssetReader<Model>` for Wavefront OBJ would be easy 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 familiarise yourself with **PVRAssets**.

1.3.4. PVRCore

About PVRCore

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

- data structures
- code helper functions
- math e.g. frustum culling, cameras
- string helpers e.g. unicode, formatting

PVRCore has several fully realised classes that can be used on 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 interesting 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 user-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 (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.

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** include asynchronous operations (a texture loading class that loads textures in the background), and the RenderManager, 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+. It will need to be included with "`PVRUtils/PVRUtilsVk.h`" or "`PVRUtils/PVRUtilsGles.h`" (`PVRUtils/Vulkan/Asynchronous.h`).

Check the "Rendering 2D with UIRenderer" document for an overview of the basic use of **UIRenderer**.

All the SDK examples use **PVRUtils** for all kinds of tasks. They use **UIRenderer** to display titles and logos (except for **HelloApi**, **IntroducingPVRShell**). **IntroducingUIRenderer** and **ExampleUI** are both examples of more complex **UIRenderer** usage. **Multithreading** showcases the Asynchronous API for Vulkan.

1.3.6. PVRCamera

About PVRCamera

PVRCamera provides an abstraction for the hardware camera provided by Android and iOS. Currently it is only implemented for OpenGL ES (not Vulkan) due to the Android API, so 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 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 version 4.0

PVRNativeApi

PVRNativeApi has been removed. High level parts, such as texture uploading, have been moved into the corresponding **PVRUtils**.

OpenGL ES low level parts, such as function pointers, have been moved into the convenient OpenGL ES header-only OpenGL binding (`DynamicGles.h`)

Vulkan low level parts have been moved into **PVRVk**.

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 and iOS. QNX implementations can be provided upon request.

1.4.2. Compilers and Toolchains

For development, we used:

- Visual Studio 2013 and tested with 2015 for Windows,
- gcc (5.2+) for Linux
- Android SDK tools with gradle and clang for Android. For tools versions, check the `…Build\Android\build.gradle`, as it is regularly updated. At the time of writing the version is 24.0.1. It is always recommended to use the latest version.
- XCode for OSX and iOS.
- Clang on Linux is unofficially supported. It runs but it is not regularly tested.

1.4.3. Filesystem (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 (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 sometimes used by **PVRUtils**. Window creation parameters such as window size, full screen mode, framebuffer formats and similar are accessed by several `setXXXXXXX` functions that the user 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 spec 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 etc. 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 strange solutions without jumping through hoops.

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

The obvious change that this causes is that all API objects (e.g. contexts, devices, surfaces) are explicitly created in application code, by the user. **PVRUtils** makes this really easy, but it is still completely obvious what is happening, and is really easy to follow the code. Additionally, by also 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 user-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:

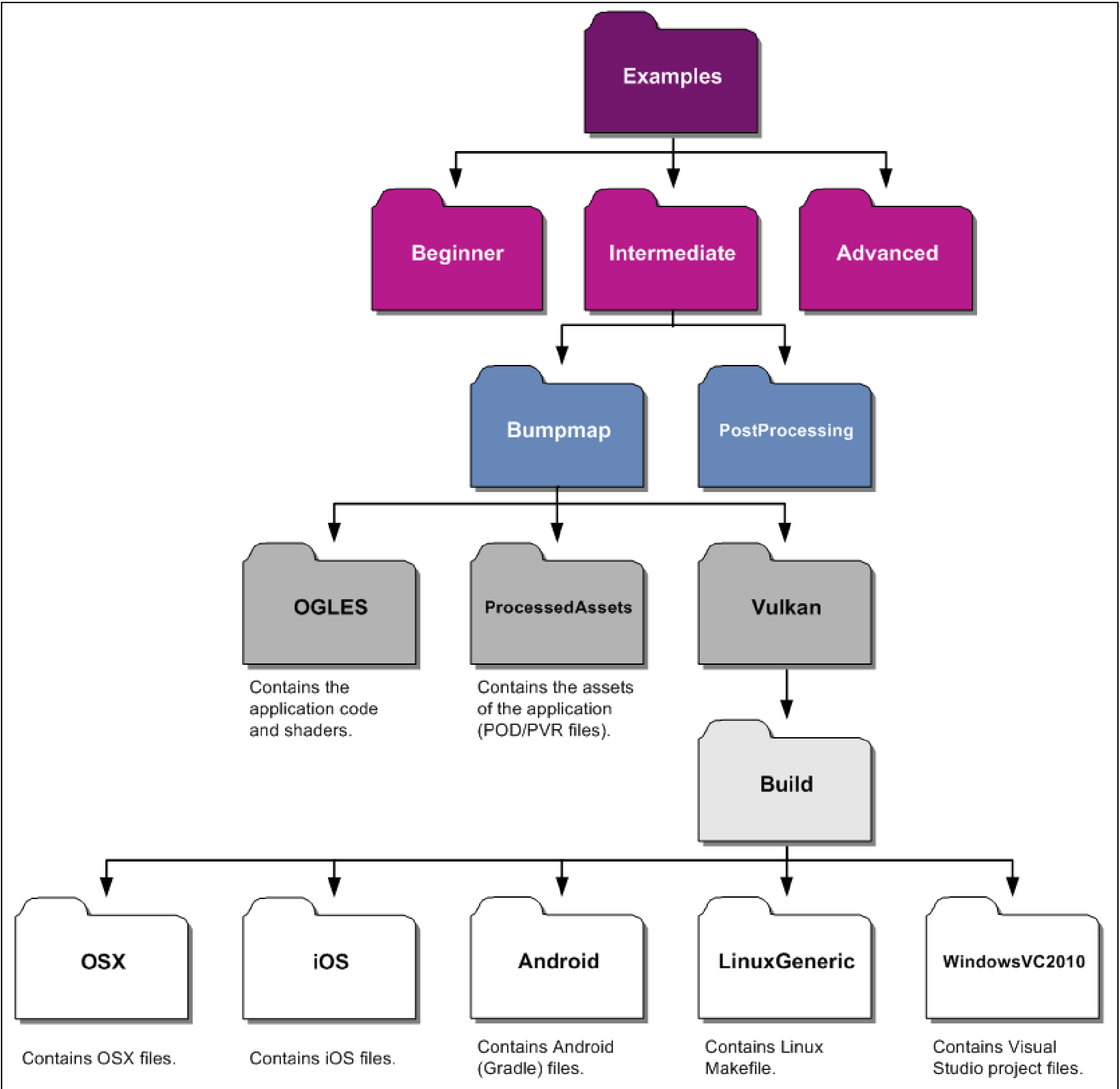

Figure 2. PowerVR SDK Examples Structure

1.6.2. The Minimum Application Skeleton

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

- The application should 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 (calling conventions etc) 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. **PVRUtils** automates much of this functionality.

There are slight changes due to the Object Oriented paradigm it follows, but they are simple and intuitive. In general:

- Vulkan *functions* become *member functions* of the class that is their first input parameter object. For example `VkCreateBuffer` becomes a member function of `pvrvk::Device`, so users will need to use `myDevice->createBuffer()`.
- *Functions* without an *input parameter* object remain *global functions* in the `pvrvk::` namespace. For example, `pvrvk::CreateInstance()`.
- Simple *structs* like Offset2D etc. 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++ 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 used "raw" but come from the **Vulkan_IMG.h** header. Therefore:
  - They become C++ scoped enums (enum class *VkTypeName*)
  - The enum *type name* remains unchanged
  - The `VK_ENUM_`*`TYPE_NAME`*`_` prefix of enum members is dropped and replaced by e_
  - *Flags/Bitfields* are used like every other enum as bitwise operators are defined for them. `VkCreateBufferFlags` and `VkCreateBufferFlagBits` become just `VkCreateBufferFlags` and is 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. This particularly applies for initialisation and any kind of CPU-GPU transfer, especially loading textures and similar multi-level tasks. Creating an instance, getting device function pointers, enabling extensions, creating surface, getting physical device, querying queue capabilities, creating a logical device, querying swap chain capabilities, deciding a configuration, creating a swapchain - it is a pretty long list.

Initially, the **PVRUtilsVk** helpers can help automate most (all) of those tasks, without taking away any of the power. The helpers can be used only when needed, or everywhere, or do an "automate everything" approach, or just use raw **PVRVk**; whichever feels suitable.

Our recommendation is to look at any SDK example for the simple use case – and then, if more information is required, follow the utility code. The helpers are usually rather simple functions, though in some cases they can get longer.

One of the more important helpers is texture loading, which is automated. The particulars of it can be daunting: Determining the exact formats, array members, mipmaps, calculating required memory type and size, and then allocating yet more memory for staging buffers, copying data into it, kicking transfers to the final texture, then waiting for the results... Alternatively, make sure to use the texture upload functions. To learn more about the particulars (staging buffers, determining formats etc), check the implementation of the texture upload function.

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

As an important note, especially during initialisation the **PVRUtilsVk** utility functions will use the `DisplayAttributes` class. The Shell auto populates this from defaults, command line arguments and the `setXXXXX()` functions. If this functionality is not required (e.g. the Shell is not being used) then it can be done manually. This is achieved by defining a `DisplayAttributes` object and then changing members where the default should be different.

The **UIRenderer** will be described on its own.

PVRUtilsEs

PVRUtilsEs is similar in use to **PVRUtilsVk**, but tailored to the OpenGL ES API.

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

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 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::SecondaryCommandBuffer`.

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 the state or "open" a command buffer. Then render any sprites required, and finally `endRendering()` follows.

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 user to `render()` it and set things like its colour, rendering mode etc.

The layout itself and the positioning are not done by the Sprite interface class; rather 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, one could have a 2DGroup representing a kind of Panel with components, add and position text sprites and an image to it, then add this 2D group into a MatrixGroup. Finally, this could be transformed with a projection matrix to display it in a Star Wars-intro marquee like way. This usecase is shown in the **IntroducingUIRenderer** example.

An example of layout

In order to better understand this, here is a simple example. The user wants 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. The user wants this to start from a point in the centre of the screen and 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 pixel offset of each text based on line spacing. Each frame, add a number to each text's pixel offset `y` to scroll them.
4. Anchor the top left corner of the top left symbol to the top left of the matrix group. Anchor the bottom right of the bottom right to the bottom right, 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** Create Font tool to create a `.pvr` file. This is actually a `.pvr` texture file that contains enough metadata to be used as a font by `PVRUIRenderer`, or the old style Print3D used in previous 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(…)` etc
2. Set up sprites e.g. set colour, set text
3. Create hierarchies by adding the sprites to groups
4. Lay out sprites/groups using `setPosition()` etc
5. Call render() on the top level sprite. Never call render() on sprites that are in containers - only call render on the item containing other items. Otherwise, any sprite on which you call render() will be rendered relative to the top level (screen), not taking into account containers at all.

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 centred on the Anchor
- "Offset" is a number of pixels to move the final position by, relative to the parent container, relatively to the finally scaled image

To Render the Sprites

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

There is no need (but 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 (i.e. on the object on which you will call `render()`), and the container itself will take care of correctly preparing its children items for rendering.

Note: It is completely 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 etc. 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 - i.e. 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 part of the other component. So if there are two images and a group that contains ten texts and another two images, three `render()` calls are needed.
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 etc. 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, unless a sprite changes every frame, reuse the command buffer and only re-record it when a sprite has actually changed (for example, the length of a text sprite changes)
- Only call `commitUpdates()` when all changes to a sprite are done. In some cases, especially if text length increases, this operation can be very expensive as (for example) VBOs may need to be regenerated etc.

Important

- If the command buffer is "open" (i.e. `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" (i.e. `beginRecording()` has **NOT** been called) when `uiRenderer->beginRendering()` is called, the command buffer will be reset and opened. In the end, the command buffer **WILL** be closed (`endRecording()` will be called) when `uiRenderer->endRendering()` is called.

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, i.e. Instance, PhysicalDevice, Device, Surface, Swapchain and Depth buffer. Unless doing a specific exercise, use the **PVRUtilsVk** helpers, otherwise the two to three lines of code will 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 etc, this may be the resource managers and similar classes.

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

Initial Setup

1. We create the instance, device, swapchain, surface, queues, on screen FBO with our helpers. These can be created with various levels of detail with different helpers, but usually we use `pvr::utils::createInstanceAndSurface()`, `pvr::utils::createLogicalDeviceAndQueues()` and `pvr::utils::createSwapchainAndDepthStencilImageView()`
2. We create `DescriptorSetLayout` objects depending on our app, i.e. `pvrvk::LogicalDevice::createDescriptorSetLayout()`
3. We create the `DescriptorSet` objects based on the layouts i.e. `pvrvk::LogicalDevice::createDescriptorSet()`
4. We create the `Pipelines`
5. We create the `PipelineLayout` objects using the descriptor layouts i.e. `pvrvk::LogicalDevice::createPipelineLayout()`
6. We configure `PipelineCreateInfo` objects (amongst others using those shader strings) and create the pipelines
7. We 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
8. We create the pipelines i.e. `myLogicalDevice->createPipeline()`

Textures and Buffers

1. We create the memory objects such as buffers, images and samplers
2. We can get streams to the textures, and then load them into `pvr::assets::Texture` objects using `pvr::assets::textureLoad()`
3. We create `pvr::api::Texture` objects with `pvr::api::textureUpload()`. This can even be done in a single line. For a method using **PVREngineUtils** to do this asynchronously in a multithreaded environment, see the Multithreading example.
4. We use `context->createBuffer()` and `context->createBufferView()`, or the shortcut `context->createBufferAndView()` for creating UBOs or SSBOs.
5. We map them with `getResource()->map()`
6. To automatically layout buffers that will have a shader representation (UBOs or SSBOs), we use `StructuredBufferView`. This is an incredibly useful class. The UBO 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.

Objects

1. If any objects are mutable (for example, a camera) create one per swapchain image using `getSwapChainLength()`, or use another kind of synchronisation
2. We update the descriptor sets with the actual objects. This might sometimes need to be done in `renderframe` for streaming resources
3. We 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.
4. Use a loop to fill them up as follows: (very simple case)

- For each `swapChainImage`, for the CommandBuffer that corresponds to that swap image:

- `beginRecording()`
- `beginRenderPass()` – pass the FBO that corresponds to the index of this command buffer
- For each material/object type:
    - `bindPipeline()` - pass the pipeline object
    - `bindDescriptorSets()` – for any per-material descriptor sets, e.g. textures
    - For each object:
        - `bindDescriptorSets()` – for any per-object descriptor sets, e.g. 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 etc. Also, this is where command buffer submission must happen for Vulkan.

renderFrame (ES)

In OpenGL ES, it is "business as usual". 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 quite normal to *generate* command buffers in `renderFrame`, although it is optimal to offload as much 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.

However, in all cases, it is highly recommended for Command buffers to only be *submitted* here, in the main thread. Otherwise, the syncrhonisation will quickly get out of hand for no reason at all. There is virtually nothing to be gained by offloading submissions to other threads, unless they are submissions to different queues.

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 signal application exit is required, 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, i.e. clean up things.

For **PVRVk**, leverage RAII: Release any API objects specifically created, by releasing any references that are held. **CAUTION:** Release can be achieved by calling `reset()` on the smart pointer itself, or by deleting the enclosing object, etc. Do **NOT** attempt to release on the underlying object. If such an API exists, it will probably mean something else. Access the `reset()` function with dot operator on a **PVRVk** object. Release 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.

It is recommended to use `device->waitIdle()` before deleting objects that might still be executing. Since the objects are being released, there is no performance worry here.

Note that RAII deletion of **PVRVk** and other framework objects is deterministic. Objects are always deleted exactly when the last reference to them are released. Also, note that objects may hold references to other objects that they require. For example, command buffers hold references to their corresponding pools, descriptor sets hold references to any objects they contain, etc. This is one of the most important features of **PVRVk**. It is something to be aware of, but does not impose any particular required deletion order – on the contrary, it means that normally this is not something to be concerned about.

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 (hopefully not!) 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 PVRUtilsApi

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 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** 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*.

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 either in Raw Vulkan or **PVRVk** is as follows:

- Create a Semaphore.
- Add it in 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, the more overlap.

For example by adding a semaphore with `FragmentShader` as the source and GeometryShader as the destination, you make sure 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 for, 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

Even the simplest case of synchronisation (e.g. `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:

- Two 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 e.g. buffers, (GPU) textures, CommandBuffers, GraphicsPipelines, RenderPasses etc.
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 etc.
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 user.

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.

The log is platform-specific. Where supported, it is shown in the debug output window (Visual Studio, XCode), the console (Linux), the system log (Android Logcat) and/or a file (Windows, Linux, OSX).

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

1.9.1. 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.

1.9.2. 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. Layers are extremely useful here.

Layers

Vulkan Layers sit between the application and the Vulkan implementation, performing all kinds of validation. They can be enabled globally e.g. in the registry or locally e.g. in application code. **PVRVkUtils** enables these for debug builds. It is extremely important not to start without these.

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. By default, this includes layers output on many platforms
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

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:

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

Or for OpenGL ES:

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

If **PVRCamera** is required:

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

2.1.2. Which libraries should be linked against?

For a typical application, add "\[sdkroot\]/Framework/Bin/\[PLATFORM_ARCHITECTURE…" as a library directory. Then, link against:

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

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 the 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**
3. **PVRCore**, **PVRVk**
4. System libraries (usually: m, thread for linux, android_native_app_glue for Android)

If you find that you are getting undefined references to functions that appear to be present, apart from needing a library that is not included, this is usually the culprit.

2.1.4. Are there any dependencies I should be aware of?

If creating your own project file:

- The `[SDKROOT]/Builds/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_IMG.h` bindings.

- The `[SDKROOT]/Builds/[PLATFORM…]` folder may contain library dependencies of the project files. For example, the makefiles that contain the main parts of the building logic for Linux are there, and then included by the Example makefiles. Also the common `Application.mak` that Android projects use is there.
- The `[SDKROOT]/External` folder contains source files for glm, concurrentqueue and pugixml that are all used by the Framework.

For libraries:

- The `[SDKROOT]/Builds/[PLATFORM…]/Lib` may contain libraries that are useful for the specific platform. There are none for Windows/Linux/Android at the time of writing.
- The `[SDKROOT]/Framework/Bin/[PLATFORM…]` will contain the PowerVR Framework libraries after they are built from their projects, usually through the Example solutions.

Wait, shouldn't I link against OpenGL ES, or Vulkan, respectively?

No you don't, 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, for the moment, global function pointers, although this will be revised in the future. 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 user 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. In this scenario, additional synchronisation might be needed with the acquire and the presentation engine.

This is usually quite tricky when it comes to rendering, as a renderpass cannot be split to multiple submissions. However operations on different render targets (especially from different frames) and Compute operations (on the same queue) can be split into different command buffers.

In this scenario, all those command buffers are independent and can be scheduled to start after the presentation engine has prepared the rendering image (backbuffer). The presentation engine will wait for these to finish before it presents the image. This scenario is applicable if no interdependencies exist between the command buffers, or if you synchronise them yourself with semaphores or events. For example, it is possible to render different objects 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 you can utilise different queues or even queue families and sometimes activating different hardware. In general you must devise your own synchronisation scheme in this case, but usually this will be connected to the basic case described above.

2.1.6. How do I create PVRVk objects?

Usually, most **PVRVk** objects with a Vulkan equivalent (buffers, textures, semaphores, descriptor sets, command pools etc) 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 will have provided defaults for as many of the parameters/create info fields as is feasible.

Remember that some "creations" are actually "allocations" from pool objects. There is no "`device->createCommandBuffer()`", instead must call "`commandPool->allocateCommandBuffer()`"

2.1.7. How do I clean up PVRVk Objects? Do I need to destroy anything I didn't create?

Discard (delete the containing object) or reset any smart pointers to them, when you are done with them. Do this, at the latest, in the `releaseView()` function. API objects do not have to be explicitly destroyed, as they are immediately destroyed when their reference count goes to zero.

Other objects may sometimes hold references to them (notably CommandBuffers and DescriptorSets hold references to objects they are using).

2.1.8. How do I clean up a UIRenderer?

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

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

2.1.9. Do I need to manually keep alive any API objects?

Mostly, you 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 or tricky bits:

- `CommandPool` and `DescriptorPool` objects. These objects are not kept alive by their `CommandBuffer` and `DescriptorSet` objects. This means that the user 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 do I load files/assets/resources?

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

- You can directly create a `FileStream`/`BufferStream`/`WindowsResourceStream` etc. 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 (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 do I update buffers?

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

Vulkan by default and exclusively uses this strategy. No synchronisation is attempted, so the user must use 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 user needs to fill it with data, the user must religiously follow the STD140 (or STD430) GLSL rules to determine the actual memory layout, bit for bit, including paddings. Then the user 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 spanners in the works. Fortunately **PVRUtils** comes to the rescue again.

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 you 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 our "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 you 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 initialization 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 you are not planning on using 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 (and Meshes, and Cameras etc)

The top-level class for models is the 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 i.e. 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))` etc 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 e.g. Mesh, Camera, 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, always call `getMesh(…)`. When trying to display the scene, iterate `MeshNodes()`. A Mesh is a description of vertices, 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.

The Framework deals with that with the Shared Refcounting feature. Call `Model->getMeshHandle()` to get a `RefCountedResource<Mesh>` that will 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, up until 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 user to add different models to different subpasses of the PFX. For example, for a Deferred Shading implementation, you might have three subpasses (a G-Buffer geometry subpass, a Shading subpass and a PostProcessing subpass). In this scenario, the user would add their objects to the first subpass, the light proxies (circles for example) 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 (**Advanced/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 & the RenderManager

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

It essentially a minimum system for completely automated Vulkan rendering. As the PowerVR Framework features a very permissive licence, it is actually encouraged to use it as the basis 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, we are 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 user can navigate all the information required to render something start-to-finish.

2.3.2. A note on Semantics

Semantics were first introduced as part of **PVRShaman** (PVR Shader Manager) and the original PFX format. It is a way for a user 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 of a model needs to be uploaded to which variable in the shader.

For example, in the old PFX, format the user 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. Apart from that, the user can create custom mappings to map different semantics together. For example, if there is a Model with an attribute semantic called `VertexPosition` 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 for them.

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.

The idea is that "inputs" of the PFX are matched to "outputs" of the application, or the Model, allowing to match them precisely and automatically update them; the **RenderManager** can generate the necessary commands to update them by reading data from the Model and writing it into whatever the PFX requires. For the moment, Attributes are read only once while Uniforms or Buffer Entries are expected to be updated once per frame (see below for details):

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

1. Definition
    1. Define a `pvr::utils::RenderManager` object
    2. Define an `pvr::utils::AssetManager` object (to support loading for the **RenderManager**)
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. Add the Effect to the **RenderManager**. - `int effectId = renderMgr.addEffect(myEffect, context, assetManager);` Using `EffectId` is not necessary, as the effects get sequential ids from 0 increasing
    4. Add the Model to a subpass of the effect. As a shortcut, add to all passes. - `int modelId = renderMgr.addModelToSubpass(effectIdx, passIdx, subpassIdx);` (normally `0,0,0`)
    5. Kick the **RenderManager** object processing - `renderMgr.buildRenderObjects()`
3. Set up Automatic Semantics, if required
    1. 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. - `renderMgr.createAutomaticSemantics()`
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, Update Commands 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 (see below)
5. For every frame
    1. Calculate/advance the current frame for any/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 (`map`/`unmap` etc. 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
- And then, 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 (`EmbeddedRefCountedResource`, `RefCountWeakReference` etc.) is basically a smart pointer class with several twists, which was developed to support the PowerVR framework. It can be found in **PVRCore**. **PVRVk** also carries its own copy of this class to avoid dependencies.

Note: Remember that with the MIT licence the PowerVR SDK is covered under, you can reuse these classes in your 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 only 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, which is normally an immediate gain as the two will usually be accessed together, and should make it as fast as accessing a raw pointer. Needless to say, the PowerVR framework exclusively uses this path.

2.4.2. Features

Single block refcounting & object

If `ptr.construct(…)` is used (note the dot "." operator: this is an operation on the pointer, not the pointed object, you should not use the arrow "->"operator). 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.

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(...)` to a pointer to the object.

Deterministic Reference Count

The only bookkeeping the user must do for these objects is to release them. This happens automatically when they go out of scope, and in general, in any case where the pointer no longer points to the object - e.g. calling `ptr->reset()`, or the equivalent `ptr→reset(NULL)`.

In general, it is quite safe, idiomatic, and recommended, to just allow the variables to go out of scope. Otherwise, just use `ptr->reset()`, when the object needs to be kept around, but not holding the reference any more i.e. for 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, and 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

The weak point of reference counting is 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 programmer care, and the tool 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 finally allow the user 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 (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: for example, the Device object may need be torn down 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 Embedded Refcount, so that it can generate such a pointer during its allocateCommandBuffer function. All objects that follow similar situations (especially Device, CommandPool DescriptorPool etc.) 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 user-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` of its actual type (and not one of its base types) 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 user 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 way to break this.

The reason this path is called the "slow" path, is because of the book-keeping 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 at a random part of memory, which 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 ref-counting data will still be just as fast as the fast path however, doing copies, ref counting inspections, deletes or other operations that require access to the ref-counting 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 call its `construct(…)` method to initially create the object. Pass to the `construct()` call the exact 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. A single memory block will be allocated both for the object and for the refcounting book-keeping data, improving memory locality when ref-counting operations and pointer dereferencing are performed "close" to each other i.e. very commonly.

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, of course, the class is aware of and can access this book-keeping information, and can thus 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 `EmbedderRefcount`.

```
class MyClass: public EmbeddedRefCount<MyClass>
 {
 private:
    temlate<typename> friend class ::pvr::EmbeddedRefCount;
    MyClass(ParamType constructorArg){ … }
 public:
     EmbeddedRefCountedResource<MyClass> createNew(ParamType constructorArg)
     {
         return EmbeddedRefCount<MyClass>::createNew(constructorArg); //you may wish to use
std::forward(...) here
     }
 }
```

Shared refcounting

Shared refcounting is an interesting twist and a convenient feature of the `RefCountedResource`. It allows the user 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 user would like to destroy it when no outside code references any of the object it contains. What the user can do in this case is this:

- Make the array ref-counted 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;
         myArray.construct(); //Create the array using fast path;
         (*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 users to use the Model class as a Mesh container. The user 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 user does not get overwhelmed with a multitude of objects they do not care about. Therefore they can load a model with (for example) 5 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 user 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 center 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 center 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 center of screen
Action2	Touch left 30% of screen
Action3	Touch right 30% of screen
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 (e.g. `keyDown` etc. on mobiles without keyboards) but can enable custom programming of the user'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 there are things that 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 multiple subpass RenderPasses (Vulkan) or multi-pass rendering (OpenGL ES). All of these optimisations are suitable for any platform, 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 (no effect). However, tile-based architectures (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 i.e. they 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 much the same, 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 revisit 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 when opening a framebuffer, using the maximum depth value/whatever the stencil needs to be.
- "Clear" for colour if any part of the screen may be left untouched and actually needs a screen baseline colour. In other words if every on screen pixel is not being rendered, use clear.
- "Ignore", if it is guaranteed that every single pixel on the screen will be rendered to. It would be fine to always set Clear in every case, but it does not hurt to be pedantic and set ignore if it is suitable. Obviously, never set Ignore and have pixels on screen that have not been specifically overwritten, as then you are getting undefined behaviour and may have artifacts, flickering etc.

Conversely, for OpenGL ES:

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

StoreOp

The StoreOp is much the same, but there the things are even more obvious: In 95% of the cases, it is necessary to:

- "`Store`" the Colour so that it can be displayed on screen
- "`Discard`" the depth and stencil as they are not be needed any more

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 dead giveaway that subpasses/pixel local storage should be used instead if possible (see below)
- 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 it. 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 an amazing optimisation on tiled architectures: The output of the fragment shader of the first subpass is never stored to main memory as it is known that they will not need to be displayed. Instead it is kept on ultra-fast on-chip memory (registers), and accessed again from the fragment shader of the next sub-pass. For example, in deferred shading, the G-Buffer contents can be kept on-chip to be used in the lighting pass, or a final blur pass in post-processing. This can have enormous performance implications in mobile architectures, as they are typically 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, in fact it cannot sample at all from the previous fbo.

To recap, 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 `GL_PIXEL_LOCAL_STORAGE`. Additionally, the shaders must have been written to explicitly take advantage of it.

In short, use subpass folding wherever suitable. Whenever you are doing 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