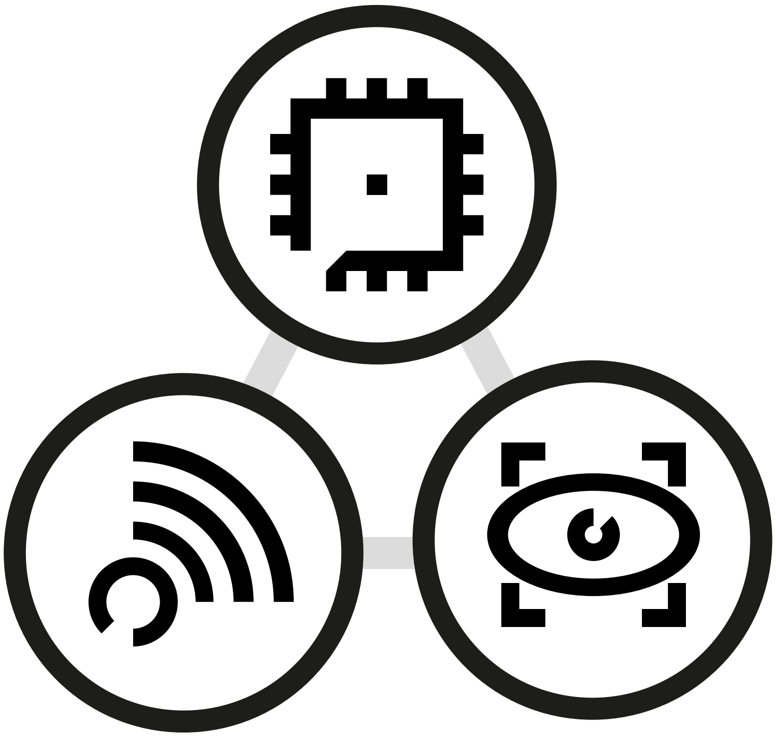
PowerVR Framework Development Guide

Revision: 1.0
11/02/2020
Public



Published: 11/02/2020-08:43

Contents

1. Overview of the PowerVR Framework

The PowerVR Framework, also referred to as the Framework, is a collection of libraries that are intended to serve as the basis for a graphics 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 Framework aims to:

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

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

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

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

2. PowerVR Framework Libraries

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

CMake is used to build the PowerVR SDK demos and Framework modules. The demos depend on the Framework, so developers can use CMake on the demo they are interested in, and it will build everything required. Top-level CMakeLists that build the entire SDK are also provided as it does not take that long to build – generally a few minutes.

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

The provided CMake files place Framework library files into:

```
[SDKROOT]/framework/lib/[PLATFORM,CONFIG...]/
```

For example:

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

or:

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

Using CMake targets to link to the Framework means that this path will be automatically used as required when linking against the Framework targets from the application's `CMakeLists.txt`.

```
target_link_libraries(MyApplication PVRCore PVRShell PVRUtilsVk)
```

3. Framework Header Files

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

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

Use of these header files is highly recommended, as they are very beneficial. This section will give a brief overview of these header files.

What are DynamicGles.h and DynamicEGL.h?

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

To use them, drop the files somewhere where the compiler will find the header file. This is usually wherever library header files are usually stored. "`#include DynamicGLES.h`" can then be written at the top of the code file, and everything will just work. The code will then have OpenGL ES functions available.

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

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

The libraries (`libGLESv2.dll/.lib/so`,`libEGL.dll/.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, and accessible to the application at runtime.

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

These are all replacements for the corresponding `gl2.h`/`gl3.h`/`gl2ext.h`/`egl.h`, so these do not need to be included 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.

What are vk_bindings.h and vk_bindings_helper.h?

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

The problem is that objects that belong to different physical devices (and their corresponding `VkPhysicalDevice`) need to dispatch to different functions. This is because they use different drivers to implement the functions. The Vulkan Loader (the `vulkan-1.dll` that is linked against) will need to do this dispatching, as the global function which is called is located in the loader. The loader must therefore determine which device it needs to be dispatched to and then call the appropriate function. It then also must return the result of the function call. This all causes a needless level of indirection.

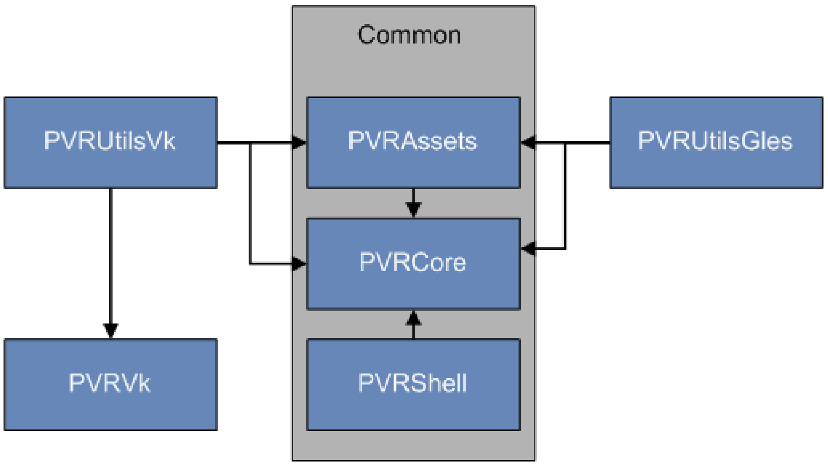
The Vulkan-recommended way to tackle this is simple. Do not use the global functions for functions dispatched to a VkDevice – these are functions that get a VkDevice as their first parameter. Instead, get function pointers for the functions specific to that device, so that the dispatching can be skipped.

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

4. Overview of the Framework Modules

As mentioned in *Overview of the PowerVR Framework*, PowerVR Framework is composed of several modules which can be used to simplify the development of graphics applications. This section will introduce each of these modules, explaining their main features and how to use them in a project.

The image below demonstrates how some of the modules interlink.



What is PVRVk?

About PVRVk

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

The main features are:

- C++ classes that wrap the Vulkan objects with their conceptual functionalities.
- Automatic reference counted smart pointers for all Vulkan objects/object lifecycle management.
- Command buffers that are aware of which objects await execution into them, and keep them alive.
- Descriptor sets that contain references to the objects they contain.
- Default parameters for all parameter objects and functions, where suitable.
- Structs and classes initialised to sensible defaults.
- Strongly-typed enums instead of C enums.

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

To make sure PVRVk can be used by as wide an audience as possible, it is completely independent from any other Framework module.

How to use PVRVk

PVRVk can be used 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. Instead, `e_` is added so that members like `VK_FORMAT_2D` are still valid C++ identifiers - for example: `e_2D`.
- Vulkan functions become 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 it is possible to release whatever is not wanted any more by null-ing or resetting its handle, or just letting it go out of scope.

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

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

For developers familiar with Vulkan, the sections *Using PVRVk* in this document and in the Tip and Tricks document may also give some useful Vulkan tips.

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

Related Concepts

What is PVRAssets?

What is PVRShell?

What is PVRCamera?

What is PVRCore?

What is PVRUtils?

What is PVRAssets?

About PVRAssets

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

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

PVRAssets is the recommended way to load and handle a multitude of assets. These include all PowerVR formats such as POD or GLTF (models), PVR (textures, fonts), and PFX (effects). The PVRAssets classes map very well to these but can easily be used by other formats as well by extending the `AssetReader` class for other formats.

How to use PVRAssets

PVRAssets requires and includes PVRCore, and PVRAssets is required by PVRUtilsVk and PVRUtilsGles. It is necessary to link against the PVRAssets library if using its functionality or PVRUtils(Vk/ES). CMake will perform these links automatically, hence if CMake is used to add PVRUtils to a project it will automatically add PVRAssets and PVRCore. Include "`PVRAssets/PVRAssets.h`" to include all normally required functionality of PVRAssets.

It is best to start from the `pvr::assets::Model` class to become familiar with PVRAssets.

Related Concepts

What is PVRVk?

What is PVRShell?

What is PVRCamera?

What is PVRCore?

What is PVRUtils?

What is PVRShell?

About PVRShell

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

It is intended for the main structure of an application to be a class implementing `pvr::Shell` as seen in *Creating a Typical Framework 5.x Application*. This way, all of the per-platform initialisation (creating the window, reading command line parameters, calling a function at initialisation, every frame, and teardown) is taken care of. Every conceivable platform operation such as loading files from device-specific storages is provided.

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

PVRShell handles everything up to the level of display/window creation. Higher levels, for instance GPU contexts/devices/API calls/surfaces including any and all

API objects, are not handled by the shell. These should be dealt with from the application, normally using PVRUtilsVK/PVRUtilsGles.

In summary, PVRShell:

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

How to use PVRShell

PVRShell uses PVRCore and therefore transitively links it in the application.

1. Add the PVRShell target from `CMakeLists.txt`.
2. Include `PVRShell/PVRShell.h` in the main application class file.
3. 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`.

Related Concepts

What is PVRVk?

What is PVRAssets?

What is PVRCamera?

What is PVRCore?

What is PVRUtils?

What is 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) as the camera texture is provided as an OpenGL ES texture. In Windows/Linux, a dummy implementation displaying a static image instead of the camera stream is provided to assist development on desktop machines.

How to use PVRCamera

Include `PVRCamera/PVRCamera.h`.

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

Related Concepts

What is PVRVk?
What is PVRAssets?
What is PVRShell?
What is PVRCore?
What is PVRUtils?

What is PVRCore?

About PVRCore

PVRCore contains low-level supporting C++ code. This includes:

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

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

How to use PVRCore

PVRCore is transitively linked by CMake 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, `moodyCamel::ConcurrentQueue` for multithreading, and `pugixml` for reading XML data.

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

Related Concepts

What is PVRVk?
What is PVRAssets?
What is PVRShell?
What is PVRCamera?
What is PVRUtils?

What is PVRUtils?

About PVRUtils

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

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

There are two versions available covering Vulkan and OpenGL ES:

1. PVRUtilsVk
2. PVRUtilsGles

They provide a similar but not identical API, as they have several key differences which allow them to be better optimised for their underlying API.

PVRUtils Features

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, uploading textures to the GPU, and much more.

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

For both OpenGL ES and Vulkan, StructuredBufferView is a very important and incredibly useful tool, as it is used to determine the shader `std140` layout of buffers. It makes mapping and setting members straightforward, eliminating complicated packing/padding calculations.

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

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

- anchoring
- custom matrices
- hierarchical grouping with inherited transformations.

Other functionality provided by PVRUtils includes:

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

How to use PVRUtils

PVRUtilsVk is built on top of PVRVk, and PVRUtilsGles is built on top of raw OpenGL ES 2.0+. The main header files of PVRUtilsVk and PVRUtilsGles need to be included with `#include “PVRUtils/PVRUtilsVk.h”` or `#include “PVRUtils/PVRUtilsGles.h”` respectively. The topic *Using UIRenderer* explains how to do this.

Almost all the SDK examples use PVRUtils for some kinds of tasks. They use UIRenderer to display titles and logos. The exceptions are *HelloAPI*, *IntroducingPVRShell and IntroducingPVRVk*.

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

Related Concepts

What is PVRVk?

What is PVRAssets?

What is PVRShell?

What is PVRCamera?

What is PVRCore?

5. Making Applications Platform Independent

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

Supported platforms

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

Build system

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

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

A number of cross-compilation toolchains are also provided. These are found in the cmake/toolchains folder of the SDK. These are:

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

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

File system (streams)

The file system is abstracted through the PVRCore and PVRShell.

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

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

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

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

Windowing system

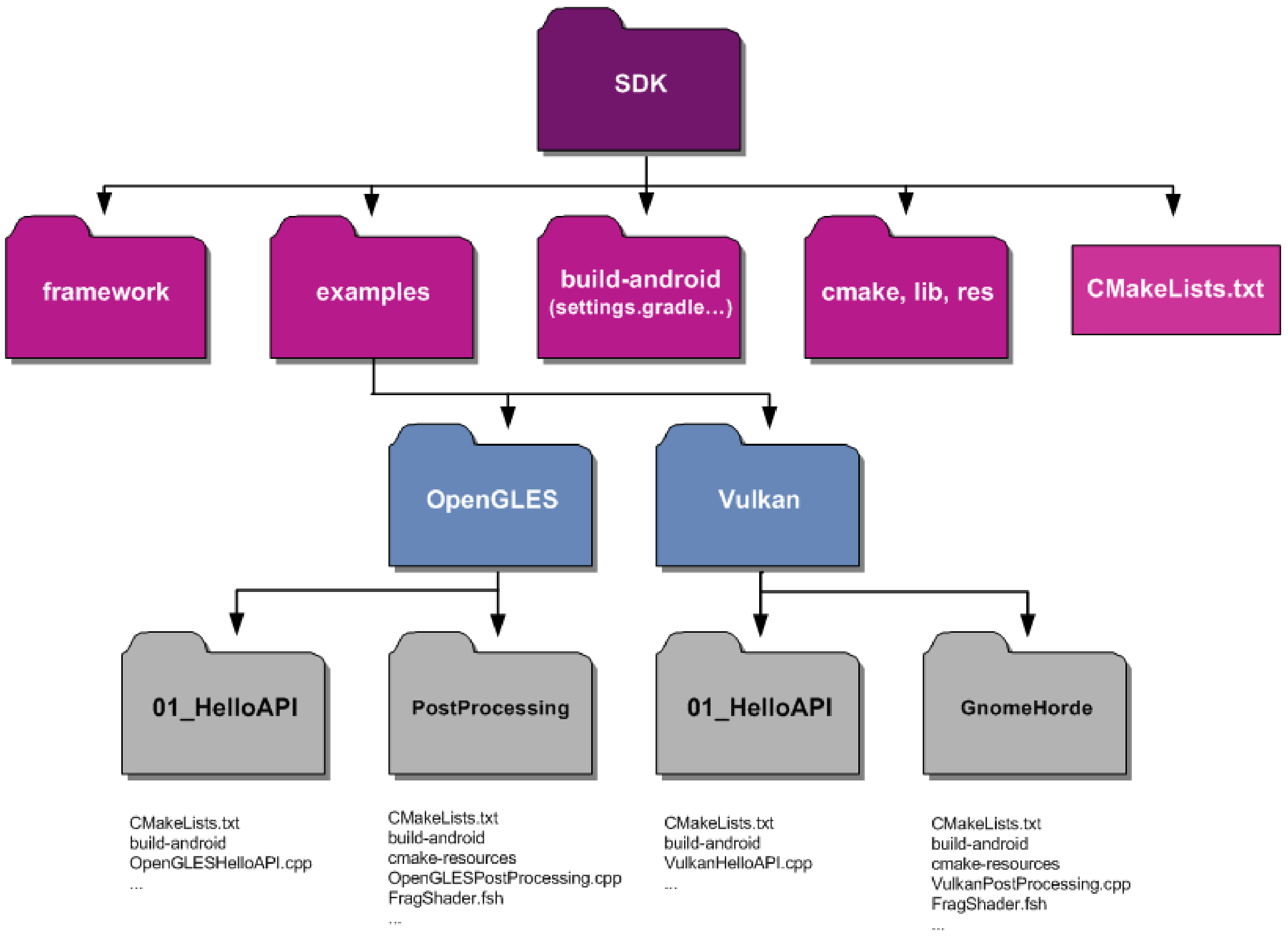
The windowing system is abstracted through PVRShell. No access is required, or given to the windowing system, except for generic `OSWindow` and `OSDisplay` objects used for context/surface creation by PVRUtils. Window creation parameters such as window size, full screen mode, and frame buffer formats are accessed by several `setXXXXXXX` PVRShell functions that can be called in the initialisation phase of the application.

6. Creating a Typical Framework 5.x Application

This section will demonstrate how to set up applications using the PowerVR Framework, and how to use the various PowerVR Framework modules individually.

The *PowerVR SDK examples* are a good resource for complete PowerVR Framework applications.

PowerVR SDK examples follow the following structure:



Setting Up the Minimum Application Skeleton in PowerVR Framework

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

1. Add the sdk folders `framework/` and `include/` as include folders
2. Link against PVRShell, PVRCore, PVRAssets, and either PVRUtilsVk (`PVRUtilsVk.lib`) or PVRUtilsGles (`PVRUtilsGles.lib`)
3. Link if needed:

- 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.

4. Include `PVRShell.h`, and `PVRUtilsVk.h` or `PVRUtilsGles.h` in the file where the application class is located.
5. 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::initApplication();
    pvr::Result::initView();
    pvr::Result::renderFrame();
    pvr::Result::releaseView();
    pvr::Result::quitApplication();
}
```

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

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

Simple Application using PVRVk/PVRUtilsVk/PVRUtilsEs

Here is the basic structure of a simple application using PVRVk, and PVRUtilsVk or PVRUtilsEs. It follows the same outline shown in *Setting Up the Minimum Application Skeleton*.

```
class MyApp : public pvr::Shell
{
    //...Your class members here...
    pvr::Result::initApplication();
    pvr::Result::initView();
    pvr::Result::renderFrame();
    pvr::Result::releaseView();
    pvr::Result::quitApplication();
}
```

This section will elaborate on what to include within each of these functions.

Writing an initApplication Function in a Typical PowerVR Framework Application

The `initApplication` function will always be called just 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.

Any changes here will override changes passed to pvr::Shell from the command line.

The initApplication function in the SDK Vulkan example, *IntroducingPVRUtils*, is shown below:

```
pvr::Result VulkanIntroducingPVRUtils::initApplication()
{
 // Load the _scene
 _scene =
 pvr::assets::Model::createWithReader(pvr::assets::PODReader(getAssetStream(SceneFileName)));

 // The cameras are stored in the file. We check it contains at least one.
 if (_scene->getNumCameras() == 0)
 {
  throw pvr::InvalidDataError("ERROR: The scene does not contain a camera");
 }

 // We check the scene contains at least one light
 if (_scene->getNumLights() == 0)
 {
  throw pvr::InvalidDataError("The scene does not contain a light\n");
 }

 // Ensure that all meshes use an indexed triangle list
 for (uint32_t i = 0; i < _scene->getNumMeshes(); ++i)
 {
  if (_scene->getMesh(i).getPrimitiveType() != pvr::PrimitiveTopology::TriangleList || _scene-
>getMesh(i).getFaces().getDataSize() == 0)
  {
   throw pvr::InvalidDataError("ERROR: The meshes in the scene should use an indexed triangle
 list\n");
  }
 }

 // Initialize variables used for the animation
 _frame = 0;
 _frameId = 0;

 return pvr::Result::Success;
}
```

Here are a few examples of the most common functions that can be called in initApplication:

- setDimensions (or setWidth and setHeight)
- setPreferredSwapchainLength
- setFullScreen
- setVsyncMode
- forceFrameTime

- `setColorBitsPerPixel`, `setDepthBitsPerPixel` and `setStencilBitsPerPixel`
- `setBackbufferColorspace`.

Writing an initView Function in a Typical PowerVR Framework Application

Generic

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

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

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

OpenGL ES

Create the EGL/EAGL context here using `pvr::createEglContext`. The context 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.

Next, 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 necessary, jump into functions to see their implementations. Not all are simple but they will usually point in the right direction.

Vulkan

Create/get the basic Vulkan objects here, including the instance, physical device, device, surface, swapchain, and depth buffer. Unless doing a specific exercise, use the PVRUtilsVk helpers, otherwise the two to three lines of code can transform into the hundreds before even starting to code application logic.

An example of how to setup `initView` in Vulkan is explained in *Example initView Setup for Vulkan*.

Setting up initView for Vulkan

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

1. Create the instance, surface, swapchain, device, and queues with PVRUtils helpers. These can be created with various levels of detail with different helpers, but usually use `pvr::utils::createInstance(…)`,

`pvr::utils::createSurface(…)`,`pvr::utils::createDeviceAndQueues(…)`, and `pvr::utils::createSwapchainAndDepthStencilImageView(…)`.

2. Create `DescriptorSetLayout` objects depending on the needs of the application using `pvrvk::Device::createDescriptorSetLayout(…)`.

3. Create `DescriptorSet` objects based on the layouts using `pvrvk::Device::createDescriptorSet(…)`.

4. Create memory objects such as `Buffer` and `Image` using `pvrvk::Device::createBuffer` and `pvrutils::uploadTexture(...)`.

5. Populate the `DescriptorSet` objects with the memory objects using the function `pvrvk::Device::updateDescriptorSets(…)`.

6. Create `PipelineLayout` objects using the descriptor layouts using `pvrvk::Device::createPipelineLayout(…)`.

7. Configure `PipelineCreateInfo` objects (amongst others using those shader strings) and create the pipelines.

8. Configure the `VertexAttributes` and `VertexBindings` usually using helper utilities such as `pvr::utils::CreateInputAssemblyFromXXXXX(…)`. This will automatically populate the `VertexInput` area of the pipeline based on our models.

9. Create the pipelines: `myDevice->createPipeline(…)`.

Creating Textures using PVRUtilsVk

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

```
_deviceResources->vmaAllocator =
                pvr::utils::vma::createAllocator(pvr::utils::vma::AllocatorCreateInfo(
                                                                _deviceResources->device));
```

2. Create the memory objects such as buffers, images and samplers.

3. Get streams to the texture data on disk: `getAssetStream(…)`:
   - To access CPU-side texture data, load the images into `pvr::assets::Texture` objects using `pvr::assets::textureLoad()`, then upload them to the GPU as `pvrvk::Image` and `pvrvk::ImageView` using `pvr::utils::uploadImageAndView`.
   - Otherwise, merge the two steps using `pvr::utils::loadAndUploadImageAndView`.

```
 pvr::Stream::ptr_type stream = assetProvider.getAssetStream(model-
>getTexture(i).getName());
 pvr::Texture tex = pvr::textureLoad(stream, pvr::TextureFileFormat::PVR);
 images.push_back(pvr::utils::uploadImageAndView(device, tex, true, uploadCmdBuffer,
 pvrvk::ImageUsageFlags::e_SAMPLED_BIT,
 pvrvk::ImageLayout::e_SHADER_READ_ONLY_OPTIMAL, &allocator, &allocator,
 pvr::utils::vma::AllocationCreateFlags::e_DEDICATED_MEMORY_BIT));
```

For a proper method to do this asynchronously in a multithreaded environment, see the *Multithreading* example.

Creating Buffers in initView using PVRUtilsVk

1. Use the shortcut utilities `pvr::createBuffer()` for creating UBOs or SSBOs.

   ```
   ubo = pvr::utils::createBuffer(device, uboView.getSize(),
    pvrvk::BufferUsageFlags::e_UNIFORM_BUFFER_BIT,
    pvrvk::MemoryPropertyFlags::e_HOST_VISIBLE_BIT,
                           pvrvk::MemoryPropertyFlags::e_HOST_COHERENT_BIT |
                           pvrvk::MemoryPropertyFlags::e_DEVICE_LOCAL_BIT, &allocator);
   ```

2. Connect the `StructuredBufferView` to the actual buffer with `StructuredBufferView->pointToMappedMemory()`.

   To automatically layout buffers that will have a shader representation (UBOs or SSBOs), use `StructuredBufferView`. This is an incredibly useful class. The UBO/SSBO configuration needs to be described, and it will then automatically calculate all sizes and offsets based on STD140 rules. This includes array members, nested structs, and so on, automatically making it possible to both determine size, and set individual elements or block values.

   ```
   uboView.pointToMappedMemory(ubo->getDeviceMemory()->getMappedData());
   ```

Setting up Objects in initView using PVRUtilsVk

1. Update the descriptor sets with the actual memory objects such as (buffers, images).

   This might sometimes need to be done in `renderFrame()` for streaming resources.

2. Create command buffers, synchronisation objects and other app-specific objects.

   a) Usually, one command buffer is needed per swapchain (backbuffer) image. Use `getSwapChainLength()` and `logicalDevice ->createCommandBufferOnDefaultPool()` for this.

   b) In a multithreaded environment at least one command pool per thread should be used. Use `context->createCommandPool()` and then `commandPool->allocateCommandBuffer()` on that thread.

      Caution: Do not use a command pool object from multiple threads, instead create one per thread. `pvrvk::CommandBuffer` objects track their command pools and are automatically reclaimed. Be careful to release them on the thread their pool belongs to, or to externally synchronise their release with their pool access.

3. Use a loop to fill them up as follows:

   Note: This is a very simple case.

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

     Note: The swapchain image index can be retrieved using `getSwapchainIndex()`.

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

Writing a renderFrame Function in a Typical PowerVR Framework Application

Generic

`renderFrame` gets executed by the shell once for each frame. This is where logic updates happen. In common scenarios, this may end up just being updates of uniforms, such as transformation matrices and updated animation bones.

OpenGL ES

In OpenGL ES, this function behaves as expected - running application logic and OpenGL ES commands as with any application with a per-frame main loop.

Note: Remember to call `eglContext->swapBuffers()` when rendering is finished or swap the buffers manually if not using PVRUtilsGles.

Vulkan

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

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

Remember: Submit the correct command buffer that corresponds to the current swapchain 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 recommended synchronisation schemes.

If application logic determines it is time to signal an exit, return `pvr::Result::ExitRenderFrame` instead of `pvr::Result::Success`, or call `exitShell()` from any function on the main thread.

Writing a releaseView Function in a Typical PowerVR Framework Application

`releaseView` will be called once when the window surface is going to be lost, but before it happens. The window surface is lost when the application loses focus or the window is going to be torn down. If the application is about to lose the window/API/surface or restarts, `releaseView` will be called before `initView` is called again.

OpenGL ES

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

Vulkan (PVRVk)

For PVRVk, make use of RAII (Resource Acquisition Is Initialisation) - release any API objects specifically created, by releasing any references that are held. Release can be achieved by calling `reset()` on the smart pointer itself, or if objects are held in containers, by emptying those containers.

Never attempt to release the underlying object. If an API to do so exists, it will mean something different and not behave as expected. Access the `reset()` function with the dot `'.'` (not arrow `'->'`) operator on a PVRVk object. Do not explicitly delete an API object. All objects are deleted when no references to them are held.

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

Note that RAII deletion of PVRVk and other Framework objects is deterministic, not garbage-collected. Objects are always deleted exactly when the last reference to them is released. Also, note that objects may hold references to other objects that they require, and these object chains are also deleted when the last reference is released. For example, command buffers hold references to their corresponding pools, descriptor sets hold references to any objects they contain. This is one of the most important features of PVRVk.

There is no particular required deletion order imposed. Keep any objects needed to reference and release them when no longer required.

It is recommended that C++ destructors are used.

This is how the Framework examples handle destruction:

A struct/class is defined 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) the dependency graph unfolds itself, and destructors are called in the correct order.

Important:

There is a very important caveat to the automatic releasing of objects. While objects like command buffers and descriptor sets hold references to objects added to them and keep them alive, this is not true for command queues.

Command queues do not explicitly report when command buffers are executed, so it would have been very difficult to automatically report to a PVRVk command queue when the GPU has finished executing a command buffer, hence command queues do not hold references to command buffers. Therefore, it is necessary to explicitly track the execution status of command buffers and only release them after a command queue has finished executing them. This is one of the reasons `waitIdle()` is recommended before starting the teardown process.

Writing a quitApplication Function in a Typical PowerVR Framework Application

`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 and database connections. 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.

Using PVRVk

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

For simplifying the operations and tasks themselves see *PVRUtils*.

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

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

- Vulkan functions whose first parameter is a Vulkan object become member functions of the class of that object. For example, `VkCreateBuffer` becomes a member function of `pvrvk::Device`, so use `myDevice->createBuffer()`.
- Functions without a dispatchable object as an input parameter object remain global functions in the `pvrvk::` namespace. For example, `pvrvk::CreateInstance(…)`.
- Simple structs like `Offset2D` 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 populated automatically.
- Vulkan objects are wrapped in C++ reference-counted classes providing them with a proper C++ interface. Usage remains the same and the `Vk…` prefix is dropped. For example, `VkBuffer` becomes `pvrvk::Buffer`.
- All enums are shadowed by C++ scoped enums (enum class TypeName)
  - The `VK_ENUM_`*`TYPE_NAME_`* prefix of enum members is dropped and replaced by e_. In many cases, for example after dropping `VK_ FORMAT_2D`, this would become `pvrvk::Format::2D` which is illegal as it starts with a number.
  - Flags/Bitfields are used like every other enum as bitwise operators are defined for them. `VkCreateBufferFlags` and `VkCreateBufferFlagBits` become `pvrvk::CreateBufferFlags` and are directly passed to corresponding functions.

Using PVRUtilsVk

Setting up the simplest Vulkan application

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

This is particularly evident in the simplest tasks that require CPU-GPU transfers, particularly when loading textures. Even this requires staging buffers and special synchronisation, so it can seem daunting at first.

For something as simple as initialisation, the process involves:

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

Each of these steps has at least a few dozen lines in raw Vulkan, with often quite involved logic.

PVRUtilsVk can simplify this process

Initially, the PVRUtilsVk helpers can help automate most, if not all of these tasks, without taking away any of the flexibility. If a fringe case cannot be done with the helpers then they can be skipped and the functions manually called instead, as required. They do not introduce unnecessary intermediate objects, so mixing and matching as needed is fine.

Many of the SDK examples provide a good introduction to PVRUtilsVk. The helpers are usually reasonably simple functions, though in some cases they can get longer if the task is more complex.

Example: Texture Loading

One of the more important processes of a graphics application, which is automated by PVRUtilsVk, is texture loading. The particulars of it can be daunting.

The basic steps required to upload a texture are:

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

Or, alternatively, PVRUtilsVk provides the `textureUpload` function which can be used instead of all of this.

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

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

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

Using PVRUtilsEs

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

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

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

Using UIRenderer in PowerVR Framework

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

- In Vulkan, UIRenderer refers to and is therefore compatible with a specific `RenderPass`, and UI Rendering commands are packaged and recorded into a `pvrvk::CommandBuffer`.
- In OpenGL ES, the OpenGL state is recorded, rendering commands are executed inline, and then the OpenGL ES state is restored.

An example of using UIRenderer can be found in the SDK. *IntroducingUIRenderer* demonstrates how to create a Star Wars intro scrolling marquee effect using the PowerVR Framework and specifically UIRenderer.

Initialising UIRenderer

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.

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

Introduction to UIRenderer Sprites and Layouts

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

The layout itself and the positioning are **not** done by the sprite interface class. Instead, subclasses of `pvr::ui::Sprite` 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, providing methods for anchoring to the corners, offsetting by X/Y pixels, rotations, scaling, and so on.

Note:

- Position refers to the position of a sprite relative to the component it is added to. For example, the top-left corner of the UIRenderer's surface.
- 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 around the anchor.
- Scale is the sprite's size compared to its natural size. Scales are relative to the anchor.
- Offset is a number of pixels to move the final position by, relative to the parent container, and relatively to the finally scaled sprite

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 using `pvr::ui::Group`. Groups are hierarchical containers of other components including other groups. For example, there could be a PixelGroup representing a panel with components. Text sprites and an image could be added and positioned to this group, which is then added into a `MatrixGroup`. Finally, the `MatrixGroup` could be transformed with a projection matrix to display it in a Star Wars intro scrolling marquee way. This use case is shown in the *IntroducingUIRenderer* example.

Setting Up a Simple UIRenderer Layout

A simple example is the best way to better understand UIRenderer layouts. This example will print a scrolling marquee, Star Wars-intro like, with some icons at the corners of the marquee, scrolling text, and some text in the corners of the screen. This starts from being a single point in the centre of the screen and blows up to take up the entire screen.

The steps are as follows:

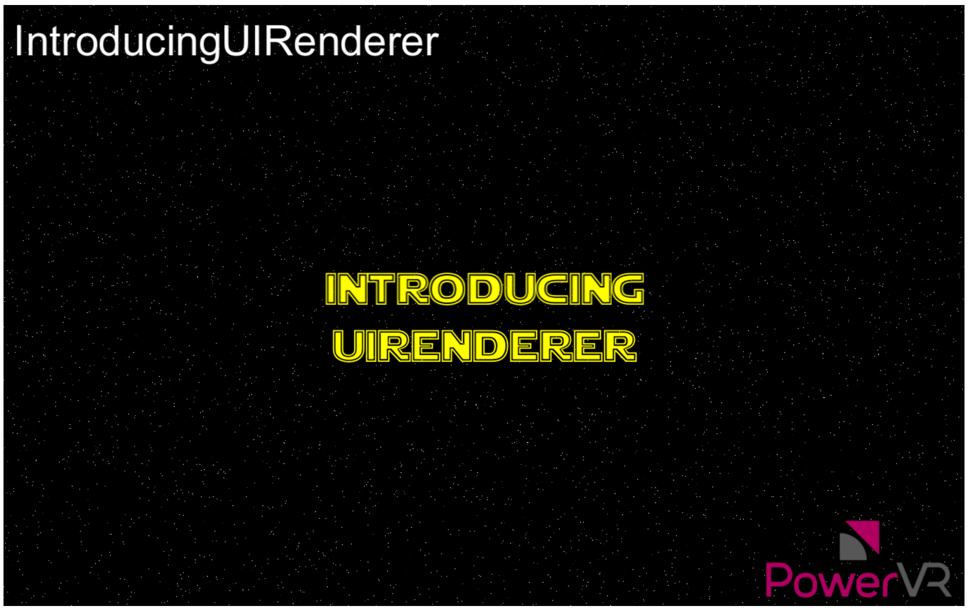
1. Create text for the lines marquee, image for the icons, and put those in a `MatrixGroup`.
2. Put this `MatrixGroup` in a `PixelGroup`.
3. Set the anchor of the marquee texts to Centre, and calculate the fixed `PixelOffset` of each text based on line spacing. On each frame add a number to each text's `PixelOffset.y` to scroll them.
4. Anchor the `TopLeft` of the top left symbol to the `TopLeft` of the matrix group. Anchor the `BottomRight` of the bottom right symbol to the `BottomRight` of the `MatrixGroup`, and so on.

5. Calculate a suitable transformation with a projection to nicely display the marquee relative to its containing group. The marquee and symbols will move as one item here.

6. Anchor the corner text with the same logic, potentially offsetting it so it does not touch the borders.

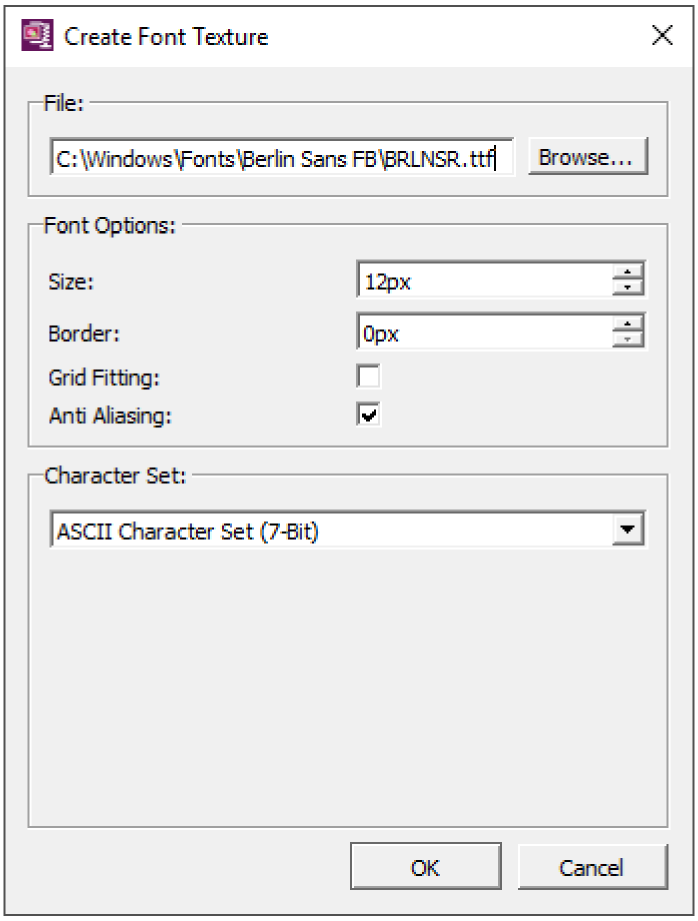
7. Finally, centre the `PixelGroup`, and set its scale to a very small number. As it increases frame by frame, the group takes up the whole screen when it reaches the value of one.

To see this in action, take a look at the PowerVR SDK demo, *IntroducingUIRenderer*.



Creating Fonts for Use With UIRenderer

If any other font besides the default (Arial TrueType font) is required, use PVRTexTool's Create Font tool to create a `.pvr` file. This is actually a `.pvr` texture file that contains metadata to be used as a font by `UIRenderer` and includes the positions of each character, kerning, and so on.


Create Font Texture

File:

C:\Windows\Fonts\Berlin Sans FB\BRLNSR.ttf

Font Options:

Size: 12px

Border: 0px

Grid Fitting: ☐

Anti Aliasing: ✓

Character Set:

ASCII Character Set (7-Bit)

Using UIRenderer Sprites

Rendering and using sprites in UIRenderer is really easy:

1. Create any sprites that are required using `UIRenderer->createFont(…)`, `createImage(…)`, `createText(…)`, `createMatrixGroup(…)`, and so on.

2. Set up sprites with `setColour()` and `setText()`.

3. Create hierarchies by adding the sprites to groups.

4. Lay out sprites/groups using `setPosition()`, and so on.

5. Call `render()` on the top level sprite.

   Warning: Never call `render()` on sprites that are themselves in containers - only call render on the item containing other items. Otherwise, any sprite on which `render()` is called will be rendered relative to the top level (screen), not taking into account any containers to which it may belong.

Rendering UIRenderer Sprites

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

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

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

The steps required to render a sprite with UIRenderer are:

1. Call `uiRenderer->beginRendering(…)`.

   In Vulkan, this takes a SecondaryCommandBuffer where the rendering commands will be recorded.

   Note:

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

2. Call the `render()` method on all top level sprites that will be rendered - the containers, not the contents.

   Do not call render on a component that is contained in another component, as the result is not what would be expected. It will be rendered as if it was not a part of the other component. So if there are two images and a group that contains ten texts and another two images, only three `render()` calls are necessary.

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

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

For Vulkan, the secondary command buffer passed as a parameter in the `beginRendering()` command will now contain the rendering commands such as bind pipelines, buffers, descriptor sets, and draw commands for the UI. It must be submitted inside the render pass and sub pass which were used to initialise the `UIRenderer`, or a compatible render pass.

UIRenderer Recommendations

There are a couple of important points to consider when rendering with UIRenderer.

1. For Vulkan, reuse the command buffer of the sprite. Even if the text or the image is changed, it is unnecessary to re-record the command buffer unless the actual

sprite objects rendered changed. UIRenderer uses indirect drawing commands to make it possible to even change the text without re-recording.

- For example, if the colour and position of a sprite is changed, the command buffers do not need to be re-recorded.
- If the actual text of a text element is changed, the command buffers do not need to be re-recorded.
- If text is removed from a container, or new text is added, the command buffers do need to be re-recorded.

2. Only call `commitUpdates()` when all changes to a sprite are done. In some cases, especially if text length increases, this operation can become expensive. For example, VBOs may need to be regenerated.

Rendering without PVRUtils

The Framework is modular, and libraries are often used separately from the others.

There are many combinations which can be used, but here are a few of the most common choices:

- Make use of everything – that's the official recommendation. Derive the application from `pvr::Shell`, use PVRUtils and PVRVk for rendering, load and use the assets with PVRAssets, and use PVRUtils for rendering 2D elements and multithreading. Most PowerVR SDK examples use this approach, making the best use from all the power of the PowerVR Framework.
- Forgo using top-level libraries such as PVRUtils or 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.
- Use raw Vulkan, without PVRVk. This is not recommended. It is better to use PVRVk or another Vulkan library to help. Check out the Vulkan *IntroducingPVRShell* or *HelloAPI* examples to get a taste of 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 modern language bindings can help. These gains are practically for free.
- Forgo using even PVRAssets. In this case, the only thing that is being used from the framework is PVRShell. In this scenario, another scene graph library will need to be used or written, and the only thing used will be the platform abstraction of PVRShell. Support classes will need to be written for everything. This is done in *IntroducingPVRShell*.

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.

Semaphores in PVRVk

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

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

The basic use of a semaphore is as follows:

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

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

The above 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 instance 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.

Fences in PVRVk

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

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

Events in PVRVk

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.

Recommended Typical Synchronisation for Presenting in Vulkan

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

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

- One set of fences, ensuring command buffers have finished executing before resubmitting them.
- One set of semaphores connecting `Acquire` to `Submit`.
- One set of semaphores connecting `Submit` to `Present`.
- Tracking of a linear progression of frames (`frameId`) and the swapchain image `id` separately.

See most *PowerVR SDK examples* for the implementation.

Additional synchronisation can be inserted inside the `Submit` phase without complicating things too much.

8. Debugging 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.

Exceptions

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

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

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

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

OpenGL ES

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

Vulkan/PVRVk

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

The most important consideration here are the Vulkan Layers.

Layers

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

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

1. Inspect the log for any errors. This includes layers output on platforms that have them.
2. Inspect the Vulkan layers and fix any issues found, until they are clear of errors and warnings.

3. Continue with other methods of debugging such as the API dump layer, CPU or GPU debuggers, PVRCarbon, or any other method suitable to the error.

9. Overview of Useful Namespaces

Namespace	Description
`::pvr`	Main namespace. Primitive types and foundation objects, such as the `Stream` and `Texture` classes and some interfaces are found here.
`::pvrvk`	The classes and global functions of the PVRVk library. This is where to find any API objects that need to be created such as buffers, (GPU) textures, CommandBuffers, GraphicsPipelines, or RenderPasses. Also functions with no corresponding object, like `pvrvk::createInstance`.
`::pvr::utils`	This extremely important namespace is the location for automations for common complex tasks. For example, automating the creation of a swapchain, uploading a texture, or creating a skybox.
`::pvr::assets`	All classes of the PVRAssets library are found here: Model, Mesh, Camera, Light, and Animation.
`::gl`	OpenGL ES bindings. Only OpenGL ES function pointers are found here.
`::vk`	Vulkan bindings. Only Vulkan function pointers are found here. Use `pvrvk::` instead.
`::?::details` `::?::impl`	Namespaces for code organisation. Not required.

10. Contact Details

For further support, visit our forum:

http://forum.imgtec.com

Or file a ticket in our support system:

https://pvrsupport.imgtec.com

For general enquiries, please visit our website:

http://imgtec.com/corporate/contactus.asp