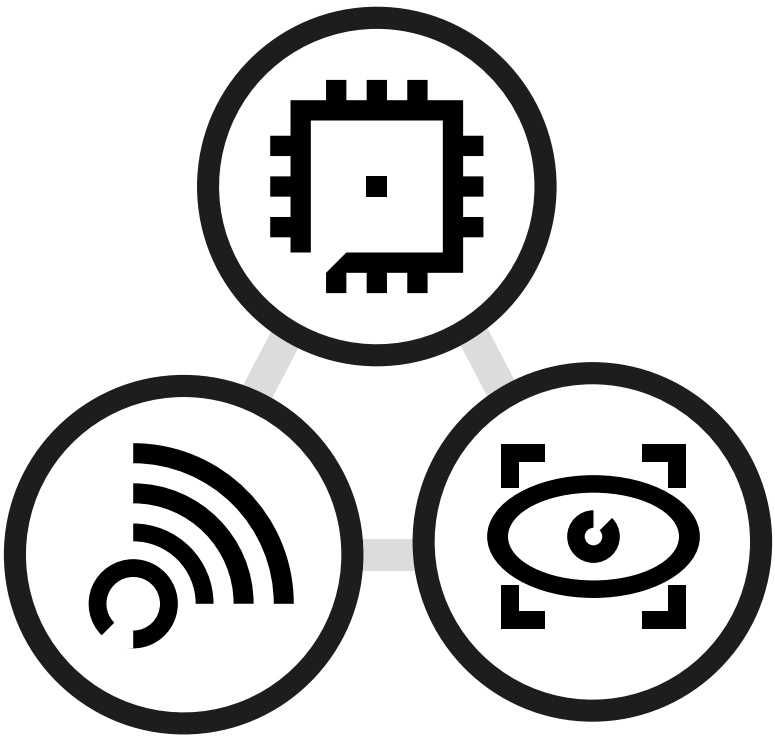
PowerVR Framework Tips and Tricks

Revision: 1.0
12/05/2021
Public



Published: 12/05/2021-19:14

Contents

1. Introduction to PowerVR Framework Tips and Tricks

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.

For more information about the PowerVR Framework, please see the *PowerVR Framework Development Guide*.

This document contains helpful tips and tricks for getting the most out of the PowerVR Framework.

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. Models, POD & GLTF with PVRAssets

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

Models, meshes, cameras, lights, animation

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

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

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

Nodes

Nodes are the building blocks of the scene and describe its hierarchy. Each node is part of a tree structure, with parent nodes (except the root node), and carries a transformation, and a reference to an object such as a mesh, camera or light. The transformations are applied hierarchically. The transformations, in general, are animated and dependent on the current frame of the scene.

Nodes are usually accessed through their indices. To make accessing objects easier, the nodes are sorted by object types, in the order:

1. `Mesh`
2. `Camera`
3. `Light`.

Always be wary when trying to access a node or its underlying object. When trying to iterate the meshes, for example, to get VBOs, attributes, textures, and so on, always call `getMesh(…)`, but when trying to display the scene, iterate `MeshNodes()`. This is a subtle difference, but it pays to consider the mesh like a blueprint of an object, while the mesh node is referencing an instance of that object.

Note: Some useful methods

- `getMesh(meshIndex)` - returns the `Mesh` with id `meshIndex`.
- `getMeshNode(nodeIndex)` - returns the `Node` with id `nodeIndex` (which may point to any `Mesh`).

- `getMeshNode(nodeIndex)->getObjectId()` - returns the `Mesh` that is referenced by the node with index `nodeIndex`.
- `getCamera(id, [output camera parameters])` - returns the parameters of the camera with Camera ID `id`.
- `getLight…` - returns the parameters of the light with Light ID `id`.

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 with or even hold the model. For example, the position and animation of objects might come from application logic and not the model.

The Framework deals with this using the `std::shared_ptr` 's shared refcounting feature. Call `Model->getMeshHandle()` to get a `s<Mesh>` that will be functional 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.

POD & GLTF

To load models, there are readers available for Imagination's .POD (*PVRGeoPOD*) and Khronos' .gltf files.

Semantics

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

For example, in the PFX format it is possible to annotate the attribute `myVertex` with the semantic `POSITION`, so that PVRShaman knows 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.

Currently, semantics in the `Model` class have been made completely flexible, allowing any vertex attribute or material attribute to be annotated with any semantic, allowing the reader (POD or GLTF currently) to define semantics, and then it is up to the application to use these semantics.

In the PowerVR SDK, the convenient PVRShaman semantics are still being used, but it is up to the two sides (the model and the renderer) to decide on the semantics and what they mean.

3. Input Handling Tips and Tricks

PVRShell simplified (mapped) input

Nearly all SDK Examples use Simplified Input. This is a model 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 one of the actions is performed that maps to a simplified event.

Lower-level input

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

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

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

Input Events on Desktop

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

Input Events on Android

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

Input Events on iOS

INPUT EVENT	How to trigger on iOS
Action1	Touch centre of screen
Action2	Touch left 30% of screen
Action3	Touch right 30% of screen
Left/Right/Up/Down	Swipe Left/Right/Up/Down
Quit	"Home" key

4. Render pass/Pixel Local Storage (PLS) Strategies

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

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

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

Setting the LoadOp and StoreOp 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 render pass, what is necessary to do with whatever contents the frame buffer 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 "the entire scene will be rendered anyway, so it doesn't matter, don't load it"
- **Preserve** means "the scene will be rendered incrementally, using whatever is already in the framebuffer, so the contents of it need to be preserved."

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

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

In OpenGL ES, the situation is very similar. When `glClear` is called at the start of a frame, or `glInvalidate` depth/stencil before swapping, the driver may be allowed to discard the contents of the frame buffer / depth buffer before the next frame.

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

Recommendations for LoadOp

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

Conversely, for OpenGL ES:

- `glClear` both colour and depth at the start of the frame.

Recommendations for StoreOp

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

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

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

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

In short:

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

Subpasses/Pixel Local Storage

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

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

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

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

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

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

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

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

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

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

5. PowerVR Framework Frequently Asked Questions

Here are the answers to the most frequently asked about the PowerVR Framework.

For any further questions please feel free to *get in touch* with us.

Which Header Files Should I Include?

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

Vulkan

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

OpenGL ES

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

If PVRCamera is required

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

Which Libraries Should Be Linked Against?

Usually, the libraries needed by the Framework should be added through CMake. This is very straightforward, and involves including the `CMakeLists.txt` from the SDK, and relevant targets for the Framework modules. Our *SDK examples* show how to do this.

For a different build system, for example to link to pre-built binaries, build the Framework libraries with CMake and add the library outputs from wherever they were built. The libraries themselves are:

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

If CMake is used, the dependencies are transiently included through the CMake targets, together with include folders and everything else necessary. If some other way is used, the dependencies are as follows:

- PVRCore: None
- PVRShell: PVRCore
- PVRAssets: PVRCore

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

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

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

Does Library Link Order Matter?

If using CMake, library link order does not matter as CMake and the libraries take care of it.

If using a different build system, it is system dependent. For Windows/OSX/macOS, it does not matter. For Android and Linux, it may matter for some underlying compilers.

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

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

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

Are There Any Dependencies to be Aware of?

In general, it is recommend that one of the `CMakeLists.txt` and/or gradle scripts should be used as a base, because this takes care of all dependencies.

Otherwise, to start from scratch:

- All external dependencies are downloaded and/or built as part of CMake and put in the `[SDKROOT]/external` folder. Pre-downloaded versions are usually bundled as well. External dependencies are as follow:
  - GLM (PVRCore and everything else) for vector maths.
  - PugiXML for XML parsing
  - GLSLang is used by PVRUtilsVk to allow online shader compilation
  - `moodyCamel::concurrentQueue` for multithreaded producer-consumer queues.

- The `[SDKROOT]/include` folder must be added as an include file search path. It contains the API header files and any other headers that are used. As well as the stock Khronos headers for most APIs, it contains PowerVR SDK's own custom `DynamicGles.h`, `DynamicEgl.h`, and `vulkan_wrapper.h` bindings. This is added automatically if including any Framework CMake target.
- The `[SDKROOT]/framework` folder must be added as an include file search path to access the Framework's headers. This is added automatically if including any Framework CMake target.
- The `[SDKROOT]/lib [PLATFORM…]` folder may contain library dependencies of the project files. For example, the PVRScope libraries are located there. This is added automatically if including any Framework CMake target.
- The Framework library files will be either wherever they were built, or by default, prebuilt would be in `[SDKROOT]/framework/lib/[PLATFORM…]` . As the PowerVR libraries do NOT have a C interface as they expose C++ classes in their API, it is strongly recommended to not use prebuilts. Instead, always build them through CMake with common compilation options with the application.

What About Linking Against OpenGL ES, or Vulkan, respectively?

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

DynamicGLES takes care of it by dynamically linking OpenGL ES.

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

What are the Strategies for Command Buffers? What about Threading?

The Vulkan multithreading model means that developers are free to generate command buffers in any thread, but they should be submitted in the main thread. While it may be possible to do it differently, this is normally both the optimal and the desired way, so there is no need to be concerned with cases of submissions for multiple threads.

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

There are numerous ways that an application can be structured, but some patterns will emerge.

Single command buffer submission, multiple command secondary command buffers

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

Multiple parallel command buffers, submitted once

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

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

Multiple parallel command buffers, submitted multiple times

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

How Are PVRVk Objects Created?

Usually, most PVRVk objects with a Vulkan equivalent such as buffers, textures, semaphores, descriptor sets, and command pools, 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, defaults are provided for as many of the parameters/create info fields as is feasible.

Remember that some creations are really allocations from pool objects. For example, there is no `device->createCommandBuffer()`, instead call `commandPool->allocateCommandBuffer()`.

These are usually hinted by the name: instead of create/destroy for objects created on a device, there is allocate/free for objects allocated on a pool.

How Are PVRVK Objects Cleaned Up?

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

their smart pointers reset, as they are immediately destroyed when their reference count goes to zero.

Other objects may sometimes hold references to them. Most notably, command buffers and descriptor sets hold references to objects they are using.

How Is a UIRenderer Cleaned Up?

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

Other objects may sometimes hold references to them. Most notably, command buffers and descriptor sets hold references to objects they are using.

Do Any API Objects Need to be Manually Kept Alive?

Mostly, they do not.

- Command buffer and descriptor set 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, image view objects will keep alive the image objects underneath, pipelines will keep their pipeline layouts alive, and so on.

There are some notable exceptions:

- Command pool and descriptor pool objects. These objects are not kept alive by their command buffer and descriptor set objects. This means that they must be kept alive until no longer required, and command/descriptor pools must be destroyed only after any of their objects are released.
- Command buffer objects must be kept alive as long as they are executing (rendering) which generally means waiting on their associated fences. This rule, in conjunction with the previous one, means that destruction must happen in the following order:
  1. Frame done.
  2. Commands released.
  3. Pools released.

How Are Files/Assets/Resources Loaded?

The PowerVR Framework uses the stream abstraction for data. `Stream` is an abstract class and can be subclassed into a concrete type of stream. Use this to determine your own.

There are four types of concrete stream classes in the SDK:

- `BufferStream` is a stream that points to a block of RAM. It is used to read and write to memory without a persistent storage. Any block of memory can be pointed to by the buffer, and this class is used when a custom part of memory needs to be passed to a function requiring a `Stream`.
- `FileStream` is a file.
- `AndroidAssetStream` is a stream accessing an .apk's assets.
- `WindowsResourceStream` is a stream accessing a Windows executable's embedded resources.

There are two ways to make use of these.

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:

- A `FileStream` for the determined path, in any of the following folders:
  - The current directory.
  - The directory where the application executable is.
  - The directory `Assets_[executable name]`.
  - Any other paths added through `pvr::Shell::addReadPath`.
- `AndroidAssetStream` will look for an asset of the specified name in the .apk's assets folder.
- `WindowsResourceStream` will look for an embedded resource of the specified name in the executable.

The parameter passed to `getAssetStream` is usually a raw filename, or a relative path, but can be anything depending on the use. The convention used is that is a raw executable name, and assumes that window resources are named with this path. Files will be searched in the current folder of the executable and in the `Assets_[ExecutableName]` subfolder. Functions that need some kind of data to create an object, most notably, asset load functions, will take streams as input. The only exceptions are PVRVk Shaders - these are passed as raw bytes to avoid a PVRVk dependency on PVRCore.

Directly create the stream

The other option is to directly create a `FileStream`/ `BufferStream`/ `WindowsResourceStream` to load a resource. In order to do this, the resource must be of the correct type and be supported on the platform. For example, a `FileStream` will work fine for Windows and Linux, but on Android it may not be what is expected. Use an `AndroidAssetStream` to read from an .apk's assets, while a `FileStream` will read and write to the application's temporary location. Finally, in order to pass custom data to a stream function, for example, a `std::string` into a function that requires a stream, the data can be trivially wrapped using the constructor of a `BufferStream`.

Defining Buffer layouts with StructuredBufferView

Calculating buffer layouts

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

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

The StructuredBufferView

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

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

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

GLSL

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

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

CPU side

An easy-to-use interface is provided to define the `StructuredBufferView`. Using C++ initialiser lists, a compact JSON-like constructor has been created that allows the developer to easily express any structure.

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

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

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

```
// SETTING VALUES
void* bones = gl::MapBufferRange(GL_SHADER_STORAGE_BUFFER, 0, ssboView.getSize(),
 GL_MAP_WRITE_BIT);
int32_t boneCount = mesh.getNumBones();
ssboView.getElement(_boneCountIdx).setValue(bones, &boneCount);
auto root = ssboView.getBufferArrayBlock(0);

for (uint32_t boneId = 0; boneId < numBones; ++boneId)
{
    const auto &bone = _scene->getBoneWorldMatrix(nodeId, mesh.getBatchBone(batch, boneId));
    auto bonesArrayRoot = root.getElement(_bonesIdx, boneId);
    bonesArrayRoot.getElement(_boneMatrixIdx).setValue(bones,
    glm::value_ptr(bone));
    bonesArrayRoot.getElement(_boneMatrixItIdx).setValue(bones,
                                glm::value_ptr(glm::inverseTranspose(bone))));
}

gl::UnmapBuffer(GL_SHADER_STORAGE_BUFFER);
```

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

6. 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