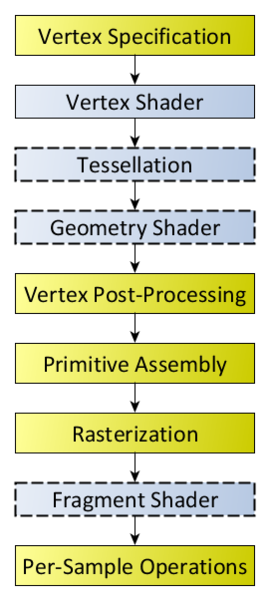
**From OpenGL to Vulkan and the framework**

This document is the story of me learning the PowerVR framework. By writing and sharing my experience I hope to help others to get up and running quickly with the PowerVR framework. To be fair, the learning phase starts with a steep slope. This is especially true for somebody with fresh knowledge in computer graphics. Once this first pass overcome, you benefit from the massive control provided by the Vulkan specification and the relative ease of using the framework. All in all, mastering all aspects of the framework is more than worth the effort. Let me present the questions I wanted answers for when I first started.

1. Why should I move from OpenGL to Vulkan and the framework?
2. Is there a super entity, something that can be assimilated to the OpenGL state machine?
3. Has the graphics rendering pipeline changed from this:



Source: <https://www.opengl.org/wiki/File:RenderingPipeline.png>, Accessed: 23 November 2016

1. How do I set my geometries? How do I fill my vbos, ibos, ubos…
2. Many examples create a DeviceResource struct/class. Why not have a dedicated class in the framework pvr::api::DeviceResource? What belongs inside it, what does not?
3. What does a descriptor set describe? How do descriptorSetLayout relate to descriptor sets?
4. What are render passes? How to set them?
5. What is a command buffer? How do I use it?   
   Do they allow rendering passes to be executed concurrently?
6. What about pools? Can I ignore them if I plan on working in a single thread?
7. What are TextureView/BufferView’s, do they contain the texture, buffer raw data (image, vertex data)
8. What language should I write my shaders in?
9. How does a simple application go about drawing a triangle on the screen (at a very high level view).

If do share some or all these questions with me, read on.

**OpenGL typical workflow**

**OpenGL ES**

* CreateContext (+load functions)
* Initialize buffers: Supply vertex data, lights, cameras, transforms
* Setup the pipeline state:
  + Primitive type
  + Culling
  + Depth/Stencil testing
  + Create Shaders
  + Create Program
  + Bind Textures, vertex data, uniforms
  + Use Program
* Render
* Release resources

**General advice**

When starting with the PowerVR framework make sure to have a web browser to view the doxygen documentation. You can find it in your installation folder at:

…Imagination\PowerVR\_Graphics\PowerVR\_SDK\SDK\_2016\_R2\Documentation\SDKBrowser\Framework.

Navigating through the website not only teaches you the framework’s API (namespaces, classes, and methods etc.), but also the structure and relationships between the different functional entities: PVRCore, PVRAssets, PVRApi ... This is a good way to get familiar with the different namespaces and understand their respective purposes.

**How is the framework structured?**

The public API consists in all information (struct, class, methods…) documented in headers which filename start with PVR\*\*\*\*\*\*\*.

These PVR\*\*\*\*\*\* files do not contain the actual header definition; they are used for doxygen doc.

**Work with RefCountedResources**

They are the smart pointers made in PowerVR.

They come in mainly 3 different flavours:

* The strong pointer: RefCountedResource: Shares ownership of the data.
* The weak pointer RefCountedWeakReference: does not participate in the pointed class lifecycle.
* The embedded pointer: EmbeddedRefCountedResource: Embeds refcounting capabilities inside the class definition.

For allocating objects that necessitate refCounting:

**Prefer** the following two steps (also called the fast path):

1. RefCountedResource<pvr::api::Model> myModelHandle;
2. myModelHandle.construct();

**To**:

* + 1. RefCountedResource<pvr::api::Model> myModelHandle(new pvr::api::Model);

**Understanding how models work**

* A model is a tree structure of nodes. Nodes are generic elements of that tree.
* Nodes reference meshes, cameras or lights.
* The framework expect nodes to be sorted : first you will find the meshes, then the cameras, then the lights
  + This is the exporter’s work to sort nodes and this is not enforced by the framework: This means there is no sorting performed automatically for custom model creation. You should make sure nodes are inserted in the right order.
* *ModelHandle* is the refCounted version of *Model*.
  + Prefer using ModelHandle instead of plain C++ Model pointers.
* One mesh can be present in different nodes. A Model will provide you different method for individual mesh and meshNode e.g: getNumMeshes() vs getNumMeshNodes() OR getMesh() vs getMeshNode()
* To create a node on the model it is necessary to first allocate a node then a mesh
  + myModelHandle->allocNodes(1);
  + myModelHandle->allocMesh(1);
* The Model class contains nested classes: Node, Mesh, Light,Texture
* The Model class manages its internal data representation with a private member *InternalData m\_data*. You can request a reference to it with getInternalData(). The documentation explicitly hints “a handle with care” message for this method. By requesting the model’s internal data you do not benefit from all the help and safety checks the Model class provides you. It is safer to assume the latter knows best how to handle its internal data and instead use its interface to perform modifications.

**Behind the scenes of pvr::api::AssetStore::loadModel**

At the time of writing this method does not have any overload.

bool pvr::api::AssetStore::loadModel(const char \* filename, pvr::assets::ModelHandle, bool force);

This high level call uses utilies in PVRAsset, called PVRAsset::AssetReaders to load your model.

Here is how it happens in more details:

* PVRApi::AssetStore::loadModel()
  + PVRAssets::Model::createWithReader() (defined in Asset.h)
    - PVRAssets::AssetReader::readAsset()
      * PODReader::readNextAsset()

AssetReader::readNextAsset() is a pure virtual function implemented for each reader’s type. Let’s take the PODReader Implementation.

When scene tag is reached the model‘s cache is initialized and PODReader::readSceneBlock is called. The scene block is parsed: each time a new tag is encountered, the value is written to the model’s internal data.

From the pod reader implementation, I understand the pod file to be formatted as:

Version

Scene Data

Scene data defines separately the **number** of components in the scene (node, mesh, camera, light..), from the actual component’s description (A constituent block) i.e If there are 2 cameras there should be 2 camera blocks.

**Conclusion:** For simple geometries, the scene information can be loaded manually in VBOs and IBOs, and other buffers. There is no need to create a model. The model is only given as a commodity to load more complex scene data from a file or stream.

In brief you can think of it as:

File 🡪 AssetStore 🡪 Model 🡪 VBOs, IBOs

The following is pointless:

VBOs, IBOs 🡪 Model

If you have simple data not yet digested in a pvr::api object, you should skip the model creation and directly load VBOs and IBOs.

**How to work directly without Models**

1. Use pvr::api::Buffer
   1. Allocate them on the current context by calling getGraphicContext().createBuffer(*SizeInByte*, *pvr::types::BufferBindingUse*, *bool isMappable*)
   2. When configuring the commandBuffer
      1. Call pvr::api::pipelineCreation::VertexInputCreateParam::addVertexAttribute
         1. VertexAttributeLayout is defined in PVRCore. Its 3 members of interest are its dataType, its width (expressed in dataType unit), and its offset expressed in bytes.
      2. Call pvr::api::pipelineCreation::VertexInputCreateParam::setInputBinding to describe the buffer binding: its stride in byte, the step rate (can be per vertex, per instance, per draw)
   3. Before drawing the mesh bind your VertexBuffer, IndexBuffer

**Writing a simple Demo with the PowerVR framework**

1. **Minimal information you should provide the framework**
   1. The geometry and scene data:
      1. Whether they are loaded from a (POD) file, or filled directly within the application through VBOs, IBOs …
   2. A graphics pipeline:
      1. It needs to be configured! Most probably you’ll want to:
         1. Attach Shaders
         2. Tell the pipeline’s input assembler what sort of geometry to expect (is it a triangle list? a triangle fan, line list, point list, quads?)
         3. Set the face culling for the pipeline’s rasterizer
         4. Configure the depth and stencil operations
         5. Configure the blending state (enabled or disabled)
      2. You configure the graphics pipeline object through a pvr::api::GraphicsPipelineCreateParam
         1. It has all the xxxxxCreateParam methods found in pvr::api::pipelineCreation.

**Textures**

**What are texture views?**

Texture views are what you should be aiming for if you want to manipulate textures in the context of the framework. This is what ends in descriptor sets. They act at a higher level than pvr::assets::Texture, and allow different instance of the TextureView class to share the same image data.

The TextureView is a texture object, ready to consume by the framework. The underlying pvr::assets::Texture object has been uploaded to the API and its raw data / medata has been processed to fit the API requirements.

**Loading a texture from file**

pvr::api::AssetStore::loadTexture() will try to retrieve the texture from its cache

If this results in a cache miss, it gets a stream to the texture file and calls pvr::assets::textureLoad() (defined in Texture.cpp). If successful, pvr::utils::textureUpload() is called.

The cache is a map <StringHash, TextureData>

TextureData is a struct {pvr::asssets::TextureHeader, pvr::api::TextureView>}

pvr::assets::TextureHeader contains all the metadata of the texture file.

pvr::assets::Texture is a specialization of pvr::assets::TextureHeader

pvr::assets::textureLoad() is almost equivalent to opening the stream to the texture, creating a pvr::assetReaders of the the right type, calling readAsset on it, and closing the stream.

**Animations**

One animation is composed of multiple frames. Each frame has its own transformation matrix.

**Loading shaders**

Use a pvr::assets::ShaderFile

**Has the pipeline changed?**

Source: <http://gpuopen.com/vulkan-barriers-explained/>

All commands will run through these steps.

Answer: No: proof:

VK\_PIPELINE\_STAGE

* TOP\_OF\_PIPE\_BIT
* DRAW\_INDIRECT\_BIT
* VERTEX\_INPUT\_BIT
* VERTEX\_SHADER\_BIT
* TESSELLATION\_CONTROL\_SHADER\_BIT
* TESSELLATION\_EVALUATION\_SHADER\_BIT
* GEOMETRY\_SHADER\_BIT
* FRAGMENT\_SHADER\_BIT
* EARLY\_FRAGMENT\_TESTS\_BIT
* LATE\_FRAGMENT\_TESTS\_BIT
* COLOR\_ATTACHMENT\_OUTPUT\_BIT
* TRANSFER\_BIT
* COMPUTE\_SHADER\_BIT
* BOTTOM\_OF\_PIPE\_BIT

**What are render passes?**

They specify the framebuffer attachments.

They allow the driver to have all the state required to draw ahead of time and perform some optimisations

They can be composed of one or more subpasses. Each subpass can refer to a region of the framebuffer and specify what operation to perform with that framebuffer when the subpass begins or ends.

Subpasses can have interdependencies.

**What are descriptor sets and descriptor sets layout**