Learning Vulkan for Real Applications.

Author

Robert R Tipton

June 2020

Introduction

Vulkan is worth the effort of learning. It's speed, flexibility, error detection, device introspection and cross platform capabilities open the possibility of write once – use many development.

I found difficult to grasp for many weeks spread out over a year. I work in the CAD/CAE industry and was team lead on some large CAD/CAE/Design projects including writing some of the production graphics using OGL. So, I have a pretty good basis for comparison.

My colleagues have all been leery of tackling Vulkan when they saw how much code needed to be written just to get a triangle or cube on the screen. The good news is that once you pass that hurdle, development speeds up very fast.

I wrote Vulkan Quick Start as the foundation for other work I am doing at Dark Sky Innovative solutions. COTS tools had features I didn't need and were lacking features I did. I'm good at writing low level code, so for me it's faster to write something new than to spend hours downloading, wrestling with build systems on two or three different platforms in multiple languages etc.

As I am targeting a small customer base who places a very high priority on capabilities and is tolerant of the occasional crash – emphasis is on old style workflow testing. Unit testing is valuable but its cost is too high at this time.

So that's why it exists and the philosophy behind the way it was written.

General comments.

It's clear that Vulkan was written by some of the world's best pipeline and game programmers. When used as intended, it's blindingly fast, robust and helps you catch errors.

That also means that the vocabulary is a bit obscure to those outside that community. Like all such communities they have an abbreviated language which is unambiguous and concise to them; but a bit confusing to the rest of us.

There are also assumptions that all users will know where certain data structures should be put, how many and their life cycles.

Some of the best instructional texts are written by the ignorant as they learn something. That is what I am attempting to do here.

The key structures of Vulkan are pipelines and descriptors. There are hundreds of others, but these are the two you should understand best.

As a new comer, the word Pipeline describes the entire end to end cycle of drawing a frame. I assumed that every application would have a single pipeline. This was reinforced by the numerous examples of doing a single thing with the pipeline. How wrong I was.

Obviously, descriptors describe something(s). It was extremely difficult to figure out exactly what. I think of them now as Shader Binding Descriptors. They describe how the code on your host is bound to your shader code. And they do some very cool things.

Such as if you have an array of 8 samplers and only write data into 7 of them – the validation code warns you that you didn't initialize your 8th sampler. Very handy.

This is what led me to understand that if you write a generic shader that can handle 8 samplers and some of your objects only use 6 of them, that's a misuse of Vulkan. The Vulkan spec is enormous, in fact Vulkan is actually the spec. Individual companies write code that meets the spec. The Vulkan group itself doesn't write anything.

Based on that, I concluded that there should be a pipeline for each shader, vertex, texture permutation. That discovery alone broke a big piece of my learning block.

While I was learning, I ran across many comments online asking “I learned C++ quickly, why is Vulkan so much harder?” Many responses claimed that it wasn't.

IMO, the metaphor is wrong.

Vulkan is not a language. It's an API without a framework. When the first GUI API, Mac OS 1984, arrived it faced a similar problem. There were any number of ways you could apply the API to create an application, but you needed a map or example. Apple, thoughtfully provided entire function examples in Inside Mac (which was three large volumes,) but that still wasn't quite enough. Most of us still had to write about 1,500 lines of event loop code to get a demo running.

Learning the pieces was easy. Learning how to solve the puzzle was not.

VQS is *a framework* that can get you up and going with Vulkan. Including a primitive GUI layer so you don't need one from the host OS. For boutique applications like CAD/CAE/Design/Games etc. That's huge benefit.

I expect you to change, it rip it apart etc. to meet your needs.

Because of that, I haven't invested a lot of time on making it pretty or complete. I'm assuming that you know how to write shaders, entities etc. and that what I write won't match your needs. So I've tried not to bother as much as possible.

VQS currently keeps a list of shader sets. Vertex, fragment and other shaders that are intended to work together.

VQS pipelines pull these shader sets from the shader pool by id.

Each VQS pipleline is bound to a shader set id and the API uses that id to create and select shaders. It's fairly robust.

The VQS Pipeline class is a template that takes a UBO_TYPE and a VERT_TYPE. These are the structures that are used to form the descriptors.

Each Pipeline has a list of scene nodes, drawn in linear sequence, which are compatible with that Pipeline type.

```
template<class UBO_TYPE, class VERT_TYPE>
class Pipeline : public PipelineBase {
public:
	using VertexType = VERT_TYPE;
	using SceneNode = SceneNode<Pipeline>;
	using SceneNodePtr = std::shared_ptr<SceneNode>;
	using SceneNodeList = std::vector<SceneNodePtr>;

	void addSceneNode(const SceneNodePtr& node);
	…
```

In order to be able to add a scene node to a Pipeline, that class must be an instantiated template based on Pipeline::SceneNode.

```
class PipelineUi : public Pipeline<UniformBufferObjectUi, VertexUi> {
public:
	using UniformBufferObject = UniformBufferObjectUi;
	using PipelinePtr = std::shared_ptr<PipelineUi>;
	using VertexType = VertexUi;
	…

class SceneNodeUi : public PipelineUi::SceneNode {
public:
	SceneNodeUi();
	…
```

This makes it difficult to put a scene node in an incompatible pipeline.

The app draws all the pipelines in creation order.

Each pipeline sets the shaders and other pipeline data and draws it's scene nodes in order. This has the effect of unwinding multiple, diverse objects into continuous streams within the pipeline.

It also means that for a rich application with many shaders, vertex types, textures and permutations of them – there will be a lot of pipelines.

Adding a Pipeline to the app looks like:

```
_pipeline = _app->addPipeline(createPipeline<PipelineUi>(_app ));
```

Adding a scene node looks like:

```
_pipeline->addSceneNode(btnPtr);
```

This follows a matrix pattern taught to me by an architect a top game house. There is a 3D structure, a painter's algorithm ordered structure, a pipeline ordered structure etc. The API places each node, or a proxy, in the appropriate place in each structure.

Once that structure is in place, it's extremely easy to drop new items into it. Making changes to the structure once it's in use is challenging. I compare it to heart surgery, the patient has to keep breathing while you work.

Wrap up

I'm stopping at this point for now. VQS is a foundation for other work and that's what I'm staring work on. I'm available for consulting on a limited basis.

I've given you a map, go explore.