idc15

Imagination Developers Connection

PowerVR Framework

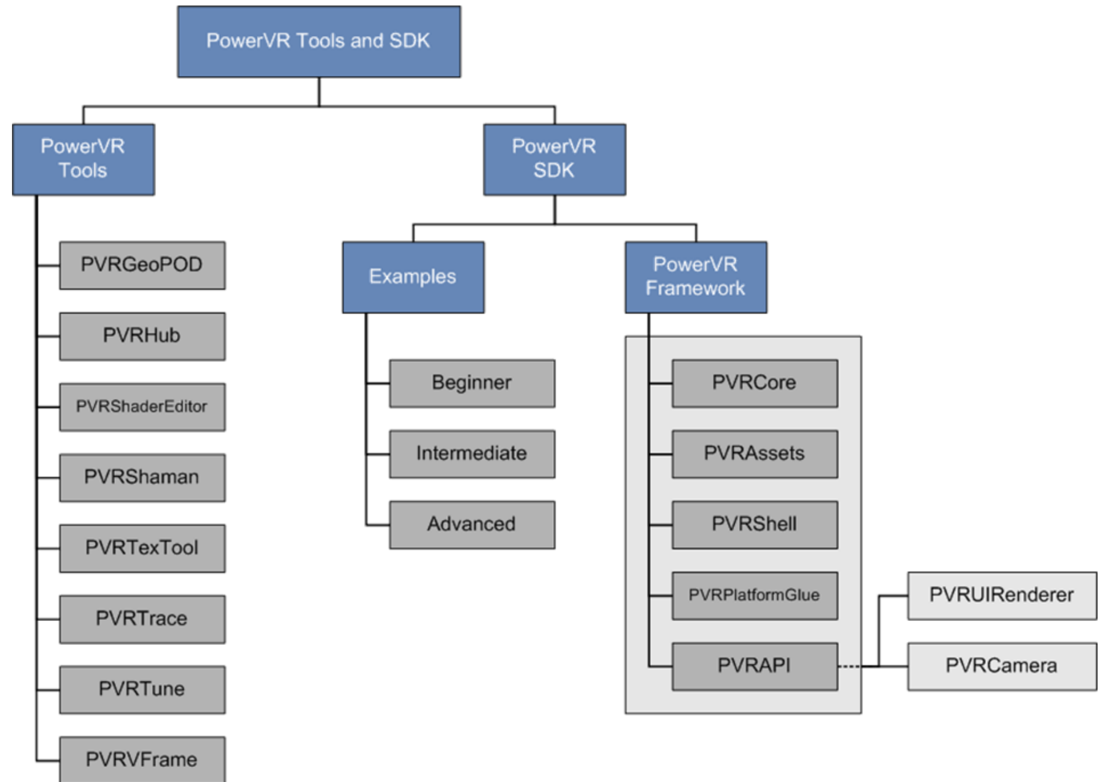
October 2015

Gerry Raptis
Leading Developer Technology Engineer, PowerVR Graphics

PowerVR Tools and SDK

Overview

- **Tools**
  - Development
  - Debugging
  - Optimisation
  - Authoring
- **SDK**
  - Development
  - Learning



Framework: What's changed?

Previously PVRShell and PVRTools

- **Structure**
  - Compilable libraries vs including cpp files in the project
  - Larger scope
  - A simple demo can be a single code file, along with shaders and resources
- **Style**
  - C++ *(namespacing, smart pointers, OOP)*
  - Uses the C++ standard library
  - Uses GLM for all maths
  - Some custom classes for cross-compiler support

The Benefits of the Framework

What's so great about it?

- Cross-platform, cross-API
- Highly permissive licence
- Provides API abstraction
- Facilitates modern graphics techniques
- Natively supports all PowerVR API extensions and PowerVR assets

PowerVR Framework

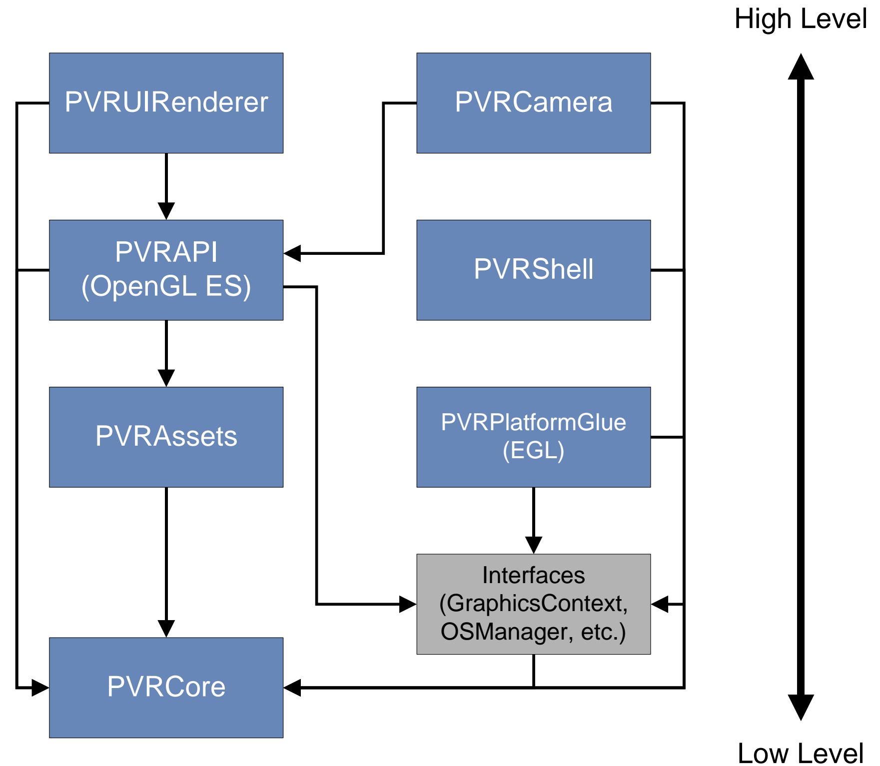
In a nutshell…

- **Premise**
  - Explicit APIs: really powerful, but also very verbose / difficult
  - Intermediate layers are important, as Explicit APIs hold Developer's hands less
- **Target**
  - Better support for explicit APIs by adding a "facilitation" layer
  - This facilitation layer then expanded to different API's
  - Enable cross-API development
  - Make the developer's life easier

PVR Framework Modules

Separate (compiled) libraries providing groups of functionality

- **Static Libraries**
  - Allows use of only parts of the framework
  - Everything depends on PVRCore
  - Possible to use only some modules and their dependencies



PVRCore

Scaffolding

- **Helpers, interfaces, and building blocks**
- **Example components**
  - File/Asset system abstraction
  - Data structures
  - Data types
  - Smart pointers/resources
  - Logging
  - Some geometry
  - Interfaces
- *Platform agnostic*
- *Completely API agnostic*

PVRCore

(Continued)

```
pvr::FileStream myTexture("BodyNormalMap.pvr");
pvr::WindowsResourceStream myTexture("BodyNormalMap.pvr");
pvr::AndroidAssetStream myTexture("BodyNormalMap.pvr");
pvr::FileStream myTexture("BodyNormalMap.pvr");
pvr::BufferStream myTexture(myTextureInMemory);
//Everything that deals with file/asset data uses streams

pvr::Logger myLog;
myLog.setMessenger(myCustomFileLoggingMessenger);
myLog(pvr::Logger::Information, "I am logging this in my custom logger");
pvr::Log(Log.Verbose, "Usually though I will just be using the global PowerVR Log object. Nobody likes globals, except for logging…");
```

PVRAssets

Asset abstractions and modelling

- **Contains code to deal with various actual asset/resource objects**
- **Natively supports all PowerVR formats**
  - POD (Models/scenes, exported with PVRGeoPOD)
  - PVR (Textures, exported with PVRTexTool)
  - PFX (Shaders/effects, created with PVRShaman, or text editor)
- **The classes `pvr::assets::Model, Texture, Effect,` work well together with POD, PVR, PFX, respectively**

PVRAssets

Some important concepts

- **Scene**
  - Abstraction: Model *(Mesh, Animation, Camera, Light, Material…)*
- **Textures**
  - Abstraction: Data, Format/Information, Metadata
- **Effects**
  - Shader sources, uniforms/attributes required to use the shader
- **Semantics**
  - Strings defining inputs or outputs of asset types

    (e.g. Mesh Vertex Attributes binding to effect VertexAttributes)

PVRAssets

From file to class

```
//"AssetReaders" abstract the loading code from representing classes
//The classes map excellently to PowerVR formats but are not limited to them

using pvr::assets;
assetReaders::PODReader podRd(myPODFileStream);
Model model; model.loadWithReader(podRd);

assetReaders::PfxReader pfxRd(myModel.getMaterial(0).getEffectFile());
Effect effect; effect.loadWithReader(pfxRd);

assetReaders::PvrReader pvrRd(myModel.getMaterial(0).getDiffuseTexture());
Texture diffuseTex; diffuseTex.loadWithReader(pvrRd); //Could check file extensions
```

PVRShell

Abstraction of the system, application main loop

- **Provides platform abstraction (file system, display, input), application lifecycle management, main loop**
  - Application inherits from the pvr::Shell class
    - Implements 5 mandatory callbacks and a few secondary *(initApplication, initView, renderFrame, releaseView, quitApplication)*
    - i.e. Defines application lifecycle
  - Transparent platform-specific asset retrieval *(filesystem, windows resources, android assets, etc.)*
  - Provides access to the main graphics context *(the one the window was created with…)*
  - Input handling

PVRShell

(Examples)

```
//The callbacks house the application
void MyApplication::initApplication { setApiTypeRequired(pvr::api::OpenGLES3);} //... etc.
pvr::Result::Enum MyApplication::renderFrame() { float dt =this->getFrameTime(); ...}


this->getAssetStream("Texture.pvr"); // Will look everywhere for assets: Files, then Windows
                                //Resources(.rc)/iOS bundled resources/Android assets


void MyApplication::eventMappedInput(SimplifiedInput::Actions evt){     //Abstracts/simplifies input
    switch (evt){case MappedInputEvent::Action1: pauseDemo();break;
           case MappedInputEvent::Left: showPreviousPage(); break;
           case MappedInputEvent::Quit: if (showExitDialogue()) exitShell(); break;
}}


void MyApplication::eventKeyUp(Keys::Enum key){...}        // Or, detailed keyboard/mouse/touch
input
```

PVRAPI

Abstraction of the graphics system

- **Abstracts the graphics API**
  - Wrappers over API objects (buffers, textures, samplers…)
  - Abstractions that go above and beyond, hammering different APIs together
  - Paradigm used is closer to explicit APIs such as Mantle or Vulkan
- **Provides scaffolding for higher-level libraries**
  - PVRUIRenderer
  - PVRCamera
  - *(Others in the future)*

PVRAPI

Features

- **Reference-counted objects, sensible defaults, helpers**
  - Regain some of the conciseness that explicitness will cost us
  - All commands/classes contain as many default values as is reasonable
  - Cleaner programming model *(pipelines/renderpasses: almost no sticky-state)*
  - Internal optimisation
  - Immutable
- **Pipeline objects** *(abstraction of the renderstate)*
  - Defaults mean you can create one with minimal input
- **Descriptor sets** *(abstractions of entire groups of API objects – textures, samplers, buffers...)*
  - Another strike to global state

PVRAPI

Features (continued)

- **RenderPass**
  - Same notion as the pipeline, but wraps Framebuffer and Read/Write ops configuration
- **CommandBuffer**
  - Submit commands creating a “package” out of them
  - Can simplify some types of multithreading

    (prepare commands in several threads, submit them in main rendering thread)

PVRAPI

Examples (initialization phase)

```
// Pipeline creation
GraphicsPipelineCreateParam pipeCreate;
pipeCreate.addDescriptorSetLayout(myDescriptorSet); //Pipeline needs to know what will be bound
pipeCreate.vertexShader = myCookedVertexShader; //Add shaders...
pipeCreate.vertexInput.addVertexAttribute(0, 0/*buffer index*/, posLayout, “inVertex”); //vertex
attributes...

// The rest though? Only whatever is not default…
pipeCreate.depthStencil.setDepthTest(true).setDepthWrite(false).setStencilTest(true).setStencilFunc
(…
// One-shot generation
GraphicsPipeline skinnedPipeline = getGraphicsContext().createGraphicsPipeline(pipeCreate);

// Buffer creation
Buffer vbo = getGraphicsContext().createBuffer(mesh.getDataSize(), BufferUsage::VertexBuffer);
buffer.update(myMesh.getDataPtr(0), 0, myMesh.getDataSize(0));
commandBuffer->bindVertexBuffer(vbo, 0/*offset*/, 0/*index, see above!*/);
```

PVRAPI

Example (Render phase)

```
CommandBuffer cb1 = getGraphicsContext().createCommandBuffer();
cb1.beginRecording();
cb1.beginRenderPass(onScreenFbo);
cb1.bindPipeline (skinnedPipeline);
cb1.bindDescriptorSet (chameleonManDescriptorSet);
cb1.bindVertexBuffer(vbo, 1 /* binding index*/);
cb1.bindIndexBuffer(ibo);
cb1.drawIndexed(0/*first index in the VBO*/, 42 /*indexes*/);
cb1.endRenderPass();
cb1.endRecording();

//This was all still Preparation phase. THIS is the actual render phase
void render(){
    cb1.submit();
}
```

Utilities of PVRAPI / PVRAssets

Cool boilerplate removers

- **Pipeline auto configure: Vertex attributes, primitive topology, input bindings**

```
utils::createInputAssemblyFromMeshAndEffect(pipelineDescriptor, Mesh)
utils::createInputAssemblyFromMesh(pipelineDescriptor, reflectiveVertexBindings)
utils::createInputAssemblyFromMesh(pipelineDescriptor, explicitVertexBindings)
```

- **Extract the VBO configuration from a Mesh. Many overloads for different uses.** *(Get model configuration, or match "Semantics" between Model and Effect)*

```
utils::createVBOsFromMesh(mesh, Vbo* etc.)
utils::createVBOsFromMeshAndEffect(mesh, effect, Vbo* etc.)
```

- **Load a Texture, upload it to the GPU, keep it and its name in a map so that you can reuse it**

```
api::Texture2D texture2D = assetManager.getTextureWithCaching("ChameleonBelt.pvr");
```

- **Get the on-screen display format, create a RenderPass matching it, create an FBO for the screen**

```
api::FBO fboOnScreen = this->createOnScreenFbo(/*Parameters like load/store ops can be set here*/);
```

PVRUIRenderer

Almost application level

- **A library built on top of PVRAPI, containing code to render text and images**
- **Text and images can be positioned in 2D or 3D**
  - Anchor to the screen in normalised coordinates
  - Anchor to their extents in normalised coordinates
  - Offset them in pixels
  - Scale
  - Rotations
  - Or, entire transformation matrix
- **Put items into MatrixGroups to transform them with a matrix**
- **Custom fonts exported easily from PVRTexTool**

PVRUIRenderer

Almost application level (continued)

```
using namespace pvr::ui;
UIRenderer spriteEngine; Text niceCustomText; Font myFont; Image myImage; Group myGroup;

spriteEngine->init(this->getContext()); //"this" is a pvr::Shell, a.k.a. our demo class
myText= spriteEngine->createText("Hello, world");
myText->setPositioning(Anchor::TopLeft, positionNdc, scale1to1, rotateRadians);

myGroup = spriteEngine->createGroup();
myGroup->setMatrix(glm::rotate(…)); myGroup->add(myText);

spriteEngine->beginRendering(commandBuffer);
myGroup->render();    //Everything is positioned relative to the screen
                      // and then transformed with the Group
myText->render(); //But we can still render it OUTSIDE of the group. It's still a sprite.
spriteEngine->endRendering();
```

PVRCamera

Helping out with accessing the camera in mobile platforms

- **Creates a `pvr::api::TextureView` out of the hardware camera system**
- **Supports iOS and Android** *(dummy class for Windows/Linux/OSX in order to assist development)*
- **Very helpful for mobile apps**

Conclusion

- **Works with you to facilitate using the newer APIs, helping with their verbosity without hiding their spirit**
- **Can be used on top of OpenGL ES**
  - Facilitates repetitive tasks
- **Features permissive licence, as well as great asset loading code**
- **Is a great starting point for your own engine**

Imagination

www.imgtec.com/idc