

FrontPanelDolphin Sample

*This sample is compatible with the Microsoft Game Development Kit (June 2020)*

# Description

FrontPanelDolphin demonstrates how to use the GPU to render to the Front Panel. We anticipate that most game developer will have plenty of code for rendering to the screen using the GPU and so the purpose of this sample is to make it easier to leverage your existing code to target the Front Panel display.

Here are a few use cases:

* You are running the game in the lab setting on a “headless” devkit and want to use the Front Panel to render what you would usually see on the screen. You can determine, at a glance, whether the game is healthy.
* Many game engines already have a “development HUD” that enables diagnostic features of the game that are not normally accessible in the retail version. For example, by performing a secret controller button combination, this would cause the HUD to appear. The HUD then provides additional options that are useful during testing and development, such as spawning monsters, jumping to a particular level, or making the character invincible. This HUD is normally rendered by the GPU, so if you have the ability to copy the results to the Front Panel, then you can reuse your existing HUD code and adapt it to the front panel. By putting it on the front panel, you don’t have to sacrifice screen real estate. Furthermore, you can use the DPAD and buttons on the Front Panel instead of using the Gamepad.

# Building the sample

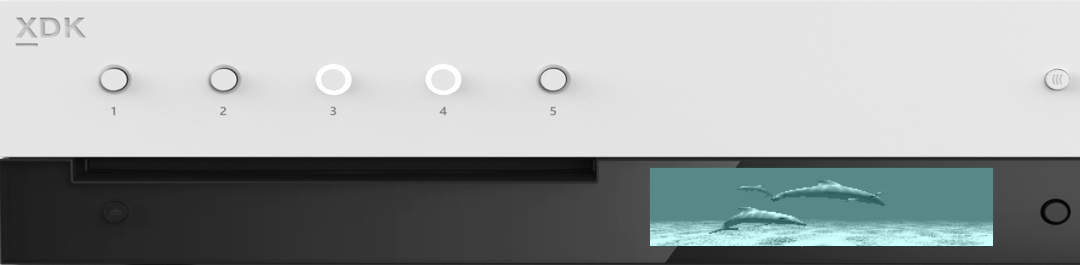
If using an Xbox One devkit, set the active solution platform to Gaming.Xbox.XboxOne.x64.

If using an Xbox Series X|S devkit, set the active solution platform to Gaming.Xbox.Scarlett.x64.

*For more information, see* Running samples*, in the GDK documentation.*

# Using the sample

The FrontPanelDolphin sample is intended for the Xbox One X Devkit and the Project Scarlett Devkit with the integrated front panel. When you launch the sample, it will render the dolphin scene on the main display and simultaneously on the front panel LCD display.



1 LED per Button

5x Programmable buttons

256 x 64 x 4bpp OLED display

DPAD + Select

## FrontPanelDolphin Main Display

## FrontPanelDolphin Front Panel Display



The sample doesn’t handle any input except for the DPAD Select button on the Front Panel. When you press the Select button it will capture the buffer from the front panel display and save the result to a .dds file located in the Title Scratch folder.

# Implementation notes

The sample uses a helper class called FrontPanelRenderTarget which, as the name suggests, is an off-screen render target that is suitable for the front panel display. The FrontPanelRenderTarget converts a provided render target resource to grayscale and then renders it to a quad. This is accomplished using a very simple vertex shader and a very simple pixel shader. The vertex shader produces a quad and the pixel shader samples the provided texture and converts each pixel to grayscale using a dot product. The render step is implemented in a method called GPUBlit():

// Render a grayscale image using passed-in renderTarget resource.

// Resource must be one of the render targets that what used to initialize this

// class.

void GPUBlit(

ID3D12GraphicsCommandList \*commandList,

ID3D12Resource \*renderTargetResource,

unsigned int renderTargetIndex);

When initialing the FrontPanelRenderTarget class, a double or triple-buffered render target list needs to be specified. GPUBlit() will render the current frame onto an intermediate double/triple-buffered resource.

After calling GPUBlit() you will then need to copy the results back to a buffer on the CPU and then present the buffer to the front panel display. The FrontPanelRenderTarget class provides two methods to make this easy: CopyToBuffer() and PresentToFrontPanel(). These two methods use the previous frame’s GPUBlit() render result to copy into a CPU buffer.

// Copy the render target from the previous frame to a staging texture and then

// copy it back to the CPU.

// Causes a GPU synchronization to ensure work from previous frame completes

// before reading on the CPU.

void CopyToBuffer(

ID3D12Device \*device,

ID3D12CommandQueue \*commandQueue,

unsigned int \*renderTargetIndex,

ATG::BufferDesc &desc);

// Copy the render target to a staging texture, copy the result back to the CPU

// and then present it to the font panel display

// Causes a GPU synchronization to ensure work from previous frame completes

// before reading on the CPU.

void PresentToFrontPanel(

ID3D12Device \*device,

ID3D12CommandQueue \*commandQueue,

unsigned int \*renderTargetIndex);

BufferDesc is a structure that keeps track of the width and height of a CPU buffer. FrontPanelRenderTarget::CopyToBuffer can copy to any address in memory, all it needs is a BufferDesc describing the dimensions of the buffer. The sample uses the FrontPanelDisplay class which manages a buffer for the Front Panel and uses FrontPanelDisplay::GetBufferDescriptor() to get the BufferDesc for the Front Panel. Then it calls FrontPanelRenderTarget::CopyToBuffer() copy the image from the FrontPanelRenderTarget. Finally, it must call FrontPanelDisplay::Present() to actually present the image to the Front Panel display.

Note that FrontPanelRenderTarget::PresentToFrontPanel() method takes care of both steps, copying to the CPU and presenting the buffer. This is convenient in case you are not already using the FrontPanelDisplay class.

Adapting the Dolphin sample to render to the Front Panel was actually quite straightforward using the FrontPanelRenderTarget. A minimal adaptation only needs to make a handful of changes:

**In Sample::Sample:**

// Construct the front panel render target

m\_frontPanelRenderTarget = std::make\_unique<FrontPanelRenderTarget>();

// Initialize the FrontPanelDisplay object

m\_frontPanelDisplay = std::make\_unique<FrontPanelDisplay>(m\_frontPanelControl.Get());

**In Sample::CreateDeviceDependentResources:**

// Create the front panel render target resources

m\_frontPanelRenderTarget->CreateDeviceDependentResources(frontPanelControl.Get(),

device);

**In Sample::CreateWindowSizeDependentResources:**

// Assuming max of 3 render targets

ID3D12Resource\* pRenderTargets[3] = {};

for(unsigned int rtIndex = 0; rtIndex < m\_deviceResources->GetBackBufferCount();

++rtIndex)

{

pRenderTargets[rtIndex] = m\_deviceResources->GetRenderTarget(rtIndex);

}

auto device = m\_deviceResources->GetD3DDevice();

m\_frontPanelRenderTarget->CreateWindowSizeDependentResources(  
 device,   
 m\_deviceResources->GetBackBufferCount(),   
 pRenderTargets);

**In Sample::Render:**

// Blit to the Front Panel render target and then present to the Front Panel

auto device = m\_deviceResources->GetD3DDevice();

unsigned int frameIndex = m\_deviceResources->GetCurrentFrameIndex();

m\_frontPanelRenderTarget->GPUBlit(commandList, renderTarget, frameIndex);

auto fpDesc = m\_frontPanelDisplay->GetBufferDescriptor();

m\_frontPanelRenderTarget->CopyToBuffer(device, commandQueue, frameIndex, fpDesc);  
m\_frontPanelDisplay->Present();

# Update history

April 2019, first release of the sample.

November 2019, support for the Project Scarlett Devkit.

# Privacy Statement

When compiling and running a sample, the file name of the sample executable will be sent to Microsoft to help track sample usage. To opt-out of this data collection, you can remove the block of code in Main.cpp labeled “Sample Usage Telemetry”.

For more information about Microsoft’s privacy policies in general, see the [Microsoft Privacy Statement](https://privacy.microsoft.com/en-us/privacystatement/).