Advanced Micro Devices Inc.

LiquidVR SDK SimpleAsyncCompute Sample

Sample Technical Guide

11-23-2015

Disclaimer

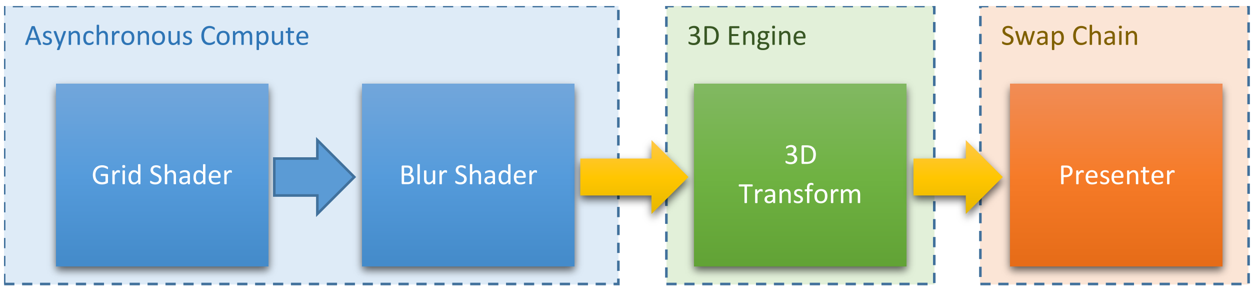
Copyright Notice

Overview

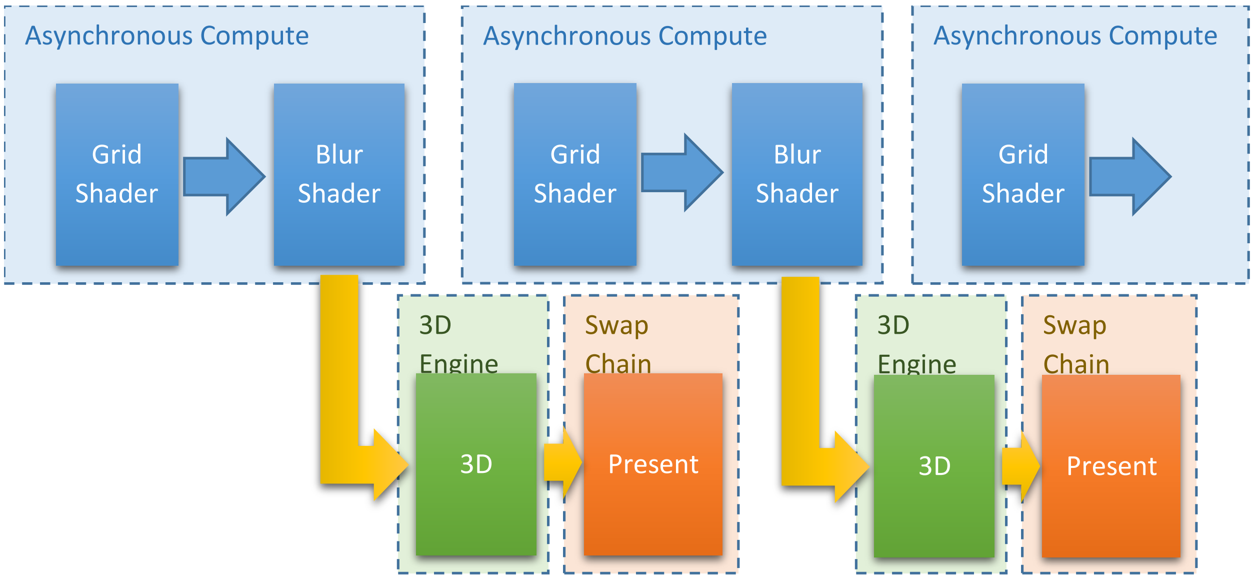
The SimpleAsyncCompute sample demonstrates the use of Asynchronous Compute in LiquidVR™. Asynchronous Compute is a performance-boosting feature unique to AMD hardware, which allows to execute the same DirectCompute™ shaders independently of 3D tasks. Since Microsoft DirectCompute API does not have the capability to submit asynchronous jobs, LiquidVR SDK provides a private API for Asynchronous Compute. This sample demonstrates how to use this API correctly.

The sample renders two rotating cubes with a blurred grid over them. It uses two compute shaders to draw the grid to imitate a rendering task and to apply Gaussian blur to it to imitate a post-processing task. After that a 3D transformation is applied to the texture containing the blurred grid to wrap it around a cube.

The following diagram illustrates the above:



Asynchronous Compute allows to execute the shaders in parallel with the 3D Transform, as shown below:



Pre-requisites

To run the sample, the following hardware and software requirements need to be fulfilled:

- A PC with an AMD GPU supported by LiquidVR™.
- Microsoft Windows 7, Windows 8.1 or Windows 10 with Radeon Crimson graphics drivers.

- To compile the sample, Microsoft Visual Studio 2013 is required.

Sample Code Overview

The sample's source code is arranged in three files:

1. SimpleAsyncCompute.cpp – contains the main() function and most of the sample's logic
2. D3DHelper.cpp – contains lower-level rendering code
3. LvrLogger.cpp – contains code to log messages to the terminal window and measure frame rates, as well as some general purpose helper functions

Initialization

All of the initialization, such as loading the LiquidVR DLL and creation of the required interfaces is performed in the Init() function located in the SimpleAsyncCompute.cpp file.

Load the LiquidVR DLL and obtain a pointer to the ALVRFactory interface used to obtain other LiquidVR interfaces:

```
ALVR_RESULT res = ALVR_OK;
g_hLiquidVRDLL = LoadLibraryW(ALVR_DLL_NAME);
CHECK_RETURN(g_hLiquidVRDLL != NULL, ALVR_FAIL, L"DLL " << ALVR_DLL_NAME << L" is not found");
ALVRInit_Fn pInit = (ALVRInit_Fn)GetProcAddress(g_hLiquidVRDLL, ALVR_INIT_FUNCTION_NAME);
res = pInit(ALVR_FULL_VERSION, (void**)&g_pFactory);
CHECK_ALVR_ERROR_RETURN(res, ALVR_INIT_FUNCTION_NAME << L"failed");
```

Create and enable GPU Affinity:

```
res = g_pFactory->CreateGpuAffinity(&m_pLvrAffinity);
CHECK_ALVR_ERROR_RETURN(res, L"CreateGpuAffinity() failed");
res = m_pLvrAffinity->EnableGpuAffinity(ALVR_GPU_AFFINITY_FLAGS_NONE);
CHECK_ALVR_ERROR_RETURN(res, L"EnableGpuAffinity() failed");
```

Create a D3D11 device (refer to the *D3DHelper* class for more details. Please note that the *D3DHelper* class is shared among several samples) and obtain a pointer to the corresponding *ALVRDeviceExD3D11* interface:

```
res = g_D3DHelper.CreateD3D11Device();
CHECK_ALVR_ERROR_RETURN(res, L"CreateD3D11Device() failed");
res = g_pFactory->CreateALVRDeviceExD3D11(g_D3DHelper.m_pd3dDevice, NULL, &g_pLvrDevice);
CHECK_ALVR_ERROR_RETURN(res, L"CreateALVRDeviceExD3D11() failed");
```

Create a swap chain using the *D3DHelper::CreateSwapChain* method:

```
res = g_D3DHelper.CreateSwapChain(g_hWindow, g_iBackbufferCount);
CHECK_ALVR_ERROR_RETURN(res, L"CreateSwapChain() failed");
```

Create a 3D scene using the *D3DHelper::Create3DScene* method:

```
res = g_D3DHelper.Create3DScene(width, height, true);
CHECK_ALVR_ERROR_RETURN(res, L"Create3DScene() failed");
```

Create an Asynchronous Compute context:

```
ALVRComputeContextDesc computeDesc = {};
computeDesc.flags = g_bHighPriorityQueue ? ALVR_COMPUTE_HIGH_PRIORITY : ALVR_COMPUTE_NONE;
res = g_pFactory->CreateComputeContext(g_pLvrDevice, 0, &computeDesc, &g_pComputeContext);
```

LiquidVR allows to submit compute jobs either to a regular priority, or a high priority queue. Using a high priority queue guarantees that compute jobs would be executed within a certain time period.

Create a shader for rendering a grid by compiling the shader and creating a Compute task for it:

```
hr = D3DCompile((LPCSTR)s_pGridShaderText, strlen(s_pGridShaderText), NULL, NULL, NULL, "main", "cs_5_0",
dwShaderFlags, 0, &pVSBlob, &pErrorBlob);
CHECK_HRESULT_ERROR_RETURN(hr, L"D3DCompile(CS, Grid) failed" <<
                           reinterpret_cast<char*>(pErrorBlob->GetBufferPointer()));
res = g_pComputeContext->CreateComputeTask(ALVR_SHADER_MODEL_D3D11, 0, pVSBlob->GetBufferPointer(),
                         pVSBlob->GetBufferSize(), &g_pGridTask);
CHECK_ALVR_ERROR_RETURN(res, L"CreateComputeTask(Grid) failed");
```

The shader that renders the grid receives its parameters in a constant buffer, which needs to be created and made available to the shader by being bound to it:

```
res = g_pComputeContext->CreateBuffer(&descBuff, &g_pGridConstantBuffer);
CHECK_ALVR_ERROR_RETURN(res, L"CreateBuffer(Grid) failed");
res = g_pGridTask->BindConstantBuffer(0, g_pGridConstantBuffer);
CHECK_ALVR_ERROR_RETURN(res, L"BindConstantBuffer(Grid) failed");
```

Since the buffer is normally located in the GPU memory, it needs to be mapped to system memory to get populated. Unmapping the buffer transfers the newly updated content from system memory to GPU memory:

```
res = g_pGridConstantBuffer->Map(&pBuffPtr);
CHECK_ALVR_ERROR_RETURN(res, L"Map(Grid) failed");
memcpy(pBuffPtr, &gridParameters, sizeof(gridParameters));
res = g_pGridConstantBuffer->Unmap();
CHECK_ALVR_ERROR_RETURN(res, L"Unmap(Grid) failed");
```

The grid shader places its output into a surface, which needs to be created and bound to the shader as output:

```
ALVRSurfaceDesc descSurf ={};
descSurf.type = ALVR_SURFACE_2D;
descSurf.surfaceFlags = ALVR_SURFACE_SHADER_OUTPUT | ALVR_SURFACE_SHADER_INPUT;
descSurf.apiSupport = ALVR_RESOURCE_API_ASYNC_COMPUTE;
descSurf.width = g_iTextureSize;
descSurf.height = g_iTextureSize;
descSurf.depth = 1;
descSurf.format = ALVR_FORMAT_R8G8B8A8_UNORM;

res = g_pComputeContext->CreateSurface(&descSurf, &g_pGridOutputSurface);
CHECK_ALVR_ERROR_RETURN(res, L"CreateSurface(Grid) failed");
res = g_pGridTask->BindOutput(0, g_pGridOutputSurface);
CHECK_ALVR_ERROR_RETURN(res, L"BindOutput(Grid) failed");
```

Create a shader for blurring the grid:

```
hr = D3DCompile((LPCSTR)s_pBlurShaderText, strlen(s_pBlurShaderText), NULL, NULL, NULL, "main", "cs_5_0",
dwShaderFlags, 0, &pVSBlob, &pErrorBlob);
CHECK_HRESULT_ERROR_RETURN(hr, L"D3DCompile(CS, Blur) failed" <<
                           reinterpret_cast<char*>(pErrorBlob->GetBufferPointer()));
res = g_pComputeContext->CreateComputeTask(ALVR_SHADER_MODEL_D3D11, 0, pVSBlob->GetBufferPointer(),
pVSBlob->GetBufferSize(), &g_pBlurTask);
CHECK_ALVR_ERROR_RETURN(res, L"CreateComputeTask(Blur) failed");
```

Similarly to the grid shader, the blur shader receives parameters in a constant buffer:

```
res = g_pComputeContext->CreateBuffer(&descBuff, &g_pBlurConstantBuffer);
CHECK_ALVR_ERROR_RETURN(res, L"CreateBuffer(Blur) failed");
res = g_pBlurTask->BindConstantBuffer(0, g_pBlurConstantBuffer);
CHECK_ALVR_ERROR_RETURN(res, L"BindConstantBuffer(Blur) failed");
res = g_pBlurConstantBuffer->Map(&pBuffPtr);
CHECK_ALVR_ERROR_RETURN(res, L"Map(Blur) failed");
memcpy(pBuffPtr, &blurParameters, sizeof(blurParameters));
res = g_pBlurConstantBuffer->Unmap();
CHECK_ALVR_ERROR_RETURN(res, L"Unmap(Blur) failed");
```

The output of the blur shader is placed into a surface:

```
descSurf.apiSupport = ALVR_RESOURCE_API_ASYNC_COMPUTE | ALVR_RESOURCE_API_D3D11;
res = g_pComputeContext->CreateSurface(&descSurf, &g_pBlurOutputSurface);
```

```
CHECK_ALVR_ERROR_RETURN(res, L"CreateSurface(Blur) failed");
res = g_pBlurTask->BindOutput(0, g_pBlurOutputSurface);
CHECK_ALVR_ERROR_RETURN(res, L"BindOutput(Blur) failed");
```

The blur shader uses the output of the grid shader as input, therefore the output surface of the grid shader is bound to the blur shader as input:

```
res = g_pBlurTask->BindInput(0, g_pGridOutputSurface);
CHECK_ALVR_ERROR_RETURN(res, L"BindInput(Grid) failed");
```

and a sampler is created and bound to the shader:

```
ALVRSamplerDesc samplerDesc;
samplerDesc.filterMode = ALVR_FILTER_POINT;
samplerDesc.addressU = ALVR_ADDRESS_CLAMP;
samplerDesc.addressV = ALVR_ADDRESS_CLAMP;
samplerDesc.addressW = ALVR_ADDRESS_CLAMP;
res = g_pComputeContext->CreateSampler(&samplerDesc, &g_pBlurSampler);
CHECK_ALVR_ERROR_RETURN(res, L"CreateSampler(Blur) failed");
res = g_pBlurTask->BindSampler(0, g_pBlurSampler);
CHECK_ALVR_ERROR_RETURN(res, L"BindSampler(Blur) failed");
```

The output surface of the blur shader needs to be passed to the 3D engine to be transformed into a cube using the standard D3D11 API. To achieve this, a LiquidVR surface needs to be converted to a D3D11 texture. The interop between LiquidVR and D3D11 is implemented as follows:

```
ALVRResourceD3D11 resource;
res = g_pBlurOutputSurface->GetApiResource(ALVR_RENDER_API_D3D11, (void**)&resource);
CHECK_ALVR_ERROR_RETURN(res, L"GetApiResource(BLur) failed");
D3D11_SHADER_RESOURCE_VIEW_DESC shaderResDesc ={};
shaderResDesc.Format = DXGI_FORMAT_R8G8B8A8_UNORM;
shaderResDesc.ViewDimension = D3D11_SRV_DIMENSION_TEXTURE2D;
shaderResDesc.Texture2D.MipLevels = 1;
shaderResDesc.Texture2D.MostDetailedMip = 0;
hr = g_D3DHelper.m_pd3dDevice->CreateShaderResourceView(resource.pResource, &shaderResDesc,
                                                        &g_BlurShaderResourceView);
resource.pResource->Release();
CHECK_HRESULT_ERROR_RETURN(hr, L"CreateShaderResourceView() failed");
g_D3DHelper.m_pd3dDeviceContext->PSSetShaderResources(0, 1, &g_BlurShaderResourceView.p);
```

The 3D tasks need to be synchronized with Compute tasks since the output of Compute shader serves as input for the 3D transformation, which should start only after the Blur shader has completed its work. This is achieved by using a GPU semaphore – a special synchronization object which allows to synchronize multiple GPUs and different queues of the same GPU.

For more information about GPU semaphores and fences please refer to the LiquidVR API documentation.

A GPU semaphore can be waited on until it is signalled. Commands for signalling and waiting for a semaphore are placed into GPU queues using the *ALVRDeviceExD3D11::QueueSemaphoreSignal()* and *ALVRDeviceExD3D11::QueueSemaphoreWait()* methods respectively. When the wait instruction is encountered in the queue, the queue would get blocked until the signal instruction is executed on the same semaphore in a different queue.

To work correctly, semaphores need to be signalled and waited on in an alternating order. You should avoid queueing a semaphore to be signalled again before the wait for the previous signal on the same semaphore has been executed. This scenario can be avoided by using several semaphores and alternating them sequentially, as shown in the sample. The number of semaphores in the set must be determined experimentally.

Create several pointers to the *ALVRGpuSemaphore* interface as an array:

```
ATL::CComPtr<ALVRGpuSemaphore>           g_pShaderComplete[2];
```

Initialize the array by calling *ALVRDeviceExD3D11::CreateGpuSemaphore* on each element of the array:

```
for(int i = 0; i < _countof(g_pShaderComplete); i++)
{
        res = g_pLvrDevice->CreateGpuSemaphore(&g_pShaderComplete[i]);
        CHECK_ALVR_ERROR_RETURN(res, L"CreateGpuSemaphore() failed");
}
```

Cycle through the array as needed (please refer to the **Rendering a 3D Scene** section for more details).

Another type of GPU synchronization objects in LiquidVR is a GPU fence. A GPU fence allows to block a thread running on the CPU until the fence is signaled by a GPU. The sample utilizes a GPU fence to imitate synchronous compute shader execution to demonstrate the performance gain that can be achieved by using Asynchronous Compute by preventing the CPU from submitting the next compute job until the previous frame's 3D job is complete, effectively serializing all GPU tasks.

A GPU fence is created using the *ALVRDeviceExD3D11::CreateFence* API, as shown below:

```
res = g_pLvrDevice->CreateFence(&g_pFenceRender);
CHECK_ALVR_ERROR_RETURN(res, L"CreateFence() failed");
```

Rendering a 3D Scene

Rendering of the scene is performed from the main loop. The general rendering sequence is as follows:

```
SetupRenderTarget();
ClearFrame();
RenderTexture();
RenderFrame(false);
RenderFrame(true);
res = g_D3DHelper.Present();
g_D3DHelper.Animate();
```

The *SetupRenderTarget()* and *ClearFrame()* functions contain only the standard D3D11 code and are self-explanatory.

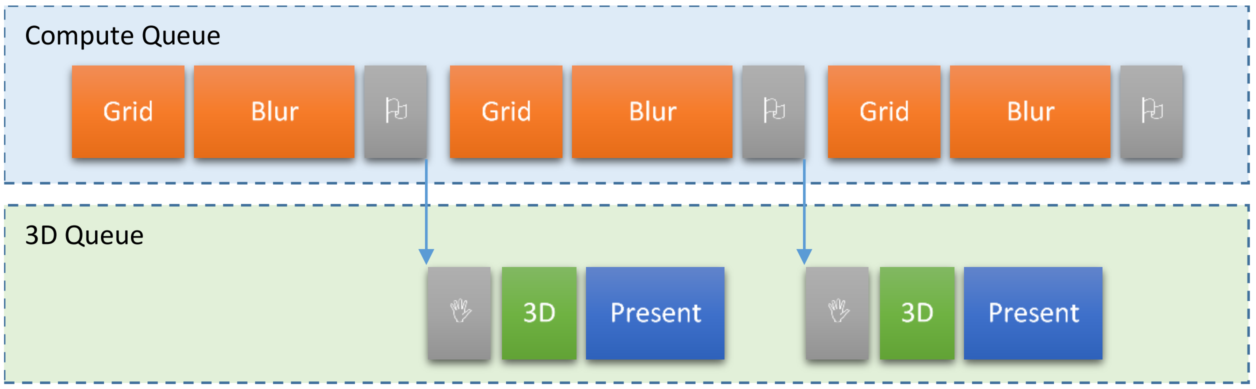
The *RenderTexture()* function submits two asynchronous compute jobs running the Grid and the Blur shaders discussed previously using the *ALVRComputeContext::QueueTask()* LiquidVR API:

```
static const int THREAD_GROUP_SIZE = 8;
ALVR_RESULT res = ALVR_OK;
ALVRPoint3D offset ={0, 0, 0};
ALVRSize3D size ={(g_iTextureSize + THREAD_GROUP_SIZE - 1) / THREAD_GROUP_SIZE, (g_iTextureSize +
                   THREAD_GROUP_SIZE - 1) / THREAD_GROUP_SIZE, 0};
res = g_pComputeContext->QueueTask(g_pGridTask, &offset, &size);
CHECK_ALVR_ERROR_RETURN(res, L"QueueTask(Grid) failed");
res = g_pComputeContext->QueueTask(g_pBlurTask, &offset, &size);
CHECK_ALVR_ERROR_RETURN(res, L"QueueTask(Blur) failed");
```

After submitting the compute tasks to render and blur the grid, the 3D engine needs to wait for their completion before wrapping the grid around the cube. This synchronization is achieved by means of GPU semaphores. As it was mentioned previously, to avoid re-queueing the same semaphore to be signaled before the wait operation on it has completed, multiple semaphores are being cycled through as shown below:

```
ATL::CComPtr<ALVRGpuSemaphore> pShaderComplete = g_pShaderComplete[g_iShaderCompleteFlip];
g_iShaderCompleteFlip++;
if(g_iShaderCompleteFlip == _countof(g_pShaderComplete))
{
   g_iShaderCompleteFlip = 0;
}
g_pComputeContext->Flush(NULL);
g_pComputeContext->QueueSemaphoreSignal(pShaderComplete);
g_pLvrDevice->QueueSemaphoreWait(ALVR_GPU_ENGINE_3D, 0, pShaderComplete);
```

An instruction to signal a GPU semaphore is added to the Compute queue after the Grid and the Blur tasks. At the same time an instruction to wait on the same GPU semaphore is added to the 3D queue, making the 3D engine wait for the completion of the Compute tasks. This sequence is repeated for every frame:



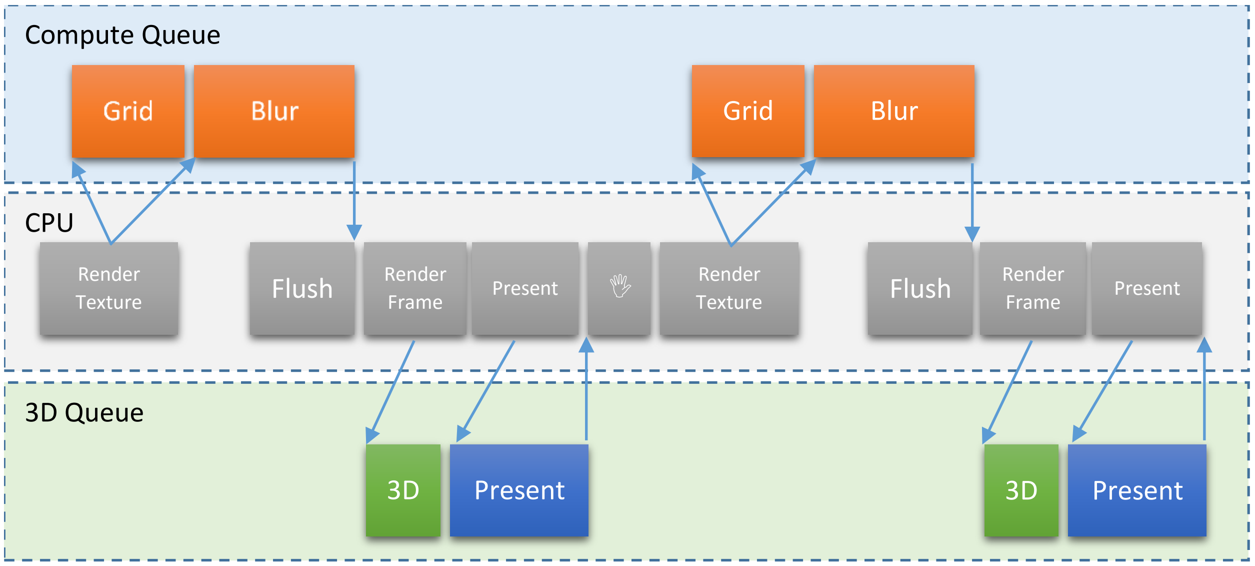
To simulate synchronous operation, the sample toggles the *g_bCpuSync* flag. When this flag is set, the sample is using a GPU fence to block further submissions until the fence is signaled:

```
ALVR_RESULT WaitForCompletion()
{
    ALVR_RESULT res = g_pLvrDevice->SubmitFence(0, g_pFenceRender);
    CHECK_ALVR_ERROR_RETURN(res, L"SubmitFence() failed");
    res = g_pFenceRender->Wait(2000);
    CHECK_ALVR_ERROR_RETURN(res, L"g_pFenceD3D11->Wait() failed");
    return ALVR_OK;
}
```

WaitForCompletion() is called at the beginning of *RenderTexture()* to prevent the compute jobs from being submitted before the 3D jobs are completed.

Also, instead of using GPU semaphores to synchronize the Compute and the 3D queues, the *ALVRComputeContext::Flush()* method is used to make the CPU wait until the queued compute jobs are finished.

When the *g_bCpuSync* flag is set, all GPU tasks are serialized, as shown below:



You can switch between the two modes by pressing the 'S' key while the sample is running.

Further Notes

Run the sample and try toggling the Asynchronous Compute mode on and off by pressing the 'S' key on the keyboard.

Try switching between the regular and the high priority queue by pressing the 'H' key on the keyboard.

Try changing the GPU load by changing the number of 3D operations using the UP and DOWN arrow keys.

Remember to always run the binaries built using the Release configuration when looking at performance numbers.