Advanced Micro Devices Inc.

LiquidVR SDK SimpleLateLatch Sample

Sample Technical Guide

11-23-2015

Disclaimer

The information contained herein is for informational purposes only, and is subject to change without notice. While every precaution has been taken in the preparation of this document, it may contain technical inaccuracies, omissions and typographical errors, and AMD is under no obligation to update or otherwise correct this information.

Advanced Micro Devices, Inc. makes no representations or warranties with respect to the accuracy or completeness of the contents of this document, and assumes no liability of any kind, including the implied warranties of noninfringement, merchantability or fitness for particular purposes, with respect to the operation or use of AMD hardware, software or other products described herein. No license, including implied or arising by estoppel, to any intellectual property rights is granted by this document. Terms and limitations applicable to the purchase or use of AMD's products are as set forth in a signed agreement between the parties or in AMD's Standard Terms and Conditions of Sale.

AMD, the AMD Arrow logo, ATI Radeon™, CrossFireX™, LiquidVR™ and combinations thereof are trademarks of Advanced Micro Devices, Inc. Other product names used in this publication are for identification purposes only and may be trademarks of their respective companies.

Windows™, Visual Studio and DirectX are trademark of Microsoft Corp.

Copyright Notice

Overview

This sample demonstrates the benefits and the use model of a special technique offered by LiquidVR called Late Latching. Late Latching improves an application's response time by allowing to update the rendering job's parameters after the job has been submitted to the GPU queue up to the moment when rendering actually starts. This allows to render moving objects closer to their intended position, thus reducing the lag in response to application controls.

The sample renders two color cubes that move to their new position following a trajectory controlled by mouse or by a simulator. You can toggle Late Latch for each cube individually and observe how the lag in their movement relative to each other is affected by Late Latch.

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™. Please note that Late Latching is currently not supported when CrossFire is enabled.
- 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. SimpleLateLatch.cpp – contains the main() function and most of the sample's logic
2. MouseInput.cpp – contains code for querying the mouse directly, bypassing Windows message queue, to improve mouse response time and eliminate the latency introduced by the Windows message queue itself.
3. D3DHelper.cpp – contains lower-level rendering code
4. 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 SimpleLateLatch.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 GPU fence:

```
res = g_pLvrDevice->CreateFence(&g_pFenceD3D11);
CHECK_ALVR_ERROR_RETURN(res, L"CreateFence() failed");
```

A GPU fence is a GPU synchronization object allowing to synchronize operations between a GPU and the CPU by letting the CPU wait on a fence that is being triggered by a GPU.

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

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");
```

Initialize the mouse monitor and switch it to the emulation (demo) mode:

```
g_MouseInput.Init((HINSTANCE)GetModuleHandle(NULL), g_hWindow, &g_D3DHelper, 200);
g_MouseInput.SetEmulation(true);
```

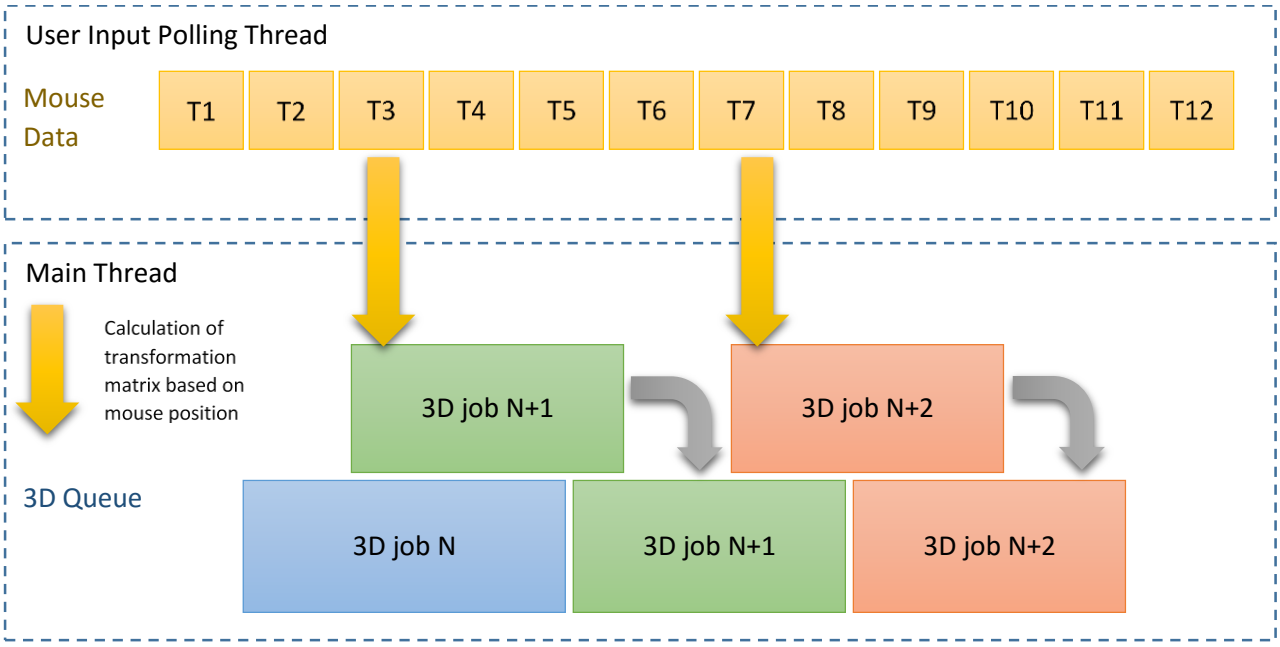
Rendering a 3D Scene

The sample is running two threads:

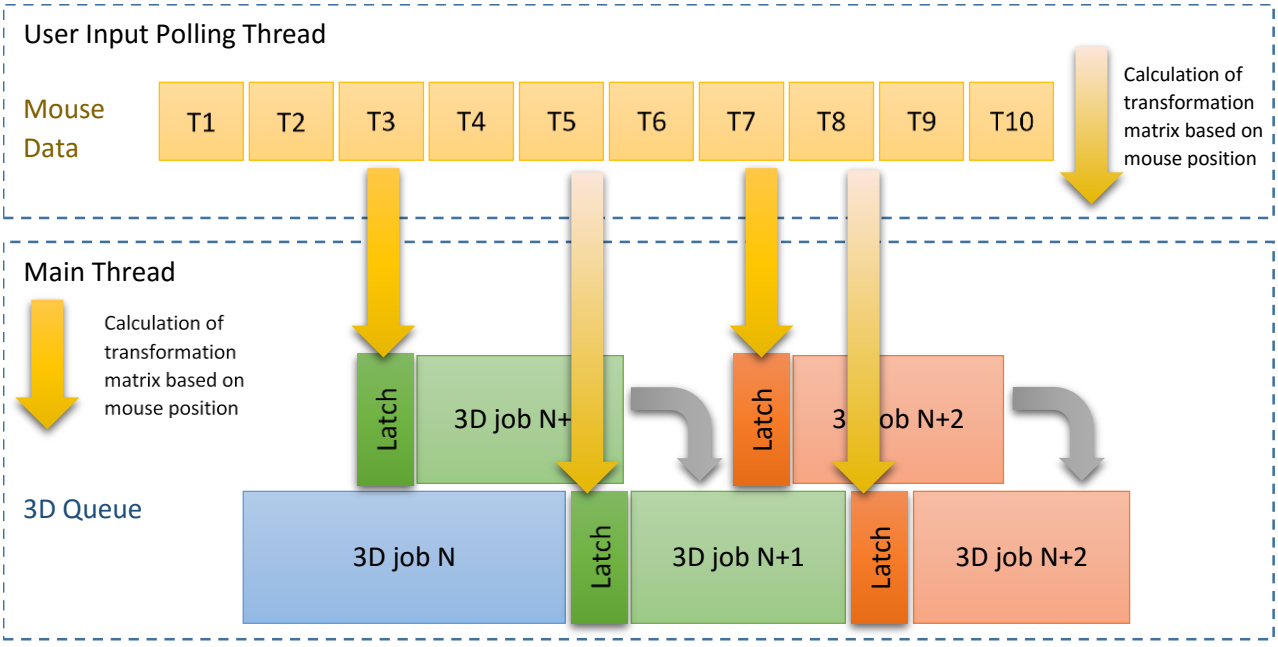
- The main thread, which performs all of the D3D and LiquidVR API calls.
- The user input polling thread, which continuously polls the mouse for its current position (or emulates it in the non-interactive demo mode). This thread is implemented by the *MouseInput::ThreadRun()* method.

Rendering of the cube is performed by the shader defined in *D3DHelperLateLatch::GetShaderText()*. The shader is using the transformation matrix based on the current mouse data to render the cube at the correct position. The transformation matrix is calculated and passed to the shader in a constant memory buffer by the *D3DHelperLateLatch::SetupMatrixForEye()* method. Calls to *D3DHelperLateLatch::SetupMatrixForEye()* are depicted with yellow arrows.

When Late Latch is not being used, the world view projection is calculated based on the sensor data current at the moment when a 3D operation is queued (T3 for job N+1 and T7 for job N+2 in the diagram below). However, by the time the job is executed, the sensor data changes to T5 and T8 respectively, which results in a lag:



Late Latch allows to update parameters passed to the shader up to the moment when the corresponding 3D job is actually executed, not when it is submitted, resulting in the object drawn by the shader having a more accurate position:



This is achieved by placing a Latch operation into the 3D queue, which will update the buffer containing shader parameters with the most current transformation matrix right before the rendering job is executed by the 3D engine. This is done with the help of the *g_D3DHelper.QueueLatch()* method, which, in turn calls the *ALVRLateLatchConstantBufferD3D11::QueueLatch()* LiquidVR API. The transformation matrix in the constant parameter buffer is updated by both threads – by the main thread at the time of a 3D job submission and by the user input polling thread from the *D3DHelperLateLatch::MouseEvent()* method:

From the main thread (in _tmain()):

```
// prepare left eye
g_D3DHelper.SetupViewportForEye(true);
g_D3DHelper.SetupMatrixForEye(true);
g_D3DHelper.QueueLatch(true);

// prepare right eye
g_D3DHelper.SetupMatrixForEye(false);
g_D3DHelper.QueueLatch(false);

// submit left eye
g_D3DHelper.SetupViewportForEye(true);
g_D3DHelper.BindBuffer(true);
g_D3DHelper.RenderFrame(g_iDrawRepeat);

// submit right eye
g_D3DHelper.SetupViewportForEye(false);
g_D3DHelper.BindBuffer(false);
g_D3DHelper.RenderFrame(g_iDrawRepeat);

// present a full frame for both eyes
res = g_D3DHelper.Present();
```

From the User Input Polling thread (in *D3DHelperLateLatch::MouseEvent()*):

```
g_D3DHelper.SetupMatrixForEye(false);
```

For the sake of simplicity, the non-Late Latch scenario is emulated by not updating the transformation matrix from the user input polling thread when Late Latch is turned off (in the *D3DHelperLateLatch::MouseEvent()* method):

```
if (g_bLateLatchRight)
{
   g_D3DHelper.SetupMatrixForEye(false);
}
```

When Late Latch is off, the transformation matrix is updated only by the main thread, which results in only submission-time values for the mouse position to be used. When Late Latch is on, the user input polling thread constantly updates the transformation matrix with the most up-to-date values.

Managing the Shader Parameter Buffer

The buffer containing parameters (transformation matrix) passed to the shader that renders the cube is allocated in the *D3DHelperLateLatch::CreateConstantBuffer()* method using the *ALVRDeviceExD3D11::CreateLateLatchConstantBufferD3D11()* LiquidVR API. A separate buffer of 1024 elements is allocated for the left and the right view:

```
m_pLateLatchBufferLeft.Release();
m_pLateLatchBufferRight.Release();

res = g_pLvrDevice->CreateLateLatchConstantBufferD3D11(sizeof(ConstantBuffer), 1024, 0,
&m_pLateLatchBufferLeft);
CHECK_ALVR_ERROR_RETURN(res, L"CreateLateLatchConstantBufferD3D11() failed");

res = g_pLvrDevice->CreateLateLatchConstantBufferD3D11(sizeof(ConstantBuffer), 1024, 0,
&m_pLateLatchBufferRight);
CHECK_ALVR_ERROR_RETURN(res, L"CreateLateLatchConstantBufferD3D11() failed");
```

The *ALVRLateLatchConstantBufferD3D11* buffer is a circular buffer consisting of up to 1024 slots, each containing an instance of the transformation matrix and an index portion, which contains the index of the

most recently updated slot. Every call to *ALVRLateLatchConstantBufferD3D11::Update()* places the most up-to-date transformation matrix into the next free slot and increments the index.

A call to the *ALVRLateLatchConstantBufferD3D11::QueueLatch()* LiquidVR API places a GPU job to retrieve the transformation matrix from the current slot into the 3D engine queue. When this job is executed by the GPU, the transformation matrix is retrieved by the GPU from the slot pointed to by the index portion of the *ALVRLateLatchConstantBufferD3D11* buffer, resulting in the most recent version of the transformation matrix being used by the shader (in the *D3DHelperLateLatch::BindBuffer()* method):

```
CComPtr<ALVRLateLatchConstantBufferD3D11>   pLateLatchBuffer =
        bLeft ? m_pLateLatchBufferLeft : m_pLateLatchBufferRight;
ID3D11Buffer *pData[2];
pData[0] = pLateLatchBuffer->GetDataD3D11();
pData[1] = pLateLatchBuffer->GetIndexD3D11();
m_pd3dDeviceContext->VSSetConstantBuffers(0, 2, pData);
```

Choosing the Late Latch Constant Buffer Size

The Late Latch Constant buffer is organized as a circular buffer consisting of the number of slots, each holding a copy of the transformation matrix. The size of each slot is determined by the size of parameters that need to be passed to the shader that renders the scene. The user interface thread continuously updates the buffer by adding another set of parameters at the tail of the buffer. Since the buffer is circular, once the end of the buffer is reached, the tail pointer will wrap around to the beginning of the buffer, potentially causing buffer overruns. To avoid buffer overruns, the buffer must contain enough slots to accommodate user input data from the moment the late latch is queued to the moment data latching actually occurs.

There is no set formula to calculate the minimum required number of slots. Generally it depends on the rate with which the user input thread is polling for user input relative to the rendering speed – the slower the rendering, the larger the buffer needs to be. The number of slots in the buffer can be determined empirically using moving objects' positioning as a criterion – movement to an incorrect position occasionally would indicate buffer overruns.

Further Suggestions

Run the sample and try toggling Late Latch by pressing the 'L' key on the keyboard. When the sample starts, Late Latch is turned off by default and both color cubes would be falling down at the same rate. Enabling Late Latch would make the right cube move slightly ahead of the left cube, for which Late Latching is not being used.

With Late Latch ON, try changing the number of times each cube gets redrawn, which equates to changing the GPU load and hence the frame rate, using the UP and DOWN arrow keys on the keyboard (the UP key increases the GPU load, while the DOWN key decreases it). Observe the difference in positioning of the cubes – the difference should increase as the GPU load increases.

Turn of the emulation mode by pressing the 'E' key on the keyboard. The cubes will stop falling down on their own. Try dragging the mouse with the left key pressed up and down and observe both cubes following the mouse. Toggle the Late Latch mode with the 'L' key while dragging the cubes and observe the difference in their movements.