HDRToneMapping11

<https://github.com/walbourn/directx-sdk-samples>

This is the DirectX SDK's Direct3D 11 sample updated to use the Windows 10 SDK without any dependencies on legacy DirectX SDK content. This sample is a Win32 desktop DirectX 11.0 application for Windows 10, Windows 8.1, Windows 8, and Windows 7.

**This is based on the legacy DirectX SDK (June 2010) Win32 desktop sample. This is not intended for use with Windows Store apps, Windows RT, or universal Windows apps.**

# Description



HDRToneMappingCS11 shows how to setup and run the compute shader(CS), which is one of the most exciting new features of Direct3D 11. Although the sample only utilizes the CS to implement HDR tone-mapping, the concept should extend easily to other post-processing algorithms, as well as to more general calculations.

## Overview

HDRToneMappingCS11 shows the use of the CS to do fast parallel reduction on the image to calculate average illuminance, and then use the calculated illuminance to do tone mapping. Until the real Direct3D 11 hardware comes along, the sample only runs on the software reference device. However, we encourage developers to start experimenting with the CS right now. Doing post-processing, as well as more general calculations using the CS, has the following advantages (at least) over the traditional of rendering a screen quad and then calculating by using the PS:

* The CS is capable of writing to any position of the output resource (also known as the ability to scatter). This allows more general and complex algorithms to be implemented on the graphics hardware.
* The CS provides mechanisms to share data, as well as to synchronize execution between threads. These abilities significantly reduce the number of redundant fetch operations, buffer or texture, especially for algorithms that involve the kernel or reduction calculations.
* Direct3D 11 has a dedicated API to explicitly launch a known number of CS threads on the GPU. This enables the algorithm to execute on an optimal number of threads, and it also provides predictable memory access and register usage. These lead to better performance and create more possibilities for optimization.
* The CS is not attached specifically to any stage of the graphics pipeline, and it has its own set of states. This means that your rendering pipeline and post-processing unit are decoupled from each other, which allows creation of code that is easier to maintain.

## CS 4.x and Post-Processing

Since CS 4.x lacks the ability of writing to textures directly, we have to:

* Use a structured buffer instead of texture 1D as the intermediate result in the reduction operation.
* Tile the image that is output from the two passes of the separable convolution (used in blur and bloom) to a structured buffer instead of outputting to texture 2D directly.
* Add an extra rendering pass to convert the image data that is stored in a structured buffer back to texture 2D before it can be sampled in the final composition pass.
* Use pixel shader to update the back buffer in the final composition pass.

Nonetheless, CS 4.x is good for the post-processing world because:

* As long as your post-processing algorithm doesn't require a texture resource as an intermediate result between passes, such as HDR tone mapping and histogram, CS 4.x could potentially provide better performance.
* As for other cases where multiple passes and intermediate texture data are necessary, although the implementation of these algorithms for CS 4.x may be slower (due to the redundant data conversion), it is still a great platform for prototyping and testing future CS 5.0 algorithms. CS 4.x is a core subset of CS 5.0, so it can help you understand most of the key benefits provided by the compute shader architecture—and you can play with these features on hardware today! Image convolution is an example.

### Harness the Power of Shared Memory in Convolutions

When doing convolution in pixel shader, for each output pixel, adjacent pixels in the input texture are always read in, multiplied by weights, and then added together to get the value for the result. There will be many redundant reads in this operation, especially when the convolution kernel gets larger.

However, if the convolution is implemented in compute shader, we could use the shared memory to load as many pixels as the number of threads in the thread group—and of course, multiple thread groups work concurrently to process multiple chunks of the input image—and then carry out the kernel convolution operation on the shared memory efficiently. Shared memory is usually implemented by hardware vendors as on-chip registers or a part of its on-chip cache, and thus, it is very fast to access. You could think of this benefit as the shared memory architecture brings the ability to explicitly program the behavior of a large cache, so that the hit ratio can be very ideal, because you know the explicit access pattern of your algorithm.

In this scheme, each pixel in shared memory is loaded only once from input image using texture read, and adjacent input pixels are always found in the shared memory when calculating result pixels. Please refer to FilterCS.hlsl for details on how this is done.

# Dependencies

DXUT-based samples typically make use of runtime HLSL compilation. Build-time compilation is recommended for all production Direct3D applications, but for experimentation and samples development runtime HLSL compilation is preferred. Therefore, the D3DCompile\*.DLL must be available in the search path when these programs are executed.

* When using the Windows 10 SDK and targeting Windows 7 or later, you can include the D3DCompile\_47 DLL side-by-side with your application copying the file from the REDIST folder.

%ProgramFiles(x86)%\Windows kits\10\Redist\D3D\ x86 or x64

# More Information

[Where is the DirectX SDK (2021 Edition)?](https://aka.ms/dxsdk)

[DXUT for Win32 Desktop Update](https://walbourn.github.io/dxut-for-win32-desktop-update/)

[Games for Windows and DirectX SDK blog](https://walbourn.github.io/)