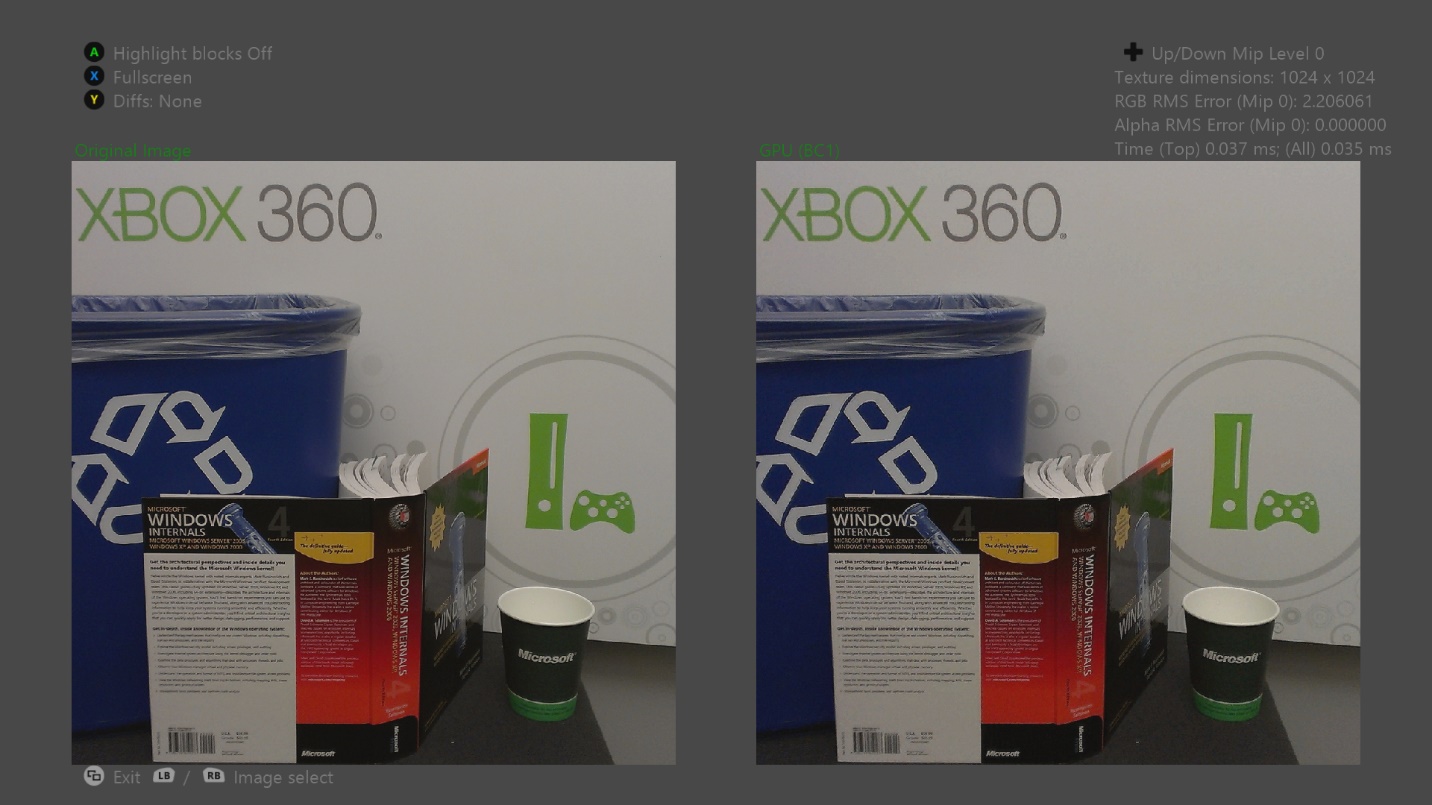


FastBlockCompress Sample

*This sample is compatible with the March 2017 XDK.*

# Description

# This sample demonstrates how you can use DirectCompute to perform fast texture compression at run time to the BC1, BC3, and BC5 formats based on the classic *Fast Block Compression* algorithm. The sample also allows you to toggle between run-time and offline compression modes to compare visual quality.



# Using the sample

|  |  |
| --- | --- |
| Action | Gamepad |
| Previous or next image | Left or right bumper |
| Previous or next compression method | D-pad left or right |
| Previous or next mip level | D-pad down or up |
| Move the camera | Right stick |
| Zoom in or out | Left or right trigger |
| Full screen vs. side-by-side | X |
| Highlight blocks | A |
| Cycle diffs mode | Y |
| Exit | View Button |

# Background

Xbox One has 5 GB of unified memory available for exclusive apps, which is a dramatic increase of 10 times the 512 MB available on the Xbox 360. Unfortunately, IO bandwidth and storage media capacity have not quite kept pace. The Blu-ray media will hold 49 GB, which is an increase of only 6.3 times the 7.8 GB available with the Xbox 360 game disc version 3.

This fact, combined with the introduction of streaming install, means that efficient compression methods are still important for minimizing load times and for packing your game assets into the available storage space.

Titles can often save a large amount of storage space by encoding game textures using an offline image-compression format. Xbox One has a hardware JPEG decoder built-in, which makes JPEG an attractive choice. However, the JPEG hardware decodes textures to an uncompressed YUV format in memory, which is not optimal for rendering. If you use this method, your title will need to recompress textures at run time to one of the block-compression formats that the GPU supports.

This sample uses the GPU to efficiently compress textures to the BC1, BC3, and BC5 formats. The standard algorithms used for offline block compression have historically been too slow to run in real-time, and the algorithm that the sample uses makes significant quality compromises in the interest of speed.

Because memory bandwidth bottlenecks the current algorithm, using other techniques, you may be able to achieve noticeable quality improvements with only a small performance penalty.

# Implementation notes

# Each DirectCompute compression shader in the sample has three variations: a one mip version, a two mips version, and a tail mips version:

# The one mip shader compresses a single mip of the source texture.

# The two mips shader reads a single mip of the source texture and down samples the mip to local data store (LDS) memory. Next, the shader compresses both the original and down sampled versions, and then the shader writes the corresponding mip levels in the output texture.

# This process saves memory bandwidth by avoiding reads from the source texture for the second mip level. But in practice, the performance gains are largely offset by the added shader complexity and the decreased occupancy due to higher GPR and LDS usage.

# The tail mips shader compresses the 16×16 through 1×1 mip levels of the source texture in a single dispatch call by selecting different threads to work on different mip levels.

# Because the minimum wavefront size is 64 threads, a technique that compressed each tail mip in a separate dispatch call would waste most of the available threads. By using only one wavefront and dispatch call, the tail mips shader avoids the bulk of this wasted work.

# Direct3D does not permit binding BC-format textures as UAVs, so you can’t write directly to a block-compressed texture from a compute shader. The sample works around this limitation by aliasing an intermediate texture in a writable format to the same memory location as the block-compressed texture. The intermediate texture is a quarter of the size, and each texel corresponds to a block in the compressed texture.

# Aliasing texture memory in this fashion requires an exact match between the two textures’ tiling modes and memory layouts. Additionally, Direct3D is oblivious to memory aliasing, so the GPU may simultaneously schedule multiple draw or dispatch calls that operate on different resources aliased to the same memory location.

# In other words, a shader that writes to the intermediate texture may be scheduled at the same time as a draw call that reads from the aliased block-compressed texture. To prevent these hazards, you should manually insert appropriate fences.

Offline line compression algorithms are implemented in [DirectXTex](https://github.com/Microsoft/DirectXTex/).

# References

Microsoft Advanced Technology Group. Fast Block Compress sample. Xbox 360 SDK. February 2010.

Tranchida, Jason. [Texture Compression in Real-Time Using the GPU](http://www.gdcvault.com/play/1012554/Texture-compression-in-real-time). GDC 2010. March 2010.

van Waveren, J.M.P[. Real-Time DXT Compression](https://software.intel.com/sites/default/files/23/1d/324337_324337.pdf). Intel Software Network. May 2006.

van Waveren, J.M.P., and Castaño, Ignacio. [Real-Time YCoCg-DXT Compression](https://www.nvidia.com/object/real-time-ycocg-dxt-compression.html). NVIDIA developer site. September 2007.

van Waveren, J.M.P., and Castaño, Ignacio. [Real-Time Normal Map DXT Compression](http://developer.download.nvidia.com/whitepapers/2008/real-time-normal-map-dxt-compression.pdf). NVIDIA developer site. February 2008.

# Update history

Originally released in 2015. This was rewritten using DirectX 12 in December 2018.

# 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/).