GPU Video Effects

This **gpu_videoeffects** demo serves two purposes:

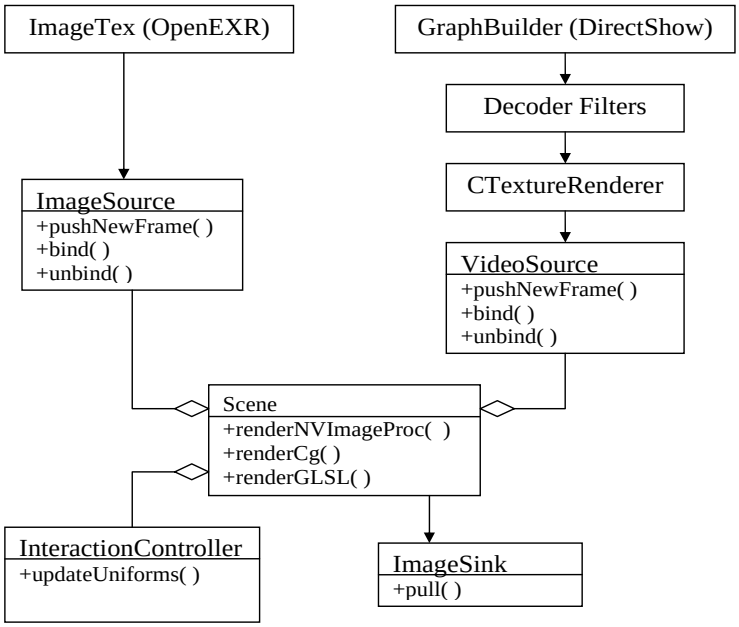
1. It provides a framework for developers to experiment with creating new GPU accelerated effects for use with Video and Images. This demo uses the nv_image_processing framework, Cg, and GLSL.
2. This demonstration code can be used as a sample implementation on how to use Microsoft DirectShow with the OpenGL EXT_pixel_buffer_object extension to quickly upload video to the GPU and read it back quickly.

NVIDIA Corporation
2701 San Tomas Expressway
Santa Clara, CA 95050
www.nvidia.com

Class Design

Core Functionality

The **gpu_videoeffects** multimedia demo uses a set of classes and libraries in to upload and render images/video on the GPU. For video data, classes GraphBuilder, CTextureRenderer, and VideoSource are used to upload video to the GPU. For static images, classes ImageTex and ImageSource upload static images to the GPU. After the video or image is uploaded, the Scene class applies GPU accelerated effects using ImageFilter (nv_image_processing), ProgramCg, and ProgramGLSL. ImageSink reads the final rendered result from GPU back to system memory:



GraphBuilder connects all the proper DirectShow filters necessary to decode and render a video stream. An OpenGL renderer (CTextureRenderer) supports the DirectShow interface and connects through a FilterGraph. This renderer connects to the Scene framework, so the GPU can render (VideoSource or ImageSource) images with hardware accelerated effects.

Classes ImageSource and VideoSource update the textures to be rendered. Each of these classes have a bind( ) function to binds the input image or video to the current OpenGL texture unit. The pushNewFrame( ) method should be called before the bind( ) to ensure the texture is uploaded to the GPU. pushNewFrame( ) also returns the number of bytes that were transferred from system up to the GPU.

The Scene class renders a screen-aligned quad to the frame buffer. The render( ) method calls pushNewFrame( ) from the ImageSource or VideoSource and binds a new texture. Scene uses any one of three methods depending on what the application sets: renderCg to render a Cg Program effect, renderGLSL to render a GLSL Program effect, and renderNVimageProc for a nv_image_processing effect.

InteractionController takes parameters from the GUI's slider controls, and passes the uniform values to the vertex and pixel shader program. Using these shader programs can be used to perform filtering effects such as: gamma correction, exposure correction, Gaussian blur, night vision filter, radial blur, and edge detection.

ImageSink reads data from the frame buffer back to system memory. The pull( ) method returns the number of bytes transferred from the GPU to system memory.

ImageSource and ImageSink are all abstract base classes. The various classes derived from these base abstract classes implement different transfer mechanisms.

ImageSource has four concrete implementations:

1. ImagePusher which uses normal glTextureSubImage( ) to upload textures to the GPU.
2. StaticImage which creates a texture and uploads it only once. Subsequent calls to pushNewFrame() will not cause any new data to be transferred to the GPU.
3. ImagePusherPBO uses the EXT_pixel_buffer_object to repeatedly transfer new texture content to the GPU (dynamic texture usage and write only modification).
4. VideoPusherPBO is similar to ImagePusherPBO, with the addition that it handles video images.

ImageSink has three concrete implementations:

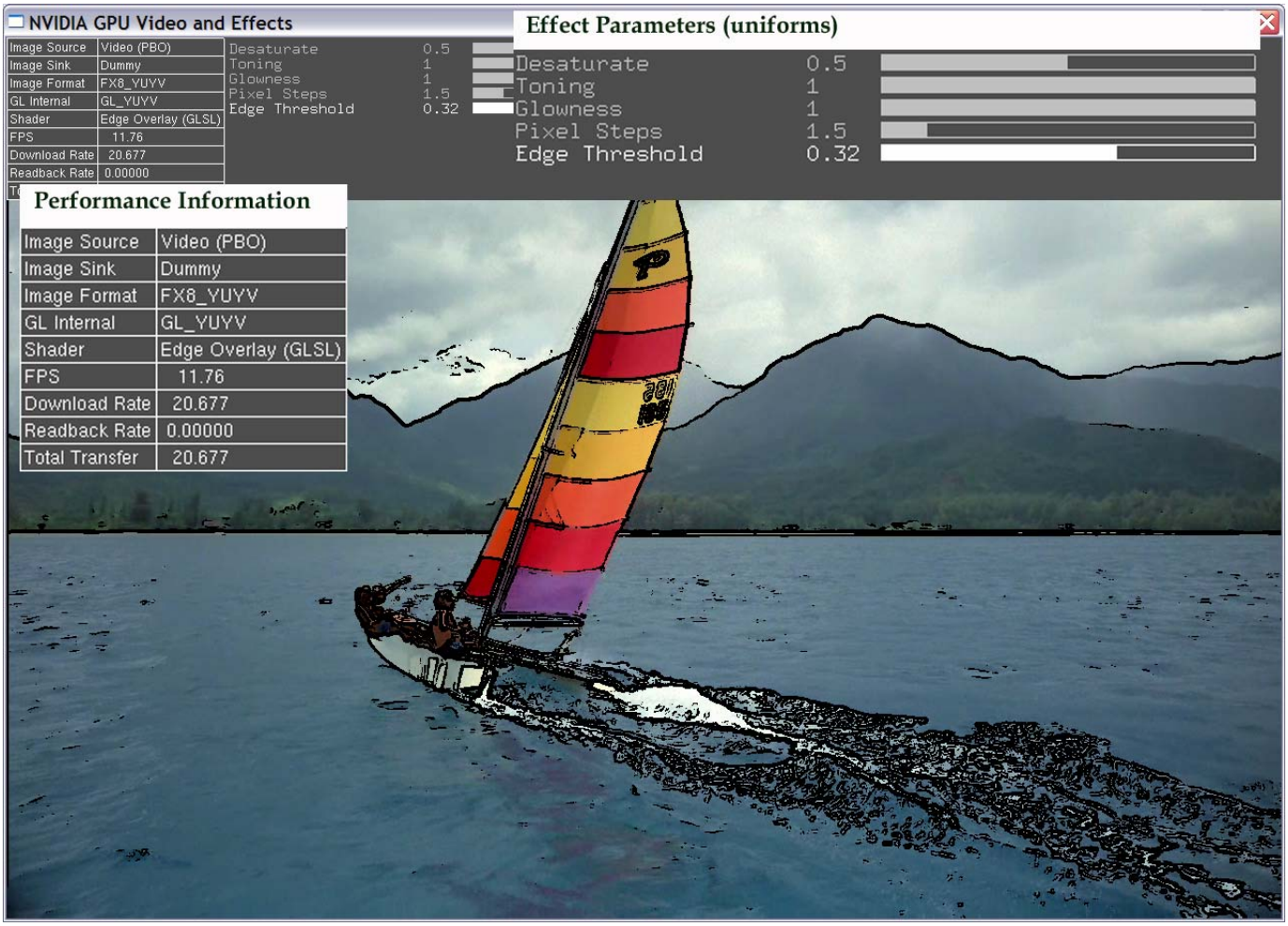
1. ImageSinkDummy which doesn't read back any data from the GPU.
2. ImagePuller which uses glReadPixels( ) in a straight forward manner to transfer the frame-buffer content back from the GPU into system memory.
3. ImagePullerPBO which uses a pixel-buffer object to readback frame-buffer data asynchronously from the GPU.

Supported Image and Video Formats

For Static Images, only OpenEXR file formats are supported.

For Video Files, any file that can be dropped into a DirectShow FilterGraph will work with this GPU video and effects demo.

The external image/video formats (i.e. formats of images stored in system memory) supported are 16bit floating point RGB and RGBA, 8 bit unsigned char RGB, RGBA, BGR, and BGRA. Texture formats supported (i.e. format of how images are stored on the GPU) are 16 bit floating point RGB and RGBA, 8 bit unsigned integer RGB, and RGBA, as well as a 16 bit packed YUYV format used for video.



Demo App in Action

Here is a screen shot of the **gpu_videoeffects** demo with different screen elements magnified. The ***performance info*** area shows all the options used for texture download, readback, shader, image format, internal texture format, and the current transfer rates upstream, downstream, and the total.

The ***effect parameter*** region allows the user to interactively manipulate the GPU parameters. Depending on the effect chosen, there are different parameters that can be configured. To change these values, hold down the left mouse button and drag the slider to set the appropriate value.

With the right mouse button, menu options can be brought up. Source File lets the user load different static images (OpenEXR) or video files into this application. Source Upload lets the user toggle between a static image or video file. Image Readback can enable the copying of the final frame back to system memory. Playback Control, allows the user to "." (pause/play), ">" skip +10 seconds, "<" skip -10 seconds, (keyboard functions also enable this to be displayed). Different GPU shader effects can be applied to the Image or Video when selected with the appropriate menu option.

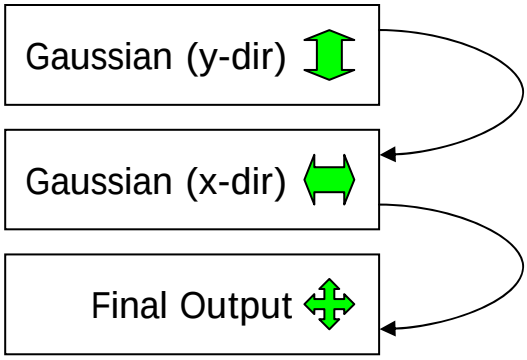
Developing effects

ProgramCg, ProgramGLSL, and ImageFilter are classes required to build GPU effects. To create a new Cg or GLSL effect, instantiate a ProgramCg or ProgramGLSL object with a Vertex and Pixel source program. These classes will load the shader code and compile it upon construction. To use this shader to render a geometry, first call the bind( ) and unbind( ) member.

To control GPU uniform parameters, fixed uniform Cg and GLSL names have been defined by this example code. Through the InteractionController class, these parameters can be updated interactively. For Cg, see uniforms.cg for a list of these fixed uniform names. GLSL uniforms are defined at the top of each Vertex and Pixel shader source file.

For more advanced effects, the ImageFilter class (nv_image_processing) provides the added flexibility of combining different effects by using p-buffers to chain these effects together. This nv_image_processing framework is a library included with the NVIDIA SDK. An article in GPU Gems, "*A Framework for Image Processing"* also shows how to build GPU accelerated ImageFilter.

Each rendering stage produces an output image that can be the source input for another stage. For the nv_image_processing framework, each ImageFilter stage can be chained to another stage by calling the setSourceOperator( ) member function (i.e. from Gaussian x-direction) and passing a pointer to the ImageFilter (i.e. Gaussian y-direction) object.



In this example, the Gaussian smoothing operation is a 2-D convolution filter that blurs images. This operation can be performed more efficiently if separated into 2-passes in the x and y direction. With the first stage applying a 1-D vertical Gaussian in the x direction. This output is applied to the input of the y direction filtering stage, where a 1-D horizontal Gaussian is done. By combine these two stages, we have a 2-D Gaussian filter.

Prototyping new shader effects are not difficult, as new shaders can be built on top of the existing examples from this demo. The NVIDIA SDK also provides many HLSL and Cg shaders, and with modifications to change the uniform names, these effects can be incorporated easily.