GPU Filtering Framework User Guide

How to create your own filter

Introduction

The NVIDIA GPU Filtering Framework provides a mechanism for developers wth the Adobe Photoshop™ SDK (available from Adobe) to quickly and easily assemble a custom, OpenGL shader-based Adobe Photoshop filter plug-in. The framework performs all relevant communication with Photoshop, as well as all the resource management necessary to support the large images that Photoshop supports.

Custom Filter Requirements

To create a custom filter using the framework, simply override the GPUFilterData class, implement all of its virtual functions, and set all of its member variables.

Many filters are applied over more than one pass, where the results from one pass are fed into the next. Separable filter kernels can be optimized in this manner, by applying a horizontal filter kernel on the first pass, and a vertical filter kernel on the next. The framework allows you to apply as many passes as you want. Each pass can use its own shader, with its own set of parameters.

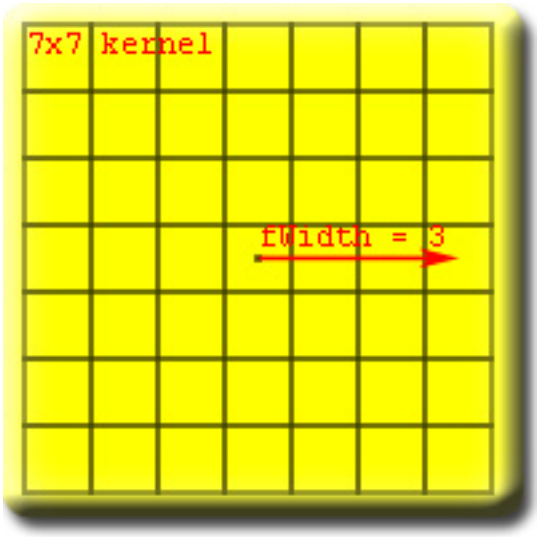

Figure 1: The filter size of a 7x7 kernel

With most image processing effects, each output pixel is only dependent on the pixels immediately surrounding it. The framework takes advantage of this in its resource management system. Because the GPU can only handle images of a certain size (4096×4096 on NV4x and NV3x cards), large Photoshop images must be broken into chunks and processed one chunk at a time by the GPU. In order to do this without creating seams in the image along the boundaries of these chunks, the framework must know the size of the area of pixels that each destination pixel is dependent on. It accepts two values for this—a **filter width** and a **filter height**, together referred to as the **filter size**. The filter size is the furthest distance away that the output pixel depends on any input pixels. For example, a 7×7 filter kernel has a filter size of 3×3,

because any output pixel depends on pixels that are, at the most, 3 pixels away in either direction.

The architecture limits the filter size to 255 pixels in any direction, in order to preserve some helpful resource management properties. This limitation applies not to each pass individually, but to the cumulative filter size of all passes. So make sure the filter size of each pass does not add up to more than 255. Future versions may lift this limitation, but it should be sufficiently large for most filters. It does pose a problem for space-warping filters, where a pixel in one corner of the image may depend on the pixel in the other corner. Again, future versions may lift this restriction. Remember, the source code for the framework is also included, allowing you to modify the framework itself, however you see fit.

The framework allows you to create a GUI, in the form of a Windows resource (or Mac resource, should the framework be ported to Macs). You can also specify a control in your GUI where the framework can render a preview image. The framework provides functions to pan and resize the preview. Zooming creates a lot of difficult resource management issues, and will be implemented in a future version.

Steps to creating a basic filter

1. **Make a copy of a sample project.** You will probably want to rename everything to make it fit the purpose of your project. Note that when you rename a file, you must delete the old file and add the new one in Visual Studio. This means that any special settings the file had before will need to be manually copied over. Sadly, if you rename an .rc file, Visual Studio will forget to include it in the final EXE. To do this right, first copy the text from the old .rc file. Then delete the reference to it in your Visual Studio project. Then add a new resource (any will do). Note that Visual Studio will automatically create a new .rc file, and its name will match the name of the project. Then open the .rc file up in the text editor, and replace its contents with the text of the old .rc file. Remember to update the reference to the .pipl file.
   Remember, you will also need the Adobe Photoshop Developers SDK, available directly from Adobe.
2. **Clean out the sample code.** All of the sample code inherits from `GPUFilterData` and `Pass`. You will need to inherit your own class from `GPUFilterData`, and in most cases you will also want to inherit your own `Pass` class. The `DoStart()` function in main.cpp serves as the equivalent of the main function. The version in the sample runs the preview dialog, and if it returns `IDOK`, executes the filter. You can change it to behave however you want. You'll also want to change the name and category of the filter in the .r file, so Photoshop identifies your filter by its proper name. Refer to the Photoshop SDK for a description of the PiPL resource (the .r file).

3. **Make a GUI.** Having a GUI will allow you to easily test your filter as you create it. You may want to delete the sample project's GUI altogether, or you can easily modify it in Visual Studio's dialog editor. To make a preview window, add a picture control to you GUI, and set `GPUFilterData::mPreview` to the resource ID. You'll need to give it to the framework later. If you don't want a GUI, you should alter the `DoStart()` function to avoid calling the preview dialog.
4. **Write the shaders.** The easiest way to do this is to write the pixel shaders in the most intuitive way possible, listing off the required parameters as you go. Then write the vertex shader to output the vertex position (the equation for this is in the sample vertex shaders, Separable-Symmetric.vs and Standard.vs) and pass along any interpolated parameters that will go to the pixel shader.
5. **Write the passes.** Override the `Pass` class, and fill in the `SetParameters()` function to send the appropriate parameters to the shaders. Keep in mind that the framework automatically sets a number of essential uniform shader parameters for you. They are listed in the shader section of the reference guide.
6. **Stitch everything together.** The shaders and passes should be constructed and destroyed along with your GPUFilterData derivative. The easiest thing to do is just declare an instance of each shader and pass as a member variable.The process of piecing everything together all occurs in the constructor of your GPUFilterData-derived class. The shaders should be compiled and linked into programs (using `GLShader::Validate()` and `GLProgram::Validate()`). Set each pass's program and filter size.
7. **Debug.** Make sure you have Photoshop CS installed. Going into the project's linker settings, change the output file from `Debug/<something>.8bf` to `<Photoshop Dir>/Plug-ins/Filters/<something>.8bf`. Because the output file is technically a DLL, Visual Studio will ask you to locate the executable that will run your DLL when you try to start a debugging session. Point it to the Photoshop executable. Photoshop will start, and you can look for your filter in the Filters menu along with all the others.

Architecture of the Filter Framework

The Modules

There are six core objects that describe the overall behavior of the filter framework.

- **`GPUFilterData`** – Describes a filter and manages the custom GUI and shader parameters.
- **`Pass`** – Describes a single filter pass. Each pass object specifies a shader, a filter size, and a function to set the shader's parameters.
- **`GPUFilter`** – Performs the filter operation on and sends the results back to Photoshop. This is probably going to be the last step in the process.
- **`PreviewManager`** – Manages a GUI preview window, including the process of drawing and displaying the preview. A properly designed GUI will allow the user to modify parameters, preview the results of those parameters, and choose whether to apply the filter or cancel the operation.
- **`TileManager`** – Manages the loading and unloading of image tiles, and does some convenient support math for tile-related operations. It also manages a set of textures that are used to represent tiles within OpenGL.
- **`Applicator`** – Does all the OpenGL work to apply the filter to a portion of the image. First, a small enough portion of the image, with a certain "fluff" factor applied to prevent border artifacts, is rendered to the applicator. Then you tell the applicator to apply the filter, and when it's done you can read the filtered results back from the applicator. `PreviewManager` and `GPUFilter` both use this to do the filtering operation.

There are also a few support classes that the framework makes use of that you should make yourself familiar with, but that aren't a foundational part of the architecture. Detailed documentation of these classes is not provided here, but they're simple enough that a quick look at their header files should explain them adequately.

- **`GLShader`** – Represents an OpenGL shader object—either a fragment shader or a vertex shader. They accept and compile source code.
- **`GLProgram`** – Represents an OpenGL program object. Add `GLShader` objects to one of these and link them together to create a usable object. Once successfully linked, a `GLProgram` can be enabled and used in the OpenGL render state. `GLProgram` objects also provide access to a shader's uniform parameters, which the CPU is responsible for setting.
- **`GLManager`** – This manages the basic initialization of OpenGL. Because OpenGL has an annoyingly long initialization sequence, that sequence was compartmentalized into a single manager class. The class can create dummy rendering and device contexts, transfer resources and settings from an old context to a new one, and also manage the destruction of the contexts. Mostly, it's a frustration-saver.

The Singleton Pattern

The architecture makes frequent use of the singleton pattern. The singleton pattern is useful when you have a class that you know will only have one instance. For example, it would be rather useless to have multiple TileManager classes, because there's only one Photoshop image being processed and only one interface to Photoshop to work with.

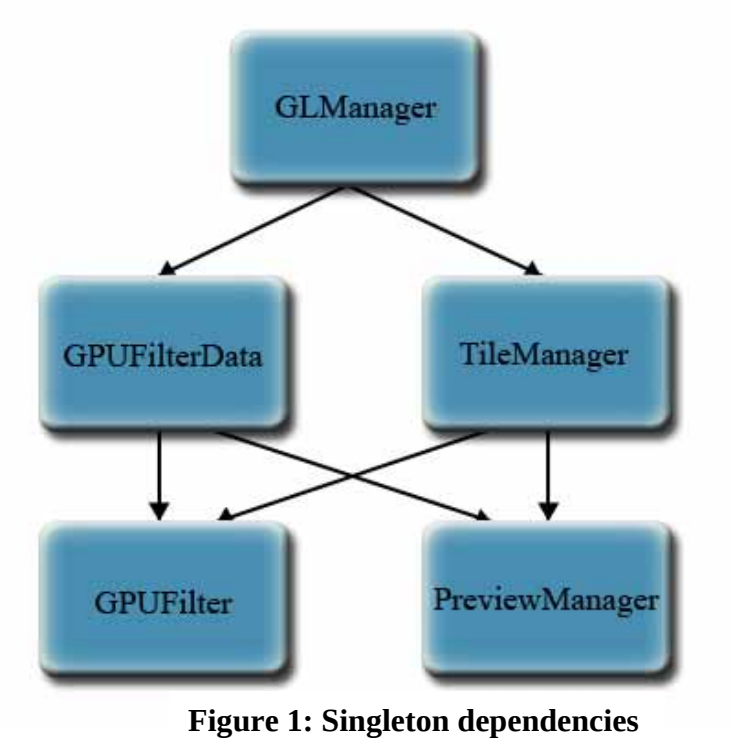

Figure 1: Singleton dependencies

Singletons maintain one static reference to themselves, so if you want to use one, simply make sure it's created, and then call its static `Get()` function to retrieve the instance of the object. Be careful when using singletons, though. If your code uses singletons too liberally, it'll be impossible to reuse your code without refactoring to accept function parameters instead of accessing a singleton.

The following modules are all singletons: `GPUFilterData`, `GPUFilter`, `PreviewManager`, `TileManager`, `Applicator`, `GLManager`, and `StopWatch`. In order to access the instances of each of these modules, call their respective static `Get()` functions. If no module exists, it will return `NULL`.

One weakness of singletons is that they may have a specific order in which they must be created because one singleton may depend on another, but they really have no natural way of enforcing any such constraint. So, it's important to note the dependency chain of these modules. The `GLManager` module doesn't require any other singletons, so initialize it first (refer to `DoStart()` in `main.cpp`). The `GPUFilterData` and `TileManager` modules requires `GLManager`. The `GPUFilter` and `PreviewManager` modules each require both the `GPUFilterData` module and the `TileManager` module. The `Applicator` module requires `GPUFilterData`, although the `GPUFilter` and `PreviewManager` modules take the responsibility of creating the `Applicator`, so unless you dig deep into the framework you'll never need to worry about it. Here's a sample initialization sequence that satisfies all of the dependencies, and can be found in the sample `DoStart()` function:

```
// Set up OpenGL
GLManager theGL(
  GetDLLInstance(
    (SPPluginRef)gFilterRecord->plugInRef));

if(!glh_init_extensions(REQUIRED_EXTENSIONS))
```

```
  throw "Couldn't support all the necessary OpenGL
extensions";

// Create everyone's favorite singletons
TestFilterData data; // derived from GPUFilterData
TileManager tiler;
PreviewManager preview;
GPUFilter filter;
```

Process Overview

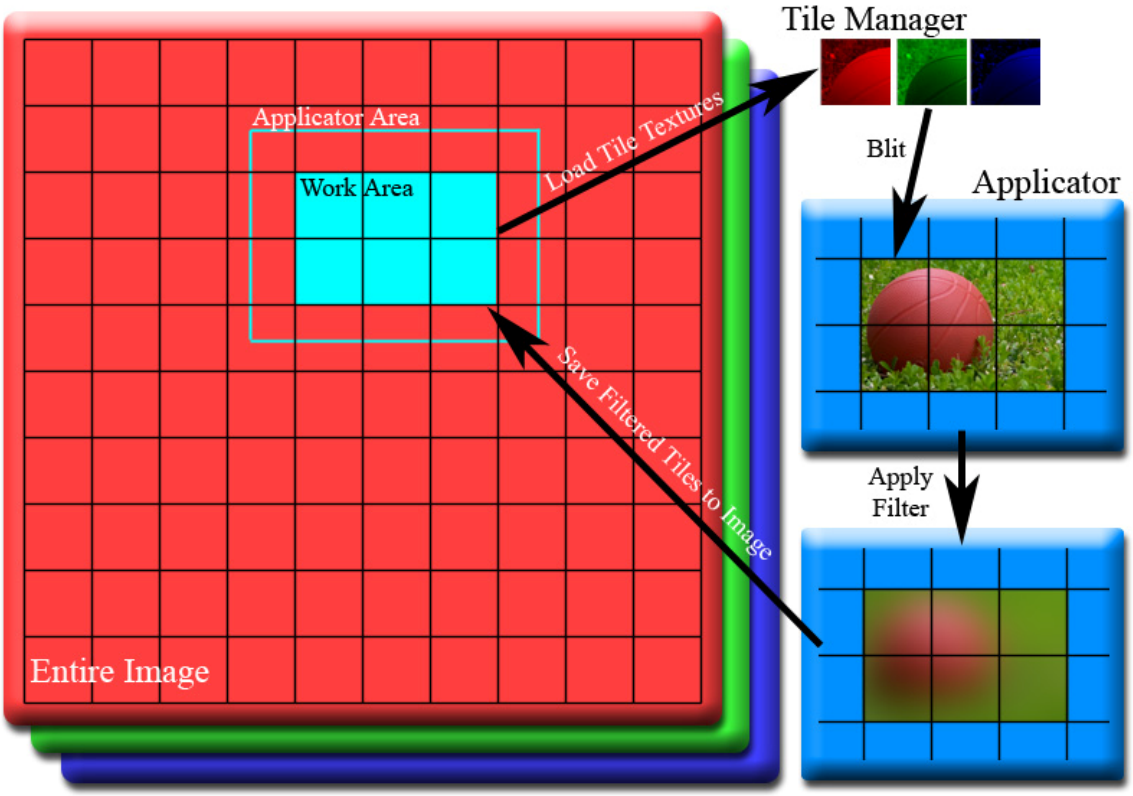

Figure 2: Applying a filter to the image.

The Tile Manager

Photoshop supports images that can be many, many times larger than the largest texture supported on the GPU. So the first step to applying a fitler on the GPU is to break the image up into chunks small enough for the GPU to handle. Thankfully, Photoshop stores its images as a grid of relatively small tiles. These tiles are assumed by the framework to always be small enough for the GPU to handle.

Also, Photoshop stores each color channel on a separate plane, and can therefore store an arbitrary number of color channels for each image. The GPU only stores up to 4 color channels per texture, and it interleaves them together.

To reconcile these differences between the two architectures, the framework uses a tile manager. The manager can load each color component of each tile as a separate texture. Once loaded, the tile manager has the ability to render the first 4 of a tile's color channels to the frame buffer. By rendering a set of contiguous tiles, the tile manager can easily and efficiently construct a region of the image inside the GPU, and in a format that the GPU was designed to work with.

The tile manager takes advantage of the fact that most of the tiles in the image are all the same size, and the few that aren't are smaller. When a tile is no longer needed, the manager doesn't delete the texture. Instead, it flags it as no longer being used, and the next time a tile needs to be loaded onto the GPU, the manager can simply use the old texture, thus saving OpenGL the hassle of reallocating the space required for the texture.

The manager maintains two linked lists: one lists all of the texture objects that are unallocated, and one lists all of the texture objects that are. The texture objects that are allocated are listed in order from most recently used to least recently used. This is so that when the manager runs out of memory, it can automatically delete the least recently used texture, and load up the new one in its place. Thus, the tile manager will never fail due to lack of memory. Also, these two linked lists allow all useful operations to be performed on the texture pool in constant (O(1)) time: allocation, deallocation, and maintaining a sorted list from most recently used to least recently used.

The Applicator

The applicator class allocates a double-buffered PBuffer to perform the filtering operation. Once the applicator is activated, any OpenGL draw commands will draw directly to the back buffer of the applicator. The tile manager is used to load appropriate tiles from Photoshop and render them into the applicator.

Once the applicator has a complete chunk of the source image in its buffer, it performs a buffer swap. This means that the source image is now in the front buffer. The applicator then binds the front buffer as a texture, sets up the first filter pass, and renders it to the back buffer. Once rendered, it does a buffer swap again. It will do this until there are no more passes to render. When it's done, the filtered image will be in the front buffer, which makes it easily accessible through standard OpenGL calls.

Applying a Filter

The tile manager and the applicator together are very powerful, and can take care of a large portion of the work required to perform a filter. Two problems remain, however: how do you partition the image into chunks, and how do you send the filtered image from the applicator back to Photoshop?

The GPUFilter class handles both of these problems. It partitions up the image by starting in the top-left corner, and "targeting" it with the applicator (see the reference guide's description of `Applicator::Target()`). It then figures out how many tiles it can completely filter at once, and tell the applicator to filter them. Once filtered, the GPUFilter module reads the data back from the applicator, and sends it off to Photoshop. Then it targets the tiles immediately to the right of the tiles it just finished, and filters them. It does this until it has filtered a horizontal strip all the way across the top of the image.

It then moves down to the tiles directly underneath the strip it just rendered, and starts filtering from right to left. It will continue to sweep back and forth on its way down the image until there is nothing left for it to filter.

The reason it sweeps back and forth, as opposed to sweeping left to right across each row, is to maximize the effectiveness of the memory manager's least-recently-used assumption. If the memory manage is forced to offload some textures as a filter is applied from left to right, the textures that are going to be offloaded are the ones on the far left. It's best to use the ones that are still in memory by sweeping back from right to left, rather than force the offloaded ones to get swapped in again as you go back to the left side.

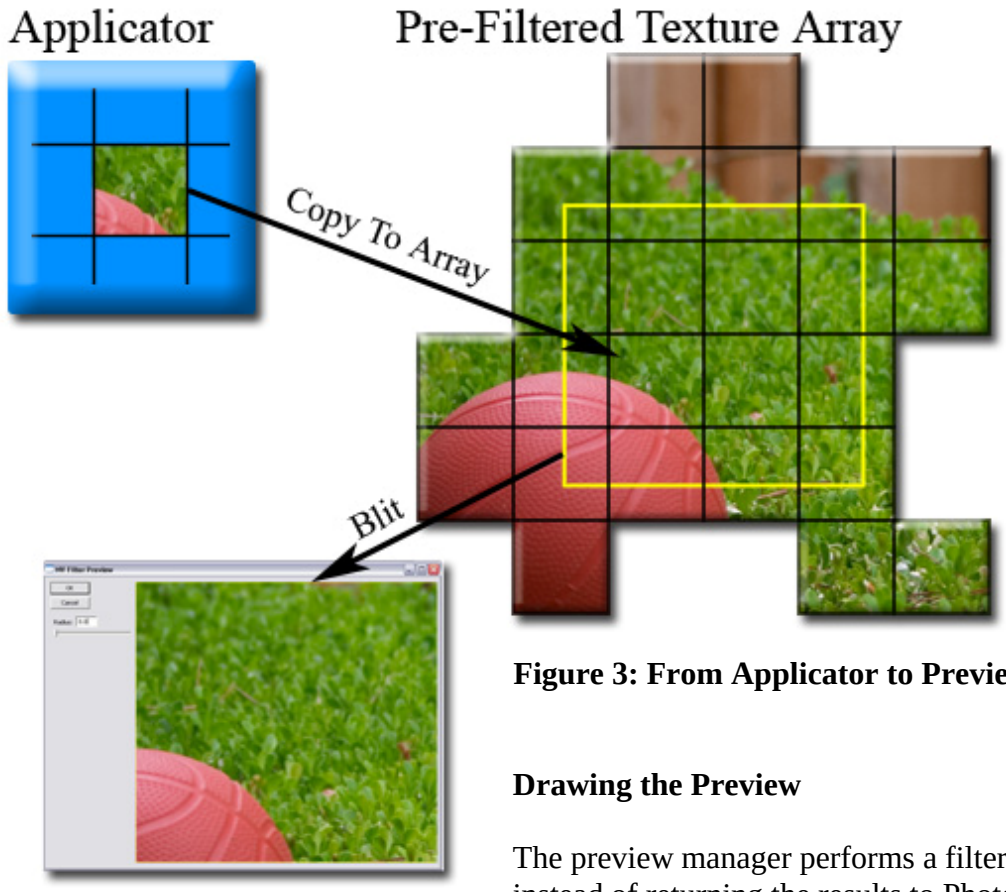

Figure 3: From Applicator to Preview

Drawing the Preview

The preview manager performs a filter operation and, instead of returning the results to Photoshop, saves the results of each filtered tile in a texture and blits the pre-filtered texture onto the screen as necessary.

Instead of trying to filter as many tiles at once as it can, the preview manager sets up its applicator so that it can only filter a single tile at a time. Once it has done this, the filtered tile is copied into a texture.

The preview manager keeps a 2D array of pre-filtered textures. As the user pans around in the preview window, this array moves along with it. When a texture gets moved, it must be re-filtered.

However, the pre-filtered textures are no ordinary 2D array. As the user pans around, instead of having the entire array move at once (which would then require an entire row or column to be re-filtered at once) the array staggers itself based on the position of the preview window. As the preview window pans across a tile, the textures in the array will be moved and updated one at a time, effectively amortizing the cost of re-filtering each texture.

Module Reference Guide

This reference guide describes, in vivid detail, each of the major architectural modules that compose the GPU filtering plug-in framework.

The framework also uses a number of global variables that aren't technically a part of any specific module, although they are usually closely associated with one. For example, nearly all constants are declared globally.

List of Modules

Shaders
GPUFilterData
Pass
GPUFilter
PreviewManager
TileManager
  TileManager::PSTile
Applicator

Global Variables

```
const int AllocateThisManyTextures = 204;
const int MaxColorChannels = 4;
const int MaxFilterSize = 255;
const VPoint ApplicatorSize = {2048, 2048};
const VPoint MaxViewArea = {1536, 1536};
FilterRecord *gFilterRecord;
int32 *gData;
int16 *gResult;
SPBasicSuite *sSPBasic = NULL;
```

AllocateThisManyTextures – Declared in TileManager.h. This tells the tile manager how many textures in can allocate before it must start recycling them. The number 204 was chosen because it is in the acceptable range to efficiently work on 128 MB cards (if you're working with REALLY big images and have a 256 MB card increasing this may help a little) and since most images have 4 color channels or less, a multiple of 12 will align well with 4, 3, and 2 color channels.

MaxColorChannels – Declared in TileManager.h. This tells the tile manager how many texture handles each tile will need (1 for each color channel).

MaxFilterSize – Declared in GPUFilter.h. This puts a limit on how large the filter size can be. Beware! If `(2*MaxFilterSize + TileSize > ApplicatorSize)`, then the system will break, because it won't be able to filter one whole tile at a time.

ApplicatorSize – Declared in GPUFilter.h. This specifies the size of the applicator buffer. The largest size that the latest GPU hardware supports is 4096×4096, but

2048×2048 is more largely supported, and the performance impact is negligible, except perhaps on enormous images (many times larger than 2048×2048) with enormous filters.

MaxViewArea – Declared in PreviewManager.h. This specifies the largest that the preview window is allowed to get. If you attempt to resize a preview window larger than this, it will be clamped at the maximum, and the resulting output will stretch as the window gets larger. This is somewhat arbitrary, but keeping it reasonable really helps minimize the preview window "stuttering" as you pan across it.

gFilterRecord – Declared in main.cpp. This is the data structure that Photoshop passes off to the plug-in. For more information regarding it, refer to the Photoshop API Guide.

gData – Declared in main.cpp. Not currently used. This maps directly to the data parameter passed directly to PluginMain(). Refer to the Photoshop API Guide for details.

gResult – Declared in main.cpp. The framework sets this to 0 before returning if the filter was successful, and 1 if it failed.

sSPBasic – Declared in main.cpp. This is a pointer to the SP Basic suite, used for accessing Photoshop's various API function suites. Refer to the Photoshop API Guide for details.

Shaders

The framework uses GLSlang programs to execute each pass of the filter. This reference assumes the reader is familiar with GLSlang. For a good introduction to GLSlang, refer to Randi J. Rost's *OpenGL Shading Language.* Each program requires both a vertex and fragment shader. The vertex and fragment shaders should each be contained within their own files. The framework uses the .vs extension for vertex shaders, and the .fs extension for fragment shaders.

In order to work correctly with the filtering framework, your shader must fulfill a number of requirements. Additionally, the framework provides a number of uniform parameters you can (and will probably need to) use to fulfill those requirements.

NOTE: The framework never uses the *gl_ModelViewMatrix* parameter, so your filter may use it as an easy means of passing a matrix parameter to your shader.

Coordinate Spaces

In 3D graphics, the important distinction is made between local object space, world space, eye space, and screen space. These terms don't apply to image processing, so we must introduce a couple new coordinate spaces so you can understand how the shaders must process vertex and pixel positions. All of these coordinate spaces exist in 2 dimensions.

Image space – Image space vectors represent the distance, in pixels, from the upper-left corner of the overall image. All framework parameters passed to the shaders are in image space.

Chunk space – Chunk space vectors represent the distance, in pixels, from the upper-left corner of the current chunk. The chunk is the portion of the image currently being processed by the GPU. To transform from image space to chunk space, simply subtract *chunkPos* from the image space vector.

Applicator space – Applicator space is the framework's equivalent of screen space, the only difference being that the frame buffer is off-screen. The only variable that the framework uses in applicator space is *gl_Position*, set in the vertex shader. The projection matrix will transform any image space vector into an applicator space vector.

Color Spaces

Photoshop supports RGB, CMYK, Lab, and grayscale color spaces. The framework does not convert the image to RGB before passing it to the shader. Nor does the framework currently have any standard way of communicating the image's color space to the shader. This shouldn't be difficult to implement manually, so this will probably be included in a future version of the framework, but in the meantime it shouldn't be much trouble to

implement it yourself. The easest method would be to #define some environment variable that would tell the compiler which version of the shader to compile.

NOTE: The color space of the image is accessible through `gFilterRecord->imageMode`. Refer to the Photoshop SDK for details.

The most important thing to note about the different color spaces is how they are encoded in the shader. The following figure indicates how each color component maps to the shader's RGBA format. First, notice that CMYK color does not handle any alpha channel. This is because the GPU can only gracefully handle 4 color channels. With the recent introduction of multiple render targets (MRT) the shader could handle up to 16 color channels, but that is not yet implemented in the framework. Secondly, notice that the green and blue components are both 0 in grayscale format.

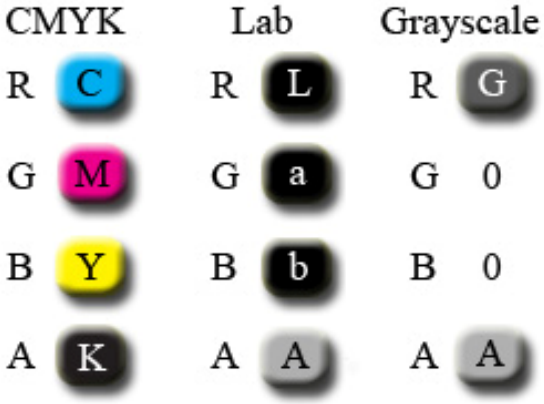

Figure 4. How other color spaces map to RGBA shader components

Framework Parameters

```
vec4 gl_Vertex;
mat4 gl_ProjectionMatrix;
vec4 gl_Position;
vec4 gl_FragColor;
sampler2DRect chunk;
vec2 chunkPos;
vec2 chunkSize;
vec2 imageSize;
vec2 filterSize;
```

gl_Vertex – This is an internal OpenGL variable. It's valid only in the vertex shader. In the case of the framework, this is the vertex position in image space. The *z* and *w* coordinates are always set to 0 and 1, respectively.

gl_ProjectionMatrix – This is an internal OpenGL variable. It's valid in both the pixel and vertex shader. In the case of the framework, this matrix transforms a vector from image space to applicator space, assuming the *z* and *w* coordinates of the input vector are 0 and 1, respectively.

gl_Position – This is an internal OpenGL variable. It's write-only, and serves as the position output of the vertex shader. In the framework, this should be the transform of *gl_Vertex* into applicator space.

gl_FragColor – This is an internal OpenGL variable. It's write-only, and serves as the color output of the fragment shader. Photoshop can use RGB, CMYK, Lab, and grayscale color spaces, and the framework does no conversion

chunk – This is the texture that contains the source chunk of the image. To read from the image, you must first transform an image space vector into chunk space by subtracting *chunkPos* from the image space vector.

chunkPos – The image space position of the upper-left corner of the current chunk.

chunkSize – The size of the current chunk, in pixels.
imageSize – The size of the overall image, in pixels.
filterSize – The filter size. The shader should not try to read pixels from the chunk farther away from the output pixel's position than the filter size.

GPUFilterData

Provides the framework necessary to describe a filter operation. Override this and initialize all the necessary members to describe your own filter.

Data Members

```
list<Pass *> mPasses;
int mGUI;
int mPreview;
static GPUFilterData *me;
```

mPasses – Public. A list of pointers to each pass required to perform a filter, in the order they will be applied. You are responsible for the creation and destruction of the passes themselves.

mGui – Protected. The resource identifier of the GUI dialog. Only the preview manager uses this, so if you don't want a GUI, don't run `PreviewManager::ShowPreview()`.

mPreview – Protected. The resource identifier of the preview window within the GUI dialog. Set this to zero if your GUI doesn't have a preview window.

me – Private. The pointer to the single instance of this object. Use `GPUFilterData::Get()` to retrieve this.

Functions

```
static GPUFilterData *Get();
```

Public. Returns the pointer to the single instance of a GPUFilterData object.

```
GPUFilterData();
```

Constructor. Clear variables and sets the singleton pointer. Throws an exception if another GPUFilterData object already exists.

```
~GPUFilterData();
```

Destructor. Clears all variables.

```
virtual int GetGUIResID();
```

Public, overridable. Returns the resource ID of the GUI dialog.

```
virtual int GetPreviewResID();
```

Public, overridable. Returns the resource ID of the preview window within the GUI dialog.

```
virtual INT_PTR HandlePreviewMessages(
  HWND hwnd,
  UINT msg,
  WPARAM wparam,
  LPARAM lparam);
```

Protected, overridable. Override this to provide custom functionality to your GUI dialog. Refer to the DialogProc function described in the MSDN library for details.

hwnd – handle to the window the message is for.
msg – The message being sent.
wparam – A parameter whose meaning depends on the message.
lparam – Another parameter whose meaning depends on the message.

Pass

Describes a single filter pass. Pass is a struct, not a class, so every member is public. Override this to supply your own parameter-setting function.

Data Members

```
GLProgram *program;
float fWidth;
float fHeight;
```

program – Pointer to the GLSlang program used to apply the filter.
fWidth – Filter width of this pass. This is the maximum distance along the X-axis between any output pixel and any input pixel on which it depends.
fHeight – Filter height of this pass. This is the same and fWidth, except along the Y-axis.

Functions

```
virtual void SetParameters(GPUFilterData *filter);
```

Overridable. The applicator calls this right before it draws any quad filtered with ths pass. Use this to set any custom uniform parameters that your shader may require.

filter – pointer to the GPUFilterData object that this pass is a member of.

GPUFilter

Uses an applicator (which it creates) to filter the Photoshop selection (or entire image, if there is no selection) and write the results back to Photoshop.

Data Members

```
static GPUFilter *me;
VPoint mImageSize;
VRect mImageRect;
VPoint mChunkPos;
VPoint mNextChunk;
bool mGoingRight;
bool mNextGoingRight;
VRect mRelevantTiles;
```

me – Private. The pointer to the single instance of this object. Use `GPUFilter::Get()` to retrieve this.

mImageSize – Private. Stores the size of the selection rectangle (the entire image is there is no selection).

mImageRect – Private. Stores the selection rectangle (the entire image if there is no selection).

mChunkPos – Private. Stores the current position of the chunk. The chunk is the area of the image that the applicator will render all at once. This is just an intermediate value used by the NextChunkPos() and ShouldRetireTileSource() functions.

mNextChunk – Private. An intermediate value that stores the next chunk position.

mGoingRight – Private. True if the algorithm is moving right across the image as it filters it. False if it's going left. The algorithm moves back and forth across the image instead of just left-to-right, in order to maximize the efficiency of the least-recently-used property of the texture manager.

mNextGoingRight – Private. An intermediate value used by ShouldRetireTileSource(). True if the next applicator iteration is going to be going right, and false if it will be going left.

mRelevantTiles – Private. Represents the tiles relevant to the current applicator chunk's position. There are all the tiles visible when rendered to the chunk.

Functions

```
static GPUFilter *Get();
```

Public. Returns the pointer to the single instance of a GPUFilterData object.

```
GPUFilter();
```

Constructor. Initializes variables in preparation for running ApplyFilter() and sets up the singleton. Throws an exception if a GPUFilter object already exists.

```
~GPUFilter();
```

Destructor. Sets the singleton pointer to null.

```
void ApplyFilter();
```

Public. Applies the filter (the current GPUFilterData instance) to the Photoshop image, and returns the result back to Photoshop. The input image may be much larger than the largest image the GPU can process at once, so this function efficiently breaks the image up into chunks that the GPU can handle individually.

```
bool NextChunkPos(VPoint &chunk);
```

Protected. Finds the next chunk position to filter. Returns true if the algorithm will be moving right across the image during the next chunk, and false if it'll be moving left.

chunk – [Out] When NextChunkPos() returns, this parameter will contain the next chunk position.

```
bool ShouldRetireTileSource(VPoint &tilepos);
```

Protected. Returns true if the given tile's textures can be retired. It determines this based on the current chunk position, and the remaining chunks to be filtered.

```
void DoChunkFilter();
```

Private. This calculates the current chunk area, renders the source image to the applicator, and executes the filter on the chunk.

```
void DoChunkDataRead();
```

Private. This reads back the results of a filtered chunk and returns the results to photoshop.

PreviewManager

This runs the GUI supplied in `GPUFilterData::mGUI`, and manages any preview window inside it (specified by `GPUFilterData::mPreview`).

Data Members

```
static PreviewManager *me;
bool mPreviewInitialized;
VRect mArea;
VPoint mTexPos;
VPoint mTileArea;
GLuint **mTextures;
bool **mDirty;
bool *mColShifts;
bool *mRowShifts;
GLShader mBlitVS;
GLShader mBlitFS;
GLProgram mBlit;
GLuint mKTextures[8];
GLuint mGridTex;
```

me - Private. The pointer to the single instance of this object. Use `PreviewManager::Get()` to retrieve this.

mPreviewInitialized – Private. True if the preview manager has been initialized. This is true after a call to InitPreview() and before a call to CleanupPreview().

mArea – Private. The rectangle of the image visible in the preview window.

mTexPos – Private. The preview manager maintains a grid of tile-sized textures. Each texture contains the filtered result for that particular tile. This way the preview manager doesn't have to re-apply the filter every time the preview window is resized or panned. This grid of textures is smaller than the overall image, and so only represents a portion of it. This member stores the position of this grid, in tile space.

mTileArea – Private. The number of tile textures stored in each dimension. See mTexPos for more details.

mTextures – Private. The OpenGL names of all the tile textures. See mTexPos for more details.

mDirty – Private. Marks each tile texture as being dirty (true) or clean (false). If a tile texture is dirty, the preview manager must re-apply the filter to that tile before it can be displayed in the preview window.

mColShifts – Private. Each column of tile textures (see mTexPos) can be shifted one tile to the left or right. By progressively shifting each column up or down one at a time as the user pans across the image, the preview manager can update the texture tiles in a smoother, amortized way, so it isn't as intrusive to the user

experience. These are True if the column is shifted one up, and false if shifted one down, and the size of the array is equal to the number of columns.

mRowShifts – Private. Same as mColShifts, except false represents left, true represents right, and the size of the array is equal to the number of rows of texture tiles.

mBlitVS – Private. A vertex shader that just blits a tile texture to the preview window.

mBlitFS – Private. A fragment shader that blits a tile texture to the preview window.

mBlit – Private. The GLSlang program composed of mBlitVS and mBlitFS.

mKTextures – Private. This is a 4D lookup table for CMYK->RGB conversion, used by the mBlit program for CMYK images. There are 8 3D textures. Each texture represents a different value of K (evenly spaced in [0,1]), while the range of values of C, M, and Y are stored in the x, y, and z, components of each texture, respectively.

mGridTex – Private. This is a 2x2 texture representing the grid that shows up "behind" images in the preview window when they are translucent/transparent. It looks the same as Photoshop's (except for grayscale images).

Functions

```
static PreviewManager *Get();
```

Public. Returns the pointer to the single instance of a PreviewManager object.

```
PreviewManager();
```

Constructor. Initializes variables, and the singleton. More importantly, it loads up the appropriate preview blitting shader for the given image color format. Accepted color formats are: RGB, CMYK, Lab, and grayscale. Throws an exception if another PreviewManager object already exists.

```
~PreviewManager();
```

Destructor. Sets the singleton pointer to null.

```
int ShowPreview();
```

Public. Displays the preview window. The return value is equivalent to the nResult parameter passed to the EndDialog() function in the Windows Platform SDK. Refer to the MSDN library for more details.

```
void Pan(VPoint &pan);
```

Public. Pans the preview window across the image. Pan() automatically ensures that you can't pan outside the bounds of the image, and it automatically redraws the preview window after the pan has occurred.

pan – The relative distance, in pixels, you want the preview window to move. A positive horizontal movement will move the preview window to the right across the image, thus making the image appear to move left on the screen.

```
void Resize(VPoint &size);
```

Public. Resizes the preview window. Like Pan(), this function prevents you from moving the preview area to anywhere outside the bounds of the image. It will also prevent you from setting size of the preview area to something larger than the preview manager's resource management system can handle. Currently, this size is hard-coded at 1536×1536.

```
void Draw();
```

Public. Draws the preview window. If there are visible tiles that have been filtered yet, the filter is applied and the results are drawn.

```
void ParamsChanging();
```

Public. Tells the preview manager that a shader parameter is currently changing (as if a slider control were being dragged, or a text box was being modified). This marks all visible tile textures as dirty. During the next Draw() command, any visible tile textures that are dirty will be re-filtered.

```
void ParamsChanged();
```

Public. This is the same as ParamsChanging() except all tile textures are marked dirty, not just the visible ones.

```
INT_PTR HandlePreviewMessages(
  HWND hwnd,
  UINT msg,
  WPARAM wparam,
  LPARAM lparam);
```

Protected. This handles messages sent to the GUI window. It only handles messages related to drawing the preview window. On `WM_INITDIALOG` it initializes the preview window (see InitPreview()) and on `WM_DESTROY` it cleans up the preview window (see CleanupPreview()), and on `WM_PAINT` it draws the preview window (see Draw()).

```
void InitPreview();
```

Private. This initializes some resources that can only be initialized after the GUI window has been created. This function is called in the `WM_INITDIALOG` handler of HandlePreviewMessages().

```
void CleanUpPreview();
```

Private. This cleans up resources initialized in InitPreview().

```
void ApplyFilter(VPoint &tile);
```

Private. Uses an Applicator object to apply the filter to a single tile, and then saves the results to a texture, which can be simply blitted to the preview window, so that the preview window can pan and resize without having to re-filter everything.

TileManager

This class manages all the tiles that compose a Photoshop image. It can load the color channel of each tile up as a texture, blit a tile to the current draw buffer, and can dynamically manage large numbers of textures representing tiles in a memory manager that uses a least-recently-used heuristic to selectively offload textures when there are too many.

Member Types

```
typedef std::pair<int,int> LRUData;
```

> LRUData creates a kind of doubly-linked list. The first element of the pair is the index of the previous element in the list, and the second pair in the index of the next element in the list.

```
struct PSTile {…};
```

> Stores the basic information necessary for each Photoshop tile. See below: `TileManager::PSTile`.

Data Members

```
static TileManager *me;
PSTile **mTiles;
VPoint mTileSize;
VPoint mNumTiles;
int mBitsPerChannel;
int mNumChannels;
short *mMagic;
GLuint *mTextures;
LRUData *mLRUTable;
int mMRUTexture;
int mLRUTexture;
int mUnallocated;
int mNumTextures;
GLShader mCompositionVS;
GLShader mCompositionFS;
GLProgram mComposition;
WORD *mBSTexData;
static VRect mZero;
```

me - Private. The pointer to the single instance of this object. Use `TileManager::Get()` to retrieve this.

mTiles – Private. 2D array of PSTile structures representing the tiles composing the image.
mTileSize – Private. The standard tile size of each tile, in pixels.
mNumTiles – Private. The number of tiles in each dimension.
mBitsPerChannel – Private. 8 or 16. The GPU filtering framework doesn't support 16 bit-per-component images right now, so unless you choose to modify it this will always be 8.
mNumChannels – Private. The number of color channels in the image.
mMagic – Private. An array of magic numbers, used for validating handles to textures in the LRU table. When a handle is freed, the magic number is incremented so that the handle that was freed is no longer valid, because its magic number no longer matches the one in the system.
mTextures – Private. An array of texture objects. This array is organized by mLRUTable. It's actually a linked list in array form. There are two linked lists that run through this array: the unallocated list, and the LRU list. The first element of the unallocated list is referenced by mUnallocated, and the ends of the LRU list are referenced by mLRUTexture and mMRUTexture.
mLRUTable – Private. An array whose elements correlate to the elements in mTextures, and describe the linked list encoded therein. See the description of LRUData.
mMRUTexture – Private. The index of the most-recently-used texture.
mLRUTexture – Private. The index of the least-recently-used texture.
mUnallocated – Private. The index of the most recently unallocated texture.
mNumTextures – Private. The number of texture slots in the texture manager. In short, it is the size of the following arrays: mMagic, mTextures, and mLRUTable.
mCompositionVS – Private. The vertex shader that composes the different color channel textures into one swizzled result.
mCompositionFS – Private. The fragment shader that goes along with mCompositionVS.
mComposition – Private. the GLSlang program that combines mCompositionVS and mCompositionFS to create the program that combines up to 4 color channel textures into one swizzled result.
mBSTexData – Private. The texture manager saves OpenGL texture objects so that as tiles are uploaded as textures, and subsequently freed, the manager can reuse old textures, since all tiles are generally the same size. However, if a tile is an odd size (say the lower-right corner tile), the texture manager must fill the remaining data with garbage in order to fill out a tile-sized texture. This member provides that garbage data.
mZero – Private. A rectangle of all zeroes. It's just a convenience variable, since the system compares rectangles to the zero rectangle in many places.

Functions

```
static TileManager *Get();
```

Public. Returns the pointer to the single instance of a TileManager object.

```
TileManager();
```

Constructor.

```
~TileManager();
```

Destructor.

```
int GetBitsPerChannel();
```

Public. Returns the number of bits per color channel. See `TileManager::mBitsPerChannel`.

```
int GetNumChannels();
```

Public. Returns the number of color channels in the image. See `TileManager::mNumChannels`.

```
const VPoint &GetNumTiles();
```

Public. Returns the number of tiles in each dimension. See `TileManager::mNumTiles`.

```
const VPoint &GetStandardTileSize();
```

Public. Returns the dimensions, in pixels, of a single, uncropped tile. Note that tiles on the far right or bottom of the image will probably be cropped. See `TileManager::mTileSize`.

```
GLProgram &GetComposer();
```

Public. Returns the GLSlang program that renders up to 4 color channel textures together into the frame buffer.

```
VRect GetTouchingTiles(VRect &area);
```

Public. Returns a range of tiles touching the area of the image specified by area.

area – A rectangular portion of the image, in pixels.

```
VRect GetContainedTiles(VRect &area);
```

Public. Returns a range of tiles wholly contained within the area specified by area.

area – A rectangular portion of the image, in pixels.

```
GLuint GetTexture(int handle);
```

Public. Returns an OpenGL texture name represented by the given handle. Handles are produced by the texture manager with calls to CreateTexture.

handle – A handle to a texture.

```
int CreateTexture(VPoint &tilepos, int channel);
```

Public. Creates an OpenGL texture out of the given color channel of the given tile. Returns a handle to the texture. You can resolve a handle into an OpenGL texture object by calling GetTexture().

tilepos – Index of the tile to create a texture for.
channel – Index of the color channel to make a texture out of.

```
void DeleteTexture(int handle);
```

Public. Deletes a texture handle from the texture manager.

handle – The texture handle to delete.

```
void Use(int handle);
```

Public. Marks a texture as having been used. This moves it to the bottom of the least-recently-used list.

handle – The texture handle to “use.”

```
VPoint GetTileSize(VPoint &t);
```

Public. Returns the size of the given tile. Because tiles on the right and bottom of the image can get cropped, not all tiles are the same size.

t – Index of the tile whose size is to be returned.

```
VPoint GetTilePos(VPoint &t);
```

Public. Returns the position of the given tile.

t – Index of the tile whose position is to be returned.

```
VRect GetTileArea(VPoint &t);
```

Public. Returns the rectanglular area that the tile covers.

t – Index of the tile whose rectangular area is to be returned.

```
void RenderSource(
```

```
  VPoint &tile,
  VRect &imageArea,
  VRect &targetArea = mZero);
```

Public. Renders a source tile to the frame buffer.

tile – Index of the tile to render to the frame buffer.
imageArea – Rectangular area of the image that the frame buffer represents. For most purposes you'll want to derive this from an Applicator object.
targetArea – Rectangular area that this tile represents. If this is set to all zeroes, the target area is derived from the position and size of the tile being rendered.

```
void RetireSource(VPoint &tile);
```

Public. Deletes all the textures allocated for the given tile.

tile – Index of the tile whose textures should be deleted.

```
void RenderRegion(
  VRect targetArea,
  bool invertY,
  VRect &imageArea = mZero);
```

Public. Renders an arbitrary portion of the image to the frame buffer.

targetArea – Rectangular region of the image (in pixels) that should be rendered to the frame buffer.
invertY – True if the image should be flipped vertically. The best way to figure out what this value should be is to test it.
imageArea – Rectangular region of the image (in pixels) that the frame buffer represents. If this is set to all zeroes, the image area is assumed to be equal to the entire target area.

```
void RenderRegionOnce(
  VRect targetArea,
  bool invertY,
  VRect &imageArea = mZero);
```

Public. This is the same as RenderRegion(), except each tile's source textures are retired as soon as they are done being rendered. This minimizes the memory footprint required, and saves in extra texture allocation time, but if the tiles that compose this portion of the image are going to be used after this call, retiring the tiles will just slow down subsequent tile renders. In such cases, use RenderRegion().

targetArea – Rectangular region of the image (in pixels) that should be rendered to the frame buffer.

invertY – True if the image should be flipped vertically. The best way to figure out what this value should be is to test it.

imageArea – Rectangular region of the image (in pixels) that the frame buffer represents. If this is set to all zeroes, the image area is assumed to be equal to the entire target area.

TileManager::PSTile

Private. Short for “Photoshop tile,” this stores basic information regarding each tile composing the Photoshop image. PSTile is a struct, so every member within it is public.

Data Members

```
int channels[MaxColorChannels];
VRect area;
VPoint size;
```

channels – Array of textures (in the form of texture handles returned by TileManager::CreateTexture) representing each color channel in the tile.
area – The rectangular region the tile occupies, in pixels.
size – The dimensions of the tile.
MaxColorChannels – A global constant declared in TileManager.h. Currently, this is set to 5.

Functions

```
PSTile();
PSTile(int x, int y, int width, int height);
PSTile(PSTile &tile);
```

Constructor. Provides a convenient way to initialize PSTile members.

x – The position of the tile, in pixels, along the X-axis.
y – The position of the tile, in pixels, along the Y-axis.
width – The width of the tile, in pixels
height – The height of the tile, in pixels
tile – A PSTile object to construct a copy of.

```
PSTile &operator =(PSTile &tile);
```

Sets all the the left-hand side PSTile members equal to the right-hand side PSTile members.

tile – The right-hand side of the = operator.

Applicator

Sets up the OpenGL read buffer and draw buffer to allow the user to first render a portion of the source image to the frame buffer, then apply the filter described in GPUFilterData with a single function call, and finally read the results back out of the frae buffer.

Data Members

```
static Applicator *me;
HGLRC mOldRC;
HDC mOldDC;
VRect mArea;
VRect mWorkArea;
VPoint mAreaSize;
VPoint mFilterSize;
VPoint mImageSize;
MyPBuffer mPad;
GLuint mTexObject;
bool mWhereIsDraw;
```

me - Private. The pointer to the single instance of this object. Use `Applicator::Get()` to retrieve this.

mOldRC – Private. The rendering context that was current when the Applicator was created. This is used by Deactivate().

mOldDC – Private. The device context that was current when the Applicator was created. This is used by Deactivate().

mArea – Private. The rectangular portion of the Photoshop image that the Applicator represents.

mWorkArea – Private. After the filter is applied to the applicator, there will be a border region that is inaccurate due to the loss of data around the border. The work area contains only the pixels that contain no error due to the border.

mAreaSize – Private. The dimensions of mArea.

mFilterSize – Private. The cumulative filter size of each pass described in the GPUFilterData object.

mImageSize – Private. The size of the entire Photoshop image.

mPad – Private. The OpenGL PBuffer that is used to apply all the passes.

mTexObject – Private. A single OpenGL texture object that the PBuffer represented by mPad is bound to. This seems redundant, but is a requirement of the PBuffer extension architecture.

Functions

```
static Applicator *Get();
```

Public. Returns the pointer to the single instance of a TileManager object.

```
Applicator(VPoint &size);
```

Constructor. Use Activate() to switch to the applicator's OpenGL context so you can render the source image to it.

size – Size of the applicator, in pixels. This is limited by the maximum texture size supported by the hardware.

```
~Applicator();
```

Destructor.

```
VRect GetArea();
```

Public. Returns the area of the Photoshop image that the applicator currently represents. See `Applicator::mArea`.

```
VRect GetWorkArea();
```

Public. Returns the applicator's work area. See `Applicator::mWorkArea`.

```
VPoint GetWorkAreaSize();
```

Public. Returns the size of the applicator's work area. See `Applicator::mWorkArea`.

```
VPoint GetFilterSize();
```

Public. Returns the cumulative filter size of every pass in the filter. See `Applicator::mFilterSize`.

```
void Activate();
```

Public. Makes the Applicator's OpenGL context current. Use `Applicator::Deactivate()` to switch back to the OpenGL context that was current before the Applicator was activated.

```
void Deactivate();
```

Public. Makes the OpenGL context that was current before the Applicator was activated current again.

```
HPBUFFERARB GetPadHandle();
```

Public. Returns the handle of the OpenGL PBuffer that the Applicator is using.

```
void Target(
  VPoint &target,
  bool top = true,
  bool left = true);
```

Public. This specifies the area of the image that the Applicator will focus on. The function accepts a target point instead of a rectangular area because it will automatically target the largest, most effective possible area.

target – Specifies a desired corner point of the Applicator's work area (see `Applicator::mWorkArea`). The target point will always end up within the work area, but may not be actually end up in the corner. Use `Applicator::GetWorkArea()` to find out precisely where the work area ended up.
top – True if the point specified by target is a top corner, false if it is a bottom corner.
left – True is the point specified by target is a left corner, false if it is a right corner.

```
VRect PeekTargetArea(
  VPoint &target,
  bool top = true,
  bool left = true);
```

Public. Returns what the Applicator's area (see `Applicator::mArea`) would be if the `Applicator::Target()` function were called with the parameters given.

target – See the target parameter in `Applicator::Target()`.
top – See the top parameter in `Applicator::Target()`.
left – See the left parameter in `Applicator::Target()`.

```
VRect PeekTargetWorkArea(
  VPoint &target,
  bool top = true,
  bool left = true);
```

Public. Returns what the Applicator's work area (see `Applicator::mWorkArea`) would be if the `Applicator::Target()` function were called with the parameters given.

target – See the target parameter in `Applicator::Target()`.
top – See the top parameter in `Applicator::Target()`.
left – See the left parameter in `Applicator::Target()`.

```
void FilterTo(
  const VRect &renderArea,
  bool invertY);
```

Public. Executes the filter operation on whatever image has been rendered to the Applicator's frame buffer. The result can be extracted using `glReadPixels()`.

renderArea – The area of the image to apply the filter to. If this area is outside the range of the Applicator, it will be cropped.
invertY – True if the Applicator should invert the Y axis of the resulting image.

```
void ResetWorkArea();
```

Public. Often when filter parameters change, the filter size will change with it. When you detect that that has happened (or may have happened), call `Applicator::ResetWorkArea()`. This tells the Applicator to recalculate the work area based on the new filter size.