Spout SDK

A software development kit for texture sharing between applications.

spout.zeal.co

Reference Manual

Spout Release 2.004 - beta

August 2015

1. Introduction

Spout SDK is a set of C++ classes that allow programmers to compile Spout texture sharing functions into their own application.

An OpenGL context must be established before the initialisation or texture sharing functions can be called.

Using the classes

Add all the class files to your project.

spout.h
spoutSDK.h
spoutSDK.cpp
spoutSender.h
spoutSender.cpp
spoutReceiver.h
spoutReceiver.cpp
spoutSenderNames.h
spoutSenderNames.cpp
SpoutSharedMemory.h
SpoutSharedMemory.cpp
spoutGLDXinterop.h
spoutGLDXinterop.cpp
spoutGLextensions.cpp
spoutGLextensions.h
spoutMemoryshare.cpp
spoutMemoryshare.h
spoutDirectX.h
spoutDirectX.cpp

In you header file declare :

```
#include "spout.h"
```

It is a good idea to put the Spout class files in a separate folder such as "SpoutSDK", perhaps a level or two above your application project.

For example this might be :

```
#include "..\..\SpoutSDK\spout.h"
```

See also the section on compiling the SDK as a dll.

2. Using Spout

To use Spout, create receiver and sender objects and use them to perform the functions available for sender or receiver.

This can be done in two ways. For example :

```
        SpoutSender mySender
```
or
```
        SpoutSender * mySender
        mySender = new SpoutSender;
        . . .
        delete mySender;
        mySender = NULL;
```

The first method is simple and the objects are deleted and all resources freed when the program terminates.

The second method is useful where you want to create and delete Spout objects while your program is running. In this case you have to specifically delete the objects before the program terminates.

2.1 Creating senders and receivers

NOTE : For any of the Spout functions that include a sender name, it is important that a character array of at least 256 characters is passed. If the character array passed is dimensioned less than this, there could be problems. Also NULL cannot be passed, a valid character array is required.

For a sender :

1) Create a SpoutSender object

```
        SpoutSender mySender;
```

2) Create a sender

```
        mySender.CreateSender(SenderName, Width, Height);
```

3) Send a texture

```
        mySender.SendTexture(myTextureID, myTextureTarget,
                                     Width, Height);
```

For a receiver :

1) Create a SpoutReceiver object

```
SpoutReceiver myReceiver;
```

2) Create a receiver

```
myReceiver.CreateReceiver(SenderName, Width, Height);
```

3) Receive a texture

```
myReceiver.ReceiveTexture(SenderName, Width, Height);
```

4) Draw the texture

```
myReceiver.DrawSharedTexture();
```

Other options are available to bind and unbind the shared texture as may be required or to receive a texture into a local texture.

To bind and unbind the shared texture

```
myReceiver.BindSharedTexture();
myReceiver.UnBindSharedTexture();
```

To receive a texture into a local texture

```
myReceiver.ReceiveTexture(SenderName,
                          width, height,
                          myTextureID,
                          myTextureTarget);
```

2.2 Releasing senders and receivers

You may wish to create Spout senders or receivers, release them and create them again within your program. This may be necessary if, for example, the OpenGL context is lost or changes when initializing something or creating a new window and the like.

The Spout objects do not need to be closed, but the sender or receiver should be released and re-created.

```
mySender.ReleaseSender();
```

or

```
myReceiver.ReleaseReceiver();
```

Once released, the sender or receiver have to be created again before any of the functions will work. The objects themselves remain and can be re-used to re-create a sender or receiver.

2.3 Creating a sender

You must create a sender by calling *CreateSender* before attempting to send a texture. This initializes a DirectX device, creates a shared DirectX texture and links an OpenGL texture to it.

```
mySender.CreateSender(SenderName, width, height);
```

<u>For success</u>

A sender with the name provided has been initialized and registered as a Spout sender for all receivers to detect. Now you will need to create a local OpenGL texture that can be used for sending.

<u>For failure</u>

An error has occurred and no further operations can be made.

NOTE: you cannot create more than one sender for the one sender object. If you require more than one sender, you have to create more sender objects.

2.4 Sending a texture

Once a sender has been created, you can send textures with it.

<u>To send a texture</u>

First, fill a local texture with the data you want to send. For example :

```
// Grab the screen into a local texture
glBindTexture(GL_TEXTURE_2D, myTexture);
glCopyTexSubImage2D(GL_TEXTURE_2D, 0, 0, 0, 0, 0, width, height);
glBindTexture(GL_TEXTURE_2D, 0);
```

Then send it out. This creates a shared texture for all receivers to access.

```
mySender.SendTexture(myTexture, GL_TEXTURE_2D, width, height);
```

NOTE : If you call SendTexture with a framebuffer object bound, that binding must be included as an argument so that it is restored, because Spout makes use of its own FBO for intermediate rendering.

For example :

```
mySender.SendTexture(myTexture, GL_TEXTURE_2D,
                     width, height, bInvert, fboId);
```

If the dimensions of the local texture that you are sending out change, the sender information that receivers access is updated within the SendTexture function.

NOTE: the local texture itself is not updated by this function and that must be done in your application.

However, if you wish to update your sender without having to send a texture out you can do so with the following function.

```
mySender.UpdateSender(SenderName, width, height);
```

To draw to a shared texture

Commonly a texture is rendered to another one by way of an fbo. An application texture can also be rendered to the shared texture in the same way with a single function.

```
mySender.DrawToSharedTexture(myTexture, GL_TEXTURE_2D,
                             width, height,
                             max_X, max_Y, aspect,
                             bInvert, fboId);
```

See below for further details.

2.5 Releasing a sender

A sender can be released so that all resources are freed. This releases the GL/DX interop, DirectX and the linked DirectX and OpenGL textures. Then a new sender can be created when required.

```
mySender.ReleaseSender();
```

When a sender is released, the sender's name is removed from the list of Spout senders and no receivers will detect it until it is created again again. The SpoutSender object cannot be used again to send a texture until you create a new sender.

It is good practice to release the sender at program termination, although all resources are freed even if you do not.

2.6 Creating a receiver

You should create a receiver by calling *CreateReceiver* before attempting to receive a texture. This initializes a DirectX device, detects a sender, accesses the shared DirectX texture and links an OpenGL texture to it.

```
myReceiver.CreateReceiver(SenderName, width, height, true);
```

The sender name you provide can be the sender that the receiver should connect to. That sender has to be running for it to be used. This is useful if you know the name of the sender you wish to connect to and only want your receiver to start when that sender is running.

However, if you provide the optional argument (`bUseActive`) "true", Spout will attempt to find the "active sender" and will connect if it is running.

The "active sender" is the one that was started first or the one that the user has selected last by using "SpoutTray", or by using "SpoutPanel" activated by another Spout receiver.

This is useful for an automatic start, where your receiver will detect and connect to any active sender that is running at the time.

<u>For success</u>

Check the sender name, width and height returned and adjust anything required such as a local texture or the window size etc..

<u>For failure</u>

A sender was not found, so just keep calling *CreateReceiver* until a sender is found. The overhead if no senders are running is very low.

NOTE: you cannot create more than one receiver for the one object. If you require more than one receiver, you have to create more receiver objects.

2.7 Receiving a texture

Once a receiver has been successfully created, you can receive textures with it.

NOTE: It is also possible to use ReceiveTexture directly without first creating a Receiver. The same sender name, width and height are returned when a sender is found. However, a specific call to CreateReceiver is recommended.

To receive a shared texture from a sender

```
myReceiver.ReceiveTexture(SenderName, Width, Height);
```

NOTE : If you call ReceiveTexture with a framebuffer object bound, that binding must be included as an argument so that it is restored, because Spout makes use of its own FBO for intermediate rendering.

For example :

```
myReceiver.ReceiveTexture(SenderName, Width, Height,
                          0, GL_TEXTURE_2D, fboID);
```

The received shared texture can also be used independently, for example for graphics operations :

```
glEnable(GL_TEXTURE_2D);
myReceiver.BindSharedTexture();
glBegin(GL_QUADS);
glTexCoord2f(0.0, 1.0);   glVertex2f(-1.0,-1.0);
glTexCoord2f(0.0, 0.0);   glVertex2f(-1.0, 1.0);
glTexCoord2f(1.0, 0.0);   glVertex2f( 1.0, 1.0);
glTexCoord2f(1.0, 1.0);   glVertex2f( 1.0,-1.0);
glEnd();
myReceiver.UnBindSharedTexture();
glDisable(GL_TEXTURE_2D);
```

or a direct draw :

```
myReceiver.DrawSharedTexture();
```

To receive an application texture from a sender

```
myReceiver.ReceiveTexture(SenderName,
                          width, height,
                          myTextureID,
                          myTextureTarget);
```

If you are using a local texture and it was received successfully, the passed width or height could have changed if the sender changed dimensions, so this always has to be tested for maintaining the correct window dimensions or to adjust the receiving texture.

If you have received into a local texture, that can be used as required.

2.8 Releasing a receiver

A receiver can be released so that all resources are freed. This releases the GL/DX interop, DirectX and the linked DirectX and OpenGL textures. Then a new receiver can be created when necessary.

```
myReceiver.ReleaseReceiver();
```

After a receiver is released, you must create a new receiver before receiving textures once more.

It is good practice to release the receiver at program termination, although, for the current version, all resources are freed even if you do not. This will ensure compatibility with future revisions.

2.9 User sender selection

SelectSenderPanel

```
receiver.SelectSenderPanel();
```

This receiver function activates an executable program *"SpoutPanel.exe"* that displays a dialog with a list of names for the currently running senders and allows the user to choose one.

NOTE: SpoutPanel.exe is detected if Spout later than 2.002 has been installed, otherwise it has to be in the same folder as the host executable.

Once the user has selected a new sender, *ReceiveTexture* will detect the change and return the new sender name, width and height.

The width and height then have to be tested within your program so that the render window or local texture size can be adjusted.

The sender name is not important unless you wish to display which sender is currently being received.

Texture formats

Details of the sender size and DirectX texture format are shown in the dialog.

The number of texture formats for sharing between DirectX11 and DirectX 9 are limited. For SDK functions that use DirectX 11 textures, Spout uses a default format that is compatible with DirectX 9 `DXGI_FORMAT_B8G8R8A8_UNORM` (87).

One alternative DirectX 11 format is `DXGI_FORMAT_R8B8G8A8_UNORM` (28) which is not compatible with DirectX 9 but may be necessary for your application. This can be set by `SetDX9compatible(false)` (see below).

In this case, only Spout applications using DirectX 11 functions can receive the texture, so for receivers which are compiled to use DirectX 9 functions, "SelectSenderPanel" allows an argument which causes *SpoutPanel.exe* to test the sender format and prevent access if it is not compatible with DirectX 9.

```
        receiver.SelectSenderPanel("/DX9");
```

With the "/DX9" argument, a warning is given if the texture format is not compatible with DirectX 9.

The default "/DX11" can also be specified to reverse the option.

User messages

There are situations, such as a dll or plugin, where a Windows MessageBox cannot be used. A MessageBox can interrupt the Windows messaging for the program and in some cases this can cause the host application to freeze or crash.

SelectSenderPanel can be used in these situations to provide the equivalent to a Windows MessageBox.

```
        receiver.SelectSenderPanel("user message");
```

SpoutPanel.exe is an independent process and so does not affect the application's messages.

2.10 Creating your own sender list

If you want to create a list of Spout senders in a menu or drop-down list, you can use the following receiver functions.

Getting the sender count

```
        int nSenders;
        nSenders = myReceiver.GetSenderCount();
```

GetSenderCount returns the number of Spout senders that are currently running.

Getting a sender name

GetSenderName returns the name of a sender with an index within the range indicated by *GetSenderCount*.

The maximum length of a Spout sender name is 256 characters, but you can specify a character array for the name with a length less than this for *GetSenderName*. The default is 256 if length is not specified.

```
char SenderName[64];
int MaxSize = 64;
int index = 0; // or based on GetSenderCount
myReceiver.GetSenderName(index, SenderName, MaxSixe);
```

Getting sender details

GetSenderInfo returns the width, height of the sender as well as the share handle and format of the DirectX 11 shared texture. For OpenGL you will only need to know the width and height.

```
unsigned int width, height;
HANDLE hShareHandle;
DWORD dwFormat;

myReceiver.GetSenderInfo(SenderName,
                         width, height,
                         hShareHandle, dwFormat);
```

If, for example, you want to construct a menu or list :

```
int index, nSenders;
char SenderName[64];
int MaxSize = 64;
 unsigned int width, height;
HANDLE hShareHandle;
DWORD dwFormat;

nSenders = myReceiver.GetSenderCount();
if(nSenders > 0) {
    for(index=0; index<nSenders; index++) {
        myReceiver.GetSenderName(index, SenderName, MaxSize);
        myReceiver.GetSenderInfo(SenderName,
                                 width, height,
                                 hShareHandle, dwFormat);
        //
        // Do something to create your list
        //

    }
}
```

Setting the active sender

When the user has selected a sender from your list, you may wish to set this sender as "active" in keeping with the function of *"SelectSenderPanel"*.

```
myReceiver.SetActiveSender(SenderName);
```

Once set, this sender will be the one that a receiver detects when it starts if the nominated sender name is not found.

Finding the active sender

When you make your selection list, you may wish to indicate to the user which sender is currently "active".

```
char SenderName[256];
myReceiver.GetActiveSender(SenderName);
```

This will return the name of the active sender.

NOTE: Make sure you have prepared a receiving character array for the sender name of no less than 256 bytes in length.

2.11 Sender size and compatibility

```
myReceiver.GetImageSize(Sendername, width, height, bMemoryMode);
```

It is sometimes useful to know whether a sender that is running has initialised in Memory Share or Texture Share mode and what the size of that sender is.

The significance of this function is that, unlike "*GetMemoryShareMode*" it does not require a sender or receiver to be created within your application and does not require an OpenGL context, so it can be used at the outset of a program to detect whether a memoryshare sender is running and determine what may need to follow.

If you find that a sender is running in MemoryShare mode, you can elect to set MemoryShare mode for our application using :

```
myReceiver.SetMemoryShareMode(true);
```

Thereafter, all calls to ReceiveTexture or SendTexture will operate in Memory Share mode. You can also use ReceiveImage and SendImage to transfer data from image pixels directly by way of the CPU.

2.12 Utility functions

Utility functions are available for both sender and receiver objects.

Texture sharing compatibility

The Spout initialisation functions *CreateSender* and *CreateReceiver* will detect whether the hardware is compatible with DirectX texture sharing and if the hardware is not compatible, Spout will default to sharing textures by way of shared memory.

NOTE: for shared memory, functions for sending and receiving are exactly the same, but there can only be one sender and one receiver and both must operate in memory share mode. Multiple senders cannot be detected or used.

After creating a sender or receiver, you may wish to determine this capability so that you can act accordingly.

```
bool bMemoryMode = myReceiver.GetMemoryShareMode();
```

This returns true if memory sharing will be used because the hardware is not texture share compatible.

Setting memory sharing mode

If you want your program to use memory sharing specifically, you can set this at the beginning.

```
mySender.SetMemoryShareMode(true); // default is false
```

NOTE: Memory Share mode has been necessary due to past problems with NVIDIA drivers which are now resolved and may not be available in future releases of the Spout SDK.

Setting DirectX 9 mode

Spout will be initialized to use either DirectX 9 or DirectX 11 textures depending on the option selectd after installation using the utility program “SpoutDirectX.exe”.

For most situations this will have no influence on operation or performance. Both DirectX 11 and DirectX 9 senders and receivers built with the Spout SDK will always communicate because the DirectX 11 texture format used (DXGI_FORMAT_B8G8R8A8_UNORM) is compatible with DirectX 9 texture sharing.

However, if you wish to use DirectX 9 or DirectX 11 specifically for your application you can set this option at the beginning of the program, typically just after creating the sender or receiver object.

```
    mySender.SetDX9(true); // default is false (DirectX 11)
```

Selecting a DirectX 11 sender texture format

If you are using DirectX 11, the texture formats for sharing between DirectX 11 and DirectX 9 are limited. DirectX 11 texture format is set by default to (`DXGI_FORMAT_B8G8R8A8_UNORM`) which is compatible with DirectX 9 receivers.

The alternative formats are :

```
DXGI_FORMAT_R8B8G8A8_UNORM
DXGI_FORMAT_R32G32B32A32_FLOAT
DXGI_FORMAT_R16G16B16A16_FLOAT
DXGI_FORMAT_R16G16B16A16_SNORM
DXGI_FORMAT_R10G10B10A2_UNORM
DXGI_FORMAT_R8G8B8A8_UNORM
DXGI_FORMAT_B8G8R8A8_UNORM
DXGI_FORMAT_B8G8R8AX_UNORM
```

These texture formats are is not compatible with DirectX 9 but may be necessary for your application.

There is a function that ensures that only senders with DirectX 9 compatible formats are listed and you can disable this :

```
    mySender.SetDX9compatible(false); // default is true
```

As with the previous “SetDX9” function, you must set this at the beginning of the program.

Setting the graphics adapter

If there is more that one graphics adapter installed, the one used for Spout output can be selected.

Functions available are :

```
// Get the number of graphics adapters in the system
int GetNumAdapters();
```

```
// Get an adapter name
bool GetAdapterName(int index, char *adaptername, int maxchars);

// Set required graphics adapter for output
bool SetAdapter(int index);

// Get the current adapter index
int GetAdapter();
```

These functions are contained in the SpoutDirectX class but are available from the Sender and Receiver classes.

Thanks and credit to Franz Hildgen for this function.

Vertical sync

Spout uses synchronisation methods to ensure that access to a shared texture by sender and receivers is controlled.

This process can be independent of monitor vertical refresh depending on the driver settings.

Spout provides functions to query and set the state of Vertical Sync using the OpenGL *SwapInterval* extensions.

```
        int SyncStatus = mySender.GetVerticalSync();
```

Retrieves the sync state state using *wglGetSwapIntervalEXT();*

```
        mySender.SetVerticalSync(true);
```

Calls *wglSetSwapIntervalEXT(1) to s*ets the OpenGL swap interval to one*,* so that the draw cycle is locked to to monitor vertical sync*.*

```
        mySender.SetVerticalSync(false);
```

Calls *wglSetSwapIntervalEXT(0)* to release the OpenGL swap interval from monitor vertical sync*.*

NOTE: Vertical sync is affected by many factors. Also applications may manage this already. These functions may be removed in future revisions.

2.13 Compiling the SDK as a dll

The Spout SDK files can either be included directly in your project or compiled as a dll.

The Spout applications and examples have been compiled with the SDK files included in the projects and so are not dependent on a dll.

This may be the preferred method, so that no dependent dll is distributed with the application.

However, using a dll is useful if you wish your application to be independent of minor updates to the SDK. Then the SDK can be re-compiled and the dll used without disturbing the application code or executable itself.

Visual Studio 2010 and 2012 projects have been included in the SDK to produce the dll.

To use the dll created from this project, in your code define SPOUT_IMPORT_DLL in your preprocessor compilation defines.

```
#define SPOUT_IMPORT_DLL
#include "Spout.h"
```

Then the application code can be exactly the same as if the SDK files were included in the project.

Note that the resulting dll and library files can only be used within a Visual Studio environment.

Thanks and credit to Malcolm Bechard for creating the dll project.

2.14 C compatible dll

The C++ dll project referred to above has the advantage that all the functions in the Spout SDK classes are available, and the application code is exactly the same as if the SDK files were included in the project.

However, it is only suitable for use with Visual Studio compilers. Other compilers will fail due to “set” functions which are not supported.

Therefore, a separate dll project is included with the Spout distribution that is compatible with compilers other than Microsoft Visual Studio.

Find it in the installation SPOUTSDK\SpoutSDK\SPOUTDLL folder.

The "VS2010" and "VS2012" folders contain Visual Studio projects to build the DLL. Visual Studio is required to build the dll but thereafter it can be used by other compilers. Refer to the readme file for details on compilation.

The resulting DLL is simple in that the functions are called directly after the “Spout2” namespace is defined. But there can only be one sender or receiver object in any one application.

An Openframeworks example is included to test the dll using CodeBlocks and the MingW compiler. Refer to the readme for further details on building the example.

In short, the files Spout2.lib, Spout2.dll and SpoutDLL.h are all you should need.

3. Spout functions

NOTE : For any of the Spout functions that include a sender name, it is important that a character array of at least 256 characters is passed. If the character array passed is dimensioned less than this, there could be problems. Also NULL cannot be passed, a valid character array is required.

3.1 Sender functions

3.1.1 CreateSender

```
bool CreateSender  (char *Sendername,
                     unsigned int width,
                     unsigned int height,
                     DWORD dwFormat = 0);
```

Creates a Spout sender. This initializes a DirectX device, creates a shared DirectX texture and links an OpenGL texture to it.

For success

A sender with the name provided is initialized and registered as a Spout sender for all receivers to detect. Now you will need to create a local OpenGL texture that can be used for sending.

For failure

An error has occurred and no further operations can be made for this object.

NOTE: you cannot create more than one sender for the one sender object. If you require more than one sender, you have to create more sender objects.

3.1.2 SendTexture

```
bool SendTexture (GLuint TextureID,
                  GLuint TextureTarget,
                  unsigned int width,
                  unsigned int height,
                  bool bInvert=true,
                 GLuint HostFBO = 0);
```

Creates a shared texture for all receivers to access.

The invert flag is optional and by default true. This flips the texture in the Y axis which is necessary because DirectX and OpenGL textures are opposite in Y. If it is set to false no flip occurs and the result may appear upside down.

The host fbo argument is optional, default 0. If an fbo is currently bound and it's ID passed, then that binding is restored within the SendTexture function.

3.1.3 SendImage

```
bool SendImage(unsigned char* pixels,
               unsigned int width,
               unsigned int height,
               GLenum glFormat = GL_RGBA,
               bool bAlignment = true,
               bool bInvert=true);
```

Creates a shared texture using image pixels as the source instead of an OpenGL texture.

The format of the image to be sent is RGBA by default but can be a different OpenGL format, for example GL_RGB or GL_BGRA_EXT.

The *bAlignment* flag can be set to false, which disables the default 4-byte alignment assumed by OpenGL. This may be necessary if, for example the pixels you wish to send are RGB and packed with single alignment.

As for SendTexture, the invert flag is optional and by default true.

3.1.4 DrawToSharedTexture

```
bool DrawToSharedTexture(GLuint TextureID, GLuint TextureTarget,
                         unsigned int width, unsigned int height,
                         float max_x = 1.0, float max_y = 1.0,
                         float aspect = 1.0, bool bInvert = true,
                         GLuint HostFBO = 0);
```

Renders an application texture to the shared texture via an FBO.

- width, height are the dimensions of the texture to be drawn.
- max_x, max_y are maximum extents of the texture that will be drawn.
- aspect is the fraction to be used within the draw vertex coordinates.

For example the default vertex coordinates are -1.0 to +1.0 in X and Y and if aspect is passed as a fraction of this (e.g. 0.75), the result is drawn inside this coordinate system.

The FBO used for texture transfer is internal to the function. If an FBO is currently bound when calling this program, the ID should be passed so that the binding can be restored within the function.

3.1.5 UpdateSender

```
bool UpdateSender (char *Sendername,
                   unsigned int width,
                   unsigned int height);
```

If the dimensions of the local texture that you are sending out change, the sender information that receivers access can be updated by this function. This update is also done by *"SendTexture"*, but *"UpdateSender"* does not require a texture to be sent out.

3.1.6 ReleaseSender

```
void ReleaseSender(DWORD dwMsec = 0);
```

Releases a sender created by CreateSender. All OpenGL and DirectX resources for the sender are freed. The dwMsec argument is a debugging aid to introduce a Sleep delay and may be removed in future releases.

3.1.7 SetMemoryShareMode

```
bool SetMemoryShareMode(bool bMemoryMode = true);
```

Sets Spout to use shared memory image transfer instead of the OpenGL/DirectX interop. There can be only one sender/receiver pair which are un-named and the sender selection is no applicable.

3.1.8 GetMemoryShareMode

```
bool GetMemoryShareMode();
```

Returns the current MemoryShare mode. This may have been set by CreateSender if the hardware is not texture share compatible.

NOTE: MemoryShare mode may be removed in future releases.

3.1.9 SetDX9

```
void SetDX9(bool bDX9 = true);
```

Sets whether the program will operate with DirectX 9 textures.

3.1.10 GetDX9

```
bool GetDX9();
```

Returns whether the the program is operating with DirectX9 textures.

3.1.11 SetDX9compatible

```
void SetDX9compatible(bool bCompatible = true);
```

Sets whether the format of the shared DirectX11 texture that is linked by way of the OpenGL/DirectX interop is compatible with DirectX 9 receivers.

3.1.12 GetDX9compatible

```
bool GetDX9compatible();
```

Returns whether the the format of the shared DirectX11 texture is set to be compatible with DirectX 9 receivers.

3.1.13 SetVerticalSync

```
bool SetVerticalSync(bool bSync = true);
```

Calls *wglSetSwapIntervalEXT() to s*et the OpenGL colour buffer swap periodicity to one video frame period (true)*,* so that the draw cycle is locked to to monitor vertical sync, or zero (false) so that buffer swaps are not synchronized to a video frame.

3.1.14 GetVerticalSync

```
int GetVerticalSync();
```

Retrieves the sync state state using *wglGetSwapIntervalEXT();*

NOTE: Vertical sync is affected by many factors. Also applications may manage this already. These functions may be removed in future revisions.

3.2 Receiver functions

3.2.1 CreateReceiver

```
bool CreateReceiver(char* Sendername,
                        unsigned int &width,
                        unsigned int &height,
                        bool bUseActive = false);
```

Creates a receiver which must be done before attempting to receive a texture. This initializes a DirectX device, detects a sender, accesses the shared DirectX texture and links an OpenGL texture to it.

The sender name you provide can be the sender that the receiver should connect to. That sender has to be running for it to be used. This is useful if you know the name of the sender you wish to connect to and only want your receiver to start when that sender is running.

If the optional "bUseActive" flag is passed as true, Spout will attempt to find the "active sender" and will connect to that if it is running.

The "active sender" is the one that was started first or the one that the user has selected last by using "SpoutTray", or by using "SpoutPanel" activated from another Spout receiver.

This is useful for an automatic start, where your receiver will detect and connect to any active sender that is running at the time.

Note that the active sender will also be detected if the sender name string is empty, i.e. the first character is 0. *Do not pass NULL as the name*.

For success

Check the sender name, width and height returned and adjust anything required such as a local texture or the window size etc..

For failure

A sender was not found, so just keep calling *CreateReceiver* until a sender is found. The overhead if no senders are running is very low.

NOTE: you cannot create more than one receiver for the one object. If you require more than one receiver, you have to create more receiver objects.

3.2.2 ReceiveTexture

```
bool ReceiveTexture(char* Sendername,
                    unsigned int &width,
                    unsigned int &height,
                    GLuint TextureID = 0,
                    GLuint TextureTarget = 0,
                    GLuint HostFBO = 0);
```

Once a receiver has been successfully created, you can receive shared textures from senders. The shared texture is internal to Spout.

The shared texture can be received into a local texture if the ID and Target are passed, or the received shared texture can be used directly for graphics operations.

Any changes to sender size are managed within Spout, however if you are receiving to a local texture, the changed width or height are passed back and must be tested to adjust the receiving texture or for for maintaining window dimensions.

3.2.3 ReceiveImage

```
bool ReceiveImage  (char* Sendername,
                    unsigned int &width,
                    unsigned int &height,
                    unsigned char * pixels,
                    GLenum glFormat = GL_RGBA,
                    GLuint HostFBO = 0);
```

Receives a sender shared texture into image pixels. The same sender size changes are passed back as for ReceiveTexture.

3.2.4 GetImageSize

```
bool GetImageSize  (char* Sendername,
                    unsigned int &width,
                    unsigned int &height,
                    bool &bMemoryMode);
```

This function returns the width, height and sharing mode of a sender. Unlike "*GetMemoryShareMode*" the function can be called before a sender or receiver is initialised and without an OpenGL context

This is useful at the start of a program for detection of a memory share sender and to determine whether to receive from a memoryshare sender.

In that case "*SetMemoryShareMode*" is used as detailed below and subsequent Spout functions will operate in memoryshare mode.

3.2.5 ReleaseReceiver

```
void ReleaseReceiver();
```

Releases a receiver created by "CreateReceiver". Al OpenGL and DirectX resources are freed for the Receiver.

3.2.6 BindSharedTexture

```
bool BindSharedTexture();
```

This function enables the internal Spout shared texture to be bound for OpenGL operations. It is used in exactly the same way as binding an OpenGL texture. This is useful where a local application texture is not used.

NOTE: since this call locks the shared texture, it very important that UnBindSharedTexture is called immediately following the OpenGL operations so that the texture is unlocked for other functions.

3.2.7 UnBindSharedTexture

```
bool UnBindSharedTexture();
```

Unbinds the the internal Spout shared texture.

3.2.8 DrawSharedTexture

```
bool DrawSharedTexture(float max_x = 1.0,
                       float max_y = 1.0,
                       float aspect = 1.0,
                       bool bInvert = true);
```

Performs a draw using the Spout internal shared texture. This may not be suitable in some environments such as Cinder or openframeWorks but can work in others such as FreeFrameGL.

Optionally "aspect" allows the ratio of width to height to be adjusted, typically this value would be width/height.

3.2.9 GetSenderCount

```
int  GetSenderCount();
```

Returns the number of named Spout texture senders running.

3.2.10 GetSenderInfo

```
bool GetSenderInfo (char* Sendername,
                    unsigned int &width,
                    unsigned int &height,
                    HANDLE &dxShareHandle,
                    DWORD &dwFormat);
```

Returns information of a running sender with a given name which may have been retrieved with "GetSenderName".

The format returned is either a DirectX 11 format or zero for a DirectX 9 sender.

3.2.11 GetActiveSender

```
bool GetActiveSender(char* Sendername);
```

Returns the name of the active sender.

3.2.12 SetActiveSender

```
bool SetActiveSender(char* Sendername);
```

Sets the name of the active sender.

3.2.13 SelectSenderPanel

```
bool SelectSenderPanel(char* message = NULL);
```

Activates an executable program "SpoutPanel.exe" which displays a list of Spout senders and allows the user to select one.

NOTE: SpoutPanel.exe must be in the same folder as the host program.

The details of the sender size and DirectX texture format are shown in the dialog.

If a "*/DX9*" argument is passed, the dialog shows a warning if the DirectX format is incompatible and the selection of a DirectX texture is disabled. This is useful if your application is using DirectX 9 functions and you wish to ensure that compatible senders are selected.

The default argument "*/DX11*" can be used to reverse the setting.

An optional text argument can be passed to this function, and then the dialog has a function equivalent to a Windows MessageBox. This is useful when the host application may freeze or is otherwise interrupted by a typical modal MessageBox.

3.2.14 Utility

```
bool GetMemoryShareMode();
bool SetMemoryShareMode(bool bMemory = true);
void SetDX9(bool bDX9 = true);
bool GetDX9();
void SetDX9compatible(bool bCompatible = true);
bool GetDX9compatible();
int  GetNumAdapters();
bool GetAdapterName(int index, char *adaptername, int maxchars);
bool SetAdapter(int index);
int  GetAdapter();
bool SetVerticalSync(bool bSync = true);
int  GetVerticalSync();
```

These functions are identical to the equivalents for a sender as detailed above.