Spout SDK

A software development kit for texture sharing between applications.

spout.zeal.co

Version 2.007

January 20th 2020

1. Changes for existing projects

With attention to the items below, existing projects will rebuild and function as before with only minor changes.

Check the “Frame count” option on with *SpoutSettings* to enable the frame counting functions.

1.1 Legacy OpenGL

Functions with legacy OpenGL code can be removed so that the SDK is compliant with more recent versions of OpenGL by disabling a define in “SpoutCommon.h”.

```
#define legacyOpenGL
```

The following functions are then disabled :

```
  DrawSharedTexture
  DrawToSharedTexture
```

1.2 New files

To update existing projects, add the following four files :

spoutFrameCount.h
spoutFrameCount.cpp
spoutUtils.h
spoutUtils.cpp

1.3 using std namespace removed

The statement :

```
using namespace std;
```

has been removed from the Spout SDK to give programmers the option for including this or not. If you receive warnings about missing std functions with existing projects, you can include this statement in your header files.

1.4 Changed functions

```
bool GetSenderName(int index, char* Sendername, int MaxSize = 256);
```

Changed to :

```
bool GetSender(int index, char* Sendername, int MaxSize = 256);
```

GetSenderName now returns the connected sender name.

```
const char * GetSenderName();
```

The GetImageSize function has been removed.

RemovePadding has been added for correction of image stride where necessary.

```
void RemovePadding(const unsigned char *source, unsigned char *dest,
        unsigned int width, unsigned int height, unsigned int stride,
        GLenum glFormat = GL_RGBA);
```

Registry functions have been moved to the SpoutUtils namespace and arguments revised as detailed below.

1.5 CPU share mode removed

CPU sharing mode was introduced in 2.006 so that shared DirectX textures were still used but data was transferred to OpenGL via CPU.

In practice, performance was limited and could not be improved so this sharing mode is discontinued for 2.007.

Existing senders and receivers using CPU share mode will still communicate with 2.007 applications as long as the updated applications are using texture share mode.

1.6 Memory share mode improved

Performance of shared memory functions has been improved by using multiple pixel buffers to load from and upload to OpenGL textures and remains a backup where hardware is not texture share compatible.

2. New for 2.007

2.1 Spout Utilities

“SpoutUtils” is a namespace containing utility functions that are accessible by any application that includes the Spout SDK.

2.1.1 Logging

Logging is introduced throughout the SDK to provide advisory, warning and error notices. Logs can be displayed in a console window or written to a file. You can also include your own logs.

Functions are broadly based on Openframeworks logs and are used in a similar manner.

Commented code examples are included in the Spout SDK openframeworks "ofSpoutExample" graphics sender application.

Logging functions

Often it is useful to open a console window during application development to output text messages. The console will only show what you print to it.

```
void OpenSpoutConsole();
```

You can close the console with an optional warning.

```
void CloseSpoutConsole();
```

You can also enable logging to see the logs generated within the Spout SDK.

```
void EnableSpoutLog();
```

Output is to a console window by default but you can instead, or additionally, specify output to a text file with the extension of your choice. The log file is re-created every time the application starts unless you specify to append to the existing one.

```
void EnableSpoutLog(const std::string filename, bool append = false);
```

The file is saved in the %AppData% folder (..\AppData\Roaming\Spout) unless you specify the full path. After the application has run you can find and examine the log file. This folder can also be opened in Windows Explorer directly from the application.

```
 void ShowSpoutLogs();
```

Or the entire log can be returned as a string.

```
 std::string GetSpoutLog();
```

You can set the level above which the logs are shown

```
 SPOUT_LOG_SILENT  - Disable all messages
 SPOUT_LOG_VERBOSE - Show all messages
 SPOUT_LOG_NOTICE  - Notices
 SPOUT_LOG_WARNING - Warnings
 SPOUT_LOG_ERROR   - Errors
 SPOUT_LOG_FATAL   - Fatal errors
 SPOUT_LOG_NONE    - Ignore log levels
```

For example, to show only warnings and errors :

```
  SetSpoutLogLevel(SPOUT_LOG_WARNING);
```

or leave set to default Notice to see more information.

You can also create your own logs. For example :

```
 SpoutLog("SpoutLog test"); // (equivalent to using SPOUT_LOG_NONE)
```

Or specify the logging level. For example :

```
 SpoutLogNotice("Important notice");
```

or

```
 SpoutLogFatal("This should not happen");
```

or

```
 SetSpoutLogLevel(SPOUT_LOG_VERBOSE); // (default is SPOUT_LOG_NOTICE)
 SpoutLogVerbose("Message");
```

Logs can include strings, integers, float numbers etc. based on the "printf" notation. For example :

```
 SpoutLogNotice("myClass::myFunction - sender %s, %dx%d",
                sendername, width, height);
```

Finally you can disable logging at any time within your application.

```
 DisableSpoutLog();
```

2.1.2 SpoutMessageBox

There are situations, such as within a plugin, where a Windows MessageBox cannot be used because it would interfere with the application GUI and cause a freeze.

SpoutMessageBox can be used to avoid this problem in situations where you still need to draw the user's attention to a message. A timeout can also be specified so that the user does not need to dismiss the message.

You can use a simple version :

```
int SpoutMessageBox(const char * message, DWORD dwMilliseconds = 0);
```

or replace an existing MessageBox call with an alternative with standard arguments.

```
int SpoutMessageBox(HWND hwnd, LPCSTR message, LPCSTR caption,
                    UINT uType, DWORD dwMilliseconds = 0);
```

SpoutMessageBox uses "*SpoutPanel.exe*", usually used to display a sender list, so Spout must have been installed. If not, a standard, untimed, topmost messagebox is displayed.

2.1.3 Registry utilities

Utility functions to read and write from the registry are now consolidated into SpoutUtils.

An "HKEY" argument has been added to make them more general purpose and the order of arguments has been revised.

```
bool ReadDwordFromRegistry(HKEY hKey, const char *subkey,
        const char *valuename, DWORD *pValue);
bool WriteDwordToRegistry(HKEY hKey, const char *subkey,
        const char *valuename, DWORD dwValue);
bool ReadPathFromRegistry(HKEY hKey, const char *subkey,
        const char *valuename, char *filepath);
bool WritePathToRegistry(HKEY hKey, const char *subkey,
        const char *valuename, const char *filepath);
bool RemovePathFromRegistry(HKEY hKey, const char *subkey,
        const char *valuename);
bool RemoveSubKey(HKEY hKey, const char *subkey);
bool FindSubKey(HKEY hKey, const char *subkey);
```

2.2 Frame counting

Version 2.007 introduces frame counting which enables a receiver to detect the frame number and frame rate of a sender.

A named counting semaphore is used so that both sender and receivers connecting with it can access the count. Receivers compare the current sender count with the last. If it is the same, the receiver's shared texture is not updated to save processing time.

An application can avoid further re-processing of the received texture by using *IsFrameNew()* after receiving the frame. The receiver can also retrieve the sender frame count and received frame rate.

Frame counting can be enabled or disabled using "*SpoutSettings*" or by the application if necessary.

When disabled, sender/receiver communication is exactly the same as Spout 2.006 and earlier. If enabled, earlier applications are not affected.

If enabled, a 2.007 application can determine whether a sender produces a frame count with "GetSenderFrame()". Applications before 2.007 will return 0.

Independently of frame counting, a utility function is included to control the frame rate of a sender application if no other means is available. It is called for each render cycle to introduce the required delay.

```
void EnableFrameCount(const char* SenderName);// Application enable
void DisableFrameCount();// Application disable
bool IsFrameCountEnabled();// Application check status
bool IsFrameNew(); // Is the received frame new
double GetSenderFps(); // Received frame rate
long GetSenderFrame(); // Received frame count
void HoldFps(int fps); // Sender frame rate control
```

Commented code examples are included in the Spout SDK Openframeworks "*ofSpoutExample*" sender and receiver applications.

2.3 Sender

A new sender function "*SendFbo*" allows for sending a texture attached to the currently bound fbo. This is particularly useful if the sending texture is larger than the size that the sender is set up for.

```
bool SendFbo(GLuint FboID, unsigned int width, unsigned int height, bool
bInvert = true);
```

This can be the case if the application is using only a portion of the allocated texture space, such as for Freeframe plugins. This was previously achieved by rendering to the shared texture using "*DrawToSharedTexture*".

SendTexture and SendImage remain unchanged.

5) Query the sender

At any time you can query the sender to retrieve the size, frame number and frame rate.

GetWidth() and GetHeight() allow test of size changes for sender update without retaining sender dimensions in the application. The Webcam example shows how this could be useful.

```
unsigned int GetWidth();
unsigned int GetHeight();
```

If frame counting is enabled, the following functions retrieve further information.

```
long GetFrame();
double GetFps();
```

2.4 Receiver

1) Set receiver name

You can specify a sender for the receiver to connect to. The application will not connect to any other unless the user selects one. If that sender closes, the application will wait for the nominated sender to open.

```
void SetReceiverName(const char * SenderName);
```

2) Receive texture or pixel data

ReceiveTextureData and ReceiveImageData are new functions that connect to a sender and inform the application to update the receiving texture or buffer if the sender has changed dimensions.

```
// Receive texture data
bool ReceiveTextureData(GLuint TextureID, GLuint TextureTarget,
                        bool bInvert = false, GLuint HostFbo = 0);

// Receive pixel data
bool ReceiveImageData(unsigned char *pixels,
                      GLenum glFormat = GL_RGBA,
                      bool bInvert = false,  GLuint HostFbo = 0);
```

If these functions are used, CreateReceiver is no longer required, and it is not necessary to monitor sender size changes from within the application. At every cycle, before receiving any data, you must check whether the sender size has changed.

```
if(receiver.IsUpdated()) {
      .. update the receiving texture or buffer
}
```

3) Query the sender connection

If receiving fails, the functions return false, but you can also test this at any time.

```
If(receiver.IsConnected()) {
      .. perform processing as required
}
```

3) Query the received frame status

The receiving texture or pixel buffer is only refreshed if the sender has produced a new frame. This can be queried if there is any performance gain by processing the data only for new frames.

```
ReceiveTextureData(..);
if(myReceiver.IsFrameNew()) {
        . . . perform processing
}
. . . render the received texture
```

4) Query the sender details

At any time you can retrieve the connected sender name, size, frame rate and number.

```
// Return the connected sender name
const char * GetSenderName();
// Return the connected sender width
unsigned int GetSenderWidth();
// Return the connected sender height
unsigned int GetSenderHeight();
// Return the connected sender frame rate
double GetSenderFps();
// Return the connected sender frame number
long GetSenderFrame();
```

5) Sender selection

```
void SelectSender();
```

To improve legibility, this overload function has been renamed from SelectSenderPanel.

6) 2.006 functions

Earlier functions for receiver management have not been changed and can still be used as detailed in the Spout 2.006 documentation.

2.5 Summary of 2.007 sender and receiver functions

Sender

```
// Send texture attached to an fbo
bool SendFbo(GLuint FboID, unsigned int width, unsigned int height,
       bool bInvert=true);
// Return sender width
unsigned int GetWidth();
// Return sender height
unsigned int GetHeight();
// Return sender frame number
long GetFrame();
// Return sender frame rate
double GetFps();
// Sender frame rate control
void HoldFps(int fps);
```

Receiver

```
// Set the sender name to connect to
void SetReceiverName(const char * SenderName);
// Receive texture data
bool ReceiveTextureData(GLuint TextureID, GLuint TextureTarget,
       bool bInvert=false, GLuint HostFbo=0);
// Receive pixel data
bool ReceiveImageData(unsigned char *pixels, GLenum glFormat=GL_RGBA,
       bool bInvert=false, GLuint HostFbo=0);
// Return whether the connected sender has changed
bool IsUpdated();
// Return whether connected to a sender
bool IsConnected();
// Return the connected sender name
const char * GetSenderName();
// Return the connected sender width
unsigned int GetSenderWidth();
// Return the connected sender height
unsigned int GetSenderHeight();
// Return the connected sender frame rate
double GetSenderFps();
// Return the connected sender frame number
long GetSenderFrame();
// Return whether the received frame is new
bool IsFrameNew();
// Open the user sender selection dialog
void SelectSender();
```

Common

```
// Return frame count status
bool IsFrameCountEnabled();
// Disable frame counting for this application
void DisableFrameCount();
```

3. SpoutLibrary

SpoutLibrary, a library that can be used with C++ compilers other than Visual Studio, has been revised to enable all functions of the Sender and Receiver classes as well as access to utility functions in the SpoutUtils namespace.

The project is now comprised of four files :

SpoutLibrary.h
SpoutLibrary.cpp
SpoutFunctions.h
SpoutFunctions.cpp

The project will build for 32 bit or 64 bit using Visual Studio 2017.

The resulting library files in either case are SpoutLibrary.dll and SpoutLibrary.lib. When using these in an application only the SpoutLibrary.h header file is required.

The original example for CodeBlocks is out of date and will be removed and replaced by an example for Visual Studio that can be maintained.

There are no examples for other compilers, but the library has been tested with QT / MingW for both 32 and 64 bit.

4. DirectX examples

Although Spout is designed around OpenGL, it can also be used for DirectX applications. However, there have been no code examples included with the Spout SDK.

More recently, tutorials originally shipped in the legacy DirectX SDK, have been updated by Chuck Walbourn for DirectX 11.

“Tutorial 04” has been modified as a Spout sender and “Tutorial 07” as a receiver.

The examples require only a sub-set of the Spout SDK :

```
SpoutCommon.h // for dll build and utilities
SpoutSharedMemory // for sender name management
SpoutSenderNames // for sender creation and update
SpoutMemoryShare // for memoryshare fallback support
SpoutDirectX // for creating a shared texture
SpoutFrameCount // for mutex lock and new frame signal
SpoutCopy // for pixel copy
SpoutUtils // for logging utilites
```

A new class (`SpoutDX`) has been created for management of the DirectX functions.

Modifications to the tutorial code can be found by search on “Spout”. They are minimal in order to work with the samples but should provide a guide for other DirectX 11 applications.

5. SpoutCam

SpoutCam has been extensively revised to use DirectX instead of OpenGL and is now in a separate repository, thanks to the work of Valentin Schmidt who created a GitHub project incorporating the DirectShow base classes and build configurations for both 32 bit and 64 bit for Visual Studio 2017. Without his help, SpoutCam could not have been updated.

His other DirectShow projects, including SpoutGrabber, SpoutRenderer, equivalents for Newtek NDI as well as encoder/decoder and renderer for HAP video should also be mentioned.

Now SpoutCam is available for both 32 bit and 64 bit host programs and can be maintained using Visual Studio 2017.

5.1 SpoutCamSettings

"SpoutCamSettings" allows SpoutCam frame rate and dimensions to be set as a typical webcam to assist with compatibility.

For 2.007, the program has been updated to allow register/un-register of SpoutCam.

Registration is normally done by the Spout installation with an option to register or not. These changes avoid the need for re-installation or manual registration.