SpoutDX

Contents

What is it ?

DirectX textures can be shared using the Spout protocol without requiring OpenGL by using a sub-set of the Spout SDK classes and the D3D11 device already created.

SpoutDX is a class that manages these low level methods and provides simple send and receive functions.

Why a separate class ?

Because the complete Spout SDK requires support from the application for the required OpenGL extensions and functions.

Usage

Where the complete Spout SDK is not used, Spout texture sharing can be achieved in three ways :

1) DirectX textures using methods from the Spout SDK classes directly.
   - A Direct3D 11.0 device is passed from the application.

2) DirectX textures using the SpoutDX support class with simple texture sending and receiving functions.
   - A Direct3D 11.0 device is passed from the application.

3) Applications that do not have either D3D11 or OpenGL can send and receive pixel buffer images.
   - A Direct3D 11.0 device is created within the SpoutDX class.

Requirements for DirectX applications

For either the basic methods or with use of the SpoutDX class, a D3D11 device with feature level 11.0 is required. D3D9 is not supported.

Because textures could be shared with OpenGL applications, devices and textures must comply with the requirements outlined in the NVIDIA *WGL_NV_DX_interop* specifications :

https://www.opengl.org/registry/specs/NV/DX_interop.txt
https://www.opengl.org/registry/specs/NV/DX_interop2.txt

When creating the D3D11 device, the **D3DCREATE_MULTITHREADED** behaviour flag must be set.

The specification requires that **D3D11_CREATE_DEVICE_SINGLETHREADED** is **not** set, so that flag can be zero.

Shared textures must be created with the following :

1) Description Usage flag - **D3D11_USAGE_DEFAULT**
2) Description MiscFlags - **D3D11_RESOURCE_MISC_SHARED**
3) BindFlags - **D3D11_BIND_RENDER_TARGET | D3D11_BIND_SHADER_RESOURCE**
4) Format **DXGI_FORMAT_B8G8R8A8_UNORM** or **DXGI_FORMAT_R8G8B8A8_UNORM**

Use **DXGI_FORMAT_B8G8R8A8_UNORM** for best compatibility. DirectX 9 applications require this format.

Reading this manual

Read this manual together with the source code of each example to gain a better understanding of the functions required and when each should be used.

Examples for the various options are well commented throughout. Search "SPOUT" for details.

1. Basic DirectX examples

Functions from the *SpoutDirectX*, *SpoutSenderNames* and *SpoutFrameCount* classes can be used directly for shared texture synchronization and copy.

The basic methods require more steps, but may be preferred if more control is required over when and where these functions are used.

If that level of control is not required, the SpoutDX class can be considered because it manages the lower level methods and presents much more simple texture send and receive functions.

Add these files to your project

```
SpoutCommon.h
SpoutCopy.cpp
SpoutCopy.h
SpoutDirectX.cpp
SpoutDirectX.h
SpoutFrameCount.cpp
SpoutFrameCount.h
SpoutSenderNames.cpp
SpoutSenderNames.h
SpoutSharedMemory.cpp
SpoutSharedMemory.h
SpoutUtils.cpp
SpoutUtils.h
```

Examples are adapted from DirectX tutorial samples by Chuck Walbourn.

The methods in the the following examples apply in general for any application where a Direct3D 11.0 device is available.

1.1 Sender

Basic DirectX example using functions directly from Spout SDK classes based on the Tutorial04 sample.

Setup

Includes

```
// Change paths as required
#include "SpoutSenderNames.h" // for sender creation and update
#include "SpoutDirectX.h" // for creating a shared texture
#include "SpoutFrameCount.h" // for mutex lock and new frame signal
#include "SpoutUtils.h" // for logging utilites
```

Objects

```
spoutSenderNames spoutSender;
spoutDirectX spoutdx;
spoutFrameCount frame;
```

Global variables (for Tutorial04 basic example)

```
char g_SenderName[256];
unsigned int g_Width = 0;
unsigned int g_Height = 0;
ID3D11Texture2D* g_pSharedTexture = nullptr; // Texture to be shared
HANDLE g_dxShareHandle = NULL; // Share handle for the sender
bool bSpoutInitialized = false;
```

Enable Spout logging if required

```
// OpenSpoutConsole(); // console only for debugging
// EnableSpoutLog(); // Log to console
// EnableSpoutLogFile("Tutorial04.log); // Log to file
// SetSpoutLogLevel(SPOUT_LOG_WARNING); // show only warnings and errors
```

After window and device creation

Give the sender a name

```
strcpy_s(g_SenderName, 256, "Tutorial04");
```

Create a sender mutex for access to the shared texture

```
frame.CreateAccessMutex(g_SenderName);
```

During render

Send the texture (the Tutorial04 example uses the backbuffer).

1. Get the texture details (width height and format)
2. If a sender has not been created yet :
   - Create a shared texture of the same size and format. The format should be either DXGI_FORMAT_B8G8R8A8_UNORM or DXGI_FORMAT_R8G8B8A8_UNORM.
   - Create a sender using the shared texture handle
   - Create a sender mutex to control access to the shared texture
   - Enable frame counting for sender frame number and fps
3. If the sender has already been created
   - Check for size change
     - Update the sender and global variables
4. Send the texture
   - Check the sender mutex for texture access
     - Copy the application texture to the sender shared texture
     - Flush immediate context
     - Set a new frame
     - Allow access to the shared texture

Fps control

Hold a target rate if necessary – see code comments

```
frame.HoldFps(60);
```

On program close

Release the sender name from the list of senders.

```
spoutSender.ReleaseSenderName(g_SenderName);
```

Close the texture access mutex

```
frame.CloseAccessMutex();
```

Close frame counting

```
frame.CleanupFrameCount();
```

Release application resources

```
if (g_pSharedTexture) g_pSharedTexture->Release();
```

1.2 Receiver

Basic DirectX example using functions directly from Spout SDK classes based on the Tutorial07 sample.

Includes

```
// Change paths as required
#include "SpoutSenderNames.h" // sender creation and update
#include "SpoutDirectX.h" // for creating a shared texture
#include "SpoutFrameCount.h" // mutex lock and new frame
#include "SpoutUtils.h" // for logging utilites
#include <direct.h> // for _getcwd
#include <TlHelp32.h> // for PROCESSENTRY32
#include <tchar.h> // for _tcsicmp
```

Objects

```
spoutSenderNames spoutSender; // the sender receiving from
spoutDirectX spoutdx;
spoutFrameCount frame;
```

Global variables and utility functions (for Tutorial07 basic example)

```
ID3D11Texture2D* g_pReceivedTexture = nullptr; // Receiving texture
ID3D11ShaderResourceView* g_pSpoutTextureRV = nullptr; // Shader resource view
char g_SenderName[256]; // Sender name
char g_SenderNameSetup[256]; // Sender name to connect to
unsigned int g_Width = 0; // Sender width
unsigned int g_Height = 0; // sender height
long g_senderframe = 0; // Sender frame number
double g_senderfps = 0.0; // Sender frame rate
bool bNewFrame = false; // The received frame is new
bool bSpoutInitialized = false; // Initialized for the connected sender
bool bSpoutPanelOpened = false; // User opened sender selection panel
bool bSpoutPanelActive = false; // Selection panel is still open
SHELLEXECUTEINFOA g_ShExecInfo; // Global info so the exit code can be tested
bool OpenSpoutPanel(); // User sender selection dialog
bool CheckSpoutPanel(char *sendername, int maxchars = 256);
```

Enable Spout logging if required

```
// OpenSpoutConsole(); // Console only for debugging
// EnableSpoutLog(); // Log to console
// EnableSpoutLogFile("Tutorial07.log); // Log to file
// SetSpoutLogLevel(SPOUT_LOG_WARNING); // Show only warnings and errors
```

After window and device creation

Optionally set the name of the sender to receive from. The receiver will wait for that sender to open. The user can over-ride this by selecting another.

```
// strcpy_s(g_SenderNameSetup, 256, "Spout DX11 Sender"); // Starting name
// strcpy_s(g_SenderName, 256, "Spout DX11 Sender"); // General name as well
```

During render

Here pseudo-code is used to explain the steps involved. Examine the Tutorial07 Basic example code for further information.

1. Create initial width and height variables and set global values for testing sender size change
2. If relevant for the application, check for user activation of the sender selection dialog (see OpenSpoutPanel utility function).
   - If a new sender has been selected, the name will be different
     - Reset all variables
     - Close access mutex and frame counting for the current sender
     - Reset intialization flag
3. Find whether a sender with the current name exists.
   If no name has been provided, the "active" sender will be detected.
   - If a sender was found
     - If not connected yet
       - Create a mutex to control access to the sender shared texture
       - Enable frame counting to get sender frame number and fps
       - Set initialization flag
     - Check for sender size changes
       - Create or re-create the receiving texture
     - Retrieve the sender's shared texture pointer
     - Check for access to the shared texture
       - Test for a new frame from the sender
       - Copy the sender's shared texture to the receiving texture
       - Set a new frame flag (it will be tested later)
       - Allow access to the sender's shared texture
     - For a new frame
       - Use the received texture as required
     - The sender frame number and frame rate can be retrieved
   - If no sender was found
     - If previously connected
       - If a connecting name has been set, reset the sender name to it. Otherwise zero the sender name
       - Zero the sender width and height
       - Close the access mutex and frame counting
       - Clear any application resources using the received texture
4. Continue with render
5. Final swapchain present
6. Hold a target frame rate if necessary – see code comments

On program close

```
// Close the named texture access mutex
frame.CloseAccessMutex();

// Close frame counting
frame.CleanupFrameCount();

// Release application resources
if (g_pSpoutTextureRV)
	g_pSpoutTextureRV->Release();

if (g_pReceivedTexture)
	g_pReceivedTexture->Release();
```

2. The SpoutDX support class

Add these files to your project

```
SpoutDX.cpp // Additional to basic examples
SpoutDX.h // Additional to basic examples
SpoutCommon.h
SpoutCopy.cpp
SpoutCopy.h
SpoutDirectX.cpp
SpoutDirectX.h
SpoutFrameCount.cpp
SpoutFrameCount.h
SpoutSenderNames.cpp
SpoutSenderNames.h
SpoutSharedMemory.cpp
SpoutSharedMemory.h
SpoutUtils.cpp
SpoutUtils.h
```

2.1 DirectX examples

For applications where a DirectX 11.0 device is already created, the SpoutDX class can be used to manage shared texture synchronization and copy.

Simple sending and receiving functions are then used which handle sender creation, connection and size changes.

The SpoutDX examples are adapted from DirectX tutorial samples by Chuck Walbourn. The source is exactly the same as for the Basic examples, but converted to use the SpoutDX class.

SendTexture

```
bool SendTexture(ID3D11Texture2D* pTexture);
```

SendTexture manages sender creation and update based on the size of the texture passed in. A shared texture is created and updated by copy from the application texture.

ReceiveTexture

```
bool ReceiveTexture(ID3D11Texture2D** ppTexture = nullptr);
```

ReceiveTexture waits for a sender unless connected already. Once connected and the sender size is known or has changed, the application is informed so that the receiving texture can be updated. The application monitors change using :

```
bool IsUpdated();
```

The receiving texture is updated by copy from the sender shared texture.

If *ReceiveTexture* fails, the sender has closed.

2.2 Sender

DirectX example using the SpoutDX class and *SendTexture.*
Based on the Tutorial07 sample.

Setup

Includes

```
#include "SpoutDX.h"// Change path as required
```

Objects

```
spoutDX sender;
```

Enable Spout logging if required

```
// OpenSpoutConsole(); // console only for debugging
// EnableSpoutLog(); // Log to console
// EnableSpoutLogFile("Tutorial04.log); // Log to file
// SetSpoutLogLevel(SPOUT_LOG_WARNING); // show only warnings and errors
```

After window and device creation

The DirectX 11.0 device pointer must be passed to the SpoutDX class so that texture copy uses the same device.

```
if (!sender.OpenDirectX11(g_pd3dDevice))
        return FALSE;
```

Give the sender a name. If none is specified, the executable name is used.

```
sender.SetSenderName("Tutorial04 sender");
```

During render

Send the texture (the Tutorial04 example uses the backbuffer).
SendTexture handles sender creation and re-sizing

```
sender.SendTexture(pBackBuffer);
```

Hold a target rate if necessary – see code comments

```
sender.HoldFps(60);
```

On program close

Close the sender

```
sender.ReleaseSender();
```

Release application resources.

2.3 Receiver

DirectX example using the SpoutDX class and *ReceiveTexture.*
Based on the Tutorial07 sample.

Includes

```
#include "SpoutDX.h" // Change path as required
```

Objects

```
spoutDX receiver;
```

Global variables (for Tutorial07 example)

```
ID3D11Texture2D* g_pReceivedTexture = nullptr; // Receiving texture
ID3D11ShaderResourceView* g_pSpoutTextureRV = nullptr; // Shader resource view
```

Enable Spout logging if required

```
// OpenSpoutConsole(); // Console only for debugging
// EnableSpoutLog(); // Log to console
// EnableSpoutLogFile("Windows sender.log); // Log to file
// SetSpoutLogLevel(SPOUT_LOG_WARNING); // Show only warnings and errors
```

Optionally set the name of the sender to receive from. The receiver will only connect to that sender. The user can over-ride this by selecting another.

```
// receiver.SetReceiverName("Spout DX11 Sender");
```

After window and device creation

The DirectX 11.0 device pointer must be passed to the SpoutDX class so that texture copy uses the same device.

```
if (!receiver.OpenDirectX11(g_pd3dDevice))
	return FALSE;
```

During render

1) Receive a texture from a sender
   - If a sender was found
   - If the sender is updated
     - Create or re-create the receiving texture
   - If the frame is new
     - Create or re-create associated resources
2) If the sender was not found or closed
   - Release the receiving texture
   - Release associated resources

On program close

Close the receiver.

```
receiver.ReleaseReceiver();
```

Close DirectX

```
sender.CleanupDX11();
```

2.4 Windows examples

These examples use the Microsoft Win32 API and do not involve textures, either DirectX or OpenGL. SpoutDX class functions *SendImage* and *ReceiveImage* are used to send or receive pixel buffer images.

The pixel images are converted to or from Bitmaps, but SpoutDX does not depend on the Win32 API and can be used for any application.

SendImage

```
bool SendImage(unsigned char * pData,
               unsigned int width, unsigned int height);
```

SendImage manages sender creation and update based on the size passed in. A shared texture is created and updated using data from the pixel buffer.

ReceiveImage

```
bool ReceiveImage(unsigned char * pixels, bool bInvert = false);
```

ReceiveImage waits for a sender unless connected already. Once connected and the sender size is known or has changed, the application is informed of size changes so that the receiving pixel buffer can be updated. The application monitors change using :

```
bool IsUpdated();
```

The receiving pixel buffer is updated by copy from the sender shared texture by way of a DirectX staging texture.

If *ReceiveImage* fails, the sender has closed.

2.5 Sender

Win32 API example using *SendImage* from the SpoutDX class.

Setup

Includes

```
#include "SpoutDX.h" // Change path as required
```

Objects

```
spoutDX sender;
```

Global variables

```
HWND g_hWnd = NULL;                     // Window handle
spoutDX sender;                         // Sender object
HBITMAP g_hBitmap = NULL;               // Image bitmap for sending
unsigned int g_BitmapWidth = 0;         // Image bitmap width
unsigned int g_BitmapHeight = 0;        // Image bitmap height
unsigned char *g_pixelBuffer = nullptr; // Sending pixel buffer
unsigned char g_SenderName[256];        // Sender name
unsigned int g_SenderWidth = 0;         // Sender width
unsigned int g_SenderHeight = 0;        // Sender height
```

Enable Spout logging if required

```
// OpenSpoutConsole(); // console only for debugging
// EnableSpoutLog(); // Log to console
// EnableSpoutLogFile("Windows sender.log); // Log to file
// SetSpoutLogLevel(SPOUT_LOG_WARNING); // show only warnings and errors
```

Create a bitmap to send. In this example we simply load an image from file. In other applications the bitmap might be created from another source or from the screen contents for example

Create a sending pixel buffer. The size does not matter at this stage. It is re-sized later as necessary.

After application initialization

Prepare DirectX 11 for Spout functions. A DirectX 11.0 device is created within the SpoutDX class.

```
if (!sender.OpenDirectX11())
	return FALSE;
```

Give the sender a name. If none is specified, the executable name is used.

```
sender.SetSenderName("Windows sender");
```

Main message loop

Include a Render() function which is called after messages have been processed.

```
MSG msg = { 0 };
while (WM_QUIT != msg.message)
{
    if (PeekMessage(&msg, nullptr, 0, 0, PM_REMOVE))
    {
         TranslateMessage(&msg);
         DispatchMessage(&msg);
    }
    else
    {
         Render();
    }
 }
```

Render

In this example, the Render function triggers a re-paint to draw the bitmap and load the pixel buffer for sending.

```
 void Render()
 {
     // Trigger a re-paint to draw the bitmap
     // and refresh the sending pixel buffer - see WM_PAINT
     InvalidateRect(g_hWnd, NULL, FALSE);
     UpdateWindow(g_hWnd); // Update immediately
     // Send the pixels
     sender.SendImage(g_pixelBuffer, g_SenderWidth, g_SenderHeight);
     // Optionally hold a target frame rate - e.g. 60 or 30fps.
     sender.HoldFps(60);
 }
```

WM_PAINT

1) Draw the image bitmap to the screen
2) Check the screen size and update as necessary
3) Copy the screen to the pixel buffer
4) Send the buffer

Examine the code in WM_PAINT for details.

On program close

Release application resources.

Close the sender

```
 sender.ReleaseSender();
```

Close DirectX

```
 sender.CleanupDX11();
```

2.6 Receiver

Win32 API example using *ReceiveImage* from the SpoutDX class.

Setup

Includes

```
#include "SpoutDX.h" // Change path as required
```

Objects

```
spoutDX receiver;
```

Global variables

```
HWND g_hWnd = NULL;                    // Window handle
spoutDX receiver;                      // Receiver object
HWND g_hWnd = NULL;                    // Window handle
unsigned char *pixelBuffer = nullptr; // Receiving pixel buffer
unsigned char *bgraBuffer = nullptr;  // Conversion buffer if required
unsigned char g_SenderName[256];       // Received sender name
unsigned int g_SenderWidth = 0;        // Received sender width
unsigned int g_SenderHeight = 0;       // Received sender height
DWORD g_SenderFormat = 0;              // Received sender format
```

Enable Spout logging if required

```
// OpenSpoutConsole(); // console only for debugging
// EnableSpoutLog(); // Log to console
// EnableSpoutLogFile("Windows receiver.log); // Log to file
// SetSpoutLogLevel(SPOUT_LOG_WARNING); // show only warnings and errors
```

Set a receiving sender name

To receive from a specific sender, set the sender name here. If the application allows it, the user can over-ride this by selecting another.

```
// receiver.SetReceiverName("Spout DX11 Sender");
```

After application initialization

Prepare DirectX 11 for Spout functions. A DirectX 11.0 device is created within the SpoutDX class.

```
if (!receiver.OpenDirectX11())
        return FALSE;
```

Main message loop

Include a Render() function which is called after messages have been processed.

```
MSG msg = { 0 };
while (WM_QUIT != msg.message)
{
    if (PeekMessage(&msg, nullptr, 0, 0, PM_REMOVE))
    {
         TranslateMessage(&msg);
         DispatchMessage(&msg);
    }
    else
    {
         Render();
    }
  }
```

Render

In this example, the Render function receives a pixel buffer from a sender, updates the receiving buffer size if necessary and triggers a re-paint to draw the pixel buffer. Examine the example code in WM_PAINT for details.

```
void Render()
{
  // Get pixels from the sender shared texture.
  // ReceiveImage handles sender detection, creation and update.
  // The data is flipped here ready for WM_PAINT.
  if (receiver.ReceiveImage(pixelBuffer, true)) {
      // IsUpdated() returns true if the sender has changed
      if (receiver.IsUpdated()) {
          // Update the sender name - it could be different
          strcpy_s((char *)g_SenderName, 256, receiver.GetSenderName());
          // Update globals
          g_SenderWidth = receiver.GetSenderWidth();
          g_SenderHeight = receiver.GetSenderHeight();
          g_SenderFormat = receiver.GetSenderFormat();
          // Update the receiving buffer
          if(pixelBuffer) delete pixelBuffer;
          pixelBuffer = new unsigned char[g_SenderWidth*g_SenderHeight*4];
          // Update the rgba > bgra conversion buffer
          if (bgraBuffer) delete bgraBuffer;
          bgraBuffer = new unsigned char[g_SenderWidth*g_SenderHeight*4];
          // Do anything else necessary for the application here
      }
  }

  // Trigger a re-paint to draw the pixel buffer - see WM_PAINT
  InvalidateRect(g_hWnd, NULL, FALSE);
  UpdateWindow(g_hWnd); // Update immediately

  // Optionally hold a target frame rate - e.g. 60 or 30fps.
  receiver.HoldFps(60);
```

WM_PAINT

Copy a pixel buffer to the screen. The buffer can be either BGRA or RGBA format. For Win32 bitmaps, BGRA can be copied directly but RGBA requires conversion. This can be done either by copy to a conversion buffer or using Win32 API methods.

1. Set up a bitmap info structure with the received image size
2. If the image is RGBA and width is a multiple of 16
   - Use a high speed function to copy a BGRA buffer
3. If the image is RGBA and width is not a multiple of 16
   - Use the extended BITMAPV4HEADER to specify pixel masks
4. Use StretchDibits to copy the BGRA buffer to the screen

WM_RBUTTONDOWN

The user has clicked the RH mouse button. Activate the sender selection panel.

```
receiver.SelectSender();
```

On program close

Close the receiver and release application resources.

```
receiver.ReleaseReceiver();
```

Close DirectX

```
receiver.CleanupDX11();
```