Linking Up with DIRECTPLAY

by Semion S.Bezrukov
deltree@rocketmail.com

INTRODUCING DIRECT PLAY

Direct Play is an API that allows easy setup and maintenance of a networked multiplayer environment. Direct Play functions provide an easy access to the HAL and HEL network layers. They isolate you as a programmer from the specifics of IPX/SPX, TCP/IP, serial and modem protocols. Direct Play also provides useful abstractions such as player and group objects, which simplify higher level design. However, Direct Play is very difficult to initialize – you will have to go through a series of complicated steps before you will be able to get any confirmation of correct setup.

A Brief Technical Overview

The following is a very brief overview of the basics of Direct Play. I will cover some theory behind the setup of a Direct Play object and player. After that, I will cover various methods that are used to transmit and receive data over the network.

Initialization and Setup of Direct Play

Here are the steps that you will have to go through before you can start transmitting data:

1 – Create the Direct Play object

At this point, use the newer method of CoCreateInstance. Do not use the DirectPlay supplied DirectPlayCreate function as it is obsolete.

2 – Select connection type

Here, you will have to use the EnumConnections method to enumerate the available connections.

3 – Initialize the DP object

Use the selected connection type to finish the initialization of the Direct Play object with the InitializeConnection method.

4 – Look for existing DP sessions

Now, look for the existing DP sessions and enumerate them with the EnumSessions method. There are two ways that you can proceed here: enumerate synchronously or asynchronously. Asynchronous enumeration is much better, since Direct Play uses a thread to do the enumeration for you. All you have to do is pass the DPENUMSESSIONS_ASYNC flag to the EnumSessions method and all will be handled for you.

5 – Join a session or start a new one

Here, use the Open method to join a session or create a new one. Now a GUID will come into play. GUID stands for global unique identifier. It is a 32-digit hexadecimal number that is guaranteed to be unique. It is usually derived from your network card id. The GUID is used by Direct Play to identify the Direct Play application, session and data being sent across a network. A valid GUID can be generated using 'guidgen.exe' that is included with a number of compilers. The Open method will be passed a DPSESSIONDESC2 structure that will contain the GUID, the number of players and the session description. Every application should have a distinct GUID, as it will be used as a basis for locating compatible sessions on the network. DPSESSION_MIGRATEHOST flag is also generally specified, so that if the player who created the session quits, the hosting responsibilities are automatically transferred by Direct Play to another player and he is notified of this by the DPSYS_HOST system message.

6 – Create a player

This is the last required step in our initialization process. You will create a player using the CreatePlayer method. This will cause Direct Play to assign a Direct Player ID to the player (which is different every time), that will be used for all the communication purposes.

7 – *Create or Join a Group*

Although this is not a necessary step, you may consider at this point creating or joining a group. Using the EnumGroups method you can enumerate the available groups and join by using the AddPlayerToGroup

method. If you want to create a new group, use the CreateGroup method. Groups can be useful in preserving bandwidth if your network supports multicasting. Multicasting allows a message that is addressed to all the players to be sent as a one step process, instead of forcing Direct Play to make separate copies for each player.

Direct Play Messages

After you complete the final initialization step, you are ready to start sending and receiving data. Direct Play uses three ways to propagate data through the network: System Messages, Custom Messages and Shared Data Areas.

System Messages

System Messages are used by Direct Play to keep each player informed of the session's state. For example: DPSYS_HOST message is sent to the player who becomes the new host once the original host has left (provided that DPSESSION_MIGRATEHOST flag was set when the session was initialized); or DPSYS_CREATE_PLAYERORGROUP message is sent when a new player or group has been created.

Custom Messages

Basically, the custom messages are used to transmit the rest of the information. In your game, custom messages will comprise 99% of the network traffic. All of the internal game messages, for example player actions, will be handled through custom messages. Custom messages come in two flavors: guaranteed and non-guaranteed. When you send a message using the Send method, specifying the DPSEND_GURANTEED flag will make the message guaranteed. It is very important to understand the difference between the two message types. Guaranteed messages are just that – they are guaranteed to arrive to their destination. Unfortunately, that makes them very slow, since the sending computer must verify that a message got to its destination. Here is where the non-guaranteed messages come in. They are very fast. In fact, they are the fastest type of

communication allowed by Direct Play. They only flaw is that you have no way of making sure that they arrive at their destination. In fact, you are guaranteed to loose some of these messages, especially when there is a heavy network load or you are sending a lot of them. Thus, your program will need a method for compensating for this possible loss of data. I will list a few approaches designed to deal with this problem later on in the chapter.

Shared Data Areas

Shared Data Areas are associated with players and groups. The data contained in them is automatically propagated though the network. There is one major problem associated with the shared data areas that makes them unsuitable for many of our purposes – they are extremely slow (slower than guaranteed messages). They are only useful if you use them to store data that does not change often and is seldom required by any player or process.

Sending Messages

There is one more major constriction on messages that is provided by DirectPlay – their size. While it is possible to transmit a message of any length, as message data is transmitted in packets, there is an implied limitation since if one packet is not received, the whole message is discarded. Therefore, it is in your best interest to keep the messages as small and compact as possible.

Here is as example of how to send a message. Let's assume the following declarations:

```
typedef struct _GENERICMSG          //the generic message type
{
        BYTE msgType;
} GENERICMSG, *LPGENERICMSG;

#define MsgFireType  1                  //each message has a distinct type
```

```
#define MsgHostType 2

typedef struct _FIREMESSG
{
        BYTE msgType;           //this indicates the type of message
        int data1;              //data
        char data2;             //data

} FIREMSG, *LPFIREMSG;
```

To send this struct across the network, all you have to do is fill in the information and use the SendGameMessage function.

```
FIREMSG MsgFire;

MsgFire.msgType = MsgFireType;
MsgFire.data1 = some_data1;
MsgFire.data2 = some_data2;
SendGameMessage( (LPGENERICMSG)&MsgFire, DPID_ALLPLAYERS );
```

The SendGameMessage function calculates the size of the message and then sends it using the Send method. Note how we cast the message to the GENERICMSG type. This is done for the following reason: when data is sent it is sent in a buffer. When the buffer is received the receiving computer knows nothing of it's type. Therefore the message will be cast to the generic type, the 'msgType' variable will be extracted and then based on it, the message will be cast to the exact type and the rest of the data will be accessible.

Receiving Messages

There are numerous ways to receive an incoming message. When it arrives, the message is placed on a message queue. Using the Receive method you can retrieve it in various way off the queue. For example: by specifying the PDRECIEVE_PEEK flag, the message will be received without actually being removed from the queue; or by specifying the DPRECIEVE_FROMPLAYER flag, the first message from the player identified by the lpidFrom parameter, will be received.

Here is an example of how the messages are received by first casting them to the generic type, extracting the msgType variable, and then casting them to their specific type.

```
pGeneric = (DPMSG_GENERIC *) lpReceiveBuffer;
switch( lpGeneric->msgType )
  {
      case MsgHostType:
          //we've been transferred host responsibilities, do stuff
      break;

      case MsgFireType:
          LPFIREMSG  lpFire;
          lpFire = (LPFIREMSG) lpGeneric;        //cast to specific type
          some_data1 = lpFire->data1;            //extract data
          some_data2 = lpFire->data2;            //extract data
      break;

      default : break;
 }
```

Introducing DPTools

The DPTools are a collection of Direct Play routines, compiled by me that will greatly simplify your introduction to all aspects of Direct Play. The key problem that I encountered when I tried to add networking support to one of my games was that there were no clearly written examples! The explanations given in the SDK help file were too general. I wanted to see real, working code. Thus, I give you DPTools. This is all that you have to do to get Direct Play working for you:

```
//your other includes go here
#include "dptools.h"

//in your initialization procedure
bool Init()
{

if !(InitializeDirectPlay(hinstance))
  {
    MessageBox(hWnd, "Error initializing Direct Play", "Error", MB_OK);
    Return(FALSE);
```

```
    }
//all your other initialization stuff goes here
}
```

Well, actually there is another addition to your program – you must add a resource file dpex.rc and call *UninitializeDirectPlay()* whenever you exit your program to destroy the local player and free the Direct Play object. But that's all!

A Note on Implementation Strategy

The DPTools code is implemented in a thread, giving it unparalleled flexibility and efficiency. A thread is simply a process that activates another process when a certain event occurs. While that awaking event has not occurred, the thread is 'asleep' and does not eat-up the precious CPU time. Thus the Direct Play message queue can and should be monitored via a thread. A thread implementation minimizes the number of CPU cycles that it takes to check the Direct Play queue for incoming messages. When such a message arrives, the thread awakes and instructs a separate procedure to begin processing the incoming message. There are a few precautions that you must take when working in a threaded environment. If you use dynamic allocation, you will need to establish protected code segments, which can be accessed by only one process at a time. Statically allocated variables are safer, but you should still consider protecting the important code segments.

Sample Program

This program opens a text window and begins to increment a counter. This counter is displayed on the left of the screen and also transmitted over the network. The counter on the right is the receive counter which displays the data that the program is receiving. Thus if there are two of those programs running, they should display their own counter on the left and each-other's counter on the right. Simple, no?
Here's the code:

```
#include "dpex.h"
#define WINDOW_CLASS_NAME "WINCLASS1"
```

```
#define WINDOW_WIDTH  400
#define WINDOW_HEIGHT 100

char        buffer[80];                 //to hold the print messages
BOOL        g_bFullScreen = FALSE;
BOOL        g_bAllowWindowed = TRUE;
BOOL        g_bReInitialize = FALSE;
int         send_count=0;               //counter we'll be sending
int         received_count=0;           //store counter we recieve
int         length = 0;                 //of message


long FAR PASCAL WindowProc( HWND hWnd, UINT message,
                    WPARAM wParam, LPARAM lParam )
{
   switch( message )
   {
        case WM_ACTIVATEAPP:
          break;

         case WM_CREATE:
           break;

        case WM_PAINT:
          break;

         case WM_DESTROY:
         {
              PostQuitMessage(0);
              return(0);
         }

        default : break;
   }

   return DefWindowProc(hWnd, message, wParam, lParam);
}



int WINAPI WinMain(          HINSTANCE hinstance,
                             HINSTANCE hprevinstance,
                             LPSTR lpcmdline,
                             int ncmdshow)
{
```

```
WNDCLASS    winclass;                   // this will hold the class we create
HWND        hwnd;                       // generic window handle
MSG         msg;                        // generic message
HDC         hdc;                        // generic dc


winclass.style                  = CS_DBLCLKS | CS_OWNDC |
                                  CS_HREDRAW | CS_VREDRAW;
winclass.lpfnWndProc            = WindowProc;
winclass.cbClsExtra             = 0;
winclass.cbWndExtra             = 0;
winclass.hInstance              = hinstance;
winclass.hIcon                  = LoadIcon(NULL, IDI_APPLICATION);
winclass.hCursor                = LoadCursor(NULL, IDC_ARROW);
winclass.hbrBackground          = (HBRUSH)GetStockObject(BLACK_BRUSH);
winclass.lpszMenuName           = NULL;
winclass.lpszClassName          = WINDOW_CLASS_NAME;

// register the window class
if (!RegisterClass(&winclass))
        return(0);

//initialize Direct Play
if (!(InitializeDirectPlay(hinstance)))
        return(0);

// create the window
if (!(hwnd = CreateWindow(  WINDOW_CLASS_NAME,
                            "Direct Play Send/Receive Demo",
                            WS_OVERLAPPEDWINDOW | WS_VISIBLE,
                            0,0,
                            WINDOW_WIDTH,
                            WINDOW_HEIGHT,
                            NULL,
                            NULL,
                            hinstance,
                            NULL)))
  return(0);

while(1)
   {
        if( PeekMessage( &msg, NULL, 0, 0, PM_REMOVE ) )
          {
                if (msg.message == WM_QUIT)
                  break;
```

```
            TranslateMessage(&msg);
            DispatchMessage(&msg);
        }
          else
            {

            hdc = GetDC(hwnd);                          //to output text
                                                                //display counters
            length = sprintf(buffer,"SENT counter: %d   ", send_count++);
            TextOut(hdc,0,0,buffer,length);
            length = sprintf(buffer,"RECIEVED : %d   ", received_count);
            TextOut(hdc,260,0,buffer,length);

            if (send_count > 30)
               send_count = 0;
            ReleaseDC(hwnd,hdc);                        //finish text operations

            SendFireMessage(send_count);                //send the counter
            received_count = ReceiveFireMessage();      //get the received count

            Sleep(200);                                 //wait a sec
          }
        }

    UninitializeDirectPlay();                           //deinitialize DP
    return(msg.wParam);                                 //quit to windows like so
}
```

INTEGRATING DIRECT PLAY INTO YOUR GAME

So, you have an excellent single-player game and you want to make it multiplayer. Beware, there are certain inherent problems that you must resolve before you can ask, “So, who’s going to host?”.

Graphic Concerns

One of the first problems that you will encounter is the increased graphics load. Your game engine will need to be able to display not only your player but others as well. That doesn’t seem too bad until you consider all the extra effects – explosions, sounds and sprites that you will need to deal with. The next concern will be with the logic engine. Will it be able to handle the multiplayer load? There is a simple test that you can run to answer most of your questions and tell you what needs further work. All you have

to do is run the logic engine and the player graphing engine ‘x’ number of times, where x is the number of players.

```
//somewhere in your main program loop that runs every frame

for (int x = 0; x < num_players; x++)
{
    //do all logic processing here
}

//draw background

for (x = 0; x < num_players; x++)
{
    //draw all game graphics(excluding the background)
}
```

Once you feel that your game engines are sufficient, you can face the second major problem.

Problems of Direct Play Overhead

No matter how efficiently you write the Direct Play implementation, the overhead will always be there – it will always take time to send, receive and process network messages. The thread implementation does minimize the overhead when the thread is waiting for a message, but once the message is received, the program will pause to process that message. This means that the game engine has to be fast enough to allow such slow-down periods. Thus, the Direct Play overhead once again pushes us to create a more efficient, faster game engine.

Synchronization and Processing Methods

In designing a network game, the synchronization method together with the processing method present the most important choices that you will have to make. For the synchronization method the two choices are the Rigid Sync Model and the Flexible Sync Model. For the processing method the two choices are: Local Processing and Remote Processing.

Rigid Synchronization Model

In the rigid synchronization model the two, or more, gaming universes are completely synchronized. A good example is a game of chess. Every player will see exactly the same board. When one player makes a move, an exact move will occur in the other player's game universe. This model works best for slow-paced games and is fairly straightforward.

Flexible Synchronization Model

Flexible synchronization model is the opposite of the rigid synchronization model. It sacrifices precision for speed. A player's action may or may not be reflected in an exactly same action in another player's universe. This type of synchronization is necessary for fast-paced games. However, this model presents us with a few associated problems. One is the need to send data quickly, which means that we have to use non-guaranteed messages and also deal with the fact that a number of these messages will be lost, even on an ideal network. Plus, there will be a few inconsistencies between the two universes and you may be forced to make decisions as to which universe is the 'right' one. For example, if player one shoots at player two and on his computer there is a hit, but on player two's computer there was a miss – who is right? The decision is largely based on the game itself and as such is up to you.

Remote Processing

In this scheme, each player processes all of his own input and sends his end-result actions to all other players. To take a car racing game example: each computer will run the physics engine and calculate the car motion based on player input and previous available data, such as previous velocity and heading. Then, it will send this player's car's orientation and position in absolute world coordinates to all the other players.

Good because:

- light load on the logic engine
- fairly easy to write the Direct Play implementation

Bad because:

- there is no way to predict other players' movement

- have to frequently send data, thus increasing the network load and chances of lost messages
- if one of the messages is not received, the car will appear to 'jump' as the next message will have the car's location far away

Local Processing

With this approach, each player sends his keystates to all other players, where the local engines process everyone's input. This sounds terrible, but in reality is more often used than the previous approach.

Good because:

- offers a possibility of motion prediction called Dead Reckoning, which is discussed later on in this chapter
- takes the strain off the network, since you don't have to send messages very often if you use keystates. Going back to our previous example, let's see how it would work with keystates. If player one presses the forward key, a message is sent, telling all other computers that the keystate has changed and specifically it was the state of the 'forward key'

Bad because:

- puts additional strain on the logic engine, since it will have to process input from all the players

Optimally, you would combine all of those approaches, sending the keystates as well as the position of your car to everyone, every time that the keystates changed. This seems inefficient, since we will be sending a lot of data, duplicating what may have been accomplished by using just one of these approaches. But, there is a method to this madness. Read on.

Dead Reckoning Approach

The concept behind Dead Reckoning is simple – when the input is unavailable, estimate the motion of other players based on previously available data. I personally

prefer estimating on the basis of previous keystates. For example: lets assume that we are writing a racing game. We will be using a flexible local synchronization model – meaning that we won't sync with extreme precision and that logic processing will take place locally on every machine. Thus, we will send everybody's keystates to everybody. Now assume we have three players in the game. Every player will possess a structure that will store his and other players' keystates (once they are received). Now let's say Player 1 presses forward. This will cause a message to be sent to Players 2 and 3 indicating that the keystate of the forward key has changed for Player 1. Thus Player 1's car will start moving forward on everybody's screen. Now, Player 1 presses left. The left message is sent to Players 2 and 3 but because of heavy net traffic only makes it to Player 2. Thus, Player 1 and 2's keystate records are the same while Player 3's record shows that Player 1 is still only pressing forward. So, on Player 1 and 2's screens the Player 1's car will start turning left, while on Player 3's screen it will only be moving forward. This seems to be a disaster, but in reality it is not that bad, since messages will be sent many times a second and during the next send the problem will correct itself. However, the effects of these desynchronizations will be cumulative, distorting the game universes over time. Thankfully, there is an easy solution. When we are sending keystates, let's also send the car's position and orientation in absolute world coordinates. So, even if a few messages are lost, once one message is received, the receiving computer will put you in the 'correct' position.

One of my colleagues, Adam Silkott, has developed an excellent algorithm that gradually adjusts the player to the correct position, if the 'perceived' position differs from the 'true' position, once the sync message is received. The basic idea involves dividing the adjustment value into small increments, to be applied over a few frames so as to reduce the 'jumping effect'.

Data Compaction Note

As I have mentioned before, it is in your best interest to reduce physical size of the sent message because if a part of it is not received, the whole message is discarded. Thus, if you are sending keyboard state data you should compress it. After all, the keystate variables are nothing but booleans, true when the key is down, false when it is

up. You can compress a few of them into an integer very quickly using shift operators like so:

```
int data=0;
data = data << Number_Flags;

data |= cmnd_fire;
data |= cmnd_left;
data |= cmnd_right;
data |= cmnd_forward;
data |= cmnd_reverse;

SendData(data);                 //sends the data
```

Once the variable data has been received, it can be uncompacted with this:

```
RecieveData(data);              //receives the data

cmnd_fire = data & 1;
cmnd_left = data & 2;
cmnd_right = data & 4;
cmnd_forward = data & 8;
cmnd_reverse = data & 16;
```

Why we hate latency

Latency is the amount of time that it takes for our action to occur in the game world after we've hit a key. Latency is always a factor in a computer game, no matter how small. As game programmers we hate latency. Unfortunately, we can't do anything about it, except to try to anticipate it and try to prevent it from completely desynchronizing our game universes. Fortunately, Direct Play has a few tools to help keep us informed about the state of latency. First, we can take a look at the dwLatency member of the DCAPS structure that is returned by the *GetPlayerCaps* method. It is a crude estimate of the network latency and is not updated in real-time. However, it is an excellent indicator of things to come. A yet better way is to use *GetMessageCount* method that returns the number of messages received by a particular player. You can compare this to the number of messages that you've sent and determine if you are sending more messages than the other machine can handle. 'But how do I use this data?', you ask. Simple – use a Temporal Synchronization Queue.

Temporal Synchronization Queue

The best part of inventing something is the authority that you get over naming it. However in this case, the name implies the exact nature of the approach. A queue acts as a delay for your own keystrokes within your own universe. If you strike a key, it is placed on top of the queue, the keystate message is sent to the other player(s) and a timer is started. The timer is equal to the latency of the network. Once the timer reaches zero, the key is taken off the queue and used as input. Any received keystate data is immediately used as input and not put on the queue. So in effect, your own input is slowed by the latency of the network so that it will occur on all the computers at the same time. Simple, isn't it?

Additional Resources

A good resource for detailed Direct Play function descriptions is the Direct Play help file, which is included with the DirectX Software Development Kit (SDK). If that's not enough for you, for an even more comprehensive coverage of Direct Play functions consult 'Inside DirectX' by Bradley Bargen and Peter Donnelly (ISBN 1-57231-696-9).