TP010: 3D Sound in RealiMation

Introduction

A frequently asked question regarding RealiMation is “How does it do sound?”. This paper aims to explain how sound can be integrated with a RealiMation application, and includes a programming tutorial that builds on the first tutorial described *in Technical Paper 004 : Your First RealiMation Application.*

This paper assumes that this first tutorial has been done. The result of that was to create a way of moving a tank object through a 3D world. This paper adds sound to that world, by adding 3D positional sounds to the tanks and the helicopter. The listening position will change as the eyepoint changes, so no matter what the camera is doing, the audio will be consistent. The fact that the sounds are 3D means that the spatial location of objects can be inferred from what the object sounds like.

The Sound System

The tutorial uses the Intel® Realistic Sound Experience (3D RSX) library to render multiple 3D sounds. It is freely available from the following web address:

http://developer.intel.com/ial/rsx/

The features of RSX include true 3D sound, reverberation control, doppler effects, and pitch shifting. It works well both from headphones and speakers. Minimum PC requirements is any Pentium CPU, 8Mb of memory, and Windows 95, 98, Windows NT 3.51 and 4. Recommended CPU is at least a 166Mhz MMX. The technology makes extensive use of the MMX instruction set if available.

There is no reason why any other 3D sound system could not be used instead of RSX. Some developers, for example, may already have their own sound subsystem.

Requirements

The Tank Application needs the following audio features:

- Sounds play when the RealiMation starts
- Sounds stop when the RealiMation stops
- Enable the listening position and orientation to be tied to the viewing position and orientation
- Enable sound files to be attached to the tanks and helicopters, and to move through space with them

The Audio Object

To add sound to the tank application, we create a new wrapper class called `CAudio`. This wraps the creation and manipulation of RSX sound objects, and links them to RealiMation cameras and instances (the camera is the listener, and the instances are the sound emitters).

The object needs to be initialised when the RealiMation starts, and un-initialised when it stops. It also needs a way of looking after multiple sounds, and joining up a sound file to a RealiMation object. Finally, it needs an update mechanism to enable the audio to be kept in sync with the visuals. Ideally, the interface to the `CAudio` object hides the fact that the actual implementation is using RSX, enabling easy switching to another audio API if desired.

In a similar fashion to how there is a global `CAnimate` object to look after the RealiMation motion, we define a global `CAudio` object to handle the application’s sounds. The following is the definition of the CAudio class.

```
/*_____________________________________*/
// TankApp - RealiBase viewer application.
//
// Copyright © Datapath Ltd. 1998
//
// Audio control class.
// This utility class uses Intel's RSX sound libraries, available on
//   http://developer.intel.com/ial/rsx
//

#ifndef __AUDIO_H__
#define __AUDIO_H__

class CAudio
{
   public :
      CAudio  ();
      ~CAudio ();

      void Initialise   (void);
      void UnInitialise (void);
      void SetEmitter   (rtID id, int index, CString &filename, Ord intensity = 1.0f);
      void SetBackground (CString &filename, Ord intensity = 0.2f);

      void Update (rtID view, Ord time);

      enum {MAXEMITTERS = 5};

   private :
      // RSX objects
      IRSX*                 m_lpRSX;   // RSX module
      IRSXDirectListener*  m_lpDL;     // Listener
      IRSXCachedEmitter*   m_lpCEback;            // Background emitter
      IRSXCachedEmitter*    m_lpCE [MAXEMITTERS]; // Array of emitters
      rtID                   m_idCE [MAXEMITTERS]; // RealiMation handles on emitters
};

// The one and only CAudio object
extern CAudio theAudio;

#endif
```

Create a new file called "`audio.h`", and paste in the above code. This class gives us an interface to upto 5 positional sounds (the `CAudio::MAXEMITTERS` constant), and a continuous background sound (for example, a music track). We will flesh out the implementation of this class later, but for moment we will concentrate on using the interfaces to it.

Initialising the Sound Environment

The first step is to initialise the RSX environment. This is actually a two step process. The first stage is to initialise the COM environment, since RSX is distributed as a COM object. Add the following line to the very beginning of the `CTankApp::InitInstance()` method:

```
// Initialize COM Library
m_coResult = CoInitialize(NULL);
```

The `m_coResult` variable should be declared private, and of type `HRESULT`, in the `CTankApp` class definition.

At the end of the `CTankApp::ExitInstance()` method, add the following to tidy up the COM environment:

```
if (m_coResult == S_OK)
{
    // Tell the COM Library we are all done
    CoUninitialize();
}
```

Since we are using COM, we have to insert "`ole32.lib`" in the 'Object/library modules' field on the 'Project | Settings | Link' dialog. While in the project settings, an adjustment is needed to how precompiled headers are used, to cope with COM declarations. From 'Project | Settings | C++', select Precompiled Headers, and ensure that the "Automatic use of Precompiled Headers" option is selected, and that "stdafx.h" is listed as the "Through header" parameter.

Now add the following line to the list of include files used by the "`TankApp.cpp`" source file

```
#include "audio.h"
```

The line including "stdafx.h" should be preceeded with a definition of the INITGUID symbol. I.e.:

```
#define INITGUID    /* RSX */
#include "stdafx.h"
…
```

And add the following to "`StdAfx.h`", which is the file containing all the predefined headers for the application:

```
#include "rsx.h" // RSX
```

Note: This assumes you have set your compiler up to search the RSX include directory. No change is needed to the library search path since RSX is a COM object, not a library.

Starting the Sound Object

The second stage of initialisation is when the user selections the Start menu to kick off the RealiMation. When this happens, we need to initialise the audio object, and specify sounds to be used with particular objects, using the `CAudio::SetEmitter()` method. This takes a RealiMation instance ID, associates a sound file with it, and gives that pairing an index. It also allows the relative intensity of that sound to be specified. The whole play method, as defined by the `OnPlay` method in the "`TankApp.cpp`" source file now becomes:

```
/*————————————————————————————*/
void CTankApp::OnPlay()
{
    // If playing, then stop. If stopped, then play
    CMainFrame *pFrame = (CMainFrame*) m_pMainWnd;
    ASSERT (pFrame);

    CChannel3D &channel = pFrame->Get3DView();

    if (theAnimations.IsMoving (&channel))
    {
        theAnimations.Stop (&channel);
        theAudio.UnInitialise ();
    }
    else
    {
        theAnimations.Start (&channel);

        // Kick off the sound
        theAudio.Initialise ();

        // Fetch objects from RealiBase and assign sounds to them
        CString fname;

        fname = "d:\\audio\\bird.wav";
        theAudio.SetEmitter (RTGetIDFromName ("Rescue Chopper"), 0, fname, 3.0f);

        fname = "d:\\audio\\tank.wav";
        theAudio.SetEmitter (RTGetIDFromName ("Tank 1"), 1, fname, 1.0f);

        fname = "d:\\audio\\tank.wav";
        theAudio.SetEmitter (RTGetIDFromName ("Tank 2"), 2, fname, 1.0f);
```

```
      // Add a background sound - uncomment if you want it
//       fname = "d:\\audio\\chopper3.mid";
//       theAudio.SetBackground (fname, 0.3f);

   }
}
```

You may need to change the locations of the sound files that you use with the objects. The "`bird.wav`" and "`chopper3.mid`" files are part of the chopper demo that comes with the RSX SDK. Also, if you have changed the names of the relevant objects in the RealiBase, you will need to make corresponding changes in the code.

As can be seen by looking at the full `CAudio` listing at the end of this paper, no RealiMation API calls have been made to set the sound up, with the exception of `RTGetIDFromName()` to get the name of an object for a particular sound, and that call is made by the user of `CAudio`.

There is one part of the `CAudio::SetEmitter()` method that requires further explanation. The lines:

```
rsxModel.fMinBack = 10.0f;
rsxModel.fMinFront = 10.0f;
rsxModel.fMaxBack = 500.0f;
rsxModel.fMaxFront = 500.0f;
```

set the range information for the particular sound. The `fMinFront` and `fMinBack` parameters describe an ellipsoid around the listener where the sound is considered as being all around - i.e. it is so close that there is no meaningful directional content, and it is at full volume. The `fMaxFront fMaxBack` values define the an ellipsoid that is the attenuation region. Within this region, audio is localised, and the intensity decreases logarithmically. Beyond the outer ellipse, the intensity is zero.

For the purposes of this example where we are dealing with a known world (Control.RBS) and known objects, we can fix these values. An enhancement that could be made is to allow these values to be changed based on the size of the 3D world, or other information.

Updating the Sounds

Once we have set up the sounds, attached them to objects in the scene, we need to call the `CAudio::Update()`, passing in the view ID and the current scene time. The best place for this is in the `CChannel3D::DrawView()` method, immediately after the call to `RTSwapPage()`:

```
   // Update audio
theAudio.Update (m_ViewID, RTInqViewTime (m_ViewID));
```

Note: you will need to include "audio.h" in "channel3d.cpp" to enable this to compile.

This is the most important part of putting sound into a RealiMation application, and is really the only part that requires RealiMation API calls. Even though we have not looked at any of the other `CAudio` method implementations yet, it is worth looking at `CAudio::Update()` as it is the code that interfaces the visuals (RealiMation) to the sound. Please refer to the code listing in Appendix A, and find the `CAudio::Update()` function.

The first part of this function simply looks up the camera being use by the view, inquires its position, and sets the listener position to be the same (the `m_lpDL` variable is a pointer to the RSX listener object). The fact that we just utilise the camera in use by the view means that if another part of the application changes the current camera (e.g. `CTankApp::OnNextCamera()`, as activated by the 'C' key), the sound automatically changes aswell.

A listener has an orientation parameter that can affect how sound is heard. In RSX, this orientation is defined by two vectors - one which points forward from the listening position, and another that defines the 'up' direction. These vectors are extracted directly from the current camera transformation matrix, and so do not require any computation at all.

The second part of the function loops through all of the sound emitting objects (i.e. those that have a positional sound source). The code assumes that the centre of the object's bounding box is the source for the sound, and so calculates this position using RealiMation API calls, and passes the result to the RSX emitter object.

Summary

The RealiMation API provides sufficient information to enable easy implementation of 3D positional sound. The example presented here can be extended in a number of ways:

- Add doppler information - RSX understands doppler, and RealiMation objects have speed, so linking the two together should be easy.
- Expose more of the RSX interface, giving the application control over pitch, volume, stopping and starting of sounds.
- Implement reverb control to give the illusion of sounds in enclosed spaces.
- Provide dynamic storage for the number of sounds, rather than limiting to 5.
- Use size information from the RealiBase to set the sound model physical extents.

Readers should keep checking the latest technical papers, which may well include implementing some or all of the above improvements.

Appendix A : CAudio Object Code

This section contains the complete code for the `CAudio` object. Paste it into a new source file called "`audio.cpp`", and add it to your project. The `CAudio` declaration was specified in the section above titled "The Audio Object".

```
/*_____________________________________*/
// TankApp - RealiBase viewer application.
// Copyright © Datapath Ltd. 1998
//
// Audio control class.
// This utility class uses Intel's RSX sound libraries, available on
//   http://developer.intel.com/ial/rsx

#include "stdafx.h"
#include "audio.h"

/*_____________________________________*/
// The one and only CAnimate object

CAudio  theAudio;

/*_____________________________________*/
CAudio::CAudio ()
{
    // Constructor
    m_lpRSX = NULL;
    m_lpCEback = NULL;
    m_lpDL = NULL;

    for (int i = 0; i < MAXEMITTERS; i++)
    {
        m_lpCE [i] = NULL;
        m_idCE [i] = RTNullID;
    }
}

/*_____________________________________*/
CAudio::~CAudio ()
{
    UnInitialise ();
}

/*_____________________________________*/
void  CAudio::Initialise ()
```

```
{
    // Set the Audio Environment
    // Load RSX
    HRESULT rval;
     rval = CoCreateInstance (CLSID_RSX20, NULL, CLSCTX_INPROC_SERVER,
                              IID_IRSX20, (void ** ) &m_lpRSX);
    if (m_lpRSX)
    {
        // Default Environment
        RSXENVIRONMENT rsxEnv;

         rsxEnv.cbSize = sizeof(RSXENVIRONMENT);
         rsxEnv.dwFlags = RSXENVIRONMENT_SPEEDOFSOUND;
        rsxEnv.fSpeedOfSound = 400.0f;
         m_lpRSX->SetEnvironment(&rsxEnv);

        // Create a listener and save the IRSXDirectListener interface
        RSXDIRECTLISTENERDESC rsxDL;             // listener description

         rsxDL.cbSize = sizeof(RSXDIRECTLISTENERDESC);
         rsxDL.hMainWnd = AfxGetMainWnd()->GetSafeHwnd();
        rsxDL.dwUser = 0;
        rsxDL.lpwf = NULL;

          if (SUCCEEDED(m_lpRSX->CreateDirectListener(&rsxDL, &m_lpDL, NULL)))
        {
            RSXVECTOR3D v3d, v3d1;

            RMV3Set (v3d, 0.0f, 0.0f, 0.0f);
             m_lpDL->SetPosition(&v3d);

            RMV3Set (v3d,  0.0f, 0.0f, 1.0f);
            RMV3Set (v3d1, 0.0f, 1.0f, 0.0f);

             m_lpDL->SetOrientation(&v3d, &v3d1);
        }
    }
}

/*_____________________________________*/
void  CAudio::UnInitialise  ()
{
    // Release Emitters
    for (int i = 0; i < MAXEMITTERS; i++)
    {
        if (m_lpCE [i] != NULL)
        {
            m_lpCE [i]->Release();
           m_lpCE [i] = NULL;
           m_idCE [i] = RTNullID;
        }
    }

    // Release background sound
    if (m_lpCEback)
    {
        m_lpCEback->Release();
       m_lpCEback = NULL;
    }

    // Release the listener
    if (m_lpDL)
    {
        m_lpDL->Release();
       m_lpDL = NULL;
    }

    // Release RSX
    if (m_lpRSX)
    {
        m_lpRSX->Release();
       m_lpRSX = NULL;
    }
}
```

```
/*_____________________________________*/
void CAudio::Update (rtID view, Ord time)
{
    // Update listener and emitter positions. The listener position is taken
    // from the camera used by the specified view.
    // The emitter positions are calculated at the specified time value, which
    // allows for instances on paths.
    RSXVECTOR3D pt;

    if (m_lpDL)
    {
        rtCamera pos;
        rtID cam;

        cam = RTInqObject (view, RTCameraID);
        ASSERT (cam != RTNullID);

        RTInqCamera (cam, &pos, NULL);

        pt.x = pos.position.pos.x;
        pt.y = pos.position.pos.y;
        pt.z = pos.position.pos.z;

        m_lpDL->SetPosition (&pt);

        RSXVECTOR3D v3d, v3d1;
        rtMatrix4 WToE;

        RTInqCameraTransforms (cam, &WToE, NULL);

        RMV3Set (v3d,  WToE [2][0], WToE [2][1], WToE [2][2]);
        RMV3Set (v3d1, WToE [1][0], WToE [1][1], WToE [1][2]);
        m_lpDL->SetOrientation (&v3d, &v3d1);
    }

    // Update all emitters
    for (int i = 0; i < MAXEMITTERS; i++)
    {
        if (m_lpCE [i] != NULL)
        {
            // Update emitter position
            rtBox3d box;
            rtPoint3 p;

            ASSERT (m_idCE [i] != RTNullID);

            RTInqObjectBox (m_idCE [i], time, &box);
            RMBoxMidPoint (&box, &p);
            pt.x = p.x;
            pt.y = p.y;
            pt.z = p.z;
            m_lpCE [i]->SetPosition(&pt);
        }
    }
}

/*_____________________________________*/
void CAudio::SetEmitter (rtID id, int index, CString &filename, Ord intensity)
{
    // Sets sound file for object 'id', and assigns it an index, which is
    // its position in the array of objects.
    // The intensity paramater specifies a relative intensity for the sound.
    // A value of 1 means "normal", and higher values make that sound louder
    // in comparison to others, given the same position in space.
    // As soon as the sound file is associated, it starts playing.
    ASSERT (index >= 0);
    ASSERT (index < MAXEMITTERS);

    if (id != RTNullID)
    {
        RSXCACHEDEMITTERDESC rsxCE;
        RSXEMITTERMODEL rsxModel;
        RSXVECTOR3D p;

        ZeroMemory(&rsxCE, sizeof(RSXCACHEDEMITTERDESC));
```

```
        rsxCE.cbSize = sizeof(RSXCACHEDEMITTERDESC);
        rsxCE.dwFlags = RSXEMITTERDESC_NODOPPLER;
       rsxCE.dwUser = 0;

        rsxModel.cbSize = sizeof(RSXEMITTERMODEL);
        strcpy (rsxCE.szFilename, filename);

       rsxModel.fIntensity = intensity;
       rsxModel.fMinBack = 10.0f;
       rsxModel.fMinFront = 10.0f;
       rsxModel.fMaxBack = 500.0f;
       rsxModel.fMaxFront = 500.0f;

        if (SUCCEEDED (m_lpRSX->CreateCachedEmitter (&rsxCE, &m_lpCE[index], NULL)) &&
(m_lpCE))
      {
          m_lpCE [index]->SetPosition(&p);

         p.x = 0.0f;
         p.y = 0.0f;
         p.z = 1.0f;

          m_lpCE [index]->SetOrientation (&p);
          m_lpCE [index]->SetModel (&rsxModel);
          m_lpCE [index]->ControlMedia (RSX_PLAY, 0, 0.0f);
         m_idCE [index] = id;
      }
      else
      {
          m_lpCE [index] = NULL;
          m_idCE [index] = RTNullID;
      }
   }
}

/*_____________________________________*/
void CAudio::SetBackground (CString &filename, Ord intensity)
{
    // Create and play background sound - one with no 3D positional
    // characteristics at all.

    // try creating background sound
    RSXCACHEDEMITTERDESC    rsxCE;
     ZeroMemory(&rsxCE, sizeof(RSXCACHEDEMITTERDESC));
    rsxCE.cbSize = sizeof(RSXCACHEDEMITTERDESC);
    rsxCE.dwFlags = RSXEMITTERDESC_NOSPATIALIZE | RSXEMITTERDESC_NOATTENUATE |
RSXEMITTERDESC_NODOPPLER | RSXEMITTERDESC_NOREVERB;
   rsxCE.dwUser = 0;

    strcpy (rsxCE.szFilename, filename);
    RSXEMITTERMODEL rsxModel;
    rsxModel.cbSize = sizeof(RSXEMITTERMODEL);
    rsxModel.fIntensity = intensity;
    rsxModel.fMinBack = 2.0f;
    rsxModel.fMinFront = 2.0f;
    rsxModel.fMaxBack = 150.0f;
    rsxModel.fMaxFront = 500.0f;

   m_lpCEback = NULL;
    if( FAILED(m_lpRSX->CreateCachedEmitter(&rsxCE, &m_lpCEback, NULL)))
   {
       TRACE ("Error creating background sound");
   }

    if(m_lpCEback)
   {
       m_lpCEback->SetModel (&rsxModel);
          m_lpCEback->ControlMedia (RSX_PLAY,0,0.0f);
   }
}
/*_____________________________________*/
```