.NET and Multimedia - Introduction of managed DirectX

Andreas Lennartz
University of Applied Sciences, Fulda
alenn@gmx.net

Abstract

DirectX is a multimedia API that provides a standard interface to interact with graphics and sound cards, input devices and more. DirectX abstracts code from the specific hardware and translates it to a common set of instructions into hardware specific commands. The actual implementation of DirectX contains an integration in managed Code, which combines the advantages of both world, and is called managed DirectX.

1 Introduction

Currently there are two vendors offering APIs for developing 3D-Applications available. On one hand Microsoft provides its DirectX-API and on the other hand there is the open source alternative OpenGL.

OpenGL is an environment for developing portable, interactive 2D and 3D graphics applications. Introduced in 1992, OpenGL is one of the industry's first choice for 2D and 3D graphics APIs (**?**). Because of its standarisation it is platform independent. With OpenGL, rapid application development with a broad set of rendering, texture mapping, special effects and other powerful visualization functions can be archieved.

DirectX first appeared in 1995 and was then called the "GameSDK". Originally it was targeted at developers using C and C++. With the release of the first managed version (9.0) of the API in December 2002 it is possible to use C# or VB.NET with DirectX (Actually any CLR compliant language can optionally be used) (**?**).

DirectX allow it to make non-hardware-specific function calls. For this purpose, DirectX is accountible for the execution on the hardware. If the function is supported from the hardware, the hardware acceleration makes the API functions work fast. If not, DirectX has software emulation that takes over. It will be slower, but in spite of that the same low-level API can be used. Developer can focus on the application development itself. The layer of DirectX that takes care of hardware-supported features is called the hardware acceleration layer (HAL), and the software emulation layer that kicks in if the hardware doesn't support a feature is called the hardware emulation layer (HEL).

When writing applications with managed DirectX or unmanaged DirectX, managed DirectX has the prejudice that it is slower than unmanaged. But this is only true in certain circumstances. So far a lot of games were created with managed code or using a mix of managed and unmanaged code. Writing in managed code makes the developers more productive and thus write more code in less time and produces safer code.

2 Managed Code

Managed code is code that has its execution managed by the .NET Framework Common Language Runtime (CLR). It refers to a contract of cooperation between natively executing code and the runtime. In this contract it is specified that at any point of execution, the runtime may stop an executing CPU. Then it may retrieve information specific to the cur-

rent CPU instruction address. Information that must be queryable generally belongs to runtime state, such as register or stack memory contents.

The necessary information is encoded in an Intermediate Language (IL) and associated metadata. The symbolic information that describes all of the entry points and the constructs is exposed in the IL (e.g., methods, properties) and their characteristics. With the Common Language Infrastructure (CLI) Standard, a description is given how the information has to be encoded. Programming languages that target the runtime will build the correct encoding. Therefore, the developer only has to know that any of the languages that target the runtime produce managed code. This code contains encoded files that contain IL and metadata. There are a lot of languages to choose from, because there are over 20 different languages provided by third parties in addition to the standard languages from Microsoft. (C#, J#, VB .Net, Jscript .Net, and C++).

Before execution of the code, the Virtual machine compiles the IL into native executable code. While the compilation happens by the managed execution environment, the managed execution environment can make guarantees about what the code is going to do. The enviroment can insert traps and appropriate garbage collection hooks, exception handling, type safety, array bounds and index checking, etc. E.g, such a compiler makes sure to lay out stack frames and everything just right so that the garbage collector can run in the background on a separate thread, constantly walking the active call stack, finding all the roots, chasing down all the live objects. In addition because the IL has a notion of type safety the execution engine will maintain the guarantee of type safety eliminating a whole class of programming mistakes that often lead to security holes.

"'Contrast this to the unmanaged world: Unmanaged executable files are basically a binary image, x86 code, loaded into memory. The program counter gets put there and that's the last the OS knows. There are protections in place around memory management and port I/O and so forth, but the system doesn't actually know what the application is doing. Therefore, it can't make any guarantees about what happens when the application runs."' see (**?**)

3 DirectX components

DirectX 9.0 for Managed Code is made up of the following major components (The deprecated parts of DirectX will not be included in the further discussion):

- Direct3D Graphics provides a single API that can be used for 3D graphics programming.
- DirectInput provides support for a variety of input devices, including full support for force-feedback technology.
- DirectSound provides support for playing and capturing prerecorded digital samples.
- AudioVideoPlayback allows for playback and simple control of audio and video media.

3.1 Direct 3D Graphics

Direct3D Graphics is for device independent access to the 3D-graphic hardware and acceleration. If no hardware acceleration is supported, the software renderer will emulate the hardware. (**?**)

Complicated 3D-figures and objects are usually modeled with a 3D-software and then saved to file. The .x file format is an example for such a file. (**?**),(**?**) Microsoft Direct3D can create objects from these files using meshes. A mesh is an abstract data container with the data for a complex model. It contains resources such as textures and materials, and attributes such as position data and adjacency data. Meshes itselves are somewhat complicated, but accessing and using them with Direct3D makes it easier. In the following an example is shown how to load, render, and unload a mesh. With this example an overview will be given over the functionality of Direct3D.

First of all, the mesh object has to be initialized. Also we need a device for the output. There are two device types in DirectX: a hardware device and a reference device. The hardware device is the primary device which supports hardware-accelerated rasterization as well as hardware vertex processing. For this purpose the display adapter of the computer has to support Direct3D. The hardware device then implements all or part of the transformations, lighting, and rasterization in hardware. The application itself does not access the hardware device directly - it calls Direct3D functions and methods which access the hardware through a hardware abstraction layer (HAL). If the hardware supports Direct3D, these functions and methods are executed with best performance of the hardware. Direct3D supports additional a device called reference device. This reference device emulates all Direct3D functions and methods via software without any hardware acceleration. Furthermore, if Direct3D functionalities are used which are not implemented in the hardware, the reference device will then emulate the corresponding part of the hardware device.

To create a new Direct3D device, the following command has to be executed, e.g. in a Windows Form Class:

```
device = new Device(Manager.Adapters.Default
.Adapter, DeviceType.Hardware,this,CreateFlags.
SoftwareVertexProcessing,new
PresentParameters())
```

For creating a new device these parameters are needed:

1. A graphics adapter to connect to the output device, e.g. to connect one monitor to the graphics card.
2. A parameter which decides whether a hardware of software rendering is done.
3. The class (e.g. the windows form) which connects Direct3D with the application.
4. Some flags for setting the behavior of the rendering pipeline.
5. Parameters for the presentation behavior.

Furthermore in the application defined *OnResetDevice* method it is possible to initialize an ExtendedMaterial object as an array that is used to capture the mesh file data to a Material structure. In the method the device can be adjusted, for e.g. turning on the z-buffer and white ambient lighting. This function is called when the device is initialized and in some different cases reseted.

Now the mesh object has to be loaded. As shown in the following code fragment as a part of the *OnResetDevice* method a mesh is loaded that represents a texture-mapped 3D tiger from a tiger.x file. In this *Mesh.FromFile* method call, the SystemMemory constant of the MeshFlags enumeration indicates that the mesh is to be loaded into system RAM not typically accessible by the device.

```
mesh = Mesh.FromFile("tiger.x", MeshFlags.
SystemMemory, device, out extendedMaterials);
```

After the mesh is loaded, a meshMaterials Material object *extendedMaterials* has its members filled with the Material structures of the materials object that came from the mesh file. These materials could now be set inside the *OnResetDevice* method to a specific ambient color or could be set to a MeshTexture object loaded from a texture of a file. A texture object file could be loaded with the *TextureLoader.FromFile* method. For example, it is possible to set for all extendedMaterials of a mesh the ambient material color. This could take place by an iteration over the length of the materials from the mesh. Afterwards the texture had to be loaded. The following code fragment (with i as the iteration counter) shows this:

```
meshMaterials[i].Ambient = meshMaterials[i].
Diffuse;
meshTextures[i] = TextureLoader.FromFile
(device, extendedMaterials[i].
TextureFilename)
```

Finally, after the mesh has been loaded and initialized, the next step is to render it. For this purpose we have to call the private *Render* method. This method is called every time an object needs to be rendered. The function is accountable for making the Mesh object looking as desired. Inside the render method,

the rendering pipeline can be adjusted. To render the mesh itself, it is divided into subsets, one for each material that was loaded. A loop needs to be run for rendering each material subset. This loop can for example be used to make the following operations on the object:

- Setting the material property
- Draw the material subset with the *mesh.DrawSubset()* method

With the commands *device.EndScene()* and *device.Present()* the end of the rendering pipeline is indicated and the object will get visible.

Now, it is realized to load, render and present a mesh with a Direct3D device. If the graphic device supports Direct3D, the benefits of hardware acceleration would be used. For a deeper look at this example and its code there are more information in (**?**)

3.2 Direct Input

With Microsoft DirectInput it is possible to establish a direct communication between the hardware drivers of an input device. Normally the communication would be provided over a Windows message relay supported by the Microsoft Win32 API. With DirectInput the application retrieves the data directly from the device, even when the application is inactive. Furthermore, with DirectInput all kind of input devices with DirectInput drivers are supported, as well as functionalities for Force Feedback. With Force Feedback it is allowed to send messages and properties to a user's DirectInput device to relay feedback in the form of physical force.

DirectInput implements the technology of action mapping. Action mapping applications can retrieve input data without knowing what kind of device is being used to generate it. DirectInput does not provide any special support for keyboard devices or mouse navigation. Action mapping provides a possibility to establish a connection between input actions and input devices. This connection does not depend on the existence of particular device objects (e.g. specific buttons). Action mapping simplifies the input loop and therefore reduces the need for custom game drivers and custom configuration user interfaces in games. (**?**)

To capture the device input using Microsoft DirectInput, it is necessary to know from which device the input has to be captured. (And, of course, if it is currently attached to the user's computer). To do this, there is a enumeration of DeviceInstance objects which can be looped. This Enumeration serves the following purposes:

- Reports what DirectInput devices are available.
- Supplies a globally unique identifier (GUID) for each DirectInput device.
- Possibility to create a DeviceInstance object for each DirectInput device as it is enumerated. With this the type of device and its capabilities can be checked.

An example code fragment for this loop would be

```
foreach(DeviceInstance di in Manager.Devices)
  string name = di.InstanceName;
```

Capturing the device input itself with DirectInput can be done in three parts. First a device object that represents the input device needs to be created. After the creation of the device object the configuration has to be done. After the configuration the device state can be checked within the application. There will be callback functions registered which are called after the corresponding action from the user (e.g. pressing a button.)

An example for the configuration of a joystick device could be:

```
foreach(DeviceObjectInstance doi in
joystick.Objects)
joystick.Properties.SetRange(
ParameterHow.ById, doi.ObjectId,
new InputRange(-5000,5000));
```

Furthermore, in the callback function *UpdateJoystick()* which is registered for joystick events, the state of the buttons could be requested with this command:

```
byte[] buttons = joystick.CurrentJoystickState.
GetButtons;
```

As seen in the examples, DirectInput provides an easy way to access an input device. The benefit of DirectInput devices is the direct communication between hardware and application, as long as the devices have corresponding hardware drivers.

3.3 Direct Sound

Microsoft DirectSound provides a system to capture sounds from input devices and play sounds through various playback devices. The input devices are realized in several secondary software or hardware sound buffers. Software buffers keep their data always in system memory and are mixed by the CPU. Hardware buffers can keep their data either in system memory or, if the application requests it and resources are available, in onboard memory and are mixed by the sound card processor. Each of them contains a static single sound or a dynamic stream of audio. When sounds in secondary buffers are played, DirectSound mixes them in the so called primary buffer and sends them to the output device. Only the available processing time limits the number of secondary buffers that DirectSound can mix. The primary buffer is always a hardware buffer on the soundcard, which means there is only one available and its memory size is limited. So the primary buffer holds the audio that the listener will hear. Unlike the primary buffer there exists several settings for the secondary sound buffer that can be changed. When creating a new secondary buffer the most important command line is:

```
sound = new SecondaryBuffer (filePath,
bufferDescription, SoundDevice)
```

As shown each secondary buffer has a buffer description which contains one or more control properties (like pan, volume, frequency, effects, 3D) from the BufferCaps structure (**?**). To allow the control of these properties the corresponding flags just have to be enabled. According to those settings different operations can be executed by simply typing in the name of the secondary buffer followed by a dot and the operation name. For example *sound.Play()* starts playing the sound or *sound.Format.** gives back additional sound information like the number of channels or samples per second.

When a secondary buffer is created with enabled flag for the control3D property, DirectSound offers the possibility to set an algorithm to simulate 3D spatial location of a sound. By default, no head-related transfer function (HRTF) processing is performed, and the location of the sound relative to the listener is indicated by panning and volume only. To optimize the 3D effects for the user's system DirectSound also uses the speaker configuration. The SpeakerConfig property from the sound device allows users to set up the sound system to the given speaker system by switching the flag to true, for example mono, stereo, 5.1 or 7.1 surround sound.

One of the most important features of DirectSound is the possibility to apply real-time audio effects to the sounds in playback. All the effects are applied in hardware (that is, implemented by the sound card driver) whenever possible, otherwise emulated via software. The latter causes an important drop in the performance. There are 9 effects existing: Chorus, Compressor, Distortion, Echo, Flangler, Gargle, Interactive3DLevelReverb, ParamEqualizer and WavesReverb. Each of them has its own different settings ((**?**)). To implement an effect on a buffer first the effect flag in the buffer description has to be set to true and then the *SecondaryBuffer.SetEffects* method has to be used. This method takes an array of EffectDescription structures that describe the effects. An effect can only be applied if the sound is not played. An example and a great summary of the Direct Sound functionalities gives *The Ultimate Managed DirectSound 9 Tutorial*[1]. It creates a small application which demonstrates all DirectSound elements that have been described before. A big disadvantage of Microsoft DirectSound is that it plays only wave and Pulse Code Modulation (PCM) files. Waveform au-

[1] http://www.codeproject.com/cs/media/DirectSound9p1.asp

dio data consists of digital samples of the sound at a fixed sampling rate, whereas PCM the representation of an analog signal by a sequence of numbers means. Managed DirectSound does neither support compressed WAV formats nor audio formats with more than two channels. Therefore AudioVideoPlayback is the better choice for playback other formats like mp3 for example.

3.4 Audio Video Playback

The AudioVideoPlayback application programming interface (API) provides like the older DirectX version the playback and simple control of audio and video files. But Microsoft has highly simplified the usage. All functionalities for the playback are packed into a single dll-file. Thus the integration of a video into an application became much easier, because the user only needs to refer to the Microsoft.DirectX.AudioVideoPlayback namespace. The command:

```
Video ourVideo = new Video("C:\\Example.avi")
```

creates an instance of the video class and provides the possibility to control the playback. For instance the method *ourVideo.Stop()* stops the playback and *ourVideo.Play()* plays it. The size of the playback window can be set by *ourVideo.Size = new Size(480, 320)*. If the video file contains audio, the Video.Audio property returns an Audio object that can be used to change the volume or the stereo balance for the audio, for example *Video.Audio.Volume = 100*.

```
Audio ourAudio = new Audio("C:\AudioFile.mp3")
```

The Audio object is similar to the Video object, but it supports only properties that relate to audio, such as Volume and Balance. That means the Audio class is primarily designed for very simple playback scenarios, or for use with the Video class. Whenever a greater control over the audio playback is needed, it is better to use Microsoft DirectSound to play audio files.

The managed DirectX AudioVideoPlayback API doesn't support video converting. The only way to do this is using the old unmanaged C++ code, where several filtergraphs have to be combined. For further information on that topic see (**?**).

4 DirectX versus OpenGL

OpenGL is an open standard, so everyone can use it for free. The fact that it is a standard for graphics cards makes it running on every operating system (as long as the OS considers a graphic user interface). Compared to that, DirectX is an API developed by Microsoft and consists of several subsystems like shown in the previous chapters. So only the DirectX graphic component can be compared with OpenGL. Considering the DirectX sound part, the analogue would be OpenAL, which is an open audio standard. As already mentioned, DirectX is a Microsoft product, so it can only be found on Microsoft operating systems where it is preferred to OpenGL. Nevertheless Microsoft cannot ignore OpenGL, because it is an industrial standard. So the graphics card manufaturers support their customers with the latest OpenGL driver versions and makes OpenGL usable under Windows.

Considering OpenGL and DirectX both are quite similar and offer equal functionalities. Anyway there still exist some differences which are illustrated in table 1 on the following page.

5 Conclusion

With Managed DirectX there exists a structured and easy to use low-level API. While in C++ the developer took great pain to simplify and integrate unmangaged C++ classes, with the .NET style and the design of namespaces, classes and properties, programming Managed DirectX is very comfortable for the developer. Compared to OpenGL, DirectX has some very important advantages for game developers. With Managed DirectX, the developers also write more productive and therefore more code in less time and produce safer code.

After a short introduction in Managed DirectSound, DirectInput, Direct3D and AudioVideoPlayback APIs the reader should for himself recognize the obviously advantages of

Features	**OpenGL**	**DirectX**
object oriented	no	yes
operating system	many	only all Windows versions
useable drivers available for	high end graphics cards	nearly all graphics cards
driver quality	mostly bad	often better than OpenGL driver
usage	university, research and development, CAD	game industry
documentations, tutorials, samples	many	less than for OpenGL
version release	every 5 years	every 15 months
trademark of	Silicon Graphics Inc.	Microsoft

Table 1: The list compares the key features between OpenGL and DirectX

managed DirectX. With regard to the coming Windows Vista where DirectX is an elemental component, DirectX will play a more important role even for application designers in every part of future application development.