4. multimedia platforms

with DirectX 9 digital convergence becomes a reality

learning objectives

After reading this chapter you should be able to characterize the functionality of current multimedia platforms, to describe the capabilities of GPUs, to mention the components of the Microsoft DirectX 9 SDK, and to discuss how to integrate 3D and video.

Almost 15 years ago I bought my first multimedia PC, with Windows 3.1 Media Edition. This setup included a video capture card and a 4K baud modem. It was, if I remember well, a 100 Mhz machine, with 16 Mb memory and a 100 Mb disk. At that time, expensive as it was, the best I could afford. Some 4 years later, I acquired a Sun Sparc 1 multimedia workstation, with a video capture card and 3D hardware accelerator. It allowed for programming OpenGL in C++ with the GNU gcc compiler, and I could do live video texture mapping at a frame rate of about one per second. If you consider what is common nowadays, a 3Ghz machine with powerful GPU, 1 Gb of memory, a 1.5Mb cable or ADSL connection and over 100 Gb of disk space, you realize what progress has been made over the last 10 years.

In this chapter, we will look in more detail at the capability of current multimedia platforms, and we will explore the functionality of the Microsoft DirectX 0 platform. In the final section of this chapter, I will then report about the work I did with the DirectX 9 SDK to implement the ViP system, a presentation system that merges video and 3D.

1

4.1 developments in hardware and software

Following Moore's law (predicting the doubling of computing power every eighteen months), computer hardware has significantly improved. But perhaps more

spectacular is the growth in computing power of dedicated multimedia hardware, in particular what is nowadays called the GPU (graphics processing unit). In Fernando and Kilgard (2003), he NVIDIA GeForce FX GPU is said to have 125 million of transistors, whereas the Intel 2.4GHz Pentium 4 contains only 55 million of transistors. Now, given the fact that the CPU (central processing unit) is a general purpose, or as some may like to call it, *universal* device, why is it necessary or desirable to have such specialized hardware, GPUs for graphics and, to be complete DSPs (digital signal processors) for audio?

a little bit of history

Almost evryone knows the stunning animation and effects in movies made possible by computer graphics, as for example the latest production of Pixar, *The Incredibles*. Such animation and effects are only possible by offline rendering, using factories of thousands of CPUs, crunching day and night to render all the frames.

At the basis of rendering lies traditional computer graphics technology. That is, the transformation of vertices (points in 3D space), rasterization (that is determining the pixel locations and pixel properties corresponding to the vertices), and finally the so-called raster operations (determining whether and how the pixels are written to the framebuffer). OpenGL, developed by SGI was the first commonly available software API (application programmers interface) to control the process of rendering. Later, Microsoft introduced Direct3D as an alternative for game programming on the PC platform.

The process outlined above is called the *graphics pipeline*. You put models, that is collections of vertices, in and you get (frames of) pixels out. This is indeed a simplification in that it does not explain how, for example, animation and lighting effects are obtained. To gain control over the computation done in the graphics pipeline, Pixar developed Renderman, which allows for specifying transformations on the models (vertices) as well as operations on the pixels (or fragments as they are called in Fernando and Kilgard (2003)) in a high level language. As vertex operations you may think of for example distortions of shape due to a force such as an explosion. As pixel operations, the coloring of pixels using textures (images) or special lighting and material properties. The languages for specifying such vertex or pixel operations are collectively called *shader* languages. Using offline rendering, almost anything is possible, as long as you specify it mathematically in a computationally feasible way.

The breakthrough in computer graphics hardware was to make such shading languages available for real-time computer graphics, in a way that allows, as Fernando and Kilgard (2003) phrase it, 3D game and application programmers and real-time 3D artists to use it in an effective way.

Leading to the programmable computer graphics hardware that we know today, Fernando and Kilgard (2003) distinguish between four generations of 3D accellerators.[1]

[1] The phrase GPU was introduced by NVIDIA to indicate that the capabilities of the GPU far exceed those of the VGA (video graphics array) originally introduced by IBM, which is

4 generations of GPU

- Before the introduction of the GPU, there only existed very expensive specialized hardware such as the machines from SGI.
- The first generation of GPU, including NVIDIA TNT2, ATI Rage and 3dfx Voodoo3, only supported rasterizing pre-transformed triangles and some limited texture operations.
- The second generation of GPUs, which were introduced around 1999, included the NVIDIA GeForce 2 and ATI Radeon 7500. They allowed for both 3D vertex transformations and some lighting, conformant with OpenGL and DirectX 7.
- The tird generation GPUs, including NVIDIA GeForce 3, Microsoft Xbox and ATI Radeon 8500, included both powerful vertex processing capabilities and some pixel-based configuration operations, exceeding those of OpenGL and DirectX 7.
- Finally, the fourth generation of GPUs, such as the NVIDIA GeForce FX and ATI Radeon 9700, allow for both complex vertex and pixel operations.

The capabilities of these latter generations GPUs motivated the development of high level shader languages, such as NVIDIA Cg and Microsoft HLSL. High level dedicated graphics hardware programming languages to control what may be called the programmable graphics pipeline.

the (programmable) graphics pipeline

Before discussing shading languages any further, let's look in some more detail at the graphics pipeline. But before that you must have an intuitive grasp of what is involved in rendering a scene.

Just imagine that you have created a model, say a teapot, in your favorite tool, for example Maya or 3D Studio Max. Such a model may be regarded as consisting of polygons, let's say triangles, and each vertex (point) of these triangles has apart from its position in (local) 3D space also a color. To render this model it must first be positioned in your scene, using so-called world coordinates. The *world transformation* is used to do this. The world transformation may change the position, rotation and scale of your object/model. Since your scene is looked at from one particular point of view we need to apply also a so-called *view transformation*, and to define how our view will be projected on a 2D plane, we must specify a *projection transformation*. Without going into the mathematical details, we may observe that these transformations can be expressed as 4x4 matrices and be combined in a single matrix, often referred to as the *world view projection matrix*, that can be applied to each of the vertices of our model. This combined transformation is the first stage in the process of rendering:

graphics pipeline

1. vertex transformation – apply world, view, projection transforms
2. assembly and rasterization – combine, clip and determine pixel locations
3. fragment texturing and coloring – determine pixel colors
4. raster operations – update pixel values

nothing more than a dumb framebuffer, requiring updates from the CPU.

The second phase, roughly, consists of cleaning up the collection of (transformed) vertices and determining the pixel locations that correspond to the model. Then, in the third phase, using interpolation or some more advanced method, coloring and lighting is applied, and finally a sequence of per-fragment or pixel operations is applied. Both OpenGL and Direct3D support among others an alpha (or transparency) test, a depth test and blending. The above characterized the fixed function graphics pipeline. Both the OpenGL and Direct3D API support the fixed function pipeline, offering many ways to set relevant parameters for, for example, applying lights, depth and texturing operations.

To understand what the programmable graphics pipeline can do for you, you would best look at some simple shader programs. In essence, the programmable pipeline allows you to perform arbitrary vertex operations and (almost) arbitrary pixel operations. For example, you can apply a time dependent morphing operation to your model. Or you can apply an amplification to the colors of your scene. But perhaps more interestingly, you can also apply an advanced lighting model to increase realism.

A simple morphing shader in ViP, see section 4.3.

2

a simple shader

When I began with programming shaders myself, I started with looking at examples from the DirectX SDK. Usually these examples were quite complex, and my attempt at modifying them often failed. Being raised in theoretical computer science, I changed strategy and developed my first shader program called *id*, which did nothing. Well, it just acted as the identity function. Then later I used this program as a starting point for writing more complex shader programs.

The *id* shader program is written in the DirectX 9 HLSL (high level shader language), and makes use of the DirectX Effects framework, which allows for

specifying multiple vertex and pixel shaders, as well as multiple techniques and multiple passes in a single file.

The program starts with a declaration, specifying the global names for respectively the texture and the world/view/projection matrix. Also a texture sampler is declared, of which the function will become clear later.

HLSL declarations

```
texture tex;
float4x4 wvp;       // World * View * Projection matrix

sampler tex_sampler = sampler_state
{
   texture = /<tex/>;
};
```

It then defines, respectively, the vertex shader input and output, as structures. This declaration follows the standard C-like syntax for specifying elements in a structure, except for the identifiers in capitals, which indicate the semantics of the fields, corresponding to pre-defined registers in the GPU data flow.

vertex shader data flow

```
struct vsinput {
   float4 position : POSITION;
   float3 normal : NORMAL;
   float2 uv : TEXCOORD0;
};
struct vsoutput {
   float4 position   : POSITION;   // vertex position
   float4 color    : COLOR0;     // vertex diffuse color
   float2 uv  : TEXCOORD0;  // vertex texture coords
};
```

When the *vs_id* function, given below, is called, the input arguments are filled by the registers corresponding to the semantics pf the input structure. Similarly, the output results in setting the registers corresponding to the semantics of the output structure.

vertex shader

```
vsoutput vs_id( vsinput vx ) {
   vsoutput vs;

   vs.position = mul(vx.position, wvp);
   vs.color = color;
   vs.uv = vx.uv;

   return vs;
}
```

The *vs_id* function does exactly what the fixed graphics pipeline would do when transforming vertices. It applies the transformation to the vertex and passes both color and texture sampling coordinates to the pixel shader.

The pixel shader has a single color as output, which is obtained by sampling the texture, using the (interpolated) vertex color to modify the result.

pixel shader

```
struct psoutput
{
    float4 color : COLOR0;
};


psoutput ps_id( vsoutput vs )
{
    psoutput ps;

    ps.color = tex2D(tex_sampler, vs.uv) * vs.color;

    return ps;
}
```

Note that the *tex_sampler* comes from the global declaration above. The function *text2D* is a built-in for obtaining a color value from the sampler.

Finally, for each technique and each pass within a technique, in our case one technique with one pass, it must be indicated which function must be used for respectively the vertex shader and the pixel shader.

technique selection

```
technique render_id
{
    pass P0
    {
        VertexShader = compile vs_1_1 vs_id();
        PixelShader  = compile ps_2_0 ps_id();
    }
}
```

To make actual use of this program, the effect must be invoked from a DirectX or OpenGL program using the interface offered by the API.

Examples of Impasto, see examples – impasto

3

a morphing shader Slightly more complex is an example adapted from the morphing shader that can be found in ATI's Rendermonkey. To make a shader that morphs a cube into a ball and back, you must manipulate the vertices and the normals of the cube. For this to work your cube must have sufficient vertices, which is a property you can set in the tool that you use to make a cube.

morphing (vertex) shader

```
float3 spherePos = normalize(vx.position.xyz);
float3 cubePos = 0.9 * vx.position.xyz;

float t = frac(speed * time);
t = smoothstep(0, 0.5, t) - smoothstep(0.5, 1, t);

// find the interpolation factor
float lrp = lerpMin + (lerpMax - lerpMin) * t;

// linearly interpolate the position and normal
vx.position.xyz = lerp(spherePos, cubePos, lrp);
vx.normal = lerp(sphereNormal, cubeNormal, lrp);

// apply the transformations
vs.position = mul(wvp, vx.position);
```

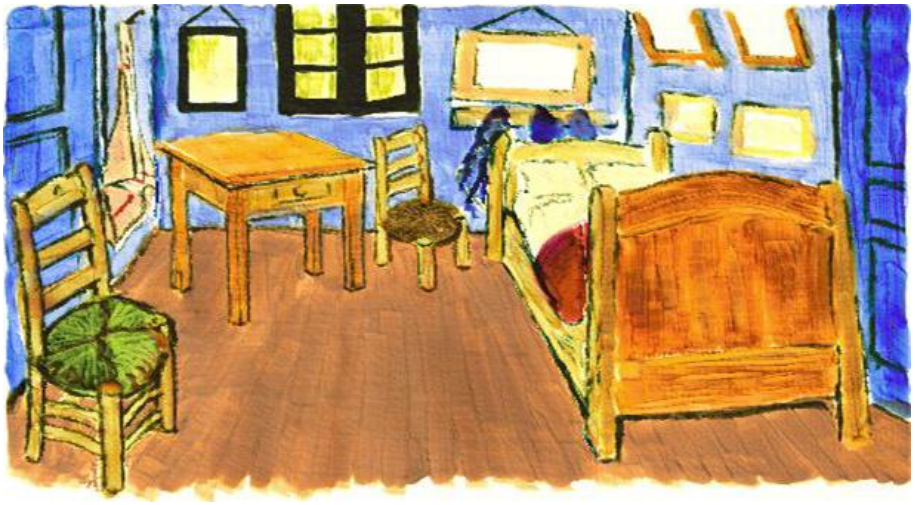
The example uses the built-in function *lerp*, that performs linear interpolation between two values using an interpolation factor between 0 and 1.

color amplification As an example of a pixel shader, look at the fragment defining an amplification of coloring below. It merely amplifies the RGB components of the colors when this exceeds a certain treshold.

coloring (pixel) shader

```
float4 x = tex2D(tex_sampler, vs.uv);
if (x.r > x.g && x.r > x.b) { x.r *= xi; x.g *= xd; x.b *= xd; }
else  if (x.g > x.r && x.g > x.b) { x.g *= xi; x.r *= xd; x.b *= xd; }
else  if (x.b > x.r && x.b > x.g) { x.b *= xi; x.r *= xd; x.g *= xd; }
ps.color = x;
```

When you develop shaders you must keep in mind that a pixel shader is generally invoked far more often than a vertex shader. For example a cube can be defined using 12 triangles of each tree vertices. However, the number of pixels generated by this might be up to a million. Therefore any complex computation in the pixel shader will be immediately noticable in the performance. For example, a slightly more complex pixel shader than the one above makes my NVIDIA GeForce FX 5200 accelerated 3 GHz machine drop to 5 frames per second!



rendering of van Gogh painting with Impasto

4

example(s) – *impasto*

IMPaSTo[2] is a realistic, interactive model for paint. It allows the user to create images in the style of famous painters as in the example above, which is after a painting of van Gogh. The *impasto* system implements a physical model of paint to simulate the effect of acrylic or oilpaint, using Cg shaders for real-time rendering, Baxter et al. (2004).

research directions – *the art of shader programming*

At first sight, shader programming seems to be an esoteric endeavor. However, as already indicated in this section, there are a number of high level languages for shader programming, including NVIDIA Cg and Microsoft HLSL. Cg is a platform independent language, suitable for both OpenGL and Direct3D. However, counter to what you might expect also Microsoft HLSL can be used for the OpenGL platform when you choose the proper runtime support.

To support the development of shaders there are, apart from a number of books, some powerful tools to write and test your shaders, in particular the already mentioned ATI Rendermonkey tool, the CgFx tool, which both produce HLSL code, as well as the Cg viewer and the effect tool that comes with the Microsoft DirectX 9 SDK.

Although I am only a beginning shader programmer myself, I find it truly amazing what shaders can do. For a good introducion I advice Fernando and Kilgard (2003). Futher you may consult Engel (2004a), Engel (2004b) and Engel (2005). Written from an artist's perspective is St-Laurent (2004).

[2] gamma.cs.unc.edu/IMPaSTo

Stacks and stacks of books on DirectX

5

4.2 DirectX 9 SDK

Many of the multimedia applications that you run on your PC, to play games, watch video, or browse through your photos, require some version of Direct X to be installed. The most widely distributed version of Direct X is 7.0. The latest version is the october release of 2004. This is version 9c. What is DirectX? And, more importantly, what can you do with it? In the DirectX documentation that comes with the SDK, we may read:

DirectX

> Microsoft DirectX is an advanced suite of multimedia application programming interfaces (APIs) built into Microsoft Windows; operating systems. DirectX provides a standard development platform for Windows-based PCs by enabling software developers to access specialized hardware features without having to write hardware-specific code. This technology was first introduced in 1995 and is a recognized standard for multimedia application development on the Windows platform.

Even if you don't use the DirectX SDK yourself, and to do that you must be a quite versatile programmer, then you will find that the tools or plugins that you use do depend on it. For example, the WildTangent[3] game engine plugin makes the DirectX 7 functionality available through both javascript and a Java interface. So understanding what DirectX has to offer may help you in understanding and exploiting the functionality of your favorite tool(s) and plugin(s).

DirectX 9.0 components

In contrast to OpenGL, the DirectX SDK is not only about (3D) graphics. In effect, it offers a wide range of software APIs and tools to assist the developer of multimedia applications. The components of the DirectX 9 SDK include:

DirectX 9 components

- Direct3D – for graphics, both 2D and 3D
- DirectInput – supporting a variety of input devices
- DirectPlay – for multiplayer networked games
- DirectSound – for high performance audio

[3] www.wildtangent.com

- DirectMusic – to manipulate (non-linear) musical tracks
- DirectShow – for capture and playback of multimedia (video) streams

In addition there is an API for setting up these components. Also, DirectX supports so-called *media objects*, which provide a standard interface to write audio and video encoders, decoders and effects.

Altogether, this is a truly impressive and complex collection of APIs. One way to become familiar with what the DirectX 9 SDK has to offer is to start up the sample browser that is part of the SDK and explore the demos. Another way is to read the online documentation that comes with the SDK, but perhaps a better way to learn is to make your choice from the large collection of introductory books, and start programming. At the end of this chapter, I will provide some hints about how to get on your way.

Direct3D

In the DirectX 9 SDK, Direct3D replaces the DirectDraw component of previous versions, providing a single API for all graphics programming. For Direct3D there is a set of simple, well-written tutorials in the online documentation, that you should start with to become familiar with the basics of graphics programming in DirectX.

Direct3D tutorials

- tutorial 1: creating a device
- tutorial 2: rendering vertices
- tutorial 3: using matrices
- tutorial 4: creating and using lights
- tutorial 5: using texture maps
- tutorial 6: using meshes

Before you start working with the tutorial examples though, you should acquire sufficient skill in C++ programming[4] and also some familiarity with Microsoft Visual Studio .NET.

One of the most intricate (that means difficult) aspects of programming Direct3D, and not only for novices, is the creation and manipulation of what is called the *device*. It is advisable to take over the default settings from an example, and only start experimenting with more advanced setting after you gained some experience.

DirectSound – the *drumpad* example

The DirectX SDK includes various utility libraries, for example the D3DX library for Direct3D, to simplify DirectX programming.

As an example of a class that you may create with DirectSound, using such a utility library, look at the *drumpad* below. The *drumpad* class can be integrated

[4] The DirectX 9 SDK also offers APIs for C# and VisualBasic .NET. See the *research directions* at the end of this section.

in your 3D program, using DirectInput, to create your own musical instrument. The header of the class, which is, with some syntactical modifications, taken from the SDK samples section, looks as follows:

```
class  drumpad {
public:
    drumpad()
    ~drumpad();
    bool    initialize( DWORD dwNumElements, HWND hwnd );
    bool    load( DWORD dwID, const TCHAR* tcszFilename );
    bool    play( DWORD dwID );
protected:
    void    CleanUp();
    CSoundManager* m_lpSoundManager;
    CSound **      m_lpSamples;
};
```

The interface offers some methods for creating and destroying a *drumpad* object, initialization, loading sounds and for playing the sounds that you loaded. The *CSoundManager* is a class offered by the utility library for DirectSound.

The *play* function is surprisingly simple.

```
bool drumpad::play( DWORD id ) {
    m_lpSamples[id] -> Stop();
    m_lpSamples[id] -> Reset();
    m_lpSamples[id] -> Play( 0, 0 );
    return true;
}
```

The *id* parameter is a number that may be associated with for example a key on your keyboard or some other input device. Using the *drumpad* class allows you to make your own VJ program, as I did in the system I will describe in the next section. In case you are not familiar with either C++ or object-oriented programming, you should study object-oriented software development first. See for example Eliens (2000).

DirectShow

DirectShow is perhaps the most powerful component of the DirectX SDK. It is the component which made Mark Pesce remark that with the DirectX 9 SDK *digital convergence has become a reality.*[5] A technical reality, that is, Pesce (2003).

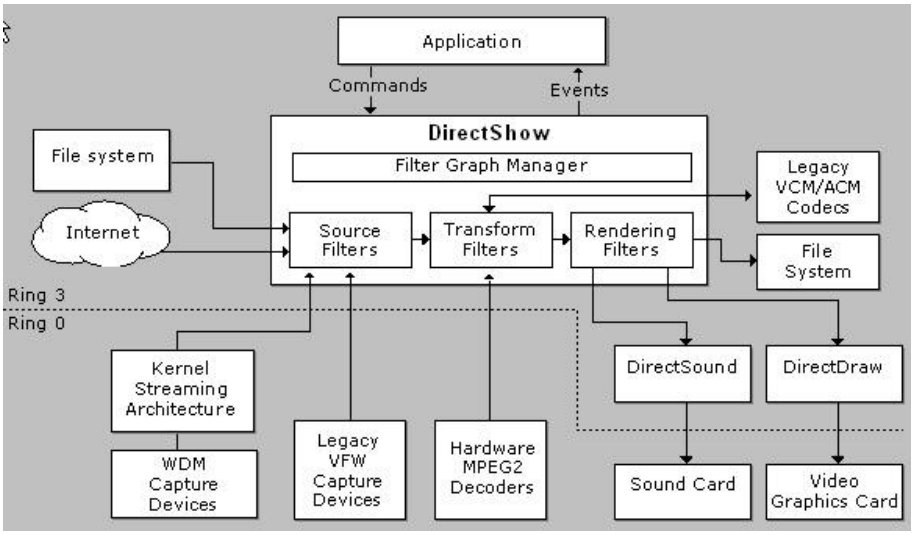
As we have seen in chapter 3, working with multimedia presents some major challenges:

multimedia challenges

[5] It is historically interesting to note that Mark Pesce may be regarded as the inventor, or initiator, of VRML, which was introduced in 1992 as the technology to realize a 3D web, as interlinked collection of 3D spaces.

- volume – multimedia streams contain large amounts of data, which must be processed very quickly.
- synchronization – audio and video must be synchronized so that it starts and stops at the same time, and plays at the same rate.
- delivery – data can come from many sources, including local files, computer networks, television broadcasts, and video cameras.
- formats – data comes in a variety of formats, such as Audio-Video Interleaved (AVI), Advanced Streaming Format (ASF), Motion Picture Experts Group (MPEG), and Digital Video (DV).
- devices – the programmer does not know in advance what hardware devices will be present on the end-user's system.

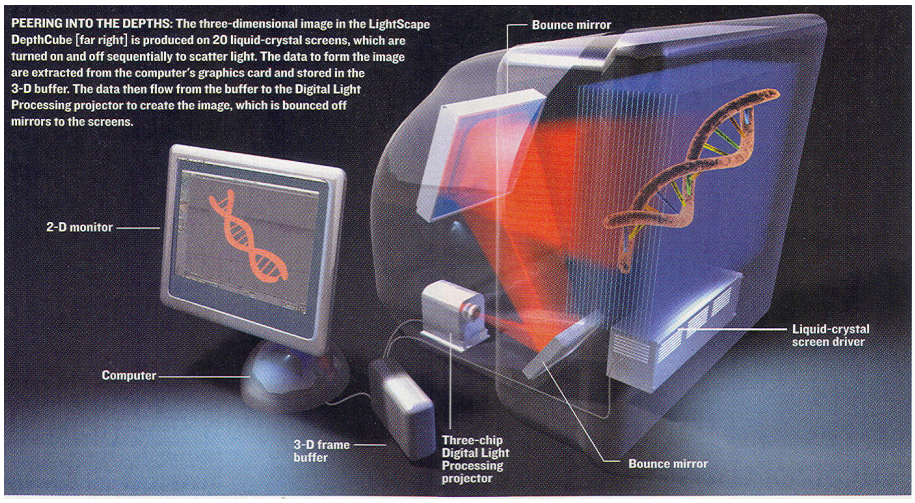
The DirectShow component was designed, as we learn from the online documentation, to address these challenges and to simplify the task of creating applications by isolating applications from the complexities of data transports, hardware differences and synchronization. The solution DirectShow provides is a modular architecture that allows the developer to set up a data flow graph consisting of *filters*. Such filters may be used for capturing data from, for example, a video camera or video file, for deploying a codec, through the audio compression manager (ACM) or video compression manager (VCM) interfaces, and for rendering, either to the file system or in the application using DirectSound and DirectDraw and Direct3D.



6

The diagram above, taken from the DirectX 9 SDK online documentation, shows the relations between an application, the DirectShow components, and some of the hardware and software components that DirectShow supports.

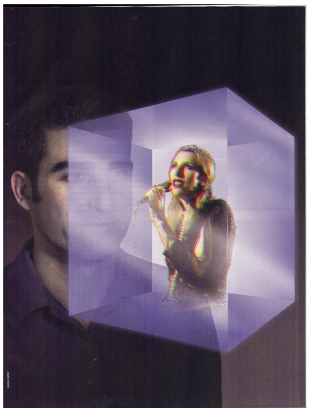
An interesting and convenient feature of the filter-based dataflow architecture of DirectShow is *SmartConnect*, which allows the developer to combine filters by indicating constraints on media properties such as format. The actual connections then, which involves linking input pins to output pins, is done automatically by searching for the right order of filters, and possibly the introduction of auxiliary filters to make things match.



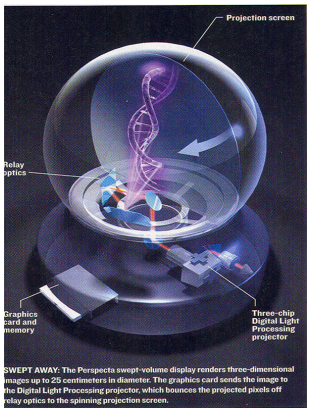
DepthCube, see example(s) – *3D vision*

DirectX application development

The examples that come with the DirectX‘9 SDK use an application utility library, which includes a general application class that takes care of most of the details of creating an application and rendering window, initialization and event handling. For each of the SDK components there are numerous examples, ranging in difficulty from beginners to expert level. There are also a number of examples that illustrate how to mix the functionality of different SDK components, as for example the projection of video on 3D, which we will discuss in more detail in the next section.



3D vision



Perspectra

DepthCube

8

example(s) – *3D vision*

Have you ever wondered how it would feel to be in Star Trek's holodeck, or experience your game in a truly spatial way, instead of on a flat LCD-display. In Sullivan (2005), an overview is given of technology that is being developed to enable volumetric display of 3D data, in particular the Perspecta swept-volume display (middle) and LightSpace DepthCube (right), that uses a projector behind a stack of 20 liquid-crystal screens.

The first approach of displaying volumetric data, taken by the Perspecta swept-volume display, is to project a sequence of images on a rotating sheet of reflective material to create the illusion of real volume. The psychological mechanism that enables us to see volumes in this way is the same as the mechanism that forces us to see motion in frame-based animation, at 24 frames per second, namely *persistence of vision*.

LightSpace DepthCube uses a stack of 20 transparent screens and alternates between these screens in a rapid way, thus creating the illusion of depth in a similar way. In comparison with other approaches of creating depth illusion, the solutions sketched above require no special eyewear and do not impose any strain on the spectator due to unnatural focussing as for example with autostereoscopic displays.

For rendering 3D images on either the Perspecta or DepthCube traditional rendering with for example OpenGL suffices, where the z-coordinate is taken as an indication for selecting a screen or depth position on the display. Rendering with depth, however, comes at a price. Where traditional rendering has to deal with, say 1024x748 pixels, the DepthCube for example needs to be able to display 1024x748x20, that is 15.3 million, *voxels* (the volumetric equivalent of a pixel) at a comparable framerate.

research directions– *the next generation multimedia platform*

Factors that may influence your choice of multimedia development platform include:

- platform-dependence – both hardware and OS
- programming language – C/C++, Java, .NET languages
- functionality – graphics, streaming media
- deployment – PC/PDA, local or networked, web deployment

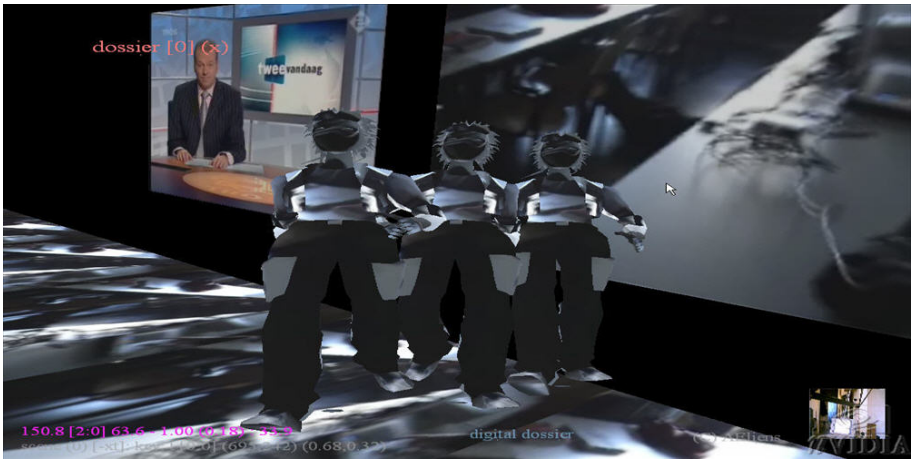
A first dividing line is whether you prefer to develop on/for Linux or Microsoft windows. Another dividing line, indeed, is your choice of programming language, C/C++, Java or .NET languages. Another factor that may influence your choice is the functionality you strive for. For example, Managed DirectX, for the .NET languages, provides only limited support for DirectShow and does not allow for capturing live video from a DV camera. And finally, it matters what deployment you wish to target for, mobile phone, PDAs or PCs, and whether you plan to make stand-alone applications or applications that must run in a web browser.

Apart from the hard-core programming environments such as the Microsoft DirectX 9 SDK, the Java Media Framework, OpenGL with OpenML extensions for streaming media, or the various open source (game development) toolkits, there are also high-level tools/environments, such as Macromedia Director MX, that allow you to create similar functionality with generally less effort, but also less control. In appendix E, a number of resources are listed that may assist you in determining your choice.

Given the range of possible options it is futile to speculate on what the future will offer. Nevertheless, whatever your choice is, it is good to keep in mind, quoting Bill Gates:

> *Software will be the single most important force in digital entertainment over the next decade.*

It should not come as a surprise that this statement is meant to promote a new initiative, XNA, which as the announcement says *is targeted to help contain the skyrocketing development costs and allow developers to concentrate on the unique content that differentiates their games.*



Animation in front of television news in ViP

9

4.3 merging video and 3D

In june 2003, I was approached by a theatre production company to advice on the use of "VR in theatre". As described in more detail in section 9.3, I explored what technology was available to realize such VR-augmented theatre. These explorations resulted in the development of the ViP system, that I once announced as follows:

www.virtualpoetry.tv

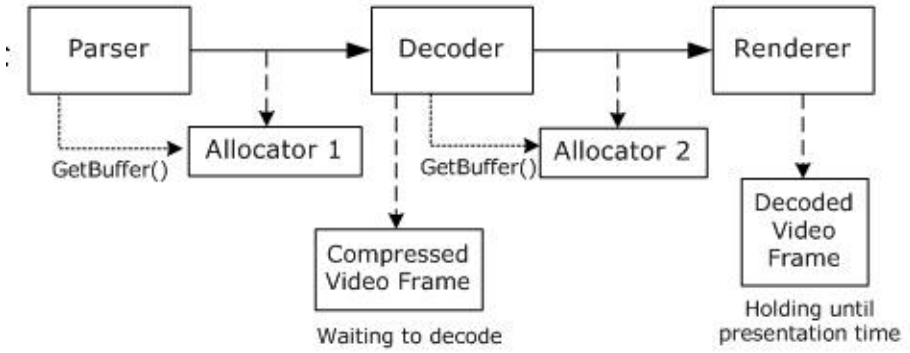
> *The ViP system enhances your party with innovative multimedia presentations.*
>
> *It supports multiple webcams and digital video cameras, mixed with video and images, enhanced by 3D animations and text, in an integrated fashion.*
>
> *For your party, we create a ViP presentation, with your content and special effects, to entertain your audience.*

In the course of time, I continued working on the system and, although I do not use it for parties, but rather for enlivening my lectures, it does include many of the features of a VJ system, such as the *drumpad* described in 3.2.

The major challenge, when I started its development, was to find an effective way to map live video from a low/medium resolution camera as textures onto 3D geometry. I started with looking at the ARToolkit but I was at the time not satisfied with its frame rate. Then, after some first explorations, I discovered that mapping video on 3D was a new (to some extent still experimental) built-in feature of the DirectX 9 SDK, in the form of the VMR9 (video mixing renderer) filter.

the Video Mixing Renderer filter

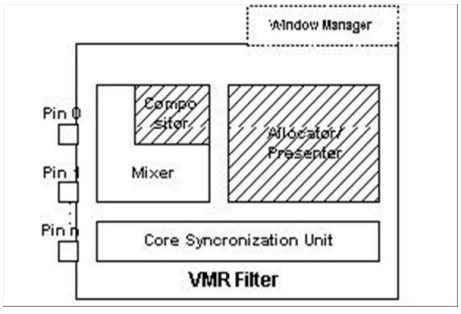
The VMR filter is a compound class that handles connections, mixing, compositing, as well as synchronization and presentation in an integrated fashion. But before discussing the VMR9 in more detail, let's look first at how a single media stream is processed by the filter graph, as depicted in the figure below.



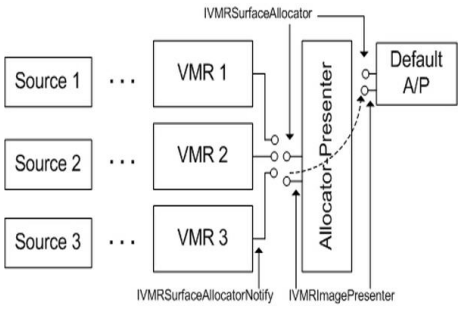
10

Basically, the process consists of the phases of parsing, decoding and rendering. For each of these phases, dependent on respectively the source, format and display requirements, a different filter may be used. Synchronization can be either dtermined by the renderer, by pulling new frames in, or by the parser, as in the case of live capture, by pushing data on the stream, possibly causing the loss of data when decoding cannot keep up with the incoming stream.

The VMR was originally introduced to allow for mixing multiple video streams, and allowed for user-defined *compositor* and *allocator/presenter* components.



(a) VMR filter



(b) multiple VMRs

11

Before the VMR9, images could be obtained from the video stream by intercepting this stream and copying frames to a texture surface. The VMR9, however, renders the frames directly on Direct3D surfaces, with (obviously) less overhead. Although the VMR9 supports multiple video pins, for combining multiple video

streams, it does not allow for independent search or access to these streams. To do this you must deploy multiple video mixing renderers that are connected to a common allocator/presenter component, as depicted on the right of the figure above (b).

When using the VMR9 with Direct3D, the rendering of 3D scenes is driven by the rate at which the video frames are processed.

Lecture on digital dossier for Abramovic, in ViP

12

the ViP system

In developing the ViP system, I proceeded from the requirement to project live video capture in 3D space. As noted previously, this means that incoming video drives the rendering of 3D scenes and that, hence, capture speed determines the rendering frame rate.

I started with adapting the simple *allocator/presenter* example from the DirectX 9 SDK, and developed a scene management system that could handle incoming textures from the video stream. The *scene* class interface, which allows for (one-time) initialization, time-dependent compositing, restore device setting and rendering textures, looks as follows:

```
class scene {
public:
        virtual int init(IDirect3DDevice9*);   // initialize scene (once)
          virtual int compose(float time);   // compose (in the case of an
    animation)
        virtual int restore(IDirect3DDevice9*);  // restore device settings
         virtual int render(IDirect3DDevice9*  device,  IDirect3DTexture9*
    texture);
protected:
```

```
...
};
```

The scene graph itself was constructed as a tree, using both arrays of (sub) scenes as well as a dictionary for named scenes, which is traversed each time a video texture comes in. The requirements the scene management system had to satisfy are further indicated in section 9.3. Leaving further details aside, observe that for the simple case of one incoming video stream, transferring the texture by parameter suffices.

Later on, I adapted the *GamePlayer* which uses multiple video mixing renderes, and then the need arose to use a different way of indexing and accessing the textures from the video stream(s). So, since it is good practice in object-oriented software engineering to suppress parameters by using object instance variables, the interface for the *scene* class changed into:

```
class scene {
public:
        virtual int load();   // initialize scene (once)
        virtual int compose(); // compose (in the case of an animation)
        virtual int restore(); // restore device settings
        virtual int render(); // display the (sub) scene
protected:
...
};
```

Adopting the scene class as the unifying interface for all 3D objects and compound scenes proved to be a convenient way to control the complexity of the ViP application. However, for manipulating the textures and allocating shader effects to scenes, I needed a global data structure (dictionaries) to access these items by name, whenever needed. As a final remark, which is actually more concerned with the software engineering of such systems that its functionality per se, to be able to deal with the multiple variant libraries that existed in the various releases of DirectX 9, it was needed to develop the ViP library and its components as a collection of DLLs, to avoid the name and linking clashes that would otherwise occur.

installation

reality of TV news

13

example(s) – *reality of TV news*

The *Reality of TV news* project by Peter Frucht uses an interesting mix of technology:

- live video capture from the device of an external USB2.0 TV card
- live audio capture from the soundcard (line in)
- display of live audio and video with java3D (had to be invented)
- autonomous 3D objects with a specified lifetime
- collision behaviour (had to be invented)
- changing of texture-, material- and sound characteristics at runtime
- dual-screen display with each screen rotated toward the other by 45 degrees about the Y-axis
- 3D sound

In the project, as phrased by Peter Frucht, *the permanent flow of the alternating adverts and news reports are captured live and displayed in a 3D virtual-reality installation. The currently captured audio and video data is displayed on the surface of 3D shapes as short loops. The stream enters the 3D universe piece by piece (like water drops), in this way it is getting displaced in time and space - news reports and advertising will be displayed partly in the same time. By colliding to each other the 3D shapes exchange video material. This re-editing mixes the short loops together, for instance some pieces of advertising will appear while the newsreader speaks.*

The software was developed by Martin Bouma, Anthony Augustin and Peter Frucht himself, with jdk 1.5, java3d 1.31, Java Media Framework 2.1.1e. The primary technological background of the artist, Peter Frucht, was the book *CodeArt*[6], Trogemann & Viehoff (2004), by his former professor from the Media Art School in Cologne, Germany. The book is unfortunately only available in German, and should be translated in English!

[6] java.khm.de

research directions– *augmented reality*

In the theatre production that motivated the development of the ViP system, the idea was to have wearable LCD-projection glasses, with a head-mounted low resolution camera. This setup is common in *augmented reality* applications, where for example a historic site is enriched with graphics and text, laid on top of the (video rendered) view of the site. Since realtime image analysis is generally not feasible, either positioning and orientation information must be used, or simplified markers indicating the significant spots in the scene, to determine what information to use as an overlay and how it should be displayed.

The ARToolkit[7] is an advanced, freely available, toolkit, that uses fast marker recognition to determine the viewpoint of a spectator. The information that is returned on the recognition of a marker includes both position and orientation, which may be used by the application to draw the overlay graphics in accordance with the spectator's viewpoint.

Augnented reality is likely to become a hot thing. In april 2005 it was featured at BBC World[8], with a tour through Basel.

4.4 development(s) – gaming is a waste of time

From a technical perspective, Second Life offers an advanced game engine that visitors and builders use (implicitly) in their activities. Before discussing how Second Life compares to (a selection of) other game engines and virtual environment frameworks, it is worthwhile to look at an overview of the main functional components of a *game engine*, which according to Sherrod (2006) encompass:

- rendering system – 2D/3D graphics
- input system – user interaction
- sound system – ambient and re-active
- physics system – for the blockbusters
- animation system – motion of objects and characters
- artificial intelligence system – for real challenge(s)

Although it is possible to build one's own game engine using OpenGL or DirectX, or the XNA[9] framework built on top of (managed) DirectX, in most cases it is more profitable to use an existing game engine or 3D environment framework, since it provides the developer with a load of already built-in functionality. In the following table, we give a brief comparative technical overview of, respectively, the Blaxxun Community Server (BlC), AlphaWorld (AW), the open source Delta3D engine (Δ3D), the Half Life 2 Source SDK (HL2), and Second Life (SL).

[7] artoolkit.sourceforge.net
[8] www.bbcworld.com/content/template_clickonline.asp?pageid=665&co_pageid=3
[9] crosoft.com/directx/XNA

	BlC	AW	Δ3D	HL2	SL
in-game building	-	+	+/-	-	++
avatar manipulation	+	++	+/-	+	++
artifical intelligence	+	-	+/-	+	-
server-side scripts	+	-	+/-	+	++
client-side scripts	++	-	+/-	+	-
extensibility	+	-	++	+	+/-
open source	-	-	++	-	+/-
open standards	-	-	+/-	-	+/-
interaction	+/-	+/-	++	++	+/-
graphics quality	+/-	+/-	++	++	+
built-in physics	-	-	+	++	+
object collision	-	-	++	++	+
content tool support	+/-	-	++	+	-

Obviously, open source engines allow for optimal extensibility, and in this respect the open source version of the SL client may offer many opportunities. Strong points of SL appear to be *in-game building*, *avatar manipulation*, and in comparison with BlC and AW *built-in physics* and *object collision detection*. Weak points appear to be *content development tool support*, and especially in comparison with Δ3D and HL2 *interaction*. For most types of action-game like interaction SL is simply too slow. This even holds for script-driven animations, as we will discuss in the next section. In comparison with a game as for example Age of Empires III[10], which offers in-game building and collaboration, Second Life distinguishes itself by providing a 3D immersive physics-driven environment, like the 'real' game engines.

modular architecture participatory deployment

In the beginning, we envisioned the realization of our climate game as a first-person perspective role-playing game in a 3D immersive environment as for example supported by the Half Life 2 SDK, with which we gained experience in creating a *search the hidden treasure*[11] game in a detailed 3D virtual replica of our faculty. However, we soon realized that the use of such a development platform, would require far too much work, given the complexity of our design. So, instead of totally giving up on immersion, we decided to use *flash* video[12], indeed as a poor-man's substitute for real 3D immersion, which, using *flash*[13] interactive animations, has as an additional benefit that it can be used to play games online, in a web browser. Together with the Flex 2 SDK[14], which recently became open source, *flash* offers a rich internet application (RIA) toolkit, that is sufficiently versatile for creating (online) games, that require, in relation to console games or highly realistic narrative games like Half Life, a comparatively moderate development effort. To allow for component-wise development, we choose for a

[10] www.ageofempires3.com
[11] www.cs.vu.nl/∼eliens/game
[12] www.adobe.com/products/flash/video
[13] ww.adobe.com/devnet/flash
[14] www.adobe.com/products/flex/sdk

modular architecture, with four basic modules and three (variants) of integration modules, as indicated below.

Fig 5. Clima Futura Architecture

1. climate model(s) - action script module(s)
2. game play interaction - event-handler per game event
3. video content module - video fragment(s) and interaction overlays
4. minigame(s) - flash module(s) with actionscript interface
5. Clima Futura - integration of modules 1-4, plus server-side ranking
6. adapted versions – educational, commercial
7. multi-user version –with server-side support

14

questions

multimedia platforms

1. What components does a multimedia platform consist of? Discuss both hardware and software components.

concepts

2. Characterize the functionality of current multimedia platforms.
3. Explain the notions of vertex shader and pixel shader.
4. Indicate what solutions exist for merging video and 3D graphics.

technology

5. Characterize the capabilty of current GPUs.
6. What does HLSL stand for? Give some examples of what it is used for.
7. What are the components of the DirectX 9 SDK?
8. Explain how the VMR9 works. Give an example.

projects & further reading As a project, I suggest the development of shader programs using Rendermonkey[15] or the Cg Toolkit[16], or a simple game in DirectX.

You may further explore the possibilities of platform independent integration of 3D and media, by studying for example OpenML[17]. For further reading, among

[15] www.ati.com/developer/RenderMOnkey
[16] www.nvidia.com/cg
[17] www.khronos.org/openml

the many books about DirectX, I advice Luna (2003), Adams (2003) and Fay et al. (2004).

the artwork

1. dutch light – photographs from documentary film Dutch Light[18].
2. ViP – screenshot, with morphing shader, see section 4.3.
3. impasto – examples, see section 4.1
4. impasto – after a painting of van Gogh, using Cg shaders,
5. 3D vision, from Sullivan (2005), see example(s) section 4.2.
6. idem.
7. photographs of DirectX and multimedia books, by the author.
8. DirectX – diagram from online documentation.
9. ViP – screenshot, with the news and animations.
10. DirectX – diagram from online documentation.
11. DirectX – diagram from online documentation.
12. ViP – screenshot, featuring Abramovic.
13. Peter Frucht – *Reality of TV news*, see section 4.3.
14. signs – people, van Rooijen (2003), p. 248, 249.

The theme of the artwork of this chapter is *realism*. In the documentary *dutch light*, it was investigated whether the famous *dutch light* in 17th century painting really existed. The photographs shown here are a selection of shots that were taken on particular locations over a period of time. However, as an art historian formulated it in the documentary: *dutch light* is nothing but a *bag of tricks* shared by dutch 17th century painters. The examples from *impasto* demonstrated that, after all, realism is an arbitrary notion.

[18] www.dutchlight.nl