Appendix A
Fundamentals of the Graphics Pipeline Architecture

A pipeline is a series of data processing units arranged in a chain like manner with the output of the one unit read as the input of the next. Figure A.1 shows the basic layout of a pipeline.

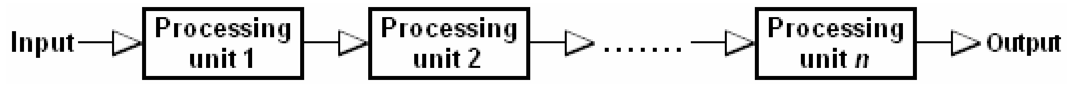

Figure A.1 Logical representation of a pipeline.

The throughput (data transferred over a period of time) many any data processing operations, graphical or otherwise, can be increased through the use of a pipeline. However, as the physical length of the pipeline increases, so does the overall latency (waiting time) of the system. That being said, pipelines are ideal for performing identical operations on multiple sets of data as is often the case with computer graphics.

The graphics pipeline, also sometimes referred to as the rendering pipeline, implements the processing stages of the rendering process (Kajiya, 1986). These stages include vertex processing, clipping, rasterization and fragment processing. The purpose of the graphics pipeline is to process a scene consisting of objects, light sources and a camera, converting it to a two-dimensional image (pixel elements) via these four rendering stages. The output of the graphics pipeline is the final image displayed on the monitor or screen. The four rendering stages are illustrated in Figure A.2 and discussed in detail below.

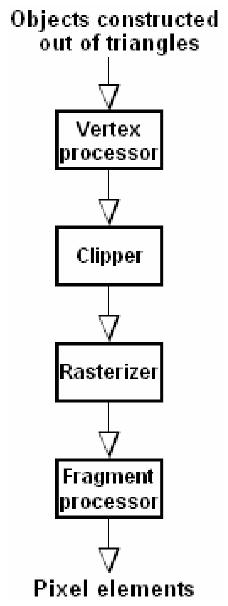

Figure A.2 A general graphics pipeline.

Summarised we can describe the graphics pipeline as an overall process responsible for transforming some object representation from local coordinate space, to world space, view space, screen space and finally display space. These various coordinate spaces are fully discussed in various introductory graphics programming texts and, for the purpose of this discussion, it is sufficient to consider the *local coordinate space* as the definition used to describe the objects of a scene as specified in our program's source code. The *world space* can be described as a coordinate space where we have a reference to the viewer's position with lighting added to our scene. *View space* is where our scene's objects are culled and clipped to determine whether an object is visible based on the position of the viewer or camera. *Screen space* is where hidden surface removal, shading and rasterization occur and it is the final stage before we enter the *display space* where the produced pixel elements are displayed via some output device (Sutherland et al, 1974). We will now look at the various stages of the graphics pipeline in detail.

A.1 Vertex Processing

The first processing unit of the graphics pipeline is the vertex processor. This processor is responsible for performing all geometric transformations and the computation of colour values of every vertex or point making up an object.

Geometric transformations (such as translations and rotations) simply refer to the process of converting the current spatial representation of an object to a different coordinate system. For example, a geometric transformation is required to represent an object, originally defined in terms of world coordinates (coordinates specified by the programmer for object representation), in terms of display coordinates (the coordinate system used by the graphics display). Each geometric transformation is defined using a matrix with a series of transformations specified by concatenating each of these matrices into a single one. Combining one matrix with another yields a third matrix that is once again combined with some other transformation matrix – an operation that clearly benefits from the use of a pipeline.

Three transformations are performed during the vertex processing stage. The first of these, namely the *modelling transformation*, takes the geometric specification of three-dimensional world objects as input. Every object, originally defined in local coordinate space, is subsequently transformed to use world-space coordinates. Each object's independent local coordinate system has now been transformed into a global coordinate system. This provides all the objects with a shared global coordinate space – i.e. one object's position can be described in terms of another's and these user defined objects can now be positioned within the same scene. All translations and rotations are performed during this transformation step.

The next transformation step, called the *viewing transformation*, transforms all world-space coordinates to coordinates specified in terms of a viewer's position and viewing direction. This transformation step leads to a viewer or camera that can be moved and rotated to any position within the world coordinate space. The original three-dimensional scene is displayed from this viewer's perspective (or point of view). Both culling (back-face elimination) and clipping are carried out in view space.

The final transformation, called the *projection transformation*, transforms the view space coordinates to two-dimensional image space or screen space so that the three-dimensional scene can be displayed on a flat plane.

The final step of the vertex processor is to assign colours, per-vertex lighting and shading to each of the vertices making up the scene (Swanson and Thayer, 1986). The rasterization stage discussed below interpolates these per-vertex lighting values for the creation of smoothly shaded lighting ranges between vertices.

A.2 Clipping and Culling

Clipping controls the field of view, i.e. managing the percentage of the world visible based on the camera's viewing angle and position. The lack of clipping does not hinder the image formation process, it is, however, crucial to ensure that this process is performed in a timely manner due to it eliminating the rendering of any unnecessary primitives that would not be visible to the viewer or camera. We define a volume similar to a stencil to block out objects not visible to the viewer. All objects and portions of objects falling outside this stencil or volume do not appear in the final image.

Clipping, unlike vertex processing, should be done on a primitive-by-primitive rather than a vertex-by-vertex basis. To accomplish this, sets of vertices are assembled into primitives, such as polygons and lines based on the implementation of some clipping algorithm such as the Cohen-Sutherland or Liang-Barsky line clipping algorithms or the Sutherland-Hodgman polygon clipping algorithm. An example illustrating the importance of clipping would be to consider a scene from a computer game consisting of numerous buildings, cars, pedestrians, shops, etc. Each of these elements are physical models stored in memory, requiring a lot of processing time for shading, texturing, animation, etc. If the scene's viewer or camera has a viewing angle of 110 degrees, then we needn't render any of the models or meshes located outside this viewing area – thus saving a lot of rendering time in the process.

Culling, or back-face elimination, refers to the process where polygons or surfaces pointing away from the camera or viewer are not rendered. For example; when a building is viewed directly from the front, then the three sides hidden from the viewer are not drawn (shown in Figure A.3). This process, just like clipping, improves the rendering speed of a scene by reducing the number of polygons or surfaces that needs to be rendered without affecting the visual output.

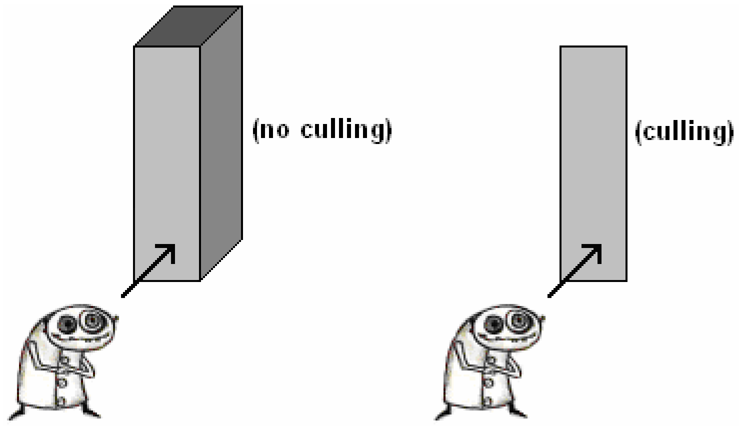

Figure A.3 Back-face elimination.

A.3 Rasterization and Fragment Processing

The rasterization, or scan conversion process converts the primitives produced by the clipper (consisting of vertices) to pixels for representation in the frame buffer and for subsequent output to a monitor. For example, a solid rectangle consisting of four vertices are transformed to two-dimensional pixels or points in the frame buffer, with these two-dimensional pixels being coloured and shaded as appropriate. The result of the rasterization process is a series of fragments for each of these primitives. A *fragment* is nothing more than a pixel with additional information about its colour, position and depth. The fragment's depth information is used to determine whether a particular pixel lies behind any of the other rasterized pixels. The matching pixel in the frame buffer is updated with the information carried by this fragment. This process of updating the pixels in the frame buffer with the fragments generated by the rasterizer is called *fragment processing*. The colour of fragments are manipulated using techniques such as texture mapping, bump mapping, texture filtering, environmental mapping, blending, per-fragment lighting, etc.

A.4 Programmable Pipelines

Today's graphics cards all have pipelines built into their graphics processing units. The operations that could be performed by earlier graphics cards were standardised by the device manufacturer with only a number of parameters and properties available for modification. Modern graphics cards allow for not only the modification of a large number of parameters, but also for complete control over the vertex and fragment processors. These programmable vertex and fragment processors enable the real time rendering of various advanced techniques only previously achievable using large rendering farms or not even possible in real-time at all (Möller and Haines, 2002). Bump mapping (used for adding depth to pixels and thus creating a lighting-dependent bumpiness to a texture mapped surface) and environmental mapping (used for the generation of reflections by changing the texture coordinates based on the position of the camera) are just two examples of techniques only possible off-line in the past (Blinn, 1976), but that have become commonplace in the games of today (Peercy et al, 1997). Figure A.4 shows a bump mapped surface with Figure A.5 showing the application of environmental mapping to simulate reflections on water.

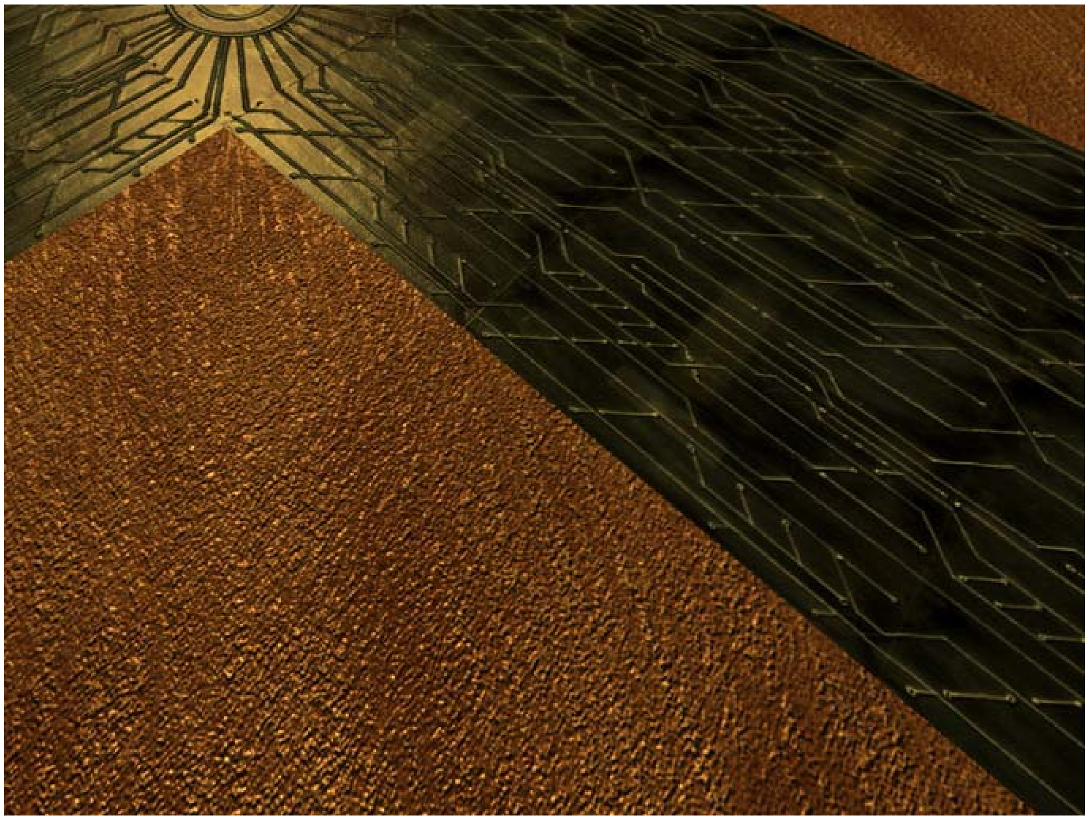
Figure A.4 Bump mapping.

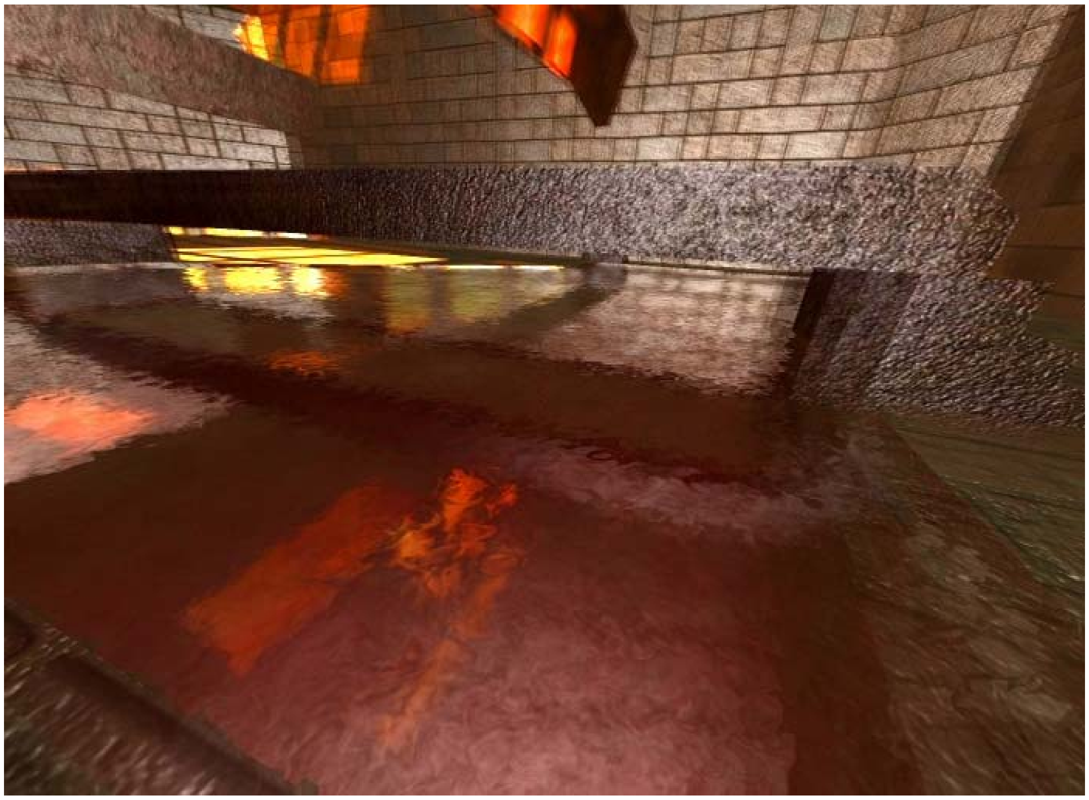
Figure A.5 Reflections on water using environmental mapping.

We will now look at Direct3D 10's programmable pipeline to fully understand the implication and use of programmable pipelines for the generation of advanced real-time graphical effects.

A.4.1 The Direct3D 10 Processing Pipeline

Each stage of the Direct3D 10 processing pipeline is configurable using the standard Direct3D application programming interface. The vertex shader, geometry shader and pixel shader are programmable using either Microsoft's proprietary High Level Shader Language (HLSL) or NVIDIA's C for Graphics (Cg). Each of these programmable processing units, including the pipeline processing states is discussed below. Figure A.6 illustrates the Direct3D 10 pipeline architecture.

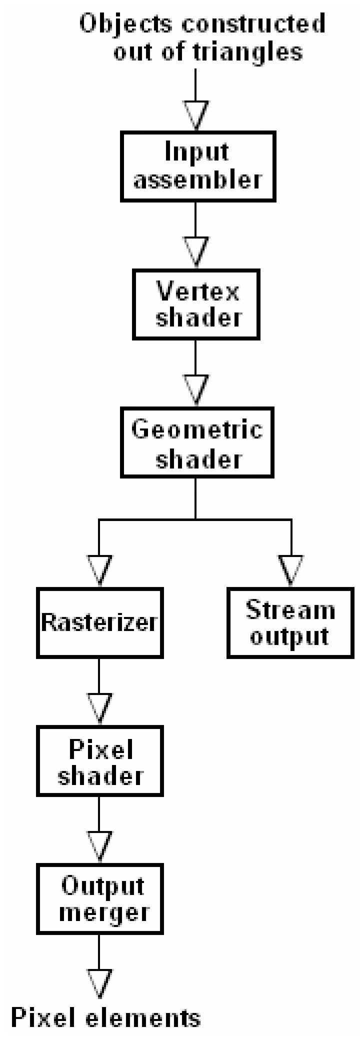

Figure A.6 Direct3D 10’s programmable pipeline.

The Input-Assembler Stage

The first stage of the programmable pipeline, namely the *input assembler stage*, is responsible for propagating geometric input data consisting of points, lines and polygons to the rest of the pipeline. This pipeline stage assembles the input data into primitives, following this it forwards these assembled primitives to the next stage in the pipeline. For example, when data is received from some buffer it contains information about a vertex in three-dimensional space, the winding direction used for determining the vertex assembly order (either clockwise or counter-clockwise) and an identifier specifying the first vertex in a sequence of vertices. This information allows the input assembler to create primitive types supported by Direct3D. Figure A.7 illustrates how this information is used to create a supported primitive type.

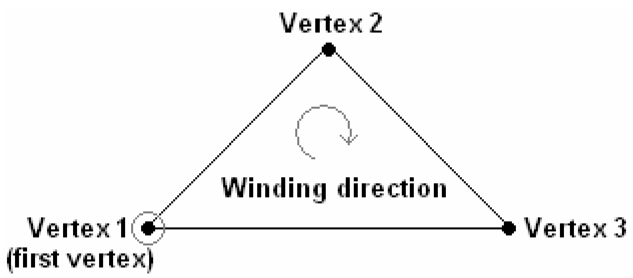

Figure A.7 Creating a triangle using three vertices, a clockwise winding direction and a vertex identifier indicating the first vertex in a set of three vertices.

The input assembler is also responsible for attaching *Shader System Values* for use by the shader core. These values (primitive id, vertex id, etc) lead to faster execution times by allowing the shader stages to ignore primitives that have already been dealt with.

Initialising the input assembler stage requires the specification of a vertex and optional index buffer that will be used for feeding the pipeline vertex data. The vertex buffer feeds the vertex data into the pipeline with the index buffer specifying indices for the vertex data stored in the vertex buffer. Creating a vertex buffer is relatively simple in Direct3D 10. We start by specifying the type of data that can be stored in the buffer (using the **D3D10_BUFFER_DESC** structure) followed by reading data into the buffer to initialise it (this data is specified using the **D3D10_SUBRESOURCE_DATA** structure). Once this is done we simply create the buffer using these descriptors. The **D3D10_BUFFER_DESC** structure describes the size of the buffer in bytes, the method how the buffer is to be read from and written to, the nature of the buffer (as a vertex buffer, index buffer, shader resource, etc), the kind of CPU access allowed (write, read, or 0 if no CPU access is necessary) and a flag to identify less regularly used options (such as resource sharing between various devices – 0 when not applicable). The D3D10.h header file specifies the **D3D10_BUFFER_DESC** structure as follows:

```
typedef struct D3D10_BUFFER_DESC {
    UINT ByteWidth;
    D3D10_USAGE Usage;
    UINT BindFlags;
    UINT CPUAccessFlags;
    UINT MiscFlags;
} D3D10_BUFFER_DESC;
```

The default values, including the alternatives, for the members of the **D3D10_BUFFER_DESC** structure are given in the following table:

Members	Flags
ByteWidth	Any number, for example: **64**
Usage	**D3D10_USAGE_DEFAULT**

	(won't be read or written to by the CPU that often)
	D3D10_USAGE_IMMUTABLE (can't be written to by the CPU at all)
	D3D10_USAGE_DYNAMIC (buffer will be written to by the CPU at least once per frame)
	D3D10_USAGE_STAGING (read from and write to the GPU)
BindFlags	**D3D10_BIND_VERTEX_BUFFER** (specify the resource as a vertex buffer)
	D3D10_BIND_INDEX_BUFFER (specify the resource as an index buffer)
	D3D10_BIND_CONSTANT_BUFFER (specify the resource as a constant buffer which can only be updated completely, not partially, and which has a limit on the buffer's byte size)
	D3D10_BIND_SHADER_RESOURCE (specify the buffer as a shader resource)
	D3D10_BIND_STREAM_OUTPUT (specify the resource as an output buffer for the stream output stage discussed below)
	D3D10_BIND_RENDER_TARGET (specify the resource as a render target)
	D3D10_BIND_DEPTH_STENCIL (specify the resource as a depth-stencil buffer)
CPUAccessFlags	**D3D10_CPU_ACCESS_READ** (the buffer's contents can be read by the CPU)
	D3D10_CPU_ACCESS_WRITE (the CPU can change the buffer's contents directly instead of using the **UpdateSubresource ID3D10Device** interface)
MiscFlags	**D3D10_RESOURCE_MISC_GENERATE_MIPS** (species the creation of mipmaps for some texture resource using the **GenerateMips ID3D10Device** interface)
	D3D10_RESOURCE_MISC_SHARED (enables resource sharing between various devices)
	D3D10_RESOURCE_MISC_TEXTURECUBE (specifies the creation of a cube texture – a three dimensional texture in the shape of a cube constructed from six textures stored in a 2-D texture array)

Table A.1 Describing a buffer resource using the **D3D10_BUFFER_DESC** structure.

Before initialising the **D3D10_BUFFER_DESC** structure, we first have to specify the vertices for some geometric object. In this case our vertices will have both a spatial location and a colour value (using the **D3DXVECTOR3** structure which has three members, an *x*-, *y*- and *z*-coordinate of a vector in three-dimensional space):

```
struct TriangleVertex
{
    D3DXVECTOR3 Location;
    D3DXVECTOR3 Colour;
};
```

We can now initialise the **D3D10_BUFFER_DESC** structure as follows for the specification of a vertex buffer description:

```
D3D10_BUFFER_DESC bufferDescription;

bufferDescription.Usage = D3D10_USAGE_DEFAULT;
bufferDescription.ByteWidth = sizeof(TriangleVertex) * 3;
bufferDescription.BindFlags = D3D10_BIND_VERTEX_BUFFER;
bufferDescription.CPUAccessFlags = 0;
bufferDescription.MiscFlags = 0;
```

Following this we create the vertex buffer using previously specified vertex data. The first step of this process is to specify an array of vertex data elements:

```
TriangleVertex array_of_vertex_data [] =
{
     D3DXVECTOR3( 0.0f, 1.0f, 1.0f ),
     D3DXVECTOR3( 0.0f, 0.0f, 0.5f ),
     D3DXVECTOR3( 1.0f, -1.0f, 1.0f ),
     D3DXVECTOR3( 1.0f, 0.0f, 0.0f ),
     D3DXVECTOR3( -1.0f, -1.0f, 1.0f ),
     D3DXVECTOR3( 0.0f, 1.0f, 0.0f ),
};
```

Next we have to initialise the **D3D10_SUBRESOURCE_DATA** structure. This data structure initialises a sub-resource using predefined data. A *sub-resource* is a portion of a resource that links back to the original resource data but with additional information about the resource so that the pipeline can easily access the data contained within this resource. The **D3D10_SUBRESOURCE_DATA** structure has three members, namely, a pointer to the data used for initialising the sub-resource, a value used for specifying the memory pitch in bytes required for two- and three-dimensional texture resources and the memory slice pitch associated with three-dimensional texture resources. The D3D10.h header file specifies this structure as follows:

```
typedef struct D3D10_SUBRESOURCE_DATA {
    const void *pSysMem;
    UINT SysMemPitch;
    UINT SysMemSlicePitch;
} D3D10_SUBRESOURCE_DATA;
```

We initialise the **D3D10_SUBRESOURCE_DATA** structure using the previously defined array of vertex data elements:

```
D3D10_SUBRESOURCE_DATA subresourceData;

subresourceData.pSysMem = array_of_vertex_data;
subresourceData.SysMemPitch = 0;
subresourceData.SysMemSlicePitch = 0;
```

The final step is to create the vertex buffer. We use the **CreateBuffer ID3D10Device** interface to do this. This interface takes three parameters, the first being a pointer to the previously defined **D3D10_BUFFER_DESC** structure, the second a pointer to the **D3D10_SUBRESOURCE_DATA** structure with the third being the address of a pointer to the **ID3D10Buffer interface** used for controlling our buffer resource (be it either a vertex or index buffer). The **CreateBuffer ID3D10Device** interface is declared as follows in the D3D10.h header:

```
HRESULT CreateBuffer(
  const D3D10_BUFFER_DESC *pDesc,
  const D3D10_SUBRESOURCE_DATA *pInitialData,
  ID3D10Buffer **ppBuffer
);
```

We can now call the **CreateBuffer ID3D10Device** interface to create the vertex buffer:

```
ID3D10Device* g_id3dDevice;
ID3D10Buffer* vertexBuffer[2] = {NULL, NULL};

g_id3dDevice->CreateBuffer(&bufferDescription, &subresourceData, &vertexBuffer[0]);
```

Defining an index buffer is comparable to the creation of a vertex buffer, with the only difference being the specification of the **D3D10_BUFFER_DESC** structure's **BindFlags** member, for example:

```
D3D10_BUFFER_DESC indexBufferDescription;
```

```
indexBufferDescription.Usage = D3D10_USAGE_DEFAULT;
indexBufferDescription.ByteWidth = sizeof(TriangleVertex) * 3;
indexBufferDescription.BindFlags = D3D10_BIND_INDEX_BUFFER;
indexBufferDescription.CPUAccessFlags = 0;
indexBufferDescription.MiscFlags = 0;
```

We also have to specify an array containing index data. This array will be used to initialise the **`D3D10_SUBRESOURCE_DATA`** structure:

```
UINT array_of_index_data [] = {0, 1, 2, 3, 4};

D3D10_SUBRESOURCE_DATA indexSubresourceData;

indexSubresourceData.pSysMem = array_of_index_data;
indexSubresourceData.SysMemPitch = 0;
indexSubresourceData.SysMemSlicePitch = 0;
```

The index buffer is created using the **`CreateBuffer ID3D10Device`** interface:

```
ID3D10Buffer* indexBuffer = NULL;

g_id3dDevice->CreateBuffer(&indexBufferDescription,
                           &indexSubresourceData,
                           &indexBuffer);
```

With the input buffers specified and properly initialised, we create the *input-layout object* which will be used to control how vertex data is fed into the input-assembler stage (by directly describing the input-buffer data). The type of the input vertex data is identified and checked against shader parameter types ensuring both type compatibility and that the needed shader data is actually stored in the buffer. We create the input-layout object using the **`CreateInputLayout ID3D10Device`** interface via the specification of five parameters. The first parameter is an array of the input-assembler stage input data type described using the **`D3D10_INPUT_ELEMENT_DESC`** structure. The **`D3D10_INPUT_ELEMENT_DESC`** structure is defined as follows in the D3D10.h header file:

```
typedef struct D3D10_INPUT_ELEMENT_DESC {
    LPCSTR SemanticName;
    UINT SemanticIndex;
    DXGI_FORMAT Format;
    UINT InputSlot;
    UINT AlignedByteOffset;
    D3D10_INPUT_CLASSIFICATION InputSlotClass;
    UINT InstanceDataStepRate;
} D3D10_INPUT_ELEMENT_DESC;
```

This structure gives a description of each input assembler stage element, specifically; the High Level Shader Language (HLSL) semantic name of the element, the element's semantic index used when more than one element with the same semantic name exists, the element's data type, an integer value used for specifying the input-assembler's input slot (described below), the byte offset used to set the location of the element in the input slot (counting in bytes from the beginning of the input slot), the input data class (either vertex data using the **D3D10_INPUT_CLASSIFICATION** enumeration with the constant set to either **D3D10_INPUT_PER_VERTEX_DATA** for per-vertex input data, or **D3D10_INPUT_PER_INSTANCE_DATA** for per-instance input data) and the data step rate controlling the number of instances of one element to draw (using the per-instance input data) before moving on to the next buffer element – must be set **0** for elements containing per-vertex data. Using the **D3D10_INPUT_ELEMENT_DESC** structure, we can specify a vertex buffer containing two vertex-data elements as follows:

```
D3D10_INPUT_ELEMENT_DESC input_layout_description[] =
{
    {L"POSITION", 0, DXGI_FORMAT_R32G32B32_UINT, 0, 0, D3D10_INPUT_PER_VERTEX_DATA, 0},
    {L"COLOR", 0, DXGI_FORMAT_R32G32B32_UINT, 1, 6, D3D10_INPUT_PER_VERTEX_DATA, 0},
};
```

Data is fed into the input-assembler stage through a number of units referred to as *input slots*. Each of these input-assembler input slots, shown in Figure A.8, are used as storage for a vertex buffer, thus storing input data.

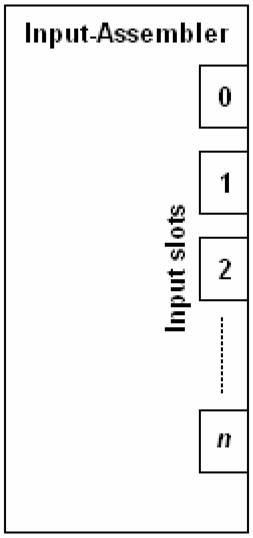

Figure A.8 The input-assembler's input slots.

The second parameter of the **`CreateInputLayout ID3D10Device`** interface is an integer value specifying the number of input-data types making up the input-elements array. The third parameter is a pointer to the compiled shader code with the fourth parameter specifying the byte size of this compiled shader code. The final parameter is a pointer to the input-layout object that will be used as output. This **`CreateInputLayout ID3D10Device`** interface is defined as follows in the D3D10.h header file:

```
HRESULT CreateInputLayout (
        const D3D10_INPUT_ELEMENT_DESC  *pInputElementDescs,
        UINT NumElements,
        const void *pShaderBytecodeWithInputSignature,
        SIZE_T BytecodeLength,
        ID3D10InputLayout **ppInputLayout);
```

We can now bind this newly created input-layout object to the input-assembler stage, after which we can call the draw functions. This object binding is done using the **`IASetVertexBuffers`** and **`IASetInputLayout ID3D10Device`** interfaces. The **`IASetVertexBuffers`** interface binds a vertex buffer array to the input-assembler stage by specifying the input slot, the total number of buffers in the vertex buffer array, a pointer to the vertex buffer array, a pointer to an array containing values indicating the byte size of elements to be read from the vertex buffer (referred to as stride values) and a pointer to an array containing so called offset values (with one offset value representing the number of bytes to be read from the first element stored in the vertex buffer to the element being accessed). This **`IASetVertexBuffers ID3D10Device`** interface is defined as follows in the D3D10.h header file:

```
void IASetVertexBuffers(UINT StartSlot, UINT NumBuffers,
                        ID3D10Buffer *const *ppVertexBuffers,
                        const UINT *pStrides,
                        const UINT *pOffsets);
```

The **`IASetInputLayout`** interface, taking a pointer to the input-layout object, is responsible for binding this object to the input-assembler stage. The following code sample illustrates this process:

```
UINT   start_input_slot = 0;
UINT   number_buffers_in_array = 1;
UINT offset_value = 0;
UINT stride_value = sizeof(TriangleVertex);

g_id3dDevice->IASetVertexBuffers(start_input_slot,
                                 number_buffers_in_array,
                                 &vertexBuffer,
                                 &stride_value,
                                 &offset_value);
```

The input-layout takes a pointer to the **`ID3D10Device`** object:

```
ID3D10InputLayout* inputLayoutObject = NULL;

g_id3dDevice->IASetInputLayout(inputLayoutObject);
```

The only remaining step is to specify the assembling of vertices into primitives and to send these primitives (controlling the rendering of vertex data to the screen) to the next step of the pipeline. This is done using the **`IASetPrimitiveTopology`** **`ID3D10Device`** interface. This interface takes one parameter, namely the primitive type specified using the **`D3D10_PRIMITIVE_TOPOLOGY`** enumerator. For example, the following code specifies the primitive type as a list of lines:

```
g_id3dDevice->IASetPrimitiveTopology( D3D10_PRIMITIVE_TOPOLOGY_LINELIST);
```

Table A.2 lists possible primitive types:

Constant	**Description**
`D3D10_PRIMITIVE_TOPOLOGY_UNDEFINED`	A primitive topology is not specified for the Input-assembler stage.
`D3D10_PRIMITIVE_TOPOLOGY_LINELIST`	The vertex data is interpreted as a list of lines.
`D3D10_PRIMITIVE_TOPOLOGY_LINELIST_ADJ`	The vertex data is interpreted as a list of lines with adjacency data.
`D3D10_PRIMITIVE_TOPOLOGY_LINESTRIP`	The vertex data is interpreted as a line strip.
`D3D10_PRIMITIVE_TOPOLOGY_LINESTRIP_ADJ`	The vertex data is interpreted as a line strip with adjacency data.
`D3D10_PRIMITIVE_TOPOLOGY_POINTLIST`	The vertex data is interpreted as a list of points.
`D3D10_PRIMITIVE_TOPOLOGY_TRIANGLELIST`	The vertex data is interpreted as a list of triangles.
`D3D10_PRIMITIVE_TOPOLOGY_TRIANGLELIST_ADJ`	The vertex data is interpreted as a list of triangles with adjacency data.
`D3D10_PRIMITIVE_TOPOLOGY_TRIANGLESTRIP`	The vertex data is interpreted

	as a triangle strip.
`D3D10_PRIMITIVE_TOPOLOGY_TRIANGLESTRIP_ADJ`	The vertex data is interpreted as a triangle strip with adjacency data.

Table A.2 Specifying a primitive type using the **`D3D10_PRIMITIVE_TOPOLOGY`** enumerator.

We can now draw these pipeline bound primitives using various **`ID3D10Device`** functions such as **`Draw`**, **`DrawAuto`**, **`DrawIndexed`**, **`DrawInstanced`** and **`DrawIndexedInstanced`**.

The Vertex-Shader Stage

Per-vertex operations are performed during this pipeline processing stage. Examples of such operations include per-vertex lighting, texture sampling operations, geometric transformations, etc. Per-vertex lighting allows us to specify distinct light sources, including the interaction of these light sources with adjacent surfaces. These interactions and reflections are considered on a per-vertex basis with the lighting values between vertices being approximated. This stage takes one vertex as input, modifies it according to some predefined operation and outputs it for further processing. There might also be cases where no vertex processing is required, leading to the definition of a pass-through vertex shader. This *pass-through vertex shader* forwards the input vertex data to the geometry-shader stage unmodified.

Input vertex data generally consist of anything from one to sixteen 32-bit vectors made up of one to four elements each. The input assembler basically feeds two data elements into the vertex-shader stage, namely; the vertex ID and the instance ID. These IDs are generated by the graphics hardware and can only be handled during this pipeline stage.

The Geometry-Shader Stage

Primitives such as vertices, lines and polygons are processed during this pipeline stage. The geometry-shader stage takes these primitives as input, and processes them based on some programmatically defined algorithm, forwarding these newly modified or, in some cases, newly generated primitives to either the stream-output stage or rasterizer stage. The geometry-shader stage takes full primitives as input, for example; lines consisting of two vertices, quads constructed out of four vertices, etc (Stam and Loop, 2003). This is in contrast with vertex shaders which only accept a single vertex as input.

One useful feature of the geometry-shader is its ability to handle edge-adjacent primitives. For example, say we have a quad as input; then the vertex data of all primitives adjacent to the quad can also be read as input. Figure A.9 shows such a quad with four adjacent quads.

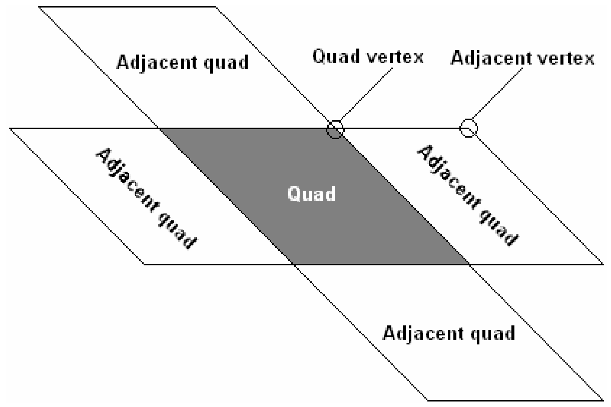

Figure A.9 A quad with edge-adjacent primitives.

The geometry-shader's generated primitives are returned as an output stream object. This output stream can be declared as a **`LineStream`** (creating a line strip output topology), **`TriangleStream`** (creating a triangle strip output topology) or **`PointStream`** (creating a point list output topology) based on the original primitive object type. We create a primitive strip by appending output vertices using the **`Append`** interface method. The appending of vertices is necessary since the geometry-shader only outputs one vertex data element at a time – requiring this vertex data to be reconstructed into primitives. The **`RestartStrip`** method is used to terminate the current primitive strip construction process, signalling the geometry-shader to start the creation of a new primitive strip. The following non-functional code sample shows the creation of a **`TriangleStream`** output object via the declaration of a geometry-shader.

We start by setting the maximum number of vertices to output using the **`MaxVertexCount`** attribute type (causing the geometry-shader to terminate once the specified number of vertices has been generated):

```
[MaxVertexCount(6)]
```

Next we declare the geometry-shader, **`GS_Sample`**, to take a triangle strip or triangle list (**`triangle float4 inputPar[3]`**) as input; with a **`TriangleStream`** object as output (the **`inout`** keyword declares the stream object, **`outputPar`**, as both an input and output):

```
void GS_Sample(triangle float4 inputPar [3], inout TriangleStream<float2> outputPar)
{
        //function body
        //e.g. using Append and RestartStrip:
        outputPar.Append(...);
        outputPar.RestartStrip();
}
```

Modern day computer games are increasingly making use of geometric shaders, mostly due to the exponential advances being made in graphics hardware and the power given to developers in controlling this hardware at a functional level using shading languages such as HLSL, Cg and the OpenGL Shading Language (GLSL). Examples of effects derived from programming DirectX 10's geometry-shader include shadow volume generation, fur animation, advanced dynamic particle systems, cube mapping, point sprite expansion and various other per-primitive operations.

Stream Output Stage

The stream output stage streams primitives from the geometry-shader stage to predefined buffers in system memory or memory present on the graphics card. This data can either be fed back into the input-assembler stage or alternatively loaded directly into shaders via load functions, or circulated to the CPU, for example (Figure A.10). The adjacency data associated with primitives outputted by the geometry-shader stage is discarded when output is directed to the stream output stage. Triangle and line strips are also converted to triangle and line lists when streamed to the buffer resources in memory.

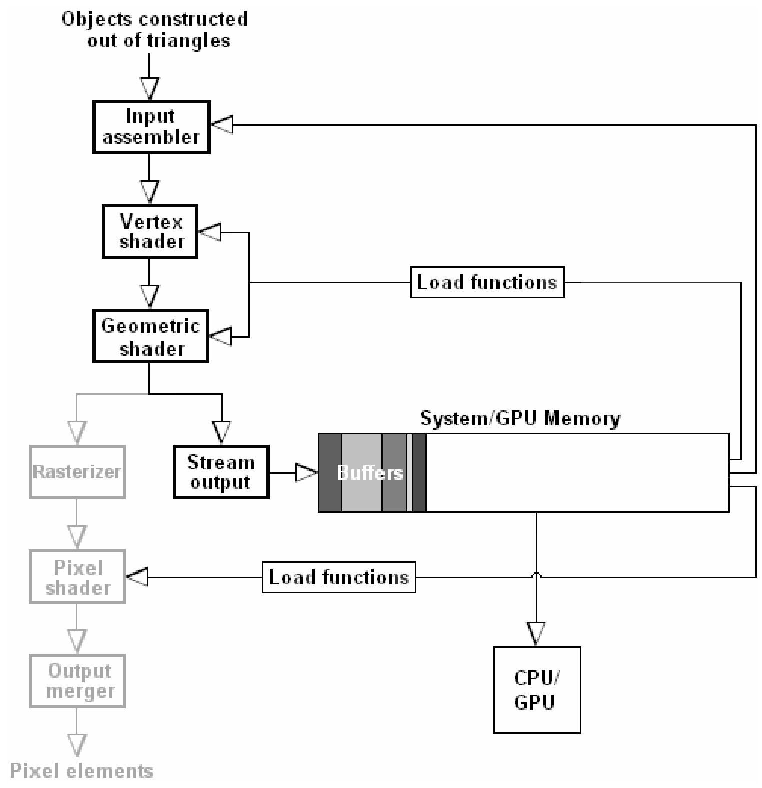

Figure A.10 Streaming of data to predefined buffers in system/GPU memory.

One or multiple buffers can be linked to the stream output stage. When one buffer is linked, then anything from 1 to 64 scalar per-vertex data elements can be written to the buffer (assuming a total size less than 257 bytes for the per-vertex output data elements). The use of multiple buffers, with each catching a single per-vertex data element, enables us to output data to a maximum of four buffers concurrently. When using multiple buffers it is not required for all the buffers to have the same size. The output of data to these varying sized buffers terminate the moment the smallest buffer is full (unable to receive any more primitives as input).

The Pixel-Shader Stage

The rasterizer stage rasterizes primitives produced by the geometry-shader stage into pixels via the interpolation of vertex values for representation in the frame buffer and for subsequent output to a monitor. The shading and colour of these pixel values need to be calculated so that each primitive are correctly rendered to the display device. The rasterizer stage calls the pixel-shader stage for the computation of these per-pixel values. Various per-pixel shading techniques such as lighting, fog, bump mapping, shadows, distortion effects and shading are performed during this stage (Legakis, 1998). In addition to these effect-based per-pixel techniques, the pixel shader is also used for implementing level-of-detail algorithms and during the process of anisotropic filtering crucial for enhancing the image quality of distant located textures.

Programs defining pixel-shader operations are called shader programs and can be written in any of the following languages: Assembly, Cg, HLSL or GLSL. These programs normally take colour values, the interpolated per-vertex data produced by the rasterizer stage, and some user defined variables as input, producing the final pixel values that are forwarded to the output-merger stage.

The Output-Merger Stage

This final stage of the Direct3D 10 programmable pipeline combines both the output generated by the pixel-shader stage with depth- and stencil buffer values to produce the final pixel colour and shading values. The output-merger stage is directly responsible for determining the visibility of pixels based on the process of depth testing. The blending of pixel data (combining two or more pixel colour values), in addition to depth- and stencil testing, is also controlled during this pipeline stage.

Appendix B
Shaders

A shader is a grouping of instructions processed by the graphics accelerator to perform some form of special effect or rendering. The previous section presented the concept of programmable pipelines, in particular focusing on the Direct3D 10 and OpenGL processing pipelines. An application program allowing direct interaction with these previously discussed programming pipelines is called a *shader*. These shader programs, written in a shading language such as NVIDIA's Cg or Microsoft's High Level Shader Language, control the movement, composition, form and appearance of objects through direct manipulation of the graphics processing unit is programmable pipelines.

The instructions listed in a shader program are executed at a specific point in the rendering pipeline – thus leading to user-defined manipulation of vertex or pixel data, for example. More specifically, three types of shader programs can be written, namely, vertex shaders, pixel shaders and geometry shaders.

Vertex shaders, operating on vertex data, are executed as part of the graphics pipeline's geometric stage and are used to alter the geometric parameters (shape) of an object. A vertex shader program is fundamental for certain special effects such as grass blowing in the wind where the real time manipulation, transformation and displacement of per-vertex material attributes are necessary. The vertices produced by this shader are forwarded as input to a geometry shader.

Geometry shaders are executed just prior to the rasterizer and stream output pipeline stages. These shaders group numerous vertices into a geometric object that can be modified by a pixel shader program. Geometry shaders are extremely important in the detection of silhouetted-edges and shadow volume extrusion. These shaders, performing per-primitive computations, are also vital in the generation of new primitives. The primitives generated by the geometry shader stage are rasterized into fragments during the pipeline's rasterizer stage. These fragments are then sent to the pixel shader as input.

Pixel shaders, also known as *fragment shaders* and performing per-pixel processing, operate on the discrete pixels of a primitive, applying some effect to a primitive (such as bump mapping, shadowing, fog, etc) during the pixel shader stage. Per-pixel lighting and shadowing has greatly contributed to the realism of modern computer games. Examples of effects made possible through this form of per-pixel processing include texture blending, environmental mapping, normal mapping, real-time shadows (stencil shadow volumes) and reflections (Levoy and Hanrahan, 1996).

These three types of shaders are unified by the Direc3D 10 architecture – known as Shader Model 4.0. *Unified shaders* provide the application programmer with a uniform instruction set independent of whether a pixel shader or vertex shader is being implemented. This unified architecture is made possible through Windows Vista's Windows Display Driver Model and the coupled DirectX 10 API. Previous architectures required different instruction sets for both pixel and vertex shaders due to specific hardware architectural requirements. By unifying the independent shader instruction sets, GPU programming has become much more flexible. This unified model also allows workload sharing amongst the various pipeline processors, for example, when the GPU is mainly performing basic geometry rendering with little or no per-pixel processing being done, then the pixel shader can be assigned vertex processing. The first GPU offering support for this unified shader model was NVIDIA's GeForce 8 series – specifically the GeForce 8800 GTX and GTS.

The term used to describe this unified shader architecture, Shader Model 4.0, encapsulates the features offered by the specific shader version in question. For example, Shader Model 3.0 (as supported by Direct3D 9.0c) limits the number of executing instructions to 65536 while Direct3D 10's Shader Model 4.0 allows for an unlimited number of executing instructions. Shader Model 2.0 (the original Direct3D 9.0 shader specification) limits the number of executing instructions to 32 texture instructions and 64 arithmetic instructions. The version number of instructions is specified in terms of the shader's version number (**`ps_mainVersion_subVersion`** for pixel shaders and **`vs_mainVersion_subVersion`** for vertex shaders). For example, a vertex shader based on Shader Model 3.0 (DirectX 9.0c) will be declared as **`vs_3_0`**, a DirectX 9.0b Shader Model 2.0 pixel shader as **`ps_2_b`**, with a Shader Model 4.0 pixel shader declared as **`ps_4_0`**. NVIDIA's GeForce FX series of GPUs provide an optimised model for Shader Model 2.0 and we can thus define a vertex shader based on this model as **`vs_2_a`**.

The capabilities of shader programs are heavily dependent on the available graphics hardware. Older graphics hardware such as first-generation GPUs (NVIDIA's RIVA TNT2 and ATI's Rage series implementing the DirectX 6 feature set) were only capable of accelerating texture mapping operations as well as the rasterization of certain primitives such as triangles. These GPUs alleviated the CPU from updating individual pixels but vertex transformations such as rotation, translation and scaling were still CPU dependant. These GPUs, although slightly configurable, were not programmable.

The second-generation of GPUs, introduced in 1999/2000 with the release of NVIDIAs GeForce 256 GPU and also including the GeForce2 and ATI's Radeon 7500, relieved the CPU from 3-D vertex transformations and lighting computations. Both the OpenGL and DirectX 7 APIs supported these hardware vertex transformations, however, although highly configurable in the sense of offering support for certain effects such as cube mapping for textures and per-pixel colouring, these GPUs were still not strictly speaking programmable.

The first truly programmable GPUs were NVIDIA's third-generation GeForce3, GeForce4 Ti and ATI's Radeon 8500 series. These GPUs offered programmable vertex pipelines, thus allowing an application program to control vertex transformations and lighting. These GPUs also featured a higher level of per-pixel configurability, although not yet offering pixel pipeline programmability. DirectX 8 and the **`ARB_vertex_program`** OpenGL extension allowed access to the vertex programmability offered by these GPUs. Pixel shaders could be written using the DirectX 8 pixel shader functionality and numerous OpenGL extensions. These pixel shaders were obviously nothing as powerful as today's pixel shader programs, and were based on configuring the pixel pipeline, rather than freeing the CPU of pixel-shading operations.

Both per-vertex and per-pixel programmability have been available since the release of NVIDIA's GeForce FX and ATI's Radeon 9700 family of GPUs. Application developers were, with the release of these GPUs, for the first time able to assign the GPU for both vertex transformations and pixel operations. With these operations offloaded to the GPU, the CPU is free to perform other calculations. The DirectX 9 API and several OpenGL extensions give access the pixel and vertex programmability offered by these GPUs. A vertex shader replaces the configurable fixed-function operations performed by the vertex processor with instructions defined by the shader along with a pixel shader executing after the rasterizer stage. This pixel shader takes the fragments processed by the fragment processor/pixel shader stage as input, performing some operation on them. Fragments are processed based on some configurable fixed function in the absence of a pixel shader program.

Table B.1 highlights some key features introduced with certain milestone GPU releases as well as their respective DirectX and OpenGL version support.

GPU	**Main Feature(s)**	**API support**
- **`NVIDIA RIVA 128`**	- Basic vertex acceleration.	DirectX 5, OpenGL 1.0.
- **`NVIDIA RIVA TNT`** - **`NVIDIA RIVA TNT2`** - **`ATI Rage 128`**	- Multitexturing (applying more than one texture to a polygon, e.g. graffiti art or 'bullet holes' on a textured wall).	DirectX 6, OpenGL 1.1.
- **`NVIDIA GeForce 256`** - **`NVIDIA GeForce2`** - **`ATI R100 (Radeon 32, 64, 7000`** and **`7500)`**	- Hardware Transformations, Clipping and Lighting. - Cube mapping. - Fixed-function vertex processing. - Register combiners.	DirectX 7, OpenGL 1.2 (ATI supporting OpenGL 1.3)
- **`NVIDIA GeForce3`**	- Quadtexturing (using four pixel	DirectX 8,

	pipelines for the rendering of four independently textured pixels or alternatively two multitextured pixels) - Texture shaders. - Shader Model 1.1. - ARB_vertex_program (OpenGL extension for vertex shaders on both ATI and NVIDIA chipsets).	OpenGL 1.4.
- **NVIDIA GeForce4 Ti** - **ATI Radeon R200 (Radeon 8500** to **9250)**	- Hardware anti-aliasing. - Pixel Shader 1.2, 1.3 or 1.4. - Vertex Shader 1.1. - ATI_fragment_shader (OpenGL extension for fragment shaders on ATI cards only).	DirectX 8.1, OpenGL 1.4.
- **NVIDIA GeForce FX** - **ATI Radeon R300 (Radeon 9500** to **9800 XT** and including **Radeon X1050)**	- Full support for vertex and fragment shader programs. - Floating-point pixel processing. - Shader Model 2.0, 2.0a or 2.0b. - OpenGL Shading Language.	DirectX 9.0b, OpenGL 1.4 (NVIDIA chipsets featured limited support for OpenGL 2.0 with ATI chipsets offering full support).
- **NVIDIA GeForce 6** - **ATI Radeon R500 (Xbox 360 Xenos, Radeon X1300** to **Radeon X1950 XTX)**	- Hardware accelerated transparency. - Scalable Link Interface (SLI – parallel graphics processing using two or more graphics accelerators interlinked). - Shader Model 3.0. - OpenGL Shading Language Improved.	DirectX 9.0c, OpenGL 2.0.
- **NVIDIA GeForce 7**	- High Dynamic Range Lighting.	DirectX 9.0c, OpenGL 2.0.
- **NVIDIA GeForce 8** - **Radeon R600 (Radeon HD 2400** to **Radeon HD 2900 XT)**	- Unified Shaders. - Shader Model 4.0.	DirectX 10, OpenGL 2.1.
- **NVIDIA GeForce 9/100/250/260-295**	- Atomic functions (thread-safe) - Coverage Sample AA - 128 bit OpenEXR	DirectX 10, OpenGL 3.3.
- **NVIDIA GeForce 210/220/240/300**	- Shader Model 4.1	DirectX 10.1, OpenGL 3.3.
- **NVIDIA GeForce 400/500**	- Shader Model 5.0	DirectX 11, OpenGL 4.1, OpenCL 1.0

Table B.1 Features introduced by selected GPUs and DirectX and OpenGL versions.

B.1 The Hardware Graphics Pipeline Revisited

We previously described a pipeline as a series of parallel stages with each stage processing the output of the previous stage, in turn sending its output to a successive stage, and so forth. The graphics pipeline consists of a number of stages such as vertex processing, clipping, rasterization and fragment processing. These stages are responsible for converting some geometrically defined scene into a two-dimensional image (pixel elements) via a number rendering stages – each physically organised as a pipeline processing unit. We will now revisit our previous graphics pipeline architecture discussion, expanding on it by focussing more on the programmable graphics pipeline's physical (hardware-level) organisation.

A modern-day GPU is sent a grouping of vertices organised into a geometric primitive such as a sequence of points, lines or a triangle, for example. Each of these vertices has a number of attributes. Attributes can range from the vertex's individual colour value, its texture coordinates, a normal vector used during lighting calculations to spatial coordinates used for the positioning of the vertex. A generic graphics hardware pipeline is show in Figure B.1.

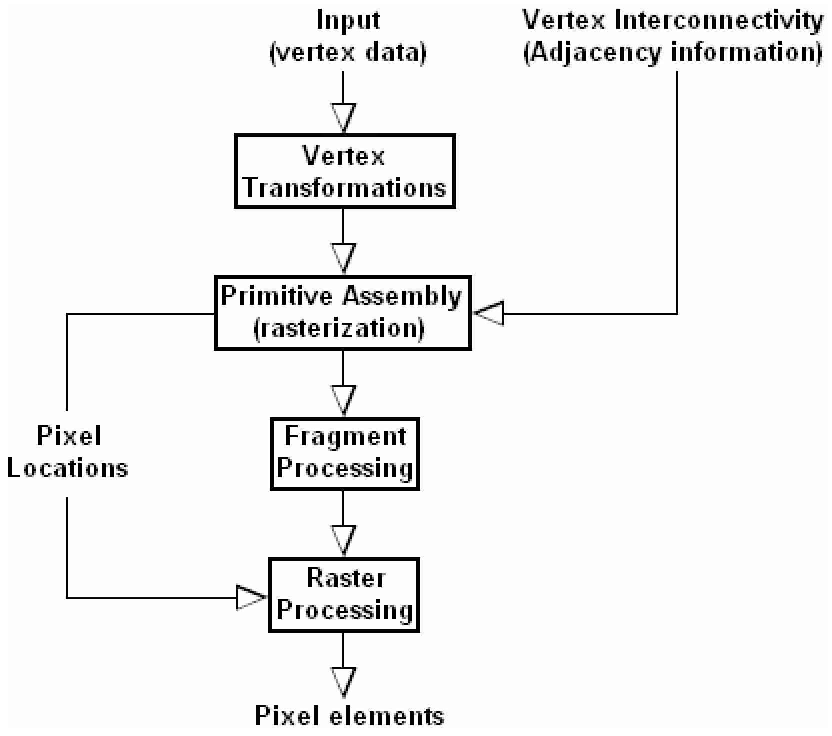

Figure B.1 A generic graphics hardware pipeline.

The vertex transformation stage performs a series of operations on each of the vertices sent to the GPU for processing. Operations include the transformation of a vertex's coordinate system into one that can be used by the rasterizer, per-vertex lighting, colouring and the generation of texture coordinates, etc.

The primitive assembly and rasterization stage assembles the vertices being passed from the vertex transformation stage into geometric primitives. The type of assembled primitive (line, polygon, triangle, etc) depends on the primitive topology data accompanying a set of vertices. Clipping to the visible view frustum is performed during this stage, resulting in the elimination of any unnecessary primitives that would not be visible to the viewer or camera (Liang and Barsky, 1984). The rasterization stage also eliminates polygons or surfaces pointing away from the camera or viewer (vertex-by-vertex culling). Following these operations, primitives are rasterized into pixels for representation in the frame buffer (Sutherland and Hodgman, 1974). Rasterization is performed according to a specific set of rules defined for each of the primitive topologies. The rasterization stage produces a set of pixels, each one mapped to a specific location, as well as a set of fragments (previously defined as a pixel with additional information about its colour, position and depth). Building on our previous definition we can now define a fragment as a state necessary for the update of a specific pixel in the frame buffer. During the rasterization process geometric primitives are broken down into pixel-sized fragments. Each fragment holds information about the pixel's location, depth, colour and texture coordinates. This information is then used to update a matching pixel in the frame buffer.

With a primitive successfully rasterized into a series of fragments, we can move on to the fragment processing stage. Fragment processing, as previously explained, is the process of updating pixels in the frame buffer with the fragments generated during the rasterization stage. The fragment processing unit is responsible for setting the colour values of fragments, their texturing as well as the interpolation of fragment parameters. These operations are modified and/or combined for numerous texturing effects such as bump mapping, texture filtering, blending, environmental mapping and so forth. Apart from calculating the fragment's final colour value, this pipeline stage can also discard a fragment based on some calculation or predefined parameter, hence resulting in the corresponding frame buffer pixel not being updated.

The final number of fragment centric operations, based on the functionality of Direct3D and OpenGL, are performed during the raster processing stage. These operations, such as depth testing (the removal of hidden surfaces), blending, stencil testing for the generation of stencil shadow volumes and stencil based reflections, etc, are performed prior to the frame buffer update. A number of tests are conducted during this pipeline stage; for example, a scissor test culls all the fragments located outside a user-specified rectangle positioned within the render target area, with an alpha test determining whether fragments are written to the render target area based on some predefined alpha-test function. A fragment is discarded whenever any of these tests fail. When passing a specific test, one of the pixel's property values (such as depth for depth testing) is updated with that of the fragment. The blending operation stage reads the fragment's colour value and combines it with the colour value of the matching pixel. We can also dither the colour values of fragments and pixels to create the illusion of colour depth in low-colour images by approximating colours not available in the palette through the diffusion of the available palette's colour values. The final operation is to write the new blended/dithered fragment colour value out to the appropriate pixel in the frame buffer. This raster processing stage, consisting of a series of pipeline stages (raster operations and tests), is shown in Figure B.2.

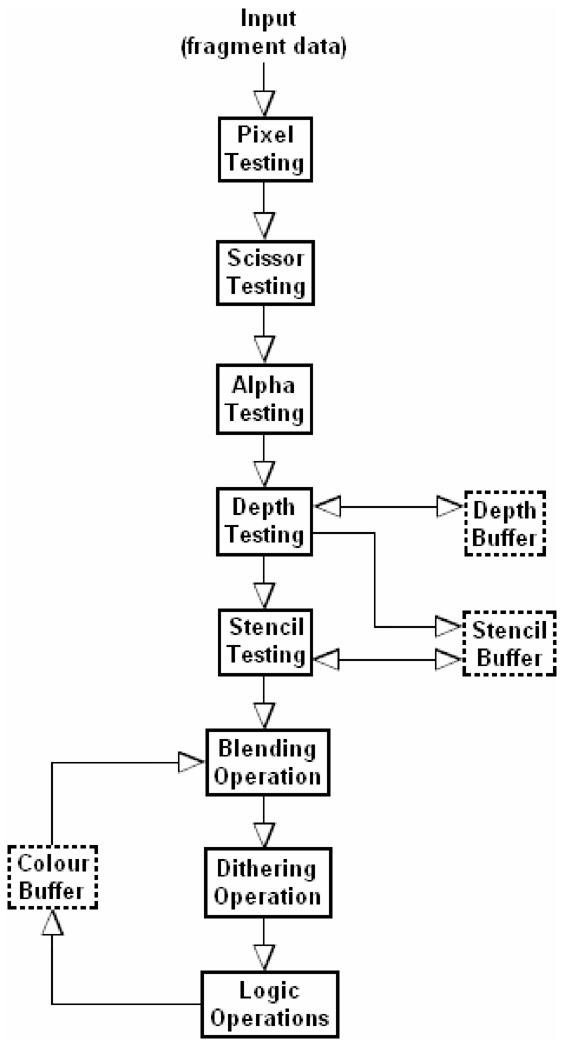

Figure B.2 Direct3D and OpenGL raster processing operations.

B.2 The Programmable Graphics Pipeline Revisited

This section extends the previous discussion of the Direct3D processing pipeline by investigating the underlying hardware configuration that makes the pipeline stages of a GPU programmable. Previous generation GPUs have separate vertex and pixel shader processing units. The GeForce 8 GPU (and better) does not follow this approach, rather offering eight shader units, with none of them limited to vertex or pixel processing. This architectural change is the product of recognising that the future of GPU design lies with programmable processing. By unifying the shaders we're not just only able to use the same instruction set for both pixel and vertex shaders or to enable workload sharing amongst these pipeline processors, but this new architecture also makes it easier to extend our current shader model with future shader types. As illustrated in Table B.1, GPU architecture has evolved from supporting configurable vertex and fragment processors, to programmable vertex processors, then fully programmable vertex and fragments processors to the current unified architecture. Extending the generic graphics hardware pipeline, we can show both vertex and fragment processing units as simple add-ons to this generic pipeline (Figure B.3).

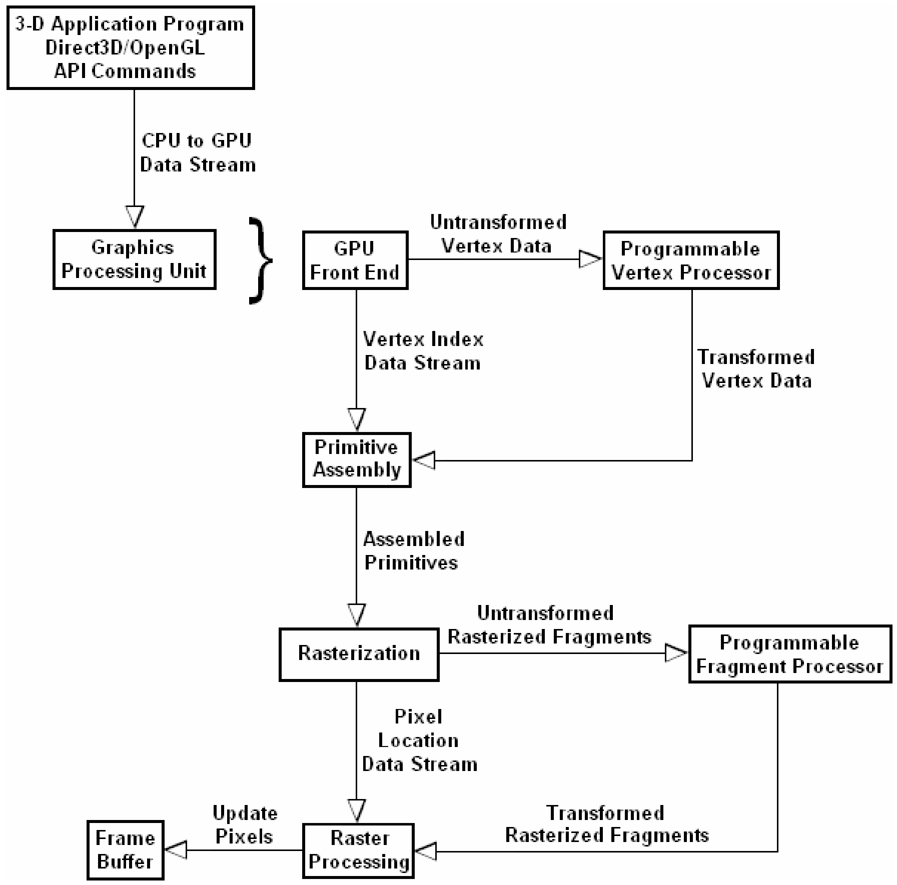

Figure B.3 Example of a hardware programmable graphics pipeline.

The unified shader architecture considered, vertex and fragment processing can still be broken down into logical programmable units; with a programmable vertex processor, the processing unit responsible for the execution of a HLSL, Cg or GLSL vertex program and a programmable fragment processor, the processing unit tasked with execution of a HLSL, Cg or GLSL fragment program.

Focussing on the programmable vertex processor, we can summarise its functionality into a number of stages. The first stage feeds vertex attributes such as coordinates, colour values and depth information into the vertex processor for processing. These vertex attributes are stored in the vertex attribute register banks. Vertex shaders actually make use of a several registers for the storage of position, data and colour data, for example. The vertex program, consisting of a sequence of instructions, is stored in memory. The vertex processor accesses this program, decoding one instruction at a time until the program terminates. Results generated from computations, the transformed vertex data, are stored in the output result registers with intermediate data, still being read by instructions, stored in the temporary register banks. Figure B.4 shows the classic flow of control for a programmable vertex processor.

Programmable fragment processors are extremely useful for manipulating texture coordinates as well as to set the final colour of a pixel. These processors also support several of the vector math operations performed by vertex processors. For example, a fragment processor can be programmed to read the texture coordinates of a textured image and to subsequently perform some operation on these values – returning a filtered sample of the texture. Similar to a vertex processor, fragment processors operate by executing a set of instructions stored in a program file – the fragment program. These instructions are executed until the fragment program terminates (when there aren't any more instructions to fetch). The fragment program reads untransformed interpolated fragments as input, storing these values in input register banks. Results generated from applying the specified instructions on input data are stored in the output registers. Intermediate data, just as with vertex processing, is stored in the temporary register banks. The output values can range from a fragment's new colour to a transformed depth value.

A texture is nothing more than a two-dimensional array consisting of colour values with each of these colour values referred to as a texel, or texture element. Each texel, being an element in this colour array, is thus assigned a unique address in the texture (simply a column and row value). Fragment processors generally include a *texture fetch instruction*. This instruction is used to compute the address of a texture, fetch texture elements, determine its Level-of-Detail and to perform texture filtering. Examples of texture filtering include nearest-point sampling, linear texture filtering, anisotropic texture filtering, bilinear filtering and filtering via mipmaps. Figure B.5 illustrates the flow of control for a typical programmable fragment processor.

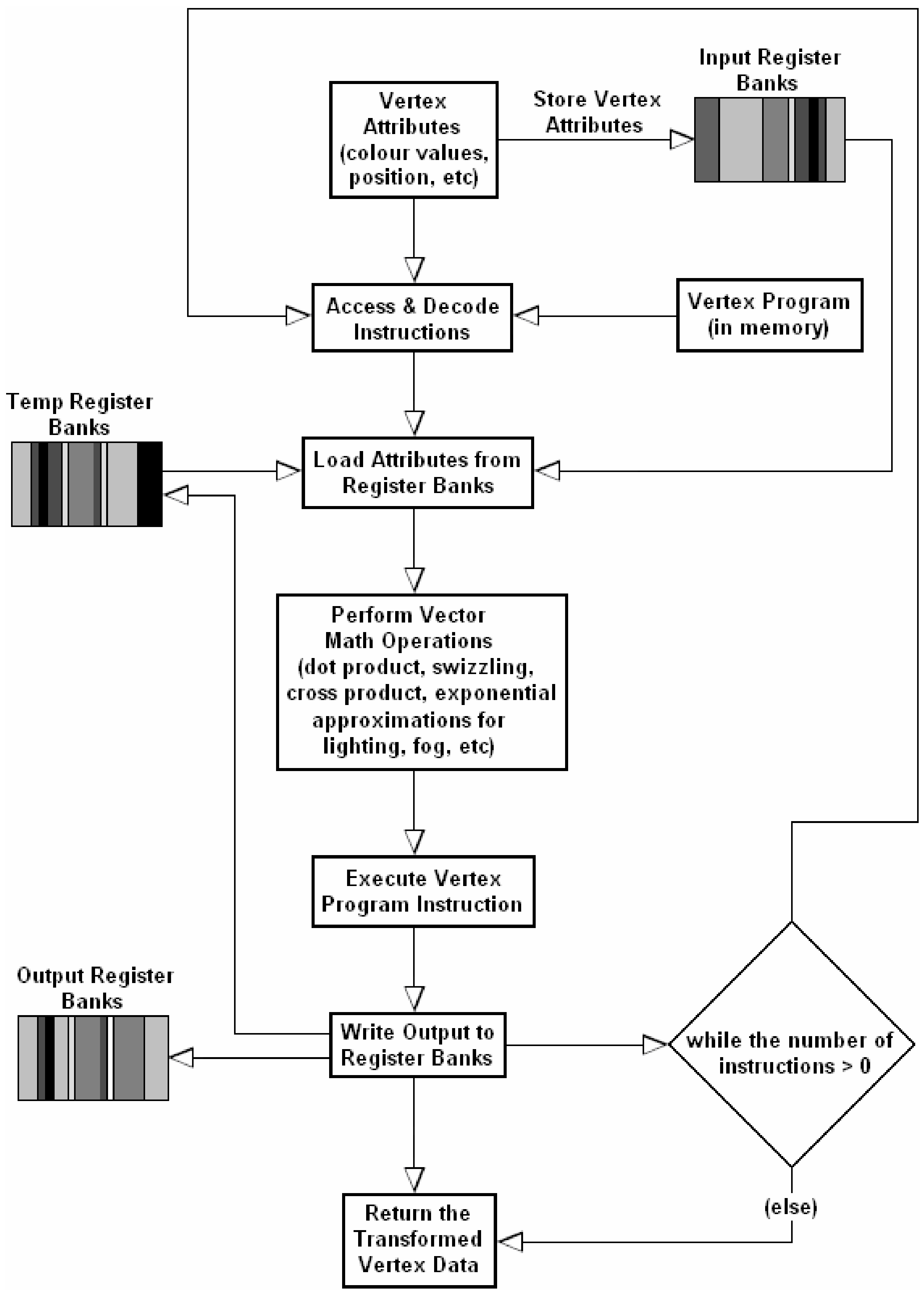
Input Register Banks
Vertex Attributes (colour values, position, etc)
Store Vertex Attributes
Access & Decode Instructions
Vertex Program (in memory)
Temp Register Banks
Load Attributes from Register Banks
Perform Vector Math Operations (dot product, swizzling, cross product, exponential approximations for lighting, fog, etc)
Execute Vertex Program Instruction
Output Register Banks
Write Output to Register Banks
while the number of instructions > 0
(else)
Return the Transformed Vertex Data

Figure B.4 Flow of control for a programmable vertex processor.

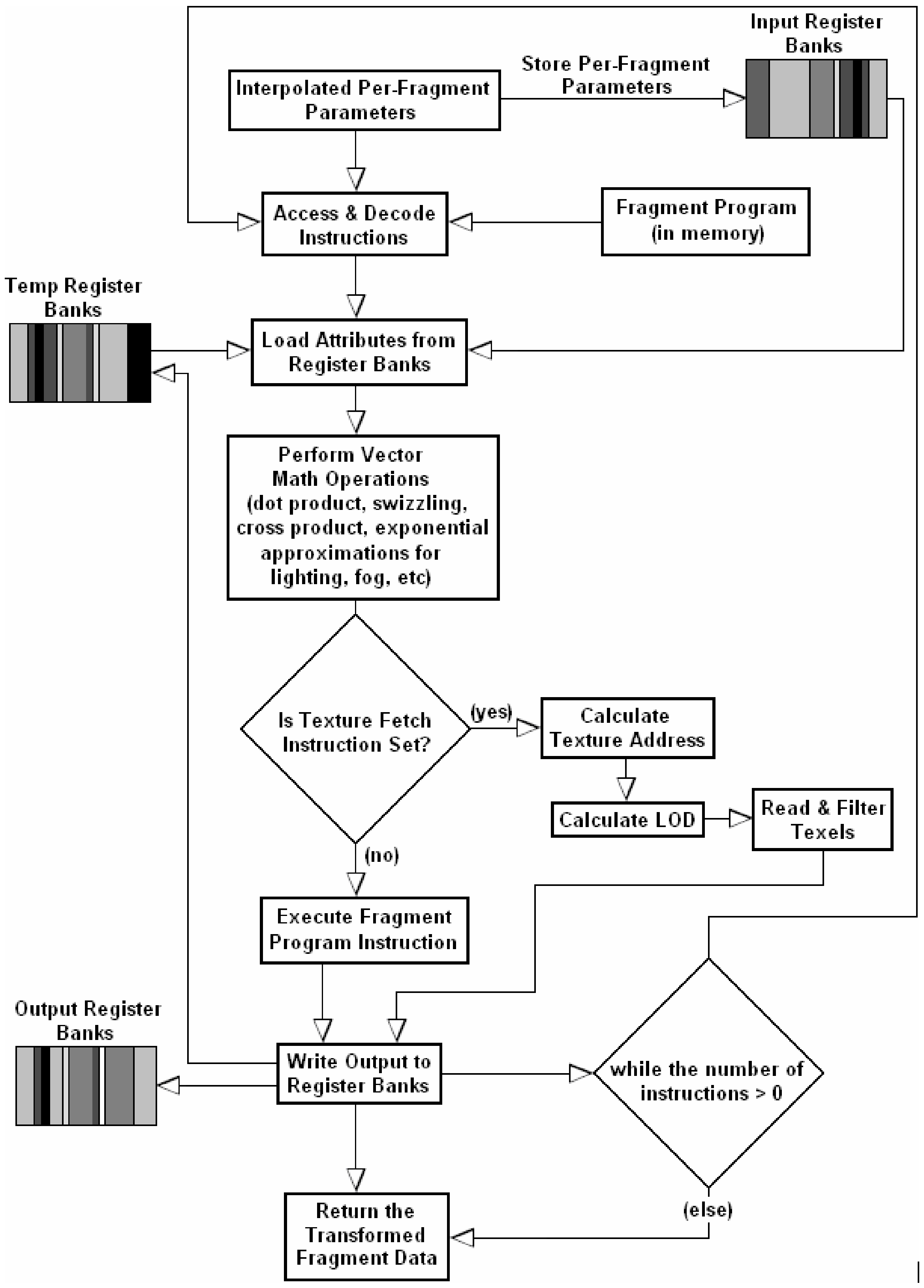
Input Register Banks
Store Per-Fragment Parameters
Interpolated Per-Fragment Parameters
Access & Decode Instructions
Fragment Program (in memory)
Temp Register Banks
Load Attributes from Register Banks
Perform Vector Math Operations (dot product, swizzling, cross product, exponential approximations for lighting, fog, etc)
Is Texture Fetch Instruction Set?
(yes)
Calculate Texture Address
Calculate LOD
Read & Filter Texels
(no)
Execute Fragment Program Instruction
Output Register Banks
Write Output to Register Banks
while the number of instructions > 0
(else)
Return the Transformed Fragment Data

Figure B.5 Flow of control for a programmable fragment processor.

B.3 High Level Shader Language (HLSL)

Microsoft's High Level Shader Language is a proprietary Direct3D shading language analogous to NIVIDIA's Cg. The Direct3D 10 High Level Shader Language allows for the creation of three types of shader programs, namely, vertex shaders, geometry shaders and pixel shaders. Similar to Cg, HLSL shaders can be compiled either statically or dynamically, depending on the preference of the developer and intended application for the shader.

As mentioned, Direct3D 10 shaders are unified to provide the application programmer with a uniform instruction set independent of whether a pixel, vertex or geometric shader is being implemented. These different shaders, offering the same core functionality, are implemented by the Shader Model 4.0 common shader core. Building on the core functionality, each shader implementation offers its own unique functionality such as stencilling done via pixel shaders or the generation of new primitives and the manipulation of 3-D models on a per-primitive basis by a geometric shader. This common shader core data-flow is shown in Figure B.6.

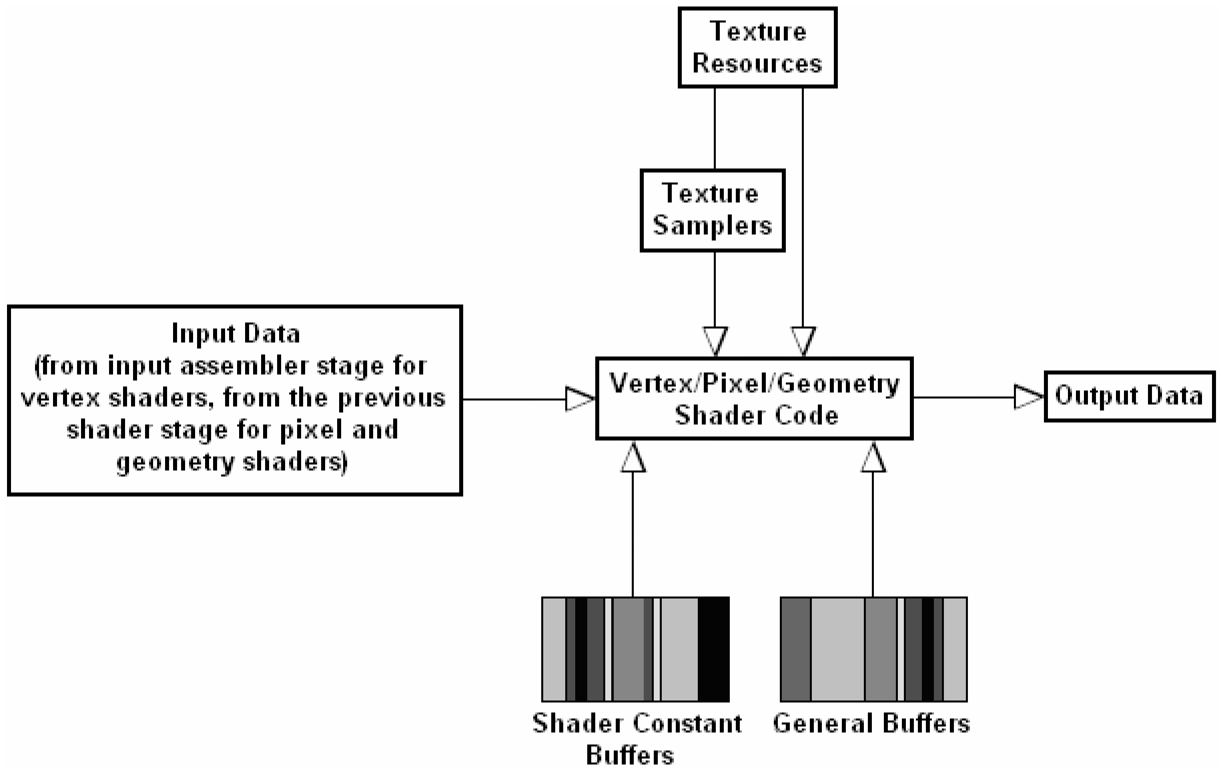
Texture Resources
Texture Samplers
Input Data (from input assembler stage for vertex shaders, from the previous shader stage for pixel and geometry shaders)
Vertex/Pixel/Geometry Shader Code
Output Data
Shader Constant Buffers
General Buffers

Figure B.6 Common shader core architecture.

The stages given in the above depicted data-flow model can be summarised as follows:

1) Input data is sent to the vertex, pixel or geometry shader for processing with the vertex shader receiving data from the input assembler stage and the pixel and geometry shaders receiving their input data from the previous shader stage.
2) The shaders can now perform some arithmetic or flow control operation on the read data.
    a. Texel data is either directly read without any filtering or sampling using the **`Load`** HLSL function or alternatively filtered and sampled by binding up to 16 HLSL samplers to the shader.
    b. General buffers are also accessed from system memory, allowing the shader program to bind up to 128 texture elements and buffer resources to the shader.
    c. Shader constant buffers can also be bound to a shader stage. These buffers are frequently updated by the CPU and are larger in size and layout than the general buffers.
3) The output generated by the shader code is passed to the next stage in the graphics pipeline.

B.3.1 The HLSL Compiler

HLSL programs have to be compiled into a GPU executable form. Compilation is based on the translation of a vertex, pixel or geometry shader program into a form readable by Direct3D. This translation of the original HLSL program is sent to the Direct3D API driver which converts it to instructions that can be processed by the GPU.

We can perform static compilation using the FXC shader compiler (fxc.exe) to compile our shader program once and thus eliminating the need to compile it again. The FXC HLSL compiler is invoked with its executable name followed by one or more options, the shader model profile label and the filename. For example, to compile a shader program saved in the file shader.fx, we can do a release build for shader model 4.0 as follows:

```
fxc /T fx_4_0 /Fo shader.fxo shader.fx
```

In this example **`fx_4_0`** specifies the target profile as a shader model 4.0 effect (shader model 2.0 effects are set using the **`fx_2_0`** profile). An effect shader can contain a combination of pixel, vertex and geometry shaders. Alternatively we could have specified the shader type as a vertex shader, pixel shader or texture shader (**`tx_1_0`**). These HLSL shader profiles are used to compile a shader to a specific shader model, thus ensuring hardware compatibility by limiting the supported shader model feature set. Possible Direct3D 10 vertex shader profiles include **`vs_1_1`**, **`vs_2_0`**, **`vs_2_a`**, **`vs_2_sw`**, **`vs_3_0`**, **`vs_3_sw`** and **`vs_4_0`** with pixel shader profiles ranging from **`ps_2_0`**, **`ps_2_a`**, **`ps_2_b`**, **`ps_2_sw`**, **`ps_3_0`** and **`ps_3_sw`** to **`ps_4_0`**. The '**`/T`**' switch option specifies the HLSL profile to compile against. The **D3D10GetVertexShaderProfile**, **D3D10GetPixelShaderProfile** and **D3D10GetGeometryShaderProfile** shader functions can be called to determine the best profile suited for a given device to compile against. These functions all take a pointer to an **ID3D10Device** interface device, returning either the best vertex shader profile, pixel shader profile or geometry shader profile depending on the function called. Shader functions can be used after including the D3D10Shader.h header file. The next switch option, '**/Fo**', is used to set the output object file name used to store the compiled shader effect.

We can alternatively compile a shader using debug mode. Debug mode is similar to that found in Visual Studio, allowing the generation of debug information and additional processing data that can be used to narrow down errors and possible bottleneck areas. We can compile the shader program saved in the shader.fx file using debug mode in the following manner:

```
fxc /Zi /Od /T fx_4_0 /Fo shader.fxo shader.fx
```

The '**/Zi**' switch option enables debugging information with the '**/Od**' switch disabling any code-based optimisations that would normally be performed by the compiler.

B.3.2 Initialising the High Level Shader Language

This section focuses on the initialisation of the High Level Shader Language so that a Direct3D application program can bind the shader program to the appropriate pipeline stage. The steps of this initialisation process are as follows:

1) Compilation of the shader to ensure that the HLSL statements are syntactically correct.
2) Create a vertex, pixel or geometry shader object.
3) Set the created shader object to bind the shader to the proper pipeline stage.

A shader program is compiled by calling the **D3D10CompileShader** shader function, declared as follows in the D3D10Shader.h header file:

```
HRESULT D3D10CompileShader(
  LPCSTR pSrcData,
  SIZE_T SrcDataLen,
  LPCSTR pFileName,
  CONST D3D10_SHADER_MACRO *pDefines,
  LPD3D10INCLUDE *pInclude,
  LPCSTR pFunctionName,
  LPCSTR pProfile,
  UINT Flags,
  ID3D10Blob **ppShader,
  ID3D10Blob **ppErrorMsgs
);
```

Its first parameter, **pSrcData**, takes a pointer to the string holding the shader source code. The second parameter, **SrcDataLen**, specifies the size of the **pSrcData** parameter in bytes with the next parameter, **pFileName**, the name of the shader program file. The **pDefines** parameter takes a pointer to a **D3D10_SHADER_MACRO** shader macro array. This Null-terminated array of macro definitions, enabling the application program to define tokens at runtime, is optional and can be set to 'NULL'. A **D3D10_SHADER_MACRO** macro definition can be specified in the following manner:

```
D3D10_SHADER_MACRO Macro[1] = {"ten", "10"};
```

The **D3D10_SHADER_MACRO** shader structure has two members, **Name** and **Definition**. The **Name** member holds the macro name and the **Definition** member the macro definition.

D3D10CompileShader's next parameter, **pInclude**, takes a pointer to the **ID3D10Include** interface allowing the opening and closing of included files when loading an effect from memory. For example, a shader program can include a file using the **#include** directive, and by calling the **Close** or **Open** **ID3D10Include** members we can open this file for reading and subsequently close it when done. Specification of the **pInclude** parameter is optional and set to 'NULL' when the shader does not contain any **#include** directives. The next parameter, **pFunctionName**, takes a pointer to a string holding the shader entry point function name indicating the function to begin the shader execution at. The **pProfile** parameter is used for setting the shader model profile with the **Flags** parameter setting the shader compile options (possible options are listed in Table B.2). The first of the final two parameters, **ppShader**, takes a pointer to an **ID3D10Blob** interface containing the debug information and compiled shader. A *blob* is a data buffer used for the storage of vertex, adjacency and material data. Blobs also return error/debug messages and object code during the compilation of pixel, vertex and geometry shaders. The last parameter, **ppErrorMsgs**, also takes a pointer to an **ID3D10Blob** interface, this time one containing errors and warning messages generated during the compilation process.

Compile Options	Description
`D3D10_SHADER_AVOID_FLOW_CONTROL`	The HLSL compiler will disable flow control as far possible.
`D3D10_SHADER_DEBUG`	The HLSL compiler enables the generation of debug information.
`D3D10_SHADER_ENABLE_BACKWARDS_ COMPATIBILITY`	The HLSL compiler will compile older shaders to the shader model 4.0 spec.
`D3D10_SHADER_ENABLE_STRICTNESS`	The HLSL compiler enables strictness on deprecated shader syntax.
`D3D10_SHADER_FORCE_PS_SOFTWARE_ NO_OPT`	The HLSL compiler will compile a pixel shader to the next best shader profile, enabling debugging and disabling compiler optimisations.
`D3D10_SHADER_FORCE_VS_SOFTWARE_ NO_OPT`	The HLSL compiler will compile a vertex shader to the next best shader profile, enabling debugging and disabling compiler optimisations.
`D3D10_SHADER_IEEE_STRICTNESS`	The HLSL compiler enables IEEE strictness – thus conforming to a pre-defined set of standards.
`D3D10_SHADER_NO_PRESHADER`	The HLSL compiler disables the use of preshaders – an optimisation where constant expressions are replaced with references to the GPU's registers and memory addresses.
`D3D10_SHADER_OPTIMIZATION_ LEVEL0`	The HLSL compiler enables level 0 warnings.
`D3D10_SHADER_OPTIMIZATION_ LEVEL1`	The HLSL compiler enables level 1 warnings.
`D3D10_SHADER_OPTIMIZATION_ LEVEL2`	The HLSL compiler enables level 2 warnings.
`D3D10_SHADER_OPTIMIZATION_ LEVEL3`	The HLSL compiler enables level 3 warnings.
`D3D10_SHADER_PACK_MATRIX_COLUMN_MAJOR`	The HLSL compiler packs the matrixes in column-major order – leading to more efficiency since matrix manipulations can be performed via a series of dot-products.
`D3D10_SHADER_PACK_MATRIX_ROW_ MAJOR`	The HLSL compiler packs the matrixes in row-major order.
`D3D10_SHADER_PARTIAL_PRECISION`	The compiler sets all calculations to be done with partial precision which will lead to some performance gains.
`D3D10_SHADER_PREFER_FLOW_ CONTROL`	The HLSL compiler will enable flow control as far possible.
`D3D10_SHADER_SKIP_OPTIMIZATION`	The HLSL compiler will disable optimisations.

D3D10_SHADER_SKIP_VALIDATION	The HLSL compiler will disable the validation of code against common constraints and capability limits.

Table B.2 HLSL compile options.

Before calling the **D3D10CompileShader** shader function, we first have to create an **ID3D10Blob** interface:

```
IPD3D10Blob * pShaderBlob;
```

We can, for instance, compile a vertex shader stored in the file vertex_shader.vsh as follows:

```
D3D10CompileShader(strPath, strlen(strPath),
                   "vertex_shader.vsh", NULL, NULL, "EffectFunctionName", "vs_4_0",
                   D3D10_SHADER_ENABLE_STRICTNESS, &pShaderBlob, NULL);
```

The shader function, **EffectFunctionName**, could have been declared in the shader program like this (taking one input parameter and returning a vertex shader structure. The declaration of shader functions, their basic, vector, texture, struct and matrix data types as well as sampler type syntax are all dealt with in the next section):

```
VS_OUTPUT EffectFunctionName (in float2 vertexPosition : POSITION)
```

A pointer to the compiled shader code is returned via the **pShaderBlob ID3D10Blob** interface. This pointer is used to create the vertex shader object using the **CreateVertexShader** function (for this example) or alternatively **CreatePixelShader** to create a pixel shader object or the **CreateGeometryShader ID3D10Device** interface function for geometry shaders. The **CreateVertexShader** function is declared in the D3D10.h header file as follows:

```
HRESULT CreateVertexShader(const void *pShaderBytecode, SIZE_T BytecodeLength,
                           ID3D10VertexShader **ppVertexShader);
```

Its first parameter, **pShaderBytecode**, takes a pointer to the compiled shader retrieved using the **GetBufferPointer ID3D10Blob** interface function. The **BytecodeLength** parameter takes the size of the compiled shader determined via the **GetBufferSize ID3D10Blob** interface function. The final parameter, **ppVertexShader**, is the address of a pointer to an **ID3D10VertexShader** interface.

The **CreateGeometryShader** and the **CreatePixelShader ID3D10Device** interface functions have the same first two parameters as **CreateVertexShader**. These functions only differ in respect to the last parameter which takes a pointer to an

ID3D10PixelShader interface in the case of the **CreatePixelShader** function and an **ID3D10GeometryShader** interface for the **CreateGeometryShader** function.

Continuing with our vertex shader program, before calling the **CreateVertexShader** function, we specify a shader object by first declaring an **ID3D10VertexShader** interface:

```
ID3D10VertexShader **ppOurVertexShader;
```

We create the vertex shader object using the **CreateVertexShader** function (using the previously declared **ID3D10Device*** interface, **g_id3dDevice**):

```
hresult_ = g_id3dDevice->CreateVertexShader((DWORD*)pShaderBlob->GetBufferPointer(),
                                             pShaderBlob->GetBufferSize(),
                                             &ppOurVertexShader);
```

We must also remember to release the pointer to the compiled shader source:

```
pShaderBlob->Release();
```

The final step requires us to set this newly created shader object to the pipeline stage. To set the vertex shader to the device, we call the **VSSetShader ID3D10Device** interface function. This function takes one parameter, namely, a pointer to the **ID3D10VertexShader** vertex shader:

```
g_id3dDevice->VSSetShader(pOurVertexShader);
```

The vertex shader stage is now initialised with the compiled vertex shader code. To initialise the pixel shader stage we need to call the **PSSetShader ID3D10Device** interface function (using an **ID3D10PixelShader** interface as parameter). The **GSSetShader ID3D10Device** interface function is called for setting a geometry shader to a device (using an **ID3D10GeometryShader** interface as parameter).

B.3.3 Creating HLSL shaders

Pixel, vertex and geometry shaders each make out a different stage of the Direct3D 10 programmable pipeline. These shaders, operating on input data and sending their results to subsequent pipeline stages, are created in the form of program files that can be compiled and executed on the GPU. To recap, vertex shaders operate on vertex data with pixel shaders reading fragments (pixels) as input and geometry shaders processing primitives as input.

Vertex shaders process a vertex read as input and generates some output in the form of a transformed vertex. Vertex data are passed to the GPU via a vertex buffer. Each vertex element stored in this vertex buffer is then sent to the vertex shader for processing. For example, the following vertex shader function returns its input data as output without doing any processing on it:

```
float4 VertexShader(float4 Position : POSITION) : SV_POSITION
{
     return Position;
}
```

The vertex shader function, labelled **VertexShader**, with the return type **float4** takes a parameter, **Position**, of type **float4** as input – **float4** being a four-component HLSL vector type with each of its vector components a floating-point value. As with Cg the declaration of the input and output parameters are followed by a colon and binding semantic to further describe the data type. The input parameter is set to the **POSITION** semantic (the input vertex's clip-space coordinates) with the output value semantic set to **SV_POSITION**. Semantics using the '**SV_**' prefix are referred to as system-value semantics meaning they are system generated values and can be used for both input and output data. The **SV_POSITION** semantic are, for example, processed during the rasterization stage and in this case used to notify the graphics pipeline that the output data will also be in the form of clip-space coordinates.

We can now create a pixel shader function to take the output produced by the above defined vertex shader function as input (a **float4** type coupled with the **SV_POSITION** semantic). This pixel shader then returns an output colour (red) using the **SV_TARGET** semantic that denotes the output as a render target format:

```
float4 PixelShader(float4 Position : SV_POSITION): SV_TARGET
{
    return float4(1.0f, 0.0f, 0.0f, 1.0f); //red
}
```

The next step is to specify an effect technique definition used for setting the previously defined vertex and pixel shaders. Such an effect technique, starting with the syntax, **technique10** to label it as a Direct3D 10 technique, is a set of rendering passes. Each rendering pass specifies the shader states used to render the geometry of a scene. An effect is thus a way for Direct3D to organise the states responsible for setting the stages of the graphics pipeline. The **technique10** label is followed by the name of the technique, **TechniqueName** and the name of the rendering pass, **P0**, containing the callback function(s) such as **SetPixelShader**, **SetVertexShader** or **SetGeometryShader** used to set the device state from an effect. Other states that can be set include the blend state (**SetBlendState**) and depth-stencil state (**SetDepthStencilState**). We can create the following effect technique for the above defined vertex and fragment shaders:

```
technique10 TechniqueName
{
    pass P0
    {
          SetGeometryShader(NULL);
          SetVertexShader(CompileShader(ps_4_0, VertexShader()));
          SetPixelShader(CompileShader(ps_4_0, PixelShader()));
     }
}
```

The **SetPixelShader**, **SetVertexShader** and **SetGeometryShader** functions take a compiled shader as parameter, setting it to the appropriate render state. The geometry shader is in this case set to 'NULL' because it has not yet been defined. The vertex and pixel shaders, as well as the effect technique, are stored in an effect file (using the '.fx' file extension).

Returning to our Direct3D application, all that remains is to create the effect object and technique object that will be used for performing the rendering operation. We call the **D3DX10CreateEffectFromFile** function to create an effect from the specified effect file. This D3DX function is specified as follows in the D3DX10Effect.h header file:

```
HRESULT D3DX10CreateEffectFromFile(
  LPCTSTR pFileName,
  CONST D3D10_SHADER_MACRO *pDefines,
  ID3D10Include *pInclude,
  LPCSTR pProfile,
  UINT HLSLFlags,
  UINT FXFlags,
  ID3D10Device *pDevice,
  ID3D10EffectPool *pEffectPool,
  ID3DX10ThreadPump *pPump,
  ID3D10Effect **ppEffect,
  ID3D10Blob **ppErrors
);
```

This function's first parameter, **pFileName**, takes a pointer to a string containing the name of the effect file. The next parameter, **pDefines**, takes a pointer to a **D3D10_SHADER_MACRO** shader macro array with the **pInclude** parameter requiring a pointer to an **ID3D10Include** include interface as previously described. The shader profile, as a string value, is set via the **pProfile** parameter with the HLSL compilation options being set by the **HLSLFlags** parameter. The sixth parameter, **FXFlags**, allows us to set the effect compilation options and it can be set to any of the following **D3D10_EFFECT** constants: **D3D10_EFFECT_COMPILE_CHILD_EFFECT** (the '.fx' file is compiled as a child effect, thus not initialising any shared data due to all shared values being set in the effect pool), **D3D10_EFFECT_COMPILE_ALLOW_SLOW_OPS** (compiles the effect file without performance mode) or **D3D10_EFFECT_SINGLE_THREADED** (the effect thread is not synchronised with other effects in the effect pool). An *effect pool* facilitates the sharing of variables, textures and shaders between different effects. The next parameter, **pDevice**, takes a pointer to an **ID3D10Device** interface that will use the resources to create a shader. The **pEffectPool** parameter takes a pointer to an **ID3D10EffectPool** effect pool interface signifying the memory pool used for the sharing of variables and resources between effects. The next parameter, **pPump**, is a pointer to an **ID3DX10ThreadPump** thread pump interface used for the asynchronous execution of routines; we will generally set this parameter to 'NULL' so that the **D3DX10CreateEffectFromFile** function completes its operation before returning. The second last parameter, **ppEffect**, takes the address of the pointer to the **ID3D10Effect** created effect. The **ID3D10Effect** interface is responsible for managing the shaders, state objects and resources constituting the effect. The **ppErrors** parameter is set to the address of a pointer to an **ID3D10Blob** interface. This final parameter is used for storing debug and compile-time error information.

We can create the effect using this **D3DX10CreateEffectFromFile** function in the following manner:

```
ID3D10Effect* g_id3dEffect = NULL;

D3DX10CreateEffectFromFile(L"file_name.fx", NULL, NULL,
                           D3D10_SHADER_ENABLE_BACKWARDS_COMPATIBILITY,
                           0, g_id3dDevice, NULL, NULL, &g_id3dEffect, NULL);
```

Following the effect creation we must obtain the effect technique using the **GetTechniqueByName** **ID3D10Effect** interface function. This function takes a string value containing the name of the technique as parameter, returning a pointer to the **ID3D10EffectTechnique** interface:

```
ID3D10EffectTechnique* g_id3dTechnique = NULL;

g_id3dTechnique = g_id3dEffect->GetTechniqueByName("TechniqueName");
```

A useful feature of effects is the ability to define multiple passes (subsets of a technique and a render state set – for example '**P0**' in the above shown technique). We can thus define multiple passes to implement multi-pass rendering. To understand multi-pass rendering, consider the following example. Say we have a geometry object with a texture and we decide to render some three-dimensional mesh on top of it, then we can render and texture the geometry in the first pass with the second pass being responsible for rendering the mesh on top of it. By specifying each phase as a render pass we can render both passes simultaneously during the render loop. Techniques are also useful when designing a shader to run across a vast range of hardware, for example, a technique can be specified using a pixel, vertex and geometry shader for the newest DirectX 10 hardware while another can be specified to limit the implementation to only vertex and pixel shaders so that the program can run on DirectX 9 hardware.

B.3.4 Common HLSL Data Types

HLSL features all the C++ derived scalar types such as **`bool`**, **`int`**, **`float`**, **`string`**, **`double`**, **`uint`** and **`half`** (a 16-bit floating point type). Shader Model 4.0 features two additional types derived from the **`float`** type, namely, **`unorm float`** (a 32-bit unsigned floating point value normalised to the range [-1, 1]) and **`snorm float`** (a 32-bit unsigned floating point value normalised to the range [0, 1]).

HLSL also allows for the use of vector and matrix types. Vector types can contain anything from one to four components with matrix types containing up to sixteen components. Matrix types are declared using the form **`ScalartypeRowxColumn`**, for example, a floating point matrix, **`fMatrixVar`**, consisting out of four rows and three columns can be declared as follows:

```
float4x3 fMatrixVar;
```

This matrix variable can be initialised in the following manner:

```
fMatrixVar = {1.5f, 5.5f, 0.1f,
              0.4f, 0.1f, 2.7f,
              0.3f, 2.6f, 0.2f,
              0.9f, 0.5f, 4.2f };
```

Matrix types can also be declared using the following syntax:

```
matrix <scalar type, number of rows, number of columns> MatrixVariableName
```

We can create the same matrix as the **`fMatrixVar`** one defined above using this alternate syntax:

```
matrix <float, 4, 3> fMatrixVar = {1.5f, 5.5f, 0.1f,
                                   0.4f, 0.1f, 2.7f,
                                  0.3f, 2.6f, 0.2f,
                                  0.9f, 0.5f, 4.2f };
```

Vector types are declared using the syntax **`Scalartype VectorVariableName`**, for example, a floating point vector holding four components can be declared in the following manner:

```
float4 fVectorVar = {1.5f, 1.7f, 0.5f, 1.0f};
```

There is also, as with matrix types, an alternative syntax for the declaration of vector types:

```
vector <vector type, number of components> VectorVariableName
```

We can create the same vector, **`fVectorVar`**, using this alternate syntax:

```
vector <float, 4> fVectorVar = {1.5f, 1.7f, 0.5f, 1.0f};
```

HLSL also allows for the definition of structures in the following manner:

```
struct structName
{
   float variable1;
   int variable2;

   float4 fVectorVar = {1.5f, 1.7f, 0.5f, 1.0f};
   matrix <float, 4, 3> fMatrixVar = {1.5f, 5.5f, 0.1f,
                                      0.4f, 0.1f, 2.7f,
                                      0.3f, 2.6f, 0.2f,
                                      0.9f, 0.5f, 4.2f };
   ...etc
}
```

The HLSL further supports a number of operators clearly inherited from the C programming language. The most commonly used ones are listed in Table B.3.

Operator Type	**Operators**	**Usage Examples**

Additive	`+, -`	`int x = 5;`
		`int y = 7;`
		`int z = x – y;`
		`int k = z + y;`

Multiplicative	`*, %, /`	`int x = 5;`
		`int y = 7;`

		`int z = x * y;`
		`float k = z / y;`
		`int l = z % y;`

Array Selection	`[i]`	`int array[2] = {3,4};`
		`array[0] = 2;`

Assignment	`+=, =, *=, -=, %=, /=`	`int x = 5;`
		`int y = 7;`
		`int z += 3;`

Bitwise	`~, &, |, ^, <<, >>, <<=, |=, >>=, &=, ^=`	`z>>y //shifts the bits of z right y positions (5 >> 2 equals 1)`

Boolean	`||, &&, ?:`	`bool a = false;`
		`bool b = true;`
		`bool c = a && b;`

Comparison	`==, !=, <, >, <=, >=`	`if (diffuseLight <= 0)`
		`    specularLight = 0;`

Prefix/Postfix Incrementing/ Decrementing	`++, --`	`int x = 0;`
		`x++;`
		`--x;`

Type Cast	`(scalar type)`	`float x = 0.5;`
		`int y;`
		`y = (int)x;`

Unary	`+, -, !`	`bool a = true;`
		`bool b = !a;`

Table B.3 HLSL Operators

B.3.5 Utilising a Created HLSL Effect

After compiling and creating an effect by loading the effect file into the effects framework (using the **`D3DX10CreateEffectFromFile`** function), we can proceed to initialise a number of effect constants before setting the effect state. Effects that have not yet been compiled will be compiled when they are loaded into the effects framework. Effect constants and variables are first declared in the effect/shader file(s), for example:

```
int numberOfLightSources;

float3 incomingAmbientLightColour[3];
float4 incomingDiffuseLightColour[3];
float3 objectspaceLightPosition[3];

float4x4 modelviewProjection;
float4x4 worldviewProjection;
```

```
Texture2D meshTexture;
```

These variables, declared using the HLSL data types, are set by the Direct3D application. We must thus declare variables in our application that will be used to update the shader variables:

```
int numberOfLights;

D3DXVECTOR3 vIncomingAmbientLightColour [3];
D3DXVECTOR4 vIncomingDiffuseLightColour [3];
D3DXVECTOR3 vObjectspaceLightPosition [3];

D3DXMATRIX mWorldviewProjectionMatrix;
D3DXMATRIX mModelviewProjectionMatrix;
```

Before we can set the HLSL variable values using the **`ID3D10EffectVariable`** update methods we first have to obtain the effect variables via **`ID3D10Effect`** retrieval functions for each of the above defined shader variables (this operation is similar to the retrieval of technique objects):

```
ID3D10EffectScalarVariable* g_pNumberOfLightSources;
g_pNumberOfLightSources = g_id3dEffect
                 ->GetVariableByName("numberOfLightSources")->AsScalar();

ID3D10EffectVectorVariable* g_pIncomingAmbientLightColour;
g_pIncomingAmbientLightColour = g_id3dEffect
         ->GetVariableByName("incomingAmbientLightColour")->AsVector();

ID3D10EffectVectorVariable* g_pIncomingDiffuseLightColour;
g_pIncomingDiffuseLightColour = g_id3dEffect
         ->GetVariableByName("incomingDiffuseLightColour")->AsVector();

ID3D10EffectVectorVariable* g_pObjectspaceLightPosition;
g_pObjectspaceLightPosition = g_id3dEffect
         ->GetVariableByName("objectspaceLightPosition")->AsVector();

ID3D10EffectMatrixVariable* g_pWorldviewProjectionMatrix;
g_pWorldviewProjectionMatrix = g_id3dEffect
                 ->GetVariableByName("worldviewProjection")->AsMatrix();

ID3D10EffectMatrixVariable* g_pModelviewProjectionMatrix;
g_pModelviewProjectionMatrix = g_id3dEffect
```

```
                ->GetVariableByName("modelviewProjection")->AsMatrix();

ID3D10EffectShaderResourceVariable* g_pMeshTexture;
g_pMeshTexture = g_id3dEffect
                ->GetVariableByName("meshTexture")->AsShaderResource();
```

The **GetVariableByName ID3D10Effect** interface function takes a string value containing the name of the variable declared in the shader/effect program as parameter, returning a pointer to the **ID3D10EffectVariable** interface. The **AsVector ID3D10EffectVariable** interface function casts this returned **ID3D10EffectVariable** interface to an **ID3D10EffectVectorVariable** interface so that we can access the vector type. The **AsScalar** interface function casts the returned interface to an **ID3D10EffectScalarVariable** interface used for accessing a scalar variable with the **AsMatrix** function casting it to an **ID3D10EffectMatrixVariable** interface so that we can read the shader variable as a matrix type.

Other frequently used **ID3D10EffectVariable** interface casting methods include: **AsBlend** (casts to an **ID3D10EffectBlendVariable** interface used for accessing blend-state variables), **AsDepthStencil** (casts to an **ID3D10EffectDepthStencilVariable** interface used for accessing depth-stencil variables), **AsRasterizer** (casts to an **ID3D10EffectRasterizerVariable** interface used for accessing rasterizer-state variables), **AsShader** (casts to an **ID3D10EffectShaderVariable** interface used for accessing shader variables), **AsShaderResource** (casts to an **ID3D10EffectShaderResourceVariable** interface used for accessing shader-resource variables) and **AsString** (casts to an **ID3D10EffectStringVariable** interface used for accessing string variables).

We can now set the values of the shader/effect variables using the following **ID3D10EffectVariable**, **ID3D10EffectVectorVariable**, **ID3D10EffectMatrixVariable** and **ID3D10EffectScalarVariable** methods: **SetRawValue** for generic array items, **SetFloatVectorArray** for four-component vector arrays containing floating point elements, **SetBoolVectorArray** for four-component vector arrays containing Boolean elements, **SetIntVector** for four-component vectors containing integer elements, **SetIntVectorArray** for four-component vector arrays containing integer elements, **SetMatrix** for a floating-point matrix, **SetMatrixArray** for an array of floating-point matrices, **SetFloat** for normal floating-point variables and **SetInt** for integer variables:

```
g_pNumberOfLightSources->SetInt(numberOfLights);


g_pIncomingAmbientLightColour->SetRawValue(vIncomingAmbientLightColour, 0,
                                          sizeof(D3DXVECTOR3) * 3);
```

```
g_pIncomingDiffuseLightColour->SetFloatVectorArray((float*)vIncomingDiffuseLightColour,
                                                    0, 3);

g_pObjectspaceLightPosition->SetFloatVectorArray((float*)vObjectspaceLightPosition,
                                                  0, 3);

g_pWorldviewProjectionMatrix->SetMatrix((float*)&mWorldviewProjectionMatrix));

g_pModelviewProjectionMatrix->SetMatrix((float*)&mModelviewProjectionMatrix));
```

The **`SetInt`** **`ID3D10EffectScalarVariable`** function takes a pointer to an integer variable as parameter. The **`SetRawValue`** **`ID3D10EffectVariable`** function has three parameters, the first taking a pointer to the variable being set, the second specifying the offset in bytes from the beginning of the input data being set and the third the number of bytes to set from the offset value. The **`ID3D10EffectVectorVariable`** **`SetFloatVectorArray`** method also takes three parameters as input, namely, a pointer to the first element of a vector array, the vector offset from the start of the array to the first vector that is to be set and the number of array elements, in that order. The **`SetMatrix`** **`ID3D10EffectMatrixVariable`** interface function sets a floating-point matrix and is passed a pointer to the first element of a matrix as parameter.

That is it, the values declared in the shader program are now set and can be changed during each rendering pass. The final step is to set the effect state within the device itself. This is done by invoking the effect state from within the render loop by selecting a technique and subsequently setting the state for each of the passes:

We start by calling the **`GetDesc`** **`ID3D10EffectTechnique`** function on the previously defined technique object which is used for storing the returned **`D3D10_TECHNIQUE_DESC`** structure, i.e. the structure describing the technique:

```
ID3D10EffectTechnique* g_pd3d10EffectTechnique = NULL;

/*   obtain the D3D10_TECHNIQUE_DESC effect-variable  description */
D3D10_TECHNIQUE_DESC technique;
g_pd3d10EffectTechnique->GetDesc(&technique);
```

The **`GetPassByIndex`** **`ID3D10EffectTechnique`** interface method is now called to acquire an effect pass object representing the first pass of the technique:

```
/* apply the effect state by looping over the number of technique passes */
for(int i = 0; i < technique.Passes; ++i)
{
        g_pd3d10EffectTechnique->GetPassByIndex(i)->Apply(0);
        ...etc
}
```

Appendix C
Lighting and Reflection

C.1 Lighting

Before considering shadows, it is very important to briefly discuss the concepts of lighting and reflection (as there can be no shadows without light). The lack of lighting results in dull, flat looking object surfaces. Texture mapping helps to enhance the overall appearance of an object but fails to convey any real sense of depth. For example, when looking at the two flat objects in Figure C.1 (a), it is clear that the three-dimensional nature of the scene, a wall positioned perpendicular on a floor, isn't being conveyed properly. Figure C.1 (b) shows this same scene illuminated by a properly defined light source.

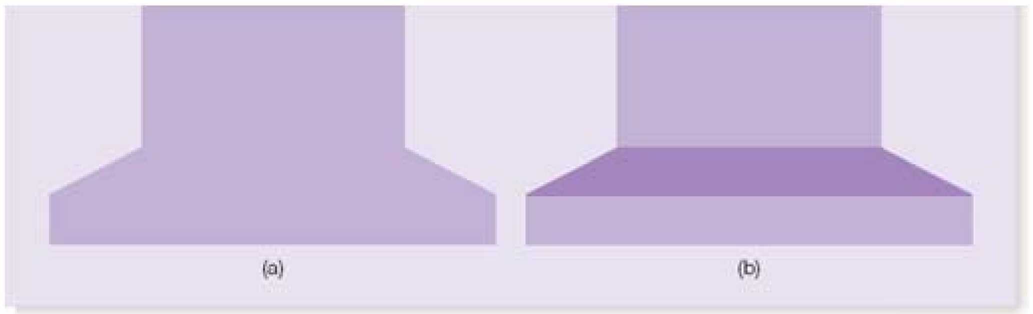

Figure C.1 (a) Two rendered rectangles, the one representing a floor, the other a facing wall. (b) The same rectangles with lighting enabled.

This lack of depth is the result of uniform lighting, i.e. the equal illumination of all surfaces. Figure C.2 (a) shows a uniform lit sphere and Figure C.2 (b) the same sphere with basic lighting enabled. The shaded sphere is the result of graduations in the sphere's colour based on the colour of the light source. In this case the colour grey is incrementally decreased from dark grey to white.

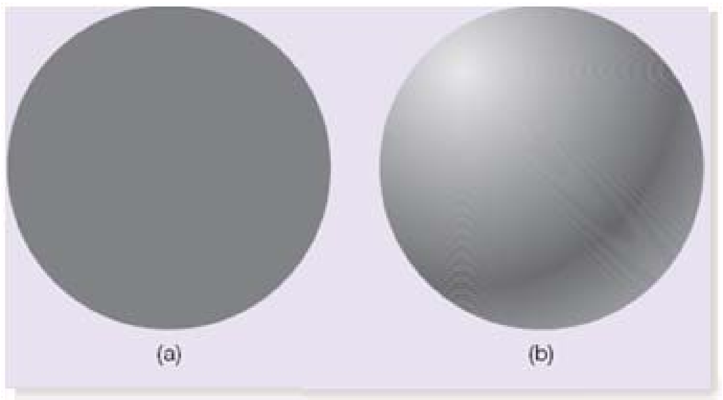

Figure C.2 (a) A uniformly lit sphere and (b) a properly lit and shaded sphere.

Light can be emitted through either self-emission or reflection (Rautenbach, 2008). When looking at a light bulb it is obvious that we are predominantly dealing with self-emission. Light sources are categorised by their light emitting direction and the energy emitted at each wavelength – determining the colour of the light.

As also mentioned previously, objects can absorb or reflect light emitted from a light source depending on the reflecting object's material properties. Light will thus only be "visible" when illuminated surfaces have the ability to reflect or absorb said light. Objects in computer generated graphical scenes are assigned so-called *Material properties.* These are user defined parameters built around rules determining the amount of scattering or reflection of incident light. Some surfaces, like a mirror, might reflect an incoming ray of light perfectly (hence appear shiny) while a carpet might reflect light in so many directions that it appears matte.

The type of light source also plays an important role in addition to the object's material properties. A *light type property* specifies the type of light to place in a scene. This property simply denotes a light source as a point light, spotlight or directional light (also called a parallel light). Lighting can thus be described as the interaction between a light source and an object's surface based on a pre-defined set of material properties. We will focus on each of these light source types in subsequent sub-sections.

A light source can be considered a geometric object, i.e. a simple light emitting surface. We can define a light emitting point on this surface (x, y, z) characterised by a wavelength energy value (λ) and an emitting direction (θ, ϕ) as shown in Figure C.3.

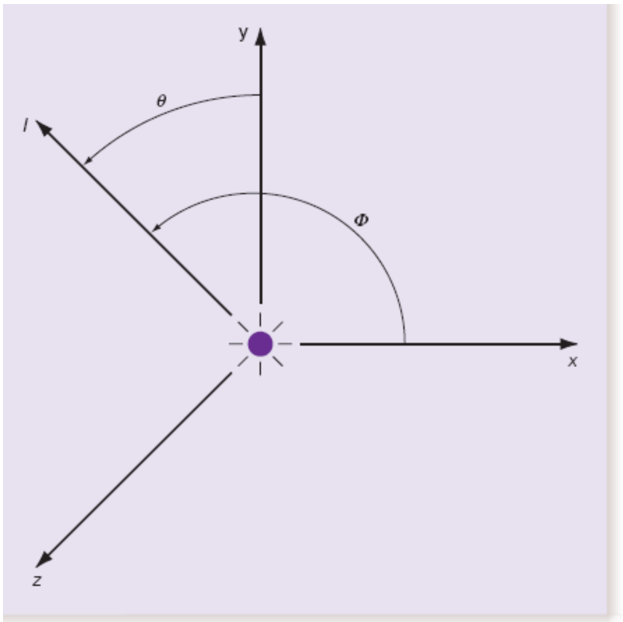

Figure C.3 A basic light source characterised via six elements.

By combining these variables, we are able to define the *illumination function* $I(x, y, z, \theta, \phi, \lambda)$ used to describe any light source in terms of six variables. For example, say a surface is being illuminated by a light source; then we can calculate the overall illumination on this surface by integrating across the surface of the light source – thus incorporating the effect of the angle between the light source and reflection surface as well as the falloff distance (the distance from the light source to the reflecting surface). Figure C.4 shows two distinct illumination functions for a pair of points located on the surface of a light source.

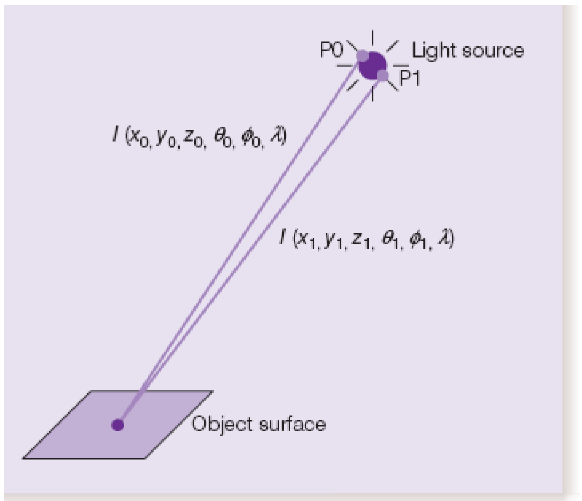
P0
Light source
P1
$I(x_0, y_0, z_0, \theta_0, \phi_0, \lambda)$
$I(x_1, y_1, z_1, \theta_1, \phi_1, \lambda)$
Object surface

Figure C.4 Two distinct illumination functions for a single light source.

Numerous colour intensities or shades can be described by additively combining various intensities of red, green and blue. Building on this, light sources can be defined using a similar red, green and blue colour component model. Each light source component is subsequently used to calculate the corresponding colour component of an illuminated surface. This three-component description is called *luminance* or *intensity*, and can be written using standard matrix notation with each component representing the intensity of either the red, green or blue colour component of the light source:

$$I = \begin{bmatrix} I_r \\ I_g \\ I_b \end{bmatrix}.$$

Furthermore, the overall lighting effect can be characterised by a lighting model (Whitted, 1980). A *lighting model* defines light-object interactions based on the type of light source and the material properties of the object. There are a number of commonly implemented lighting models and it is important to note that the basic graphics pipeline is constrained to the use of just one lighting model, namely, the *fixed-function lighting* model. This lighting model is basically an extended version of the Phong lighting model. The dawn of shader programming allows for full programmability of the graphics pipeline, thus facilitating the implementation of custom user-specified lighting models such as Lambertian lighting, anisotropic lighting, Fresnel lighting and Blinn lighting.

C.1.1 Point Lights

A *point light* emits light uniformly in 360 degrees. Point lights have fixed colour and position values and are omnidirectional in nature. The objects illuminated by this light type appear either oversaturated (overly bright with a high contrast) or too dark – a side effect easily corrected through the addition of ambient lights (Newman and Sproull, 1973). The primary factor influencing brightness is the distance between the illuminated surface and the point light. Point lights are the easiest of all light types to implement, resulting in their widespread use regardless of their unrealistic simulation of real-life light sources. Figure C.5 illustrates the effect of a point light illuminating a surface.

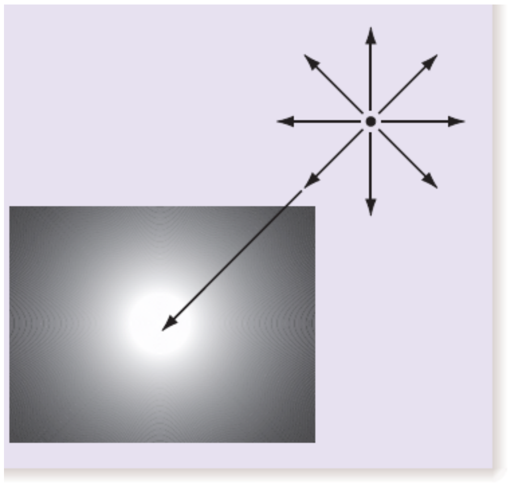

Figure C.5 Point light illumination.

Using the previously discussed luminance function, we can define a point light located at point P_1 as follows:

$$I(P_1) = \begin{bmatrix} I_r(P_1) \\ I_g(P_1) \\ I_b(P_1) \end{bmatrix}.$$

Using this luminance function, we can calculate the level of illumination at a specific point, *k*, on a surface by multiplying the intensity of the light with the inverse square distance between the light source and illuminated surface:

$$I(k, P_1) = \begin{bmatrix} I_r(P_1) \\ I_g(P_1) \\ I_b(P_1) \end{bmatrix} * \frac{1}{|k - P_1|^2}.$$

C.1.2 Spotlights

Spotlights are specified by a colour, spatial position and some specific direction and range in which light is emitted. A spotlight is basically a point light with its emitting light constrained within an angle range. This range is defined using two cones: a bright inner cone and an encircling outer cone. The inner cone has a high intensity (correlating to the user-defined luminescence of the light source), with the outer cone used for fading or attenuating the light source's intensity in an outwards direction. This gradual reduction of light intensity is referred to as *falloff*. Falloff governs the decrease in light intensity from the inner cone to the outer cone and a falloff value of 1.0 generally denotes an evenly distributed light intensity decrease. Figure C.6 illustrates this diminishing property.

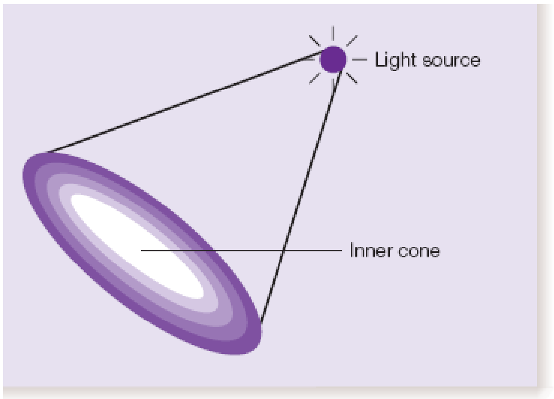

Figure C.6 Spotlight falloff.

The intensity of a spotlight can be calculated by considering the angle between the direction of the light source and a vector to the point being illuminated. The simplest way of formulating this intensity is to calculate the cosine, to the power of *e*, of the direction angle:

$$I = \cos^e \theta, \quad \text{with } e \text{ representing exponential falloff}.$$

We can also calculate the dot product of the spotlight's direction vector and the vector to the point being illuminated. This calculation results in the cosine of the angle between these two vectors (shown in Figure C.7):

$$\cos\theta = (spotlightDirectionVector) \bullet (vectorToSurface).$$

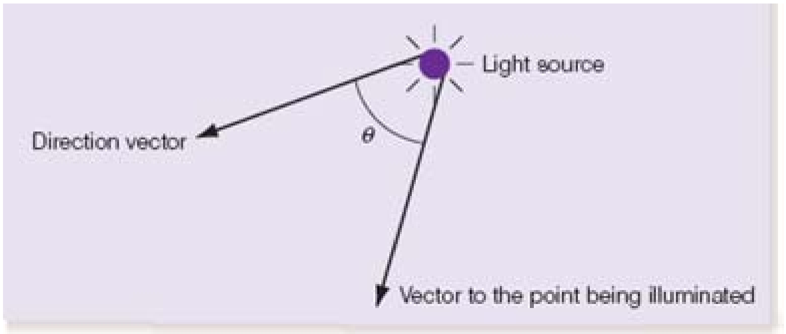

Figure C.7 The relationship between the direction vector and the vector to the point being illuminated.

C.1.3 Ambient Lights

Ambient lighting provides a uniform level of illumination throughout a scene. Numerous large light sources are generally positioned in such a way as to scatter emitted light in all directions, thus making it impossible to determine the original position of the light source. Even though ambient light hitting a surface is scattered equally in all directions, we can still determine the ambient intensity at each point on the surface.

This type of illumination has a luminance, *I*, which is the same for all points in the scene (with the manner of reflection being completely dependent on the material properties of a surface):

$$I_A = \begin{bmatrix} I_{Ar} \\ I_{Ag} \\ I_{Ab} \end{bmatrix},$$ with I_{Ar} the red, I_{Ag} the green and I_{Ab} the blue colour component of the ambient light intensity.

C.1.4 Parallel Lights

A *parallel* or *directional light* illuminates objects through a series of parallel light rays. These light sources can be considered as point lights located a significant distance from the surface of an object. Moving from one closely located object to another has little influence on the direction at which light hits the object. Sunlight can be considered a parallel light source due to it illuminating closely located objects at the same angle. Thus, the vector to the point being illuminated does not change a great deal when moving from one object to the next. We also use this direction vector to describe the light source. Figure C.8 illustrates a parallel light source.

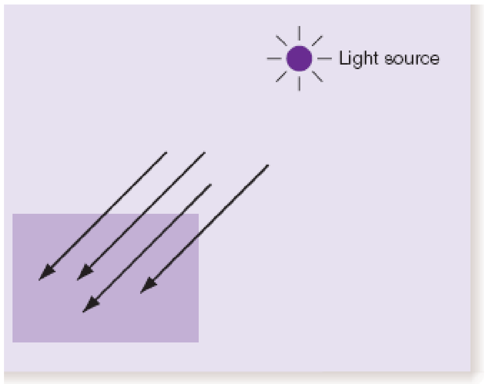

Figure C.8 A parallel light.

Parallel lights do not exhibit attenuation or range properties. Consequently, they do not require any calculations dealing with illumination effects such as falloff. They are thus excellent light sources when computational overhead is being considered.

C.1.5 Emissive Light

Emissive light is radiated (can be considered self-reflecting) light originating from an object's surface. This type of light blends with our other light types, resulting in a surface smoothly coloured through the combination of all global light colour components. An object coloured using emissive light appears flat and unshaded; this is due to emissive reflection not considering vertex normals or "incoming" light direction. We can describe emissive lighting using a three-component intensity function:

$$I_E = \begin{bmatrix} I_{Er} \\ I_{Eg} \\ I_{Eb} \end{bmatrix},$$

with I_{Er} the red, I_{Eg} the green and I_{Eb} the blue colour component of the emissive light intensity.

C.2 Reflection

A surface is only visible when it has the ability to reflect or absorb light. This ability is the result of the surface's material properties, i.e. rules determining the amount of scattering and/or reflection of incident light (Rautenbach, 2008). We can specify material properties for any surface, the most common types being the Phong reflection model, ambient reflection, diffuse reflection, specular reflection and transparency (Schlick, 1993).

The basic lighting model can be considered as a high-level equation summing an ambient, diffuse and specular component to calculate the colour of an object's surface (Sillion and Puech, 1989):

Surface colour = ambient lighting term + specular lighting term + diffuse lighting term.

This surface colour is actually equal to the overall amount of light present in a scene, commonly called global illumination and extended to include an emissive lighting term, resulting in the following lighting model equation used to simulate a wide range of lighting conditions (Walter et al, 1997):

Global illumination = ambient lighting term
+ specular lighting term
+ diffuse lighting term
+ emissive lighting term.

We will now look at each of these lighting/reflectance components as functions of material properties (e.g. surface reflectance, colour) and light source properties (e.g. light direction, colour, position, attenuation).

C.2.1 Ambient Reflection Model

Ambient reflection, also called continuous reflection, occurs whenever light emitted from a source is reflected so much that its origin is impossible to determine. Ambient light is omnidirectional in nature. Omnidirectional light is radiated uniformly in all directions, or more commonly, it is light scattered uniformly in all directions (Warn, 1983). This is also the reason for ambient reflection being described as continuous reflection – it being continuous in all directions, affecting the entire surface in an equal fashion. Thus, some of the light hitting a surface is absorbed while the rest is reflected – resulting in ambient reflection. Also, every point in a scene receives the same amount of ambient lighting, with only the reflection of this light varying. Figure C.9 illustrates this concept.

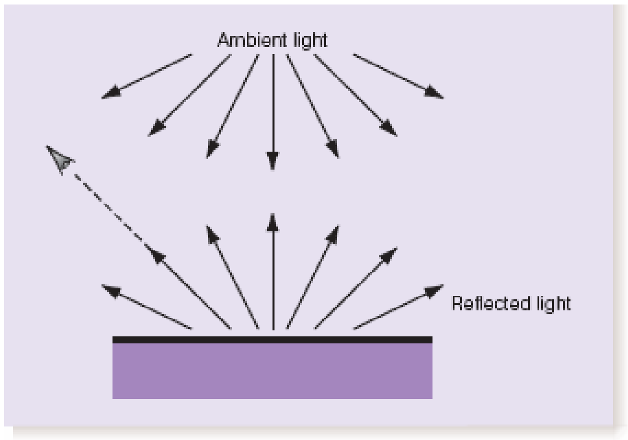

Figure C.9 Ambient reflection

The problem with ambient reflection is that illuminated objects appear rather flat and unshaded; Figures C.1 (a) and C.2(a) show the classic appearance of ambient lit surfaces.

This ‘flatness’ is the result of ambient lighting not factoring in vertex normals or the direction, position, range, and additional light source properties such as attenuation or falloff. Ambient reflection is thus the most computationally efficient of all the reflection models. The ambient reflection coefficient is an indication of the reflected amount and is comprised out of red, blue, and green ambient reflection coefficients collectively. The equation for calculating ambient lighting factors in the material’s ambient reflectance and the colour of the incoming ambient light:

Ambient lighting term = material’s ambient reflectance x incoming ambient light colour.

We can also define the intensity of ambient reflection using the ambient luminance function (IA), the incoming ambient light colour (I) and the material’s ambient reflectance consisting of three reflection coefficients – RAr, RAg and RAb, representing the red, green, and blue ambient reflection coefficients, respectively:

$$\begin{bmatrix} I_{Ar} \\ I_{Ag} \\ I_{Ab} \end{bmatrix} = \begin{bmatrix} R_{Ar} \\ R_{Ag} \\ R_{Ab} \end{bmatrix} * \begin{bmatrix} I_r \\ I_g \\ I_b \end{bmatrix}.$$

C.2.2 Specular Reflection Model

Specular reflection occurs whenever light, from a single incoming direction, is reflected at a single outgoing direction (Torrance and Sparrow, 1967). Specular reflection is characterized by bright highlights on the surface of an object reflected in the direction of the view vector. This concept is illustrated in Figure C.10.

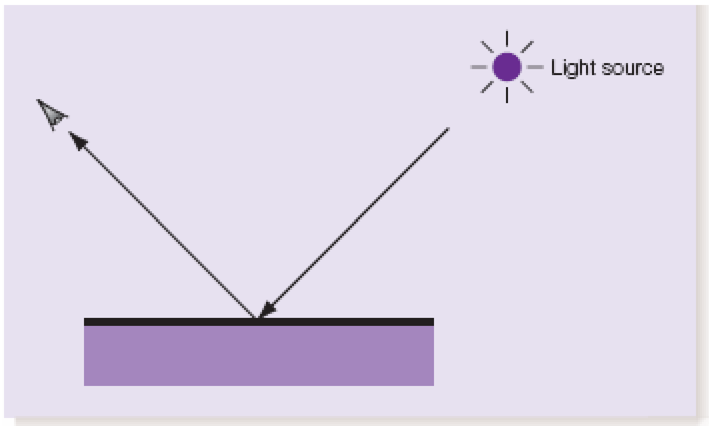

Figure C.10 Specular reflection

Specularity can be defined the amount of shininess exhibited by an object with the level of specular reflection attributed to a user definable value, namely, the shininess coefficient. The bigger this coefficient, the smoother the object's surface and the closer we are to a perfect mirror. For example, values ranging from 100 to 500 represent most metallic surfaces while smaller values represent materials with broader highlights such as plastic and wood. Figure C.11 shows several spheres with specular highlights.

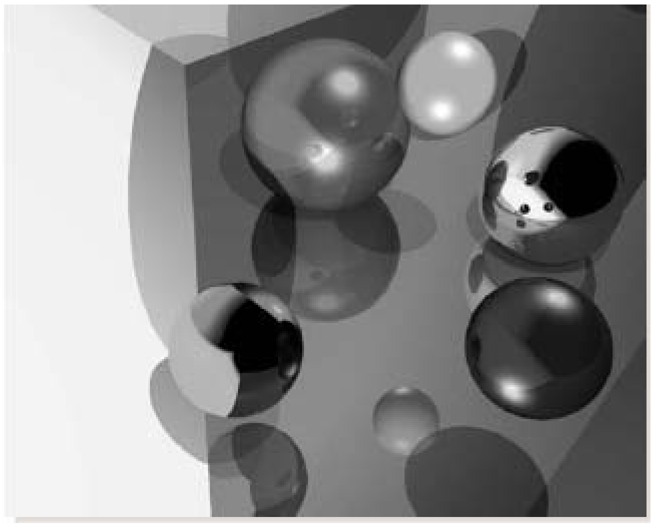
Figure C.11 Examples of specular highlights

To calculate specular reflection we need information about both the incoming light direction and location of the viewer as well as the colour properties of the material, light source and shininess of the surface. The equation for calculating specularity is:

$$\begin{aligned}\text{Specular lighting term} = {} & \text{material's specular colour} \\ & \times \text{colour of incoming specular light} \\ & \times \text{geometryFacingFlag} \\ & \times (\max(\text{normalized surface normal} \\ & \qquad \bullet \text{normalized halfway vector}, 0))^{\text{shininess}}\end{aligned}$$

The *geometryFacingFlag* element is a flag ensuring that specular highlights are limited to geometry facing a light source – its value is calculated by taking the dot product between the normalized surface normal and the normalized vector pointing to the light source. If this dot product is greater than zero then the *geometryFacingFlag* element is set to 1, otherwise 0. The *normalized halfway vector* element is the vector halfway between the normalized vector pointing towards the viewpoint and the normalized vector pointing in the direction of the light source. Specular highlights are prominent when the angle between these two vectors is small. Figure C.12 shows the vectors used in the calculation of this specular term.

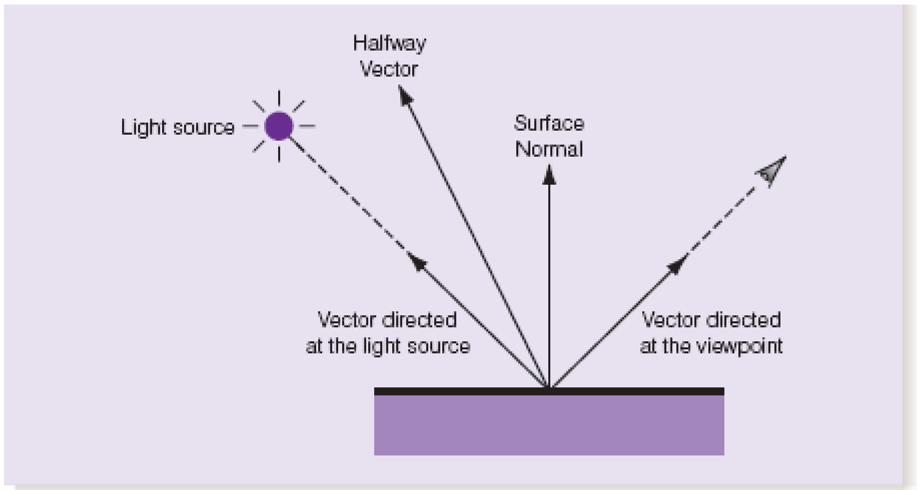
Halfway Vector
Light source
Surface Normal
Vector directed at the light source
Vector directed at the viewpoint

Figure C.12 Vectors used in the calculation of the specular term

Alternatively we can define the intensity of specular reflection using the specular luminance function (I_S); the angle between the reflection vector (the direction of a perfectly reflected ray) and the vector directed at the viewpoint; the intensity of the specular light, I; the shininess coefficient, α; and Rs, the fraction of the incoming specular light being reflected:

$$\begin{bmatrix} I_{Sr} \\ I_{Sg} \\ I_{Sb} \end{bmatrix} = \begin{bmatrix} R_{Sr} \\ R_{Sg} \\ R_{Sb} \end{bmatrix} \begin{bmatrix} I_r \\ I_g \\ I_b \end{bmatrix} \cos^{\alpha} \phi.$$

C.2.3 Diffuse Reflection Model

Diffuse reflections occur when incoming light is reflected in arbitrary directions (Goral et al, 1984). The main contributing factor to this form of reflection is an uneven or rough surface. A diffuse surface appears identical to all viewers, regardless of their respective point of view. This type of reflection is common for matte or uneven surfaces (such as carpets or brushed metal) and is used for shading surfaces in such a way as to convey a sense of depth.

Diffuse reflection is a function of the incoming light direction and surface normal, in other words, the reflection of incoming light is dependent on the surface roughness and incoming light angle (Hall, 1989). The equation for calculating diffuse lighting is:

```
Diffuse lighting term = material's diffuse color
                        x color of incoming diffuse light
                        x max(normalized surface normal
                               • normalized vector towards light,0)
```

The dot product between the normalized surface normal and normalized vector pointing towards the light source gives the measure of incident light received by the surface – the smaller the angle between these two vectors, the greater the dot product result, and the greater the amount of incident light falling on the surface. The max (normalized surface normal. normalized vector towards light, 0) element in the equation ensures that only surfaces facing a light source reflect some diffuse lighting – surfaces facing away from a light source result in a negative dot product. Figure C.13 shows a diffuse surface with the normalized surface normal and normalized vector pointing at the light source.

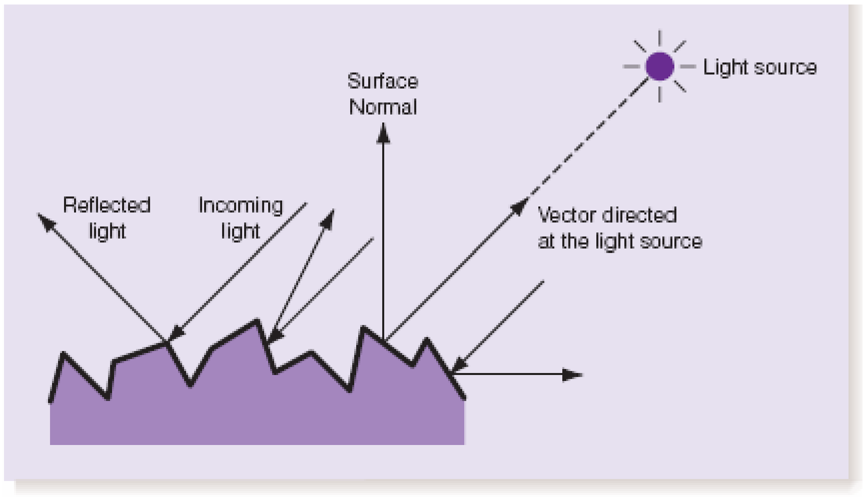

Figure C.13 Diffuse reflections

We can also define perfect diffuse surfaces, i.e. surfaces reflecting light in no particular direction. These surfaces, also called Lambertian surfaces, are generally so rough that it is mathematically impossible to determine a preferred angle of reflection. Also, Lambertian light has a consistent intensity regardless of the distance between the reflecting surface and light source.

Perfect diffuse reflection can be modelled using Lambert's cosine law. This law states that the reflection or radiance observed from a perfect diffuse surface is directly relative to the cosine of the angle between the vector directed at the light source and the surface normal:

$R_D \propto \cos\phi$, where R_D is the diffuse reflection and ϕ is the angle between the vector directed at the light source and the surface normal.

Simply put, Lambert's law states that a perfectly diffuse surface always reflects the same amount of light, regardless of the viewing angle. For example, say a surface is being illuminated using a parallel light source, when this light is positioned perpendicular to the surface; the surface will appear brightly lit. Placing this light source at, say, a 135 degree angle will result in a more dimly lit surface due to the light rays covering a larger surface area. Figure C.14 illustrates Lambert's cosine law.

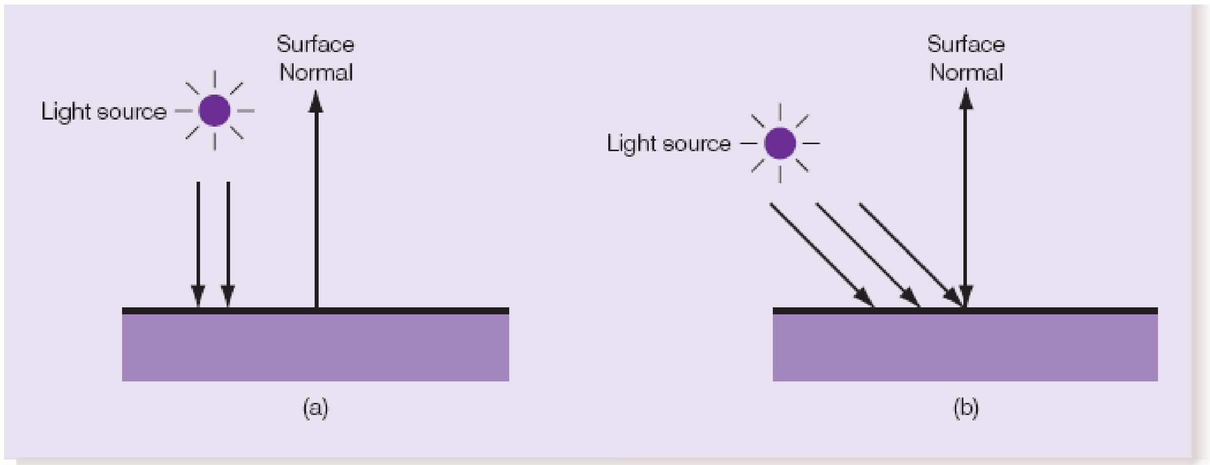

Figure C.14 A perfect diffuse surface being illuminated by (a) a light source positioned perpendicular to the surface and (b) a light source positioned at a 135 degree angle

C.2.4 The Phong Reflection Model

The *Phong reflection model*, also loosely called Phong shading, was developed in 1973 by Bui Tuong Phong (the late computer graphics researcher and pioneer) and later extended to include a halfway vector in the calculation of the specular term by Jim Blinn. The Phong model is an illumination model that controls the shading of individual pixels; it is computationally efficient and leads to realistic looking reflections. Phong's goal was to create realistic looking objects in as close to real time as possible. The Phong reflection model basically combines ambient, specular and diffuse lighting components to closely approximate real world reflections. This concept is shown in Figure C.15. We can consequently write the combination of these lighting terms as:

$$I = Ia + Id + Is.$$

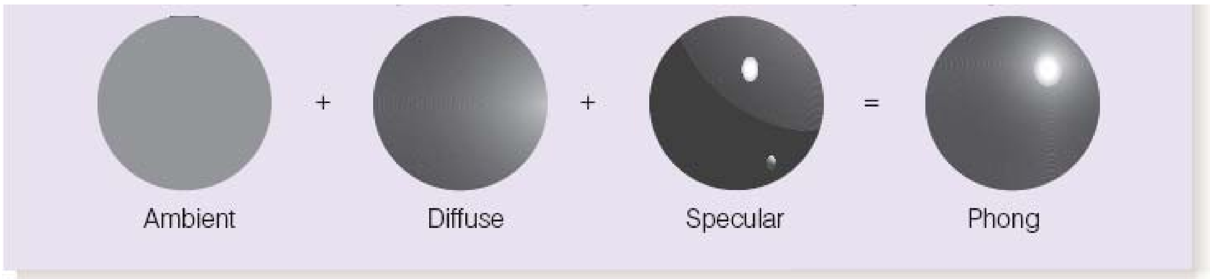

Figure C.15 Combining the lighting terms, producing a Phong reflection

Mathematically, the Phong reflection model considers reflected light as a function of the cosine between the surface normal and the incoming light direction. More precisely, the colour value of a point on the surface being illuminated is a function of four vectors, as shown in Figure C.16: the normal vector at this point, the vector directed at the viewpoint, a vector directed at the light source, and the reflection vector (indicating the direction of a perfectly reflected ray).

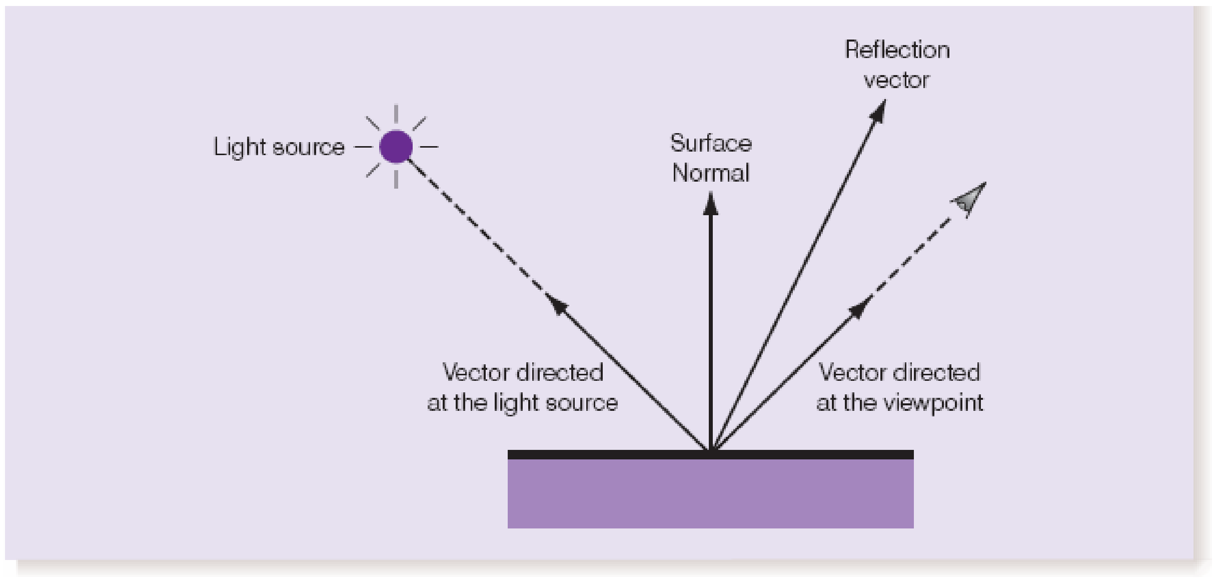
Reflection vector
Light source
Surface Normal
Vector directed at the light source
Vector directed at the viewpoint

Figure C.16 Vectors used in the calculation of the Phong reflection model

The following equation can be used to calculate the Phong reflection of a point on the surface of an object:

$$\textit{Phong reflection} = \mathrm{k_a} i_a + \sum (k_d (L \cdot N) i_d + k_s (R \cdot V)^{\alpha} i_s)$$

with k_a the material's ambient reflectance, i_a the colour of incoming ambient light, k_d the material's diffuse reflectance, L the vector directed at light source, N the surface normal, i_d the colour of incoming diffuse light, k_s the material's specular reflectance, R the reflection vector, V the vector directed at the viewpoint, a the shininess coefficient and is the colour of incoming specular light. The Phong reflection, using this equation, is typically calculated for individual intensities of red, green, and blue. The sum component in the above given equation defines a set of light sources. The effect of each light source, on the point being illuminated, is thus considered by the equation.

Appendix D

Real-time Shadow Generation

D.1 Introduction

Real-time shadow generation contributes heavily towards the realism and ambience of any scene being rendered. Research dealing with the calculation of shadows has been conducted since the late 1960s and has picked up great momentum with the evolution of high-end dedicated graphics hardware. Shadows are produced by opaque or semi-opaque objects obstructing light from reaching other objects or surfaces. A *shadow* is a two-dimensional projection of at least one object onto another object or surface (Crow, 1977). The size of a shadow is dependent on the angle between the light vector and light blocking object. The intensity of a shadow is in turn influenced by the opacity of the light-blocking object. An opaque object is completely impenetrable to light and will thus cast a darker shadow than a semi-opaque object. The number of light sources will also affect the number of shadows in a scene (with the darkness of a shadow intensifying where multiple shadows overlap). Figure D.1 illustrates shadow generation, specifically the implementation of stencil shadow volumes – a popular shadow rendering technique.

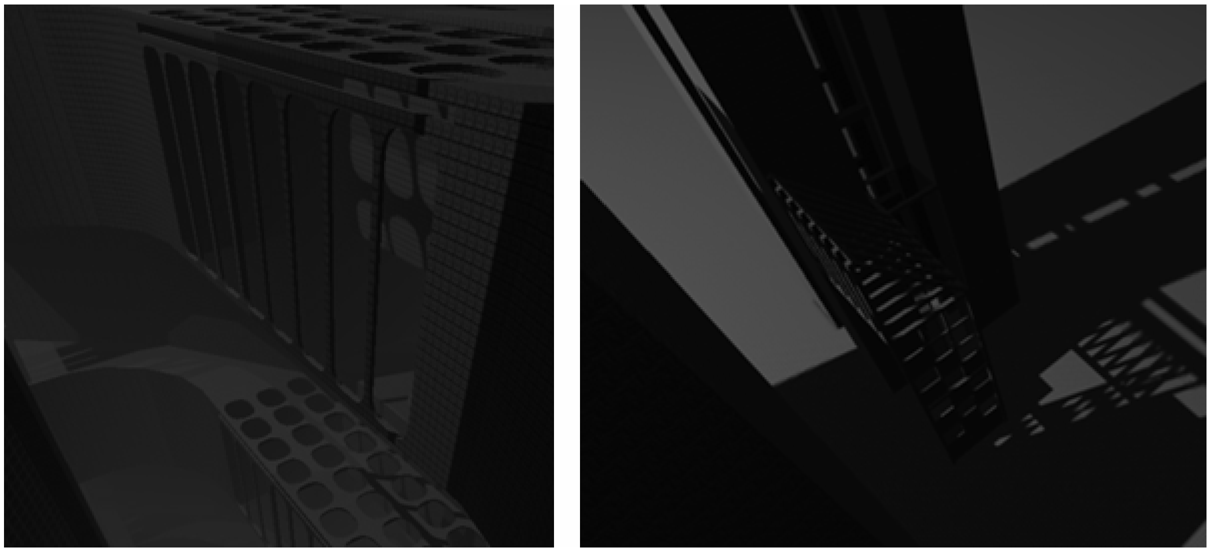
Figure D.1 Example of stencil shadowing – note the darkening of overlapping shadows.

The drive towards realism has led to the development of many shadowing algorithms. Some of these algorithms, like shadow mapping and shadow volumes, are more successful than others. The success of an algorithm is dependent on the balance between speed and realism and techniques like shadow mapping and stencil shadow volumes are particularly amenable to hardware implementation – thus freeing the CPU of a substantial processing burden and making the real-time rendering of shadows feasible (Kilgard, 1999). Other shadowing approaches, such as the one proposed by Boulanger et al (2003), have in turn focussed on visually pleasing approximations for computationally expensive natural scenes.

Looking at shadows from a foundational perspective reveals them as a product of an environment's lighting. Shadows can have either hard or soft edges. This is dependent on the type of light source used and the distance between the light source and object. In the case of soft shadows we differentiate between both an umbra and penumbra. The darkest area of a shadow, receiving no light at all, is referred to as the *umbra* with the *penumbra*, receiving a small amount of light, indicating the partially shadowed edge (Akenine-Möller et al, 2002). Figure D.2 illustrates a shadow's umbra and penumbra.

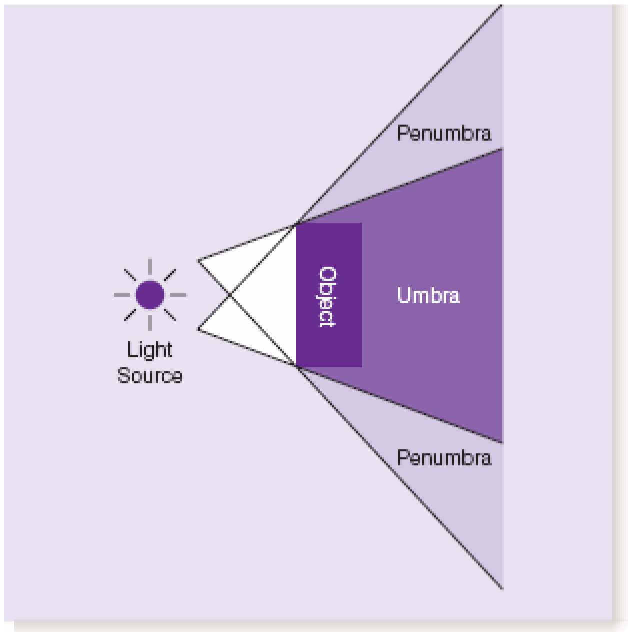

Figure D.2 A soft shadow with related umbra and penumbra.

It should be noted that there is always a gradual intensity transformation from the umbra to penumbra (Akenine-Möller et al, 2002). However, the fading of the shadow (as its distance from the casting object increases) need not necessarily be gradual. Point lights will, for example, produce non-fading hard-edged shadows, with ambient light sources producing soft-edged shadows fading into the distance. The area of a light source also affects the gradual softening of shadows. The larger the light source's area, the more quickly the shadow grades off. Figure D.3 shows the difference between shadows produced by point and ambient light sources.

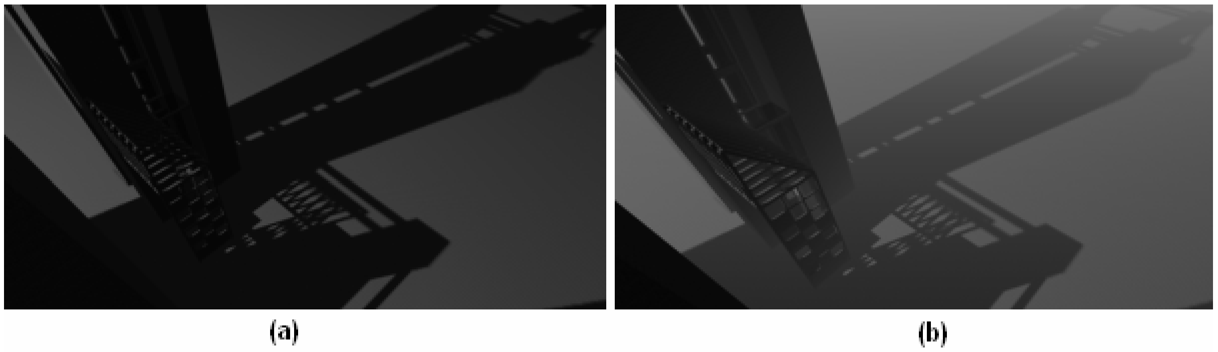

Figure D.3 (a) Hard-edged shadow produced by a point light source. (b) Soft-edged shadow produced by an ambient light source.

We will now investigate several shadowing algorithms, including the fundamentals of shadow volumes and shadow mapping. The first two algorithms, namely scan-line polygon projection and Blinn's shadow polygons, are historic in nature. We describe these algorithms here not only for the sake of completeness but also since some of the elements introduced by them form the basis of general shadow computation. These first two techniques aren't suited for real-time implementations. However, more recent algorithms such as stencil shadow volumes and hardware shadow mapping remedy this situation by emphasising the balance between processor efficiency and realism.

It is necessary to note, before continuing, that shadowing remains one of the most processor intensive tasks and despite each technique's limitations, it is important to consider each algorithm with its intended application area in mind.

D.2 Shadow Rendering Algorithms

D.2.1 Scan-Line Polygon Projection

A quite complex, and now mostly redundant shadow algorithm was introduced by Appel (1968) and further developed by Bouknight and Kelley (1970). This algorithm, commonly known as *scan line polygon projection*, adds shadow generation to scan-line rendering (Lane et al, 1980). A *scan-line algorithm* operates on a row-by-row basis, as opposed to a pixel-by-pixel or polygon-by-polygon basis. A *scan-line* itself is a single line or row composed of a series of successive pixels stored in an array or list. The overall image is rendered as a result of the consecutive downwards repositioning of the scan-line (Bresenham, 1987). To enable both pre-rendered and real-time shadow generation via scan-line algorithms, it is necessary to append the original algorithm with a pre-processing stage. This pre-processing stage builds up a secondary data structure linking all the polygons that will cast a shadow on some other polygon.

The scan-line projection algorithm has an additional stage where all the polygons of a scene are projected onto a sphere centred at the light source (the centre of projection). This allows for the identification of all polygons casting shadows on other polygons. It is important to remember that, in a scene with *k* polygons, one will have at most *k*(*k* – 1) shadows – the detection and elimination of polygon groups not interacting are thus of crucial importance. With all the shadow casting polygons linked in a secondary data structure, we can now project the edges of these polygons onto polygons intersecting the scan-line. A pixel's colour value is modified wherever the scan-line traverses one of these shadow edges. Hence, the light source (at the centre of projection) and shadow polygon cast a shadow onto the polygon intersected by the scan-line. The following cases denote whether a given pixel is in shadow or not:

1) The scan-line algorithm continues normally if no shadow casting polygon for the given pixel exists.
2) Decrease the brightness of the scan-line segment's pixels if a shadow casting polygon fully overlaps the intersected polygon.
3) If a shadow casting polygon partially overlaps the intersected polygon, subdivide the intersecting scan-line segment recursively until condition 1 or 2 is reached.

Scan-line polygon projection only allows for the generation of hard-edged shadows via point light sources. Figure D.4 illustrates the above described process.

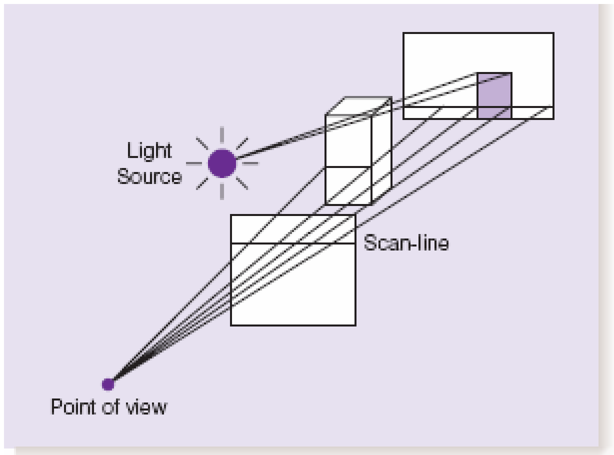

Figure D.4 Scan-line polygon projection.

D.2.2 Blinn's Shadow Polygons

An extremely easy to use shadow generation technique was described by Blinn (1988). This method simply calculates the projection of an object on some base-plane. In short, a shadow cast by a point light and a polygon onto another polygon can be rendered by projecting the first polygon onto the plane of the second polygon (Blinn, 1988). The point light is in this case at the centre of projection and the resulting shadow is referred to as a *shadow polygon*. Figure D.5 illustrates the projection of a shadow polygon (onto the *xy*-plane) with the light source located at the centre of projection.

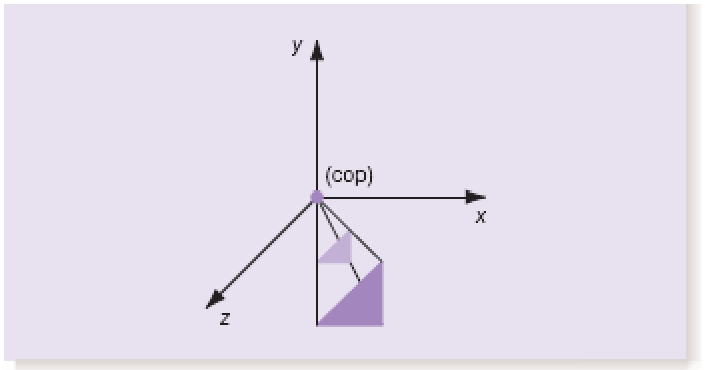

Figure D.5 Shadow polygon with a point light source at the centre of projection.

The local illumination approximation states that if we have an infinitely positioned point light source, then we can consider its light rays as parallel (Phong, 1975). These rays, emanating from a light source located at the point (x_l, y_l, z_l), will cast a shadow at the point (x_s, y_s, z_s) based on the intersection of any point (x_o, y_o, z_o) located on an object positioned between the light source and some plane.

Generally though, if we have some finitely positioned point light, then we can translate the scene by some matrix, $T(-x_l, -y_l, -z_l)$, so that the light source is positioned at the centre of projection. This translation yields the following projection matrix:

$$M = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & \dfrac{1}{-y_l} & 0 & 0 \end{bmatrix}.$$

After applying this projection matrix we have to translate the scene back to its original position with the generic translation $T(x_l, y_l, z_l)$. By concatenating the two translation matrices with the projection matrix, we are able to define the shadow projection (x_s, y_s, z_s) of the original point (x_o, y_o, z_o) as:

$$(x_l - \frac{x - x_l}{(y - y_l)/y_l}, 0, z_l - \frac{z - z_l}{(y - y_l)/y_l}).$$

The following steps outline the process of creating a shadow polygon:

1) Define and initialise the shadow projection matrix **M**.
2) Render the polygon normally.
3) Translate the light to the origin (centre of projection).
4) Calculate the projection of the object with the shadow projection matrix.
5) Translate everything back to their original positions.
6) Render the shadow polygon.

This method is often utilised to render the shadows of single polygons (Blinn, 1988). It is, however, only useful for the projection of shadows on flat surfaces, not for inter-object shadows. We will much rather implement an alternative method whenever objects are expected to cast shadows on other objects. For example, we could create a relatively uncomplicated shadow algorithm by simply modifying a hidden surface removal algorithm. The premise behind our modification would be that shadows are in fact areas hidden from light sources.

D.2.3 Shadow Mapping

Lance Williams introduced the concept of shadow mapping in 1978. His primary aim was the rendering of shadows on curved surfaces. *Shadow mapping* adds shadows to a scene by testing whether a particular pixel is hidden from a light source. It does this by first constructing a separate shadow Z-buffer for every light source and then storing the depth information of a scene in this buffer with the light source as view point. This depth information leads to a *depth image* or *shadow map* consisting of all the polygons not hidden from the light source (Shade et al, 1998). Hidden pixels are discovered through a comparison with this depth image (Everitt et al, 2001). The shadow map partitions a light's view volume into shadowed and non-shadowed regions and we store this depth buffer image (shadow map) as a texture in the 3-D accelerator's texture unit. This texture is subsequently projected onto an area and/or object(s) for the shadow effect.

Although the shadow map is now stored in the display adaptor's texture memory, it must still be updated every time changes are made to the scene's light sources, geometry or object positions. However, no updating of the shadow map is required when altering the camera's point of view. We will typically partition the scene when implementing shadow maps, thus limiting the time it takes to update the depth image.

The final step of the algorithm is to render the scene via a Z-buffer algorithm. More specifically, if a pixel is not hidden from the light source then the related vertex is translated from the view point's screen space to light space (screen space with the light at the centre of projection). After all the vertices of an object have been translated, we have the object's spatial location from the light source's point of view.

The *x*- and *y*- coordinates of a translated vertex are used to index the shadow Z-buffer. Its *z*-component is used during the depth comparison test. This test simply compares a vertex's depth value to the corresponding value stored in the shadow map, determining whether the specific vertex will be shadowed or not. More explicitly, the vertex is in shadow if its depth value is greater than the value stored in the shadow map. For all other cases we can say that the vertex is closer to the light source than another arbitrary shadow casting surface and will thus be rendered without a shadow. Figure D.6 shows a 3-D object and its resulting shadow map.

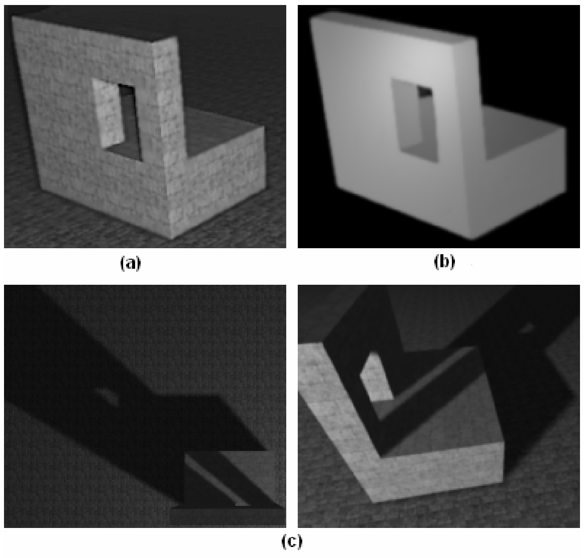

Figure D.6 (a) Object as seen from the light's point of view (b) Object's depth map from the light's point of view (c) Shadow polygon rendered via the horizontal projection of the depth map.

Shadow mapping can be implemented as either a single- or multi-pass algorithm (Everitt et al, 2001). That is, if a fragment shader is used to render shadows by performing the depth comparison test, then we will not require additional passes to produce the shadow maps (Fernando et al, 2001). However, if we do not make use of programmable shaders (such as NVIDIA's Cg or DirectX's High Level Shader Language) then we won't have access to predefined lighting models (lit or shadowed) and will consequently have to implement an additional shadow map generation pass for each light source (Lauritzen, 2006). In more complete terms, we can outline the dual-pass shadow mapping process as follows:

1) Create the shadow map by rendering the Z-buffer with regard to the light's point of view.
2) Draw the scene from the viewer's point of view.
3) For each rasterized fragment, calculate the fragment's coordinate position with regard to the light's point of view.

4) Use the x- and y- coordinates of step 3's translated vertex to index the shadow Z-buffer.
5) Do the depth comparison test, if the translated vertex's depth value (the z-value of step 3's translated vertex) is greater than the value stored in the shadow Z-buffer, then the fragment is shadowed, else it is lit.

Shadow mapping suffers from aliasing errors due to the use of a projection transformation mapping shadowed pixels to screen pixels, often causing changes in a pixel's screen size. This is a direct result of the Z-buffer algorithm's use of point sampling. The rendered shadow's edges are often jagged due to point sampling errors occurring during the calculation phase of the shadow Z-buffer. These errors are further amplified when accessing the shadow Z-buffer for the projection of pixels onto the shadow Z-buffer map. The only way of minimising the visibility of a shadow's jagged edges is to implement some form of pre-filtering and to use very large (high resolution) shadow maps.

D.2.4 Shadow Volumes

A *shadow volume* is a volumetric area defined by light rays extending outwards about the silhouette edge of an object (Crow, 1977). All the objects positioned within a shadow volume are hidden from the light source and are thus in either full or partial shadow. The contour of an object's surface is defined as a *silhouette edge* when the normal vector of the surface is perpendicular to the view vector (Everitt et al, 2002). A silhouette edge can more generally be considered as an outline or edge separating a front- and back-facing surface (Heidmann, 1991). The shape of the shadow volume is determined by the shape of the object's silhouette edge and a shadow volume is made up of so-called "invisible" shadow polygons. We refer to these shadow polygons as "invisible" since they are never rendered and only used to determine the shadowed areas. Shadow volumes are theoretically infinite volumes produced by polygons; however, for practical usability we intersect an infinite shadow volume with the view volume to produce a finite front- and back-capped shadow volume. Figure D.y shows the silhouette edge of a cube with Figure D.8 illustrating the capping of a semi-infinite shadow volume.

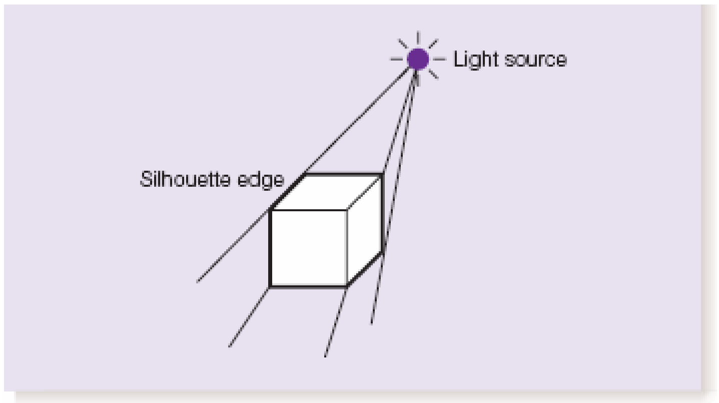

Figure D.7 A simple silhouette edge.

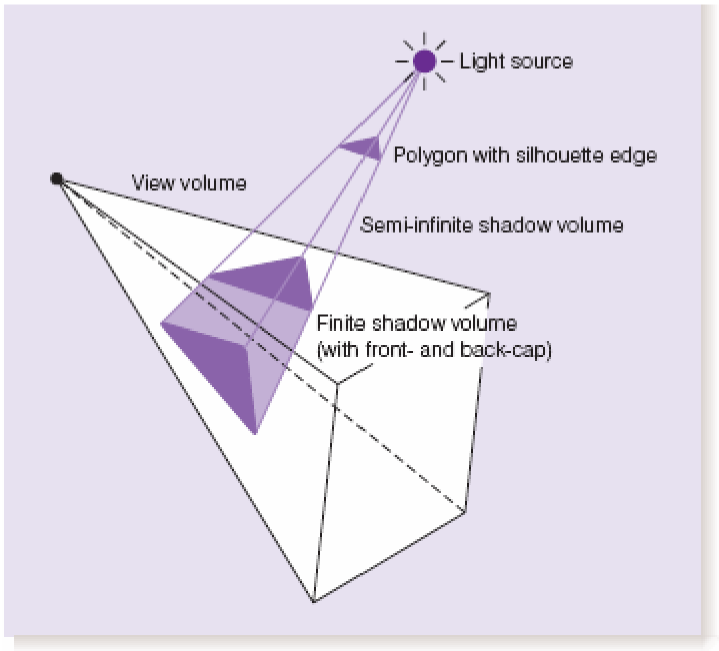
Light source
Polygon with silhouette edge
View volume
Semi-infinite shadow volume
Finite shadow volume
(with front- and back-cap)

Figure D.8 Construction of finite shadow volume.

The original shadow volume concept was introduced by Frank Crow in 1977. He defined a shadow volume as three-dimensional area occluding objects and surfaces from a light source. This original approach has since been extended to incorporate the generation of soft-edged shadows, including revision of the algorithm to utilise modern-day 3-D acceleration capabilities. The advent of dedicated 3-D acceleration hardware and the direct control of this hardware via APIs such as OpenGL and Direct3D have significantly contributed to the use of shadow volumes in modern computer games such as id Software's *Doom 3* and Bioware's *Neverwinter Nights* (Carmack, 2000).

The first feasible real-time shadow volume algorithm was introduced by Tim Heidmann in 1991. His algorithm made use of the 3-D accelerator's stencil buffer – effectively limiting the render area (called stencilling). The *stencil buffer* controls rendering by enabling or disabling drawing to a specific pixel. Heidmann discovered that the stencil buffer could be used to count the number of front- and back-facing shadows in front of an object if we rendered the shadow surfaces in two passes. By counting these shadow surfaces we are able to determine whether an object's surface is in shadow or not. Heidmann's technique became known as the *depth-pass* stencil mask generation algorithm.

The general Heidmann stencil shadow volume process is summarised by the following phases:

1) Assume the scene in entirely shadowed.
2) Render the shadowed scene.
3) Calculate the shadowed scene's depth information.
4) Use this depth information to define a mask via the stencil buffer to indicate the lit areas.
5) Assume the scene is entirely lit.
6) Render the lit scene, applying the stencil buffer mask to cast the shadows.

There are two variations to the depth-pass technique, namely, depth-fail and exclusive-or (the latter of which is omitted due to its failure in dealing with intersecting shadow volumes). All shadow volume algorithms follow the above described shadow generation process and differ only in their approach of calculating the stencil mask. The depth-pass and depth-fail stencil shadow volume algorithms are described in detail below.

Depth-pass

Shadow volume algorithms operate on a per-pixel basis, performing a shadow test for every pixel in the frame buffer. We refer to all the data needed for the rendering of a pixel (stored in the frame buffer) as a fragment. Our algorithms will thus focus on all rasterized fragments to determine whether a specific fragment is in shadow or not. In more complete terms, we can write the above outlined stencil shadow volume process as follows:

1) For each rasterized fragment, render the fragment using ambient lighting, updating the Z-buffer after each fragment has been rendered.
2) Now we have to compute which fragments are in shadow. We once again look at each rasterized fragment, rendering the fragment as lit if not shadowed.

We can use the depth-pass method to test whether a fragment is in shadow or not. This method computes the fragments in shadow by generating a stencil mask. Using the stencil buffer, we count the number of front- and back-facing shadows in front of an object by rendering the front- and back-faces of the shadow surfaces in two passes. By counting these shadow surfaces we are able to determine whether an object's surface is in shadow or not. If there are more front-facing shadow surfaces than back-facing ones, then we can conclude that a shadow is projected onto an object. The following process is used to compute the number of fragments in shadow:

1) For each rasterized fragment, render the fragment using ambient lighting, updating the Z-buffer after each fragment has been rendered.
2) Determine the silhouette edges of a shadow casting object. Following this the shadow volume polygons (shadow surfaces) are calculated (from the light source using the silhouette edges of the shadow casting object). These two steps are performed for each shadow casting object.
3) Now deal with the front- and back-facing shadow surfaces with regard to the point of view, **incrementing** the stencil buffer value for each front facing shadow surface if the depth-test **passes** (depth-pass using the Z-buffer) – counting the shadows in front of the object. Following the test for front-facing shadow surfaces, we focus on each back-facing shadow surface with regard to the view point – **decrementing** the stencil buffer value if the depth-test for a specific shadow surface **passes**.

Following the above process, we simply have to check the stencil buffer value for each fragment to identify the fragments in shadow. If a fragment's stencil buffer value is greater than zero then we need not draw this fragment during the second rendering pass – hence causing the fragment to be in shadow. Figure D.9 illustrates the above described process:

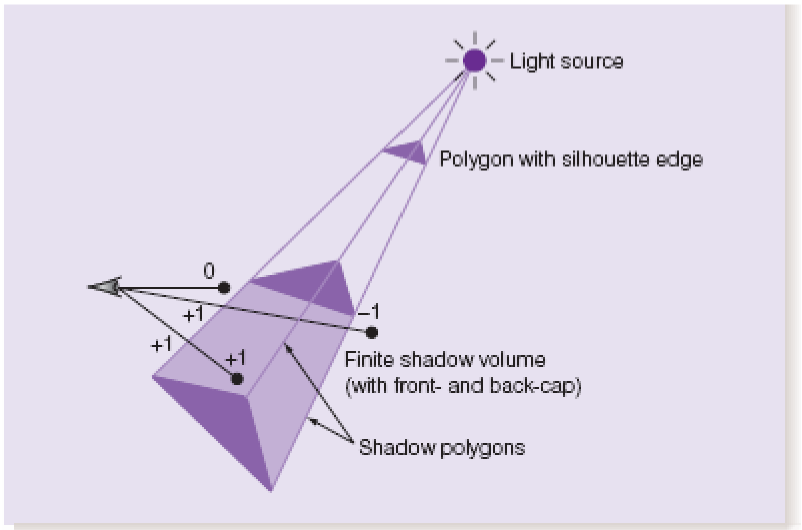

Figure D.9 Testing whether a fragment is in shadow.

The described depth-pass process is extremely efficient; however, certain issues become apparent upon implementation. The most common problem occurs whenever the point of view (camera or viewer) is positioned within a shadow volume. This leads to visibility of the shadow's back-face. The depth-test will pass in this case, causing the stencil buffer value to be decremented, thus becoming -1 due to a back-face being visible prior to any front-facing shadow surfaces. This problem is referred to as *stencil counting inversion* and it can be resolved by capping the front of the shadow volume. Alternatively we can initialise the stencil buffer to 2^{K-1}, with K the precision of the stencil buffer. These approaches are, however, less than efficient and the depth-fail technique is generally implemented as an alternative.

Depth-fail

The depth-pass approach computes the stencil buffer values by incrementing for front- and decrementing for back-facing shadow surfaces. The depth-fail approach modifies this calculation process (originally counting from the point of view) by counting from infinity. So, by reversing the depth and counting the shadow surfaces behind an object instead of those in front of it, we no longer face the *stencil counting inversion* issue. The only general issue with this approach is that we must cap the end of the shadow volume to avoid the condition where shadows point to infinity. The following process is used to compute the number of fragments in shadow:

1) For each rasterized fragment, render the fragments using ambient lighting, updating the Z-buffer after each fragment has been rendered.
2) Determine the silhouette edges of a shadow casting object. Following this the shadow volume polygons (shadow surfaces) are calculated (from the light source using the silhouette edges of the shadow casting object). These two steps are performed for each shadow casting object.
3) Now deal with the front- and back-facing shadow surfaces with regard to the point of view, **decrementing** the stencil buffer value for each front facing shadow surface if the depth-test **fails** (depth-fail using the Z-buffer). Following the test for front-facing shadow surfaces, we focus on each back-facing shadow surface with regard to the view point – **incrementing** the stencil buffer value if the depth-test for a specific shadow surface **fails**.

Although the depth-fail method effectively avoids the stencil counting inversion issue it still requires the additional back-capping of shadow volumes. This results in some extra rasterization time which can lead to considerable performance slowdowns under certain conditions. It is thus in some cases more advantageous to use the depth-pass method while explicitly dealing with the cases where the point of view is located within a shadow volume. It is also often possible to increase the performance of a stencil shadow volume implementation by utilising some hardware extension such as NVIDIA's *depth bounds test* enabling the culling of shadow volume sections not affecting the visible area.

It is interesting to mention though that Kolic el al (2004) developed a shadowing technique purely focussing on the utilisation of current GPU advances. Their algorithm specifically deals with the casting of shadows on concave complex objects such as trees. Koloc et al (2004) formally state that "for those objects, silhouette calculation that is usually preformed by other shadow volume algorithms is complicated and poorly justified. Instead of calculations, it is better to assume a worst case scenario and use all of the edges for construction of the shadow volume mesh, skipping silhouette determination entirely. The achieved benefit is that all procedures, i.e. the object and shadow calculation and rendering, could be done on GPU. The proposed solution for shadow casting allows open edges. Indexed vertex blending is used for shadow projections, and the only calculation required is determining projection matrices. Once created, shadow volume is treated like any other mesh." When Crow implemented and defined the original shadow volume model back in 1977, he simply did not have access to any of these modern hardware acceleration aids and hence did not develop the now commonly used stencil shadow volume algorithm with modern day graphics accelerators in mind.

Thakur et al (2003) also developed a discrete algorithm for improving the Heidmann original. Chapter 3 deals with this algorithm in detail. Another significant algorithmic improvement over the Heidmann original was made by Chan and Durand (2004). They specifically combined the strengths of shadow maps and shadow volumes to produce a hybrid algorithm for the efficient rendering of pixel-accurate hard-edged shadows. Their method uses a shadow map to identify pixels located near shadow discontinuities, using the stencil shadow volume algorithm only at these pixels.

Soft-edged Shadows using Penumbra Wedges

Implementation of the above discussed shadow volume techniques always result in pixel-accurate hard-edged shadows. Soft-edged shadows can be simulated through the construction of several shadow volumes by translating the original light source to various positions close to that of the original. Following this we simply have to combine the resulting shadows. The problem with this approach is rendering performance due to shadow volume construction taking up a substantial amount of processor time. One solution is the calculation of *penumbra wedges* as proposed by Akenine-Möller and Assarsson (2002). A penumbra wedge is defined in place of a shadow polygon for each silhouette edge of an object – combining a series of these penumbra wedges result in the creation of a soft-edged shadow.

The penumbra wedge algorithm calculates the amount of light that reaches a certain point *p*. This amount of light intensity ranges from '0' to '1'. When the light intensity is '0' we can define the point *p* as fully shadowed or conversely as fully lit with a light intensity of '1'. For all other values we can define point *p* located within the penumbra region. The light intensity inside the penumbra region is calculated using a signed 16-bit buffer. This light intensity buffer is simply a high precision stencil buffer. The lower the number of bits used for the buffer, the higher the implementation's performance and the lower the number of shades in the penumbra region. The varying shade levels in the penumbra region are created by multiplying each light intensity value stored in this buffer with some value *s*. This value is normally chosen as '255' since colour buffers allow for 8-bits per component, leading to at least '256' on-screen penumbra wedges. The following process is used for calculation of the penumbra wedges (illustrated in Figure D.10):

1) Initialise the light intensity buffer to '255' – indicating that the viewer is now positioned outside of the shadow volume.
2) Draw the scene using both specular and diffuse lighting.
3) Draw the penumbra wedges using the following algorithm:
    a. For some light ray, compute the entry and exit points on the outside penumbra wedge. This must be done for each visible fragment. The entry point is defined by an *x*- and *y*-coordinate, with the corresponding *z*-value stored in the Z-buffer.
    b. Transform this point to world space coordinates (the point's independent local coordinate system has now been transformed into a global coordinate system. This provides all the points with a shared global coordinate space – i.e. one point's position can be described in terms of another's and all user defined points can now be positioned within the same scene).
    c. Test whether the point is located within the penumbra region.
        i. If the point is located within the penumbra region, compute the light intensity of this point and the entry point, scaling the light intensity by subtracting the computed light intensity of the point located within the wedge from the entry point and multiplying this result by '255'.
        ii. Add the above calculated light intensity to the light intensity buffer.
4) Add ambient lighting to the rendered scene.

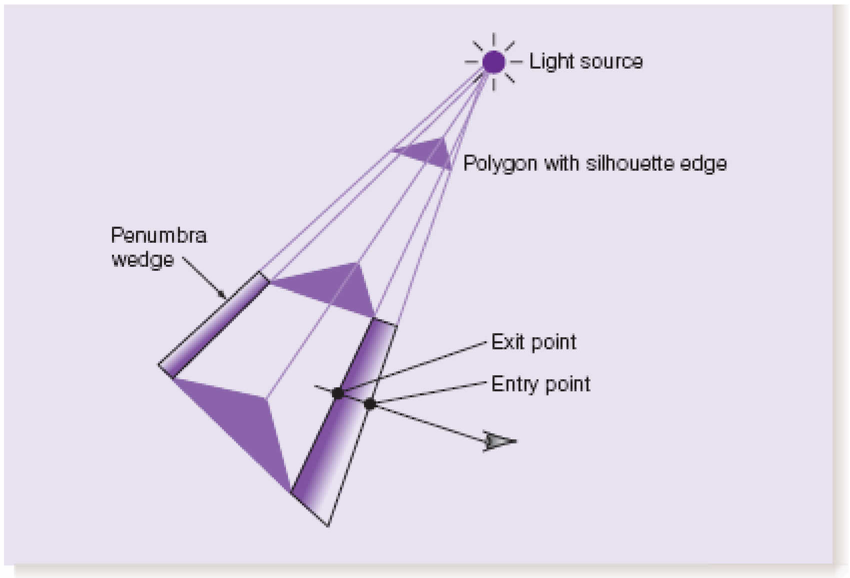

Figure D.10 Locating a point within the penumbra region.

The possibility of overlapping penumbra wedges exists in situations where the volume is entered more than once. Such cases result in negative light intensity values, thus requiring the clamping of the values stored in the light intensity buffer to the range [0, 255]. It is also possible to leave the volume more than once whenever the viewer is located within the volume. By setting the maximum possible light intensity value to ‘255’, we effectively avoid higher light intensities than that of the areas outside the volume – which clearly isn’t possible.

Akenine-Möller and Assarsson’s penumbra wedges algorithm (Akenine-Möller and Assarsson, 2002) can be implemented using either OpenGL or Direct3D. The main problem is the large vertex and pixel shader programs required, making true real-time performance only achievable on extremely high-end hardware. The following steps outline a hardware-accelerated implementation of the penumbra wedge algorithm:

1) Render the scene using either OpenGL or Direct3D.
2) Implement the wedge rasterization, initialising the Z-buffer prior to rasterization.
3) Rasterize the front facing triangles of the penumbra wedges – the entry point’s plane is now identified.
4) Identify the exit point by calculating the ray’s intersection with the back facing planes and picking the one closest to the ray.
5) Specify the point in world space coordinates via a transformation based on the Z-value.
6) Determine whether this currently selected point falls within a penumbra wedge or not by substituting the point’s coordinates into the plane equations:
    a. If the point falls within a wedge, calculate the intersection distances from the point to the planes.

Brotman and Badler (1984) developed a similar algorithm for the generation of soft-edged shadows (adding penumbras to hard-edged shadows). They proposed the use of an enhanced Z-buffer algorithm, thus retaining the benefits inherent to the Z-buffer rendering approach. They extended the Z-buffer to represent a pixel location as a record of five fields. During the shadow polygon rendering phase, these pixel records are modified based on whether a point is lit or not. The penumbras are created by representing a distributed light source as a series of point light sources. This approach is processor intensive due to the combination of shadow volume calculations with Z-buffer memory access costs. Crowe's ideas were also extended by Bergeron (1985) to include non-planar polygons and objects.

Appendix E
Physics

Simulating Newtonian physics through the use of quantities such as mass, acceleration, velocity, friction, momentum, force, etc allows for the prediction of object behaviour under certain conditions (Halliday et al, 2007). For example, through physics modelling we can simulate the expected behaviour of several stacked barrels falling over or even an explosion ripping through a bunker complex.

Physics modelling is generally implemented as part of a physics engine. Physics engines are classified into two classes: real-time engines such as the Havok physics engine and high-precision physics engines such as those used by scientists. *Real-time physics engines* “approximate” physics modelling to balance computational accuracy with the speed of the simulation (as the case with our quality scaling). *Scientific physics engines* are employed by organisations like NASA and universities for various simulations, for example, Figure E.1 shows the computational fluid dynamics model used for simulating the air flow around a space shuttle during atmospheric re-entry.

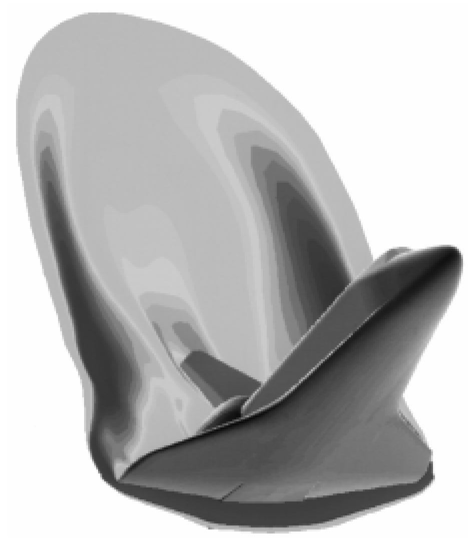

Figure E.1 Simulated air flow around a space shuttle during atmospheric re-entry.

The shown computational fluid dynamics model requires an incredible amount of processing power to simulate (Belleman et al, 2008). This is mostly due to the use of numerical methods and advanced algorithms when analysing the flow of particles – each particle is assigned a force vector which are then combined across the entire region to illustrate the resulting particle flow (Reeves, 1983).

When adding Newtonian physics to a game we must always keep processing constraints in mind. Our biggest problem is not performing the physics calculations but dealing with a fluctuating frame rate and rounding errors that can result in unrealistic motion (Witken and Heckbert, 1994). On the other hand, increasing data precision will solve the problem of rounding errors (Reeves and Blau, 1985) but with a significant impact on CPU and/or GPU usage.

We will now model Newtonian physics by looking at the conservation and transfer of momentum as well as the modelling of gravitational pull, trajectories, friction and object collision.

E.1 Linear Momentum

Action-oriented games without collisions would simply not work. Whether it is a projectile fired from a weapon striking a monster, a car skidding across the Daytona Speedway or the player activating a switch; without the ability to simulate one object striking another we would simply not "have game".

At the core of collision simulation is the conservation and transfer of momentum (Moore and Wilhelms, 1988). The *conservation of momentum* is described as a rule of nature stating that if we have a closed system of objects, without any external interaction, then the total momentum of this system will remain constant. This rule links back to Newton's first law of motion, that is, a body in motion will remain in motion unless a net force is exerted upon it. Building on this; Newton's third law of motion states that for every action there is an equal and opposite reaction – a law that can be proven by considering the conservation of momentum.

To understand conservation of momentum, consider a game of squash in a perfect world where no energy is lost when the ball hits the squash court's wall (in the real word energy will be released in the form of sound, heat and deformation the moment the ball hits the wall, thus resulting in a slower velocity (and less momentum) after the collision than before. However, in a perfect world we don't consider loss in momentum and the velocity of the ball remains the same after the collision than as before.

The *transfer of momentum* describes the situation where a collision occurs and momentum is transferred from the one object to the other. Thus, the lost of momentum at the one side must equal the momentum gained at the other (assuming conservation of kinetic energy as well as momentum before and after the collision). This concept is described mathematically as follows:

$\Delta p_{object1} = -\Delta p_{object2}$, where Δp is the change in momentum of each object.

A well known example demonstrating the conservation and transfer of momentum is Newton's cradle – a device consisting of five (or more) pendulums neighbouring one another. Figure E.2 shows Newton's cradle, when the midair pendulum is released, it will collide with the left-most static pendulum. On impact, energy is transferred from one pendulum to the other until the right-most pendulum is pushed outwards by the transferred force. The motion will eventually cease due to a continuous energy loss (mostly released as sound energy i.e. "clacking" sounds).

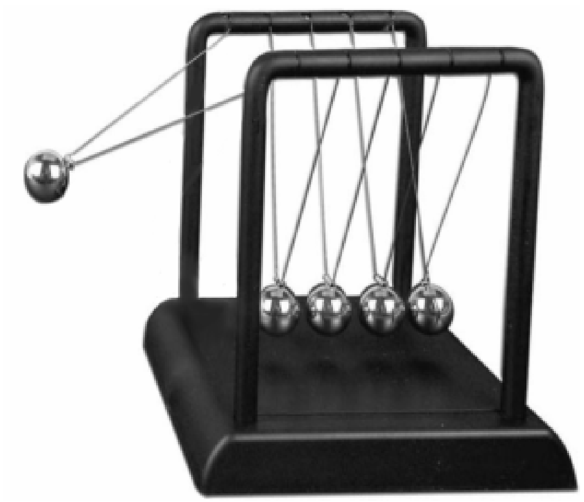
Figure E.2 Newton's cradle used for demonstrating the conservation/transfer of momentum and energy.

To fully understand perfect collisions and the conservation of momentum, consider the two objects shown in Figure E.3.

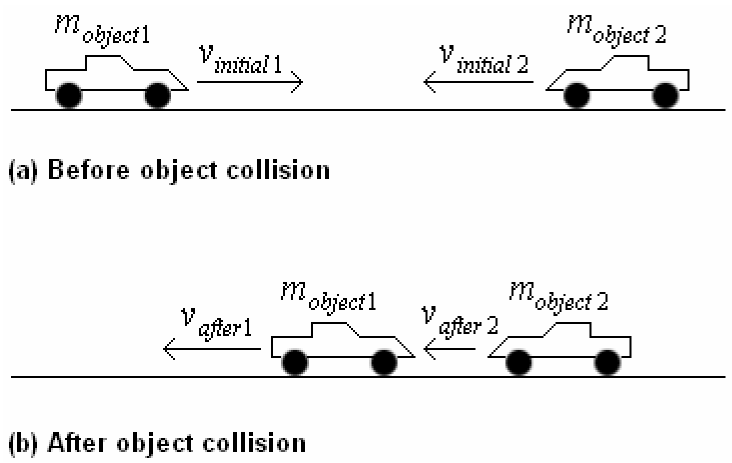

Figure E.3 Collision and the transfer of momentum.

Both objects have a mass ($m_{object1}$ and $m_{object2}$) and initial velocity ($v_{initial1}$ and $v_{initial2}$). After collision each will have a new velocity – two unknown values at this stage (v_{after1}

and v_{after2}). Using these variables we can now describe the conservation of momentum mathematically using the following equation:

$$m_{object1} \times v_{initial1} + m_{object2} \times v_{initial2} = m_{object1} \times v_{after1} + m_{object2} \times v_{after2}$$

One problem with this equation is that we normally wish to calculate each object's vector velocity after the collision, something which is impossible because we'll always end up with two unknowns. For example, say object 1 has a mass of 250kg and an initial velocity of 1200m/s while object 2 has a mass of 300kg and an initial velocity of 2400m/s, then by substituting these values in the above equation, we get:

$$250kg \times 1200m/s + 300kg \times 2400m/s = 250kg \times v_{after1} + 300kg \times v_{after2}$$

The only logical approach is to combine this equation with something we already know, in this case the conservation of energy, specifically the *conservation of kinetic energy*.

Kinetic energy is energy stored in a moving object, or more specifically, the mechanical work needed to accelerate this object from rest to its current state. *Mechanical work* is the total amount of energy transferred to an object through the application of force. The simplest way of calculating work, measured in joule (J), is to use the following formula:

$W = Fd$, where F is the force exerted on the object and *d* the distance travelled by the object.

This formula can also be written as:

$W = \frac{1}{2}mv^2$, with *m* the mass of the object and *v* its velocity.

Applying external work to an object causes a change in its kinetic energy. For example, say an object has an initial kinetic energy of $E_{k\_initial}$ and some force is applied to it resulting in a new kinetic energy, E_{k_final}, then we can represent the relation between work and kinetic energy as follows:

$$\begin{aligned} W &= \Delta E_k \\ &= E_{k\_final} - E_{k\_initial} \end{aligned}$$

Kinetic energy (E_k) is the ability to do work and can easily be calculated using the following equation:

$E_k = \frac{1}{2}mv^2$, with *m* the mass of the object in kilograms and *v* its velocity in meters per second.

Kinetic energy, akin to work, is measured in Joules (J), with one Joule being equal to one kilogram-meter squared per second squared (kgm^2/s^2). This energy remains constant before and after a collision – a condition described as the *conservation of kinetic energy*. In the real world energy will of course be lost in the form of sound, heat and deformation; however, this is only something that will be considered for the implementation of a scientific physics engine. Using this conservation property we can now describe the total kinetic energy before and after a collision via the following equation:

$$\frac{1}{2}m_{object1} \times v_{initial1}{}^2 + \frac{1}{2}m_{object2} \times v_{initial2}{}^2 = \frac{1}{2}m_{object1} \times v_{after1}{}^2 + \frac{1}{2}m_{object2} \times v_{after2}{}^2$$

We can now use this equation in combination with the previous listed one describing the conservation of momentum to solve the given example's two unknown velocities following the collision:

$$250kg \times 1200m/s + 300kg \times 2400m/s = 250kg \times v_{after1} + 300kg \times v_{after2}$$

$$\frac{1}{2}(250kg)(1200m/s)^2 + \frac{1}{2}(300kg)(2400m/s)^2 = \frac{1}{2}(250kg)v_{after1}{}^2 + \frac{1}{2}(300kg)v_{after2}{}^2$$

The simplest approach would be to write V_{after1} in terms of V_{after2} for the second equation, substituting it into the first equation and solving V_{after2}.

Our treatment of particles extends this discussion by looking at the simulation of bouncing objects and inter-object collision detection and response.

E.2 Gravitational Pull

When looking at any early 1990s side-scrolling game, such as *Super Mario World* or *Commander Keen*, one can quickly see the effect of gravity on the player. For example, jumping vertically into the air is quickly followed by the game character returning to its previous position. This is an early example of gravity in games with modern games modelling gravity much more closely.

Gravity is the natural phenomenon where objects attract each other due to each object being surrounded by a gravitational field. This field, interpreted as an attractive power, exerts a pulling force on all surrounding objects, as shown in Figure E.4.

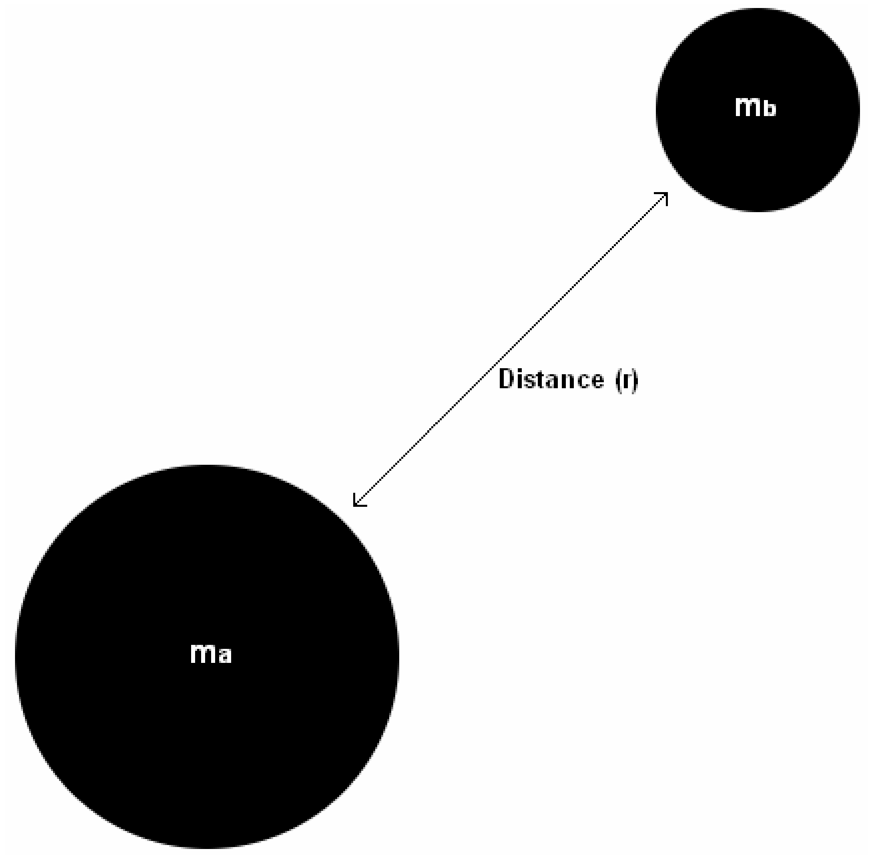

Figure E.4 The gravitational pull between two objects of mass m_a and m_b, respectively.

Each of the two objects shown in Figure E.5 will experience the effect of gravity, with the exact gravitational force between the two objects given by the following equation:

$F = \frac{G \times m_a \times m_b}{r^2}$, where G is the universal gravitational constant (equal to $6.67 \text{x} 10^{-11} \text{Nm}^2/\text{kg}^2$), m_a the mass of the one object and m_b the mass of the other with r the distance in meters between the two objects.

Simulating gravity in games does not generally require advanced calculations that involve the universal gravitational constant or the exact mass of an object. For example, when modelling gravity for an object being dropped to the ground, we can start with the assumption that the acceleration of this object will be 9.8m/s^2 regardless of its mass (standard acceleration due to the earth's gravitational field). We can now define the velocity and position of this object as follows:

$$V_{new} = V_{old} + (9.8)t$$

$$\begin{aligned} Pos_{new} &= Pos_{old} + (v_{old} \times t) + (\frac{1}{2} \times a \times t^2), \\ &= Pos_{old} + (v_{old} \times t) + (\frac{1}{2} \times 9.8m/s^2 \times t^2) \end{aligned}$$

Now, let’s assume a crate is dropped at an initial velocity of 0 m/s from a position located at coordinates (0, 17, 0) as shown in Figure E.5.

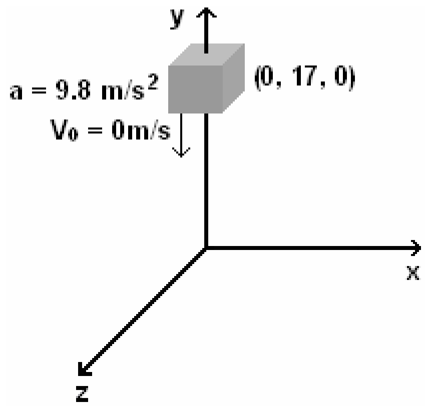

Figure E.5 Gravitational attraction of an object towards the *zx*-plane.

Substituting these values into the above given equations yield the following equations (assuming the coordinate y = 17 equates to a virtual height of 17 meters):

$$\begin{aligned} V_{new} &= 0m/s + (9.8)t \\ &= 9.8t \end{aligned}$$

$$\begin{aligned} Pos_{new} &= 17m + (0m/s \times t) + (\frac{1}{2} \times 9.8m/s^2 \times t^2), \\ &= 17m + (\frac{1}{2} \times 9.8m/s^2 \times t^2) \end{aligned}$$

We can now implement these equations in the following manner – thus simulating gravity:

```
/* initialise the object’s initial position */
float objectXPos = 0;
float objectYPos = 17;
float objectZPos = 0;
```

```
/* set the object's initial velocity */
float objectXVelocity = 0;
float objectYVelocity = 0;
float objectZVelocity = 0;

/* initialise the object's rate of fall – hence its gravity */
float worldGravityConstant = 1.5f;

/* use a loop to update the object's position and velocity until the zx-plane is
   reached */
while(objectYPos > 0)
{
    /* increase the velocity as the object falls */
    objectYVelocity = objectYVelocity + worldGravityConstant;

    /* calculate the object's new position */
    objectYPos = objectYPos + objectYVelocity;
}
```

This object will only fall in a straight vertical line, by incrementally adjusting its x-coordinate in the loop, for example, we can simulate a curved falling trajectory as shown in Figure E.6.

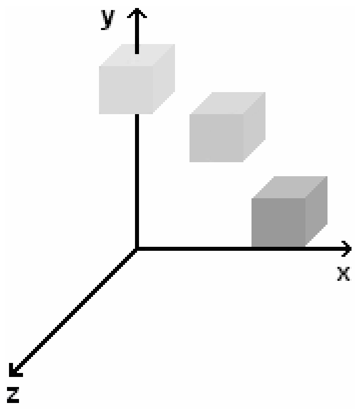

Figure E.6 Gravitational attraction of an object thrown in the x-direction.

We can now modify the above listed code snipped to simulate a curved falling trajectory as follows:

```
/* initialise the object's initial position */
float objectXPos = 0;
float objectYPos = 17;
float objectZPos = 0;
```

```
/* set the object’s initial velocity */
float objectXVelocity = 0;
float objectYVelocity = 0;
float objectZVelocity = 0;

/* initialise the object’s rate of fall – hence its gravity */
float worldGravityConstant = 1.5f;

/* use a loop to update the object’s position and velocity  until the zx-plane is
   reached */
while(objectYPos > 0)
{
    /* increase the velocity as the object falls */
    objectYVelocity = objectYVelocity + worldGravityConstant;

    /* calculate the object’s new y-position */
    objectYPos = objectYPos + objectYVelocity;

    /* calculate the object’s new x-position by adding a constant x velocity */
    objectXPos = objectXPos + 3;
}
```

E.3 Trajectory Paths

Without accurate projectile simulation, we would not be able to model bomb drops from aeroplanes, a kick off in a football game or the trajectory of a baseball after being hit by a batter. Figure E.7 shows the trajectory path of a ball being kicked in the positive *x*-direction.

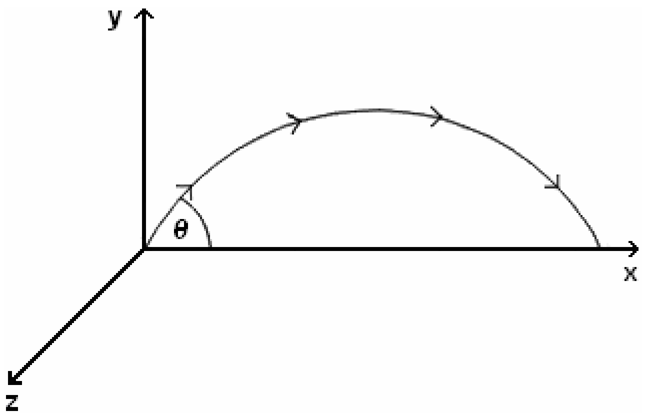

Figure E.7 The trajectory path of a ball being kicked in the positive *x*-direction.

Trajectory can be described as the path or course travelled by an object. Calculating this path often requires the consideration of gravitational forces, aerodynamic factors, wind shear, etc. For most game-based implementations we'll assume uniform gravity while negating wind and other aerodynamic factors. For example, to model the trajectory path shown in Figure E.7 we can define the ball's initial velocity in terms of an *x*- and *y*-component as follows (with θ the inclination angle):

$$V_x = V_{initial} \times \cos\theta$$
$$V_y = V_{initial} \times \sin\theta$$

We can also assume that V_y will equal "0" at the apex of the arch (the maximum height reached by the projectile).

Modelling a trajectory path involves applying a constant velocity along the *x*-axis (in the case of the above shown path) as well as the effect of gravity in the direction of the negative *y*-axis. We also factor in air resistance without needlessly complicating our simulation. The following code sample simulates a trajectory path as illustrated in Figure E.7:

```
/* initialise the object's initial position */
float objectXPos = 0;
float objectYPos = 0;
float objectZPos = 0;

/* set the object's initial velocity */
float objectXVelocity = 0;
float objectYVelocity = 0;
float objectZVelocity = 0;

/* initialise the object's rate of fall – hence its gravity */
float worldGravityConstant = 1.5f;

/* set the inclination angle to 45 degrees in radians */
float initialAngle = 0.79

/* set the air resistance that will be factored in to simulate the deceleration of the
   projectile */
float airResistance = 0.01f

/* calculate the velocity's x- and y-component */
objectXVelocity = objectXVelocity*cos(initialAngle);
objectYVelocity = objectYVelocity*sin(initialAngle);
```

```
/* use a loop to update the object’s position and velocity until the zx-plane is
   reached */
while(objectYPos > 0)
{
     /* update the object’s velocity */
    objectYVelocity = objectYVelocity + worldGravityConstant;
    objectXVelocity = objectXVelocity - airResistance;

    /* calculate the object’s new y-position */
    objectYPos = objectYPos + objectYVelocity;

    /* calculate the object’s new x-position */
    objectXPos = objectXPos + objectXVelocity;
}
```

E.4 Friction

Friction, stemming from electromagnetic forces between atomic particles, is an energy consuming force between two objects in contact. The most common form of friction is known as Coulomb friction. *Coulomb friction* is an approximation stating that the maximum force exerted by friction (F_f) is always less than or equal to the direct normal force (F_n) between two objects multiplied by the material’s friction coefficient (μ):

$$F_f \leq F_n \times \mu$$

The *normal force* (shown in Figure E.8) is a force component perpendicular to the surface of contact with the coefficient of friction an empirically determined constant that varies depending on the type of material surface and whether the surface is perfectly clean, etc.

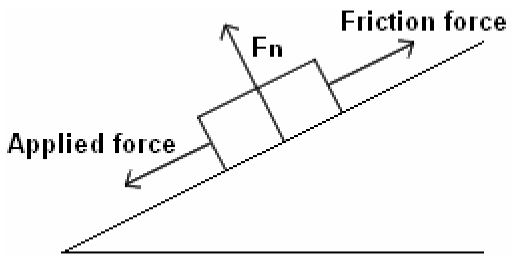

Figure E.8 The normal, friction and applied (sliding) forces exerted on an object.

Table E.1 gives some of the most common friction coefficients; also note that friction varies depending on whether an object is static or in motion.

Material	**Static**	**In Motion (kinetic)**
Aluminium on aluminium	1,05-1,35	1,4
Aluminium on steel	0,61	0,47
Copper on cast iron	1,05	0,29
Copper on steel	0,53	0,36
Glass on glass	0,9 - 1,0	0,4
Glass on nickel	0,78	0,56
Leather on wood (along the grain)	0,61	0,52
Nickel on nickel	0,7-1,1	0,53
Nylon on nylon	0,15 - 0,25	
Steel on steel (high level hardness)	0,78	0,42
Steel on steel (relative hardness)	0,74	0,57
Wood on wood (against the grain)	0,54	0,32
Wood on wood (along the grain)	0,62	0,48

Table E.1 Common coefficients of friction.

We generally calculate the force required to move a static object via the following equation:

$F_f = m \times g \times \mu_{static}$, where *m* is the mass of the object, *g* the gravitational constant (9.8m/s^2) and μ the material's static friction coefficient.

The object will only move once a force greater than F_f is applied to it, after which its friction coefficient normally decreases. For example, consider an aluminium object weighing 90 kilograms placed on a flat polished steel surface – we can calculate the maximum force exerted by friction as follows:

$$\begin{aligned} F_f &= m \times g \times \mu_{static} \\ &= 90kg \times 9.8m/s^2 \times 0.61 \\ &= 538.02N \end{aligned}$$

We will thus require a force of at least 538.03N to move this object, once it is in motion we can recalculate it frictional force using aluminium on steel's kinetic friction coefficient:

$$\begin{aligned} F_f &= m \times g \times \mu_{kinetic} \\ &= 90kg \times 9.8m/s^2 \times 0.47 \\ &= 414.54N \end{aligned}$$

Friction on a flat plane can be modelled just like air resistance (which is in fact a form of friction):

```
/* initialise the object’s initial position */
float objectXPos = 0;
float objectYPos = 0;
float objectZPos = 0;

/* set the object’s initial velocity */
float objectXVelocity = 15;
float objectYVelocity = 0;
float objectZVelocity = 0;

/* set the friction value */
float friction = 1.5f

/* use a loop to update the object’s position and velocity  until the object’s speed
   reaches zero */
while(objectXVelocity > 0)
{
     /* update the object’s velocity */
    objectXVelocity = objectXVelocity - friction;

    /* calculate the object’s new x-position */
    objectXPos = objectXPos + objectXVelocity;
}
```

E.5 Simulating Object Collisions

Let’s start with a two dimensional “asteroid field” from Atari’s 1979 cult-hit, Asteroids. This game, as shown in Figure E.9, is heavily dependent on object collisions such as asteroids colliding with other asteroids, alien spaceships or with the player’s ship.

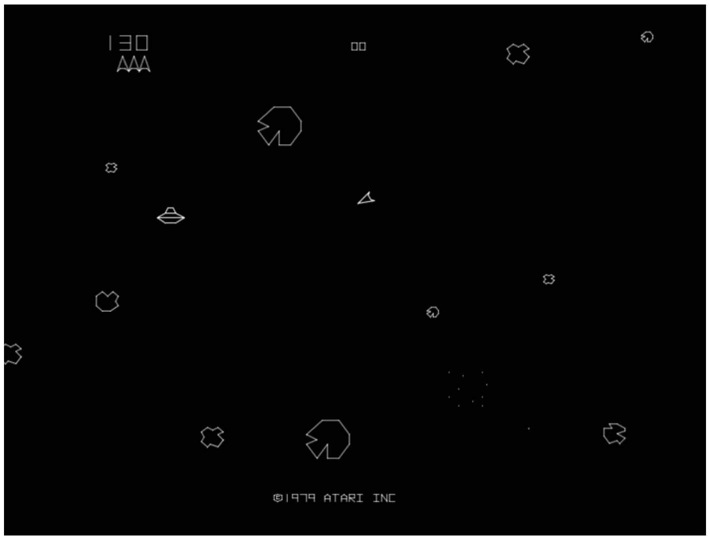

Figure E.9 Screenshot of Atari's arcade game Asteroids.

The game Asteroids illustrates the basic problem of collision detection and response in one of the simplest forms possible. Before, however, discussing object-to-object collision response as encountered in Asteroids, let's look at the game Breakout.

Breakout features a ball that can either bounce from the boundaries of the game window or movable paddle while also destroying bricks upon collision. Bouncing the ball off the screen boundaries requires very basic collision detection mainly because we already know where the boundaries of the screen are while at the same time only considering collisions with two horizontal and two vertical edges. Also, an object such as the ball in Breakout will always reflect at an angle equal and opposite to its initial incoming angle (illustrated in Figure E.10).

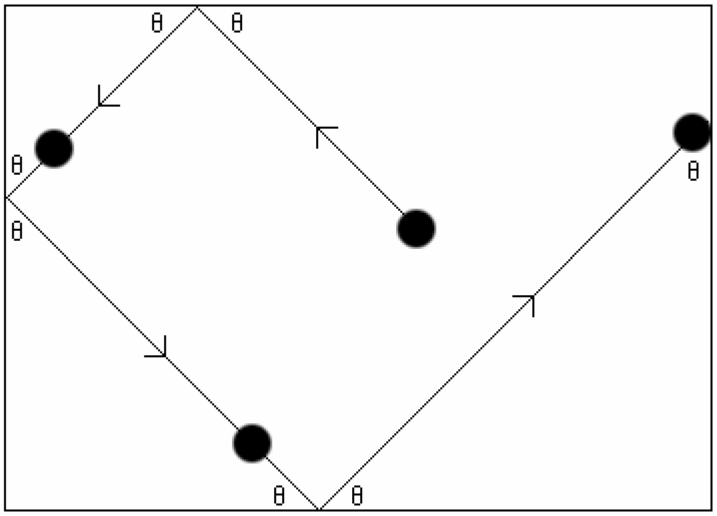

Figure E.10 A ball always reflects at an angle equal and opposite to its initial incoming angle.

Now, considering the shown image; it can be deduced that when the ball hits either vertical edge, then its direction can be changed by reversing the x-component of its velocity. Similarly, reversing the y-component of the ball's velocity upon collision with one of the horizontal edges will result in a perfect direction change:

```
/* initialise the object's initial position */
float objectXPos = 5;
float objectYPos = 2;
float objectZPos = 0;

/* set the object's initial velocity */
float objectXVelocity = 15;
float objectYVelocity = 20;
float objectZVelocity = 0;

/* update the object's velocity due to a vertical collision */
if(objectXPos > LEFT_EDGE || objectXPos < RIGHT_EDGE)
{
     /* update the object's velocity */
    objectXVelocity = -objectXVelocity;

    /* calculate the object's new x-position */
    objectXPos = objectXPos + objectXVelocity;
}

/*update the object's velocity due to a horizontal collision*/
if(objectYPos > BOTTOM_EDGE || objectYPos < TOP_EDGE)
{
     /* update the object's velocity */
    objectYVelocity = -objectYVelocity;

    /* calculate the object's new y-position */
    objectYPos = objectYPos + objectYVelocity;
}
```

This technique can now be extended to simulate one object bouncing off another. The simplest approach would be to test for horizontal and vertical collisions with the sides of a bounding volume. For example, consider the screenshot of the Asteroids clone in Figure E.11 where the bounding volume of each object is shown (these volumes are specified using the contained object's minimum and maximum *x*- and *y*-values).

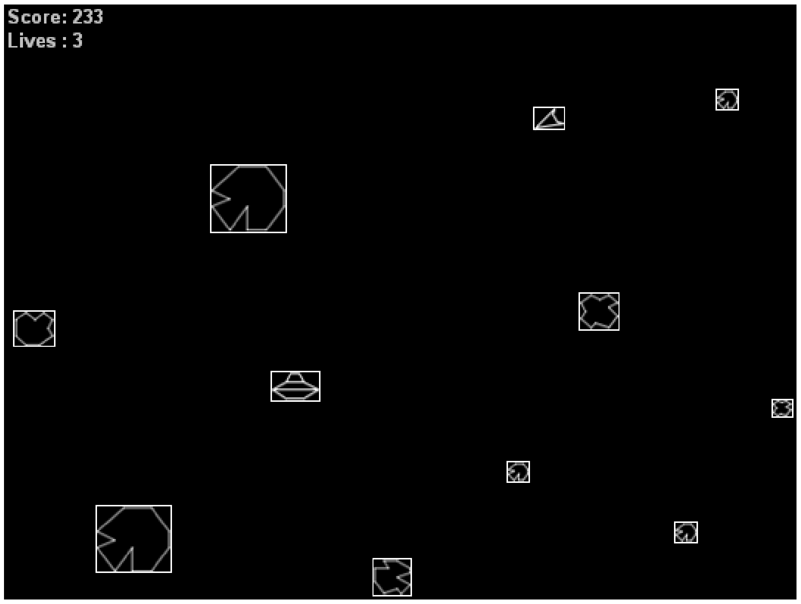

Figure E.11 Using bounding boxes to simulate inter-object collisions.

We can implement this approach in much the same way as with our horizontal and vertical screen boundary collision example – for instance, when we have a ball bouncing off objects as shown in Figure E.12, then we can change its direction by reversing the x-component of its velocity when it hits a vertical edge of another object. Similarly, reversing the y-component of the ball's velocity upon collision with one of the horizontal edges will result in a perfect direction change.

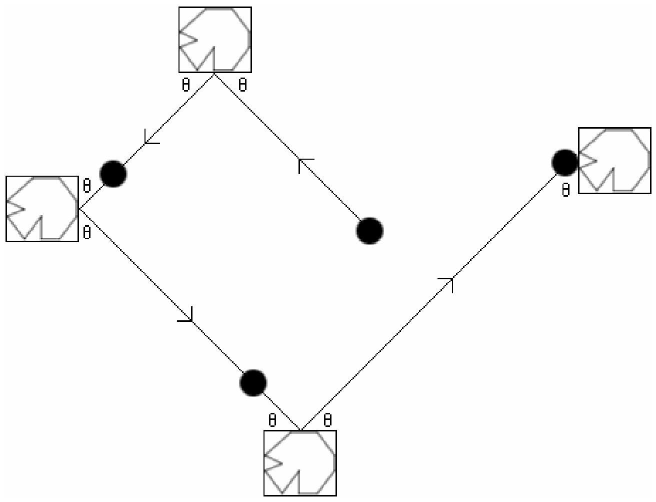

Figure E.12 Inter-object collisions – the same rules hold true as with screen boundary collisions.

The above given object collision approach works extremely well for horizontal and vertical surfaces, but in nearly all action-oriented games written today we'll need to calculate vector reflections for arbitrarily rotated surfaces. For example, consider the object shown in Figure E.13. This object has several flat planes, with each of these positioned at an arbitrary angle.

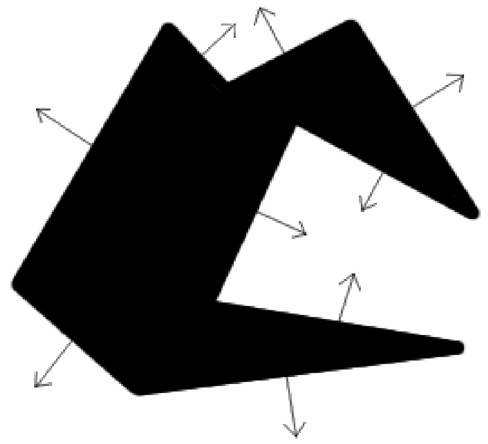

Figure E.13 An object with numerous arbitrarily positioned faces (the normal of each shown).

The core of collision detection, when dealing with arbitrarily positioned faces, is vector calculations; specifically the calculation of a reflection vector when we have an initial vector direction and a normal to the plane (Blinn, 1977). We've already looked at vector and normal calculations in Appendix C and will now look at an example to illustrate vector-based object reflections for arbitrarily rotated surfaces.

Figure E.14 illustrates our vector reflection problem; showing an incoming vector **I**, the surface normal **N** and the unknown reflection vector **R**.

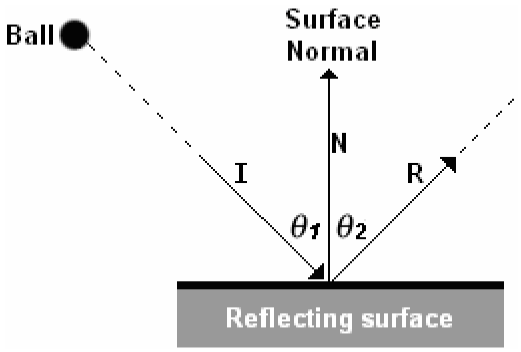

Figure E.14 Vector reflection for an arbitrarily rotated surface.

We use vector addition to create a third, composite vector. This process involves summing the related scalar components of two successive vectors (using the head-to-tail rule). In Figure E.14 we have three vectors, namely, **I**, **N** and **R**; using these vectors we define a third and forth vector, **P** and **Q** (the resultant of **I** and **N** and **R** and **N** respectively) by summing the scalars of vector **I** and **N** and **R** and **N** in the following manner (graphically illustrated in Figure E.15):

P = **I** + **N**
 = (*Ix* + *Nx*, *Iy* + *Ny*, *Iz* + *Nz*)
 = (*Px*, *Py*, *Pz*).

Q = **R** + **N**
 = (*Rx* + *Nx*, *Ry* + *Ny*, *Rz* + *Nz*)
 = (*Qx*, *Qy*, *Qz*).

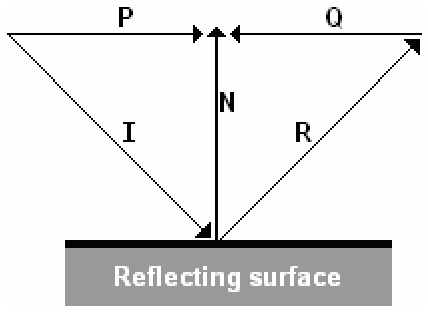

Figure E.15 The head-to-tail rule, creating a third composite vector.

Using the above given information, we can now algebraically calculate the reflection vector by stating that **P** = **Q** and substituting the first equation into the second:

I + **N** = **R** + **N**
 R = **N** + **(I** + **N)**
  = 2**N** + **I**

Returning to our example, if the object has an incoming speed with an x-component of -16 and a y-component of 8 then we can calculate the vector of reflection (thus the exiting speed of the object) in the following manner (the normal in this case equals y = 1):

R = 2**N** + **I**
 = 2(-**I** · |**N**|)*|**N**| + **I**
 = 2[(*Ix*, *Iy*) · |(*Nx*, *Ny*)|]*|(*Nx*, *Ny*)| + (*Ix*, *Iy*)
 = 2[-(-16, 8) · |(0, 1)|]*|(0, 1)| + (-16, 8)
 = 2[(16, -8) · |(0, 1)|]*|(0, 1)| + (-16, 8)
 = 2(16*0 – 8*1)*|(0, 1)| + (-16, 8)
 = 2(– 8)*|(0, 1)| + (-16, 8)
 = -16*(0, -1) + (-16, 8)
 = (0, 16) + (-16, 8)
 = (*0* – 16, 16 + 8)
 = (– 16, 24).

Collision detection and response in modern games often require considerable resources to implement. A number of collision detection algorithms (such as the detection of collisions using hierarchy trees) have consequently been developed to simulate collisions at various degrees of accuracy. The study of these algorithms is, however, beyond the scope of this thesis and our physics engine implementation.

Appendix F
The DXUT Framework

The Direct3D Utility Framework, or DXUT, is a high-level framework built on top of Direct3D. This framework provides a series of functions, call-backs, structures, constants and enumerations to reduce the complexity of low-level Direct3D routines. It encapsulates the Win32 and Direct3D APIs for ease of use, making it easier to create Direct3D applications. To summarise, DXUT allows simplified window creation, enables rapid Direct3D device setup and initialisation as well as the easy handling of Windows messages.

The DXUT framework provides a vast array of functionality, from basic window creation, Direct3D device initialisation and the control of these components to more advanced elements such as 3-D mesh control, camera control and the creation of graphical user interfaces. We will now look at the most important functional components provided by this framework.

The process of window creation and control using the DXUT framework is relatively simple when compared to using the Win32 API which entails created a window, registering a window class, creating a window object and handling messages to and from the window. The DXUT framework simplifies this process in the sense that it is not necessary to register the window class (using the **`WNDCLASSEX`** structure) or to create the window using the **`AdjustWindowRect, CreateWindow`** and **`ShowWindow`** functions. The following series of DXUT function calls manages this entire window creation process:

```
/ * initialise the DXUT framework */
DXUTInit(true, true, NULL);

/* configure mouse cursor settings for full-screen usage */
DXUTSetCursorSettings(true, true);

/* create the application window */
DXUTCreateWindow(      L"DXUT Sample", NULL, NULL, NULL, NULL,
                       NULL);

/* create the Direct3D device */
DXUTCreateDevice(true, 800, 600);

/* enter the main DXUT framework render loop */
DXUTMainLoop(NULL);
```

The **`DXUTInit`** function initialises DXUT by taking three parameters, namely a Boolean value used for the processing of command-line arguments (with the most common ones listed in Table F.1), another Boolean parameter controlling whether an error message box is to be displayed whenever an error occurs and a string value for the specification of additional command-line parameters.

Argument	**Description**
`-adapter:X`	Defines the specific hardware adapter to use.
`-automation`	Enables user interface navigation via the keyboard (enabled by default)
`-constantframetime`	Defines a specific time per frame lapse when the desired effect is to render some scene at a FPS value less than real-time.
`-forceapi:X`	Forces the application to use either the Direct3D 9 or Direct3D 10 API.
`-forcehal`	Forces the use of a HAL device type.
`-forcehwvp`	Forces the use of hardware vertex processing (not applicable for Direct3D 10 – only supported by the Direct3D 9 API).
`-forcepurehwvp`	Forces the use of pure hardware vertex processing (not applicable for Direct3D 10 – only supported by the Direct3D 9 API).
`-forceref`	Forces the use of a reference device type.
`-forceswvp`	Forces the use of software vertex processing (not applicable for Direct3D 10 – only supported by the Direct3D 9 API).
`-forcevsync:X`	Specifies whether vertical sync is to be used (**`X`** is set to ‘0’ to disable vertical sync).
`-fullscreen`	Forces the application into full-screen mode on startup.
`-height:X`	Specifies the default window height.
`-noerrormsgboxes`	Disables DXUT’s error message boxes.
`-nostats`	Disables the display of statistics such as the current number of frames per second.
`-output:X`	Forces the use of a specific adapter output (only supported by the Direct3D 10 API)
`-quitafterframe:X`	Sets an exit frame – i.e. forcing the application to terminate after the specified frame, **`X`**, has been rendered.
`-startx:#`	Sets the *x*-coordinate of the window’s upper left corner when running in windowed mode.
`-starty:#`	Sets the *y*-coordinate of the window’s upper left corner when running in windowed mode.

`-width:X`	Specifies the default window width.
`-windowed`	Forces the application into windowed mode on startup.

Table F.1 DXUTInit command-line parameters.

The next called function, **`DXUTSetCursorSettings`**, sets the visibility and clipping of the mouse cursor when used in full-screen mode. This function takes two parameters, the first a Boolean value specifying whether the mouse cursor will be visible for a window in full-screen mode (**`true`** if yes), and the second, also a Boolean value, defining whether the cursor will be limited from leaving the screen boundaries for a full-screen window (**`true`** if yes).

`DXUTCreateWindow` creates the application window through the initialisation of six parameters, namely, a string value defining the window's caption, a **`HINSTANCE`** handle to the application instance ('NULL' by default), a **`HICON`** handle to the window's icon ('NULL' by default), a **`HMENU`** handle to the window's menu resource ('NULL' for no menu) and the upper left *x*- and *y*- window coordinates.

We create the actual Direct3D 10 device by calling the **`DXUTCreateDevice`** function. Its first parameter takes a Boolean value specifying whether the application will launch in windowed (**`true`**) or full-screen mode (**`false`**). **`DXUTCreateDevice`**'s final two parameters set the initial width and height of the back buffer, respectively.

The **`DXUTMainLoop`** function enters the main DXUT framework render loop (the main message loop), updating and rendering each frame via callbacks to the application. It takes one parameter, namely a handle to an accelerator table – this parameter is set to 'NULL' when no accelerator table is defined. *Accelerator tables* are created as resources and used for the translation of keyboard messages received from the message queue. One example of a common accelerator is the "Ctrl+S" key combination used as shortcut for the "File Save" menu item.

All these functions return the value "**`S_OK`**" if successful. In the event of a failure they return one of the error codes listed in Table F.2. Calling the **`DXUTGetExitCode`** function returns an exit code with '0' indicating successful execution.

Error code	**Description**
`DXUTERR_CREATINGDEVICE`	Unable to create a Direct3D device.
`DXUTERR_CREATINGDEVICEOBJECTS`	A problem has been encountered while creating the Direct3D device objects.
`DXUTERR_DEVICEREMOVED`	The initialised Direct3D device is no longer accessible.
`DXUTERR_MEDIANOTFOUND`	The requisite media could not be loaded.

DXUTERR_NOCOMPATIBLEDEVICES	Unable to find any Direc3D capable devices.
DXUTERR_NODIRECT3D	Direct3D could not be initialised.
DXUTERR_NONZEROREFCOUNT	The Direct3D device was not properly released by a previous application.
DXUTERR_RESETTINGDEVICE	Unable to reset the Direct3D device.
DXUTERR_RESETTINGDEVICEOBJECTS	An issue was encountered while resetting the Direct3D device objects.

Table F.2 Error codes returned by DXUT functions.

Using the Win32 API, after registering the window class and creating the window, we enter the main message loop by declaring an empty **MSG** structure, **msg**, and passing it as parameter to the **WndProc** function. Using DXUT we no longer need to define a **MSG** structure or **WndProc** function for the handling of messages sent to and from the window. We will now rather create a series of callback functions, passing each one as a parameter to the appropriate **DXUTSetCallback*** DXUT function. For example, the following callback function handles all keyboard events:

```
void CALLBACK OnKeyPress(UINT nChar, bool bKeyDown,
                                  bool bAltDown, void* pUserContext)
{
     /* test whether some key is being pressed */
    if(bKeyDown)
    {
        switch(nChar)
        {
            case VK_TAB: //if 'Tab' is pressed do something
                  break;
        }
    }
}
```

This keyboard event callback function, **OnKeyPress**, is then passed as parameter to the **DXUTSetCallbackKeyboard** function:

```
DXUTSetCallbackKeyboard(OnKeyPress, NULL);
```

This function, initialising the previously defined callback function, takes two parameters, the first being a pointer to a **LPDXUTCALLBACKKEYBOARD** keyboard event callback function, and the second a pointer to some user-specific variable passed to the callback function – by default set to 'NULL'.

The **`LPDXUTCALLBACKKEYBOARD`** DXUT keyboard event callback function is called every time a keyboard event occurs. It is declared as follows in the DXUT.h header file:

```
VOID LPDXUTCALLBACKKEYBOARD(
  UINT nChar,
  bool bKeyDown,
  bool bAltDown,
  void* pUserContext
);
```

Its first parameter holds a virtual-key code describing the pressed key (the most commonly used virtual-key codes are given in Table F.3). The second parameter, **`bKeyDown`**, holds the Boolean value 'true' if a key is currently being pressed with the **`bAltDown`** parameter set to 'true' if the 'Alt' key is also being pressed. The last parameter, **`pUserContext`**, takes a pointer to a user-specific variable passed to the callback function – by default set to 'NULL'.

Constant	**Description**
VK_LBUTTON	Left mouse button.
VK_RBUTTON	Right mouse button.
VK_BACK	Backspace key.
VK_TAB	Tab key.
VK_RETURN	Enter key.
VK_ESCAPE	Escape key.
VK_UP, VK_DOWN, VK_LEFT, VK_RIGHT	Up, down, left and right keys respectively.
VK_NUMPAD0 to **VK_NUMPAD9**	Numeric keypad keys '0' to '9'.
VK_F1 to **VK_F24**	F1 to F24 keys.

Table F.3 Virtual-Key codes.

DXUT provides a number of these so-called application-defined callback functions. The above defined **`OnKeyPress`** function is, for example, a **`LPDXUTCALLBACKKEYBOARD`** keyboard event callback. These DXUT event callback functions simplify the message handling process. In addition to a keyboard event callback we also have to define a device acceptable callback function (set using the **`DXUTSetCallbackD3D10DeviceAcceptable`** DXUT initialisation function), a device created callback function (set via **`DXUTSetCallbackD3D10DeviceCreated`**), a swap chain resized callback function (set using **`DXUTSetCallbackD3D10SwapChainResized`**), a swap chain release callback function (set via **`DXUTSetCallbackD3D10SwapChainReleasing`**), a device destroyed callback function (set via **`DXUTSetCallbackD3D10DeviceDestroyed`**) and a frame render callback function (set through the **`DXUTSetCallbackD3D10FrameRender`** DXUT initialisation function). In addition to these callback functions we also need to create a window message callback function dealing with Windows messages (set using **`DXUTSetCallbackMsgProc`**), a callback function dealing with frame updates (set by **`DXUTSetCallbackFrameMove`**) and a callback function that allows for the change of device settings before the creation of the device (set through the **`DXUTSetCallbackDeviceChanging`** DXUT initialisation function).

`DXUTSetCallbackD3D10DeviceAcceptable` initialises the application specific callback function responsible for building an enumerated list of Direct3D 10 capable devices. It takes two parameters, namely, a pointer to a **`LPDXUTCALLBACKISD3D10DEVICEACCEPTABLE`** callback function and a pointer to a user-defined variable passed to the callback function – 'NULL' by default.

The **`LPDXUTCALLBACKISD3D10DEVICEACCEPTABLE`** callback function returns true for each acceptable Direct3D device. All acceptable Direct3D 10 devices are enumerated into a list by the **`DXUTSetCallbackD3D10DeviceAcceptable`** function. DXUT then selects the best rendering device from this list. This callback is declared as follows in the DXUT.h header file:

```
bool LPDXUTCALLBACKISD3D10DEVICEACCEPTABLE(
  UINT Adapter,
  UINT Output,
  D3D10_DRIVER_TYPE DeviceType,
  DXGI_FORMAT BackBufferFormat,
  bool bWindowed,
  void* pUserContext
);
```

Its first parameter, **`Adapter`**, holds a value indicating the position of the current Direct3D 10 device in a series of enumerated Direct3D 10 video adapters. The second parameter, **`Output`**, holds an index value of the current enumerated video adapter's output (such as a monitor). The **`DeviceType`** parameter holds the current Direct3D 10 capable video adaptor's driver type (commonly set to **`D3D10_DRIVER_TYPE_HARDWARE`** for a hardware device and **`D3D10_DRIVER_TYPE_REFERENCE`** for a reference device). The **`BackBufferFormat`** parameter indicates the back buffer format of the Direct3D 10 device (such as a four-component, 64-bit floating-point format). The next parameter takes a Boolean value that is set to 'true' for windowed application and 'false' for those running in full-screen mode. The final parameter, **`pUserContext`**, is a pointer to a user-specific variable passed to the callback function – 'NULL' by default unless context information for the callback function is needed.

We create a **LPDXUTCALLBACKISD3D10DEVICEACCEPTABLE** callback function which is passed as first parameter to the **DXUTSetCallbackD3D10DeviceAcceptable** DXUT function as follows:

```
/* return 'true' for all acceptable D3D10 devices passed to it */
bool CALLBACK OnDeviceAcceptable(UINT Adapter, UINT Output,
                                 D3D10_DRIVER_TYPE DeviceType,
                                 DXGI_FORMAT BufferFormat,
                                 bool bWindowed, void* pUserContext )
{
        return true;
}

DXUTSetCallbackD3D10DeviceAcceptable(OnDeviceAcceptable,NULL);
```

The **DXUTSetCallbackD3D10DeviceCreated** function sets the created **ID3D10Device** device. This device interface is used for the rendering of primitives as well as the creation of shaders and resources. The callback is used for the allocation of resources and the initialisation of buffers. The **DXUTSetCallbackD3D10DeviceCreated** function takes two parameters, namely, a pointer to a **LPDXUTCALLBACKD3D10DEVICECREATED** callback function and a pointer to a user-define variable passed to the callback function – 'NULL' by default. This function is declared as follows:

```
VOID DXUTSetCallbackD3D10DeviceCreated(
  LPDXUTCALLBACKD3D10DEVICECREATED pCallback,
  void* pUserContext
);
```

The associated **LPDXUTCALLBACKD3D10DEVICECREATED** callback function is declared as follows:

```
HRESULT LPDXUTCALLBACKD3D10DEVICECREATED(
  ID3D10Device * pd3dDevice,
  CONST DXGI_SURFACE_DESC * pBackBufferSurfaceDesc,
  void* pUserContext
);
```

This resource callback function forwards a pointer to the newly created **ID3D10Device** interface – the Direct3D 10 device. This pointer, sent to the **DXUTSetCallbackD3D10DeviceCreated** function, is defined as the first parameter. The second parameter is a **DXGI_SURFACE_DESC** structure with four members describing the width, height, format and multisampling parameters of the surface resource respectively. The third parameter, **`pUserContext`**, is a pointer to a user-specific variable passed to the callback function – 'NULL' by default unless context information for the callback function is needed.

A **`LPDXUTCALLBACKD3D10DEVICECREATED`** callback function can be defined in the following manner:

```
HRESULT CALLBACK OnCreateDevice(ID3D10Device* pd3dDevice,
                         const DXGI_SURFACE_DESC *pBufferSurfaceDesc,
                         void* pUserContext)
{
          /* - set up, create and set the input layout
             - create and set the vertex buffer
             - create and set the index buffer
             - specify the primitive topology
             - load all texture resources
             - initialise the world and view matrices */
}
```

This function is now set using the **`DXUTSetCallbackD3D10DeviceCreated`** DXUT initialisation function:

```
DXUTSetCallbackD3D10DeviceCreated(OnCreateDevice, NULL);
```

We also have to deal with the callbacks sent to the application whenever the Direct3D 10 swap chain is resized (see section 4.5.2), this is done using the **`DXUTSetCallbackD3D10SwapChainResized`** function. This function has two parameters, the first a pointer to a **`LPDXUTCALLBACKD3D10SWAPCHAINRESIZED`** callback function with the second a pointer to a user-specific variable passed to the callback function – 'NULL' by default.

The **`LPDXUTCALLBACKD3D10SWAPCHAINRESIZED`** callback function commonly used to set resources dependent on the back buffer – such as perspective projection matrices based on the field-of-view is declared as follows:

```
HRESULT LPDXUTCALLBACKD3D10SWAPCHAINRESIZED(
  ID3D10Device * pd3dDevice,
  IDXGISwapChain * pSwapChain,
  CONST D3DSURFACE_DESC * pBackBufferSurfaceDesc,
  void* pUserContext
);
```

Its first parameter, **pd3dDevice**, is a pointer to the newly created Direct3D 10 device (**ID3D10Device**). The second parameter is a pointer to an **IDXGISwapChain** interface (see section 4.5.2) with the third holding a pointer to a structure describing the back buffer surface's format. The last parameter, **pUserContext**, is a pointer to a user-specific variable passed to the callback function.

This **LPDXUTCALLBACKD3D10SWAPCHAINRESIZED** swap chain resized callback function, passed to **DXUTSetCallbackD3D10SwapChainResized**, can be defined in the following manner:

```
HRESULT CALLBACK OnSwapChainResize(ID3D10Device* pd3dDevice,
                         IDXGISwapChain *pSwapChain,
                         const DXGI_SURFACE_DESC* pBufferSurfaceDesc,
                         void* pUserContext)
{
          /* - reset the aspect ratio using the back buffer's new width and height
             -  set the perspective projection matrix using the
                new aspect ratio */
}
```

We set this callback function using **DXUTSetCallbackD3D10SwapChainResized**:

```
DXUTSetCallbackD3D10SwapChainResized(OnSwapChainResize);
```

All the Direct3D 10 device resources created in the **LPDXUTCALLBACKD3D10SWAPCHAINRESIZED** callback function must also be released. This is done using a **LPDXUTCALLBACKD3D10SWAPCHAINRELEASING** callback which is set using the **DXUTSetCallbackD3D10SwapChainReleasing** swap chain releasing function. This DXUT function takes two parameters, a pointer to a **LPDXUTCALLBACKD3D10SWAPCHAINRELEASING** callback function and a pointer to a user-specific variable passed to the callback function – 'NULL' by default:

```
HRESULT DXUTSetCallbackD3D10SwapChainReleasing(
  LPDXUTCALLBACKD3D10SWAPCHAINRELEASING pCallback,
  void* pUserContext
);
```

The **LPDXUTCALLBACKD3D10SWAPCHAINRELEASING** callback function has only one parameter, a pointer to a user-specific variable passed to the callback function when context information for the callback function is needed:

```
VOID LPDXUTCALLBACKD3D10SWAPCHAINRELEASING(
  void* pUserContext
);
```

This **`LPDXUTCALLBACKD3D10SWAPCHAINRELEASING`** swap chain releasing callback function, called whenever the swap chain created in **`OnSwapChainResize`** is being released can be defined as follows:

```
void CALLBACK OnSwapChainReleasing(void* pUserContext )
{
          /* release all the Direct3D 10 resources created in
             OnSwapChainResize */
}
```

We can now set the **`OnSwapChainReleasing`** callback function via the **`DXUTSetCallbackD3D10SwapChainReleasing`** DXUT function:

```
DXUTSetCallbackD3D10SwapChainReleasing(OnSwapChainReleasing);
```

We also require a callback function to release the Direct3D 10 resources created in the **`OnCreateDevice`** callback function. This resource deletion callback, **`LPDXUTCALLBACKD3D10DEVICEDESTROYED`**, is executed by the DXUT framework immediately after the Direct3D 10 device has been destroyed. The **`DXUTSetCallbackD3D10DeviceDestroyed`** function, with its first parameter taking a pointer to a **`LPDXUTCALLBACKD3D10DEVICEDESTROYED`** function, sets the device destroyed callback. Its second parameter is a pointer to a user-specific variable passed to the callback function whenever context information is needed:

```
VOID DXUTSetCallbackD3D10DeviceDestroyed(
  LPDXUTCALLBACKD3D10DEVICEDESTROYED pCallback,
  void* pUserContext
);
```

The **`LPDXUTCALLBACKD3D10DEVICEDESTROYED`** callback function specifies only one parameter, namely a pointer to a user-specific variable for the gathering of context information, **`pUserContext`**:

```
VOID LPDXUTCALLBACKD3D10DEVICEDESTROYED(
  void* pUserContext
);
```

A **`LPDXUTCALLBACKD3D10DEVICEDESTROYED`** resource deletion callback function can be defined as follows:

```
void CALLBACK OnDeviceDestroy(void* pUserContext)
```

```
{
         /* release all the Direct3D 10 resources created in the
            OnCreateDevice callback function */
}
```

This callback function is then subsequently set using the **`DXUTSetCallbackD3D10DeviceDestroyed`** DXUT function:

```
DXUTSetCallbackD3D10DeviceDestroyed(OnDeviceDestroy);
```

Another significant DXUT callback function is one that deals with frame rendering. This **`LPDXUTCALLBACKD3D10FRAMERENDER`** callback function renders a scene using the created Direct3D 10 device by clearing the back buffer, depth-stencil buffer, updating all variable changes per frame and rendering the geometric objects constituting the scene. This function has four parameters and is declared as follows in the DXUT.h header file:

```
VOID LPDXUTCALLBACKD3D10FRAMERENDER(
  ID3D10Device * pd3dDevice,
  DOUBLE fTime,
  FLOAT fElapsedTime,
  void* pUserContext
);
```

Its first parameter, **`pd3dDevice`**, is a pointer to an **`ID3D10Device`** interface – the rendering device. The second parameter, **`fTime`**, holds the time that has elapsed since initialisation of the application with the third parameter, **`fElapsedTime`**, holding the time that has passed since the last frame update. Both these time values are given in seconds. The final parameter holds a pointer to the user-specific variable that is passed to the callback function whenever context information is needed. Just as with all the other DXUT callback functions, we will also set this one to 'NULL'.

Such a **`LPDXUTCALLBACKD3D10FRAMERENDER`** callback function can be declared as follows:

```
void CALLBACK OnRenderFrame(          ID3D10Device* pd3dDevice,
                                      double fTime, float fElapsedTime,
                                      void* pUserContext)
{
     /* - clear the back buffer using ClearRenderTargetView
        - clear the depth-stencil buffers using
          ClearDepthStencilView
        - update all changed variables
        - render all geometric objects */
}
```

This **`OnRenderFrame`** callback function is set by the **`DXUTSetCallbackD3D10FrameRender`** function:

```
DXUTSetCallbackD3D10FrameRender(OnRenderFrame);
```

All that remains now is to handle all process messages originating from the DXUT message pump and to set the callback function responsible for doing the frame updates for the scene. We also require a facility that allows us to change the settings of a device before it is created.

Processing messages for the DXUT message pump requires the declaration of a **`LPDXUTCALLBACKMSGPROC`** callback function similar to the previously defined WinProc function. This function takes six parameters, the first being a handle to the window, the second an integer value identifying the message to process, the third and fourth parameters specifying additional message information, with the fifth a Boolean value that controls whether further message processing should be done ('true' preventing further message handling). The final parameter is a pointer to a user-specific variable passed to the callback function whenever context information is needed:

```
LRESULT LPDXUTCALLBACKMSGPROC(
  HWND hWnd,
  UINT uMsg,
  WPARAM wParam,
  LPARAM lParam,
  bool * pbNoFurtherProcessing,
  void* pUserContext
);
```

We can declare a **`LPDXUTCALLBACKMSGPROC`** callback function as follows:

```
LRESULT CALLBACK MsgProcCallback( HWND hWnd, UINT uMsg,
                                  WPARAM wParam,
                                  PARAM lParam,
                                  bool* pbNoFurtherProcessing,
                                  void* pUserContext)
{
          /* handle all messages sent to the application */
}
```

The **`DXUTSetCallbackMsgProc`** DXUT function sets this window message callback function with its first parameter a pointer to the **`LPDXUTCALLBACKMSGPROC`** function and its second a pointer to a user-specific variable passed to the callback function whenever context information is needed:

```
DXUTSetCallbackMsgProc(MsgProcCallback);
```

Frame updates of the scene are done via the **`LPDXUTCALLBACKFRAMEMOVE`** callback function. This function takes three parameters, namely, the time that has elapsed since initialisation of the application, the time elapsed since the previous frame and a pointer to a user-specific variable passed to the callback function whenever context information is needed:

```
VOID LPDXUTCALLBACKFRAMEMOVE(
  DOUBLE fTime,
  FLOAT fElapsedTime,
  void* pUserContext
);
```

Such a **`LPDXUTCALLBACKFRAMEMOVE`** callback function handling updates to a scene can be declared as follows:

```
void CALLBACK OnMoveFrame( double fTime, float fElapsedTime,
                           void* pUserContext)
{
          /* update the scene */
}
```

This callback function is subsequently set using the **`DXUTSetCallbackFrameMove`** DXUT function:

```
DXUTSetCallbackFrameMove(OnMoveFrame, NULL);
```

One final callback function is needed for the modification of Direct3D device settings as required. This callback function, **`LPDXUTCALLBACKMODIFYDEVICESETTINGS`**, takes a pointer to a **`DXUTDeviceSettings`** structure storing the settings of our Direct3D 10 device, and a pointer to a user-specific variable passed to the callback function whenever context information is needed:

```
bool LPDXUTCALLBACKMODIFYDEVICESETTINGS(DXUTDeviceSettings * pDeviceSettings,
                                        void* pUserContext);
```

An example of a **`LPDXUTCALLBACKMODIFYDEVICESETTINGS`** callback function is given here:

```
bool CALLBACK ModDevSettings(DXUTDeviceSettings* pDeviceSettings, void* pUserContext)
{
          /* allow modification of device settings */
          return true;
}
```

This callback function is called just before the creation of the Direct3D device. It returns a ‘true’ indicating that DXUT can proceed to create the device, and a ‘false’ indicating otherwise. The **`DXUTSetCallbackDeviceChanging`** function sets this callback function, allowing the application program to modify the device settings as needed. This function takes two parameters, a pointer to a **`LPDXUTCALLBACKMODIFYDEVICESETTINGS`** callback function and a pointer to a user-specific variable passed to the callback function whenever context information is needed:

```
DXUTSetCallbackDeviceChanging(ModDevSettings, NULL);
```

The functions presented in this section illustrate the fundamentals of the DXUT framework. This framework is useful for experimental applications where the desire is to minimise the amount of time spent on setting up a Direct3D environment. Although the DXUT framework’s effectiveness in the simplification of Direct3D API calls cannot be disputed, it must be used with utmost caution as it does impose some level of performance overhead.