Contents

Introduction

The DBO format has recently been introduced to Dark Basic Professional and provides many benefits.

- easier for internal management of objects within Dark Basic Professional
- load times improved when using DBO objects
- more control over how your object will look when drawn at runtime – can set properties externally in your file

---

Header

At the start of a DBO file there is information relating to the identity of the file -

Type	Name	Size	Information
DWORD	dwLength	4	length of string
Char*	pszString	-	string
DWORD	dwVersion	4	version
DWORD	dwReserved1	4	reserved 1
DWORD	dwReserved2	4	reserved 2

DWORD dwLength
Strings are always preceeded by a DWORD value that lets us know the length of the string. This allows you to allocate the correct amount of memory to store the string that follows.

char* pszString
This must contain the text “MAGICDBO”. It lets us know that we’re dealing with a DBO file.

DWORD dwVersion
This is the version number of the file. At this time the current version will be set to “1”.

DWORD dwReserved1;
DWORD dwReserved2;
These variables are reserved and will have a value of “0”.

Data Blocks

The actual data contained in the file is accessed using identifiers. A code is read in, the size of data and then the data itself. Based on this information you know what type of information you’re accessing.

The following three variables need to be read in -

DWORD dwCode;
DWORD dwCodeSize;
DWORD pBlock;*

DWORD dwCode;
The code lets us know what type of data we’re dealing with. As an example if “dwCode” is set to “1” then we know that the data is a “root frame”.

DWORD dwCodeSize;
Indicates the size of the data block to be read in.

DWORD* pBlock;
Pointer to the actual data. This is optional. When a data block is composed of sub objects then “pBlock” does not exist.

Here’s an example of how to access this data –

```
// assume buffer points to file data
BYTE*    pBuffer = g_pFileData;

// variables to store code ID and size
DWORD dwCode     = 0;
DWORD dwCodeSize = 0;
BYTE* pData      = NULL;

// get code and move buffer pointer forward
dwCode   = *( DWORD* ) pBuffer;
pBuffer += sizeof ( DWORD );

// get code size and move buffer pointer forward
```

```
dwCodeSize = *( DWORD* ) pBuffer;
pBuffer   += sizeof ( DWORD );

// get data
pData = pBuffer;
```

Frame

Code	Name
1	Root frame
104	Child frame
105	Sibling frame

After reading in the header data the next block of data will be that of the root frame. Retrieve the data from the data block – the code, the size and the pointer to the data. If the code has a value of “1” then the data is valid and we then know we’re dealing with the root frame. The root frame is exactly the same as child and sibling frames with the exception that it has an added identifier.

No data pointer is associated with frames. Only sub blocks will be found.

Once you have ascertained that you are dealing with a frame, get the next code to access the sub data. The following sub objects will be found in a frame

ID	Name
101	Frame Name
102	Frame Matrix
103	Frame Mesh
104	Frame Child
105	Frame Sibling
106	Frame Offset
107	Frame Rotation
108	Frame Scale

When reading in sub blocks for the frame a code of “0” will indicate that no more sub blocks exist.

The process of loading in the frame and it’s sub data is outlined -

- read the code
- determine if the code is valid and identifies the root frame
- read the next code in the buffer
- loop while the code is valid (greater than 0)
    - check the data block
    - read in data
    - move to next block

Here is a code example to demonstrate how the process works -

```
bool GetRootFrame ( void )
{
	// variables to store code and size
	DWORD	dwCode     = 0,
		dwCodeSize = 0;

	// get the code
	ReadCODE ( &dwCode, &dwCodeSize );

	// is this a root frame
	if ( dwCode == 1 )
	{
		// read the next code
		ReadCODE ( &dwCode, &dwCodeSize );

		// loop while we have valid sub blocks
		while ( dwCode > 0 )
		{
			// check the code
			switch ( dwCode )
			{
				case 101:
				{
					// get frame name
				}
				break;

				default:
				{
					// skip data
					g_pBuffer += dwCodeSize;
				}
				break;
			}

			// read in next code
			ReadCODE ( &dwCode, &dwCodeSize );
		}
	}

	return true;
}
```

Frame Name

Code	Name
101	Frame name

This is the given name of a frame. The data contained is as follows –

Type	Size	Information
DWORD	4	length of string
char*	-	string

Whenever a string value is stored in the file it is always preceeded by a DWORD that specifies the length of the string.

The code shows a possible method of extracting the string –

```
bool ReadString ( char** pString )
{
	// get length of string
	DWORD dwLength = *( DWORD* ) g_pBuffer;

	// move buffer to next part of data
	g_pBuffer += sizeof ( DWORD );

	// is length valid
	if ( dwLength > 0 )
	{
		// create new string and add extra character
		*pString = new char [ dwLength + 1 ];

		// copy data
		memcpy ( *pString, g_pBuffer, sizeof ( char ) * dwLength );

		// add null at end
		*( *pString + dwLength ) = 0;

		// move to next block of data
		g_pBuffer += sizeof ( char ) * dwLength;
	}

	return true;
}
```

Frame Matrix

Code	Name
102	Frame matrix

A matrix that is used for the transform of the frame at runtime.

Type	Size	Information
float [4] [4]	64	matrix

The matrix is arranged as a 4 * 4 array of floating point values. The code example demonstrates how this data could be loaded –

```
bool GetFrameMatrix ( void )
{
	// declare matrix
	float matrix [ 4 ] [ 4 ];

	// copy data from buffer into matrix
	memcpy ( matrix, g_pBuffer, sizeof ( matrix ) );

	// move buffer past matrix
	g_pBuffer += sizeof ( matrix );

	// done
	return true;
}
```

Frame Mesh

Code	Name
103	Frame mesh

A mesh structure within the frame. Refer to the mesh documentation for more information.

Frame Child

Code	Name
104	Frame child

This is a child frame that is linked to the current frame. It is read in the same way as you would read a frame.

Frame Sibling

Code	Name
105	Frame sibling

This is a sibling frame that is linked alongside the current frame. It is read in the same way as you would read a frame.

Frame Offset

Code	Name
106	Frame offset

Stores an offset position that is applied to the frame when it's being positioned.

Type	Size	Information
float [3]	12	position

Frame Rotation

Code	Name

107	Frame rotation

Stores a set of rotation angles (in radians) that are applied to the frame when it's being positioned.

Type	Size	Information
float [3]	12	rotation

Frame Scale

Code	Name
108	Frame scale

Stores 3 scale values that are applied to the frame position.

Type	Size	Information
float [3]	12	scale

Mesh

Code	Name
103	Mesh

The mesh structure stores a list of sub objects that contain data for the mesh e.g. vertex positions. As with the root frame no data pointer is stored. Access the next code and then continue to load the data. When the code is "0" no more sub objects exist.

The table below shows a list of sub objects contained within the frame mesh.

ID	Name
111	Mesh FVF
112	Mesh FVF Size
113	Mesh Vertex Count
114	Mesh Index Count
115	Mesh Vertex Data
116	Mesh Index Data
117	Mesh Primitive Type
118	Mesh Draw Vertex Count
119	Mesh Draw Primitive Count
120	Mesh Vertex Declaration
121	Mesh Bone Count
122	Mesh Bones Data
125	Mesh Use Material
126	Mesh Material
127	Mesh Texture Count
128	Mesh Textures
129	Mesh Wireframe
130	Mesh Light
131	Mesh Cull
132	Mesh Fog
133	Mesh Ambient
134	Mesh Transparency
135	Mesh Ghost
136	Mesh Ghost Mode
137	Mesh Linked (internal – skip this block)
138	Mesh Sub Frames (internal – skip this block)
154	FX Effect Name
155	Abitary Value
156	Z Bias Flag
157	Z Bias Slope
158	Z Bias Depth
159	Z Read
160	Z Write
166	Alpha Test Value
123	Use Multiple Materials
124	Multiple Material Count
139	Multiple Materials
140	Mesh Visible

The sub blocks are accessed in the same way as sub blocks for the root frame –

```
bool GetFrameMesh ( void )
{
```

```
	// variables to store code and size
	DWORD	dwCode     = 0,
		dwCodeSize = 0;

	// get the code
	ReadCODE ( &dwCode, &dwCodeSize );

	// loop while we have valid sub blocks
	while ( dwCode > 0 )
	{
		// check the code
		switch ( dwCode )
		{
			case 111:
			{
				// get mesh fvf
			}
			break;


			default:
			{
				// skip data
				g_pBuffer += dwCodeSize;
			}
			break;
		}

		// read in next code
		ReadCODE ( &dwCode, &dwCodeSize );
	}

	return true;
}
```

Mesh FVF

Code	Name
111	Mesh FVF

This is the flexible vertex format (FVF) of the mesh and describes the vertex data that it uses.

Type	Size	Information
DWORD	4	FVF value

The data stored in the DWORD is set up using logical bits. The following table shows possible values

Type	Value	Information
FVF_XYZ	0x002	float X, Y, Z positions
FVF_XYZRHW	0x004	float X, Y, Z transformed
FVF_NORMAL	0x010	float X, Y, Z normal vector
FVF_PSIZE	0x020	float point size for sprites
FVF_DIFFUSE	0x040	DWORD diffuse colour
FVF_SPECULAR	0x080	DWORD specular colour
FVF_TEX0	0x000	texture coordinates 0
FVF_TEX1	0x100	texture coordinates 1
FVF_TEX2	0x200	texture coordinates 2

FVF_TEX3	0x300	texture coordinates 3
FVF_TEX4	0x400	texture coordinates 4
FVF_TEX5	0x500	texture coordinates 5
FVF_TEX6	0x600	texture coordinates 6
FVF_TEX7	0x700	texture coordinates 7
FVF_TEX8	0x800	texture coordinates 8

As an example the mesh fvf may be set as

DWORD dwMeshFVF = FVF_XYZ | FVF_NORMAL | FVF_TEX1;

This indicates that the mesh vertex data is comprised of 3 floating point values for the position (x, y, z), 3 floating point values for the normal (nx, ny, nz) and finally 2 floating point values for the texture coordinates (tex1).

To access this data

```
dwMeshFVF    = *( DWORD* ) g_pBuffer;
g_pBuffer   += sizeof ( DWORD );
```

Mesh FVF Size

Code	Name
112	Mesh FVF Size

This value indicates the size of the flexible vertex format. This is based on the values found in the mesh FVF.

Type	Size	Information
DWORD	4	FVF size

Take the following FVF as an example

DWORD dwMeshFVF = FVF_XYZ | FVF_NORMAL | FVF_TEX1;

This tells us that the FVF is composed of

3 floats for the x, y, z position
3 floats for the x, y, z normal
2 floats for the texture coordinates

If a float is 4 bytes then we can determine that size will be (3 + 3 + 2) * 4 = 32.

Mesh Vertex Count

Code	Name
113	Mesh Vertex Count

Contains the number of vertices in the mesh.

Type	Size	Information
DWORD	4	Vertex count

Mesh Index Count

Code	Name
114	Mesh Index Count

Contains the number of indices in a mesh.

Type	Size	Information
DWORD	4	Index count

Mesh Vertex Data

Code	Name
115	Mesh Vertex Data

This block contains the actual mesh vertex data. The size of the data to be read in is based on data obtained from previous blocks – the mesh size and the mesh vertex count.

Type	Size	Information
BYTE*	Mesh FVF Size * Mesh Vertex Count	Vertex data

The code example shows how to retrieve the vertex data

```
// get mesh vertex data
DWORD dwSize = dwFVFSize * dwVertexCount;
BYTE* pData  = new BYTE [ dwSize ];

// check memory was allocated
if ( !pData )
    return;

// copy data from buffer into vertex array
memcpy ( pData, g_pBuffer, dwSize );

// move buffer past vertex data
g_pBuffer += dwSize;
```

Accessing the values within the vertex data is dependant on the FVF. If we have a mesh that uses an FVF of “FVF_XYZ | FVF_NORMAL | FVF_TEX1” then the data can be accessed like this

```
float fX  = *( ( float* ) pData + 0 );
float fY  = *( ( float* ) pData + 1 );
float fZ  = *( ( float* ) pData + 2 );

float fNX = *( ( float* ) pData + 3 );
float fNY = *( ( float* ) pData + 4 );
float fNZ = *( ( float* ) pData + 5 );

float fTU = *( ( float* ) pData + 6 );
float fTV = *( ( float* ) pData + 7 );
```

Mesh Index Data

Code	Name
116	Mesh Index Data

This block contains the mesh index data. The size of the data is based on the index count.

Type	Size	Information
WORD	sizeof (WORD) * Index Count	Index data

The code example shows how to access the indices –

```
// get mesh index data
DWORD dwSize = dwIndexCount * sizeof ( WORD );
WORD* pData  = new WORD [ dwSize ];

// check memory
if ( !pData )
       return false;

// copy data
memcpy ( pData, g_pBuffer, dwSize );

// move past index data
g_pBuffer += dwSize;
```

Mesh Primitive Type

Code	Name
117	Mesh Primitive Type

Contains a value indicating the type of primitive that the mesh is constructed from.

Type	Size	Information
DWORD	4	Primitive Type

Values for this include

Type	Value	Information
POINTLIST	1	Point list – particles
LINELIST	2	Line list
LINESTRIP	3	Line strip
TRIANGLELIST	4	Triangle list
TRIANGLESTRIP	5	Triangle strip
TRIANGLEFAN	6	Triangle fan

Mesh Draw Vertex Count – 118

Code	Name
118	Mesh Draw Vertex Count

Stores the number of vertices that will be drawn at runtime.

Type	Size	Information
DWORD	4	Draw vertex count

Mesh Draw Primitive Count

Code	Name
119	Mesh Draw Primitive Count

Contains the number of primitives that will be drawn at runtime.

Type	Size	Information
DWORD	4	Draw primitive count

Mesh Vertex Declaration

Code	Name
120	Mesh Vertex Declaration

Custom vertex declaration.

Mesh Bone Count

Code	Name
121	Mesh Bone Count

The number of bones used for the mesh.

Type	Size	Information
DWORD	4	Bone count

Mesh Bones Data

Code	Name
122	Mesh Bones Data

Refers to a bone structure. See the relevant section on bones for more information.

Mesh Use Material

Code	Name
125	Mesh Use Material

A flag to indicate whether the mesh uses a material.

Type	Size	Information
bool	1	use material flag

Mesh Material

Code	Name
126	Mesh Material

Stores material data for the mesh.

Type	Size	Information
float [4]	16	diffuse
float [4]	16	ambient
float [4]	16	specular
float [4]	16	emissive

float	4	power

Example code for loading this data

```
// structure to store material data
struct sMaterial
{
    float diffuse  [ 4 ];
    float ambient  [ 4 ];
    float specular [ 4 ];
    float emissive [ 4 ];
    float power;
};

// declare material
sMaterial material;

// copy data from buffer to material
memcpy ( &material, g_pBuffer, sizeof ( sMaterial ) );

// move buffer forward
g_pBuffer += sizeof ( sMaterial );
```

Mesh Texture Count

Code	Name
127	Mesh Texture Count

Contains the number of textures used by the mesh.

Type	Size	Information
DWORD	4	number of textures

Mesh Textures

Code	Name
128	Mesh Textures

Texture data for a mesh. Refer to the section on textures for further information.

Mesh Wireframe

Code	Name
129	Mesh Wireframe

A boolean value that is used to toggle wireframe rendering. A value of true will have the mesh drawn in wireframe mode whereas a value of false will turn wireframe rendering off for the mesh.

Type	Size	Information
bool	1	wireframe on / off

Mesh Light

Code	Name
130	Mesh Light

A boolean value that is used to enable or disable lights for meshes. When this value is set to true the mesh will respond to lights. When the value is set to fales then the mesh will not react to lights in the scene.

Type	Size	Information
bool	1	lights on / off

Mesh Cull

Code	Name
131	Mesh Cull

By setting this value you can control whether culling is used for the mesh. When the value is true culling will be enabled, when the value is false culling will be disabled.

Type	Size	Information
bool	1	culling on / off

Mesh Fog

Code	Name
132	Mesh Fog

When set to true the mesh will respond to fog. When false the mesh ignores fog in the scene.

Type	Size	Information

bool	1	fog on / off

Mesh Ambient

Code	Name
133	Mesh Ambient

When set to true the mesh will respond to ambient light. When false the mesh ignores ambient light settings.

Type	Size	Information
bool	1	ambient on / off

Mesh Transparency

Code	Name
134	Mesh Transparency

When the value is true the mesh textures will have transparency turned on. When the value is false transparency is switched off.

Type	Size	Information
bool	1	transparency on / off

Mesh Ghost

Code	Name
135	Mesh Ghost

When true ghosting is switched on for the mesh. The ghost mode is dependant on “mesh ghost mode”. When false, ghosting is switched off.

Type	Size	Information
bool	1	ghost on / off

Mesh Ghost Mode

Code	Name
136	Mesh Ghost Mode

Sets the ghost mode to be used for the mesh. Ghosting must be enabled for the mesh for this setting to take effect.

Type	Size	Information
int	4	ghost mode

Possible values for ghosting modes

Type	Value	Information
blend and inverse	0	
dest and source	1	
source and one	2	
src alpha x 2	3	
source and dest	4	

Mesh Use Multiple Materials

Code	Name
123	Mesh Use Multiple Materials

Set to true when the mesh uses multiple materials.

Type	Size	Information
bool	1	multiple materials

Mesh Multiple Material Count

Code	Name
124	Mesh Multiple Material Count

The number of materials for the mesh.

Type	Size	Information
DWORD	1	multiple materials count

Mesh Multiple Materials

Code	Name
139	Mesh Multiple Materials

See the section on multiple materials for more information.

Mesh Visible

Code	Name
140	Mesh Visible

When this value is true the mesh will be drawn at runtime as long as it's within the viewing frustum of the camera. When false the mesh will not be drawn.

Type	Size	Information
bool	1	visible on / off

Bones – 122

Code	Name
122	Bones

Contains information for the bones in a mesh. The number of bones is stored in the mesh structure. The bone count will always be set before coming to the bone data block.

This data block contains the following sub objects

ID	Name
301	Bones Name
302	Bones Num Influences
303	Bones Vertices
304	Bones Weights
305	Bones Translation Matrix

When accessing codes a value of "0" will indicate that no more sub objects exist or if multiple bones exist then you need to move onto the next bone.

This code example demonstrates how the bones data can be extracted from the data buffer

```
bool GetBonesData ( DWORD dwBoneCount )
{
	// get bone data

	// variables to store code and size
	DWORD	dwCode,
		dwCodeSize = 0;

	// get the code
	ReadCODE ( &dwCode, &dwCodeSize );

	// loop while we have valid sub blocks
	for ( int i = 0; i < ( int ) dwBoneCount; i++ )
	{
		// used to store name of bone
		char* pName = NULL;

		while ( dwCode > 0 )
		{
			// check the code
			switch ( dwCode )
			{
				case 301:
				{
					// get bone name
					ReadString ( ( char** ) &pName );
				}
				break;


				// default is to skip data
				default:
				{
					g_pBuffer += dwCodeSize;
				}
				break;
			}

			// read in next code
			ReadCODE ( &dwCode, &dwCodeSize );
		}

		// move to next bone
		ReadCODE ( &dwCode, &dwCodeSize );
	}

	return true;
}
```

Bones Names

Code	Name
301	Bone Names

Contains the name of a bone.

Type	Size	Information
DWORD	4	length of string
char*	-	string

Bone Num Influences

Code	Name
302	Bone Num Influences

Number of bone influences in the mesh.

Type	Size	Information
DWORD	4	number of influences

Bone Vertices

Code	Name
303	Bone Vertices

A list of indices that relate to vertices for the bone animation.

Type	Size	Information
DWORD	sizeof (DWORD) * number of influences	vertex list

Bone Weights

Code	Name
304	Bone Weights

A list of weights for bones.

Type	Size	Information
float	sizeof (float) * number of influences	bone weights list

---

Bones Translation Matrix

Code	Name
305	Bones Translation Matrix

Translation matrix for the bone. This is applied to the vertices for the animation at runtime.

Type	Size	Information

float [4] [4]	64	translation matrix

Textures

Code	Name
128	Textures

Texture data for a mesh. No data pointer is included. This block contains sub objects. When accessing codes a value of “0” will indicate that no more sub objects exist or if multiple textures exist then you move onto the next texture.

The following table shows a list of sub objects that can be found within the texture block

ID	Name
141	Texture Name
142	Texture Stage
143	Texture Blend Mode
144	Texture Arg 1
145	Texture Arg 2
146	Texture Address U
147	Texture Address V
148	Texture Mag
149	Texture Min
150	Texture Mip
151	Texture TC Mode
152	Texture Primitive Start
153	Texture Primitive Count

A code example to show how the data can be loaded

```
bool GetTextureData ( DWORD dwTextureCount )
{
	// get texture data

	// variables to store code and size
	DWORD   dwCode,
		dwCodeSize = 0;

	// get the code
	ReadCODE ( &dwCode, &dwCodeSize );

	// loop while we have valid sub blocks
	for ( int i = 0; i < ( int ) dwTextureCount; i++ )
	{
		char*   pName = NULL;
		DWORD   dwStage,
				dwArgument1,
				dwArgument2,
				dwAddressU,
				dwAddressV,
				dwMag,
				dwMin,
				dwMip,
```

```
                        dwTCMode,
                        dwPrimitiveStart,
                        dwPrimitiveCount,
                        dwBlendMode = 0;

                while ( dwCode > 0 )
                {
                        // check the code
                        switch ( dwCode )
                        {
                                case 141: ReadString ( ( char** ) &pName );  break;
                                case 142: ReadDWORD  ( &dwStage );           break;
                                case 143: ReadDWORD  ( &dwBlendMode );       break;
                                case 144: ReadDWORD  ( &dwArgument1 );       break;
                                case 145: ReadDWORD  ( &dwArgument2 );       break;
                                case 146: ReadDWORD  ( &dwAddressU );        break;
                                case 147: ReadDWORD  ( &dwAddressV );        break;
                                case 148: ReadDWORD  ( &dwMag );             break;
                                case 149: ReadDWORD  ( &dwMin );             break;
                                case 150: ReadDWORD  ( &dwMip );             break;
                                case 151: ReadDWORD  ( &dwTCMode );          break;
                                case 152: ReadDWORD  ( &dwPrimitiveStart );  break;
                                case 153: ReadDWORD  ( &dwPrimitiveCount );  break;

                                // default is to skip data
                                default: g_pBuffer += dwCodeSize; break;
                        }

                        // read in next code
                        ReadCODE ( &dwCode, &dwCodeSize );
                }

                ReadCODE ( &dwCode, &dwCodeSize );

        }

        return true;
}
```

Available values for the blend mode

Type	Value	Information
Disable	1	
Select arg 1	2	
Select arg 2	3	
Modulate	4	
Modulate 2X	5	
Modulate 4X	6	
Add	7	
Add signed	8	
Add signed 2X	9	
Subtract	10	
Add smooth	11	
Blend diffuse alpha	12	
Blend texture alpha	13	
Blend factor alpha	14	
Blend texture alpha pm	15	
Blend current alpha	16	
Pre modulate	17	
Modulate alpha add colour	18	
Modulate colour add alpha	19	
Modulate inv alpha add colour	20	

Modulate inv alpha add alpha	21	
Bump env map	22	
Bump env luminance map	23	
Dot product 3	24	
Multiply add	25	
Lerp	26	

Values for argument 1 and argument 2

Type	Value	Information
Current	0x00000001	
Diffuse	0x00000000	
Select mask	0x0000000f	
Specular	0x00000004	
Temp	0x00000005	
Texture	0x00000002	
Factor	0x00000003	

Values for address u and address v

Type	Value	Information
Wrap	1	
Mirror	2	
Clamp	3	
Border	4	
Mirror once	5	

Values for mag, min and mip filters

Type	Value	Information
None	0	disable mip mapping
Point	1	use point filtering
Linear	2	use linear filtering
Ansiotropic	3	use ansiotropic filtering
Pyramid al quad	6	use pyramid filtering
Gaussian quad	7	use gaussian filtering

Values for texture tc mode

Type	Value	Information
Pass through	0x00000000L	
Camera space normal	0x00010000L	
Camera space position	0x00020000L	
Camera space reflection vector	0x00030000L	
Sphere map	0x00040000L	

Texture Name

Code	Name
141	Texture Name

Contains the name of the texture.

Type	Size	Information
DWORD	4	length of string
char	-	name of texture

Texture Stage

Code	Name
142	Texture Stage

The stage in the pipeline that the texture is used for.

Type	Size	Information
DWORD	4	stage

Texture Blend Mode

Code	Name
143	Texture Blend Mode

The blend mode of the texture.

Type	Size	Information
DWORD	4	blend mode

Texture Argument 1

Code	Name
144	Texture Argument 1

Blend argument 1 for the texture.

Type	Size	Information
DWORD	4	argument 1

Texture Argument 2

Code	Name
145	Texture Argument 2

Blend argument 2

Type	Size	Information
DWORD	4	argument 2

Texture Address U

Code	Name
146	Texture Address U

Texture address U

Type	Size	Information
DWORD	4	Address u

Texture Address V – 147
Texture address V

Type	Size	Information
DWORD	4	Address v

Texture Mag – 148
Magnification filter for texture.

Type	Size	Information
DWORD	4	blend

Texture Min – 149
Minification filter for texture.

Type	Size	Information

DWORD	4	blend

Texture Mip – 150
Mipmap filter to use for minification.

Type	Size	Information
DWORD	4	blend

Texture TC Mode – 151
Blend argument 1

Type	Size	Information
DWORD	4	blend

Texture Primitive Start – 152
Blend argument 1

Type	Size	Information
DWORD	4	blend

Texture Primitive Count – 153
The texture primitive count

Type	Size	Information
DWORD	4	blend

Multiple materials – 128

Stores material data for when a mesh has multiple materials attached. No data pointer is included. This block contains sub objects. When accessing codes a value of “0” will indicate that no more sub objects exist or if multiple textures exist then you move onto the next texture.

The following table shows a list of sub objects that can be found within the mutliple material block –

ID	Name
161	Name
162	Material
163	Start
164	Count
165	Polygon

The following code demonstrates how this data could be accessed –

```
bool GetMultipleMaterialData ( DWORD dwMultiCount )
{
	// get multiple material data
```

```
	// variables to store code and size
	DWORD	dwCode,
		dwCodeSize = 0;

	// get the code
	ReadCODE ( &dwCode, &dwCodeSize );

	// loop while we have valid sub blocks
	for ( int i = 0; i < ( int ) dwMultiCount; i++ )
	{
		char*		pName = NULL;
		sMaterial	material;
		DWORD		dwIndexStart,
				dwIndexCount,
				dwPolyCount = 0;

		while ( dwCode > 0 )
		{
			// check the code
			switch ( dwCode )
			{
				case 161: ReadString   ( ( char** ) &pName );  break;
				case 162: ReadMaterial ( &material );          break;
				case 163: ReadDWORD    ( &dwIndexStart );      break;
				case 164: ReadDWORD    ( &dwIndexCount );      break;
				case 165: ReadDWORD    ( &dwPolyCount );       break;

				// default is to skip data
				default: g_pBuffer += dwCodeSize; break;
			}

			// read in next code
			ReadCODE ( &dwCode, &dwCodeSize );
		}

		ReadCODE ( &dwCode, &dwCodeSize );

	}

	return true;
}
```

Multiple Material Name – 161
The name of the material.

Type	Size	Information
DWORD	4	length of string
char	-	name of texture

Material – 162
Contains a reference to a mesh material.

Multiple Material Start – 163
Start index for where the material is applied.

Type	Size	Information
DWORD	4	start index

Multiple Material Count – 164
Number of indices to affect.

Type	Size	Information
DWORD	4	count

Multiple Material Polygon – 164
Number of polygons to affect.

Type	Size	Information
DWORD	4	count

Animation Set - 2

After reading in frame and mesh data for an object the next section that will come up is for animation. Read in the next code and if it's value is "2" then the animation set is valid.

No data pointer is associated with the animation set. Only sub blocks will be found.

Retrieve the next code to access the sub data. The following sub objects will be found in an animation set -

ID	Name
201	Animation Name
202	Animation Length
203	Animation Data

When reading in sub blocks for the frame a code of "0" will indicate that no more sub blocks exist.

Animation Name – 201
Name of the animation.

Type	Size	Information
DWORD	4	length of string
char	-	name of texture

Animation Length – 202
The amount of frames in the animation.

Type	Size	Information
DWORD	4	frame count

Animation Data – 203
Refer to the section on animation data for more information.

Animation Data – 203

This data block contains information relating to a particular animation. No data pointer is associated with this block – only sub blocks.

The following sub blocks can be found –

ID	Name
211	Animation Name
212	Num position keys
213	Position data
214	Num rotation keys
215	Rotation data
216	Num scale keys
217	Scale data
218	Num matrix keys
219	Matrix data
220	Next animation

Animation Name – 211
The name of the animation.

Type	Size	Information
DWORD	4	length of string
char	-	name of animation

Num position keys - 212
The number of position keys within the animation.

Type	Size	Information
DWORD	4	count

Position data - 213
Contains position data which affects the animation.

Type	Size	Information
DWORD	4	time
float [3]	12	position
float [3]	12	interpolation

Allocate an array based on the number of position keys found and then copy the data in.

Num rotation keys - 214
The number of rotation keys within the animation.

Type	Size	Information
DWORD	4	count

Rotation data - 215
Contains rotation data which affects the animation.

Type	Size	Information
DWORD	4	time
float [4]	16	rotation

Allocate an array based on the number of rotation keys found and then copy the data in.

Num scale keys - 216
The number of scale keys within the animation.

Type	Size	Information
DWORD	4	count

Scale data - 217
Contains scale data which affects the animation.

Type	Size	Information
DWORD	4	time
float [3]	12	scale
float [3]	12	interpolation

Allocate an array based on the number of scale keys found and then copy the data in.

Num matrix keys - 218
The number of matrix keys within the animation.

Type	Size	Information
DWORD	4	count

Matrix data - 219
Contains matrix data which affects the animation.

Type	Size	Information
DWORD	4	time
float [4] [4]	64	matrix
float [4] [4]	64	interpolation

Allocate an array based on the number of matrix keys found and then copy the data in.

Next animation - 220
Links back into animation data – 203.

Custom Data
The custom data component of the object is at the top of the object hierachy. It can be used in cases when an object needs some specific data to be stored that can not be stored in other parts of the DBO structure.

An example of it's use is with the advanced terrain plugin. When a terrain is loaded height map data is saved in an array for collision checks. When the terrain is saved there is no specific area in the DBO structure to save this information so we turn to the custom data structure. The advanced terrain plugin creates a block of data to be stored in this structure (the height map collision field). Upon loading terrains it can check this custom data and extract the height map data it requires and so allowing all of the data we need to be stored within the DBO file.

Code	Name
406	Custom Data

Type	Size	Information
DWORD	4	Size of the data block
void*	-	Pointer to the data

Example Project

The example project provided with the release of the DBO format demonstrates how to load a DBO file. The project is created around a Win32 console application as it's only purpose is to demonstrate the loading and saving of objects and not rendering.

Several files are included

- main.cpp
- DBOData.h
- DBOData.cpp
- DBOFile.h
- DBOFile.cpp
- DBOBlock.h
- DBOBlock.cpp

main.cpp
Contains the "main" function. This is the main entry point for the progam. The code for this file is shown below

```
#include "DBOFile.h"

int main ( int argc, char* argv [ ] )
{
	sObject* pObject = NULL;
```

```
        if ( !LoadDBO ( "examples/cube.dbo", &pObject ) )
               return 0;

        delete pObject;
        pObject = NULL;

        return 0;
}
```

An include is made for “DBOFile.h”. Then we declare an “sObject” pointer and set it to NULL. A call is then made to “LoadDBO”. This loads the model passed in by the first parameter and the sObject pointer is filled with the information. The final part of the source code frees up the memory that was previously allocated for the object when it was loaded.

DBOData.h
This file contains a list of all the structures used within the DBO format. Here’s a brief overview of the contents

- at the top of the header you can see include directives for Windows and DirectX,
- several #defines are then set up which are useful for common tasks
- moving on we have forward declarations for several structures
- next we have a typedef which is later used for a function pointer
- the special effects class is then defined along with an effect structure
- going past this we come to the structures used by a DBO object
- you can see all of the properties of each structure and how each structure is used within an object

DBOData.cpp
Within this file there are a collection of constructors for several DBO structures. The source code sets properties of the structures to default values.

DBOFile.h
Header file for “DBOFile.cpp”. Contains function declarations for “LoadDBO” and “SaveDBO”.

DBOFile.cpp
Contains the functions that are the starting point for loading and saving DBO files. These function are listed as

- int LoadDBO (LPSTR pFilename, sObject** ppObject)
- int SaveDBO (LPSTR pFilename, sObject* pObject)

DBOBlock.h
The header file for “DBOBlock.cpp”. Contains a list of defines. The defines contain the codes that are used when referencing object data blocks for example DBOBLOCK_FRAME_NAME is defined as 101 which tells us that when a block of data is loaded with the code 101 then it contains the name of a frame.

Several functions are also declared in this header file. These functions are called by “LoadDBO” and “SaveDBO” in “DBOFile.cpp”.

DBOBlock.cpp
All of the functions for parsing DBO files and saving them is contained within this file.

The Load Process
When the “LoadDBO” function is called the following takes place

- a call is made to “DBOLoadBlockFile”
  - the DBO file is opened
  - the size of the file is found
  - an array is created that is the same size as the file
  - the data from the file is read into the newly allocated array
  - the file is then closed
- a call to “ConstructObject” is made and the array holding the contents of the file is passed in
-

Example Models

A collection of models are included with the DBO source code. Loading these models into the example project and examining the layout and structure of the data and following through the code will help to gain a better understanding of the format.

Cube
The cube model is very simple and contains minimal data. After loading the model you will find that only a few structures have been filled. The data contained within such structures as collision and position for example is filled internally by the engine when the object is added into the scene. For the most part when it comes to loading data these structures can simply be ignored. They only come into use for those writing plugins who need access to object information at runtime.

Try loading this model in the example project and stepping through the code. After the call has been made to the “LoadDBOEx” function examine the “pObject” pointer in the debugger.

From the fields shown in the debugger we can ascertain the following information

- the mesh count and frame count are set to 0
- the mesh list and frame list pointers are null
- there is a frame

- there is no animation set for the object
- collision data is all set to default
- position data is all set to default
- the object has a collection of properties that are set e.g. bVisible is true indicating that the cube should be shown at runtime
- the animation properties are all at default 0 telling us that no animation data is associated with the object
- there is no instance data for the object
- the delete structure is null
- no custom data is supplied with the model

The main area of interest for the cube model is “pObject->pFrame”. By looking at the frame we can gather the following

- the frame has a name – box
- it has no parent frame, nor child or sibling frames
- the frame transforms are all identity matrices
- the position data is all set to default 0, 0, 0
- there is no shadow mesh
- there is no bound box mesh
- there is no bound sphere mesh
- the frame contains a pointer to a mesh

The mesh pointer contained within the frame stores the main bulk of the data for the cube model. Take a closer look at the mesh by looking at “pObject->pFrame->pMesh” in the debugger.

From the information in the debugger we can find out about the mesh including details such as

- the flexible vertex format is 274 which indicates it has position, normal and texture coordinates
- the size of the vertex structure is 32 bytes
- we can see pointers to the vertex and index data
- there are 24 vertices in the vertex list
- there are 36 indices in the index list
- the primitive type is 4 indicating a triangle list
- 24 vertices are sent to the graphics card for drawing
- we end up with 12 triangles being drawn
- there is no shader data associated with the mesh
- there are no bones associated with the mesh
- we can see that the mesh has 1 texture
- no image is referenced by the texture data
- only a material is referenced by the texture data
- the mesh has several internal properties that are used by the engine
- the mesh has a collection of external properties that control things such as the wireframe state of the mesh
- there is no animation data for the mesh

- the collision fields are all set to their default, this data is left alone until the object is added into the engine

Cylinder & Sphere
Again these are simple models and contain pretty much the same data as the cube with each model containing one frame and one mesh. Try loading these models in the debugger and looking through the hierachy to see the overall layout.

Terrain
The terrain mesh contains much more data than the previous models we have encountered - instead of containing just the one frame and mesh there are several to contend with.
Change the example code so it loads “terrain.dbo” and step through with the debugger. After the loading stage take a closer look at the “pObject” pointer.

As with the previous models we start off by looking at the root frame which is accessed by “pObject->pFrame”. Most of what you see here will be familiar, the frame has a name and it contains a link to a mesh which is filled with such things as vertex and index data.

With the terrain model we get to see the introduction of child meshes. In the sFrameLinks part of the object pointer we can see a link to a child frame e.g. pObject->pFrame->pChild. The “pChild” pointer is a pointer to an sFrame structure which is exactly the same type as the root frame we have been viewing.

If we examine “pObject->pFrame->pChild” we can see the sFrame representation and how this frame contains a link to a mesh in the form of “pMesh”. This frame also contains a child link pointer.

In total there are 64 frames contained within this frame and they are all accessed from the root frame and then getting the “pChild” pointer and then the “pChild” pointer of that pointer and so on.