PVRExp

User Manual

Filename : PVRExp.User Manual.mht

Version : 1.2f (PowerVR SDK 2.04.24.0759)

Issue Date : 10 Feb 2009

Author : PowerVR

Contents

1. OVERVIEW

The intention of this plug-in for 3DStudio MAX is to be able to use the 3D data generated from this package inside your C language application in the easiest and fastest way possible. Its intention is simplicity so not all possible features have been implemented.

The files generated by the PVRExporter are 'include' files that you can compile with your own application.

For a description of the structures and flags exported by the plug-in, go to the end of this help.

Note: Textures are exported in a different include file with the extension *.HTX or as stand alone BMP textures.

***Note**: as of May 2005 PVRExp has been superseded by PVRMAXExport. PVRMAXExport is a plug-in for 3D Studio MAX 6 and 7 and is available as part of the PowerVR SDK. PVRMAXExport supports the same features as PVRExp but also provide new and powerful functionality used for rendering techniques and performance. It is strongly recommended that developers use PVRMAXExport instead of the legacy PVRExp as support for the latter will gradually be removed.*

2. INSTALLATION

Copy the required plug-in file in the directory called **\plugins** inside 3D Studio MAX installation directory. It will be loaded automatically during start-up.

From File\Export select PowerVR.
Write the name of the file to be saved and press OK. A dialog window is showed where you can select the different options.

3. DIALOG BOX DESCRIPTION

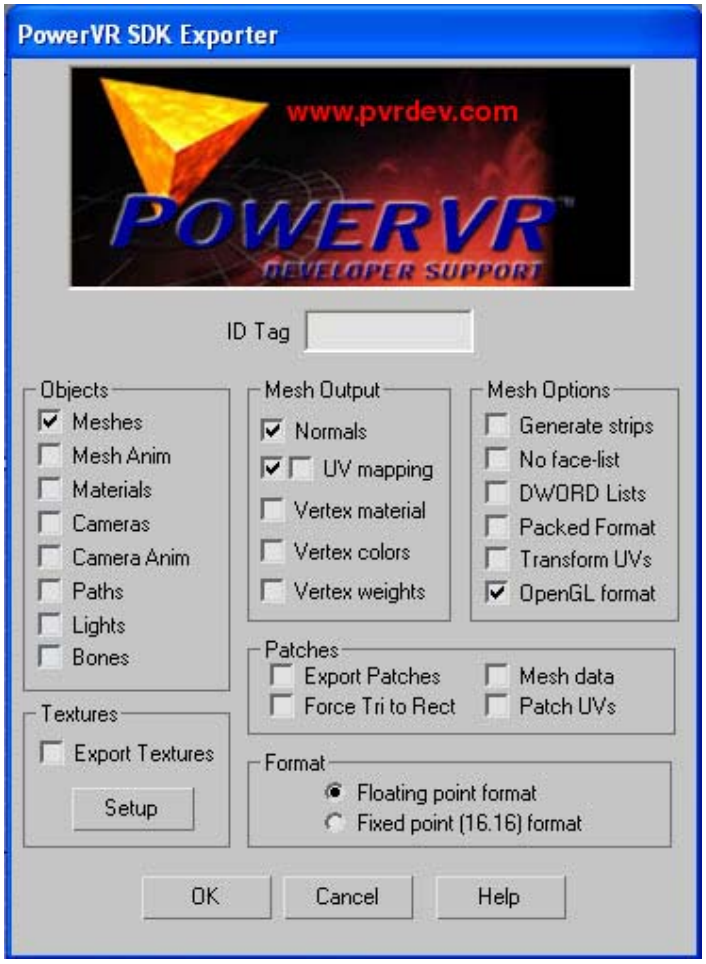

Figure 1: PVRExp Dialog box

3.1. ID TAG

The string edited in this box will be added to the end of all structure and variable names allowing the load of several include files without compiling errors.

3.2. OBJECTS

3.2.1. Meshes:

Export the 3D data that defines every mesh in the scene.
This data includes the position (3 floats), normal (3 floats), UV(2 floats), color (1 DWORD) and material (1 DWORD) for each vertex in the object as well as the polygon list, the strip list and the strip length list.
Other generic data exported per mesh are the center of the mesh (position of the link), the generic material and the flags that indicate which options have been enabled.

Notice that the vertex computation is an exponential operation. That means that a scene with a single object of 50000 polygons can take several minutes for exporting, but a scene with 10 objects of 5000 polygons each just takes a few seconds.

3.2.2. Mesh Animation:

Export the transformation matrix, frame by frame, of all the objects in the scene.
Every matrix is defined by a chunk of 12 floats:
M11, M12, M13, M21, M22, M23, M31, M32 and M33.
To build a 4 by 4 transformation matrix the user has to set the additional values:
M14 = 0 M24=0 M34 = 0 M44 = 1
To animate an object, set these values in your transformation matrix before your own transformations.
Note: Only the transformation matrix is exported, mesh morphing is not supported.

3.2.3. Materials:

Export the definition of all materials used in the scene.
These definitions include all the texture file names, the ambient, diffuse and specular colors, the shininess and shininess strength and the type of shading used.

3.2.4. Cameras:

Export Camera data.
This includes the camera position and camera target, the FOV (field of view), the near clipping plane position and the far clipping plane position.

3.2.5. Camera Animation:

Export camera animation.
The camera animation is exported as two arrays of 3D positions (3 floats per chunk), frame by frame. The first array is for the camera position and the second one is for the target position.

3.2.6. Paths:

Export paths as an array of Cartesian coordinates (3 floats per vertex).

3.2.7. Lights:

Export lights data.
It includes the type of light (target, directional, free or omni), position, target, color, intensity, hotspot and falloff.

3.2.8. Bones:

Export bones animation matrices with the same format than the Mesh Animation matrices explained above.

3.3. MESH OUTPUT

If any of these output options are disabled, the pointer to that data will be NULL and it will not be exported.

3.3.1. Normals:

Export Normals per vertex. (3 floats per vertex).

3.3.2. UV mapping:

Export texture-mapping information per vertex. (2 floats per vertex).
It can export as well a second set of UVs as 4 floats per vertex.

3.3.3. Vertex material:

Export material ID per vertex. (1 unsigned int per vertex_.

3.3.4. Vertex colors:

Export color per vertex. (1 unsigned int per vertex, 8A8R8G8B format)

3.3.5. Vertex weights:

Export vertex weights (for skinning) as vertex colours. (M1W1M2W2 format)

3.4. MESH OPTIONS

3.4.1. Generate Strips:

Export a list with the strip indexes and another list with the strip lengths.
The number of vertices (indices) in a strip is the number of polygons plus 2 so a strip length list with values of 3,2,8,32,… means that the (3+2) first indexes in the strip list define a single strip of 3 polygons, the next (2+2) indexes in the strip list defines a strip of 2 polygons and so on. If this box is off, the pointer to strip lists will be NULL and these lists will be not exported.

Note: The triangle list is always exported.

3.4.2. No face-list:

Export vertex data without the list of indices.
If 'Generate Strips' box is off, the vertices data will be reordered so every three contiguous vertices will define a single polygon. The final number of vertices will be 3 times the number of polygons.
If 'Generate Strips' box is on, the vertices that define every strip will be dumped in order. The strip length list will be exported as well.

3.4.3. DWORD List:

All the lists are defined as DWORD (unsigned int).
Usually generic 3D APIs use by default WORD (unsigned short) for defining vertex lists, but that is not always the case. Don't forget to check your case, but in general this box must be OFF.

3.4.4. Packed Format:

Export 3D data in D3DVERTEX format.
There is a special array in the mesh structure to store this data. Vertex, normal and UV arrays will be NULL.
D3DVERTEX format stores vertex data in chunks of 8 floats: position (3 floats), normals (3 floats) and UVs (2 floats).

3.4.5. OpenGL Format

The 3D data is transformed (Z axis and V texture coordinate are inverted) to match OpenGL standard. This will allow displaying an object as it looks in 3DSMAX just rendering the data as it comes from the exporter.

3.4.6. Transform UVs:

The mapping data is modified with the texture transformation stored in the material.

3.5. TEXTURES SETUP

3.5.1. Shrink to square:

The texture is reduced to a square with size equal to the closest power of 2 value to the smaller length.

3.5.2. Enlarge to square:

The texture is enlarged to a square with size equal to the closest power of 2 value to the bigger length.

3.5.3. Maximum size:

If the final size is greater than this one, the output texture is reduced to this value.

*.HTX (include file)
Include file with 24 bits color data for all the textures. The data follows the BMP format layout, meaning 3 bytes per color (Blue, Green and Red) and with the texture upside-down.
Every texture exports the original file name as well. This name can be used to show which material requested that texture.

*.BMP (binary file)
Standard binary files (one per each texture).

NOTE: Do not use texture export if rectangular textures are required.

3.6. Patches

Patches are exported as sets of 9 or 16 control points.

3.6.1. Export patches

Select to export patches.

3.6.2. Force Tri to Rect

Any triangular patch is converted to a degenerated rectangular one so all patches will be in the same form.

3.6.3. Mesh Data

All patches are converted to standard 3D mesh data.

3.6.4. Patch Uvs

Uvs are calculated for the control points based on actual mapping.

3.7. Vertex Format

3.7.1. Floating Point

Default mode. All vertex data will be exported in floating point format.

3.7.2. Fixed Point

Vertex data will be converted to a 16.16 fixed-point format. There will be a new type defined as int called `fixed_16` that will substitute all float types in the data structures.

4. STRUCTURE DEFINITIONS

4.1.1. Mesh structure

```
typedef struct { unsigned int      nNumVertex;
                 unsigned int      nNumFaces;
                 unsigned int      nNumStrips;
                 unsigned int      nFlags;
                 unsigned int      nMaterial;
                 float             fCenter[3];
                 float             *pVertex;
                 float             *pUV;
                 float             *pNormals;
                 float             *pPackedVertex;
                 unsigned int      *pVertexColor;
                 unsigned int      *pVertexMaterial;
                 unsigned short    *pFaces;
                 unsigned short    *pStrips;
                 unsigned short    *pStripLength;
                 struct
                 {
                     unsigned int  nType;
                     unsigned int  nNumPatches;
                     unsigned int  nNumVertices;
                     unsigned int  nNumSubdivisions;
                     float         *pControlPoints;
                     float         *pUVs;
                 } Patch;
               }   Struct_Mesh;
```

4.1.2. Material structure

```
typedef struct { char     *sMatName;
                 char     *sAmbientFile;
                 char     *sDifusseFile;
                 char     *sSpecularFile;
                 char     *sShininessFile;
                 char     *sShinStrenghtFile;
                 char     *sSelfIlluminationFile;
                 char     *sOpacityFile;
                 char     *sFilterColorFile;
                 char     *sBumpFile;
                 char     *sReflectionFile;
                 char     *sRefractionFile;
                 unsigned int nMatAmbientColor;
                 unsigned int nMatDiffuseColor;
                 unsigned int nMatSpecularColor;
                 float    fMatShininess;
                 float    fMatShineStrength;
                 float    fMatOpacity;
                 unsigned nMatShadingType;
               }   Struct_Material;
```

4.1.3. Light structure

```
typedef struct { int      nLightType;
                 float    fPosition[3];
                 float    fTarget[3];
                 unsigned nColor;
                 float    fIntensity;
                 float    fHotspot;
                 float    fFalloff;
               }   Struct_Light;
```

4.1.4. Camera structure

```
typedef struct { float    fPosition[3];
                 float    fTarget[3];
                 float    fFOV;
                 float    fNearClip;
                 float    fFarClip;
               }   Struct_Camera;
```

4.1.5. Path structure

```
typedef struct { unsigned int  nNumVertex;
                 float         *pVertex;
               }   Struct_Path;
```

4.1.6. Animation structure

```
typedef struct { unsigned int  nNumFrames;
                 float         Pivot[3];
                 float         *pData;
               }   Struct_Animation;
```

4.1.7. Bones structure

```
typedef struct {    float         *pData;
               }   Struct_Bones;
```

4.1.8. Camera Animation structure

```
typedef struct { unsigned int  nNumFrames;
                 float         *Position;
                 float         *Target;
                 float         *FOV;
               }   Struct_CameraAnimation;
```

4.1.9. Texture structure

```
typedef struct { char*            sTextureFilename;
                 int              nWidth;
                 int              nHeight;
                 unsigned char*   pBitmapdata;
               }   Struct_Texture;
```

5. FLAGS AND DEFINES

5.1.1. Number of...

```
#define NUM_MESHES (any)
#define NUM_MATERIALS (any)
#define NUM_LIGHTS  (any)
#define NUM_CAMERAS (any)
#define NUM_PATHS (any)
#define NUM_BONES (any)
```

5.1.2. Mesh flags

```
#define MF_MATERIAL 1
#define MF_UV 2
#define MF_NORMALS 4
#define MF_VERTEXCOLOR 8
#define MF_VERTEXMATERIAL 16
#define MF_STRIPS 32
#define MF_NOFACELIST 64
```

5.1.3. Light types

```
#define OMNI_LIGHT 0
#define TARGET_LIGHT 1
#define DIRECTIONAL_LIGHT 2
#define FREE_LIGHT 3
```

6. CODE EXAMPLE

These are a few examples of code to show the use of the structures and data exported by the PVRExp plug-in.

```
/*!****************************************************************************
 * Function Name  : CreateObjectFromHeaderFile
 * Input/Output   :
 * Global Used    :
 * Description    : Function to initialise object from a .h file
 ******************************************************************************/
void CreateObjectFromHeaderFile (OBJECT *pObj, int nObject, int nVertexType)

{
int     i;
unsigned int    colour, nStride;
unsigned int    nVertexCount = 0, nCurrentVertex;
float       *pUV;
float       *pMGLVert;

    /* Get model info */
    pObj->nNumberOfVertices= Mesh[nObject].nNumVertex;
    pObj->nNumberOfTriangles    = Mesh[nObject].nNumFaces;
    pObj->nNumberOfStrips       = Mesh[nObject].nNumStrips;
    pObj->fCentreX          = Mesh[nObject].fCenter[0];
    pObj->fCentreY          = Mesh[nObject].fCenter[1];
    pObj->fCentreZ          = Mesh[nObject].fCenter[2];
    pObj->nMaterial     = Mesh[nObject].nMaterial;

    /* Vertices */
    pObj->pVertices=(float *)Mesh[nObject].pVertex;

    /* Normals */
    pObj->pNormals=(float *)Mesh[nObject].pNormals;

    /* Get triangle list data */
    pObj->pTriangleList=(unsigned short *)Mesh[nObject].pFaces;

    /* Get strips data */
    pObj->pStrips = (unsigned short *)Mesh[nObject].pStrips;
    pObj->pStripLength = (unsigned short *)Mesh[nObject].pStripLength;

    /* UVs */
    pUV = Mesh[nObject].pUV;

    /* Allocate memory for our */
    pObj->pVGPVertices = (float *)malloc(pObj->nNumberOfVertices * sizeof(float) * nStride);

    pMGLVert = pObj->pVGPVertices;

    if(pMGLVert==NULL)
    {
        printf ("\nERROR: Not enought memory for allocating vertices.\n");
        return;
    }

    /* Fill-up vertices in hardaware fashion */
    for (i=0; i<pObj->nNumberOfVertices; i++)
    {
        pMGLVert = (pObj->pVGPVertices+i*nStride);

        nCurrentVertex = i;

        if (nVertexType & VGP_POSITION)
        {
            *pMGLVert++ = pObj->pVertices[nCurrentVertex*3+0];
            *pMGLVert++ = pObj->pVertices[nCurrentVertex*3+1];
            *pMGLVert++ = pObj->pVertices[nCurrentVertex*3+2];
        }

        if (nVertexType & VGP_NORMALS)
        {
            *pMGLVert++ = pObj->pNormals[nCurrentVertex*3+0];
```

```
            *pMGLVert++ = pObj->pNormals[nCurrentVertex*3+1];
            *pMGLVert++ = pObj->pNormals[nCurrentVertex*3+2];
        }

        if (nVertexType & VGP_DIFFUSE)
        {
            if(Mesh[nObject].nFlags & MF_VERTEXCOLOR)
            {
                colour = Mesh[nObject].pVertexColor[nCurrentVertex];
            }
            else
            {
                colour = 0xFFFFFFFF;
            }
            *pMGLVert++ =LONG_TO_FLOAT(colour);
        }

        if (nVertexType & VGP_SPECULAR)
        {
            colour = 0x00000000;
            *pMGLVert++ =LONG_TO_FLOAT(colour);
        }

        if (nVertexType & VGP_UV1)
        {
        *pMGLVert++ = (Mesh[nObject].nFlags & MF_UV) ? pUV[nCurrentVertex*2+0] : 0.0f;
        *pMGLVert++ = (Mesh[nObject].nFlags & MF_UV) ? pUV[nCurrentVertex*2+1] : 0.0f;
        }

        if (nVertexType & VGP_UV2)
        {
        *pMGLVert++ = (Mesh[nObject].nFlags & MF_UV) ? pUV[nCurrentVertex*2+0] : 0.0f;
        *pMGLVert++ = (Mesh[nObject].nFlags & MF_UV) ? pUV[nCurrentVertex*2+1] : 0.0f;
        }
    }


}

/*****************************************************************************
 *  Function Name   : SetAnimMatrix
 *  Inputs          :
 *  Outputs         :
 *  Returns         :
 *  Globals Used    :
 *  Description     : To set a animation from 3DStudio MAX, feed the transformation matrix
 *                    with the values exported by the PVRexp plug-in.
 *****************************************************************************/
void SetAnimMatrix (MGLTMATRIX *pMatrix, int i, unsigned Count)
{

    /* If the frame is out of range return the identity matrix */
    if (Count<0 || Count > NUM_FRAMES)
    {
        MGLTMatrixReset (pMatrix);
        return;
    }

    /*
     *  Every chunk has 12 floats so, for example, the data for frame 32 of the object 2
     *  starts at (Animation[2].pData + 32*12 + 0) and carrys on for 12 floats.
     *  Note: M_14 = 0, M_24 = 0, M_34 = 0 and M_44 =1 are fixed values.
     */
    Count = Count*12;

        pMatrix->_11 = *(Bones[i].pData + Count + 0);
    pMatrix->_12 = *(Bones[i].pData + Count + 1);
    pMatrix->_13 = *(Bones[i].pData + Count + 2);
    pMatrix->_14 = 0.0f;

    pMatrix->_21 = *(Bones[i].pData + Count + 3);
    pMatrix->_22 = *(Bones[i].pData + Count + 4);
    pMatrix->_23 = *(Bones[i].pData + Count + 5);
    pMatrix->_24 = 0.0f;

    pMatrix->_31 = *(Bones[i].pData + Count + 6);
```

```
    pMatrix->_32 = *(Bones[i].pData + Count + 7);
    pMatrix->_33 = *(Bones[i].pData + Count + 8);
    pMatrix->_34 = 0.0f;

    pMatrix->_41 = *(Bones[i].pData + Count + 9);
    pMatrix->_42 = *(Bones[i].pData + Count + 10);
    pMatrix->_43 = *(Bones[i].pData + Count + 11);
    pMatrix->_44 = 1.0f;

}
```