S3D File Format

Overview 1
General file format conventions 1
Comments 1
Matrices and Euler Angles 2
S3D File Format 2
Master file format 3
Part list records 3
Texture list records 3
Triangle list records 4
Vertex list records 4
Light list records 4
Spotlight type-specific data (type=0) 5
Omni type-specific data (type=1) 5
Camera list records 5
S3D Extensions 6
Extension matProp – extended material properties [obsolete] 7
Extension matProp2 – extended material properties 7
Extension partTree – part hierarchy list 8
Extension posOrientList – part position/orientation list 9
Extension partUserTextList – arbitrary user text per part 9

Overview

This document describes the S3D file format. It has been taken from a larger document which describes several of the file formats used at Terminal Reality, Inc (TRI).

General file format conventions

All of the formats described here are simple text files, easily parsed using standard C functions like `fgets` and `fscanf`.

Comments

Files often have embedded "comments" at the beginning of lists or just before headers, to help the human reader interpret the file. However, none of the file formats allow for comments to be placed arbitrarily. (This would complicate the parsing of the file.) Comments are almost always one line long, on a line by themselves. Again, this is to make it as simple as possible to parse the file, while still retaining the usefulness of the comments to document the file format. Unless otherwise stated, comments are always present, even if the list they precede is empty.

It is often convenient to be able to skip one or more lines from a file, either to skip over comments, or simply to ignore portions of the file. The following function is extremely handy for this:

```
int     skipLine(FILE *f, int howMany = 1) {
        while (howMany > 0) {
                for (;;) {
                        int c = fgetc(f);
                        if ((c < 0) || (c == EOF)) {
                                // Error reading file
                                return 0;
                        }
                        if (c == '\n') break;
                }
                --howMany;
        }
        return 1;
}
```

Matrices and Euler Angles

Orientations are stored either in transform matrix form, or with TRI Euler angle pitch, bank, heading format. A transform matrix is stored where row 1 is the X axis (right vector) after transformation, row 2 is the Y axis (up vector) after transformation, and row 3 is the Z axis (forward vector) after transformation. Row 4 (if translation as well as rotation/scale is contained in the transform matrix) is the position of the origin after transformation, which is simply the translation portion of the transform matrix.

The conversion from Euler P,B,H angles to 3x3 matrix is:

rightX	=	ch * cb + sh * sp * sb
rightY	=	sb * cp
rightZ	=	-sh * cb + ch * sp * sb
upX	=	-ch * sb + sh * sp * cb
upY	=	cb * cp
upZ	=	sb * sh + ch * sp * cb
fwdX	=	sh * cp
fwdY	=	-sp
fwdZ	=	ch * cp

rightX is the x component of the right vector, after transformation, etc.
ch is cos(heading), etc.

All angles in the files are in stored as radians.

S3D File Format

The .S3D (*Simple 3D*) file format is used to store a basic texture-mapped triangle mesh. The file format also handles:

- part hierarchy identification within the mesh
- animated vertices (but not animated topology)
- lights and cameras,
- animated part positions/orientations.

Master file format

An overview of an S3D file looks like this:

> *comment – version field description*
> *version number*
> *comment – header field descriptions*
> *textureCount*, *triCount*, *vertexCount*, *frameCount*, *partCount*,
>   *lightCount*, *cameraCount*
> *comment – part list field descriptions*
> *partCount part records*
> *comment – texture list field descriptions*
> *textureCount texture records*
> *comment – texture list field descriptions*
> *triCount triangle records*
> *comment – vertex list field descriptions*
> *vertexCount*frameCount vertex records*
> *comment – light list field descriptions*
> *lightCount light records*
> *comment – camera list field descriptions*
> *cameraCount camera records*
> *extensions*

Part list records

A single part record has the following format:

> *firstVertexIndex*, *vertexCount*, *firstTriIndex*, *triCount*, "*partName*"

Use the following format string to parse:

```
"%d,%d,%d,%d,\"%[^\"]\"\n"
```

Duplicate part names are allowed, but **empty part names are not allowed**.

Texture list records

Each texture is stored simply using the filename of the texture, with no surrounding quotes. Watch out for filenames with spaces in them! The following format string properly parses out the texture name:

```
"%[^\n]\n"
```

Triangle list records

A single triangle record has the following format:

```
textureIndex, vertexIndex1,u1,v1, vertexIndex2,u2,v2,
        vertexIndex2,u2,v2
```

Use the following format string to parse:

```
"%d,%d,%f,%f,%d,%f,%f,%d,%f,%f\n"
```

textureIndex is an index into the texture list, identifying the texture to use on this triangle. If –1, then the face is not texture mapped.

vertexIndex1, *vertexIndex2*, and *vertexIndex3* are vertex indices into the master vertex list. They are not offset relative to the start of the vertices for the part containing the triangle. (In other words, you could ignore the part list and still parse the mesh correctly.)

The UV coordinates are floating point values. 0,0 corresponds to the upper-left-hand corner of the texture image, and 256,256 corresponds to the lower-right-hand corner. The values may exceed this range to accommodate texture tiling. However, most of the exporters "normalize" the UVs on a per-triangle basis by adding/subtracting the appropriate multiple of 256 to obtain the smallest possible positive values. This does not effect the appearance of the model. If the face is not texture-mapped, the UVs will be present, but their value is undefined.

Vertex list records

The vertices are stored in the obvious format:

```
x, y, z
```

Use the following format string to parse: (Duh)

```
"%f,%f,%f\n"
```

NOTE: Vertices are always given in absolute world coordinates, NOT local part coordinate space. (To determine the node coordinate space coordinates, you can multiply by the inverse of the node's transform matrix, which can be determined from the **posOrientList** extension. See page 9 for more details.) If more than one frame is contained in the file, then the vertices for all parts for the first frame are listed, followed by the second frame, etc…

Light list records

If we had this to do over, the lights and cameras would be extensions, rather than a required part of the file. Oh well, just ignore them if you don't need them.

Each light record is on a single line. The format is as follows:

> *"name"*, *type*, *x*, *y*, *z*, *r*, *g*, *b*, *type-specific-data*

Use the following format string to parse the data before the type-specific data

```
"\"%[^\"]\",%d,%f,%f,%f,%f,%f,%f "
```

Currently the following light types are supported:

0	Spot light
1	Omni light

The RGB color values are floating point values in range [0…255].

Spotlight type-specific data (type=0)

> *pitch*, *bank*, *heading*

Omni type-specific data (type=1)

> *attenuationStart*, *attenuationEnd*

These are floating point attenuation distance values. If there is no attenuation, then both values will be –1.

Camera list records

Each camera record is 5 lines long, as follows:

```
"name", x, y, z, pitch, bank, heading,
        horizontalFieldOfViewInRadians
rightX, rightY, rightZ
upX, upY, upZ
fwdX, fwdY, fwdZ
x, y, z
```

Lines 2-5 form the 4x3 camera transform matrix. The matrix is always orthonormal. (Not corrected for zoom or aspect ratio.) Yes, this is redundant since the matrix can be computed from the Euler angles, and the last row is the same as the camera position. But at one point we were trying to get cameras to line up with cameras in Max, and the Euler angle computations were suspect, so now the matrix is there as a legacy. Just ignore it if you don't need it.

Use the following format string to parse out the first line of the camera record. (The other lines are pretty obvious how to parse, aren't they?)

```
"\"%[^\"]\",%f,%f,%f,%f,%f,%f,%f\n"
```

S3D Extensions

Rather than incrementing the version number every time a new feature is added to the .S3D format, a simple extension system was introduced. All extensions are optional for both exporters and importers. However, **no exporter may export extensions out of order with respect to this document**. An importer may assume that all extensions that appear in the .S3D file appear in the same order as they are listed in this document.

The first line of text for each extension is the same:

extensionName length

extensionName is a unique name which identifies the extension.
length specifies the number of lines of text which follow.

Some rules regarding extension names:

- less than 40 characters in length
- alphanumeric characters only. **No spaces**.
- Case is not significant.

If the importer does not recognize the extension (or simply does not need to process it), it may simply ignore the next *length* lines, and continue with the next extension.

Be very careful when parsing extensions, because, unlike other parts of the file, some extensions may possibly contain completely blank lines, and a naïve parser could get off track.

A typical importer will parse the extensions using code similar to the following. (Note that the following code correctly handles blank lines.)

```
char   extensionHeader[100];
char   extensionName[100];
int    lineCount;
while (fgets(extensionHeader, sizeof(extensionHeader), f) != NULL) {
       if (sscanf(extensionHeader, "%s %d", extensionName, &lineCount) != 2)
       {
              // error - file is invalid
       }

       // Handle the extensions we know how to handle

       if (stricmp(extensionHeader, "matProp2") == 0) {
              // handle the extension
              continue;
       }

       if (stricmp(extensionHeader, "posOrientList") == 0) {
              // handle the extension
              continue;
       }

       // Anything else, just ignore the extension.

       if (!skipLine(f, lineCount)) {
              // error - the file is invalid
       }

}
```

Extension matProp – extended material properties [obsolete]

Note: This extension is obsolete, please use matPropX instead.

The **matProp** extension gives additional material properties other than the simple diffuse texture map name. There is a simple 3-line comment header which documents the fields, followed by 3-line records, one for each entry in the original texture list. Each record is formatted as follows:

```
shininess, shininessStrength, opacity
"bumpMapFilename"
"opacityMap"
```

If there is no bump map or opacity map, then the corresponding filename will be empty, and the line will only contain an empty pair of quotes.

Extension matProp2 – extended material properties [obsolete]

Note: This extension is obsolete, please use matPropX instead.

The **matProp2** extension gives additional material properties other than the simple diffuse texture map name. There is a simple 5-line comment header that documents the fields, followed by 5-line records, one for each entry in the original texture list. Each record is formatted as follows:

kDiffuseR,kDiffuseG,kDiffuseB
kSpecularR,kSpecularG,kSpecularB, specularPower
"bumpMapFilename"
"detailMapFilename"
diffuseToDetailUVMatrix (3x2 matrix)

The diffuse and specular colors are floating-point color values in range [0…255]. The specular exponent is a nonnegative floating-point value.

If there is no bump map or detail map, then the corresponding filename will be empty, and the line will only contain an empty pair of quotes.

The detailMapTransform is the 3x2 matrix which transforms UV coordinates from diffuse map space into detail map space. The transform is defined by:

```
         |m11 m12|
[U V 1]  |m21 m22| = [U' V']
         |m31 m32|
```

And the values are listed in the following order:

```
m11,m12, m21,m22, m31,m32
```

If no detail texture is used, then an identity matrix should be listed:

```
1,0, 0,1, 0,0
```

Extension matPropX – extended material properties

The **matPropX** extension gives additional material properties other than the simple diffuse texture map name. It uses a simple, extensible tag-based system so that new material properties can be added easily without breaking the format.

Each material starts with a single 1-line header that contains the number of lines of text to follow for that material (not including the header line itself). Each of these subsequent lines is of the form:

tag:value

tag is a short identifier, containing no spaces, that identifies what material property is being defined. *value* is the value of the property. The format of this field varies depending on the property being defined.

The following tags are currently supported:

```
heightMap:"filename"
glossMap:"filename"
groundType:"name"
diffuseTile:[u={clamp|wrap}] [v={clamp|wrap}]
specular:r,g,b,power
```

Extension partTree – part hierarchy list

The **partTree** extension is provided so that the part tree linkage can be reconstructed. The linkage is represented simply by listing the index of the parent of each part. For parts with no parent, an index of –1 is used. The parts are assumed to be in the same order as they appear in the part list. The number of lines in the extension will always be the same as the number of parts in the file.

Extension posOrientList – part position/orientation list

The **posOrientList** contains the position and orientation of each part in world space, exported simply as a position and set of Euler angles for each part for each frame. All the positions/orientations for the first frame are exported, followed by the second frame, etc. Each position/orientation record is exported on a single line as follows:

x,y,z, p, b, h

The number of lines in the extension should be *partCount x frameCount*.

Note that the position and orientation are in absolute world space, NOT relative to its parent. To determine the orientation relative to the parent, you must multiply by the inverse of the parent's transformation matrix.

Extension partUserTextList – arbitrary user text per part

The **partUserTextList** extension is used to assign arbitrary user text data to each part. The data for each part is stored simply as a count of lines of text for that part, followed immediately by N lines of text. This is immediately followed by the next part's line count, etc. For example, if the scene contains 4 parts, the following would be a valid **partUserTextList** block:

```
partUserTextList 10
3
This is arbitrary user data for the first part.
It has 3 lines of text.
This is the last data for the first part.
1
The second part has only one line of text.  This is it.
0
2
Notice how the 3rd part didn’t have any user data.
But this part (the 4th part) has two lines.
<the next extension would begin here>
```

NOTE: Leading and trailing whitespace should not be depended onto be maintained exactly. However, internal whitespace will be maintained properly. Blank lines are highly discouraged, simply because it could produce parse bugs if the parser is written naively. If it is critical to maintain leading and trailing whitespace, you should “armor” your data, by enclosing it in quotes, or using some other similar scheme. Each line of text should not exceed 512 characters.