



File Formats

File Formats
Copyright and trademarks

StudioTools 13

Software copyright information is located in the application, and can be accessed from the menu by choosing Help > About StudioTools.

All documentation ("Documentation") is copyrighted © 2001-2005 Alias and contains proprietary and confidential information of Alias. The Documentation is protected by national and international intellectual property laws and treaties. All rights reserved. Use of the Documentation is subject to the terms of the license agreement that governs the use of the software product to which the Documentation pertains ("Software"). The authorized licensee of the Software is hereby authorized to print no more than one (1) hardcopy of any Documentation provided in digital format per valid license of the Software held by such licensee. Except for the foregoing, the Documentation may not be translated, copied or duplicated in any form (physically or electronically), in whole or in part, without the prior written consent of Alias.

Alias and the swirl logo, Maya and DesignStudio are registered trademarks and Alias Natural Phenomena, Alias OpenAlias, Alias OpenModel, Alias PowerCaster, Alias PowerTracer, Alias RayCasting, Alias RayTracing, Alias SDL, ImageStudio, Alias Spider, StudioPaint, StudioViewer, StudioTools and SurfaceStudio are trademarks of Alias Systems Corp. ("Alias") in the United States and/or other countries. Silicon Graphics, SGI and IRIX are registered trademarks and Inventor is a trademark of Silicon Graphic, Inc. in the United States and/or other countries worldwide. Microsoft and Windows are either registered trademarks or trademarks of Microsoft Corporation in the United States and/or other countries. Renderman is a registered trademark of Pixar Corporation. Apple, Quicktime and Macintosh are trademarks of Apple Computer, Inc. registered in the United States and other countries. Adobe, Postscript and Illustrator are either registered trademarks or trademarks of Adobe Systems Incorporated in the United States and/or other countries. Unigraphics, NX, and I-deas are registered trademarks or trademarks of UGS Corp. or its subsidiaries in the United States and in other countries. Arius3D is a registered trademark of Arius3D Inc. Cyberware is a registered trademark of Cyberware Laboratory Inc.. Cyrax is a registered trademark of Leica Geosystems HDS Inc. Steinbichler is a registered trademark of Steinbichler Optotechnik GmbH. Autodesk and AutoCAD are either registered trademarks or trademarks of Autodesk, Inc./Autodesk Canada, Inc. in the USA and/or other countries. CATIA is a registered trademark of Dassault Systèmes S.A. PTC, Pro/ENGINEER and Granite are trademarks or registered trademarks of Parametric Technology Corporation or its subsidiaries in the U.S. and in other countries. All other trademarks mentioned herein are the property of their respective owners.

All PTC Technology logos are used under license from Parametric Technology Corporation, Needham, MA, USA.

Not all features described are available in all products.



Alias Systems Corp., 210 King Street East, Toronto, Canada M5A 1J7

Contents

File Formats 1

Platforms 2

Alias Pix image file 3

Matte file 5

Alias camera depth map file 7

Animation SDL 9

Maya IFF 16

Particle file 20

Bulge definition file 22

Index 27

File Formats

Describes the internal structure of various file formats used by StudioTools.

Platforms

Describes file format support on the different platforms.

The following table shows which file formats are supported on the different StudioTools platforms.

File Format	Platforms
Alias Pix image file	IRIX, Windows
Matte file	IRIX, Windows
Camera depth map file	IRIX, Windows
Animation SDL	IRIX, Windows
Maya IFF	Windows (reading only)
Particle file	IRIX, Windows
Bulge definition file	IRIX, Windows

Alias Pix image file

Platforms

IRIX, Windows

Description

An Alias pix file has a 10-byte header containing 5 short integers (there is no explicit magic number) which is then followed immediately by image data in a simple run-length encoded scheme.

Only RGB information is contained in the file. Matte files are similar but exist in a separate file (see *Matte File Format* for details).

bytes	header value	notes
0, 1	width	x resolution in pixels
2, 3	height	y resolution in pixels
4, 5	xoffset	unused
6, 7	yoffset	unused
8, 9	bits/pixel	24 for pix files (0x18)

The pixels are then run-length encoded in 4-byte packets on a per-scanline basis (runs do not extend beyond a single scanline) starting with the top scanline in the image.

bytes	data range	notes
runlength	1 - 255	number of pixels in succession with given RGB
blue	0 - 255	value of blue component
green	0 - 255	value of green component
red	0 - 255	value of red component

Example

Here is the output of `od -x` for a `pix` file that is 8 pixels wide and 6 pixels high, representing a ramp that goes from black at the bottom of the image to blue at the top:

```
0000000 0008 0006 0000 0005 0018 08ff 0000
08cc
0000020 0000 0899 0000 0866 0000 0833 0000
0800
0000040 0000
0000042
```

This is read as describing an image that is 8 pixels wide [0008] and 6 scanlines high [0006]. The next four bytes describe the obsolete offset information. This is a `pix` file since there are 24 bits/pixel [0018]. The first (top) scanline is composed of a run of 8 pixels of (B=255, G=0, R=0)[08ff 0000]. The next scanline (since this one is complete) is 8 pixels of (B=204, G=0, R=0) [08cc 0000]. The rest of the scanlines are coded in the same fashion with the last scanline of eight pixels of (B=0, G=0, R=0) [0800 0000].

Matte file

Platforms

IRIX, Windows

Description

Alias matte files are a variant of Alias pix files.

bytes	header value	notes
0, 1	width	x resolution in pixels
2, 3	height	y resolution in pixels
4, 5	xoffset	unused
6, 7	yoffset	unused
8, 9	bits/pixel	8 for matte files (0x8)

The coverage (matte) information is then run-length encoded in 2-byte packets on a per-scanline basis (i.e. runs do not extend beyond a single scanline) starting with the top scanline in the image, where a value of zero indicates no coverage and a value of 255 indicates complete coverage of that pixel:

bytes	data range	notes
runlength	1 - 255	number of pixels in succession with given coverage
coverage	0 - 255	value for coverage

Example

The following is an octal dump of an 8x6 Alias matte file for a sphere that nearly fills the image:

```
0000000 0008 0006 0000 0005 0008 0100 0110 015f
0000020 02bf 015f 0110 0100 0100 015f 04ff 015f
0000040 0100 0100 01bf 04ff 01bf 0100 0100 01bf
0000060 04ff 01bf 0100 0100 015f 04ff 015f 0100
0000100 0100 010f 015f 02bf 015f 010f 0100
0000116
```

This is read as describing an image that is 8 pixels wide [0008] and 6 scanlines high [0006]. The next four bytes describe the obsolete offset information. This is a matte file since there are 8 bits/pixel [0008]. The description below shows the remainder of the file which describes the matte from the top scanline to the bottom.

```
0100 - one pixel of 0/255 (0%) coverage,
0110 - one pixel of 16/255 (6%) coverage,
015f - one pixel of 95/255 (37%) coverage,
02bf - two pixels of 191/255 (75%) coverage,
015f - one pixel of 95/255 (37%) coverage,
0110 - one pixel of 16/255 (6%) coverage,
0100 - one pixel of 0/255 (0%) coverage.
```

Now you know you are on the next scanline since the first is filled.

```
0100 - one pixel of 0/255 (0%) coverage,
015f - one pixel of 95/255 (37%) coverage,
04ff - four pixels of 255/255 (100%) coverage,
015f - one pixel of 95/255 (37%) coverage,
0100 - one pixel of 0/255 (0%) coverage.
```

Moving on to the next scanline:

```
0100 - one pixel of 0/255 (0%) coverage,
01bf - one pixel of 191/255 (75%) coverage,
04ff - four pixels of 255/255 (100%) coverage,
015f - one pixel of 95/255 (37%) coverage,
0100 - one pixel of 0/255 (0%) coverage.
```

And so on. The next scanline is exactly the same, the one after that matches the second and the last one is the same as the first. This is the fortunate result of choosing a sphere.

Alias camera depth map file

Platforms

IRIX, Windows

Description

A StudioTools Camera Depth file contains depth information corresponding to the image created from that camera. The Camera Depth file is used for post-render 3D compositing. The file contains a magic number, an X and Y resolution, and an array of floating point depth values.

bytes	header value	notes	C-type
0, 1, 2, 3	magic number	Uniquely identifies files of this type	int
4, 5	width	x resolution in pixels	short
6, 7	height	y resolution in pixels	short

The magic number for StudioTools camera depth files is 55655. The rest of the file contains an X by Y array of floating point values in row order.

Example

The following C-code is an example of how to read a camera depth file:

```
filein = open( infile, O_RDONLY );

read( filein, &magic, sizeof( int ) ); /* magic
number */
if ( magic != 55655 ) {
    fprintf( stderr, "given input file '%s' does
not have proper magic number (55655)\n", infile
);
    exit(0);
}

read ( filein, &width, sizeof(short) ); /* Xres */
```

```
read ( filein, &height, sizeof(short) ); /* Yres */

size = width * height;

buffer = (float *)malloc ( size * sizeof( float ) );
read( filein, buffer, sizeof(float)*size2 ); /*
fill the array */

close( filein );

for (i = 0; i < height; ++i) {
    for (j = 0; j < width; ++j) {
        /* Do something to the pixel. */
    }
}
```

Animation SDL

Platforms

IRIX, Windows

Description

An Animation SDL file is a regular Scene Description Language file with an extra section to describe the model's hierarchies.

An Animation SDL file has two sections, DEFINITION and HIERARCHY.



The HIERARCHY section may be omitted. In this case, the Animation SDL file contains only a library of animation curves (or actions).

In the DEFINITION section, you specify the description of the curves (or actions) as you do in regular SDL. The syntax for actions is the same for Animation SDL and regular SDL.

In the HIERARCHY section, you describe how the animation curves in the DEFINITION section are applied to the model's hierarchy.

- Braces, { }, bracket these descriptions. You must supply at least one set of braces.
- The order of the braces describes the hierarchy.
- A matching pair of braces inside another refers to an object that is a child of the outer pair's object.
- Matching pairs that follow one another indicate that the pairs are children of the same parent.

Example 1

```
HIERARCHY
{ <----- Picked object
Apply animation to picked object.
{ <----- A child of the picked object.
Apply animation to child of picked object.
{ <----- A child of a child of the picked object.
Apply animation to child of a child of picked
object.
```

```

}
}
{ <----- Another child of the picked object.
Apply animation to other child picked object.
}
}

```

If an object has no animation, but a descendant does, you must still specify the braces as a place-holder for the object, but omit the statements applying any animation to it. This is also necessary for siblings. If the leftmost sibling is unanimated, but the one to its right is animated, specify an empty pair of braces as a place-holder for the leftmost child.

There are two statements in the HIERARCHY section that apply animation to objects. The first is the type statement, which indicates the type of object that will receive the animation.

Specifying the type of object

The general form of the type statement is:

```

type "animatable item type name" ( <additional
information> );

```

Examples of the "animatable item type name" are Dag Node, Camera, Light, Shader, Surface CV, and Curve CV.

- For a harvDag Node, specify a name for the dag node.
- For Animation SDL, a Shader includes textures. Lights and shaders have no additional information, so the parentheses should be empty.
- For a Surface CV, specify the U and V values for the CV, separated by a comma. For example, if the u=2 and v=3 CV on a surface is animated, the type statement is written as:
 - type "Surface CV" (2, 3);
- For a Curve CV, specify which CV of the curve is being referred to.

If the curve is part of a face, two values are specified separated by a comma. The first value specifies the curve of the face, and the second value specifies the CV on the curve. For example, if a face is made up of 3 curves, then to refer to the 4th CV on the 2nd curve of the face, the type statement is written as:

```
type "Curve CV" (2, 4);
```

- For a Polyset Vertex, specify the vertex number in the polyset.

```
type "Polyset Vertex" (62);
```

- No other animatable items have additional information, so their parentheses should be empty.

Applying the animation

The statements following the type statements indicate how animation is applied to each of the item's animation parameters. These are called channel statements, and specify how the animation parameter uses the actions from the DEFINITION section.

The general form of the channel statement is:

```
channel "channel name" (action_name [extract axis]
(action_name...));
```

Each animatable item type has its own set of animation parameter names, which can be seen in the param control window (see [Animation > Editors > Param control](#) for details). For example, Dag Nodes have X Translate, Y Translate, Z Scale, and Visibility. These names, which appear in the param control window, can be used as channel name.



Do not assign a channel to an item that does not animate in that parameter. Also, do not assign a channel that is not part of the current animation item type. For example, Animation SDL does not differentiate between types of lights, so it is possible to try to read a *Spot spread* channel onto a point light, but that results in an error.

Within the parentheses of the channel statement, you specify the list of actions that make up the channel. If the action is a motion path, you must specify which axis of the 3D NURBS curve to use. Additional actions that are within additional parentheses act as timewarps on the original curve.

Example 2

The following is an example Animation SDL file of an animated cylinder.

```
DEFINITION
    /* the 3-D NURBS curve used by the motion
    path action */
    curve curve#2 (
        degree = 3,
        knots = (0.0, 0.0, 0.0, 1.0, 1.0 , 1.0
    ),
        cvs = (
            cv( (0.0, 0.0, 0.0),1.0),
            cv( (3.0, 3.0, 3.0),1.0),
            cv( (-4.0, 2.0, -6.0),1.0),
            cv( (-3.0, 2.0, 3.0),1.0) )
    );
    /* a motion path action */
    motion_curve motion_path ( curve#2, in =
    PRE_CONSTANT, out = POST_CONSTANT );

    /* several parameter curve actions */
    parameter_curve param_curve.Timing ( in =
    PRE_LINEAR, out = POST_LINEAR, cvs = (
        parameter_vertex(1.0,0.0 ,
    TAN_SMOOTH, ( -0.27852, -0.96043 ), TAN_SMOOTH,
    (0.27852,0.96043) ),
        parameter_vertex(30.0,30.0,
    TAN_SMOOTH, (-0.27852,-0.96043 ), TAN_SMOOTH,
    (0.27852,0.96043) )
    ) );

    parameter_curve param_curve.X_Scale ( in =
    PRE_CONSTANT, out = POST_CONSTANT, cvs = (
        parameter_vertex(1.0 , 1.0 ,
    TAN_SMOOTH, (1.0 ,0.0), TAN_SMOOTH, (1.0 , 0.0) ),
        parameter_vertex( 30.0 , 4.0 ,
    TAN_SMOOTH, (1.0 ,0.0), TAN_SMOOTH, (1.0 , 0.0) )
    ) );

    parameter_curve param_curve.Z_Rotate ( in =
    PRE_CONSTANT, out = POST_CONSTANT, cvs = (
        parameter_vertex(1.0 , 0.0,
    TAN_SMOOTH, (-1.0 , 0.0), TAN_SMOOTH, (1.0 , 0.0)
    ),
        parameter_vertex(30.0 ,360.0 ,
    TAN_SMOOTH, (-1.0 , 0.0 ), TAN_SMOOTH, (1.0, 0.0) )
    ) );
```

```

        parameter_curve timewarp ( in =
PRE_IDENTITY, out = POST_IDENTITY, cvs = (
            parameter_vertex(1.0 , 1.0,
TAN_SMOOTH, (-0.70711,-0.70711 ), TAN_SMOOTH,
(0.70711,0.70711) ),
            parameter_vertex(30.0 ,30.0
,TAN_SMOOTH, (-0.70711,-0.70711 ),TAN_SMOOTH,
(0.70711,0.70711) )
        ) );

        parameter_curve param_curve.X_Position ( in
= PRE_CONSTANT, out = POST_CONSTANT, cvs = (
            parameter_vertex(1.0 , -0.2612 ,
TAN_SMOOTH, (-1.0 ,0.0), TAN_SMOOTH, (-1.0 ,0.0) ),
            parameter_vertex( 30.0 , -0.18593,
TAN_SMOOTH, (-1.0 ,0.0), TAN_SMOOTH, (-1.0 ,0.0) )
        ) );

        parameter_curve param_curve.Y_Position ( in
= PRE_CONSTANT, out = POST_CONSTANT, cvs = (
            parameter_vertex(1.0 ,0.2612 ,
TAN_SMOOTH, (-1.0 ,0.0 ), TAN_SMOOTH, (-1.0 ,0.0)
),
            parameter_vertex( 30.0 , 0.58722,
TAN_SMOOTH, ( -1.0 , 0.0 ), TAN_SMOOTH, (-
1.0 ,0.0))
        ) );

        parameter_curve param_curve.Z_Position ( in
= PRE_CONSTANT, out = POST_CONSTANT, cvs = (
            parameter_vertex( 1.0 , 0.5 ,
TAN_SMOOTH, ( -1.0 , 0.0 ), TAN_SMOOTH, (
1.0 , 0.0 ) ),
            parameter_vertex( 30.0, 0.90137,
TAN_SMOOTH, ( -1.0 , 0.0 ), TAN_SMOOTH, (-
1.0 ,0.0))
        ) );

HIERARCHY
{
    type "Dag Node" ( cylinder );

    /* The cylinder is moved along a
motion path. */
    /* Each channel is extracted from a
motion path */
    /* with one timing curve modifying all
three. */
    channel "X Translate" ( motion_path
[X] ( param_curve.Timing ));
}

```

```

        channel "Y Translate" ( motion_path
[Y] ( param_curve.Timing ));
        channel "Z Translate" ( motion_path
[Z] ( param_curve.Timing ));

        /* The width of the cylinder is also
animated. */
        channel "X Scale" (
param_curve.X_Scale );
        {
            /* This is the first child of the top
level */
            /* of the cylinder. It spins around
the Z */
            /* axis. A timewarp has been applied.
*/
            type "Dag Node" ( cyl_body );
            channel "Z Rotate" (
param_curve.Z_Rotate ( timewarp ));
        }
        /* Cap A of the cylinder is not
animated, */
        /* nor are any of its CVs, but these
braces */
        /* are necessary to maintain to
maintain the*/
        /* hierarchy structure.
*/
        /* If these braces were omitted Cap
B's */
        /* animation would be read onto Cap A.
*/
        }
        {
            /* Cap B is not animated but one of
its cvs */
            /* is.
*/
            {
                type "Surface CV" ( 2, 4 );
                /* Surface CV u = 2, v = 4 is
animated. */
                channel "X Position" (
param_curve.X_Position );
                channel "Y Position" (

```

```
param_curve.Y_Position );  
        channel "Z Position" (  
param_curve.Z_Position );  
    }  
}
```

Maya IFF

Platforms

Windows

StudioTools for Windows can read Maya IFF files and save them as TIFF or Alias pix files.

Description

- The basic element of an IFF file is a block, sometimes called a chunk.
- Each block has an identifier called a tag or ID and a block length to allow it to be skipped (in some cases the block length is missing and the end of the block is indicated by a special marker).
- Several grouping block types allow the file to be structured as a hierarchy.

How to Read Maya IFF files in StudioTools

The Maya IFF file format is not recognized inside file lister as a usable format. You can use the commands *file->show->pix* or try to use an IFF file as a texture. Maya IFF files can be brought in as images planes for Cameras.

To use IFF files as a texture you must enter the absolute path for the texture and not use the browse button to open file lister.

Basic file structure

The structure is based on the use of tags to identify blocks of data called chunks or structures of chunks called groups. Each tag is made up of four characters and is immediately followed by the size of the chunk or group that it describes coded on 4 bytes. Tags are handled as pseudo-character strings and all other data is written in big-endian format.

Block type tags

The main tag types are FORM, CAT, LIST and PROP.

They come in several flavors, such as FOR4, FOR8, CAT4 and CAT8 to specify 4-byte or 8-byte alignment boundaries.

- FORM defines a structure that is similar to a C struct, that is, an ordered set of structured data.
- CAT defines a concatenation of independent objects with no order relation between them.
- LIST is used to group objects with similar properties, avoiding redundancy as the common properties can be defined before the list in a PROP block. A List has an extra tag value to indicate what it is a list of.
- PROP further defines the properties of objects.

Groups

Four tags are used to arrange blocks into groups: FORM, CAT, LIST, and PROP. The first four characters following the size are used to identify the type of the group.

The FORM defines a structure that is like a C struct.

```
FORM 38 TEXT
    CHAR 6 "Times"
    CHAR 12 "Hello World"
EOF
```

is like

```
struct Text t = {
    char *f = "Times";
    char *c = "Hello World";
};
```

The size of the group (38) equals the size of the data it contains (6 plus 12) plus the size of the headers (4 for TEXT, 8 for CHAR 6 and 8 for CHAR 12). In this case, the result is $6+12+4+8+8 = 38$.

As in C structures you can nest groups; for example:

```
FORM 52 TEXT
    FORM 8 FONT
        CHAR 6 "Times"
        LONG 4 <12>
        LONG 4 <0>
    CHAR 12 "Hello World"
EOF
```

or in C terms:

```
struct Text t = {
    struct Font f = {
        char *n = "Times";
        int s = 12;
        int d = 0;
    };
    char *string = "Hello World";
};
```

This example may not show that blocks are not constrained to use a unique data type and may contain the equivalent of a complete C structure.

The FORM tag separates independent blocks of data that can be handled separately and specifies the meaning of each subunit.

In the example above, the CHAR chunk in the FONT FORM does not mean the same thing as the CHAR chunk in the TEXT FORM. The FORM tag determines how you interpret an ordered set of data types.

The CAT tag defines a concatenation of independent objects with no order relation between them. Two typical uses of CATs are for libraries of objects (pictures in Example 1) or clipboards (Example 2).

Example 1:

```
CAT 3632 PICT
    FORM 1234 PICT ...
    FORM 2378 PICT ...
EOF
```

Example 2:

```
CAT 2130 CLIP
    FORM 1234 PICT ...
    FORM 876 DRAW ...
EOF
```

Searching through a structured file is generally greatly accelerated, even in a CAT that has no order amongst its members, through the knowledge of the size of every group or chunk specified in the header.

The LIST tag is used to group objects with similar properties, avoiding redundancy. For example, a sequence of equal-sized images might be represented in the following way. One image would have a structure like:

```
FORM ... PICT
      IHDR 32 (image size info)
      BODY ... (image data)
EOF
```

then a sequence of like sized images could be done as follows, sharing the common header information:

```
LIST ... ANIM
      PROP 44 PICT
            IHDR 32 (common size info)
      FORM ... PICT
            BODY .... (data)
      FORM ... PICT
            BODY .... (data)
      FORM ... PICT
            BODY .... (data)
EOF
```

The information in a PROP construct is valid until the end of the LIST. It can be redefined locally in a FORM statement. (In the above example the common IHDR is valid in all PICTs that don't include an IHDR block of their own.)

Alignment

IFF blocks align to 2-byte boundaries. The size specified in the header does not take padding into account. Many computers typically align their memory on 4-byte or 8-byte boundaries. **Flib** uses eight extra TAGs to let you specify alignment information:

- These four are used to align to 4-byte boundaries: FOR4, CAT4, LIS4 and PRO4.
- These four are used to align to 8-byte boundaries: FOR8, CAT8, LIS8 and PRO8.

Particle file

Platforms

IRIX, Windows

Description

The particle file format is a simple ASCII listing of the current status of every particle in the system. Rendering, mass, generator, and other information is carried in the wire file or SDL file. The file is all space-separated:

Field	Type	Comments
<file_type>	Integer	Two possible values currently: 4 or 5
<num_blobs>	Integer	The number of particles in this file.

<blob_info> (See the following table. There are 'num_blobs' entries of this type.)

Normal particles

For type 4 particles (normal), information for each blob is described below.

Field	Type	Comments
<start_x>	Float	X position of particle at start of frame.
<start_y>	Float	Y position of particle at start of frame.
<start_z>	Float	Z position of particle at start of frame.
<end_x>	Float	X position of particle at end of frame.
<end_y>	Float	Y position of particle at end of frame.
<end_z>	Float	Z position of particle at end of frame.
<R>	Float	Red color of particle in this frame.
<G>	Float	Green color of particle in this frame.
	Float	Blue color of particle in frame.

<cycles_left>	Integer	Number of cycles left in the particle's life.
<cycles_total>	Integer	Total number of cycles in the particle's life.

Diffusing gas particles

For type 5 particles (diffusing gas particles), information for each blob is described below.

Field	Type	Comments
<start_x>	Float	X position of particle at start of frame.
<start_y>	Float	Y position of particle at start of frame.
<start_z>	Float	Z position of particle at start of frame.
<end_x>	Float	X position of particle at end of frame.
<end_y>	Float	Y position of particle at end of frame.
<end_z>	Float	Z position of particle at end of frame.
<R>	Float	Red color of particle in this frame.
<G>	Float	Green color of particle in this frame.
	Float	Blue color of particle in this frame.
<cycles_left>	Integer	Number of cycles left in the particle's life.
<cycles_total>	Integer	Total number of cycles in the particle's life.
<start_size>	Float	Size of particle at start of frame.
<end_size>	Float	Size of particle at end of frame.

Bulge definition file

Platforms

IRIX, Windows

Description

The character builder feature allows you to create muscle bulges on the geometry around skeleton bones. You can freely define their shape using the Section editor in the CHARACTER BUILDER frame type of the deformation control window.

You can define any number of bulge shapes. The bulge definition shapes are not stored in the wire file, but in an ASCII file called `bulge_types` which resides in your `misc_data` directory of your current project. The advantage of storing these bulge shapes in a separate file outside of your wire file is that you can build a catalogue of bulge shapes, and reuse them from wire file to wire file, and even from project to project (you can copy the `bulge_types` file from one project directory to another).



Be careful when you modify or delete bulge definitions in the interactive package.

When you “assign attributes” to a skeleton joint, one of the attributes is the bulge definition. The skeleton joint does not store the whole bulge definition, but rather the bulge definition code. Thus if you have several wire files that use the same bulge definition and you modify the bulge definition in the interactive package, you are actually modifying the bulge definition for all wire files that reference that bulge code. In particular, if you delete a bulge definition in StudioTools, you may be deleting a bulge definition that is used not only by a skeleton joint in your current StudioTools session, but also in any other wire file you may have previously saved that used that bulge code.

In StudioTools, if you start to delete a bulge definition, a confirm box will appear warning you to this effect.

bulge_types file format

Each bulge shown in the Bulge Definitions Lister in the Deformation Control window is defined in the `bulge_types` file in the `misc_data` directory.

If this file doesn't exist, StudioTools copies a default version of the file to the `misc_data` directory.

A bulge definition comprises a sequence of keywords (followed by a colon) followed by values related to the keyword.

A bulge definition is complete when it contains one "Initialization" section, at least one "Section Definition" per Initialization, and at least two "Keypoint Definitions" per Section Definition.

Any incomplete or poorly defined bulge definition is omitted from the Bulge Definition Lister entries. The errlog will contain a record of bulges not added to the Lister.

Bulge initialization

- The beginning of a bulge definition is marked by a line containing two keywords: "bulge" and "code".
- `>> bulge: "bulge name" code: unique_id`
- The "bulge name" string *must* be enclosed in double quotes. Bulge names may contain spaces as well as most other standard characters (ASCII range 32 to 126) *except* the double quote.
- The `unique_id` value is an integer that uniquely identifies this bulge in the `bulge_types` file. If the bulge file contains a definition that uses a bulge code already used by another definition, a new and unique code will be assigned to that bulge definition and a message is appended to the errlog. The bulge code must be an integer 0 or larger.

Section Definition

- The next part of the file defines how bulges should appear at various locations or "sections" around the bone.
- `>> angle: 0`

- The Section Definition contains the keyword “angle” followed a degree value, locating the profile curve (defined by the keypoint definitions that follow) around the bone at the given angle. The angle value must be an integer between 0 and 360.



A bulge must contain at least one Section Definition.

Keypoint Definition

- The next lines of the section definition are floating point triplets that locate “keypoints” in the section's profile curve.

```
>>      0.000000  0.000000  0.000000
>>      0.291667  0.291667  0.012503
>>      0.629167  0.629167  0.120864
>>      1.000000  0.954167  0.012503
```

- The first value is the location of the keypoint along the bone. In StudioTools, this is represented by the circle icon. It is this value that determines where the bulging clusters are positioned along the length of the bone. A value of “0.0” represents the upper endpoint of the bone while a “1.0” represents the lower endpoint.
- The second and third values of a keypoint determine the overall shape of the bulge. The two values determine the direction and amount of bulge contributed at that keypoint. In StudioTools, these two values represent the (x, y) co-ordinate of the X icon.



If the first and second values of a keypoint are not the same, you get bulging occurring not only away from the bone, but also along the bone.

Restrictions

- A section must have at least two keypoint definitions, one at “bone” location 0.0 and one at “bone” 1.0.
- A keypoint definition's first two values must be greater than the previous definition's corresponding values, if any exist.
- A keypoint definition's first two values must be between 0.0 and 1.0, inclusive.

- A keypoint definition's third value must be between -0.5 and 0.5, inclusive.

Example

A sample bulge definition (from the default `bulge_types` file) is as follows:

```
>> bulge: "Generic" code: 5
>>   angle: 0
>>       0.000000 0.000000 0.008335
>>       0.500000 0.500000 0.250000
>>       1.000000 1.000000 0.000000
>>
>>   angle: 180
>>       0.000000 0.000000 0.000000
>>       1.000000 1.000000 0.000000
```



Sections in the same bulge definition do not have to have the same number of keypoint definitions.

Index

A

Alias camera depth map file
file format 7

Alias pix file
image 3
pixels 3
RGB information 3

Animation SDL
apply animation 11
file format 9

B

Bulge definition file
file format 22
initialization 23
keypoint definition 24
restrictions 24
section definition 23

C

camera
Alias camera depth map
file 7

character builder
Bulge definition file 22

F

file formats
platforms 2

I

image
Alias pix file 3

M

Matte file
file format 5

Maya IFF
file format 16

P

Particle file
diffusing gas particles 21
file format 20

platforms
file formats 2

R

RGB information
Alias pix file 3

T

TIFF
Maya IFF 16

