# TDSB

Jamie Doornbos, MAXIS, February 2000.

TDSB performs a number of importing and editing functions on Sims data. The command line options determine which functions are to be performed. It can handle more than one at a time, but in practice it only does one at a time. For some functions, an OMK directs the process, while others are just wired into the code, controlled in part by the command line options. The output of TDSB is always a resource or resource file. Thus the options always include a destination file name, usually the name of some object resource file (2 physical files), or an IFF file. Examples:

```
### generates the aquarium sprites
Tdsb aquarium -spec aquarium.omk -graphics
### generates one wall style (id 1, cutaway)
Tdsb -wallstyle 1c Wall1.tga walls.iff
```

## General OMK file format

OMK is a general format that looks somewhat like HTML. All relevant information is enclosed in a tag, in angle brackets, “<>”. Everything else is ignored. The tag parsing is not complicated. Thus a tag should not contain any ">" characters, even in quotes. Within a tag, name-value pairs are scanned for. Basically, this expression is used:

```
[token[=token]]*
```

A token is a consecutive sequence of non-whitespace characters, or a quote-bracketed sequence of any characters (except '>'). The first token in a tag is the name of the tag. If “=” appears after a token (a name), then the following token is the value of that name. If a new token is found instead of an “=”, then the name has no value, and the new token becomes the next name. Sample OMK text:

```
foo tag: <foo x=1 "test 1" y=2>
name-value pairs: foo, x=1, test 1, y=2
foo bar tag: <"foo bar" baz=5 "baz factor">
name-value pairs: foo bar, baz=5, baz factor
```

## Command line options for OMK functions

### *-spec [OMK-file]*

Causes given OMK file to be used in all functions that require it. (“Spec” is an older terminology used in the TDSB code.)

### *-graphics [TGA-directory]*

Causes the sprite and draw group generation code to be run. The TGA directory is used as a base path for all graphics files specified in the OMK, which is required. The OMK specifies all the necessary information to generate the object’s in-world graphics resources. This includes all sprite information such as symmetry, size and palette sharing, and a sprite assemblage for each draw group. Relevant OMK tags: sizes, sprite, drawgroup, item, wall style, headline.

### *-bmps [BMP-directory]*

Causes the BMP packaging code to be run. The BMP directory is used as a base path. The OMK is required and each "bmpfile" tag specifies a source file and a destination resource id. TDSB then copies the file into the BMP_ resource for each tag. Note this option removes all BMP_ resources before adding the new ones. Relevant OMK tag: bmpfile.

### *-slot*

Causes the slot creation code to be run. The OMK is required and the relevant tags are processed to determine the id of the target SLOT resource, and build a list of slot structures, which is then saved out to the resource. Relevant OMK tags: slots, slot.

### *-skills*

Causes skill tables to be created for use by animation primitive. The skill tables are just standard string resources, STR# 129-133. The OMK is required and specifies a number of skill tables to be created, as well as the strings to fill them in. Relevant tags: skill table, skill.

### *-sounds*

Causes FWAV resource to be created for use by the "play sound" primitive. The OMK is required. Each sound tag produces one FWAV resource. Relevant OMK tag: sound.

### *-catalog*

Causes OBJD resources to be modified with new catalog information. The OMK is required. The target OBJD is specified by guid, and is loaded, modified then saved. Relevant OMK tag: catalog.

## OMK tags for –graphics option

By far, the most complicated tags are used by the "-graphics" option. This function is the original reason for writing TDSB.

### *Sizes*

The "sizes" tag indicates the dimensions of the sprite rectangles in the input TGA and BMP files. Each "sizes" tag applies to all subsequent "sprite" tags, until another "sizes" tag is encountered. Sprites in the TGA/BMP files are assumed to be laid out horizontally, tops flush, with 4 rotations for each of the large, medium, and small zooms, in that order.

Example:

```
<sizes lheight=384 lwidth=136 mheight=192 mwidth=68 sheight=96
swidth=34
slopwidth=4>
```

#### lheight, lwidth, mwidth, mheight, swidth, sheight

These are the required dimensions of the sprite rectangles. L is for large, m for medium, s for small. An integer value is required.

#### Slop width, uniform slop width

The slop width is the number of pixels around the sprite rectangle that are not on the tile, but may contain valid pixels. For example, the Sims tiles are 128 pixels in width, but lwidth is 136 to allow sprites that bleed into adjacent tiles. The two values are mutually exclusive, and if neither value is present, 0 is assumed. The plain "slop width" value indicates the pixels for the large graphic, and the medium is one half of that and the small is one quarter. The "uniform slop width" value indicates a single number for all 3 sizes. The slop width is actually only needed for the automatic y origin calculation (see below).

#### Border

This value indicates the number of pixels around the edge of each sprite rectangle that should be ignored. For hand drawn art, border=1 allows a non-key colored boundary for easy editing. If the value is not present, 0 is used.

### Lxorigin, mxorigin, sxorigin

These are the x-coordinates from the upper left of the sprite rectangle of the center-of-tile. If the value is not present, then the horizontal center of the rectangle is used.

### Lyorigin, myorigin, syorigin

Same but for y coordinates. If the value is not present, it is computed from the bottom of the sprite rectangle by subtracting a half tile height and the slop width and border width for each size.

## *Sprite*

Each row of graphics in a TGA/BMP corresponds to a sprite list. The bounding boxes for the components are given by most recent "sizes" tag. The "sprite" tag specifies all the other information necessary to generate one SPR# or SPR2 resource, and to embed the sprite in a draw group.

Example:

```
<sprite name="aquarium" id=100 tgafile="aquarium.tga" position=0
paletteID=0 symmetry=radial-flip>
```

### Id, name

These specify the id and name of the resulting resource. Id is required. Name is optional and defaults to empty string.

### Position

This value specifies the starting row (numbered from 0) of the TGA/BMP to be compressed to generate the sprite resource. If the value is not present and the last "sprite" tag had the same value for "tgafile", it uses its last position +1. Otherwise, if the value is not present, it uses 0.

### Frames

This value specifies the number of rows of the TGA/BMP files that are to be compressed into the sprite resource. If the value is not present, 1 is assumed. Currently, there are only 2 uses for this. Firstly, in headline graphics, additional rows are used to animate the headlines. Secondly, for objects, if frames=2, the second row is used for diagonal graphics. Only the duck object uses it, which was not shipped in V1.

### Tgafile, bmpfile

These values specify the name of the file(s) to be used in generating a SPR2 or SPR#. "Tgafile" always overrides "bmpfile". One or the other is required. The full path is always determined by prepending the directory that was passed in with "-graphics". The exact conditions that determine which type of resource is generated are complicated, but there are basically 2 common cases, the first one being the case for almost all object sprites:

- The "tgafile" value is given and the TGA file is 32 bit and a BMP file of the same root name exists. In this case, an SPR2 is generated. The Z information is given by the alpha channel of the TGA, and the color information is given by the BMP. Also in this case, if a file of the same name with "Alpha.TGA" appended is present (minus extentsion), then the alpha channel of that 32bit TGA is used for alpha information. Otherwise the sprite will have no alpha. For example X.TGA, X.BMP, and XALPHA.TGA would generate a SPR2 from the pixel colors and palette in X.BMP, Z buffer from the 4$^{th}$ channel in X.TGA, and alpha information from the 4$^{th}$ channel of XALPHA.TGA.
- In the second case, the "bmpfile" value is given and there is no TGA by the same name and no file with "alpha.tga" appended. In this case, an SPR# is generated with no z and no alpha information. (The format does not even support alpha.)

### PaletteID

Specifies the resource id of the PALT resource to be generated. Each sprite generates a palette and saves it, so all sprites that specify the same palette id must have the same palette in the BMP file (guaranteed by the

artist). Otherwise, only the most recent sprite will have the right colors. If the value is not present, no palette is saved, the palette id for the sprite will be set to –1, and will end up using the global palette.

### Mask

This value specifies an optional BMP file to be applied as a mask before compression. The BMP file should have the same size as one row of the TGA file, and is applied to all 3 images (color, Z, and alpha) before compressing. For complicated renders that use symmetry heavily, like for the dynamic flood object, this allows pixel perfect tiling.

### Columns

This value specifies the number of columns per zoom level (typically rotations) in the TGA/BMP files. If the value is not present, 4 is assumed.

### Shrink

This option specifies how the shrink-wrapping is to be done. If no value is present or the value is "true", then the bounding box of the sprite is made as small as possible before compressing. If the value is "false", then the bounding box is not modified. If the value is "top", then the top coordinate of the bounding box is dropped as much as possible, leaving other edges unchanged.

### Reverse

If present, this option indicates that the sprites are ordered from small to large instead of large to small. Formerly used for floors.OMK.

### Symmetry

When assembling draw groups, the value of this name indicates which rotations are equivalent (possibly flipped). If the value is not present, "none" is assumed. The following values are accepted:

- None: no symmtery. All 4 rotations graphics are compressed for each size.
- Bi-lateral: The two rear-facing and two front-facing graphics are horizontally flipped versions of each other. Only the first 2 columns are used.
- Radial: All graphics are the same. Only the first column is used.
- Radial-flip: Every other graphic is a flipped version of the first. The first column is used, and the graphic is flipped in every other rotation.

### Pairing

This value indicates which multi-tile sprites are flipped versions of each other. For example, the left and right halves of a symmetric couch are flipped versions of each other. If the couch is 3 tiles, the middle piece would have no pairing.

### Rotation

This value allows the OMK to indicate which column corresponds to object rotation 0. Legal values are 0,2,4 and 6, the non-diagonal object directions. The individual sprites that are saved out are always ordered by rotation, with 0 being first. Typically, this is only needed if a render has the object facing the wrong direction in the first column. For example, a rotation of 2 will cause column 1 to be the $0^{th}$ sprite in the list. If the value is not present, 0 is used.

## *Drawgroup*

This tag begins the specification of a draw group resource. Each draw group corresponds to one graphical state of an object. Subsequent "item" tags (up until the next "drawgroup") add sprites to the group.

Example:

```
<drawgroup name="aquarium live frame 1" id=101>
```

### Id and name

These specify the id and name of the resulting resource. Id is required. Name is optional and defaults to empty string.

### Ordering

This tag specifies how the sprites are to be assembled. The possible values are:

- Fixed: The sprite items are drawn in the given order for all rotations. This is the default value.
- Obscured: The sprite items are drawn in the order given for front facing rotations, and in the reverse order for rear facing rotations.

"Obscured" is the best name I could think of for this. "Painters" might be a better name. It is sometimes necessary to use this ordering so that multiple alpha sprites are drawn correctly.

## *Item*

An item tag adds a new item to the current "drawgroup". At one time, a draw group could have more than just sprites, but currently all item tags are of type "sprite". Z-buffer and 3D people obviated the need for other item types, such as "slot".

Example:

```
<item sprite id=100 luminous=true>
```

### Sprite

Indicates that the item is a sprite. Must appear right after the "item" name. A warning issued if anything other than "sprite" appears in this position.

### Id

This value indicates the id of the sprite that is placed into the drawgroup. Ids are found in sprite tags.

### Alt, x, y

These values specify an offset to be added to the tile origin before drawing the sprite. Coordinates are in feet, and in canonical object reference frame. For example, to offset a sprite in the direction of the object, y is <0.

### Direction offset

This value allows a sprite within a draw group to be a rotated version of a sprite. For example, a direction offset could be used to implement a 2x2 table from a single sprite, where each tile is the same sprite, but rotated.

### Luminous

This value may be "true" or "false" If not present, "false" is assumed. A "true" value causes the sprite to be drawn with a brightened palette, and the ambient lighting disregarded.

## *Wall style*

This tag is used to embed a custom wall style (hole in the wall) in an object, allowing for plug-in windows. It uses the same code as the "-wallstyle" command line option, but with a different id mapping.

Example

```
<"wall style" name="storm window style" id=150
tgafile="window1style.tga"
      "cutaway tgafile"="window1cutawaystyle.tga">
```

### Id and name

These specify the id and name of the resulting resource. Id is required. Name is optional and defaults to empty string. For a window to use a plug-in window style, the customWallStyleID field of the object definition must be set.

### Tgafile

This value is the tga file (relative to directory provided in "-graphics" option) used in creating the style. The TGA is expected to be laid out as in the "-wallstyle" option.

### Cutaway tgafile

Same as "tgafile", but the one to be used for the cutaway graphic.

### *Headline*

This tag is deprecated. The code still accepts it, but all uses have been replaced with "sprite" tags. The newer sprite options such as "columns" made this possible.

## OMK tags for –slots option

### *Slots*

Used only by "-slots" option. Creates a new SLOT resource.

### Id and name

Specifies the name and id of the resulting resource. Id is required. Name is optional.

### *Slot*

This tag adds a new slot to the current SLOT resource (most recent "slots" tag).

### X, y, alt

Specifies the coordinates of the slot. TDSB does not care about the units, but stores them as floating point values. The game interprets them as Ftile and Alt units (Ftile units are 16 per tile, and Alt is a special discrete unit for specifying a vertical position) and in the canonical object reference frame. For example for a routing slot to be one tile in front of an object, y=-16.

### Name

The value is the name of the slot when viewed in edith.

### Type

Specifies the type of slot to be created. The value may be one of the following:

- **Contained object**

The slot is for another object to be placed on. For example, a table specifies a relative location for a plate.

Example:

```
<slot type="contained object" name="On top" x=0 y=0 height="low
table" size=50 height=table>
```

- **Sprite, headline**

The slot is for an overhead graphic. For example, the food has a sprite slot so flies can be animated above it without multiplying the necessary graphical states.

Example:

```
<slot type=headline name="flies slot" alt=1>
```

- **Routing**

The slot is used to determine routing destinations. For example, a counter can specify to choose any one of four adjacent tiles.

Example:

```
<slot name="route to back"       type=routing standing=1 south>
```

- **Handle**

The slot is for a person to hold the object. Not used since mostly obviated by design call for 3d accessories.

### Height

For contained object slots only, this value specifies that the slot is at one of the game's standard heights. The current legal values are:

- ground
- low table
- end table
- table
- counter
- non-standard (ha ha, not really standard, is it?)
- sitting – people only

If no value is given, the height is set to kUndefined, which has certain limitations.

## *Values specific to routing slots*

Two primitives in the Sims use routing slots: "goto routing slot" and "snap". (Snapping slots should be their own type, but they were unfortunately lumped early on.) Most values provide preferences as to where and how a person should end up.

### Standing, sitting

Routing only. These values are multipliers for the preference of the ending idle state of the person routing. For example, a table specifies that a person should stand when picking up a plate if possible, but fall back to sitting if necessary. The default is 0, so a routing slot that does not have one of these multipliers is degenerate. Another multiplier value is "ground" which was another idle state of people that got cut from the game design.

Example:

```
standing=10 sitting=9
```

### North, south, east, west, northeast, northwest, southeast, southwest

Orientation values. The presence of these values causes the destination scoring algorithm to consider the goals in that direction. North is in front of the object. If none of these are present in the tag, the route is considered to be to a single point relative to the object, and only the x, y coordinates are used.

### Min, max, optimal

These values specify the three basic parameters to the destination scoring algorithm. All are floating point and in tile units, and are used to generate a list of possible goal points in the area around an object. Goals at the optimal distance are scored highest, and goals outside of the min and max distances are ignored. All 3 values default to 1.0.

## Gradient

Specifies how steeply the scores deteriorate as the distance from optimal increases. Default is 3.0. Typically objects only lower this value to indicate that the distance is less important, like when watching television, where sitting is far more important than distance.

## Random

The presence of this value causes the destination scoring algorithm to add in a random value for each goal, which is good for a little chaos. Note: gradient mitigates the effect of this, so there may still be a preference for distance.

## Resolution

Specifies how far apart the chosen goals are. The value is in floating point and in tile units. The default is 1.0. Only a handful of routes in the game vary the value. Sometimes it is sneakily used to collapse the goal finding into a specific single distance. For example, a lamp specifies a min, max, optimal, and resolution of 0.57 to force the person to any of the 8 adjacent positions at that distance.

## Facing not required, opposite facing

These two mutually exclusive values allow the person's final facing direction to specified. The default is to face the object. "Facing not required" causes the person to not turn at all after the route. "Opposite facing" causes the person to face away from the object after the route.

## Ignore rooms

This value signifies to the router that the route may cross rooms. By default, a goal in another room is ruled out. It is mainly used by doors to have a person walk through.

## Absolute

Causes the router not to transform the goals to the coordinate frame of the destination object. This is useful for assuring that a person is not facing a diagonal direction at the end of a route to another person. As far as I know it is only used by the attack interaction since the smoke cloud must overlay the two people and cannot face diagonally.

## Alt not required

Causes the router to ignore altitude when considering goals. By default, a goal must be on a flat tile and have the same altitude as the tile of the destination.

## Average object location

Causes the goal finding code to use the average location of a multi-tile object when finding goals. Added very late to fix a couple of tricky routes with dressers and stereos.

## Target

For the snapping variety of routing slots, this value gives the slot number of the location to snap to. Slots are numbered by their order in the OMK file, and independently of the slot type. For example, if an object specifies one routing slot and two containment slots, the routing slot is always #0, and the containment slots will always be #0 and #1. For example, a chair uses a "snap to target slot" to instruct a person to relocate to be contained in the chair before running the sit down animation. By default, the snap is to the location relative to the object given by x, y, and alt. This value overrides those coordinates.

Example:

```
<slot type="routing" target=0 name = "snap to sitting">
```

# Other OMK tags

For other options that require an OMK file.

## *Bmpfile*

The "bmpfile" tag is used only by the "-bmps" option. It creates a new BMP_ resource.

### Id

Required id of the resulting BMP_ resource (it is named after the file).

### Filename

Required filename to be put into the resource. The file is found relative to the directory given by the "-bmps" option.

### Language

A late breaking addition to the resource manager allowed a resource per language in a file. This value specifies which 8-bit integer language code the BMP_ should be attached to. This is transparent to the game. The acceptable values are single byte and client dependent. See function GetCurrentLanguageCode() in the sims.

## *Skill table*

Begins the skill table definition for an object. Formerly it specified the id an name of the skill table, but now an object file is only allowed to have only 4 skill tables with hard-wried id numbers. TDSB still requires the tag, but ignores the values within.

## *Skill*

This tag specifies an entry in one of an object file's 4 skill tables.

### Index

This is index of the skill in the table. 0 is reserved. Typically values are consecutive.

### Name

The value is the name of the skill. For example, name=a2o-lamp-turnon.

### Type

Specifies which skill table the skill is supposed to go into. These correspond to the skill types in the animation data-base. The accepted values and corresponding resource ids are:

- a2a, id 129
- a2o, also id 129
- c2c, id 130
- c2o, also id 130
- a2c, id 131
- c2a, id 132

## *Sound*

### Id

The id of the resulting FWAV resource.

### Name

The name of the sound event. For example, "lamp_click".

## *Catalog*

Tag specifies catalog information. The values within are largely exported from the data base. Near the end of the project, the names and descriptions were moved to Slang. The code in TDSB that sets up catalog names and descriptions is de-activated.

### GUID

This value is used to find which OBJD in the object file the tag is to be applied to. It is not necessary if there is only one OBJD in the file. For multi-tile objects, the GUID should be the one of the master, catalog stub object (this is also the GUID required by the data base).

### Id

Determines the value to write into the "catalog id" field of the OBJD. If no value is found, the one already in the OBJD is used. The normal path is for the object programmer to assign the id in Edith, and for the object DB not to export it.

## *Optional db export fields*

All of the following values are optional, and typically exported from The Sims object data base. Each value writes to a field in the OBJD. If a value is not found, the existing current value in the OBJD is used.

### Cost

Writes to the "price" field. Units are Simoleans, and number is charged when the object is bought. Also initializes the kCurrentValue of an instance.

### Initial depreciation

Writes to the "initialDepreciation" field. Value ends up getting subtracted from the current value at the end of the first day after an object is bought.

### Daily depreciation

Writes to the "dailyDepreciation" field. Value is subtracted from the current value at the end of every day# >1 after an object is bought.

### Depreciation limit

Writes to the "depreciationLimit" field. The value is the minimum that an object's current value can attain.

### Standard depreciation

Writes to the "selfDepreciating" field (inverted). If standard depreciation in non-zero, the game computes and tracks the current value of the object. Otherwise, the current value is not touched and may be modified by the object's main simulation.

### Room sort

Writes to the "roomFlags" field. The value determines where the object appears in the catalog when sorted by room. An object may appear in multiple categories, but this was designed out, so TDSB sets the field to only one flag. The accepted values are:

- Bathroom
- Bedroom
- Dining Room
- Family Room

- Kitchen
- Study
- Outside
- Miscellaneous

### Function sort

Writes to the "functionFlags" field. Determines where the object appears in the catalog when sorted by function. The following accepted values:

- Appliances
- Decorative
- Electronics
- General
- Plumbing
- Seating
- Surfaces
- Lighting

### Hunger

Writes to the "ratingHunger" field. The value shows up in the catalog popup in the object's ratings.

### Comfort

Writes to the "ratingComfort" field. The value shows up in the catalog popup in the object's ratings.

### Hygiene

Writes to the "ratingHygiene" field. The value shows up in the catalog popup in the object's ratings.

### Bladder

Writes to the "ratingBladder" field. The value shows up in the catalog popup in the object's ratings.

### Energy

Writes to the "ratingEnergy" field. The value shows up in the catalog popup in the object's ratings.

### Fun

Writes to the "ratingFun" field. The value shows up in the catalog popup in the object's ratings.

### Room

Writes to the "ratingRoom" field. The value shows up in the catalog popup in the object's ratings.

### Cook

Boolean value ("true" or "false" expected). Writes to the "ratingSkillFlags" field. If true, the cook flag is set. The flag ends up causing the skill to appear in the catalog popup.

### Mechanical

Boolean value writes to the "ratingSkillFlags" field. If true, the mechanical flag is set. The flag ends up causing the skill to appear in the catalog popup.

### Logic

Boolean value writes to the "ratingSkillFlags" field. If true, the logic flag is set. The flag ends up causing the skill to appear in the catalog popup.

### Body

Boolean value writes to the "ratingSkillFlags" field. If true, the body flag is set. The flag ends up causing the skill to appear in the catalog popup.

### Creativity

Boolean value writes to the "ratingSkillFlags" field. If true, the creativity flag is set. The flag ends up causing the skill to appear in the catalog popup.

### Charisma

Boolean value writes to the "ratingSkillFlags" field. If true, the charisma flag is set. The flag ends up causing the skill to appear in the catalog popup.

### Study

Boolean value writes to the "ratingSkillFlags" field. If true, the study flag is set. The flag ends up causing the skill to appear in the catalog popup.

## Other command line options

Other functions of TDSB do not require an OMK file but get all their information from the command line parameters.

### *-tmpl [source file]*

Copies the objects from the source file into the destination file, generating a new GUID for each OBJD resource found. Also, new OBJD resources are named after the given source file, unless one of the following two parameters is used.

### *-prefix [object name prefix]*

When using the –tmpl option, causes the newly copied objects to have the given prefix attached to the name. This is important when copying files with more than one object. Otherwise all the objects have the same name and cannot be easily found in the object browser in edith.

### *-retain*

When using the –tmpl option, causes the newly copied objects to keep the same names as in the source file.

### *-famhist [output file]*

Treats the destination file as a FAM file and outputs the history of the family within. The code it uses is the same as the code used by the "history" cheat in the game.

### *-wall [tga file] [bmp file] [id and style]*

Creates a wall style sprite in the destination file. This option probably does not work anymore, since –wallstyle appears to be used by the script "makestyles.ksh".

## Wall and floor command line options

These options are done a little differently. In general, the game expects the resources generated by these options to be present (in walls.iff or floors.iff). Also, the necessary calls to TDSB and all the parameters are all stored in Korn shell script files (KSH) in the same directory as the graphics files. For example, all the game's wall styles are generated by running the script $/tdscontent/sprites/walls/styles/makestyles.ksh. (Note this requires some envirnment variables to be set. See separate CB script document.)

### *-wallstyle [style identifier] [tga file] [iff file]*

Creates a wall style sprite set in the IFF file from the TGA file. The style identifier consists of the id of the style, defined in wallstyles.h, with an option letter "c" appended to denote cutaway. Wall styles correspond to the different background graphics for walls. Windows can embed their own styles, but door styles are stored globally and are created using this option. Each fence type is a style. The construction style is used when dragging out a wall. There are also some special styles used to do the cutaway transitions. Each style has a style id, and two sprite sets, one for the normal, up, state and one for the cutaway state. Each call of tdsb generates one sprite set.
The TGA file is black and white and is expected to have a particular layout. In sourcesafe, see $/tdscontent/sprites/walls/styles for examples.
The resulting sprite resources generated have id's that are a function of the style identifier.

### *-wallpattern [pattern id] [BMP file] [iff file]*

Creates a wall pattern sprite set in the IFF file from the BMP file. Wall patterns correspond to the items in the wall paper menu in the game. The BMP file is expected to have a particular layout. In sourcesafe, see $/tdscontent/sprites/walls/patterns for examples. The sprite (and palette) resources generated are a function of the pattern id, and are chosen to be distinct from other wall resources.

### *-thickwalls [zoom] [bmp file] [iff file]*

### *-thickcutaway [zoom] [bmp file] [iff file]*

Generates a sprite set in the IFF file for the sprites used in drawing the spoofed "thick" pieces on top of and next to normal walls. The BMP file is expected to have a particular layout. In sourcesafe, see $/tdscontent/sprites/walls/thick for examples. The zoom specifies which game zoom level the sprites are for. 1 for small, 2 for medium, 3 for large. The ids of the resulting sprite resources are a function of the zoom, and are chosen to be distinct from other wall resources.

### *-floor [floor id] [bmp file] [iff file]*

Generates a floor sprite set in the IFF file for the given floor id from the BMP file. For each floor id, there is an item in the floors menu in the game. The BMP file is expected to have a particular layout. In sourcesafe, see $/tdscontent/sprites/floors/ for examples. The ids of the resulting sprite resources are chosen based on the floor id, and to be distinct from other floor resources.

### *-floorshadow [shadow id] [bmp file] [iff file]*

Generates a floor shadow sprite set in the IFF file for the given id from the BMP file. The BMP file is expected to have a particular layout. In sourcesafe, see $/tdscontent/sprites/floors/shadows for examples. Multi-tile object shadows are algorithmically generated, and may not be specialized. Single tile shadow id's are special and must currently be allocated in the graphics code. A single-tile object refers to the shadow id directly from the shadow field in the object definition. Currently, there are only two shadows, round and square. The ids of the resulting sprite resources are chosen based on the shadow id and to be distinct from other floor resources.

### *-pooltile [tile id] [bmp file] [iff file]*

Generates a pool tile sprite set from the BMP file in the IFF file for the given tile id. The BMP file is expected to have a particular layout (same as –floor BMP files). The ids of the resulting sprite resources are chosen based on the pool tile id and to be distinct from other floor resources. The different sprites are used by the game to compose pool images based on the layout of the adjacent pool tiles.

### *-watertile [tile id] [iff file]*

Generates a water tile sprite set in the IFF file of the given id. Tdsb expects a BMP file and a TGA file to be in the current directory, and both should be named after the id (two digits, e.g. 01.BMP and 01.TGA). The TGA provides alpha information and the BMP provides palette and color information. The ids of the

generated sprites are chosen based on the tile id and to be distinct from other floor resources. The BMP and TGA files are expected to have the same, particular layout. In sourcesafe, see $/tdscontent/sprites/water for examples. The game uses the resulting sprites to render the water tiles on lots that are next to the stream.