
BUILD–DOCUMENTATION


.□□□□□□.□.□3□.□□□□□□.□□□.□□□.□□□.□□□□.□□□.□□□□□□□.□□□.□□□□.□□□□□□.□□□.□□□.□□□□□.□□□□□□□.□□□□□□□□□□~.□□□.□□□□□.□□□□□□□.□□□□□□□□□□.□□□.□□□□.□□□□□□□□□□□□□□□.□□□.□□□□□□□□.□□.□□.~.□□.□□□.□□□.□□□□.□□□□.□□□□□□□□□□.□□□□□.□□□.□□□□□.□□□□□□□□□□.□□.□□□□□.□□□~.□□□□□□.□□.□□□.□□□.□□.□□□□□□□□□.□□□□□.□□□.□□□□□.□□□□□□.□.□□□.□□□□□.□□□□□□□□□□□□□□□~

# Shadow Warrior Design Techniques

## Copyright & Legal Stuff

# *Table of Contents*

# 1) Introduction

This Design Techniques document was written by Steffen „*Duke Addict*“ Itterheim and Keith Schuler. Thanks to Keith Schuler who provided a set of brief explanations on the real complex Shadow Warrior Design Techniques like „true“ room-over-room situations and Sector Objects.

In this document you will learn how Shadow Warrior can be told when and how it should perform a special effect and you will be given examples on designing doors or underwater areas. Once you’re familiar with most of the basic effects you might want to learn how to create „true“ room-over-room areas or Sector Objects, like a driveable tank. This is for the advanced user only, so never mind if you don’t understand it at first. You will learn as you design.

# 2) Advanced Shadow Warrior Level Design

by Steffen Itterheim

Designing good levels for Shadow Warrior means taking use of Shadow Warrior’s special effects. With these you can create diverse effects such as exploding walls, driveable tanks, true room-over-room situations or just simple doors. Shadow Warrior provides the Level Designer with great flexibility, but this also means complexity and that the designer has to tell the game when to do certain special effects and how to do them, so they behave as intended.

## *Advanced SW Level Design Contents*

## 2.I) *How it Works*

To tell a game how to do a special effect the programmer writes one or more functions (basically a set of commands and variables in a certain order using certain algorithms) in a programming language. It wouldn't be a good idea to program a specific function for each door, lift, teleporter or all the other game effects. So the programmer writes a basic function which takes parameters, telling the function how to behave in certain situations. Take the doors that move up and down as an example. The function itself that makes the door behave like a door basically only moves the floor or ceiling of the sector up and down, depending on how the level designer sets it up. You may already figure that with such a function you're not limited to doors, since you can move the floor also. This function could be used for elevators as well. Most of Shadow Warrior's special effects are created by passing parameters to a function to make it behave correctly. This means the designer has to pass the function parameters which control the speed of the door, the direction it goes to (up or down), whether it can be triggered manually or only by a switch, etc. Of course the function also has to know on which sector (floor/ceiling) it has to operate and optionally which other effects to trigger when it reaches its destination, or its original position again. This triggering effect is used to link the door function with another function which plays a specific sound whenever the door opens or closes, for example.

## 2.II) *The Shadow Warrior Tag System*

In Shadow Warrior, you do not have to write your own functions, the programmers already did that for you and you can take use of the predefined set of functions. You do not need to be a programmer to design levels for Shadow Warrior. Since you do not have a scripting language of some sort available in Shadow Warrior, you can only pass parameters to built-in functions.

### 2.II.1) What are „Tags" and „Flags"?

Passing function parameters is accomplished by using a set of „tags" and „flags". Every Wall, Sector and Sprite has its own set of tag values and flags. Sprites, for example, can be given up to 15 different tag values plus another 11 boolean flags. Boolean means there are only two possible values for this variable - true or false, yes or no - represented by two numbers: 0 or 1. Although walls and sectors can be assigned most of these tags there is no wall or sector function that takes use of more than the two standard tags (Hitag and Lotag). Consider all these tags and boolean flags as slots for variables that are passed to a certain function in order to operate correctly when the game runs. Even the function itself is referenced by a tag value. Many functions only require a few tags/flags and many are optional. However, it is easily accomplished to forget to set a specific tag/flag, or to give it an incorrect value. In that case the function will either behave erradically or not at all, or the game crashes. There is no error checking so the designer is responsible for providing the functions with all the required variables as well as to make sure that they are valid. Passing parameters using the tags and flags is not the only way to tell a function what to do. For example the sprite that makes a sector behave like a door is to be placed in the sector that is to become a door, and more often than not a door moves the ceiling upwards to open. In that case the door-sprite will have to be flipped in order to tell the function to affect the ceiling rather than the floor. The door-sprite, as I named it, is in fact a special, multi-purpose, multi-functional sprite, called the ST1 sprite. Most of the effects seen in Shadow Warrior can be done with that sprite, so let's take a closer

look at this „special sprite“. But first let me tell you more about the most important tag values and flags. For the moment, know the ST1 as *the* sprite used to create most of the game effects.

### 2.II.2) The HiTag

Every Wall, Sector and Sprite can be assigned a Hitag value. The Hitag is the first of all the tags and thus also referred to as TAG1. The Hitag can hold values in the range from -32768 to 32767.

The Hitag is mainly used to assign a function to an ST1 sprite. There are over 100 functions available for the ST1 sprite. Give the ST1 sprite a specific and valid Hitag value (see SWREF.DOC, ST1 Sprite Reference) and its parameters (tags, flags as well as its position, height, etc.) will be passed to this specific internal function. For example a Hitag of 92 will turn the ST1 sprite into a SECT_VATOR function sprite which is used to create doors and lifts. Thus, when the game runs, all the parameters of this ST1 sprite are passed to the SECT_VATOR function and will be processed by the game. Whether it works or not depends on the correctness of the ST1 parameters. To assign a Hitag to a sprite point at the sprite and press ALT+H when in 2D mode, or ‘+H when in 3D mode and enter the desired value. Alternatively you can assign a Hitag by pressing ‘+1 in both modes, this will prompt for a value for TAG1 (the Hitag).

For Walls and Sectors the Hitag has multiple purposes, sometimes it is used as a simple „variable slot“ providing the Wall or Sector function with a parameter, it might not be used at all or it is used as a „match“ tag. See the paragraph on „match“ tags below. To assign a Hitag to a Wall or Sector point at the wall and press ALT+H for Wall Hitag, or H for Sector Hitag when in 2D mode, or ‘+H when in 3D mode and enter the desired value. Alternatively you can assign a Hitag by pressing ‘+1 in 3D mode, this will prompt for a value for TAG1 (the Hitag). This shortcut does not work very well for walls and sectors in 2D mode, it is only intended to affect sprites.

### 2.II.3) The LoTag

Every Wall, Sector and Sprite can also be assigned a Lotag value. The Lotag is the second of all tags and thus also referred to as TAG2. The Lotag can hold values in the range from -32768 to 32767.

The Lotag is mainly used to assign a function to a Wall or Sector. When assigning a Lotag „function number“ to a wall or sector it will not show the descriptive name of the function in 2D mode, as opposed to the ST1 sprite. It works the same way though. To assign a Lotag to a Wall or Sector point at the wall and press Alt+T for Wall Lotag, or T for the Sector Lotag when in 2D mode, or press ‘+T when in 3D mode. Alternatively you can assign a Lottag by pressing ‘+2 in 3D mode, this will prompt for a value for TAG2 (the Lotag). This shortcut will not work properly for walls and sectors in 2D mode.

For sprites the Lotag is often used as a simple parameter „variable slot“ or quite often as a „match“ tag. See the „match“ tag paragraph below. To assign a Lotag to a sprite point at the sprite and press ALT+T when in 2D mode, or ‘+T when in 3D mode and enter the desired value. Alternatively you can assign a Lotag by pressing ‘+2 in both modes, this will prompt for a value for TAG2 (the Lotag).

## 2.II.4) The Sprite's „Angle“ Tag

The angle of a sprite, meaning the direction it is facing, is not measured in degrees. Build calculates angles as values from 0 to 2048, meaning a turn of 90 degrees would be 512 degrees in Build measurement. Some ST1 functions use the angle respectively the TAG4 of a sprite for their calculations. Most functions, like teleporters, use the angle as that what it is, a tag that can normally hold values in the range from 0 to 2048, although its true range goes from -32768 to 32767, and will make the player or an object facing or heading the direction indicated by the ST1 angle value. There are some functions though, like the SOUND_SPOT ST1 (Hitag 134), that rely on an exact angle value or at least one that is within a certain range. If, for example, the angle of a SOUND_SPOT ST1 is somewhere above 614 the game will crash because the SOUND_SPOT function will not find a sound referenced as number 615 or above (see SWREF.DOC, Digital Sound Reference). This is important to know because if you accidentally run over an ST1 sprite and change its angle with the , or . (comma or period) keys it will change by 128 which is often a big enough change to make the ST1 function behave erradically or even crash the game.

## 2.II.5) The BOOL1 Flag

This is a special flag, if it is set to „1“ on any sprite then this sprite will behave as if it were an ST1 sprite. This is generally only used to create breakable sprites.

## 2.II.6) The BOOL11 Flag

By default, all sprites will move with the height of the floor they're on. That means, if there's an item, decorative or any other sprite on a lift and the lift goes down or up the sprite will go down or up accordingly. If you set the BOOL11 flag to „1“ on any sprite it will remain at its initial position and not ride up or down with the floor. This is used very rarely though, one of the few purposes is to make a wall sprite remain stationary when it is on a wall in a lift shaft. Otherwise it might move up or down along the wall.

## 2.II.7) The „Match“ Tag

Now, what is this mysterious „match“ tag? It is *NOT* another tag value besides the 15 existing ones. A „match“ tag is a synonym for any tag value that is used to link one object with another. The Lotag of many ST1 functions is used as a „match“ tag, for example the FIREBALL_TRAP ST1 (Hitag 43) uses a Lotag as a „match“ tag.

What does a „match“ tag do? Consider the idea behind setting up a fireball trap. You don't want to shoot the trap the whole time, rather it should be triggered when the player comes in dangerous vicinity of the trap and then, as the player comes closer to the fireball trap, he/she stands facing a fireball all of a sudden.Ouch! So there must be something telling the fireball trap when to shoot fireballs. This is usually accomplished by using a trigger, and most often for these kind of traps it's a floor trigger. That means, when the player steps on a specific trigger sector this trigger will call the fireball trap and then the trap will shoot.

And how do the trigger and the trap know of each other? This is where the „match“ tag comes into play. Both the trigger and the fireball trap have a „match“ tag, whether

this is the Lotag, Hitag or any other tag doesn't matter as long as the function uses this tag as a „match" tag (see SWREF.DOC). To link two „objects" together you would give both of them a *COMMON* and *UNIQUE* tag value. It would be more convenient to use „match" tags like „THIS_SECTOR_TRIGGERS_FIREBALL_TRAP" but Build only accepts numeric values. Due to this reason you should keep track of your match tags from the beginning on. Keep a list of tags and note in which place they're used and which „objects" they connect. This helps a lot to avoid unwanted double uses of „match" tags, like if you open a door it might trigger off a major explosion in some other place of the map and the player will never get to see it. Bugs like these are sometimes hard to resolve, so it's better to avoid them from the beginning. A good way of avoiding problems is to make use of the full range of the tag values. For example, for doors you could use tag values from 1-200, for explosions use the 500 series, for traps use the 600 series and so on. This greatly helps to keep your „match" tags sorted.

By the way, Build keeps track of the „match" tags already used in your map and whenever you need a new number, point at a sprite in 2D mode and press the F5 key. Build will show you the next unused „match" tag number, to the right of the „Build - by Ken Silverman" text in the status bar.

### 2.II.8) All the Tags, all lined up...

So we have Walls, Sectors and Sprites and each of them use numerous tags and flags. Confusing? At first, but not when you take a closer look. For one, Walls and Sectors only take use of the Hitag and the Lotag (TAG1 and TAG2 respectively) so you can assume as if there were only Hitags and Lotags available for Walls and Sectors.

| Tag No. | Name | Range of Values | 2D Hotkey | 3D Hotkey |
|---|---|---|---|---|
| **TAG1** | Hitag | -32768 to 32767 | **‘+1** or H (sector) or Alt+H (sprite/wall) | **‘+1** or ‘+H |
| **TAG2** | Lotag | -32768 to 32767 | **‘+2** or T (sector) or Alt+T (sprite/wall) | **‘+2** or ‘+T |
| **TAG3** | clipdist | -128 to 127 | **‘+3** | **‘+3** |
| **TAG4** | Angle | -32768 to 32767 | **‘+4** or , and . (comma and period) | **‘+4** or , and . (comma and period) |
| **TAG5** | xvel | -32768 to 32767 | **‘+5** | **‘+5** |
| **TAG6** | yvel | -32768 to 32767 | **‘+6** | **‘+6** |
| **TAG7** | zvel 1 | -128 to 127 | **‘+7** | **‘+7** |
| **TAG8** | zvel 2 | -128 to 127 | **‘+8** | **‘+8** |
| **TAG9** | owner 1 | -128 to 127 | **‘+9** | **‘+9** |
| **TAG10** | owner 2 | -128 to 127 | **‘+0** | **‘+0** |
| **TAG11** | Shade | -128 to 127 | **Shift+‘+1** | **Shift+‘+1** or ‘+S |
| **TAG12** | Palette | -128 to 127 | **Shift+‘+2** or P (floor and ceiling) | **Shift+‘+2** or Alt+P |
| **TAG13** | x/y-offset | -32768 to 32767 | **Shift+‘+3** | **Shift+‘+3** |
| **TAG14** | x/y-repeat | -32768 to 32767 | **Shift+‘+4** | **Shift+‘+4** |
| **TAG15** | Z | -32768 to 32767 | **Shift+‘+5** | **Shift+‘+5** |
| **BOOL1** | - | 0, 1 (true or false) | **;+1** | **;+1** |
| **BOOL2** | - | 0, 1 (true or false) | **;+2** | **;+2** |
| **BOOL3** | - | 0, 1 (true or false) | **;+3** | **;+3** |
| **BOOL4** | - | 0, 1 (true or false) | **;+4** | **;+4** |
| **BOOL5** | - | 0, 1 (true or false) | **;+5** | **;+5** |
| **BOOL6** | - | 0, 1 (true or false) | **;+6** | **;+6** |
| **BOOL7** | - | 0, 1 (true or false) | **;+7** | **;+7** |
| **BOOL8** | - | 0, 1 (true or false) | **;+8** | **;+8** |
| **BOOL9** | - | 0, 1 (true or false) | **;+9** | **;+9** |
| **BOOL10** | - | 0, 1 (true or false) | **;+0** | **;+0** |
| **BOOL11** | - | 0, 1 (true or false) | **Shift+;+1** | **Shift+;+1** |

Sprites instead take use of 15 tag values, these are slots for variables which are passed to the designated function if the sprite is a ST1 sprite with a proper Hitag (the Hitag determines the kind of function called by a ST1 sprite). Sprites also have 11 boolean flags which can hold only two states, yes or no, true or false, represented by the numeric values 0 and 1. Most of these tags/flags are only important when used on a ST1 sprite. To make this a bit more complicated, some of these variables serve multiple purposes, for one they are parameters passed to the ST1 function while at the same time they determine the sprite’s angle, palette or shade. So be careful not to change these attributes with the standard angle, palette or shade „modification keys“ when they’re used for the ST1 sprite’s function, use the designated TAG keys instead. See the table below for a summary of sprite tags and flags as well as all the hotkeys that change these tags and flags. The designated TAG keys are printed in bold letters. See also the Key Command Reference in SWREF.DOC.

## 2.III) *The ST1 Sprite*

The ST1 sprite uses tile number 2307. If you place a sprite and assign this texture to it it will become an ST1 sprite, which is called the „Sector Effector“, as it effects sectors and makes

them move, rotate or whatever. The ST1 sprite is a special sprite in that it will never appear in the game itself, it can only be seen in the Build editor.

To place an ST1 sprite in your map simply place a Sprite by pressing the **S** key, point at it in 3D mode and press the **V** key. If this is the first ST1 sprite in the map press **V** again to see the full list of textures. Then go to tile number 2307 (press **G** and type in „2307“) and press **Return** to apply this texture to the sprite. Go back to 2D mode and take a look at it. It now reads „S:2,xSECT_SINK,“. The „S:2“ means that it is a sprite that appears in skill mode 2 and below. Skill mode 3 is the hardest skill and skill mode 4 means that this sprite will not appear in Single Player games, only in WangBang (Deathmatch) games. „xSECT_SINK“ means that this ST1 sprite currently has the SECT_SINK function assigned to it, which is Hitag 0. The small „x“ in front of the function's name is an indicator that this is a function name as opposed to, say the „NUKE_BOMB“ sprite's name, to tell apart ST1 function sprites from other sprites that have names in 2D mode. Note that the „x“ will not appear on sprites that have their BOOL1 flag set to „1“ in order to make them behave like an ST1 sprite.

Now, to assign the desired function to the ST1 sprite give it an appropriate Hitag value, by pointing at it in 2D mode and pressing either Alt+H or ‘+1. In 3D mode use the ‘+H or ‘+1 key. See the Sector Effector (ST1) Sprite Reference in SWREF.DOC for a list of functions as well as their parameters that can or must be passed to the function.

The next paragraphs contain some example uses of the ST1 sprite, and the first example uses the SECT_SINK function, so leave the Hitag at 0 for now. To begin with the examples all you need is a single rectangular sector, we will expand on that as we go on. All the examples can be found in the TUTORIAL.MAP.

### 2.III.1) ST1 Example #1: Simple Water

This example will use an ST1 sprite with the SECT_SINK function (Hitag 0). We need a single rectangle sector and in this sector is a child sector in which we want to place a SECT_SINK ST1 sprite to make the player sink into it as if it were water. Lower the floor of the child sector which is to contain the ST1 sprite a bit and assign a water texture to it, tile number 780 is fine, to make it look like a pool of water. Press **S** while the mouse cursor is over the floor of the water pool to insert a sprite there. Change the texture of the sprite to tile number 2307 to turn it into a ST1 sprite. Go to 2D mode and take a look at the sprite. It should read „S:2,xSECT_SINK,“. Finally, assign the „sink amount“ to the SECT_SINK ST1 by giving it an appropriate Lotag value, 40 is a good sink value. Load the map into Shadow Warrior and notice that not only Lo Wang sinks into the water as if he were swimming, the water also makes a splash sound and animation. This is done automatically by the SECT_SINK ST1 in conjunction with the water texture (#780).

You might want to add a water current to the water pool sector. Add another ST1 sprite to the water pool sector and give it a Hitag of 3 (SECT_CURRENT) and a Lotag of 128 in order to create a slow current. Change this SECT_CURRENT ST1 sprite's angle (TAG4) to make it point in the direction you want the current to flow.

### 2.III.2) ST1 Example #2: Diveable Water

The water pool behaves like water already but you can't dive in it. To create a diveable water area take the existing water pool and copy it to the outside of the map, or create a new sector with congruent shape and size. Copying is easier though because it will also copy the sprites in the sector. Add yet another ST1 sprite to this water pool sector and give it a Hitag of 7 (SECT_DIVE_AREA) before copying. This SECT_DIVE_AREA function needs a match tag that links it with a SECT_UNDERWATER ST1 sprite, and the match tag is the Lotag of the SECT_DIVE_AREA ST1. I have chosen a Lotag of 300 as match tag because it can't be confused with one of the ST1 function numbers. Now copy the whole water sector. Hold down the **Right Alt** key and select the water sector, and only the water sector, so it flashes green. Move your mouse cursor in the green area and hold down the left mouse button, then press the **Insert** key and while still holding down the mouse button move the sector to a new place, completely outside the existing sectors. The walls will turn white, and that's just what we want. Press **Right Alt** again to deselect the sector. You now have an exact copy of the water sector. Point at the SECT_DIVE_AREA sprite and give it a Hitag of 8 which will turn it into a SECT_UNDERWATER ST1 sprite. Leave the Lotag as it is, since the Lotags of the SECT_UNDERWATER and SECT_DIVE_AREA sprites connect these two together so that Shadow Warrior always knows where to go when the player dives or surfaces in that area. Do not move the SECT_UNDERWATER or SECT_DIVE_AREA sprites, unless you place them in the same relative position in their respective sectors. Also, do not place them on a wall or vertex, although this might be a good idea to be sure that both sprites are in the same relative position it can cause problems. Before

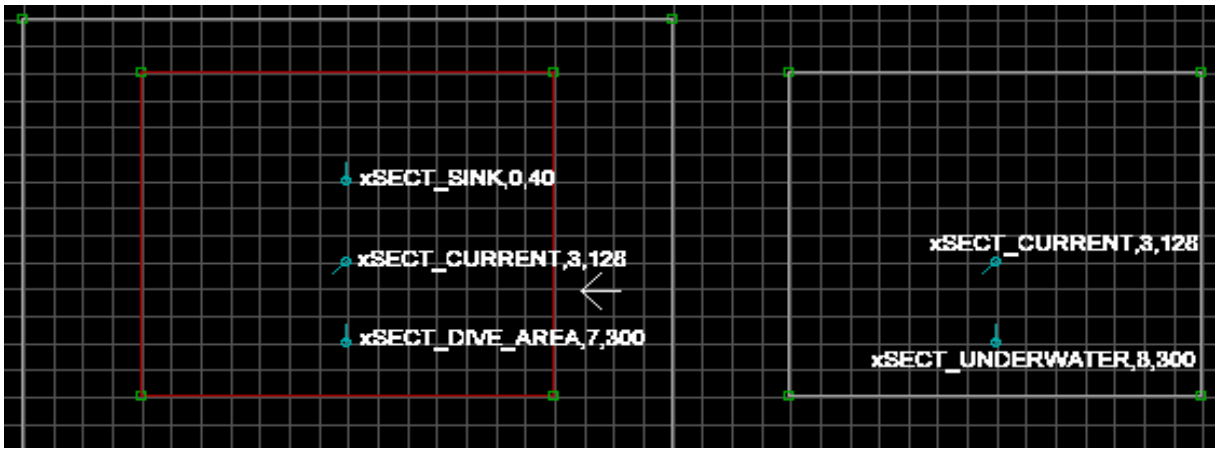


**Figure 1: Above (left) and underwater (right) sectors.**

you go to 3D mode delete the SECT_SINK sprite in the new water pool because it is not needed there. In 3D mode change the ceiling texture to the water texture and replace the water texture on the floor with a „solid" one. You're done. View your first diveable water area in Shadow Warrior and enjoy the fresh water.

If you're a perfectionist like me, change the palette of all underwater textures to 9. Remember that you can use copy & paste to quickly copy textures as well as their attributes, including the palette value. This will give the underwater area a blue-ish look. It might not be easily noticeable at first but such details are well appreciated by the player. Always keep that in mind when designing levels.

## 2.III.3) ST1 Example #3: Teleporter

A Teleporter is probably the best and easiest example of how two match tags work together. In TUTORIAL.MAP two teleporters were added. The glimmering textures are just visual enhancement, the teleporters work without them. Create a small sector

inside the existing room and another such sector in a different area of the map. Then place one ST1 sprite in each sector and give each of them a Hitag of 84 so they become WARP_TELEPORTER ST1s. Now adjust their lotags and give them a common and unique lotag value, this is their match tag. I have used 300 as a match tag before so I have chosen 301 as match tag for the WARP_TELEPORTER sprites. Whenever the player steps into one of the two sectors he will be teleported to the other sector and face the same direction as the respective WARP_TELEPORTER sprite. Note that for teleporting back the player has to step out of the teleporter and back in. Well, that's quite easy, isn't it? So let's do something more complex, like a door.

## 2.III.4) ST1 Example #4: Door

In the room with the second teleporter I have prepared a door. It works but it is not yet complete and also shows two common design flaws. This is for you to work on! I have set up this door by placing a SECT_VATOR ST1 (Hitag 92) in the door sector as seen in figure 2. To the south of the door sector I have added a door frame, this sector is as high as the door should go. It was necessary as you can see when you go

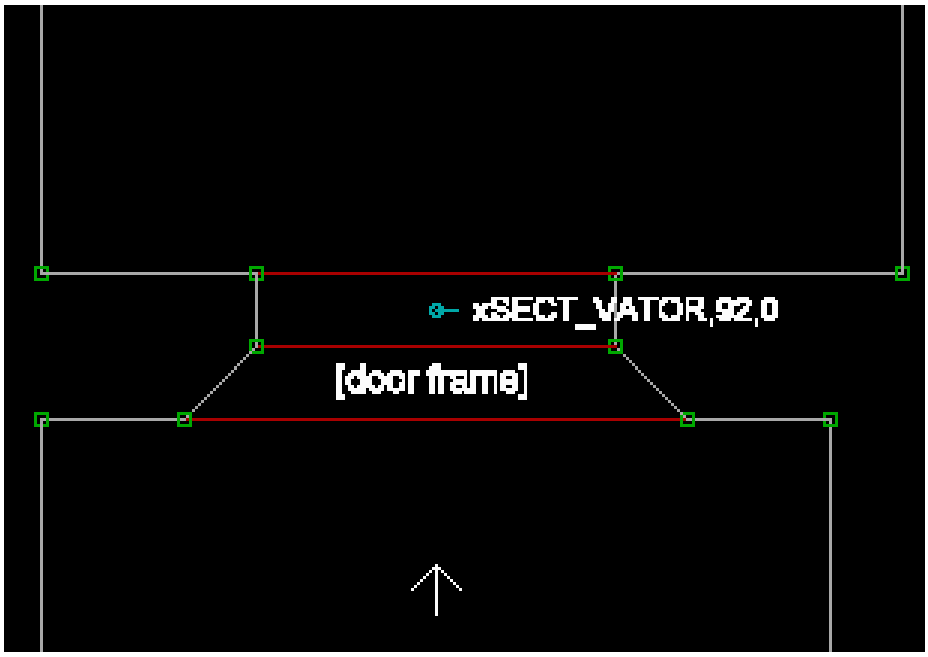


**Figure 2: The basic door with a door frame on its lower side.**

to the other side and take a look at the door from the northern side in 3D mode or in the game. Doesn't look much like a door, does it? Such door frame sectors are always necessary when the sectors next to the door sector are higher than the door is supposed to open. In that case the door texture would repeat up to the ceiling of the adjactent sector(s) and the only way to avoid that is to create door frames.

I have adjusted the angle (TAG4) of the SECT_VATOR sprite to make the door open rather slow by setting it to 130. I have also set BOOL1 and BOOL3 to „1". BOOL3 = 1 means that the door will not crush the player and BOOL1 = 1 tells the SECT_VATOR function that the door will start in the ON position. Have a look at the door in 3D mode and you will notice that the door is open and by setting BOOL1 to „1" the door will be closed when loading the map in Shadow Warrior. When building a door or similar functional object in Build then the position of the object in Build is the OFF position, whereas the position of the ST1 sprite would be the ON position. This is the reason why the SECT_VATOR ST1 sits on the floor - if it wouldn't be on

the floor the door wouldn't close fully, only up to the height of the ST1 sprite. In addition to that the ST1 sprite is flipped, it looks like if it were „mirrored". This tells the SECT_VATOR function that it should operate on the ceiling rather than the floor. To flip the ST1 sprite so it is upside down point at it in 3D mode and press **F** twice. It is already flipped so leave it as it is.

When you operate the door in Shadow Warrior you will notice two things, for one the door is silent, it doesn't make any noise plus the two sides of the actual door frame will move up and down together with the door sector's ceiling. This looks rather bad and is easily fixed. In 3D mode, point at one of the two sides and press the **O** key, do the same on the other side. This keeps the textures from moving with the ceiling because they're now oriented (anchored) to the floor.

To have the door emit a sound when it is operated place a SOUND_SPOT ST1 (Hitag 134) somewhere near or in the door sector (where exactly doesn't really matter) and give both the SECT_VATOR and the SOUND_SPOT Lotag the same, unique and non-zero match tag value so they're linked together. The SOUND_SPOT's angle (TAG4) determines which sound it will play, take a look at the Digital Sound Reference in SWREF.DOC to find an appropriate sound. I have set up a SOUND_SPOT for you and moved it far north. It is improperly tagged, so you would have to change its lotag or the SECT_VATOR's lotag first so that they match before you can hear the sound. Take a look at this ST1 if you have problems setting the SOUND_SPOT up.

## 2.IV) *Learning More*

To sum it up, creating effects in Shadow Warrior can be very easy, once you know how. It is also very easy to screw things up, in that case, when you try to create an effect and it doesn't work but you can't see why, start over from scratch, maybe using a different set of parameters. If it is a complex object, design the easiest elements first and keep adding to it, always testing in between if it still works. Undoubted, the best way to learn designing good Shadow Warrior levels is to play it and then having a look at the original levels in Build to see how the designers set up certain effects. Concentrate on the easier things first and try to rebuild them in a small test map. When you figured it out go on to explore another special effect.

Feel free to use the TUTORIAL.MAP as a basis for your own experiments. Try playing with the tag values and see what effects you can provoke. I only ask you NOT to upload this map or a modified version of it.

# 3) Timing Vators

by Keith Schuler

Timing vators can be used to time events, causing things to happen after a pause or whatever. A timing vator is usually just a small sector somewhere that the player can't get into. A SECT_VATOR ST1 (Hitag 92) in the sector has a TAG6 match value equal to the event you want to trigger. The timed vator will be activated by a normal trigger or an event triggering it. Once the timed vator reaches its destination it will trigger the event with the same match tag as the Vator's TAG6 value. Adjust the speed or height of the vator to make a longer or shorter timer. Examples of timing vators exist in almost every map of Shadow Warrior.

# 4) Room-Over-Room

by Keith Schuler

Room-Over-Room is the most significant feature of Shadow Warrior. It allows for building genuinely 3-dimensional areas by placing one layer of sectors over another, and allowing players and sprites to pass freely between the two layers. It is, admittedly, a hack, and as such several rules of construction must be followed in order for it to work at all. This being the case, we'll start off by explaining what the heck the BUILD engine is doing to produce this effect anyway.

## *Room-Over-Room Contents*



## 4.I) *How it works*

The BUILD engine is sector based, which means every sector must have walls, a ceiling, and a floor. Sectors can overlap each other, but these overlapping sectors can never see each other, or the view becomes garbled.

### 4.I.1) BOUND_FLOOR_BASE_OFFSET and BOUND_FLOOR_OFFSET

These two ST1's (Hitags 202 and 203, respectively) are used to drag groups of sectors on top of each other at premap. The BASE_OFFSET serves as an „anchor point.“ The next BOUND_FLOOR_OFFSET processed is moved to the same x,y location as the BASE_OFFSET, dragging every sector connected to it along for the ride.

### 4.I.2) VIEW_LEVEL1 and VIEW_LEVEL2

Let's assume for now that the player will be viewing the lower layer (level 1) from within the upper layer (level 2). That's when the VIEW_LEVEL1 ST1 (Hitag 110) kicks in. Behind the scenes, VIEW_LEVEL1 causes Shadow Warrior to draw level 1 as though the player were standing in it, but with one difference: the ceiling is moved up really, really high. If the player had been instead standing in the lower layer (level 1) and looking up at level 2, then the VIEW_LEVEL2 (Hitag 111) sprite does a similar action.
Behind the scenes, VIEW_LEVEL2 causes Shadow Warrior to draw level 2 as though the player were standing in it, but with one difference, the floor is moved down really, really low.

### 4.I.3) The Floor Mirror

Okay, so behind the scenes, we've drawn this weird looking area with a really high ceiling or low floor. Now Shadow Warrior draws the layer that the player is actually standing in, but it doesn't draw anything where the floor mirror is. The „floor mirror“ texture (tile #341) is a special texture used expressly for this purpose. Because Shadow Warrior didn't draw anything where the floor mirror was, the scene appears to „see through“ it into the other layer. That's the image that the player sees on the screen at the next refresh.

## 4.II) *The Rules of Construction*

### 4.II.1) BOUND_FLOOR_BASE_OFFSET and BOUND_FLOOR_OFFSET

As stated above, these are necessary in order to drag one level over the other. The Lotag is the order in which they are processed, so you'd set a BASE_OFFSET first, with a Lotag of 0. Then you'd set up all the BOUND_FLOOR_OFFSETS you wanted to align to that BASE_OFFSET.

### 4.II.2) VIEW_THRU_FLOOR and VIEW_THRU_CEILING

One VIEW_THRU_CEILING ST1 (Hitag 120) must be placed in a sector in level 1 with a „floor mirror“ texture on the ceiling. The Lotag is a view match tag, and must be the same as the Lotag for the VIEW_THRU_FLOOR sprite, as well as all the Lotags of the VIEW_LEVEL1 and VIEW_LEVEL2 sprites in the room-over-room area. One VIEW_THRU_FLOOR ST1 (Hitag 121) must be placed in a sector in level 2 with a floor mirror texture on the floor, again, the Lotag must be the same as the VIEW_THRU_CEILING sprite. Both of the VIEW_THRU sprites must be in the same relative positions in their respective sectors. Any sectors in level 1 with a floor mirror texture must be congruent to their matching sectors in level 2.

## 4.II.3) VIEW_LEVEL1 and VIEW_LEVEL2

These VIEW sprites are responsible for the actual drawing of room over room, so there must be one in every sector where the player can see the other level. Remember how the first area is drawn as though the player's view were there, with the ceiling pushed up? This means that there must be valid player space in level 1 everywhere the player can view it from level 2. This means that if a player is standing anywhere in level 2, he cannot be standing over „null space“ in level 1. When you take a look at one of these room over room areas in the game maps you will notice that some of these areas use red wall loops with the ceiling and floor at the same height to accomplish this, so the player never stands in „null space“ in either level. Use the pool area in $WHIRL.MAP as an example and have a look at figure 3. Also note that the VIEW_LEVEL sprites should always be angled downward in 2D mode.

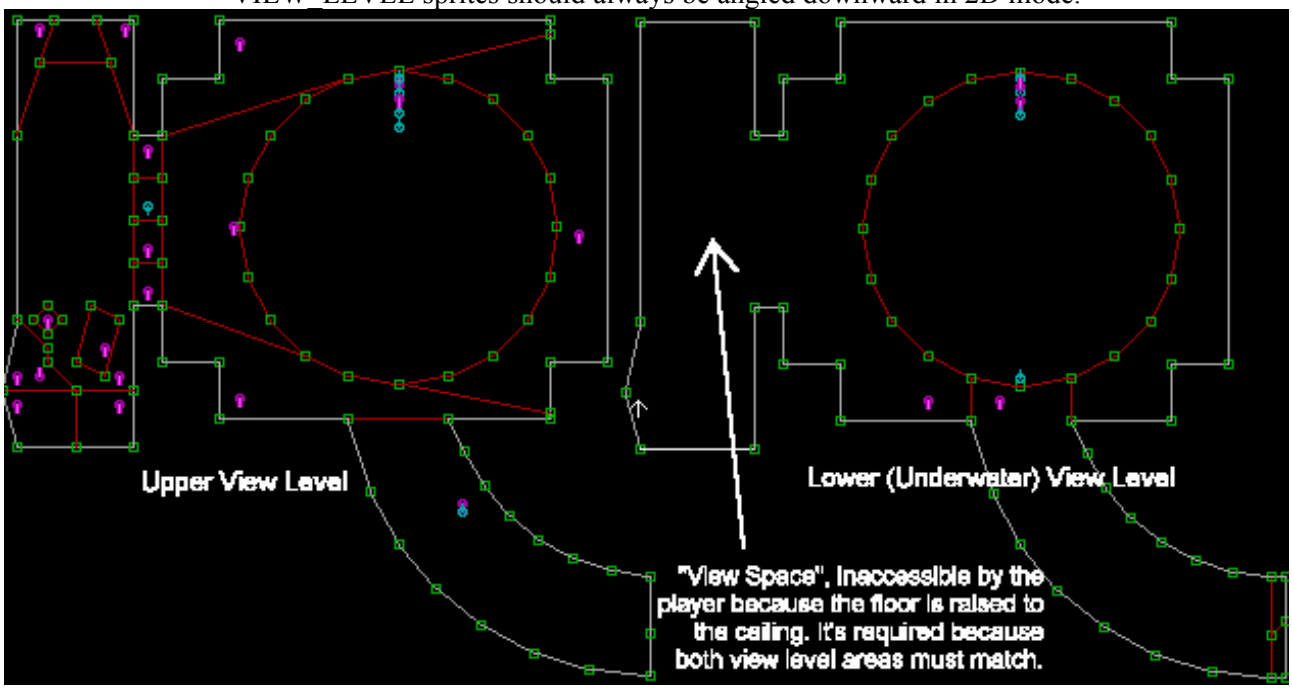


**Figure 3: $WHIRL.MAP secret pool area. To the left is the above water area and to the right the underwater area. On the right side only the pool can be accessed by the player, the surrounding sector is valid player space but with the floor raised to the ceiling.**

## 4.II.4) Raising Ceilings and Lowering Floors

Because Shadow Warrior temporarily alters ceiling and floor heights in room over room areas, two side effects will occur. The first is that you can never see the ceiling of level 1 from level 2, nor see the floor of level 2 from level 1. The second is that wall textures will move depending on whether they are oriented to the ceiling or floor. Keep this in mind when constructing your room over room areas.

## 4.II.5) Z Heights and Overlap

Level 1 and level 2 must be constructed with proper Z heights, because Shadow Warrior won't do it for you. By this I mean that the floor of level 2 must actually be higher than the ceiling of level 1. The difference between the two is called the

„overlap“. At least some overlap is necessary for room over room to behave correctly. To build overlap correctly, follow these guidelines:

1) The height of the „floor mirror“ (on the ceiling) in level 1 must be the exact same as the height of the -floor- (not the „floor mirror“) in level 2.
2) The height of the „floor mirror“ in level 2 must be the same as the height of the -ceiling- (not the „floor mirror“) in level 1.

# 5) Advanced Room-Over-Room

by Keith Schuler

Confused by room over room yet? Now let's move on the exceptions and special cases!

## *Advanced Room-Over-Room Contents*



## 5.I) *Visible Floors and Ceilings*

Above, we stated that you can never see the floor of level 2 from level 1, and you can never see the ceiling of level 1 from level 2. We lied. In $SHRINE.MAP, you can see sloping floors in level 2 from level 1 out in front of the temple as seen in figure 4. In $AUTO.MAP you can see a car on the floor of level 2 from level 1. This is a special trick, and you'll need to look at those maps to see how it's done. Here are some guidelines:

**Figure 4: The Temple in $SHRINE.MAP.**

1) You must use two sets of VIEW_LEVEL and VIEW_THRU tags. Depending on the sector, some VIEW_LEVEL tags will be turned „on" (pointing down) and some will be turned „off" (pointing up.)
2) The player can only see the floor/ceiling from the sector with the „floor mirror" texture, so be sure to construct your area accordingly.
3) The player cannot see the floor/ceiling and see more floor mirror on the other side. It just won't work.

## 5.II) *Translucent Water*

To do this, build level 1 like any room over room area, but tag it to be water, too, by placing a SECT_UNDERWATER (Hitag 8) sprite in there. The floor mirror sector is the sector you can enter and exit the water from. The floor mirror sector in level 2 should not have a DIVE_SECTOR (Hitag 7) sprite, but it will need a FLOOR_Z_ADJUST (Hitag 98) with a Lotag of 40. This allows the player to wade along the surface without „falling" underwater. Use a CEILING_FLOOR_PIC_OVERRIDE (Hitag 136) to give the water a texture.

*NOTE:* This technique was also used to create reflective or masked floors in Shadow Warrior; as seen in $WHIRL.MAP (see figure 5).

**Figure 5: The reflective floor area in $WHIRL.MAP without the transparent floor. It is not a true reflection but the same room designed twice with everything placed upside down.**

## 5.III) *Sloping Room-Over-Room*

Yes, it can be done. Look at $AIRPORT.MAP and figure 6 for an example.


**Figure 6: Sloped room-over-room area in $AIRPORT.MAP. Viewpoint is in View Level 1 and can not see the floor in View Level 2 because it was moved way down by the 3D engine.**

# 6) Sector Objects

by Keith Schuler

A „Sector Object“ is a group of connected sectors that operate as a unit. Driveables, amoebas, gun turrets and numerous other things can be built using Sector Objects.

## *Sector Objects Contents*

## 6.I) *The Rules of Construction*

### 6.I.1) The Wall Loop

Any Sector Object must be entirely surrounded by an unbroken „wall loop“. This means that no line can connect a vertex on the wall loop to a vertex within the Sector Object. One line of this wall loop must have a Lotag set to TAG_WALL_LOOP_OUTER (Walltag 504).

### 6.I.2) The Bounding Box

Every sector must have two SECT_SO_BOUNDING ST1 (Hitag 500-600). One sprite (BOUND_SO_UPPER) is placed in the upper left corner of the Sector Object, while the other (BOUND_SO_LOWER) is placed in the lower right corner. These two sprites form an imaginary rectangle.
Place these sprites so that this „imaginary rectangle“ is large enough to contain the entire outer wall loop plus any sprites you want to move with the Sector Object. For Sector Object number 0, the BOUND_SO_UPPER sprite has a Hitag of 500 and the BOUND_SO_LOWER sprite has a Hitag of 501. For Sector Object number 1, use 505 and 506, respectively. Sector object number 2 uses 510 and 511. This continues all the way up to Sector Object number 19, which uses Hitags 595 and 596.

### 6.I.3) The Center Sector Lotag

Every Sector Object must contain one and only one sector tagged as its „center sector“. This determines the center point around which the Sector Object will pivot. The Lotag of the center sector will always be the same as the Hitag of the Sector Object's BOUND_SO_LOWER sprite. So, Sector Object number 0 uses 501 as its Lotag, SO number 1 uses 506, number 2 uses 511, and so on all the way up to Sector Object number 19, which has a center sector Lotag of 596.

### 6.I.4) The Center Sector Hitag

The sector Hitag of the center sector is the track number that the Sector Object will follow. See SWREF.DOC for a description of track sprites. If you don't want the Sector Object to follow a track, set its Hitag to -1. Other special cases include: 95 for a killable Sector Object, 96 for an auto-turret, and 98 for a driveable.

### 6.I.5) Sector Object Limitations

You can have up to twenty (0 - 19) Sector Objects in a map. Each Sector Object can contain up to 30 sectors.

## 6.II) *The Many Uses of Sector Objects*

### 6.II.1) Follow A Track

This is the most common use of a Sector Object. Set the Hitag equal to a track number and make the thing wander around. See SWREF.DOC to find out what you can do with tracks.

### 6.II.2) Auto Turret

Set the center sector Hitag to 96, then put an AUTO_TURRET ST1 (Hitag 81) in the center sector. Put in a SO_ANGLE (Hitag 16) to tell it which way is the front, and then the Sector Object will always turn to track the movement of the player. Add a SHOOT_POINT and the auto-turret will shoot at the player.

### 6.II.3) Driveables

Set the center sector Hitag to 98, then put a SECT_OPERATIONAL (Hitag 1) and a SO_ANGLE in the center sector and your Sector Object can now be driven around by the player. It can never leave the sector surrounding the wall loop, though, so keep that in mind.

### 6.II.4) Bind It Across Floors

If you want to place a Sector Object in water or in room over room areas, you're going to want a „bottom“ that moves with the top half of the SO. This can be done pretty easily with BOUND_FLOOR_BASE_OFFSET and BOUND_FLOOR_OFFSET. If the Sector Object finds itself in a BOUND_FLOOR area, it will automatically try to find its other half in the matching BOUND_FLOOR area. All the secondary part needs is a wall loop tagged with 504 or 508, and it will move as a unit. See rooms FB,FC, FE and FF in EXAMPLE.MAP.

# 7) SWSAVE Debugging Feature

by Keith Schuler

SWSAVE is a powerful and convenient debugging feature of Shadow Warrior. We found it to be immensely useful for finding stacked sector walls that don't match up, as well as other odd uses. While in the game, press „T“ and type in „SWSAVE“, just like a cheat code. The program will save the map in its current state as SWSAVE.MAP which you can then load into BUILD. BOUND_FLOOR_OFFSET dragging will have taken place, so you'll see room over room area sectors in their actual positions during the game.