Avara Level Design Manual

version 1.0.0

Introduction

One primary consideration in the Avara design has always been to make level editing as easy as possible. No special purpose level editors are necessary, although they are quite possible to write, if standard tools are not available. Avara is a set of tools for producing several different kinds of networked games. The design is 100% object oriented, so it is easy for us to expand it, yet remain compatible with existing levels.

Avara level design may seem complicated at first, because there are so many parameters that you can change. The key is that you don't have to change them if you don't want to. Start by just using the default object parameters and then explore the possibilities by changing the parameters one by one.

Level Directory Files

Avara levels are defined as PICT files or resources. ClarisWorks™ is the recommended program for designing Avara levels, but other programs may also be used, as long as they create similar PICT files. To use PICT files, you need a level directory file where the level you edit has been declared to exist in a file. The level file has to be in the same folder as the level directory file. To use PICT resources, the level directory file must indicate that the level is loaded from a resource and the resource must be placed in the level directory file and named correctly.

Level files (PICT files) are recommended for debugging and trying out levels and level resources (PICT resources) are recommended for shipping or distributing levels after they have been fully developed and debugged. With PICT files, you don't even have to quit Avara to make a change to a level. Just open the directory file into Avara and every time you make a change, reload the level from the file by clicking on it.

Resources

The level directory file contains one 'LEDI' resource with a resource id of 128. This resource serves as a directory to all the levels this file contains or accesses. In addition to this resource, the file may contain any number of 'BSPT' (Binary Space Partitioned Tree) resources for describing object shapes and any number of 'HSND' resources for compressed sounds. The only required resource is the 'LEDI' resource. If the following paragraphs seem difficult to grasp, don't worry: you may not need the informaton.

'BSPT' resources are created from DXF (AutoCAD™ drawing exchange file) files by running them through the BSPSplitter program and the BSPViewer program. Colors for 'BSPT' resouces are defined in a ColorLib file where DXF layer names are mapped into RGB (red, green, blue) colors. The BSPViewer program can be used to verify that visible surfaces are correctly defined in an object and correct the faults, if necessary. There's a more detailed section on 3D modeling for Avara later in this manual.

Resource ID numbers between 1000 and 2999 are allowed in a level directory file. All other resource IDs are reserved for use by Avara or the system. The only exception is the 'LEDI' resource.

Level Files and Resources

A level file is just a drawing of the level map. The order in which the shapes are in the drawing is important. Drawing programs typically have commands such as: 'Send Behind', 'Bring Forward'. A shape that is behind another one appears before that one in a level file. Avara processes level files from back to front (the same order that they are drawn onto the screen). If this seems confusing, there's a more detailed explanation at the end of this section, along with an example.

The most important shape is the text shape. This is just a box of text where you type a program that Avara will then parse and execute when it loads that level. More information on the language itself is given later in this manual.

If you wish to declare variables or constants for your own use in a level file, make sure that the text shape is behind all the other shapes on a level. Optionally you can place the definitions in a 'TEXT' resource of ID 1000. It will be loaded before any of your Level PICTs. This is in fact an ideal place to define named constants for any shape or sound resources that you may have created.

Important: Text objects should always be transparent. Some drawing programs implicitly place a white rectangle as a background for a solid text box. If you can't see what's behind the text, you probably have a solid rectangle there. The rectangle color will interfere with the object colors that you have set elsewhere.

Filled rectangles and rounded rectangles are also important, since they are the easiest way to place walls in an Avara level. The default wall height is 3 meters, but it can be changed with the wallHeight parameter.

Round cornered rectangles can be used to create walls of various heights. In ClarisWorks™, if you double-click on a rectangle or a roundrect, a dialog box will prompt you to enter the radius of the corners. The conversion from rounding radius is done by multiplying the diameter by the variable pixelToThickness. The default value for pixelToThickness is 1 / 8.

This means that by default 12 pt radius is equivalent to a wall height of 3 meters. Each pt in radius is thus 25 cm (a quarter meter). Note that some drawing programs may want you to

enter the rounding diameter instead of the radius. In this case, each point in diameter is 12.5 cm (one eighth of a meter).

Walls can also be placed off the ground and on top of each other, to construct more complicated structures like roofed areas.

Arc segments (like this: ) are used to indicate the position, direction and color of an object and can also be used to create quarter, half and 3/4 domes. The tip points to the direction the next object will be facing. If you want to place several objects in the exact same place, you can use the same arc segment for all of them. A good example of this would be a hologram (a 3D shape that you can walk through) combined with an object that does not have a shape (such as an reincarnation spot).

Ovals are used to mark areas or to create full domes. In the current implementation, only circular areas are supported, so if the horizontal and vertical dimensions of an oval are not equal (in other words it is not a circle), the larger dimension will be used for both. In practice the areas also extend up to form a full sphere, so keep that in mind when designing a level in three dimensions. (The area may extend from one floor to another.)

Currently all other drawing shapes are simply ignored, but they are all reserved for future use, so you should not use them in your levels.

Details and Troubleshooting PICTs

Part of this subsection assumes you know things that come later in the manual. You may want to read it through now and read it again after you have read through the manual once.

Q: *I don't get this "Back" and "Front" deal.*
What needs to be in back of what, and what needs to be in front?

A PICT file is a bit like a program or a cooking recipe. For instance, if you have rectangle with some text in it, the instructions to do so might be something like:

```
draw a filled rectangle with coordinates {{0,0},{200,100}}
outline that same rectangle
a block of text starts here
draw text "Hello" with Helvetica 12 at coordinate {2,14}
draw text "World" with Helvetica 12 at coordinate {2,30}
text block ends
```

That's what Avara will see. Whatever comes first ends up behind what comes after, if the graphics overlap. Visual overlap however has absolutely no significance in Avara: all that matters is the order in which the opcodes (instructions) in the picture come.

The next thing you need to know is when objects are actually created. Walls are a special case: they are created when the rectangle frame frame is drawn. Other objects are usually created by writing:

```
object Type
  parameters = expressions
end
```

This means that if the object "Type" needs a location that has been defined with an arc segment somewhere, that arc segment has to be the last arc segment before the text.

On the other hand, if you need to set a wall parameter, like the altitude at which the wall is placed (parameter wa), then that text has to come before the rectangle frame.

So, for instance: (omitting PICT stuff that Avara doesn't need)

```
1       draw text "wa = 0.5"
2       draw a red filled rectangle with coordinates {{0,0},{200,100}}
3       frame the last rectangle with black
4       fill a blue arc segment with coordinates {{300,300},{350,350}}
        from 0 degrees with 90 degree angle
5       frame last arc segment with white
6       draw text "object Incarnator end"
7       draw text "object Hologram end"
```

What this does is place a wall 0.5 meters from the ground. The wall color is defined at step 2, but the wall is created at step 3 (because walls are created when they are framed, but the fill color is used for color).

The arc itself does nothing, except changes the current colors to blue and white and remembers its location and orientation.

The text coming at step 6 will create an incarnator at the tip of the arc. Incarnators do not have color, so the color is just ignored.

The text coming at step 7 will create a hologram at the same location and the hologram will be colored with blue and white. The default hologram only uses one color (the blue fill color), so the other color is in this case ignored.

I hope this helps. I guess the confusing part is that Avara is treating the PICT like a computer program and people are used to thinking of pictures as pictures, instead of sequences of instructions or opcodes.

Q: *If I copy a PICT into ClarisWorks, make a change and put it back, it no longer works!*

If you read back the PICT version of, say Fort of Bob, ClarisWorks will join lines of text without putting returns or even spaces in between.

So, the following:

```
object Incarnator
end
```

May become:

```
object Incarnatorend
```

This is a ClarisWorks bug and will hopefully be corrected in a future release. Other programs may or may do not better in this respect.

The Scripting Language

The Avara scripting language is roughly similar to the language that is used to declare Arashi levels. It follows a fairly simple syntax that allows you to define formulas for variables, declare enumerated constants and create objects.

All keywords and variables names are case-sensitive. That means that my_variable is not the same as My_Variable because 'm' does not match 'M' and 'v' doesn't match 'V'. Object classes start with a capital letter, parameter names with a small letter. ('Door' is a class, 'speed' is a parameter.)

Variables are declared simply by

```
variable = expression
```

Where the variable name may contain any characts a-z, A-Z, '[', ']', '\', '|', '{', '}' '_', 0-9 and '.' as long as the first character is a letter. The expression may contain references to other variables, numerical constants and the following operators: +, -, *, /, % (modulo), ^ (power), | (absolute value a = |-1 is the same as a = 1), <, >, =. Usually you will just need to assign a constant value to a variable like this:

```
speed = 10
```

But of course you could just as well have declared that:

```
speed = 5 + 5
```

For clarity, please use spaces as illustrated above.

The Avara language uses lazy evaluation. This means that it doesn't evaluate variables unless it has to. To illustrate:

```
a = 1
b = a + 1
a = 2
```

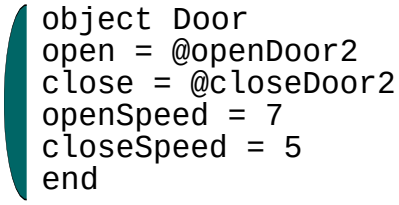
Most programming languages would evaluate this differently from Avara. In Avara, new definitions for variables replace old ones, so in Avara a = 2 and b = a + 1, which means that

evaluating b gives you 3.

The most important kind of statement is the object declaration statement. It is used to create objects in an Avara level. In most cases you precede the object placement statement with a graphical shape (like a filled arc segment) to specify the location and direction of the object. The syntax of an object declarition is like this:

```
object class
optional assignments
end
```

A list of classes that you can use is given later in this manual. The assignments are used to override default values for object parameters. Here's a real-life example of how you would define a door:



```
object Door
open = @openDoor2
close = @closeDoor2
openSpeed = 7
closeSpeed = 5
end
```

In the above example, @openDoor2 and @closeDoor2 are constants that are used as message numbers. Messages are broadcast between objects, so that if a player enters or leaves an area, it can trigger a message, which can in turn open or close any number of doors or activate other kinds of events. Notice that the door opens faster than it closes (openSpeed = 7, while closeSpeed = 5). The arc segment on the left side of the of the text defines which way the door is facing and where it is placed. The size of the arc is ignored.

The object initializes all its parameters to default values. It then parses and compiles the assignment statements until an end keyword is found. At this point, it evaluates all its parameters and the object is placed in the game (if necessary).

You can use the an enum statement to declare a series of numerical constants:

```
enum 1 One Two Three Four end
```

The first expression after the enum keyword is used for the first variable. In this case, the variable One receives the value 1, Two receives 2, and so on. The 'end' keyword ends the sequence. Enumerated variables may later be reassigned with other values, so be careful when you choose variable names. Enums are mostly useful when you are declaring names for resource numbers (for sounds and 3D shapes) that you use in your level directory files.

An alternative to the enum statement is the unique statement. The difference is that normally you do not give the value of the first number. Instead, the numbers are given out sequentially from a base. In Avara, this base is defined to start at 30000. All message numbers equal to or higher than 30000 should be considered reserved for this purpose. Here's an example of a "unique" statement:

```
unique mOpenGate mCloseGate end

object Door
            openMsg = mOpenGate
            closeMsg = mCloseGate
end

object Area
            enter = mOpenGate
            exit = mCloseGate
end
```

The advantage of unique constants is that you can reuse the same symbols many times. For instance, the above script can be used several times on a level and the openGate and closeGate variables will have new values each time, so that the area only controls one door.

Because the danger exists that an object may use a variable name as a parameter and change your value for it, it is recommended that local message variable names be prefixed with a lower case m character. This guarantees compatibility with future versions of Avara.

You can also convert variable names into numerical constants. For instance, imagine that you have the variables One Two Three and Four. In addition to the their value (which can change over time and be anything), they all have unique indexes to identify them. (if you are familiar with programming languages, you can think of it as the address of that variable, although it's not an actual address in the computer's memory). To get the index number (or address) of any variable, use the @-operator. You can only apply it to variable names. If the variable does not already exist, it is automatically created for you. Use it like this:

```
open = @openNorth
close = @closeNorth
```

In the above example, open and close each receive a unique value. If you use the expression @openNorth anywhere within that level, the value is guaranteed to be the same that was assigned to open.

In general, you should use "unique" statements for local message numbers and @-sign notation for any global messages. The actual values of @-sign constants start from zero and grow sequentially. This means that message number values of less than 1000 should be considered reserved for @-sign constants.

Strings are enclosed in quotes like this:

```
text = "This is a string."
```

Strings can include return characters, but tabs will probably not work. If you want to include the quote character, use two consecutive quote characters like this:

```
text = "Feel free to ""quote"" me on this one."
```

Comments work the same way as in C++, although you may want to be careful with the // comment style, because it signifies that the rest of that line is to be considered a comment. Because drawing programs wrap text automatically, some words from the end of the comment may move to the next line.

Normally it is safest to enclose comments like this:

```
/* This is a comment */
```

The alternate style is:

```
// This is a comment.
```

Passing Messages

Object communicate between each other using messages. The message consists simply of a message number. A message has a sender and any number of receivers. When the objects are created, you have the option to assign numbers to certain message reception slots. When that message is triggered, the object is notified and it acts accordingly.

Avara Objects

This section of the manual describes object classes in a tabular format. Each class description begins with a line like this:

Superclass	Class	Description

If a class is abstract, it can not be placed in the game. The names of abstract classes are placed in brackets and they are not available for placement in the game. Abstract classes are incomplete, but their non-abstract subclasses can and should be used in the game.

It is important to understand inheritance when you read the table of objects. Each object has the parameters of its superclasses as well as its own parameters. Default values for these parameters may differ and in some cases some parameters may be ignored by a class or simply used slightly differently (these differences are noted and documented in the comments).

After the first line, a list of parameters with default value assignments for this object class are listed. Please note that default parameter values may change slightly from one version of Avara to another, so if you want to be absolutely certain of a value of a parameter, assign it in the object declaration statement instead of relying on the default. New versions of Avara may also introduce additional parameters. Old levels should still work, because default parameter values are chosen so that they are compatible with old level files.

Superclass	Class	Description
class	**[actor]**	The following variables are supported by all actor types.

`hitSound = hitSoundDefault`

This is the resource id for the sound that the object makes when it is hit with a missile or something explodes close to it, possibly causing damage. The variable hitSoundDefault is initialized by Avara, but can be changed in the level file. Changing the default allows you to change the sound for all objects that normally make the default sound when they are hit.

`hitVolume = 25`

This changes the volume of the sound that is played when the object is hit. Don't change this unless you have a good reason.

`blastSound = snBlast`
`blastVolume = 40`

Defines the sound that is played when this actor is destroyed.

`shield = -1`

Defines the shield for this actor. Negative numbers mean infinite shield (indestructible objects).

`power = 0`

The explosion damage done at a distance of 1 meter or less. From 1 meter on, the damage done is power/distance^2. The minimum damage is 1/16 units, so a mine with a blasting power of 2 will do the minimum 0.0625 units of damage at a distance of 5.6 meters.

`team = 0`

This assigns a team to the object. By default, most objects belong to the computer team (team zero). Changing this value may change the behavior of certain objects and can affect how scores are calculated.

`isTarget = false`

If isTarget is set to true, the object is considered a target by the player HUDs and the HUD indicators will light up accordingly.

`hitMsg = 0`
`killMsg = 0`

These messages are sent when an actor has been hit or destroyed (in which case the hitMsg is also usually sent).

`hitScore = 0`
`killScore = 0`

Points are awarded according to the power of the hit (energy * hitScore) or when the actor is killed.

`shapeScale = 1`

This variable allows you to scale any bsp shapes in an object. Note that a scaled object uses more memory than an unscaled one because certain structures can not be shared if the object scale changes.

`yon = 0`

This determines how distant the "yon" clipping plane of a 3D object is. The default value of 0 means that the default

value (set by the user with the viewing distance) is used.

`stepOnMsg = 0` This message is triggered when a player steps on this object.

```
traction = defaultTraction
friction = defaultFriction
```

If the object can be walked on, these values define the characteristics of the walking surface. There's a section on these values near the end of the manual, after the level variables section.

[actor] [placed] Arc segments are used to place, rotate and color objects that belong to this class.

`y = 0` The height of the object from the ground. You can think of the coordinate system as the metric system. Since the walker (or HECTOR) that the player controls is slightly under 2 metres high, placing an object at `y = 1.5` would put it at about eye level. Most objects are designed to be placed on the ground.

[placed] [glow] The actors in this class glow when they are hit, assuming their shields are set to destructible by default. Glowing actors with non-negative default shield values also use a different default shield hit sound. As usual, you can change it to whatever you want.

`canGlow = true (or false)` The value of canGlow depends on what the default shields setting is for the subclass of [glow]. In any case, if you want an object to glow when it is hit and it isn't glowing, set 'canGlow' to true. If it is glowing when it shouldn't, set 'canGlow' to false.

[glow] [shooter] Shooters are objects that can shoot in any direction. These currently include the "Ufo" , "Ball" and "Pill" objects.

`mask = allTeams`
`watch = playerMask` Similar to "Guard". See description of Guard or Ufo.
`activeRange = 100` This is the radius within which the shooter looks for targets to shoot at.
`shotPower = 0.5` Power of a single shot.
`burstLength = 2` How many shots can be fired in a single burst.
`burstSpeed = 5` How many frames between each shot in a burst
`burstCharge = 40` How long it takes to recharge after the first shot of a burst has been fired.

[placed] Hologram Holograms can be used to place markings and graphics on the floors and walls and above them. Holograms are not solid, so you can walk and shoot through them.

`shape = bspGroundStar` The default shape is a four-pointed star that is on the

	ground. In many cases, you want to use something more interesting. Marker white (color 254 254 254, or named "marker") is replaced with the fill color of the arc.
`roll = 0`	You can rotate the shape around the Z axis. A bent arrow pointing left can be made to point right, if it is rolled by 180 degrees.
`isAmbient = false`	Ambient holograms can be turned off by the user to make the program run faster. If you want to decorate your level with holograms and yet those holograms are not essential to game play, it is advisable to make them ambient by setting isAmbient = true.
[glow] Solid	Solids are used when simple rectangular, axis-aligned walls are not enough. If you wanted to, you could place a sculpture of the Venus d'Milo in the middle of an Avara level. You should be aware that collision detection is only done with bounding boxes and spheres, so even complicated shapes will still behave as if they were boxes or spheres. For example you can't shoot through a donut shaped solid.
`shape = w1x1`	The default shape is a 5 square meter wall box that is 3 meters high. Marker white (color 254 254 254, or named "marker") is replaced with the fill color of the arc.
`roll = 0`	You can rotate the shape around the Z axis.
Solid WallSolid	See the description for "WallDoor" to learn how a WallSolid differs from a regular Solid. Mostly useful if you want to create a destroyable wall.
Solid FreeSolid	The FreeSolid is a bit like a WallSolid and Solid, but external impulses from shots and explosions can move them around. They primarily fly and slide on surfaces like ice cubes.
`shape = 0`	You can use any BSP shape resource, but if you leave "shape" as 0, the last wall created will be used just as if this was a WallSolid or a WallDoor.
`mass = 50`	Mass determines how easily the FreeSolid moves when it is hit. A heavier mass will take more power to move.
`customGravity = 0.12`	Acceleration under standard 1.0 gravity. Note that you can control the "gravity" variable for the whole level. The acceleration for this object is calculated by gravity * customGravity.
`accel = 0.95`	When in motion, the FreeSolid usually decelerates slowly. The default rate is 5% slowdown per frame. Changing the

	accel value to be closer to one will reduce the "friction". Making it smaller will increase the "friction", and the rate at which it slows. Note that FreeSolids do **not** use the friction and traction values set for different surfaces when they are sliding on those surfaces. Reasonable values for this parameter range from slightly over 0 to around 1.0 (values greater than 1.0 will make the FreeSolid accelerate until it hits something.)
`shotPower = 0`	Like Doors, FreeSolids can cause damage if they hit you while they are moving.

`start = @start`
`stop = 0`
`status = false`
	Use these messages to freeze and unfreeze FreeSolids. For instance, a switch might release a bridge and make it fall down. The uses are of course endless. (A bridge might fall down as soon as you touch it. Of course you can use a Door object for that, but you can't get realistic acceleration under gravity from a door.)

[shooter] Ball	The ball is like a freesolid that is attracted to a HECTOR. It can be carried by players or placed in a goal.
`group = -1`	A ball can only be accepted by a goal, if the binary "and" of their groups is nonzero.
`shotPower = 0`	Power of a single shot. (Balls do not shoot by default)
`burstLength = 3`	How many shots can be fired in a single burst.
`burstSpeed = 5`	How many frames between each shot in a burst
`burstCharge = 20`	How long it takes to recharge after the first shot of a burst.
`mass = 30`	
`goalMsg = 0`	Triggered when the ball locks into a goal.
`ejectPitch = 20`	Angle at which the ball is thrown away from a HECTOR.
`ejectPower = 1`	Speed at which it is ejected.
`shieldChargeRate = 0`	Balls have shields that charge, although the default shields are se to -1 (infinite) and the charging is off.
`maxShield = -1`	Max shields are set to infinite by default.
`shootShield = 10000`	The ball will only shoot if its shield has at least this much power.
`grabShield = 0`	The ball can only be grabbed by players if its shield is lower than this value.
`carryScore = 0`	For each game frame, this many points are awarded to the player carrying the ball.
`dropEnergy = 1`	Damage energy at which the ball is dropped by a player that was hit by explosions or shots. Shots that hit the ball count as well.
`changeHolderEnergy = 0.3`	Minimum single hit damage that you have to do to the ball to consider goals after the event to be made by you. For instance, you shoot at a ball that is going towards the enemy goal and you hit it just before it enters: you get the points, if the hit was worth 0.3 in shield energy.
`changeOwnerTime = 0`	You have to carry the ball for a certain time before it

changes sides and starts shooting your enemies. Any hits that it makes are added to your score, after the ball has changed sides. Time is measured in frames.

`customGravity = 0.04` Acceleration under gravity.

`acceleration = 0.97` Similar to the FreeSolid param of the same name. Specifically the rate of slowdown per frame.

Ball Pill Pills (for pillbox) are balls that shoot back and can be captured. The only difference is in the default parameter values. The same object is used internally for both classes.

```
group = 0
shotPower = 0.3
burstLength = 4
burstSpeed = 6
burstCharge = 32
shieldChargeRate = 0.005
maxShield = 17
shootShield = 15
holdShieldLimit = 15
customGravity = 0.12
shape = bspStandardPill
changeOwnerTime = 100
acceleration = 0.8
ejectPitch = 5
ejectPower = 0.3
watch = playerMask + scoutMask + robotMask
```

actor Goal Goals accept balls and pills, if their group masks agree (see description for ball).

```
group = -1
roll = 0
pitch = 0
deltaX = 0
deltaY = 1.3
deltaZ = 0
```

The position the ball is attracted to.

`goalScore = 500` Score that player gets for depositing a ball in this goal. The team setting of the goal determines if the points are positive or negative.

`motionRange = 1.3` Attract area of goal.

`activeRange = 0.1` When ball enters this area, it locks in and the goal action is triggered.

`goalMsg = 0` Triggered when a goal is made.

`goalAction = goalReset` What to do when ball enters. Value can be one of: goalNull (0, ball stays attached), goalDestruct (2, destroys ball), goalRelease (3, releases ball and ignores it for a while), goalReset (5, sends the ball back to its initial position as soon as there is free space there).

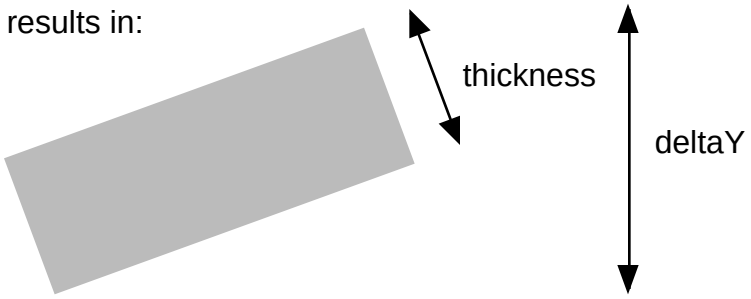
[glow] Ramp Ramps are like walls, but they are tilted and you use a thick-bordered rectangle (3 point frame thickness is

recommended) with an arc segment and the normal `object Ramp parameters = values end` to create the object.

`thickness` — You can change the thickness of the ramp by using a rounded rectangle or by changing this value.

`deltaY = 1` — This is the difference in altitude that the ramp provides. (Look at the illustration below)

```
object Ramp
deltaY = 2
end
```



The angle of the arc controls which way is up. The point of the arc points in the down direction. Note how the thickness of the ramp controls how steep it is too. Think of the deltaY parameter and the rectangle borders as a 3D box. The ramp itself fits perfectly inside this box, as long as thickness is smaller than deltaY. The thinner the ramp is, the steeper the climb itself becomes.

Note that a rectangle with 0 rounding will produce a zero thickness ramp, unless you change the thickness parameter.

[glow] Mine — This is a proximity mine. It activates if a player moves close to it or if it is damaged severely enough. An activated mine can change shape or start to pulsate at a different rate After the activation period, it explodes. Any actors within a certain radius will be damaged when the mine explodes.

`shape = bspMine` — This is the primary shape for a proximity mine. This shape is used for collision and hit detection, so the alternate shape is just window dressing (so to speak).

`altShape = bspMineActive` — This is the alternate shape for a proximity mine.

`shield = 1` — The amount of damage that the mine can take before blowing up.

`activateEnergy = 0.2` — The amount of damage that the mine can take before it activates.

`activeRange = 2` The activation range of the mine. If a player comes within this distance, the mine actives.

`phase = 0` This is the phase that the mine starts up in at the beginning of the level. For each game frame, the phase is incremented by one. This variable is useful if you want to have a group of mines that pulsate in sequence.

`freq = 4` The range of the mine is checked for targets once every `freq` frames.

`activeTimer = 0` The number of frames from activation to blowing up. With this value set to zero, the mine will blow up immediately on activation. Think of this as the length of the fuse of the mine.

`idleShapeTimer = 20`
`idleAltShapeTimer = 0`
`activeShapeTimer = 2`
`activeAltShapeTimer = 2` These variables control the speed at which the alternate and primary shape alternate. A value of zero means that the shape is not used. The default values mean that an inactive mine doesn't change shape and an active mine alternates between the primary and alternate shape every four frames.

`activateSound = snMineBleep`
`activateVolume = 1`
`blastSound = snMineBlow`
`blastVolume = 25`
The activation sound starts playing when the mine is activated and is silenced when the destruct sound is played. You can set a sound loop for the activation sound so that it plays for an indeterminate period (for example a periodic beeping sound to warn that the mine has been activated). The destruct sound should be the sound of an explosion or energy burst.

`power = 4` The damage done at a distance of 1 meter or less. From 1 meter on, the damage done is power/distance^2. The minimum damage is 1/16 units, so a mine with a blasting power of 2 will do the minimum 0.0625 units of damage at a distance of 5.6 meters.

`shield = 1`
`destructScore = 20`
`hitScore = 5`
`activate = 0` Message used to activcate a mine. For instance, you could set a long fuse (activeTimer) set activate = @start.

`start = @start`
`stop = 0`
`status = false`

[glow] Ufo Ufo stands for Unintelligent Flying Object. They may seem quite smart, but actually it's all instinct and no intelligence. :-) It uses happiness scores to determine what it should do. A course of action is evaluated and if it gets higher

	points than the current action, the new action is implemented.
`shape = bspUfo`	A funky flying saucer with two colors defined. Two color replacements can be made. The default shape uses the fill color for most parts and the outline color for the interior.
`mask = allTeams` `watch = playerMask`	What to watch for and shoot at. You can make Ufos that only attack scouts or that attack just about anything. For more information on what kinds of values the watch and mask can take, please see the section near the end of this manual.
`acceleration = 0.017`	Maximum acceleration per frame. Larger numbers increase the top speed, smaller ones decrease it. The new speed = (old speed + acceleration) * 0.99
`checkPeriod = 45`	The frequency at which the Ufo re-evaluates its position. Note that if the Ufo is shot at or collides with something, this period is significantly (and automatically) reduced.
`attack = 0.9`	The probability that the Ufo will prefer move to a position where it can attack the target.
`defense = 0.5`	The probability that the Ufo will choose to attack soon after it has been damaged. The actual probability is usually between attack and defense and always starts at attack. When the Ufo is hit, the probability changes towards defense. If it is left alone, it slowly returns to the attack value.
`activeRange = 100`	This is the radius within which the Ufo looks for targets to shoot at.
`motionRange = 40`	Range in meters that the Ufo will consider when it moves to a new position. Smaller values make the Ufo move in shorter segments.
`verticalRangeMin = -1` `verticalRangeMax = 4`	Vertical range of motion from ground level.
	These scores are used internally by the Ufo AI to determine how good a course of action seems to it:
`visionScore = 1`	A value relative to this score is awarded or deducted when a Ufo sees a target. When it is attacking, seeing the enemy is obviously a bonus, so this number is added. When it is defending, being visible is a bad thing, so the score is subtracted.
`hideScore = 0.3`	If there's an interesting target close by, this score is awarded even if the target is not visible. It doesn't matter if the Ufo is attacking or defending.
`homeSick = 0.1`	The Ufo is biased towards going back to its "home" (or original) position. Higher homeSick values will make it more eager to keep close to home.
`homeRange = -1`	A negative homeRange value means that the algorithm

	only uses the "homeSick" value and keeps the Ufo in loose vicinity of its home base. For positive homeRange values, anything within "homeRange" radius from the home base is ok and doesn't affect Ufo movments, but outside that, the distance * homeSick is subtracted from the "score". A short homeRange combined with a high homeSick value will keep the Ufo close to its home base.
`homeBase = 0`	See the description for the "Base" object to learn how Ufos can be guided through the use of this message input.
`shotPower = 0.5`	Power of a single shot.
`burstLength = 2`	How many shots can be fired in a single burst.
`burstSpeed = 5`	How many frames between each shot in a burst
`burstCharge = 40`	How long it takes to recharge after the first shot of a burst has been fired.
`start = @start`	This message activates the Ufo so that it moves around and tries to shoot its targets.
`stop = 0`	This message stops the Ufo.
`status = false`	Initial status for Ufo (active/inactive).
[glow] Parasite	Parasites are normally dormant, but start moving towards a player if they see one. When they touch a player, they clamp on and stay attached until destroyed. Parasites draw energy from the host while clamped on.
`shape = bspParasite`	Default parasite shape is small. Two color replacements can be made. The default shape uses the fill color for most parts and the outline color for the interior.
`activeRange = 40`	Parasites scan the area close to them for visible players. This variable defines that range.
`accel = 0.03`	This is the acceleration/frame when moving. Friction is fixed at 5% of speed, so theoretically the top speed is accel * 0.95 / 0.05 = 0.57 meters/frame.
`sound = snParasiteAttach` `volume = 0.5`	Parameters for sound that is produced when the parasite connects with a host.
`drain = 0.01`	Energy drawn each frame. Shields are drained at half this speed.
`maxPower = 1`	Half the energy is used to increase the energy of the parasite. When it reaches maxPower, the parasite explodes.
`mass = 50`	When a parasite is attached, its mass reduces the maximum acceleration of the player. If a HECTOR's mass is 200 kg, a 50 kg parasite will reduce acceleration by 20%.
[glow] Guard	Guards are stationary guns that rotate to track the closest target and fire when they have a clear line to the target.

`shape = bspGuard`	This resource controls the form of the guard. Missile bolts are fired from the origin of the shape.
`fireMsg = 0`	This message causes the guard to fire. Normally guards fire automatically when they track a target, but you can use this message to make them fire on other occasions as well.
`start = 0`	This message activates the guard so that it starts tracking targets in its zone.
`stop = 0`	This message makes the guard stop tracking targets. Combined with the start message, you can make guards watch for activity in a small area only.
`speed = 0.5`	Adjusts the maximum speed at which the guard turns to track targets.
`shotPower = 1`	Adjusts the power of the missiles that are fired. Higher numbers cause more damage to the object that they hit.
`freq = 30`	Determines how often the guard can shoot. (In frames)
`watch = playerMask`	Guards only track and shoot objects of certain types. playerMask and scoutMask are valid values and may be added to combine them.

`shield = 5`
`destructScore = 100`
`hitScore = 50`
`power = 1`	

[placed] Dome	Domes are special in that the size of the arc and its opening angle also determine the shape. You can create quarter domes, half domes, 3/4 domes and full domes (with circles!)
`hasFloor = false`	By default, domes do not have floors. If you want it to have a floor (if you want to view it from underneath), set hasFloor to true.
`pitch = 0` `roll = 0`	You can pitch and roll domes.
[placed] TriPyramid	A tripyramid is just a special kind of solid. Place it with an arc segment. The base of the "pyramid" is a triangle. The only thing that is special about this object is that collision detection is accurate, because the object actually consists of a corner of a cube that is rotated so that it is upright on the ground.
[glow] Door	A door can be any shape that moves when it receives an open or close message. The closed state is the default position and orientation of the object. When opened, the door tries to move from it's closed state to the open state. The open state is defined by a 3D translation and three rotation (pitch, yaw and roll). Note that you can use any BSP shape as a door. A

particularly useful shape for a door is the w1x1 wall block, since you can create walls that move around.

```
deltaX = 0
deltaY = 2.6
deltaZ = 0
```

These variables define the position of the door in its open state. The default door moves straight up.

```
pitch = 0
yaw = 0
roll = 0
```

These variables define orientation of the object in its open state. The rotations are applied in the order they are listed here.

```
openSpeed = 2
closeSpeed = 2
```

To change the speed at which the door opens or closes, change these variables. The numbers are in percents/frame, so anything between 0 and 100 is valid.

```
open = 0
close = 0
```

These are the messages that cause the door to open and close. See the chapter on messages to get an understanding on how these variables work.

```
didOpen = 0
didClose = 0
```

The door sends these messages when it closes or opens completely. Useful for making perpetually moving doors.

```
openDelay = 0
closeDelay = 0
```

Once a message is received, the door waits for a certain number of frames before it starts to open or close.

```
status = isClosed
```

This is the initial state of the door. (isClosed = 0, isOpen = 1)

```
openSound = 400
closeSound = 400
stopSound = 401
volume = 15
```

The door can make a sound when it is opening, closing or stopping. The numbers are resource numbers for 'HSND' resources. You should define a sound loop for the opening and closing sounds so that the sound plays as long as the door is moving.

```
guardDelay = 5
```

When a door bumps into something, they move back a little and then try to resume their motion again after a few frames. You can control how many frames they move back by setting the guardDelay. If you don't want the door to move back, set it to 0. But remember that this can cause the player to become stuck between a door and a solid with no other way out except to self-destruct!

```
shotPower = 0
```

Normally doors act like polite elevator doors should: if you

step in their way, they'll kindly move back and forth until you are no longer in the way. If you set shotPower to something other than 0, the object that is in the way of the door will get a jolt similar to a mine blast of shotPower going off at the location of the door. This pushes the obstacle away and does some damage too.

Door Door2

This is a slightly modified version of the regular door. In addition to an "open" and closed" state, an intermediate position is also defined.

```
middle = closed
```

This is the middle position. A useful setting would be something like 0.5, but if you keep it at closed, you can actually change the orientation of the door when it is in the closed state.

```
midX = 0
midY = 0
midZ = 0
```

These variables define the position of the door in its middle state. The default matches a closed door.

```
midPitch = 0
midYaw = 0
midRoll = 0
```

These variables define orientation of the object in its middle state. The rotations are applied in the order they are listed here.

Door2 WallDoor

This is a slightly modified version of the Door2 object. It takes the last wall that was added in the game and uses that shape and location as the door shape and location. Note that the y parameter is added to the existing location of the wall, making it unnecessary to use the "wa" variable with the wall itself.

[placed] Switch

This is a solid actor with two possible shapes. Remember that you can set the shields value so that after a certain amount of damage from hits, the switch will destruct! (Or it might destruct if you hit it too hard.) If the switch destructs, it will of course not toggle.

```
shape = bspSwitchOff
altShape = shape+1
```

If altShape is 0, only one shape will be loaded and the switch will not change appearance when toggled.

```
togglePower = 0
```

You can set a minimum missile energy level to toggle a switch. Zero means that any missile hit will toggle it.

```
blastToggle = 1000
```

This is the power that is necessary to toggle the switch when the damage is caused by an explosion. If this value is set high enough, exploding objects (such as grenades, guided missiles and mines) can not be used to toggle the state of the switch. The default value is large enough to prevent the switch to be toggled in most cases. Lower this parameter only if you want explosions to affect the switch.

`status = false` — This is the initial status of the switch.

`hitSound = snSwitch`
`hitVolume = 2` — This sound is played when the switch changes state. If you want two different sounds, set sound to zero and use two separate sound objects.

`restart = true` — If restart is false, the switch can change state only once.

`out = 0`
`out[0] = out`
`out[1] = out` — The out[0] message is sent when the switch turns from on to off (from altShape to shape) and the out[1] message is sent when it turns from off to on (from shape to altShape). To set both messages at once, just modify the "out" variable.

`in = 0`
`in[0] = in`
`in[1] = in` — The message in[0] turns the switch off, in[1] turns it on. If in[0] is equal to in[1], the state is toggled each time a message is received. If the switch doesn't change state when a message is received, no outgoing message is sent. You only need the in messages to control a switch under program control.

Note: — If the switch states have different bounding volumes, it is possible to block a switch so that it doesn't change state when it actually should. In this case, the switch changes state as soon as it becomes unblocked. The default ON/OFF switches do not have this behavior, because their bounding volumes are identical and thus they can not become blocked.

[placed] Teleporter — Teleporters transport players between distant locations. The player enters a teleporter and if there is another one in the destination group that is vacant, is transported there.

`group = 0` — Each teleporter belongs to a group (note that you can use @name to name groups, similarly to the way messages can be named). The group is a destination address and can be shared between several teleporters.

`destGroup = group` — If a player enters the teleporter and a teleporter from this group number is available, the player is transported to it. If several teleporters of this group are available, the one that has been used least will be selected. Usually this means that if two players enter a teleporter, they end up in different locations if there is more than one free teleporter in the destination group.

`shape = bspTeleporter` — This is the default teleporter shape. If you want an invisible teleporter, set the shape value to 0 (zero).

`sound = snTeleporter`
`volume = 10` — A transported player makes a sound while traveling. The sound actually moves from the source to the destination,

	so at very long distances, the doppler shift can be very severe if the sound is short.
`speed = -15`	By default, the teleporter is rotating at this angular speed (in degrees) per frame. At this speed, the teleporter makes a full rotation every 24 frames. Setting the speed to 0 can help performance, but will not look quite as nice with the default shape.
`fragment = true`	For effect, the player leaves behind an imploding set of fragments and arrives with a small set of exploding parts. These of course slow things down, so if you want you can disable this option. Note that the sending teleporter controls the effect at both ends.
`spin = true`	For effect and disorientation, the player emerges from the other end spinning around. You can disable this option by setting the variable to false.
`win = -1`	The player who enters a teleporter with a 'win' setting greater or equal to 0, wins the level and gains as many points as defined by 'win' . It is usually a good idea to use a different shape or color for the win teleporter or mark it otherwise (with a hologram, maybe). If you do not change the group variable to something else than 0, players may end up teleporting right to the end teleporter, since the end teleporter can act as a receiver just as well as any other teleporter.
`mask = allTeams`	You can prevent certain teams from being transported from a teleporter (and from winning by entering a 'win' teleporter) by changing this mask. To allow only teams 1 and 2 to use this teleporter, `mask = T1+T2`

`start = @start`
`stop = 0`
`status = 0`
	You can activate and deactivate teleporters with messages.

`isAmbient = false`	Teleporters can be used to make rotating holograms. You simply set start to 0 (to disable the teleporter), group to something you do not use (I suggest -1) and the shape and speed according to your liking. Set isAmbient to true, if the shape is not essential to game play.
`activeRange = 0.25`	Teleporters will "energize" the player only if the player is within this range of the teleporter origin. This means that if you set this value to something big (like 2000), all the acceptable players in the game will be transported.
`deadRange = 0`	If the player is within this range, he/she will not be transported. This is useful for forcing the player to play within a bubble. If the player is outside deadRange, but inside activeRange, he/she will be transported to wherever you want. Sort of like an invisible boundary.
`showAlways = false`	Normally the teleporter is only shown when it is active. If this is set to true, inactive teleporters look identical to

active ones.

`hitScore = -1000000` — The "hitScore" parameter has a special meaning for teleporters. You can use it as a minimum entry limit for the teleporter. This way, you can allow the player to enter when he/she has at least "hiScore" points.

`in = 0`
`out = 0` — Unlike "Logic" objects, both the in and out parameter are used as message outputs. The "in" message is triggered when someone is transported into this teleporter and the "out" message is triggered when someone is transported somewhere else. The "out" message is also triggered when someone wins through this teleporter.

[glow] Walker — Biped machine controlled by the user (the HECTOR). By placing these actors in the game, you specify initial positions for players who choose the same team. Alternatively you can also use "Incarnator" objects for placing Walkers, but then you do not have access to the parameters below.

`team = 1` — Specifies the team that the player belongs to. Team numbers range from 1 to 6 with zero reserved for the neutral team. (Zero may not be used for human players.)

`incarnateSound = snIncarnate`
`incarnateVolume = 12` — These define the sound and volume that is used when a player is created or recreated after being destroyed.

`winSound = snWin`
`winVolume = 12` — These define the sound and volume that is used when the player reaches a "win" teleporter and is beamed out of the game.

[placed] Incarnator — Incarnator posts are used to recreated dead or new players. They are not visible, unless you mark the location with a hologram. If no suitable Walker objects can be found for a player at the start of a game, a new walker is created at an incarnator spot.

When players are destroyed, but they have lives left, they are recreated at incarnator spots after a short delay. There should be at least one incarnator spot on every level. Otherwise dead players will remain in limbo forever.

`mask = allTeams` — You can control which teams use a particular incarnator spot. For multiple team games, you could recreate players only in their home base.

`status = true` — By default, incarnators are enabled so that they can be used for creating new players. If you have a level where game play progresses from one point to another, you may

want to enable incarnation points as the player[s] progress through the level.

```
start = open
stop = close
open = 0
close = 0
```

Use the start and stop messages to enable and disable incarnators. Open and close work as well, because the default values of start and stop reference them. (In other words: use the ones you feel more comfortable using…it's a matter of personal preference.)

[actor] Area

Areas are used to watch for players and other actors entering and exiting circular areas. The circle preceding the object definition defines the area to be watched.

```
watch = playerMask
```
What to watch for (similar to Guard).

```
mask = allTeams
```
What teams to watch for.

```
freq = 5
```
How often the area is checked.

```
enter = 0
exit = 0
```
Messages that are triggered on entering and exiting this area.

[actor] Field

Force fields push objects within their range in a certain direction.

```
visible = false
```
If this value is set to true, the shape that defines the area of the force field will be visible. Force fields are invisible by default.

```
shape = 0
```
If the shape value is 0, the last wall created will be converted into the force field shape.

```
deltaX = 0
deltaY = 0.2
deltaZ = 0
```
This is the direction in which the force field accelerates the player. Note that the actual acceleration depends on the mass of the player. Heavier players are harder to accelerate.

```
start = @start
stop = 0
```
You can turn the force field on and off using these messages.

```
watch = playerMask
```
What to watch for (similar to Guard).

```
mask = allTeams
```
What teams to watch for.

```
enter = 0
exit = 0
```
Similar to an Area object, these messages are triggered when something enters this area.

```
speed = 0
```
If you are using a visible hologram as the force field, you can have the hologram spin at "speed" (degrees per frame).

Area Text

Text objects are used to place messages on the console displays of players. They can be activated by the area or a message or a combination of both.

`text = "text"` This is the text that is shown.

`in = 0` This message activates this Text object. If it is zero, anyone entering the area will receive the message upon entering.

`showEveryone = false` Only the players occupying the the area will receive the message. Here's a table of how this works with the "in" message:

showEveryone	*in*	*Effect*
false	0	A player entering the area will see the text
false	set	Players within the area will see the text when the in message is received.
true	0	Everyone will see the message when any player enters the area.
true	set	Everyone will see the message whenever the in message is received.

`restart = 75` This is a delay in frames until this message can be shown again. Use this parameter to prevent showing the same message over and over many times when an in message might be received frequently or the player moves in an out of area bounds.

`wait = 0` This is a delay in frames from the time the message was triggered to the time that it will be shown. If you have a long message to show, you can break it into parts and show the parts with a delay.

`start = @start`
`stop = 0`
`status = false` Enable/disable showing this message by changing the initial status and the messages.

`sound = snTextBlip`
`volume = 0.25` Sound to play for this text message.

`align = centerAlign` How to align the text. Can be: leftAlign, centerAlign or rightAlign.

[placed] Goody Goodies are used to add to the capabilities of a player. It works similarly to a force field: a player that overlaps with the hologram will receive the prize.

`shape = bspGoody` The default shape looks a bit like a mayan pyramid. You can control the two colors used in the shape with the fill and outline color of the arc used to place the goody. **Note the recommended color codes below for each type of goody.** Alternatively, you can use the shapes for different objects, such as bspMissile and bspGrenade to represent various powerups.

`altShape = 0` This shape is shown when the goody is disabled. The default value of 0 makes the goody invisible when it is not enabled.

`grenades = 0` Number of grenades awarded. (yellow)

`missiles = 0` Number of missiles awarded. (magenta)

`boosters = 0`	Number of boosters awarded. (red)
`lives = 0`	Number of extra lives awarded. (white)
`rearShield = false`	Set to true, if a rear shield is awarded. (?)
`shield = 0`	Shield strength awarded. (blue)
`power = 0`	Energy awarded. (green)
`boostTime = 0`	An immediately activated booster is awarded. The booster lasts for boostTime frames. (red)
`hitScore = 0`	Points are awarded for touching this goody. (Goodies can NOT be hit or destroyed.) (black)
`start = 0`	
`stop = 0`	
`status = true`	The goody can be controlled with messages and its initial status can be set with status.
`out = 0`	The goody can trigger a message when it is taken.
`sound = snGoody`	
`volume = 0.5`	This sound is played when the goody is taken.
`openSound = 0`	This sound is played when the goody is enabled.
`closeSound = 0`	This sound is played when the goody is disabled.
`speed = 0`	Similar to teleporters (rotating speed for shapes)
`text = true`	**NOT IMPLEMENTED YET** Show a text message describing the contents of this goody when it is activated.

[placed] Sound

Sound objects can be used for a wide variety of purposes. They can either be tied to a place or you can just play any arbitrary sound at any volume. You can even play music with them.

`y = 10`	Y coordinate of location (X and Z are determined by th last arc segment, exactly like any placed actor).
`isPlaced = true`	This sound is tied to a place. Player movements affect the way the sound is played (left/right balance, phase balance and Doppler shift).
`isMusic = false`	
`isAmbient = true`	Ambient or music sound tracks can be disabled through a menu selection by the user. A user might want to disable ambient sounds to leave more processing power for graphics, to hear non-ambient sounds more clearly or to simply save some memory. If a sound object is in any way non-essential to game play, leave the isAmbient flag set to true. If the obejct plays music that is not absolutely essential to the level, set the "isMusic" parameter to true.
`rate = 1`	Sound playback rate relative to the default rate of the sound resource.
`loopCount = 0`	Sound resources can contain looped sections. This variable defines how many times the looped section is played. A value of 0 is special and means that the default loop count of the sound in question is used.

`sound = 0` — Resource id of 'HSND' resource to play.

`volume = 10` — Sound volume for a placed sound.

`volume[0] = volume`
`volume[1] = volume` — If the sound is not placed in a certain location, you can control the left and right volumes of the sound. Note that for a true 3D effect you should also specify a phase difference. volume[0] is the left side, volume[1] is right. Use lower volume settings than you would for placed sounds.

`phase = 0` — The phase difference only applies to non-placed sounds (same as volume[0] and volume[1]). It has no effect on sounds that have "isPlaced" set to true. A negative phase will place the sound to the left and a positive will place it to the right. The exact number depends on the output sampling rate. Experiment with values between -100 and 100 to see how it works.

`restart = true` — If restart is true, the sound can be played again by issueing a new start command.

These are the message inputs for controlling the sound:

`start = @start` — The start message obviously makes the sound start playing. By default it is set to the @start message, which is issued when the game starts.

`stop = 0` — The stop message causes the sound to stop immediately.

`kill = 0` — The kill message causes the sound to stop and sets restart to false so that another start message will not activate it.

Logic

Logic objects are not visible, but allow you to combine messages in strange and interesting ways.

[actor] [logic] — An abstract class of objects. They share the out[#] variables and restart flag

```
out = 0
out[0] = out
out[1] = 0
out[2] = 0
…
out[9] = 0
```

When a logic object activates, it can send up to ten messages out. To send the @openSouthWestDoor message, you would declare:

```
out = @openSouthWestDoor
```

`restart = true` — If this variable is set to true, the same logic object can be used more than once. When the conditions are triggered and messages are sent out, the logic object is reset to its original condition.

in = 0

in[0] = in
in[1] = 0
in[2] = 0
…
in[9] = 0 — You can have up to ten input messages in a logic object.

[logic] Timer — Counts frames from activation and sends message when the timer expires. Note that a new incoming message can be used to postpone an active timer. (For instance, you can create a door that stays open as long as a switch is triggered regularly.)

`wait = 16` — Wait for 16 frames before triggering. A negative timer value indicates that the timer should start counting immediately.

[logic] Delay — Similar to Timer, but several messages can be in a sort of pipeline. If a timer receives more than one message before it activates, it will activate after "timer" frames from each activate message. One Delay logic timer can have up to 32 delayed messages queued.

If you need a one-shot timer, set restart to false. New messages that arrive after the first one will no delay the triggering any further, because they will be queued.

`wait = 16` — Delay in frames.

[logic] And — All defined in[#] ports have to become true before this logic object activates.

[logic] Counter — Counts incoming messages. A message on any input port is counted and when the count is reached, the out messages are sent.

`n = 1` — Count to active. With this setting and the restart value set to "true", the Counter object effectively acts as an "or" gate.

[logic] Distributor — Each activation activates a different out[#] message. The first time it's out[0], then out[1], out[2] and out[3].
Any undefined (0) out[#] messages are simply skipped. At least one has to be defined, of course. If the restart option is true, (it is set by default) after out[3] has been triggered, the next one is out[0] again.

[logic] Base — This is a very special logic object. It has a location, so it uses the 'y' and 'baseHeight' variables and it has to be associated with an arc segment. When it receives an 'in' message, it sends an immediate message for each 'out' that has been defined. In this respect, it's a lot like an "Or" object, so it can be used as such. Ufo objects can detect a

message coming from a "Base" object and they will change their home base to the location of the base object.

The idea is that you can make a Ufo object travel a route or change destination by using Base objects combined with the "homeBase" variable of the Ufo.

Adjusters

Adjusters are a special class of object. They are not actually kept in the game, but they are used to change various game parameters.

[placed] **GroundColor** The last arc segment drawn before this object was created defines the color of the ground for this level.

[placed] **SkyColor** The last arc segment drawn before this object was created defines the color of the sky for this level.

`n = 8` The horizon is shaded with this many colors. The colors range from the arc frame color (sky) to the fill color (horizon).

`verticalRangeMin = 0`
`verticalRangeMax = 1000` The shading range is controlled by these two parameters. You may have to increase either one of the values, if your level has very high places and you want to avoid the effect of painting the ground with the horizon colors.

Here's how you use these two adjusters:

```
adjust GroundColor end
```

```
adjust SkyColor end
```

[actor] **[yon]** Yon limiter objects are special objects that do not have an effect on other game objects. Instead, they control the distance that is visible from within them. The primary reason for them is that if you have a level with tight spaces and insides, you can change the visible distance within a room or area to a smaller value to optimize drawing speed.

Avara rendering is significantly slowed down by walled areas, because the objects behind the walls also need to be considered for drawing. If the yon distance is limited, the number of objects considered for rending is diminished and performance is improved.

You can overlap multiple yon objects and the lowest yon value will then be used in the overlapping area.

`y = 0` — Yon objects are placed using either thick rectangle or circle shapes and their altitude on the map is defined by y + baseHeight. (Just like normal objects)

`yon = -15` — Negative values are treated so that the actual yon bound is an extension of the area's diameter. So a sphere with a 10 meter activeRange (or radius) would have a yon value of 10+10+15 = 35 meters by default.

[yon] YonSphere — This object is created in very much the same way as the area object, except you can use the range variable if you want. The idea is that when the viewpoint is in this sphere, the view distance is controlled by the "yon" variable of this object.

`activeRange = 0` — If you set the activeRange to something other than 0, it will be used instead of the circle radius.

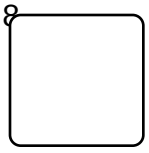
[actor] YonBox — Similar to the YonSphere, but with this object you can define a box-shaped area for the yon limiter. You use a thick-bordered box to define the area. Use a pen thickness of 3 points, for best results. (It's similar to a Ramp object, except it doesn't need the arc that ramps use)

deltaY = 0 — If deltaY is zero or negative, the rectangle corner rounding radius is used as the height of the yonBox. If the rectangle corners are unrounded, deltaY is "infinite" and the box area has no top or bottom limit.

Other Variables

Walls are created with rectangles and rounded rectangles. By default, they are all placed on ground level, but you can adjust the level of individual blocks by setting the `wa` variable *before* the wall is created. After each wall block, this variable is reset to zero. Here's an example:

```
wa = 1.8
```



Note how the text is *under* the rectangle. It doesn't have to be located underneath the rectangle, but it has to come before the rectangle in order to have an effect on that rectangle.

The variable wallHeight is used to control the height of walls that are made with regular (non-rounded) rectangle. It applies to any walls that are created after the parameter has been changed.

The `baseHeight` variable is added to the altitude of all walls and is not reset after each wall. If you want to have a set of walls at a certain altitude, change wb to that altitude and change it back to zero after the walls have been created. Note that it also affects all objects that inherit from the "placed" class, so if you change the baseHeight, it will affect practically all visible objects.

The variable wallShield sets the shield strengths for all the walls that are defined after the change. A negative shield value is indestructible. This is similar to the "shield" parameter for regular objects with the difference that once changed, it does not reset back to a default value, so you have to change it back yourself. The "wallPower" variable sets the blasting power of a wall. When a wall is destroyed, it behaves like a mine of "wallPower" power.

The variable "wallYon" is used to control the clipping distance of walls. See the discussion of the "yon" parameter for regular actor objects.

Walls also have traction control with the variables wallTraction and wallFriction. See the section on traction control variables for more information.

Level Variables and Special Messages

Level variables do no affect a particular object, so they do not need to be inside object definitions. They affect the whole level. Here's the list of variables:

```
// Lights (default light settings are stored here)
ambient = 0.4
light[0].i = 0.4  light[0].a =  45  light[0].b = 20
light[1].i = 0.3  light[1].a =  20  light[1].b = 200
light[2].i = 0.0  light[2].a =  45  light[2].b = 90
light[3].i = 0.0  light[3].a =  45  light[3].b = 180
```

There are four posible light sources in addition to ambient light. To place a lightsource, give it an intensity (the i component) between 0 and 1.0. The two angles indicate the direction that the light is coming from. The b angle is the compass reading and the a angle is the angle from the horizon. All lights are monochromatic white.

Ambient light is not directional. For the most natural lighting, use some ambient light along with at least one or two directional lightsources. Directional lightsources cause a significant amount of calculations to be done, so in most circumstances you only want to use two. Using only one directional light source makes objects look quite dull on the side that is in the shadow.

For each level, you should set the level designer name using the designer variable and the level information using the information variable:

```
designer = "Inigo Montoya"
information = "You killed my father - prepare to die."
```

There are a few special special messages in Avara. The @start message that is issued at the start of a level. It was originally designed to start ambient sounds, but can be used to start timers and all kinds of interesting logic constructs. If you need a clock, you would use a delay:

```
object Delay
   in[0] = @start
   in[1] = @clockLights
   out[0] = @clockLights
   delay = 15   // about one second
end
```

If you need to send messages to objects in a sequence, use a distributor with the above clock:

```
object Distributor
   out[0] = @turnOn1
   out[1] = @turnOn2
   out[2] = @turnOn3
   out[3] = @turnOn4
   in[0] = @clockLights
end
```

The @win message is issued when in a multiplayer game the last player is left standing. This is useful for a deathmatch where everone fights everyone else. @winTeam is issued when only one team is left to battle. This is useful for team battle where you want th message to be triggered when a team has annihilated all other teams.

The following variables control the special player weapons:

```
grenadePower = 2.5
missilePower = 1.0
missileTurnRate = 0.02
missileAcceleration = 0.2
```

The following variables control the maximum amounts of weapons that a player may carry at the start of a level:

```
maxStartGrenades = 20
maxStartMissiles = 10
maxStartBoosts = 5
```

The "lives" variable controls how many lives players have at the start of a level. The default value is 3.

"gravity" controls the gravity on a level. The default value is 1.0.

"friendHitMultiplier" is used when a player hits an object belonging to the same team. The default value is -1, meaning that the points are deducted from your score instead of being added. For a game where everyone is fighting everyone else, regardless of team, use a multiplier value of 1.

Target Groups and Object Masks

Ufos and Guards can be made to target different kinds of objects, belonging to various different teams. The default is that they belong to the neutral (computer) team and shoot players of all other teams. The "watch" variable determines what types of objects are to be shot at. It can be a combination of: playerMask, scoutMask, robotMask, targ1, targ2, targ3, targ4, targ5 and targ6. Guards, Ufos and Parasites and other moving automatons are robots, so they have the robotMask set. Players will have the playerMask and scouts will have the scoutMask set. The "targ#" masks exist so that you can create your own classes of targets for robots to shoot at. For instance, you can have a destructible solid that a group of Ufos is trying to demolish while the player's task is to protect the target. For instance:

```
object Solid
  targetGroup = targ1
  shape = bspShapeShip
  shield = 10
  killMsg = @playerLoses
end

object Ufo
  watch = targ1 + playerMask
end
```

The above will create a Solid that looks like a spaceship and can be destroyed and a Ufo that is interested in destroying the spaceship or any player within its range.

Note that the "mask" parameter can be used to make robot objects take sides. For instance, to make a Ufo that plays on the green side:

```
object Ufo
  mask = allTeams - T1
end
```

The above Ufo will shoot at any players not belonging to the green (team 1) team.

Traction Control Variables

There are three pairs of traction control variables:

```
defaultTraction
defaultFriction

wallTraction = defaultTraction
wallFriction = defaultFriction

traction = defaultTraction
friction = defaultFriction
```

The default settings control how the bare ground behaves, so if you want to change only the

ground, change the default values as the last thing on the level description. If you want to change all objects along with the ground, you can change the default settings as the first thing on the level.

The "traction" variable represents a speed difference between the legs and the structure that a HECTOR is standing on. Speeds slower than "traction" exhibit static friction, which means that the leg is not sliding on the surface. Anything faster and the legs will start to slip (or slide) and the friction variable is used instead.

possible values for traction and friction:

Material	Traction	Friction
Slippery ice	0.05	0.05
Smooth steel	0.20	0.10
Ground (default)	0.40	0.15
Rough ground	0.40	0.40

You can set the traction or friction variables to zero, but do no set both of them to zero or the player may have a lot of difficulty getting off the slippery surface. (Of course this could be done on purpose and you would have to use missiles or grenades for propulsion.)

The highest legal value for the friction variable is 1.0, although in practice you should never exceed 0.75. The smallest legal value for friction is 0. Using a negative value may produce interesting, but totally unrealistic results. Traction values can be anything, although values below 0 will be treated the same as 0. The practical range for both values seems to be from 0 to around 0.4.

3D Modeling for Avara

You can use your own 3D models in Avara levels that you create. To do so, you have to convert your 3D model files into 'BSPT'-resources. Two separate applications are provided for this purpose.

The first step is creating a 3D model. There are several considerations that you have to be aware of. Most importantly, you can't just use any 3D shape and expect Avara to be able to perform well. You have to learn to keep the number polygons low.

Many 3D renderers allow you to use smooth shapes to describe your objects. These smooth shapes have to be converted into often large numbers of flat faces, because Avara can only draw polygons. The BSPViewer application is useful for determining how many polygons an object uses. Modeling programs often allow you to control the smoothness of the objects that are exported into DXF file format.

Once you have created a 3D object that you would like to use in an Avara level, you have to export it to a format that the BSPSplitter application can use. BSPSplitter currently understands a subset of DXF files and the OFF .geom file format. Since DXF files provide very poor means to describe the colors of an object, the layer names in the DXF file are

mapped into colors. For instance, StrataStudio Pro uses the layer names to store the names of the shapes. This means that all you have to do is to group each color into a separate shape and name that shape. You then map that name in the ColorLib text file into an RGB color or a set of colors (see the provided sample ColorLib file for details).

The next step is to create a 'BSPT' resource with BSPSplitter. Select the level directory file you are using as the destination file, choose a suitable resource number between 1000 and 2999, make sure that the ColorLib file is chosen and that the right 3D geometry file is chosen and then process the geometry file into a 'BSPT' resource. For simple objects, processing is done quickly, but complicated objects can take a long time. If processing takes more than a minute or two, your object probably has too many polygons for it to be realistically used in Avara. You can switch out of the BSPSplitter program while it is processing the object or you can interrupt the process at any time by holding down the option, shift and control keys simultaneously (this will quit the application).

You can then drop your level directory file into the BSPViewer application and look at the BSP resource. Use the left and right arrow keys on your keyboard to scan through the 'BSPT' resources in that file, type in a few letters from the beginning of your model name or type the resource id number that you used.

The most important function of BSPViewer is backface removal. A polygon has two faces: a front face and a back face. For most polygons created with a program like "StrataVision", only the front face should be visible, which is what the default is for shape resources created by BSPSplitter. In some cases, you want both sides of the polygon to be visible or you may want the backface (inside) to be visible. BSPViewer allows you to get some control over what faces are visible.

Hit command-0 (zero) to clear the visibility flags of all faces. This makes the screen go blank, so have the object somewhere where you can see it. If you hold down the shift key, BSPViewer will draw all the faces regardless of their visibility status. **Important:** if you hit space, BSPViewer finds out which faces and which sides of the faces would be visible if you held down the shift key and adds those sides to the visibility flags. If you rotate the object while holding down shift, all the faces visible from the outside of the object become visible.

If you want only the front faces to be visibile, hit command-1. If you want backfaces to be visibile, hit command-2. To have both sides of all polygons visibile, hit command-3.

Remember to save the resource (with command-S) after you are done with it and pleased with the results. If you don't like what you have done, start over (with command-0 or command-1, usually) or go back to the saved version (just close and open the file again or choose another shape without saving).

The up and down arrow control the "roll" of the object. Combined with shift, option or command, the step of rotation increases.

Clicking and dragging moves the object further or closer (up is further, down is closer).

3D Wall and Floor Template Models

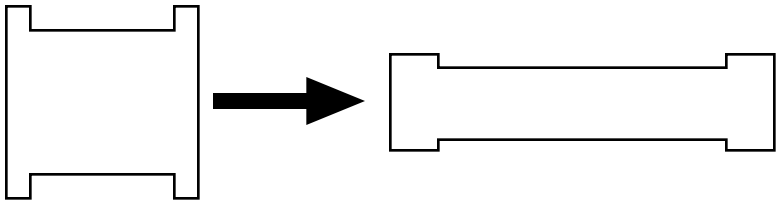
This is a very advanced topic. If you are just starting level design for Avara, you can probably either skim through or skip this chapter.

Normally walls are represented by simple 3D boxes and thin walls (with height 0) are rectangles. You can change this behavior by changing the variables wallTemplate and floorTeamplate. Their default values are:

```
wallTemplate = bspStandardWall
floorTemplate = bspStandardFloor
```

In addition to the obviously needed 'BSPT' resource, you can optionally also create a 'BSPS'. A template for this resource type is provided with Avara. It defines the outside dimensions of the template. A wall template should be cubical in shape. The default cube extends 0.5 meters in all directions, thus the "size" in the template is 0.5. A floor template should be "square". Note that you can round the corners, if you want, but the aspect ratios of the shapes should always be 1:1:1 and 1:1. You can make the floor templates thick: the y direction is always left unscaled/unstretched.

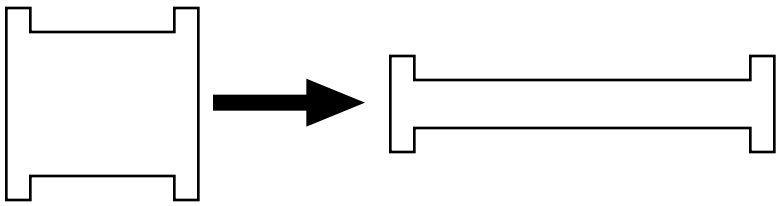
Templates can either be scaled or stretched. Scaling produces results like this:



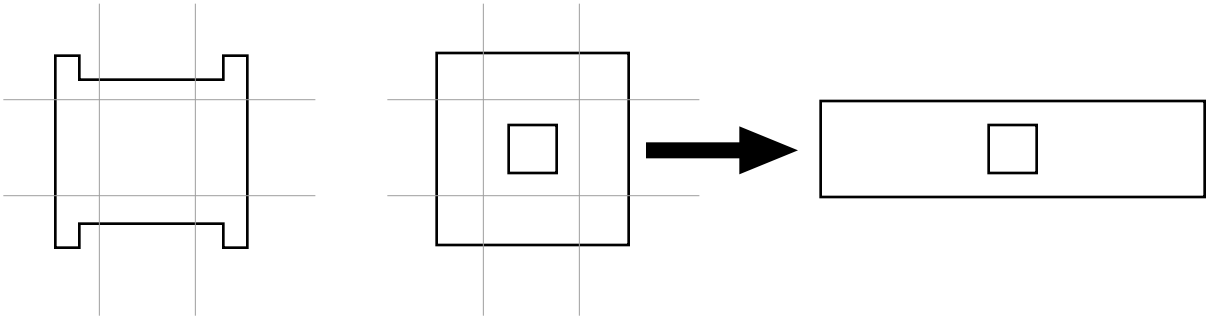
Scaling works for practically all BSP shapes, so it is used by default. You can not however use it to make boxes or rectangles with constant width borders. You have to use the more limited "stretching" mode for that.

Stretching takes the points that are half the dimension away from the origin and offsets those to make the template the desired size. This imposes some very strict limitations on the shape of the template, because Avara is unable to change the surface normals of the 3D faces when this method is used. For this reason, try to keep the normals of all the surfaces axis-aligned or test your shape properly to see that it draws and shades correctly from every position.

Stretching the same shape above to the same dimensions as above produces the following result:



Note how the ends stay the same width as in the template. The following illustrates how stretching works: the areas inside the dotted lines are not touched during a stretch:



As shown this allows you to have an unchanging shape inside the stretched area.

Appendix: Avara Definitions

This script is loaded before every level. It contains all the built-in variable declarations and constants for Avara levels.

```
//          Avara variables and constants

// Do not make changes to this resource. To use the
// Variables described here, redefine them in your
// level files. Making changes here will probably
// cause severe compatibility problems. Treat this
// resource as a read-only guide to the variables and
// constants that are available for designing levels.
//
// This resource has to end with a space or return.

  designer = 0
  information = 0
  timeLimit = 2400 // Default time limit is 30 minutes.

  gravity = 1
  customGravity = 0.12

  shape = 0     // Primary BSP shape resource id
  altShape = 0  // Possible alternate shape
  scale = 1
  yon = 0
  wallYon = 0
  hasFloor = 0

  wallTemplate = 400
  floorTemplate = 401

  y = 0           // Height from ground level.
  mask = -1       // Bitmask for object type
  team = 1        // Team id
  wallHeight = 3
```

```
wa = 0
baseHeight = 0
pixelToThickness = 1 / 8
mass = 0
visible = 0
thickness = 0

winTeam = 0

targetGroup = 0

// Sound stuff:
hitSound = 0
hitVolume = 0
hitSoundDefault = 210
shieldHitSoundDefault = 211
playerHitSoundDefault = 211
blastSound = 0
blastVolume = 0
blastSoundDefault = 230

stepSound = 161
groundStepSound = 160

isTarget = 0

sound = 0
openSound = 400
closeSound = 400
stopSound = 401

volume = 15

// Player related:

defaultLives = 3
lives = defaultLives

incarnateSound = 0
incarnateVolume = 12

winSound = 0
winVolume = 12

loseSound = 0
loseVolume = 6

// Scoring:
killScore = 0
hitScore = 0
friendlyHitMultiplier = -1

// Damage and explosions
smallSliverCount = 0
mediumSliverCount = 0
largeSliverCount = 0
```

```
smallSliverLife = 0
mediumSliverLife = 0
largeSliverLife = 0
canGlow = 1

// Messages sent when hit/destroyed
killMsg = 0
hitMsg = 0

// Other messages
stepOnMsg = 0

// Motion
accel = 0

// Toggle switches
togglePower = 0
blastToggle = 1000

// Doors:
open = 0
close = 0
didOpen = 0
didClose = 0
openDelay = 0
closeDelay = 0
guardDelay = 0
status = 0
openSpeed = 0
closeSpeed = 0
pitch = 0
yaw = 0
roll = 0
deltaX = 0
deltaY = 0
deltaZ = 0

middle = 0
midPitch = 0
midYaw = 0
midRoll = 0
midX = 0
midY = 0
midZ = 0

// Misc:
power = 0
maxPower = 0
drain = 0

// Guards:
fireMsg = 0
trackMsg = 0
stopTrackMsg = 0
speed = 0
shotPower = 0
```

```
// Ufos:
checkPeriod = 0
attack = 0
defense = 0
visionScore = 0
hideScore = 0
motionRange = 0
pitchRange = 0
verticalRangeMin = 0
verticalRangeMax = 0
burstLength = 0
burstSpeed = 0
burstCharge = 0

homeSick = 0
homeRange = 0
homeBase = 0

// Mines:
shield = 0
activateEnergy = 0
activeRange = 0
phase = 0
activeTimer = 0
boom = 0 // start mine timer message

idleShapeTimer = 20
idleAltShapeTimer = 0
activeShapeTimer = 4
activeAltShapeTimer = 4

activateSound = 0
activateVolume = 0

// Teleporters:
group = 0
destGroup = 0
spin = 1
fragment = 1
win = 0
deadRange = 0
showAlways = 1

// Activators: (areas, etc.)
watch = 0
freq = 0
enter = 0
exit = 0

// Text
text = 0
showEveryone = 0
align = 0

// Goody
```

```
grenades = 0
missiles = 0
boosters = 0
boostTime = 0

// Ball & goal
goalMsg = 0
goalAction = 0
goalScore = 0

ejectPitch = 0
ejectPower = 0
ejectSound = 0
ejectVolume = 0

shieldChargeRate = 0
maxShield = 0
shootShield = 0
grabShield = 0

carryScore = 0
dropEnergy = 0
changeHolderEnergy = 0
changeOwnerTime = 0

changeOwnerSound = 0
changeOwnerVolume = 0

snapSound = 0
snapVolume = 0

// Sounds:
isMusic = 0
isAmbient = 1
isPlaced = 1
rate = 1
loopCount = 0
volume[0] = volume
volume[1] = volume
start = @start
stop = 0
kill = 0

// Logic:
in = in[0]
in[0] = 0    in[1] = 0    in[2] = 0    in[3] = 0    in[4] = 0
in[5] = 0    in[6] = 0    in[7] = 0    in[8] = 0    in[9] = 0

out = out[0]
out[0] = 0     out[1] = 0     out[2] = 0     out[3] = 0     out[4] = 0
out[5] = 0     out[6] = 0     out[7] = 0     out[8] = 0     out[9] = 0

restart = 0
n = 0
wait = 0
```

```
  // Lights (default light settings are stored here)
  ambient = 0.4
  light[0].i = 0.4  light[0].a =  45  light[0].b = 20
  light[1].i = 0.3  light[1].a =  20  light[1].b = 200
  light[2].i = 0.0  light[2].a =  45  light[2].b = 90
  light[3].i = 0.0  light[3].a =  45  light[3].b = 180

  // Advanced weapon powers:
  grenadePower = 2.25
  missilePower = 1.0
  missileTurnRate = 0.025
  missileAcceleration = 0.2

  maxStartGrenades = 20
  maxStartMissiles = 10
  maxStartBoosts = 5

  // Hull types
  hull[0] = 0
  hull[1] = 0
  hull[2] = 0

  defaultTraction = 0.4
  defaultFriction = 0.15

  wallTraction = defaultTraction
  wallFriction = defaultFriction

  traction = defaultTraction
  friction = defaultFriction

  wallShield = -1
  wallBlast = 4

           dummyVar = 0

//   some defines
  true = 1
  false = 0

  playerMask = 1
  scoutMask = 2
  robotMask = 4
  collisionDamageMask = 128

  targ1 = 256
  targ2 = 512
  targ3 = 1024
  targ4 = 2048
  targ5 = 4096
  targ6 = 8192

  canPushMask = 32768

  allTeams = -1
  T1 = 2
```

```
  T2 = 4
  T3 = 8
  T4 = 16
  T5 = 32
  T6 = 64

// Doors:
  isClosed = 0
  isOpen = 1

// Text alignment
  rightAlign = -1
  centerAlign = 1
  leftAlign = 0

//         bsp resources...doesn't mean you should use them all:
  bspAvara = 100
  bspMissionComplete = 101
  bspAvaraA = 102

  bspGrenadeSight = 200
  bspGrenadeSightTop = 201
  bspMarker = 202
  bspShot = 203
  bspDirInd = 204
  bspTargetOff = 205
  bspTargetOn = 206
  bspSmartMissileHairs = 207
  bspSmartMissileSight = 208

  bspHECTORBoundBox = 210
  bspHECTORLegHigh = 211
  bspHECTORLegLow = 212
  bspHullLight = 215
  bspHullMedium = 216
  bspHullLarge = 217
  bspScout = 220

  bspTeleporter = 230
  bspGoody = 240

  bspStandardBall = 250
  bspGoal = 251
  bspStandardPill = 252

  bspSphere = 300

  bspMine = 310
  bspMineActive = 311

  bspStandardWall = 400
  bspStandardFloor = 401

  bspW1x1 = 411
  bspW2x1 = 421
  bspW2x2 = 422
```

```
bspW3x1 = 431
bspW3x2 = 432
bspW3x3 = 433
bspW4x1 = 441
bspW5x1 = 451

// Sphere segments
bsp16SphereNF = 460
bsp16Sphere = 461
bsp4SphereNF = 462
bsp4Sphere = 463
bsp2SphereNF = 464
bsp2Sphere = 465
bsppSphere = 466

bspSliver0 = 500
bspSliver1 = 501
bspSliver2 = 502
bspSliver3 = 503

bspDoor = 550

bspSwitchOff = 560
bspSwitchOn = 561
bspWallSwitchOff = 562
bspWallSwitchOn = 563

bspGroundStar = 600
bspGroundArrow = 601
bspGroundArrowLeft = 602

bspVines = 610
bspCrack = 611
sbpTree = 1002

bspTriPyramid = 650
bspOnSwitch = 701
bspOffSwitch = 702
bspTower = 703
bspGrid10 = 704
bspMushroom = 705
bspHingeDoor = 706
bspFlower = 707
bspTree = 708
bspGrid7.5 = 709
bspHill = 710
bspStreet = 711
bspTurn = 712
bspLock = 713
bspDeadTree = 714
bspBigIce = 715
bspIce = 716
bspShell = 717

bspCubeFrame = 720
bspDoubleCube = 721
```

```
  bspFloorFrame = 722
  bspGobbleRect = 723
  bspGobbleTriangle = 724

  bspGuard = 800
  bspBolt = 801
  bspMissile = 802

bspLargeDome = 806
  bspParasite = 807
  bspUfo = 808
  bspShuriken = 809
  bspPlatform = 812
  bspGrenade = 820

  bspStarFighter = 830
  bspTractorTower = 831
  bspShooter = 832

// HSND (sound) resources:
  snUnderwater = 129
  snJungle = 130
  snArcticWind = 131
  snBubbles = 132
  snBirds = 133

  snTextBlip = 150
  snMessageSend = 151
  snMessageReceive = 152
  snStep = 160
  snShot = 200
  snDoorClang = 210
  snHit0 = 210
  snHit1 = 211
  snShieldHit = 220
  snShieldHit1 = 221
  snShieldHit2 = 222
  snBlast = 230
  snBlast1 = 231
  snSwitch = 240
  snGoody = 250
  snMineBleep = 300
  snMineBlow = 301
  snParasiteBlow = 310
  snDoor = 400
  snDoorStop = 401
  snTeleport = 410
  snIncarnate = 411
  snWin = 412
  snLose = 413
  snParasiteAttach = 420

  snBallBuzz = 430
  snBallSnap = 431
  snBallEject = 432
  snBallReprogram = 433
```

```
  snGobble = 440

  lightHull = 128
  mediumHull = 129
  heavyHull = 130

// Misc adjustments
  hitSoundDefault = snHit0
  blastSoundDefault = snBlast
  shieldHitSoundDefault = snShieldHit
  playerHitSoundDefault = snShieldHit2
  incarnateSound = snIncarnate

  hull[0] = lightHull
  hull[1] = mediumHull
  hull[2] = heavyHull

designer = "unknown"
information = "

No additional information on this level is available."

unique 32000 end
```