Blades Of Avernum Scenario Editor Instructions

Table of Contents

The appendices (which list all the calls, spells, skills, and other designer resources) are in the file Blades of Avernum Editor Docs Appendices.

Introduction to Scenario Design

Welcome to the Blades of Avernum Scenario Editor! The Blades of Avernum system enables you to make fantasy role-playing adventures of great detail and complexity. And, hopefully, fun. It is a demanding program, and creating the next great adventure is not for the faint of heart. However, with time and ingenuity, you can create stories that will captivate Blades of Avernum fans all over the world.

Using the editor is a long journey. So, before you start, here are a few words of advice, intended to make your job a lot easier:

If You Are a Beginner, Read the Tutorial

If you've never used this scenario editor before, be absolutely sure to read Chapter 1.1, "Making Your First Scenario, Step By Step". It's the very first chapter. It will take you, step by step, through all of the most important features for scenario design. For the novice, it is absolutely invaluable.

Play the Game!

Don't even think about starting to use the editor until you have played through all of the scenarios that come with Blades of Avernum. Before you can work in this medium, you have to really understand how the game works. You should be comfortable with how the game flows, how the terrain works, and what the game system is like. That is the only way you can really understand what you are doing.

Don't Expect Miracles

I won't lie to you. You can create very basic, simple scenarios with this editor without a lot of work. However, you will have to put in a lot of hours if you want to figure out how to make this editor truly shine. Scripting is powerful, but it is not always easy.

Expect Limitations

The Blades of Avernum system is not created to be infinitely versatile. You can't create arcade elements. It won't lend itself well to making science fiction adventures, or adventures in settings far from Avernum. Also, you are sure to think of something extremely clever you want to do that the engine doesn't make possible. I tried to anticipate everything, but I am sure I failed many, many times.

It Is Work

People are often surprised at how much work and concentration is takes to make adventures. It takes time to make good stuff. I strongly suggest, for your first adventure, aiming small. A few outdoor sections. Ten or twenty towns and dungeon levels. Don't try to make the world-changing epic until you really understand what you're getting into. It is better to release and share one small adventure than to make half of a huge one that nobody ever sees.

Learn From Examples

The best way to learn how to do anything is to work from an example. If, for example, you want to see how to place a locked door, open one of the scenarios that came with the game and look at the doors in it. If you want to see how to make scripts, read the scripts in scenarios already completed. Do not be ashamed of looking at the work of others. It is the very best way to learn.

Back Up Your Work!

I have lost count of the times that some poor user has relayed to me a tale of a crashed machine, file corruption, or similar problem that destroyed their scenario (along with many hours of work). Computers are machines, and machines can fail! Back up your work frequently!

Use Online Resources!

Scenario design is complicated, but you don't have to face the struggle alone. Go to the Scenario Workshop at http://www.avernum.com to read helpful tutorials on scenario design, and go to the Blades of Avernum forum on our web site (http://www.spiderwebsoftware.com) to ask questions and trade tips.

Designing role-playing games can be really fun, exciting, and satisfying. I've been doing it for ten years, and I'm only just getting started. So if you want to give it a shot and see what people really think of your work, read on. If I haven't scared you off ... let's go!

- Jeff Vogel
Keeper of Avernum

Section 1: The Basics

Chapter 1.1: Making Your First Scenario, Step By Step

This chapter contains step-by-step and click-by-click instructions for making a scenario, editing the outdoors a little, making a new town, and populating the new town with some creatures and treasure. Going through these steps will take only a few minutes, but save hours of confusion later on.

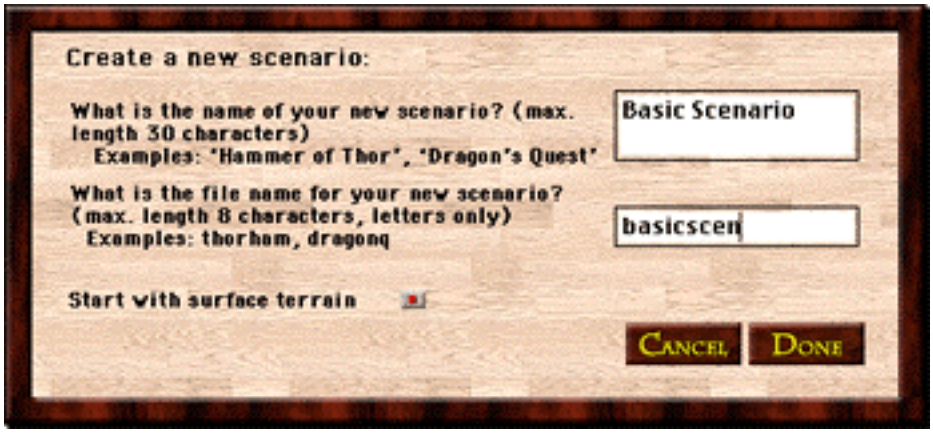
I. Get the editor running.

Copy the Blades of Avernum Editor application into the "Blades of Avernum Files" folder. Open that folder and run the editor.

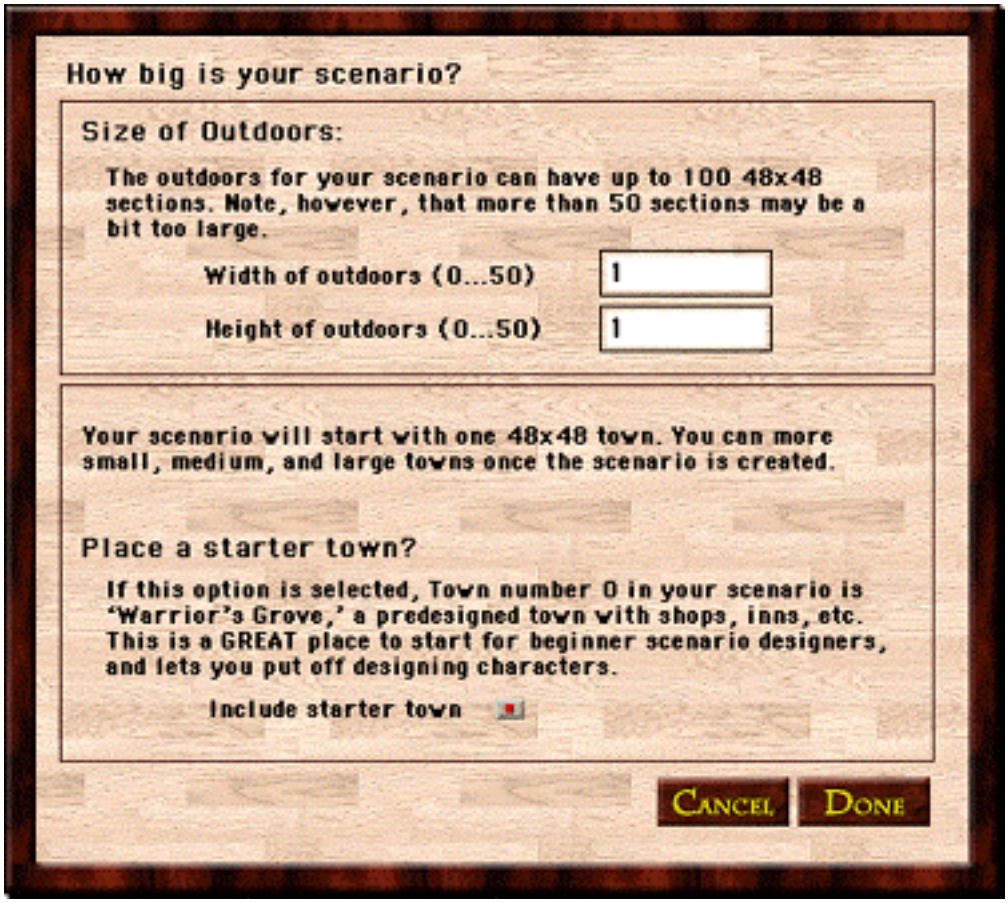
II. Make your scenario.

Select New Scenario from the File menu. You will see a window that asks for your new scenario.

In the first text area, just enter some name. Try "Basic Scenario". In the next text area, you must enter the file name for the scenario. Just enter "basicscen". Finally, select the bottom check box to have the scenario be on the surface (instead of the underworld). Press Done.



The correct setting for the first window.

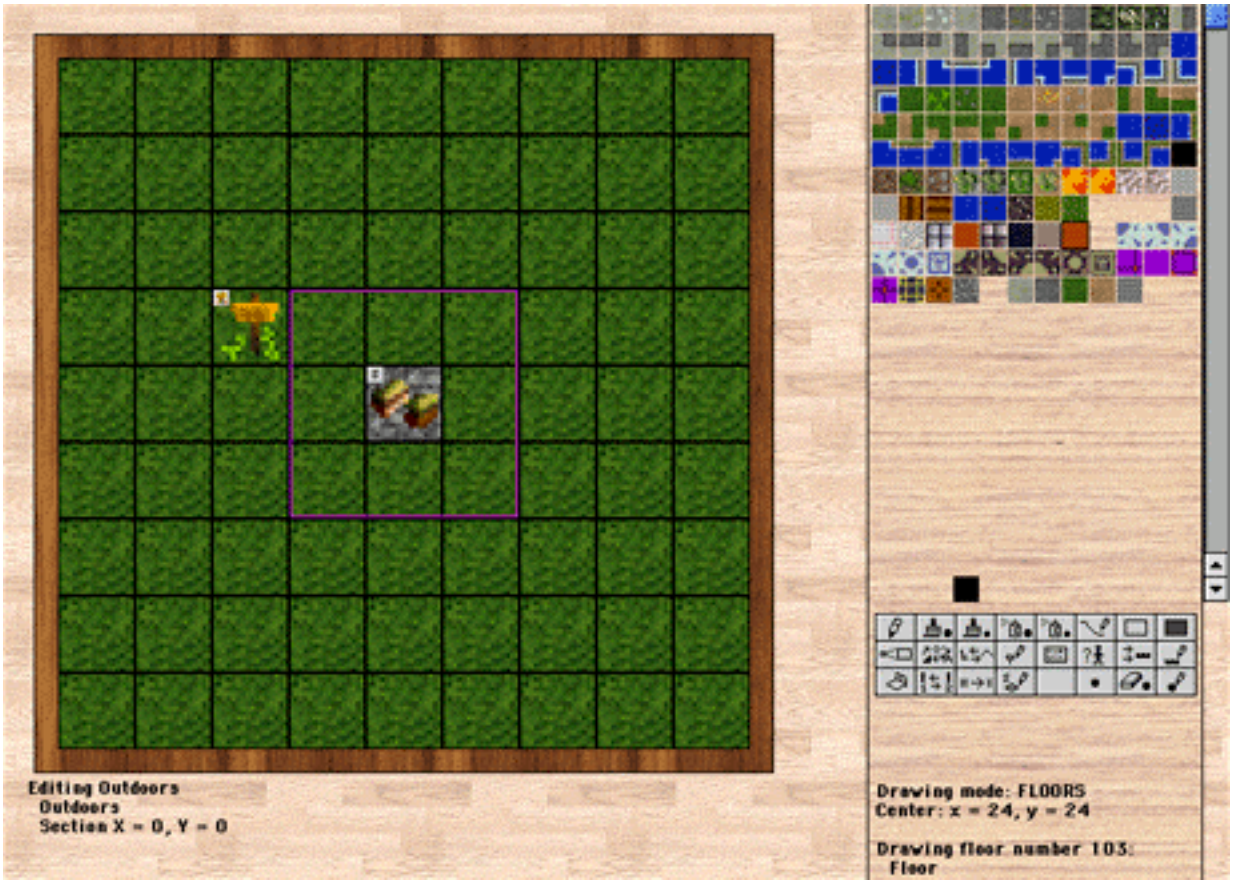


The correct setting for the second window.

In the next window, you will be asked for the size of the Outdoors. Leave the fields saying one and one. But below, be sure to select the checkbox by "Include starter town." Press Done, and your scenario will be created.

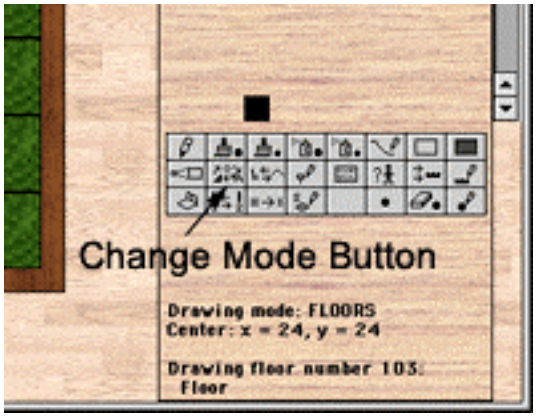
III. Open your scenario.

Select Open from the file menu. Open the folder "Basicscen" in the "Blades of Avernum Files" folder. The file basicscen.bas is your scenario file. Open it. You will now be looking at the outdoors. That town in the middle is Warrior's Grove, a simple starting town that has been provided for you.



The Outdoors When You Begin

Warrior's Grove is a full service town, with shops, an inn, and a way to leave the scenario. You don't have to be able to program or write scripts to enable players of your scenario to have access to the resources they need. So for now, we can just focus on painting a little terrain and making a dungeon.



The Change Mode Button

IV. Shifting the View. Painting Terrain

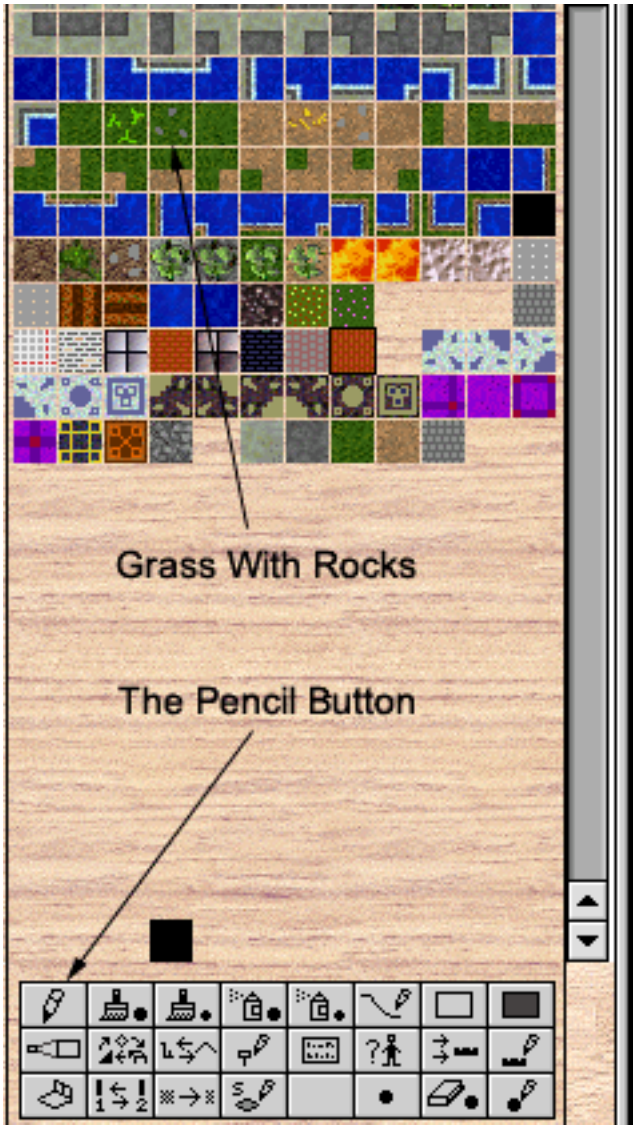
First, let's shift our view a little. Click on the wooden edge of the terrain view. Note how the view scrolls around. You can also shift the view with the keypad. Move the view around and little, and then return to the view being centered on the main town.

The editor has three modes: place floors, place terrain, and change heights. You start out in place floors mode. All those icons to the upper right are the available floor types.

Press the change mode button. You are now in place terrain mode. Notice how all of the icons to the upper right have changed to terrain.

Press the change mode button again. You are now in change height mode. This isn't important right now. Press the change mode button one last time to return to place floor mode. (You can also change the mode by hitting the space bar.)

Let's change the floor. First press the Pencil button and then select grass with rocks (see figure below).



The Pencil and Grass

Click on the terrain a few times. You are placing rocks on the grass. Click on some of the rocks again. Notice they disappear, returning you to the base ground (grass). Placing a terrain onto the same sort of terrain a second time erases it.

Now change to drawing terrain mode. The party will start in this town, but we don't want them to be able to walk off the edge of the world. We want to draw trees around the town, so the party has to stay close. Select trees by clicking on them, and then draw trees around the town, like this:

A Ring Of Trees

Finally, we want to make a dungeon to explore. Scroll the list of terrains down until you see the Stairs Down terrain (see below). Click on it and then place it by the edge of your trees. We're ready to make a dungeon.

V. Save your scenario.

Select Save from the file menu to save your work. Do this often. You should also back up your scenario frequently, but we aren't quite there yet.

VI. Making a new town.

Select Create New Town from the Scenario menu. We want to make a new dungeon to explore. Set the fields in the Create Town window like this:

The Create New Town Window

We are making a town called "Icky Dungeon." It is the default size, 48 x 48 spaces. And uncheck the box by "On surface?". We want this dungeon to be underground. When ready, press Done.

The town Warrior's Grove was the first town in our scenario's town list, so it is town 0. Our new town is town 1.

Now you need to tell the game that this pit is the entry to town 1. Press the Create town entrance button (shown below). You will need to select a rectangle that is the entrance to the town. Click on the pit once to mark the upper left corner of the rectangle, and then click on the same spot to indicate the lower right corner. You have made the one space containing the pit the entrance for your dungeon. You will be asked what number of town this is the entrance to. Enter 1.

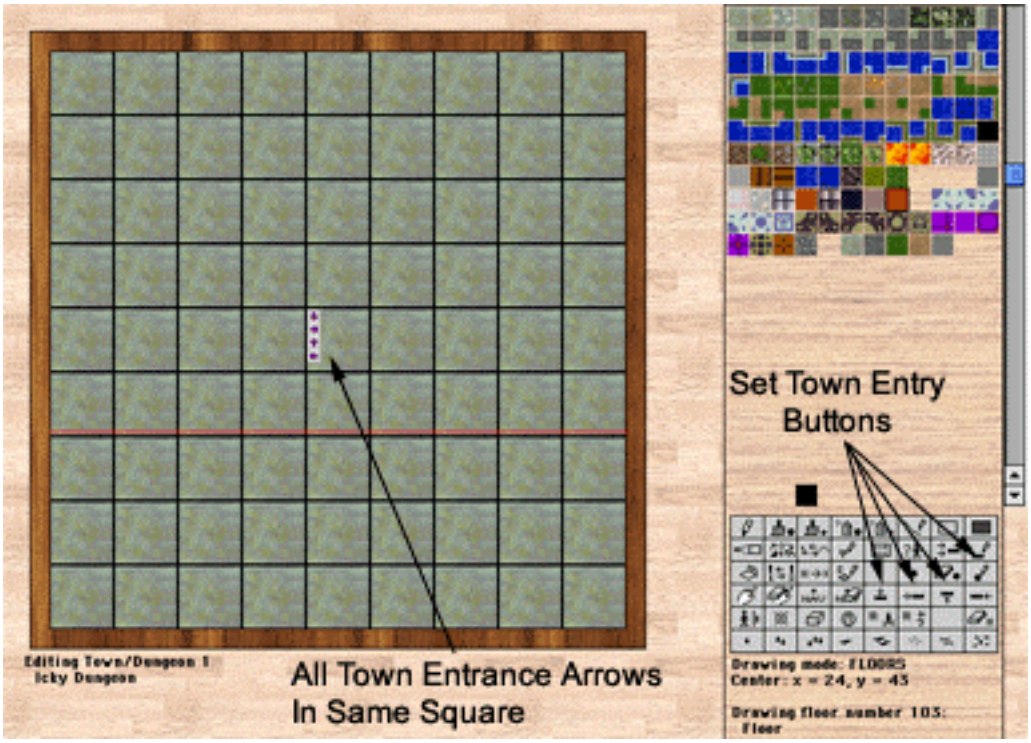
The Pit Is Now a Town Entrance.

To switch to editing the town, select Edit Town from the Scenario menu.
You will be staring at a blank cave floor. Time to create some corridors and some occupants.

VII. Set the Party's Starting Location.

First, you need to set where the party is placed when they enter this town. There are four different directions the party can enter a dungeon from. Entering from each location can put them in a different place. You set the locations where the party appears using the Place entrance buttons.

For now, we will keep it simple. Scroll all the way straight down to the bottom and place all four entrances in the same place. Press each Place Entrance button and click on the space with the arrow.

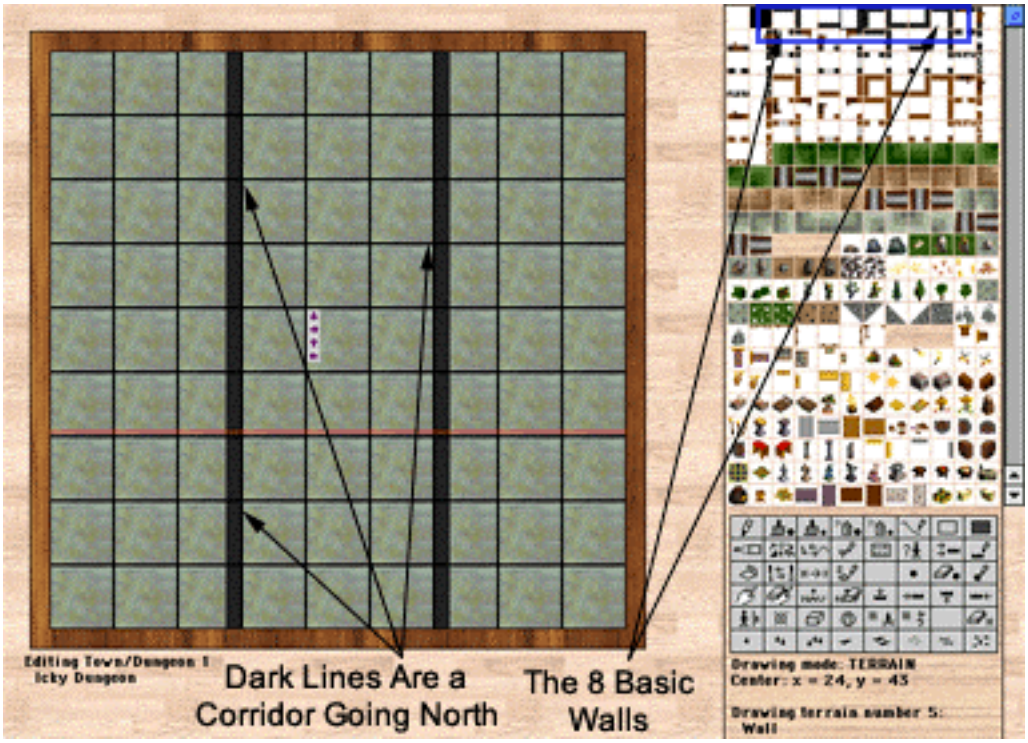


Put All Entrances In the Same Place

VIII. Paint Walls to Make a Corridor

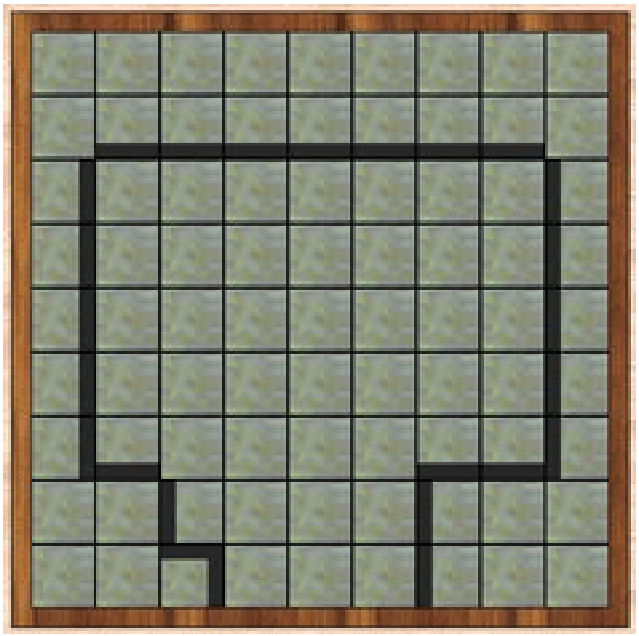
Switch to drawing terrain mode. Look at the terrain in the top row. The second through tenth icons, in particular. Those are basic walls. There are four straight walls, along the north, west, south, and east of the space. Then there are 4 L-shaped walls.

You want to paint some basic walls to make a corridor for the party to walk in. Select the walls and draw a corridor to the north, like this:



The Walls, and the Corridor

Then scroll north a little and make a rectangular room.



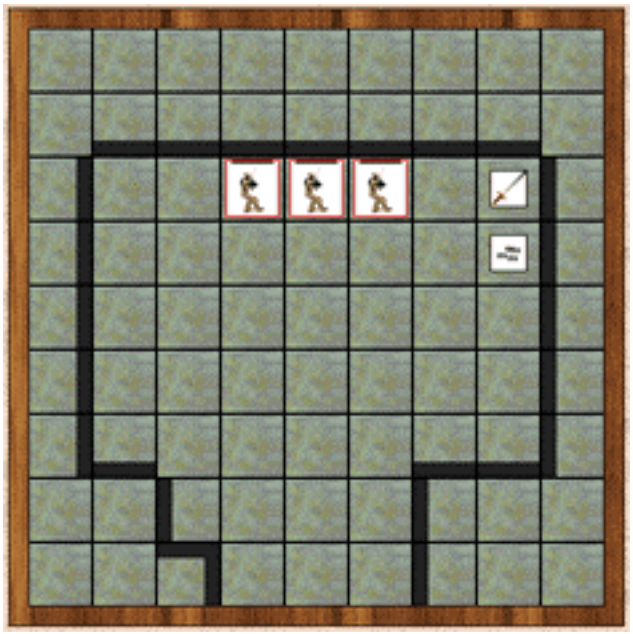
Your Room

Note that the corners of the room are not filled in. That is OK. The game will fill them in automatically.

IX: Place Some Monsters and Treasure

Let's place some foes! From the C1 menu, select "42-Skeleton" and then click on the north edge of the room. Do this 2 more times, selecting different spaces. There are now 3 nasty skeletons in the room.

Now lets place some treasure for the party to find. From menu I1, select "2 - Silver Coins". Click on a space in the upper corner of the room. Select "50 - Stone Short Sword." Click on the same space in the terrain.



Monsters and Treasure

There! A genuine encounter for the party to fight through! Now that we have a scenario, let's try it out. Select Save from the File menu and quit the editor.

X. Playing Your Scenario

Move your "basicscen" (you can change the name if you want) folder into the "Blades of Avernum Scenarios" folder. Run the game, load a party, select Enter Scenario, and select "Basic Scenario."

Select Basic Scenario

You're in! Leave Warrior's Grove, enter your dungeon, and fight some skeletons! You're playing a scenario you made!

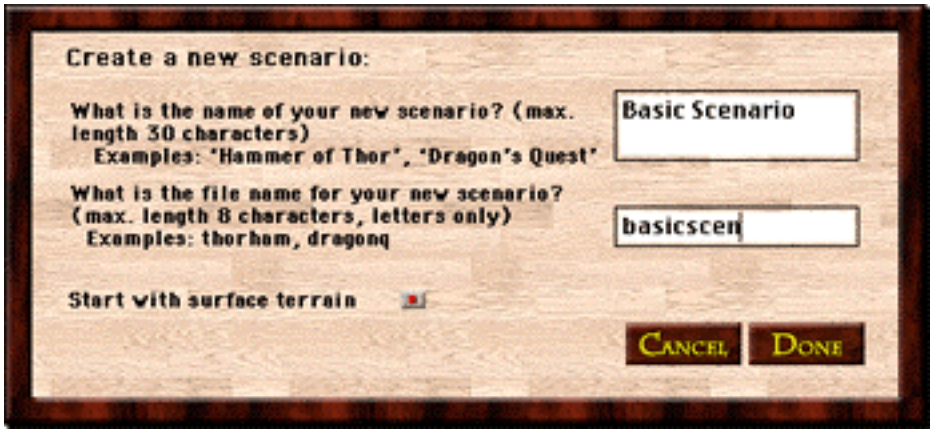
And There It Is!

Play a bit more. Make your dungeon bigger. Add more monsters and treasure. Maybe even make a second, nastier dungeon somewhere else. You already have the tools to have a lot of fun. When you're comfortable with what you are doing, return to these instructions and dig in to the really exciting stuff.

Chapter 1.2: Creating A Scenario

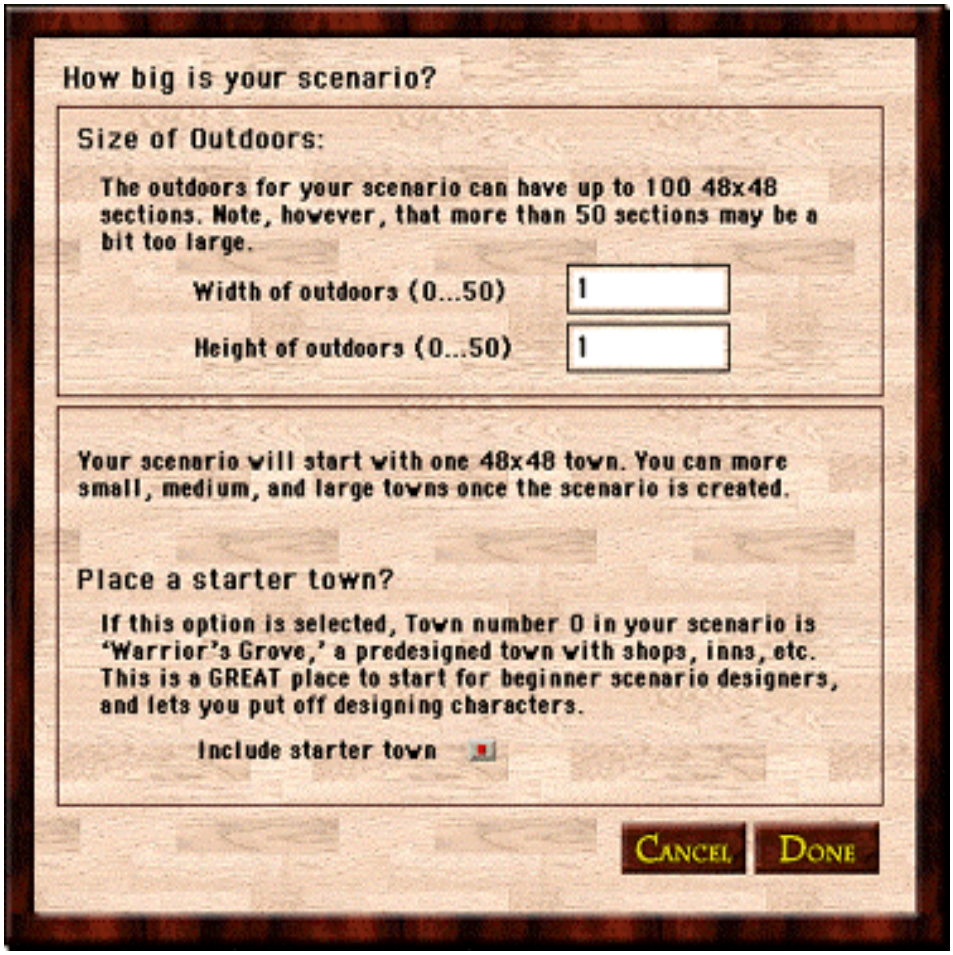
Upon running the editor, your first step will most likely be to create a new scenario. (If you have made scenarios using Blades of Exile, you might want to import an old scenario. If so, read Section 3.)

To make a scenario, select New Scenario from the File menu. You will be presented with a series of windows where you provide information about your new adventure.



In the first field, start by entering the name of your new scenario. This is its actual name, in regular text (eg "Valley of Dying Things", "A Small Rebellion"). In the second field, enter the file name of the scenario. This is the name the actual scenario file will have, and can have no spaces or other punctuation (eg valleydy, rebellion).

Finally, if you want your starting terrain to all be on the surface, click on the check box at the bottom. When ready, press Done to bring up the second window.



Now you provide the size of the outdoors (width by height) in the first two fields of this window. This is an important choice, as you CAN NOT change the size of the outdoors later. It is recommended that you make the outdoors a little bigger than you think you'll need. You can always not use sections later. The outdoors can be at most 100 sections.

Finally, you can opt to have your scenario start with a town called Warrior's Grove. This is a basic town with an inn, merchants, and so on. Having Warrior's Grove in your adventure means that you don't need to be able to make a town (with all the scripting this involves) to create an adventure. If you want to have Warrior's Grove in your adventure (a very good idea for beginners), select the appropriate check box. If you don't make Warrior's Grove, your scenario will start with a blank 48x48 town (which you don't have to use if you don't want to).

When ready, press Done.

Finally, in the third window, press Done to make your scenario. It will be created in a folder in the same folder as the editor. The scenario file itself will be inside this folder, and the file name will have .bas appended to it (so valleydy becomes valleydy.bas). You can change the name of the folder, but you have to leave the name of the scenario file itself alone.

To test this scenario, you will need to drag it into the Blades of Avernum Scenarios folder. Once your scenario is in place, you are ready to use the editor.

Chapter 1.3: A Few Things To Know

Before learning about the editor, there are a few things it is helpful to know to avoid confusion.

Relearning How To Count

In Blades of Avernum, everything is numbered starting with 0. So if you have 10 towns, they are numbered 0 to 9. If your outdoor section is five sections wide and six high, the sections are numbered (0 .. 4) by (0 .. 5). The 256 creature types are numbered 0 to 255.

Outdoor Sections and Towns

A Blades of Avernum section contains outdoor sections (each 48x48 spaces) and towns. The outdoors is split up into a grid of these sections.

Town is a generic term for any outdoor area, be it a town, ruin, or dungeon. Towns can have 3 sizes: 32 x 32, 48 x 48, and 64 x 64. Most towns in Blades scenarios are 48 x 48. When you walk off the edge of a town, you will be in the outdoors.

When editing a scenario with the Blades of Avernum editor, the editor holds one town and one outdoor section in memory for you to edit. When you want to switch to editing a different section/town, there are menu options to load the new one into memory.

Basic Terminology

When you are playing the game and a little window comes up with text or something to do (a window that describes an area or asks you to select a character), this is called a dialog box. Dialog boxes are referred to frequently in these instructions, since you may well want to display them to the player from time to time.

In a dialog box, you may be asked to enter a value into a little edit area (enter the name of an area, for example, or the number of a town). A little rectangle where you enter information is called a Field.

Chapter 1.4: Using the Editor

When you are ready to get to work, you will need to load your scenario and start editing its basic traits. To get started, you need to use the File menu, described below.

File Menu Options

Open - Brings up a file selection window. Select the scenario you want to edit. (The file name will have the format [name].bas.)
Save - Save your work in the current scenario. Use this often.
New Scenario - Makes a new scenario. Described in detail in the previous chapter.
Import Blades of Exile Scenario - Described in detail in section 3.
Quit - When you are done.

Once you have opened your scenario, you can do three things inside the editor: edit a town, edit an outdoor section, and edit the overall scenario properties. (Other things you may want to edit, like dialogue, creature behaviors, custom terrain types, are edited in scripts, which are written using a text editor or word processor. More on this in Section 2.)

The editor has loaded in memory at any one time one outdoor section and one town. You can switch between editing the outdoor section or the town and load in new sections/towns. To switch between editing modes or edit overall scenario properties, use the Scenario menu.

Scenario Menu Options

Edit Town - Begin editing the town currently in memory.
Edit Outdoor Section - Begin editing the outdoor section currently in memory.
Create New Town - Creates a new town, appending it on the end of the scenario's town list. (So if the scenario has 20 towns, numbered 0 to 19, the new town will be number 20.) Selecting this brings up a window:

The Create Town Window

In the fields, provide the name of the new town, select its size, and use the bottom check box to determine whether the starting terrain will be on the surface or underground.

If you have made any changes to your scenario, you will not be allowed to use this command until you save them.

Basic Scenario Details - Enables you to set the description that will be visible when people see your scenario in their scenario selection window. You can set the version number (starts at 1.0), the credits (viewable in the game), the description, the suggested level range (which is only a suggestion, the player can take any party into any scenario), and the rating.

Note that only the first bit of descriptive text is currently used. Also, if your descriptive text is too long, it may not all be visible on the scenario selection window. Be sure to test it.

Set Label Icon - Sets the 64 x 64 icon that will be next to your scenario's name in the scenario selection menu. If left at 0, gives a default adventure icon. A list of the possible adventure icons is in the appendices.

Set Intro Text 1,2,3 - When the adventure is entered by the player, you can have him or her see up to three screens of text. Enter that text in these windows. Each window can have up to six paragraphs, and each paragraph can be up to 256 characters long.

Each screen of text can be accompanied by a custom graphic (provided by you) at the left side of the screen. Enter the sheet number of that graphic in the top field. If left at -1, no graphic is drawn. If you aren't sure about what you're doing with custom graphics, leave this field alone.

Reload Scenario Script (Advanced) - When editing a scenario, if you change the scenario's custom objects script, you can select this to have the new changes loaded into the editor. For more information on custom objects scripts, read the section on scripting.

Clean Up Walls (Advanced) - In the section on editing terrain, you will learn how certain configurations of walls look better than others in the game. This command has the editor go through the walls in your town/outdoor section and reshape the walls to make them look nicer. This command is safe to use ... it will not mess up the walls or overwrite any other terrain.

Import Town - This command imports a town from a different Blades of Avernum scenario into the town you are currently editing. You will be prompted for the number of town and then asked to select a scenario, and the town in that scenario will be copied over the current town in the scenario you are editing.

The scenario you are editing and the scenario you import from can be the same. The town you import and the town you are editing must be the same size.

Import Outdoor Section - This command imports an outdoor section from a different Blades of Avernum scenario into the outdoor section you are currently editing. You will be prompted to select a scenario file and then an outdoor section, and the new section will be loaded over the section in the scenario you are currently editing.

Set Variable Town Entry (Advanced) - When the party enters a certain town, you can shift the number of the town they end up in using this option. If you enter a town number in the first field of a row in this window and the two coordinates of a SDF in the second and third fields, when the party enters that town, the game adds the value of the SDF to the town number and that is the town the party ends up in.

For example, town 7 might be the town of Dunghill, and town 8 might be the same town after orcs have destroyed it. If you enter 7, 15, and 3 in the first row of fields and set stuff done flag (15,3) to 1, when the party enters town 7, they will be placed in town 8 instead.

Edit Item Placement Shortcuts (Advanced) – If there are certain sorts of items you place in particular terrains a lot (like copper coins and clothes in dressers), you can create sets of items that are randomly put on those terrain types when you select Add Random Items from the Town menu.

Selecting this option brings up the Editing Item Shortcuts window. Choose a terrain type in the field at the top. Select up to 10 items and assign to each one a percentage chance it is placed on the terrain type. You can create up to 10 shortcuts.

Delete Last Town - Eliminates the highest numbered town from the scenario. You can only use this option when the scenario has more than one town and you are editing a town besides the one you are deleting.

Write Scenario Data To Text File - Creates a file called Scenario data and dumps into it a list of all the town's floor, terrain, item and creature types. This file can be a handy reference.

Setting the Starting Location

One thing you will need to do for every scenario is edit where the party starts out when they first enter it. The party always starts somewhere inside one of your towns. There are two steps for this.

First, go to the outdoor section containing the town the party starts in. Select Set Starting Location from the Outdoor menu and then select the location of the town the party starts in.

Then load the town the party starts in. Select Set Starting Location from the Town menu, and then select the location in the town the party starts at.

For more details on editing/loading towns and outdoor sections, read the chapters on Editing the Outdoors and Editing Towns.

Leaving the Scenario

Just as important as a place to start a scenario is a place to leave it. The call end_scenario takes the party out of a scenario. Be sure to have a place (or many places) where the party can get out.

If you choose for your scenario to contain Warrior's Grove, it contains a way for the party to leave the scenario.

Chapter 1.5: Editing Terrain

Once your scenario is created and loaded, you will be looking at the terrain editing screen. You start out editing the outdoors. You can switch back and forth from editing outdoors to editing towns using the Scenario menu.

Each town/outdoor section in Blades of Avernum is divided into a square grid of spaces. Each outdoor section is 48x48, and each town is 32x32, 48x48, or 64x64, depending on the size you selected when you created it. Each space has a floor, a terrain on the floor, and a height (a number from 0 to 99). You will switch from floor editing mode to terrain editing mode to height editing mode frequently when editing your terrain.

To change drawing mode, press the switch mode button or hit the space bar.

Shifting the View

You start out zoomed in on the terrain. You will see only a 9x9 square of the town/outdoor section you are editing. To shift the view, click on the frame around the terrain area or use the keypad (2 is down, 6 is right, etc.). If you hold down the control key while clicking the frame or pressing the key, the view will be moved the maximum distance in that direction.

If you are editing an outdoor section and scroll to the very edge of the view, you will see the edge of the next outdoor section to the north/south/east/west. This is to help you line up roads, rivers, etc. You can't edit the edge of the different outdoor section. If you look at the edge of the whole outdoors (for example, go to one of the outdoor sections on the left edge of the world and scroll all the way to the left), you will see a row/column of cave floor.

Buttons:

1. Pencil
2. Large Paintbrush
3. Small Paintbrush
4. Large Spraycan
5. Small Spraycan
6. Change Heights
7. Hollow Rectangle
8. Full Rectangle
9. Zoom In/Zoom Out
10. Switch Drawing Mode
11. Automatic Hills
12. Edit Sign
13. Create Area Description
14. Set Wandering Monst. Points
15. Create Town Entry
16. Edit Town Entry
17. Place Walls
18. Switch Wall Types
19. Change Terrain
20. Place Terrain Script
21. Create Special Enc.
22. Delete Special Enc.
23. Edit Special Enc.
24. Select/Edit Object
25. Delete an Object
26. Place Waypoint
27. Delete Waypoint
28. Place North Town Entry
29. Place West Town Entry
30. Place South Town Entry
31. Place East Town Entry
32. Place Blocked Space
33. Place Web
34. Place Crate
35. Place Barrel
36. Place Fire Barrier
37. Place Force Barrier
39. Clear Space
40. Small Blood Stain
41. Medium Blood Stain
42. Large Blood Stain
43. Small Slime Pool
44. Large Slime Pool
45. Dried Blood Stain
46. Bones
47. Rocks

The Toolbar

The buttons to the lower right are where you select all the tools you will use to edit your scenario. Only the top three rows of tools are used when editing the outdoors.

The tools/buttons are described below (with keyboard shortcuts, if available).

Pencil - Draw terrain or change the height. When drawing floors or terrain, click on the terrain to place the floor or terrain. If you place a floor/terrain type where that floor/terrain type already is, it will erase it.

When editing heights, click on a square to increase its height by one. Hold down the Command key (if using Macintosh) or the Control key (in Windows) and click on a space to lower its height by one.

Paintbrush (Large, Small) - Draws a circle of the current floor/terrain where you click. Has no effect in height mode.

Spraycan (Large, Small) - Draws a random scattering of the current floor/terrain where you click. Has no effect in height mode. The spraycan does not copy terrain over existing terrain.

Change Height Of Rectangle - Prompts you to click the upper left and then the lower right of a rectangle in the terrain area. You will then be prompted to select a height. Changes the height of all the spaces in the selected rectangle to the given height.

Paint Rectangle (Hollow, Full, Shortcut for full rectangle is Shift-R) - Prompts you to click the upper left and then the lower right of a rectangle in the terrain area. Fills either the edge of or the whole rectangle (depending on which tool you use) with the current floor/terrain type.

If editing heights, you will then be prompted to select a height. Changes the height of all the spaces in the selected rectangle to the given height.

Zoom Out/Zoom In - Switches the view of the terrain from zoomed in (only see a 9x9 square) to zoomed out (see the whole town/outdoor section). You can still edit terrain/floor/heights in this mode, but you can't edit anything else.

Change Drawing Mode (Shortcut is spacebar) - Switches you from editing floors to editing terrain to editing heights. When you are editing heights, the numbers in the terrain spots are the heights of the spaces.

Automatic Hills - When editing heights outdoors, you might wants hills to be automatically placed to reflect how you are shifting the terrain. If you want this, press this button to turn Automatic Hills on, and press it again to turn them off. When on and you change the height or draw new terrain around a place with height differences, the game will automatically draw hills as best it can to create smooth hills.

Be very careful with this feature, because it can erase cliffs and other terrain formations you might have struggled to make. Also, when creating pits or unusually shaped hills, gaps in heights might be created. Check to make sure you haven't created any unwanted pits.

Edit Sign - When you place a sign, press this button and click on a sign to edit the sign text. Sign text can be up to 256 characters long.

If you want to force a new line in sign text, you can do this by including the character "|".

Create Area Description - Prompts you to click the upper left and then the lower right of a rectangle in the terrain area. You will then be asked to give a brief text description of the area (up to 30 characters). This is the description the player will see in the description bar to the side when he/she enters that rectangle.

Area descriptions can be deleted using the Edit Area Descriptions options in the Town and Outdoor menus.

Place Wandering Monster Spawn Point - Each town has six wandering monster spawn points, and each outdoor section has four. Press this button and then select spaces in the current town/outdoor section to place the spawn points. Of course, if you don't set wandering monsters, this has no effect.

Be careful not to place spawn points in walls or completely enclosed areas.

Create Town Entrance - This can only be used outdoors. Prompts you to click the upper left and then the lower right of a rectangle in the terrain area. You will then be asked for a town number. When the party walks into that rectangle, they will be placed in that town. For more details about how the party is placed in the town, read the chapter on editing towns.

It is a bad idea to create town entrance rectangles that touch or overlap. Don't place town entrance rectangles so they touch the edge of their outdoor section. This can cause problems.

Edit Town Entrance - Press this button and click on a town entrance rectangle to change what town it leads into. If you press the Cancel button, it will delete the town entry.

Place Walls (Advanced) - It can be inconvenient and time-consuming to place walls. This tool can help you do it quickly. Prompts you to click the upper left and then the lower right of a rectangle in the terrain area. The editor will then place walls in any space in the rectangle which has the floor type "Blackness" (floor type 255, the last floor type) so that the walls form a ring around the darkness. The walls placed are type 1. If there is no blackness in the rectangle, nothing happens. The editor will only put walls in spaces with no terrain already there.

Switch Wall Types - Prompts you to click the upper left and then the lower right of a rectangle in the terrain area. Switches all walls of type 1 in the rectangle to type 2, and all walls of type 2 to type 1.

Change Terrain Randomly - Enables you to randomly replace floor/terrain of one type in the town/outdoor section with another type. If you are in floor mode when you press the button, you will replace floors. If you are in terrain or height editing mode, you will replace terrain.

Selecting this option brings up the Change Terrain window. You are asked for the number of a floor (or terrain) type to replace, the floor (or terrain) type to replace it with, and the percentage chance of changing each given spot. If you don't know the number of the terrain you want to replace or place, press the Choose buttons. Press the Done button to replace the floors/terrain.

Place Terrain Script (Advanced) - Enables you to create or edit a terrain script. This can only be used in town mode. Press this button and then select a space in the terrain. If there is no terrain script there, a dummy script is placed and selected. You can then edit it. (To edit a terrain script already placed, use the Select/Edit Object button).

Create Special Encounter (Advanced) - Prompts you to click the upper left and then the lower right of a rectangle in the terrain area. You will then be asked for the special encounter id, which is the number of a state in the current town/outdoor section's script. Special encounter Ids should always be from 10 to 100 (states outside that range may be reserved for other purposes). Then, when the party steps in that rectangle, the game will call that state in the script.

Note that a special encounter only takes effect when a character steps from a space outside that rectangle to a space inside it. Walking around inside a special encounter rectangle won't trigger it.

Delete Special Encounter (Advanced) - When pressed and you select a space in the terrain, deletes a special encounter whose rectangle contains that point (if two special encounters overlap over that point, one of them is deleted).

Edit Special Encounter (Advanced) - When pressed and you select a space in the terrain, you can enter a new special encounter id for the special encounter whose rectangle contains that point.

The buttons below are only visible and selectable when editing a town (not outdoors).

Select/Edit Object (Shortcut: Shift-S) - Prompts you to click on a placed object (i.e. an item, creature, or terrain script). When you select an object by clicking on it, its properties and options for editing it will appear in the lower left corner. Click repeatedly on a space to cycle through the objects on it. Select the options to the lower left to edit the object. This is described in more detail in the section on editing towns.

You can only use this command when the view is zoomed in.

Delete An Object (Shortcut - Delete key) - Prompts you to click on a placed object (i.e. an item, creature, or terrain script). Deletes it. If there are several objects in the space, they are all deleted.

Place Waypoint (Advanced) - Each town can have up to 8 waypoints. These are fixed, preset locations which can be useful when writing scripts. Select this tool and then a spot in the town to place waypoint.

Delete Waypoint (Advanced) - Select this tool and then a waypoint in the town to delete it.

Set Town Entrance, North, West, South, East - Sets the location in the town the party appears at when they enter it from that direction. For more information on how the game determines what direction the party entered a town from, see the chapter on towns.

Be sure that these points are not placed inside walls, and that enough room is left to place the entire party.

Place Blocked Space - Select this tool and then a space in the terrain to place a blocked space. Characters will not step on blocked spaces. When the game tries to place the party (say, when entering a town), it will, if possible, not place them on a blocked space. It is good to place blocked spaces around secret doors so that a monster standing on one side of the door doesn't keep players on the other side from coming through.

Place Web, Barrel, Crate, Fire Barrier, Force Barrier - Select this tool and then a space in the terrain to place the appropriate object.
Clear Space - Select this tool and then a space in the terrain to delete any blockages, stains, webs, barrels, crates, or barriers in that space.
Place Small Blood Stain, Average Blood Stain, Large Blood Stain, Small Slime Pool, Large Slime Pool, Dried Blood Stain, Bones, Rocks - Select this tool and then a space in the terrain to place the appropriate stain.

About the Floor and Terrain Types

There are a maximum of 256 floor types and 512 terrain types in a Blades of Avernum scenario. When you make a new scenario, it will have the default terrain and floor types (there are far less of these than the maximum number). All of the terrain types and special notes about them are given in the Appendices.

You can create your own custom terrain and floor types. You can learn about this in the section on scripting. However, the basic terrain and floor types are more than adequate to create a good scenario.

There are some special terrain types that bear mentioning.

Floor Type 4, Rough Cave Floor - When drawn, the editor will automatically adjust the edges of this floor so it blends smoothly with cave floor. Use rough floor #7 if you don't want it to be automatically smoothed.
Floor Type 23, Cave Water - When drawn, the editor will automatically adjust the edges of the water so they line up right. Use water floor #87 if you don't want it to be automatically smoothed.
Floor Type 41, Dirt/Desert Floor - When drawn, the editor will automatically adjust the edges of this floor so it blends smoothly with grass. Use dirt floor #44 if you don't want it to be automatically smoothed.
Floor Type 57, Surface Water - When drawn, the editor will automatically adjust the edges of the water so they line up right. Use water floor #87 if you don't want it to be automatically smoothed.
Floor Type 71, Pit - This terrain type is meant to be used for black areas the party can see (like the bottom of bottomless pits). For the blackness where the party can never reach, use floor type 255 (blackness).
Floor Type 80, Fake Lava – The second lava looks scary, but it does no damage.
Floor Types 125-129, Blocked Floors - These function like regular floors, except they are blocked. Other characters can't walk on them.
Floor Type 255, Blackness - Use this floor type to signify areas the party can never, ever reach (the areas between walls, and so on).
Terrain Type 1, Blackness – This terrain is basically an impenetrable black cube. The party cannot see through it or walk through it.
Terrain Type 74-121, Hills - In the editor, the high edges of hills are signified by the lighter edges of the graphic, and the lower edges are signified by the darker edges of the graphic. For example, hill 80 slopes up to the north.
Terrain Type 289-296, Stairs - The arrow on these icons points in the upward direction. For example, stairway 289 is up to the north.
Terrain Type 380-389, Beam Projectors and Mirrors - These are described in more detail in the chapter on towns. Don't use these terrain types until you're sure you know what you're doing. They do not function outdoors.

Painting Hills

Creating changes in elevation and getting hills to line up right can take a while. Fortunately, the editor provides a powerful tool for doing this quickly.

To start changing the height of the terrain, press the Change Drawing Mode (or hit the spacebar) until you are in the height drawing mode. Numbers will appear on the terrain spaces, each representing the height of the terrain (from 0 to 99). Click on a space to increase its height. Hold the Command key (Control key in Windows) down and click on a space to lower the height.

You can have the editor automatically adjust the hills for you. To get the editor to properly paint the hills, press the Automatic Hills button. Then, when you change the height, the editor will take its best guess of what the hills should look like for this elevation level.

Words About Walls

The terrain types you will place most frequently are walls. There are two sorts of walls. Wall Type 1 is represented by dark gray lines in the editor, and Wall Type 2 is represented by dark brown lines. The two wall types look different in the game. How they appear depends on whether the party is in a town or outdoors.

When outdoors, walls look like stone cliffs if the party is on the surface or blue gray cave walls if the party is underground. Use the Outdoor Details command in the Outdoors menu to set where the outdoor section is. Wall Type 1 is walls that are short (1 icon high) and Wall Type 2 is a wall that is tall (2 icons high).

When indoors, you select what the two wall types look like and how tall they are in the Town Details window. You will enter numbers in the Wall 1 Sheet and Wall 2 Sheet fields that tell the game what the walls look like. The most common wall types are listed at the lower left corner of that window. All available wall types are listed in the Appendices.

Placing walls can be tedious, but the Place Bounding Walls and Swap Walls tools should help.

The different wall types are listed below, in the order they appear in the terrain types palette:

Regular walls (4 straight walls, 4 L's) - Ordinary walls.
Closed Doors - When placed, they are, by default, unlocked. To learn how to make them locked, read the section on doors later in this chapter. Doors do not function outdoors.
Open Doors - Doors that have already been opened.
Secret Doors - When placed in the terrain, a small SD icon will appear. The party can walk through these walls.
Windows - Walls that can be seen through.
Closed Gates, Open Gates - Gates can be closed and opened using scripts.
Crumbling Walls - Function exactly like normal walls, but they can be destroyed by a Move Mountains spell. Be sure not to place them unless you don't mind the party seeing what is on the other side.

One very important note is that not all wall graphics exist for all walls. For example, for the outdoor cliff wall, only the regular wall sections exist. If you place a door outdoors or set a town's walls to cliff wall and try to place doors, the game will only draw static where that door should be.

All wall types and the graphics that are available for them are listed in the Appendices. It's easy to avoid trouble, though. Just use only the wall types 600-605 listed in the Town Details window (these types have all icons available) or use the two outdoor cliff types (614, 616) and only place simple wall sections (or secret doors).

Cliffs

When you create height differences in the editor, the game draws cliffs when appropriate to make the terrain look nice. When outdoors, the game selects the most appropriate cliff for whether the party is on the surface or in the underworld.

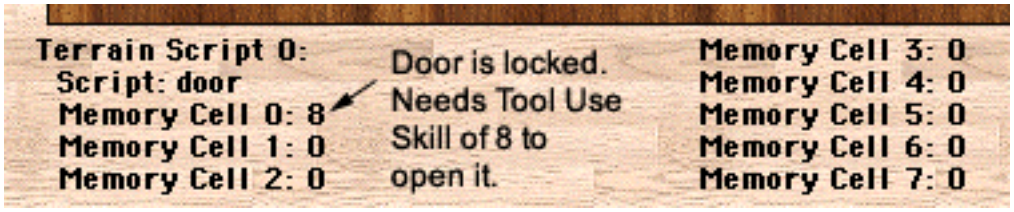
When in a town, you enter a number in the Cliff Sheet field of the Town Details window to pick a cliff. The most common options are listed to the lower left of that window.

Placing and Locking Doors, Intro To Terrain Scripts

When you place an open or closed door in the town terrain (never place doors outdoors), you will see a small scroll icon appear on the terrain square (it will only be visible when you are zoomed in). This scroll means that the door has a terrain script attached to it.

Terrain scripts are complicated, so you shouldn't try to absorb the topic quite yet, but you will need to understand how to edit them to make the door locked.

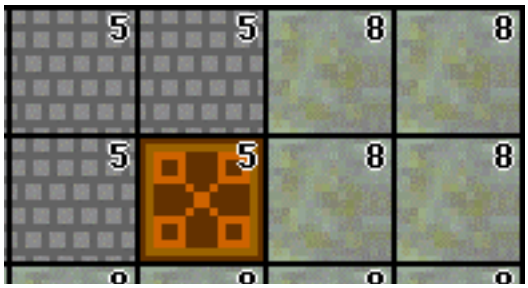
Terrain scripts are files containing text instructions that determine how the terrain spot behaves. For example, when you walk into an unlocked door, the terrain scripts sees you have walked into it and turns the closed door terrain into open door terrain (and plays an opening noise).



Locking a door

To lock the door, first go to the Zoom In view. Then press the Select/Edit Placed Object button and click on the door until you see the area to the lower left has the header "Terrain Script" and a number. You will then see the name of the script (should read "Script: door") and seven Memory Cell entries (all of which are 0). Memory cells are numbers you can set which tell the script how to behave. The script looks at them to see whether the door is locked, and so on.

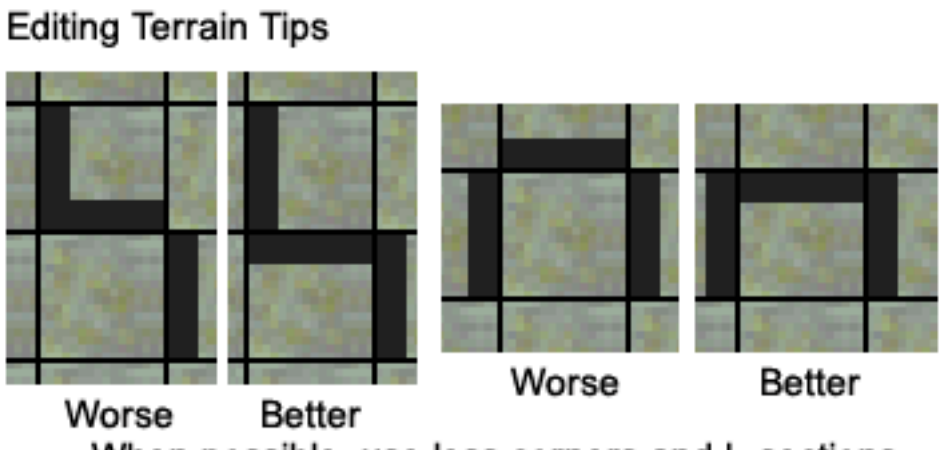
Memory Cell 0 determines whether the door is locked, and how much Tool Use skill is required to open it. If left at 0, the door is unlocked. If you change it to be greater than 0, the door is locked, and the number is the Tool Use skill (plus lockpick bonus) a party member needs to unlock the door.
There are a lot of other things you can do with doors (and you can make your own custom doors that behave in different ways), but they are advanced. They are covered in the chapter on Terrain Scripts.



Tips For Editing Heights

When in editing height mode, each space will have a number on it. This is the height of the space, which can be from 1 to 99. The higher the number, the higher the space.

Blades of Avernum draws areas to the north and west to the upper left, and south and east to the lower right. This means that, if you are placing a cliff, you should put the higher areas to the northwest. Cliffs that are higher to the southeast will probably block the player's view of the action.



Tips For Editing Walls

This figure gives some advice for how to place walls so they look their best in the engine. Some configurations look better than others. For example, the editor draws corners in when you place two walls at ninety degrees

to each other but they don't line up. However, it tends to look a little better if you avoid this. Also, simple, straight wall sections tend to look better than L-shaped wall sections. Use the Clean Walls command in the Scenario menu to have the editor make your walls look a little better automatically.

When editing your terrain, it is important to remember that the Blades of Avernum terrain drawing techniques are designed for simplicity and speed, not taking care of every possible unusual configuration. If you want to use lots of walls on the edges of cliffs, or to create situations where the terrain to the northwest is lower instead of higher, it may take some trial and error to get things to look just right.

Leaving Room For the Party

When outdoors, the party only takes up one space, so it's easy to leave room for them.

Inside, though, the party will most often take up four spaces, and can take 5 or 6 if you make characters that can join the party. Thus, anywhere where the party enters town and anywhere where they can leave a boat needs to have enough space to fit the party.

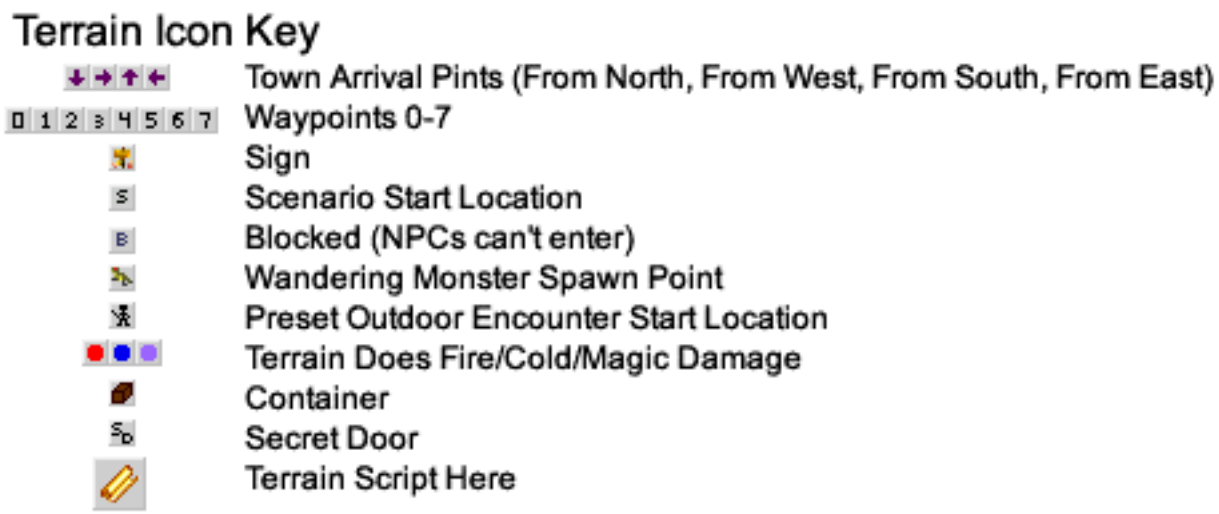
Stain/Feature/Blockage Limit

You can place at most 50 blocked spaces, features (i.e. barrels, webs, etc) or stains in a town. You can't place any of these things in the outdoors.

Boats and Horses

A scenario can have up to 30 boats and horses placed in it. However, to create them requires scripting. To learn more, read the section on scenario scripts.

Icons on the Terrain



Here is a key to the icons you will see on the terrain area to tell you what you've just placed.

Chapter 1.6: Editing the Outdoors

Your scenario will have a rectangular block of outdoor sections (maximum number of 100). Each of the outdoor sections is 48 x 48 spaces. The game does not hold the entire outdoors in memory when the user plays. Instead, it will hold in memory the 2x2 square of outdoor sections the party is currently in. The game automatically loads and unloads outdoor sections as the party walks around.

When you load a scenario, you automatically start in editing the outdoors mode. Later, to edit the outdoor section currently loaded by the editor, select Edit Outdoor Section from the scenario menu.

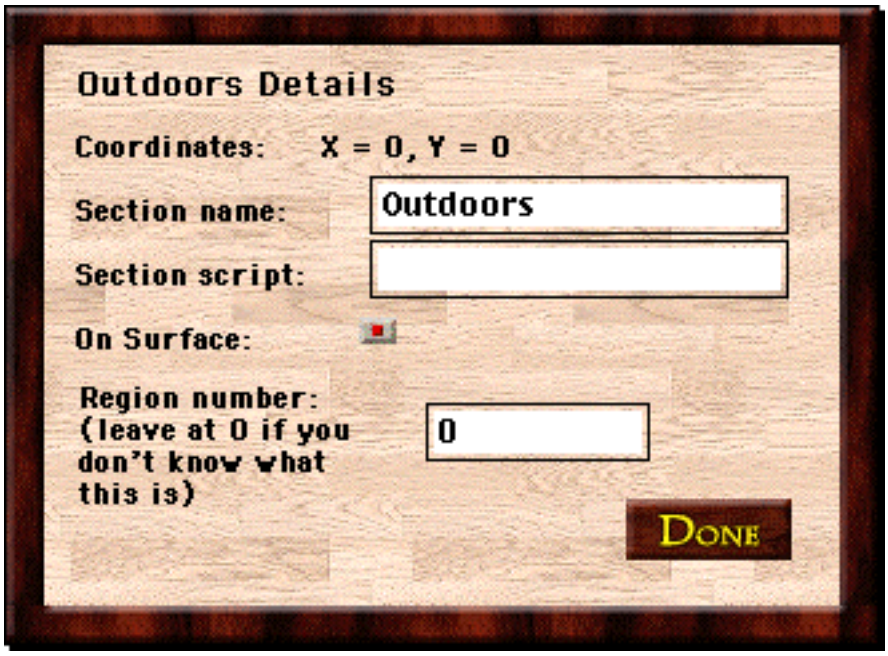
The previous chapter describes how to edit the terrain outdoors.

You can only work on editing one outdoor section at a time. To load a new section for editing, select Load Different Outdoor Section from the Outdoors menu.

In addition, you can set many useful properties for the outdoors by using options in the Outdoor menu:

Load Different Outdoor Section - Prompts you to select one of the outdoor sections in the current scenario. The editor will load the new section and you can begin editing it. You can only work on one outdoor section at a time.

Outdoor Details - Brings up a window where you set the properties of the current outdoor section:



Outdoor Details Window

Section Name - The name of the outdoor section, in regular text. (Such as "Northern Underworld").

Section Script (Advanced) - The name of the outdoor section's script, with the .txt extension left off. (Example: if the script is called "northunderw.txt", enter "northunderw" here.) Leave blank for no script.

On Surface - Select this checkbox is this section is meant to be on the surface of the world, leave it unchecked if the section is in the underworld. This affects what the default terrains are when painting the terrain of the section, and what graphics the game uses in the inventory area.

Region Number (Advanced) - This is a number. If the party stands in a section with one region number, they can not see any of the terrain in a section with a different region number. (For a good example of this, look at sections (0,2) and (1,2) in the Valley of Dying Things scenario.) This way, if you want section (0,0) to be underworld and (1,0) to be on the surface, you won't have to worry about the party seeing unwanted bits of the underworld when they are at the west edge of (1,0).

Outdoor Wandering Monsters, Outdoor Special Encounters (Advanced), Outdoor Preset Encounters (Advanced) - You may want the party to have to fight groups of enemies when they walk outdoors. All such encounters are designed on the window for editing outdoor encounters. This is described in the section on The Editing Outdoor Encounters Window, below.

Frill Up Terrain - Randomly shifts certain terrain types to different but very similar terrain types, to give the terrain a less uniform appearance. Turns some shrubs to different shrubs, grass to slightly different grass, and so on.

Remove Terrain Frills - The opposite of Frill Up Terrain. Takes all sort of grass and turns it into simple grass, turns all shrubs to one sort of shrub, etc.

Edit Area Descriptions - Brings up a window where you can change the text of all the area descriptions and, if you want, delete them.

Edit Starting Location - Sets the location of the town the party starts in. To set, select this option and then select the space of the town the party starts in.

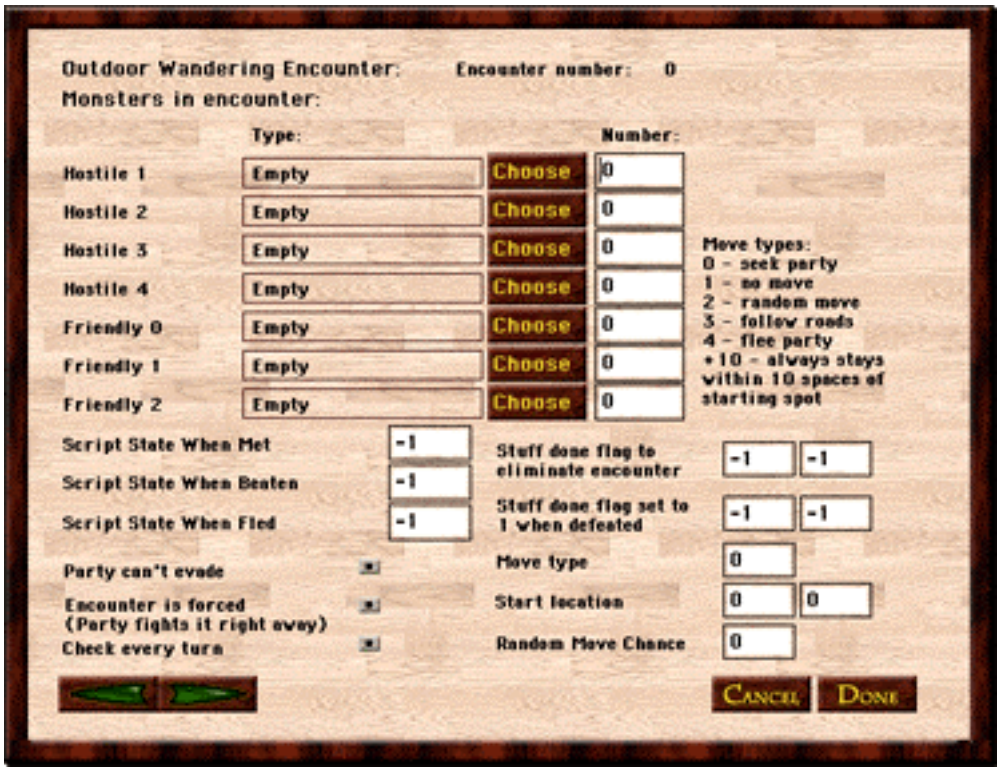
The Editing Outdoor Encounters Window

You may want the party to have to fight groups of enemies when they walk outdoors. All such encounters are designed on the window for editing outdoor encounters.

There are three different sorts of encounters: Wandering Encounters, which are occasionally randomly placed on the wandering monster spawn points and hunt down the party. Special Encounters, which are spawned into existence by commands on the outdoor section script. And Preset Encounters, which are set to always be there when the party enters the given outdoor section, until you do something to make them disappear.

One section can have at most 4 different types of wandering and special encounters and 8 preset encounters.

When you select one of these three menu options, you will see the Editing Outdoor Encounters window.



This window has a lot of options.

Hostile 1-4 - You can have up to 4 sorts of foes in an outdoor encounter. To choose a foe, press one of the Choose buttons, and then enter the number of the foe you want to appear.

Note that you must put a creature of some sort in the Hostile 1 slot. If the Hostile 1 slot is empty, the encounter will not exist. The creature in the Hostile 1 slot determines what the encounter looks like to the player when wandering around.

Friendly 1-3 - You can also have several friends appear by the party. They will fight on the party's side.

Party Can't Evade - Normally, a party can use Nature Lore skill to evade outdoor encounters. If this box is selected, the party can't evade it. You will usually want this checked for Special and Preplaced encounters.

Encounter Is Forced - Normally, a party only interacts with an encounter if it is right next to it. If this is checked, the party encounters it instantly, no matter what the distance. You will probably always want to use this for Special encounters, and never for Preplaced and Wandering encounters.

Check Every Turn - If the encounter has a script, the script will only normally run once every eighth turn (so the party doesn't keep triggering a script again and again, annoying a player). If this box is checked, the script is run every turn the party is by the encounter.

Script State When Met (Advanced) - A state in the section's outdoor script. When the encounter is met, that state in the script will be called. The script can display text, set the encounter to be non-hostile, or do other things. Leave at -1 for nothing to happen.

Script State When Beaten, Fled (Advanced) - Numbers of two states in the outdoor section's script. These states are called when the encounter is beaten or fled, respectively. Leave at -1 for nothing to happen.

Stuff Done Flag To Eliminate the Encounter (Advanced) - The coordinates of a Stuff Done flag. If this flag is non-zero, the encounter never appears. If one of the coordinates is left at -1, ignored.

Stuff Done Flag Set To 1 When Defeated (Advanced) - The coordinates of a Stuff Done flag. If this encounter is met, fought, and beaten, sets that flag to 1. If one of the coordinates is left at -1, ignored.

Move Type - How the encounter moves in the outdoors.

0 - Seeks the party.
1 - Doesn't move.
2 - Moves randomly.
3 - Follows roads. If not near a road, doesn't move.
4 - Flees party.
10 - Seeks the party, doesn't go more than 8 spaces from where it starts.
12 - Moves randomly, doesn't go more than 8 spaces from where it starts.
13 - Follows roads, doesn't go more than 8 spaces from where it starts. If not near a road, doesn't move.
14 - Flees party, doesn't go more than 8 spaces from where it starts.

Random Move Chance – You can have an encounter have a chance of moving in a random direction whenever it moves. This field contains the percentage chance that the creature will, on any turn, move in a random direction. This overrides the move type above. So, if an encounter seeks out the party but has a random move chance of 50%, half the time it will move closer to the party and half the time it will move in a random direction.

This feature doesn't work well on encounters who follow roads.

More About Outdoor Encounters (Advanced)

An encounter which wants to follow the party will normally follow it as far as it has to to catch up. However, if an encounter can call a script (when met, defeated, etc.) it will never leave the section it starts in. If the encounter walks to the edge of the section, it won't walk past it.

If you are using scripting for an encounter, you can choose to not have an encounter actually attack the party. You do this by using the outdoor_enc_result call in the script.

Remember, if an encounter has no creature in the Hostile 1 spot, it does not exist. If you want to have a group of guards that meets the party, talks to them, and leaves, you need to put the guards in the Hostile 1 spot and use a outdoor_enc_result call to make them not attack the party.

Creatures in outdoor encounters always have the default script for their creature type.

Terrain For Outdoor Encounters (Advanced)

When an outdoor encounter begins, the game selects a terrain type for the battle to take place on, depending on the terrain and floor type the party were standing on when they met their foes. For your custom terrain types, you can set what terrain is used by setting the fl_out_fight_town_used and te_out_fight_town_used fields of the floor or terrain types. The options are:

1000 - Cave Floor
1001 - Rough Cave Floor
1002 - Cave Hills
1003 - Cave Rubble
1004 - Cave Mushrooms
1005 - Cave Road
1006 - Cave Swamp
1007 - Cave Bridge
1008 - Volcanic Floor
1009 - Lava
1010 - Grass
1011 - Grassy Hill
1012 - Grass and Rocks
1013 - Grass and Shrubs
1014 - Grassy Road
1015 - Grassy Swamp
1016 - Grass and Bridge
1017 - Dry Ground
1018 - Dry Ground With Rocks
1019 - Dry Ground With Swamp
1020 - Dry Ground Hill
1021 - Dirt
1022 - Cave Floor (Floor Types 81,82)
1023 - Blackened Stone (Floor Type 89)
1024 - Floor
1025 - Walkway
1026 – Cave tunnel (north-south)
1027 – Campsite on grass
1028 – Campsite on cave floor
1029 – Cavern

In addition, you can create your own custom outdoor fight terrains. To do this, first, make a 48x48 town in your scenario. Then give this number in the fl_out_fight_town_used or te_out_fight_town_used fields of the terrain/floor type.

Finally, you can, for any encounter, set the terrain that the encounter takes place on. When setting up the encounter, you can set a state in the outdoor script that is called when the encounter is met (using the Script State When Met field). In that state, you can use the set_out_fight_town_loaded call to select a new battlefield for the encounter (using the numbers above or the number of a 48 x 48 town in your scenario).

When placing characters for an outdoor encounter, the game places all of the good characters near space (22,23) and all of the enemies around space (22,16). Design your terrain with this in mind, and make sure you have left open space for the characters to be placed.

Avoid the Edges (Advanced)

When outdoors, the game will have 4 outdoor sections in memory, and 4 scripts to choose from when the party meets encounters. To avoid any confusion on the game's part, It is best to avoid placing towns, special encounter, or preset creature encounters on the edges of an outdoor section.

When the game tries to call a state in an outdoor script, it always uses the script for the section the party is currently standing in.

Chapter 1.7: Editing Towns

Most of the action and excitement in Blades of Avernum scenarios take place in towns. Most of the special encounters, dialogue, and other excitement occurs here.
Note that Town is the generic term for any indoor section, which can include dungeons, castles, and so on.

Your scenario will have a list of towns (maximum number of 200). Each of the towns is 32 x 32, 48 x 48, or 64 x 64 spaces. The game does not hold the entire list of towns in memory when the user plays, only the town they are currently in.

To edit the current town, select Edit Town from the Scenario menu.

Editing terrain for towns (and placing waypoints, barrels, special encounters, and other terrain features) is described in the chapter on Editing Terrain.

You can only work on editing one town at a time. To load a new town for editing, select Load Different Town from the Town menu.

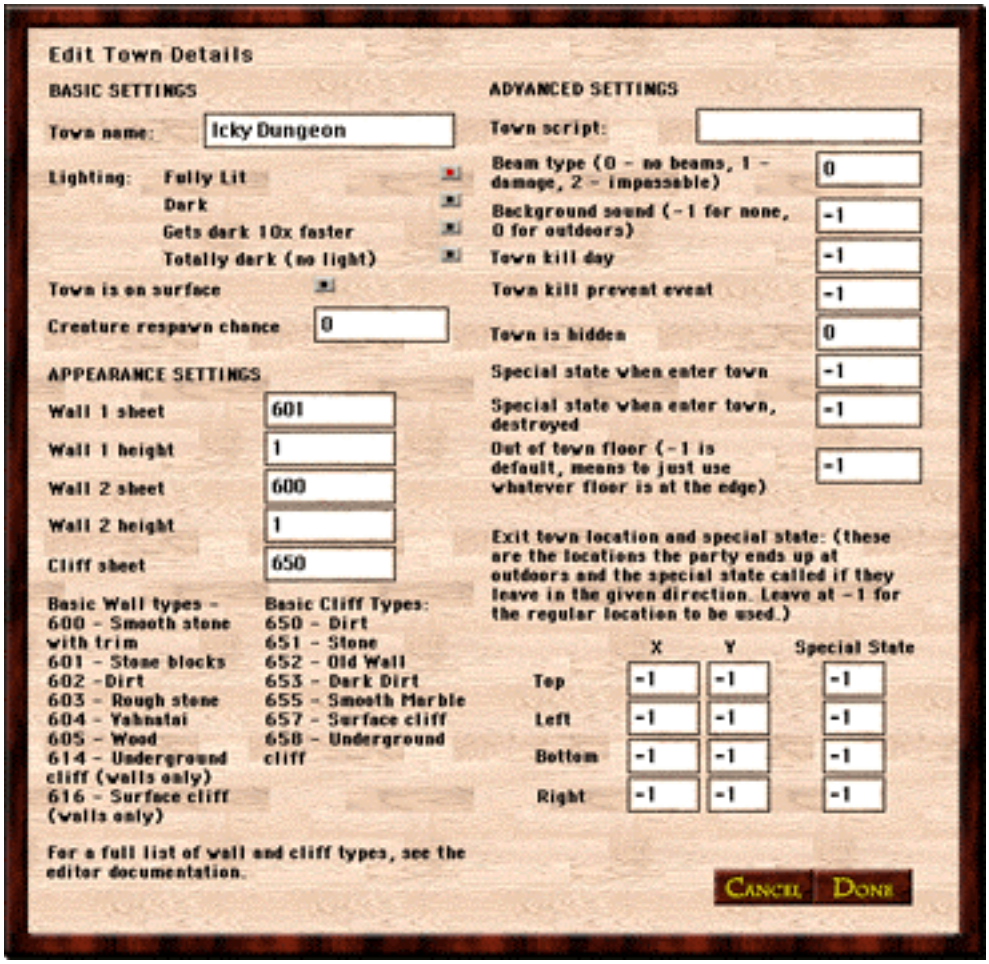
Many of the other details for your Town are edited using the Town menu:

Load Different Town - Prompts you to select a town to start editing. Loads into memory the town you select.
Town Details - Lets you set most of the details for the current town. Read The Town Details Window section below for more information.
Town Wandering Monsters - Occasionally, the game will spawn some wandering monsters randomly around one of the wandering monster spawn points. Choose them on this window. There are 4 different sorts of groups that can appear, each of which can have up to 6 different monster types in it. Leave a field at 0 or -1 to represent no monster.

Enter the number of the monster types you want to appear, or press the Choose buttons to select them from a list.
Set Town Boundaries - Prompts you to select the upper left and then the lower right of a rectangle. This rectangle is the boundary of the town. When the party steps outside of it, they are returned to the outdoors.
Frill Up Terrain - Randomly shifts certain terrain types to different but very similar terrain types, to give the terrain a less uniform appearance. Turns some shrubs to different shrubs, grass to slightly different grass, and so on.
Remove Terrain Frills - The opposite of Frill Up Terrain. Takes all sort of grass and turns it into simple grass, turns all shrubs to one sort of shrub, etc.
Edit Area Descriptions - Brings up a window where you can change the text of all the area descriptions and, if you want, delete them.
Edit Starting Location - Sets the location in the town the party starts the scenario in. To set, select this option and then select the space in the town the party starts on. Of course, you will only want to set this in one town.
Add Random Items (Advanced) – Goes through the towns and places random items you have selected on terrain types you selected. You make item shortcuts by selecting Edit item placement Shortcuts from the Scenario menu.
Set All Items Not Property - Sets all items in the town to not be someone else's party. The player will be able to pick them up at will.
Clear All Items - After you give confirmation, erases all items in the town.
Clear All Monsters - After you give confirmation, erases all creatures in the town.
Clear All Special Encounters (Advanced) - After you give confirmation, erases all special encounters in the town. Note the script is left alone. Only the special encounter rectangles you have placed are affected.

The Town Details Window

Select Town Details from the Town menu to edit the details for the town. There are many things that can be set on this window, some basic, some quite advanced.



The Town Details Window

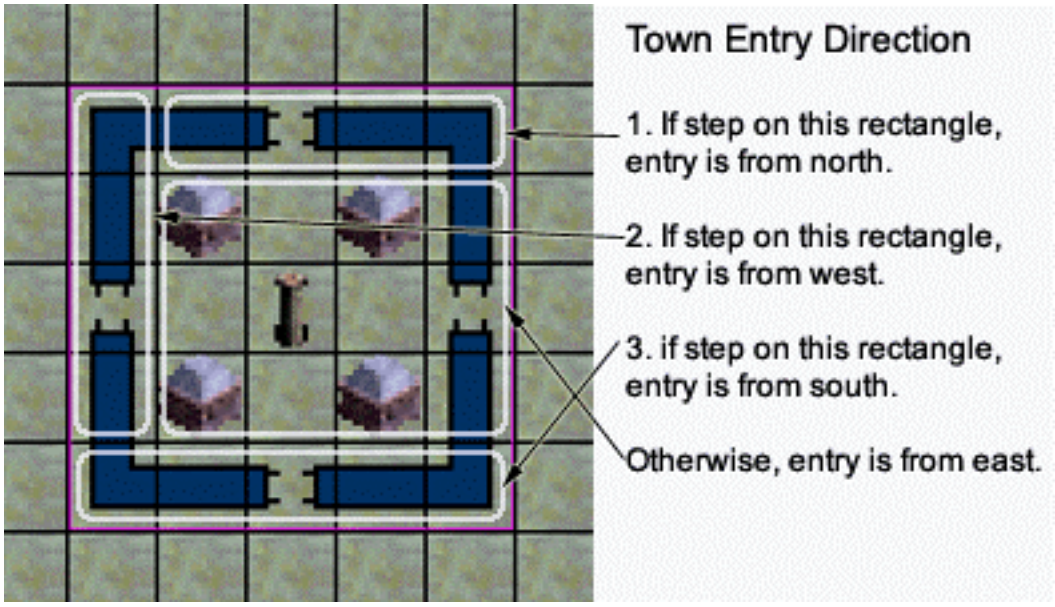
Town Name - The name of the town, in regular text.
Lighting - Select the box by the light level you want.
On Surface - Select this checkbox is this town is meant to be on the surface of the world, leave it unchecked if the town is in the underworld. This affects what the default terrains are when painting the town of the section, and what graphics the game uses in the inventory area.
Creature Respawn Chance - Normally, when you kill something, the game remembers that it is dead. It won't be back when you return later. This field is the percentage chance that a creature will reappear when the player reenters the town. If 100, all dead creatures come back to life every time.
Wall 1 Sheet, Wall 2 Sheet - Lets you select the appearance of wall types one and two in the town. Enter the number of the graphic to use. All of the possible wall types are listed in the appendices, and the most commonly used ones are listed at the lower left corner of this window. For example, if you want Wall 1 to look like wood, enter 605 in Wall 1 Sheet.

Remember, not all terrain types are available for all wall types. If you choose Underground Cliff or Surface Cliff (614 and 616) you can only place regular walls of that type, not doors, windows, gates, etc.
Wall 1 Height, Wall 2 Height - The number of icons high Walls 1 and 2 are. If 1, normal height, if more, higher.
Cliff Sheet - The number of the sheet the game draws cliffs from. The most commonly used are listed to the lower left and all are listed in the appendices. For example, set this to 653 if you want the cliffs to look like dirt.
Town Script (Advanced) - The name of the script for the town, without the .txt extension (so if the town's script is "hellpit.txt", enter "hellpit" in this field). If no script, leave this field blank.
Beam Type (Advanced) - Determines whether beam projectors are active in the town (more on Beam Projectors below). If left at 0, no beams. If 1, beams can be walked through, and they do damage. If 2, beams are impassable.

Background Sound (Advanced) - The sound that loops when the party is in this town. Leave at -1 for no sound, 0 for the default outdoor sound (which will depend on whether the town is on the surface or not) and a number from the list in the appendices for other background sounds.
Town Kill Day (Advanced) - The day in the scenario on which the town is cleaned out of creatures (because it is abandoned, destroyed, whatever). Leave at -1 for none.
Town Kill Prevent Event (Advanced) - If a day is set in Town Kill Day, and an event number is given here, and the event happens before the town kill day, the town is not killed. The event is assumed to be an event that prevents the town dying. (Example: If the Town Kill Day is 15 and the Prevent Event is 5, if event 5 happened on day 13, the down would not die, and if the event happened on day 17, the town would die.)

To set an event as having happened, use the set_event_happened call in a script.
Town Is Hidden (Advanced) - If set to 1, the town is invisible in the outdoors and can't be entered. This continues until the town is made visible using the set_town_visibility call.
Out Of Town Floor (Advanced) - The number of the floor to draw when the party can see off of the edge of the town. If left at -1, the game just extends to infinity the floor at the edge of the town.
Exit Town Locations (Advanced) - When you leave the town from a certain direction (north, west, south, east) you can force the location in the current outdoor section the party is placed at. Be sure to not place the party inside the boundaries of a town. If you don't set any value for this, the game just picks an appropriate location for the party at the edge of the town's boundary rectangle.
Exit Town Special State (Advanced) - You can have the game call the town's script when the party exits a town in one of the four directions. Enter the state of the town script you want the game to call when they leave in that direction, or leave at -1 for none.

Enter Town Starting Position



Determining Town Entry Direction

To create the entryway to a town in the outdoors, you set a rectangle in the terrain. When the party walks into that rectangle, they are dropped into the town. You use the Create Town Entrance button to make town entries.

When the party walks into that rectangle and are put into town, they can be entering that town from the north, south, east, or west (which determines where they are put in the town). The question is: how does the game determine where the party is placed, depending on how they walk into that town rectangle. This can be tricky if the town entry rectangle is 1x1.

This is how the game determines the town entry direction depending on what edge of the town entry rectangle the player steps on:

Step 1. If the player is on the south edge, the entry direction is from the south. (So if the town entry rectangle is 1x1, the player ALWAYS enters from the south.)
Step 2. Otherwise, if the player is on the west edge, the entry direction is from the west.
Step 3. Otherwise, if the player is on the north edge, the entry direction is from the north.
Step 4. If the party isn't on the south, west, or north edge, the entry direction is from the east.

When you place the entry points for a town, be sure to leave enough space to place the entire party. The game will not place the party on blocked spaces or spaces with terrain that has an effect when crossed (like swamps and lava).

About Objects

To flesh out your town, you can place items, creatures, and terrain scripts. These 3 things are referred to collectively as Objects. Once placed, you can select them, delete them, edit them, and copy and paste them using the Edit menu.

To select an object, press the Select/Edit Placed Object button (or type Shift-S). Then click on the Object to edit. If there are multiple objects on the space, click repeatedly to cycle through them.

Once an object is selected, information on it appears to the lower left. You can click on the fields there to edit them (more details on this later).

When an object is selected, select Copy from the Edit menu and then Paste, and you can click on a different place to place another copy of that object.

When an Object is selected, you can use the arrow keys to move it around.

Placing and Editing Items

The menus I1, I2, I3, I4, and I5 list all of the items in the scenario. To place an item, select it from a menu and click on the terrain where you want it to be.

When you place several items in a space, the editor will automatically offset them so that they are not all drawn on top of each other. This can be manually changed later.

When you place an item or select it, information about the item appears at the lower left corner of the screen. You can click on all of the entries to change them:

Name of Item - Used to change the item type. Select this to bring up a window with a list of all of the available items. Select a new item type.
Charges/Amount - This only appears for items with charges or amounts (like gold coins or wands). You can manually set the number of charges for the item or, in the case of coins, how many coins are there.
Property/Not Property - Whether or not the party can get the item without being guilty of a crime.
Contained/Not Contained - Whether the item is inside a container or not. The game automatically sets items on spaces with containers to be contained. If you don't want this, you can change the containment manually, though it is not recommended.
Drawing Shift X,Y - The number of pixels the item is offset when drawn in a space. X is the horizontal offset and Y is the vertical offset. If these values are outside the –6 to 6 range, the game will draw them in the center of the space.

You can place at most 144 items in a town.

Placing and Editing Creatures

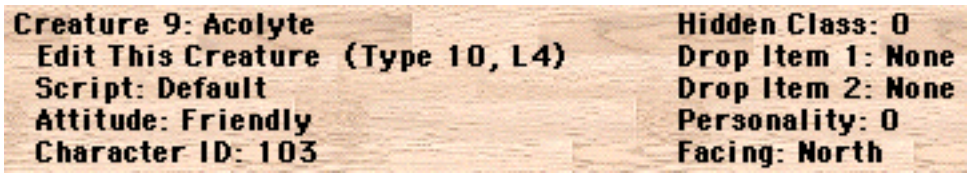
The menus C1, C2, C3, and C4 list all of the creatures in the scenario. To place a creature, select it from a menu and click on the terrain where you want it to be. There is a number after each creature name in the menus. This is the level of the creature.

A creature in the game is sometimes referred to as an NPC (which stands for Non-Player Character).

The editor will not let you place two creatures on the same spot. You can use the arrows keys to force two creatures be on the same spot, but it is not recommended.

You can place at most 80 creatures in a town.

When you place a creature or select it, information about the creature appears at the lower left corner of the screen. You can click on all of the entries to change them:



Creature Number and Name - The number is the number of the creature in the town. When you want to refer to this creature in a script, use this number. The name of the current creature type. Click this entry to select a different creature type.

Edit This Creature - Click this to bring up the edit creature window (described below).

Script (Advanced) - The name of the script for the creature. If at default, uses the script that is set as its default when the creature is defined in a script. Click this to set a new script. Give the name of the script without the .txt extension (so if the script name is "badwolf.txt", enter "badwolf").

Attitude - A creature's attitude can be Friendly (won't attack party and will attack party's enemies), Neutral (doesn't attack anyone), Hostile A, and Hostile B (both attack the party, and Hostile A and Hostile B attack each other). Select this line to cycle through the options.

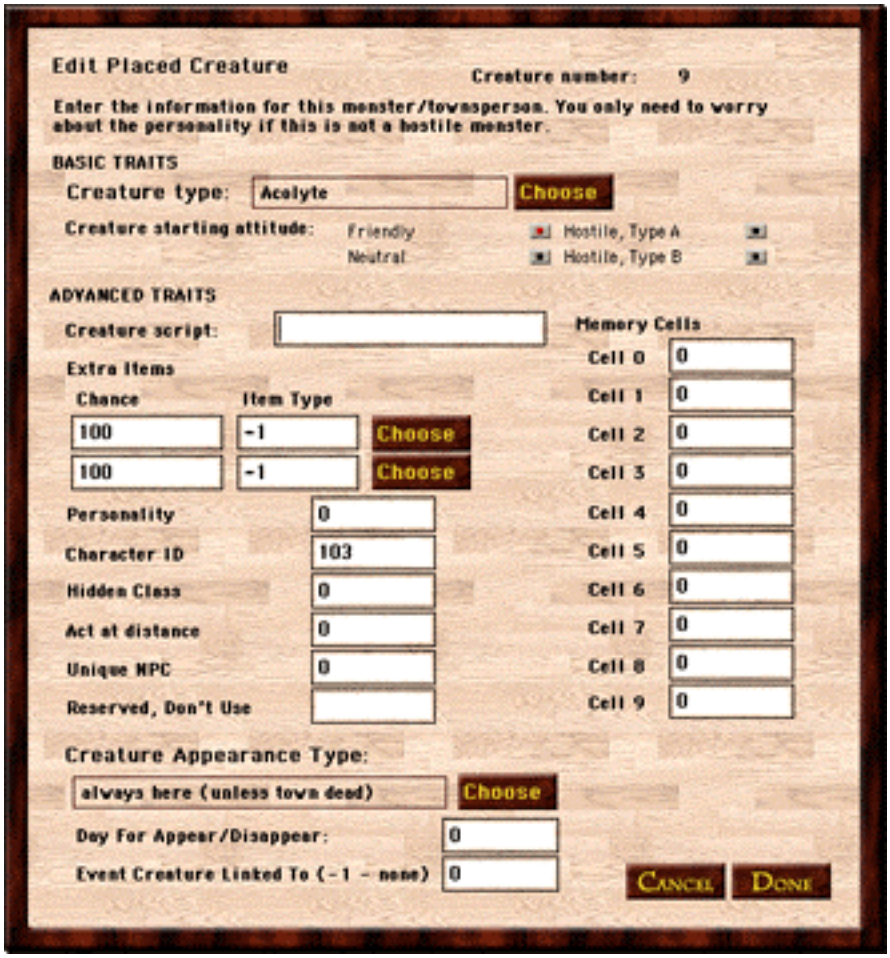
Character ID - A unique number that identifies the creature to the game. When a creature with a given ID dies, creatures with that ID never appear in the town again (unless the town has a positive respawn chance). You probably want to leave this alone.

Hidden Class (Advanced) - If this is not zero, the creature does not appear when the town is entered. Instead, it only appears when a script uses a spawn_creature or activate_hidden_group call is used.

Drop Item 1, Drop Item 2 - When these are selected, you will be prompted to pick an item and give a percentage chance. This is the chance that, when the party enters the town, this creature will have the chosen item.

Personality (Advanced) - Every creature the party talks to should have a unique personality (a number that identifies it). This is used by the game to keep track of whether the creature has been talked to before. Personalities must be in the range from 0 to 3999.

Facing - When the creature is placed, what direction it is facing. Select this repeatedly to cycle through the options.



The Edit Creature Window

When you select Edit This Creature, the Edit Creature window appears. The options on it are:

Creature Type - What type the creature is. Press Choose to select a new one.
Attitude, Script, Drop Item 1 and 2 - Same as for the entries on the main window, described above. Press Choose next to the Drop Item fields to choose items.
Personality, Character ID, Hidden Class - Also same as for the entries on the main window, described above.
Act At Distance (Advanced) - Running creature scripts is CPU intensive, so it is best if not all creature scripts are running all the time. Creature scripts default to running every 8 moves, and only when the party is nearby. If a creature needs to be active even when the party is far away, you can enable that behavior here. Possible values are:

0 - Run the creature's script every 8 moves, unless the party is far away.
1 - Run the creature's script every 8 moves, even if the party is far away.
2 - Run the creature's script every move, unless the party is far away.
3 - Run the creature's script every move, even if the party is far away.

Unique NPC - You can set for each town a percentage chance that killed creatures respawn. If this set to 1, the creature never respawns.
Memory Cells (0 .. 9) (Advanced) - Here, you can set 10 values the creature's script can consult to see how to behave. For more on this, read Interacting With Scripts For the Complete Beginner, below.
Creature Appearance Type, Day To Appear/Disappear, Event Creature Linked To - By default, a creature always appears in a town, unless the town has been destroyed (destroy a town by

using the set_town_status call or by setting a destruction date in the Town Details window). Using these fields, however, you can change when this creature is present when the party enters a town.

To change the creature's appearance/disappearance type, press the Choose button. The creature will then be present or missing depending on what you select and what the values in the Day and Event field are. The options are:

Always here (unless town dead) - The default. Creature is here unless town destroyed.
Here at time t unless town dead - The creature is here if the scenario's current day is at least the value in the Day field. If town is destroyed, creature is not here.
Disappear at time t - The creature is here if the scenario's current day is less than the value in the Day field. If town is destroyed, creature is not here.
Here if event not done by time t - The creature is here unless the event given in the Event field is done before the day given in the Day field. (You set an event as having happened using the set_event_happened call). If town is destroyed, creature is not here.
Gone if event not done by time t - The creature is here if the scenario's current day is less than the value in the Day field, unless the Event in the Event field was done before that day. If town is destroyed, creature is not here.
Here if event happened - The creature is here if the event in the Event field has happened.
Gone if event happened - The creature is here unless the event in the Event field has happened. If town is destroyed, creature is not here.
Here on day 0-2 of every 9 days - If the scenario's current day is in the first third of every 9 day period (i.e. days 0-2, 9-11, 18-20, etc.), the creature is here. Otherwise not. If town is destroyed, creature is not here.
Here on day 3-5 of every 9 days - If the scenario's current day is in the second third of every 9 day period (i.e. days 3-5, 12-14, 21-223, etc.), the creature is here. Otherwise not. If town is destroyed, creature is not here.
Here on day 6-8 of every 9 days - If the scenario's current day is in the last third of every 9 day period (i.e. days 6-8, 15-17, 24-26, etc.), the creature is here. Otherwise not. If town is destroyed, creature is not here.
Here if and only if town dead - The creature is here if and only if the town is destroyed.

Creature Levels

Like characters, each creature has a level (which is given after the creature's name in the menu and editing creature area). The level is the most important value determining how dangerous a creature is. A creature's strength increases exponentially with its level (so a level 20 creature is far more than twice as dangerous as a level 10 creature).

You should make sure to place creatures in a scenario of about the level of the characters you expect to play it.

Changing a creature's level in a script is a good way to rebalance a scenario on the fly. Read about the set_level and set_creature_type_level calls for more information.

Interacting With Scripts For the Complete Beginner

Scripts are the most powerful, most versatile, and most complicated element of the Blades of Avernum system. Basically, a script is a file containing a chunk of text. This text is instructions, telling some element of the game how to behave. For example, each creature has a script attached to it, which tells it what to do: when and where to move, what to attack, what to say when someone talks to it, etc.

Similarly, every terrain type the player interacts with (say, doors) has a script.

Most users of Blades of Avernum will want to be able to make simple scenarios without ever looking at scripts. This is possible. When you create a scenario, the most important scripts are automatically placed in your scenario folder (such as basicnpc.txt and door.txt). You should not remove them, whatever you do.

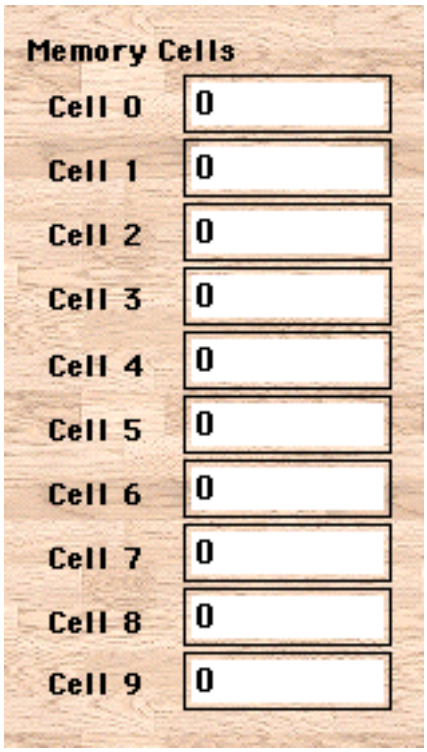
When you place a creature, it automatically uses the script basicnpc.txt (remember, NPC stands for Non-Player Character). The creature will wander around randomly, never going too far from its start spot,

and attack any enemy that gets close. Basic, simple, sensible behavior. However, you can easily edit how the character acts.

Two characters with the same script can act very differently. That is because scripts consult things called memory cells. Each creature has ten memory cells, and you can put a number into each one. The script can look at these numbers to see how it is supposed to act (for example, whether it should move around or stand still).

To edit a creature's memory cells, select it, and then select Edit This Creature at the lower left. The memory cells can be edited on the dialog box that comes up.

For the basic creature script, the meanings of the memory cells are as follows:



Memory Cell 0 - How creature moves.
0 - If 0, wander randomly.
1 - Stands still until a target appears.
2 - Completely immobile, even if target appears.

Memory Cell 1,2 (Advanced) - Stuff done flag. If both 0, nothing. Otherwise when this is killed, set to 1. (Example: If cell 1 is 3 and cell 2 is 5, when creature is killed, sets SDF(3,5) to 1.)

Memory Cell 3 (Advanced) - Dialogue node to start with if talked to. If left at 0, this character doesn't talk.

For the basic scenario designer, only the first memory cell is important. Set Memory Cell to 1 to make the creature only move when it sees a foe. Set the cell to 2 to have the creature never ever move.

Terrain Scripts work much the same way. When you want a spot of terrain to act in an unusual way, you can place a script on top of it. You can fine-tune that behavior (for example, set a door from unlocked to locked or set the difficulty of a trap) by changing the script's memory cells.

Placing and Editing Terrain Scripts (Advanced)

Place a door in the zoomed in terrain view, and you will see that a little scroll icon appears on the space. This icon means that there is a terrain script on that spot. Terrain scripts are peculiar and abstract and can be hard to understand, but they are also extremely versatile and powerful.

A terrain script is a script that exists on one spot in the town's terrain. You can place many terrain scripts in a town, though you can only have one script on each space. Each terrain script corresponds to a script (a text file) in your scenario's folder. Every scenario starts with the script door.txt in its folder. You can place many doors in a town, and each will have its own terrain script on it, each of these terrain scripts using the file door.txt. (The same text file can be used many times in the scenario.)

You can put any terrain script on any space. So you can, if you want, place a terrain script using door.txt on a space that doesn't have a door. It won't work right, but you can do it.

Anyway, now that all those confusing things have been said, here is a simple, real-life use for terrain scripts. Go to that door you have just placed. Press the Select/Edit Placed Object button (or type Shift-S) and then click on the door until you see the object described to the lower left begins "Terrain Script [number]". You can edit the script using the fields to the lower left.

The fields are:

Script - The name of the script (a text file) the terrain script will use, without the .txt extension. (So if it's a door with script file "door.txt", enter "door").
Memory Cells 0-7 - The eight memory cells for this particular terrain script. The script will use these values to determine how to behave (whether a door is locked or unlocked, for example). For more on memory cells and how they work, read Interacting With Scripts For the Complete Beginner above.

Terrain types like doors have scripts attached to them by default. You may want to place a terrain script on your own. To do this, press the Place Terrain Script button and then select a spot on the terrain.

If you want to place unusual terrain types without actually writing the scripts, you can get scripts from other people or places. For example, go into the Valley of Dying Things scenario folder and read the file trap.txt. At the top, you will see a text description of what the trap door is and how the memory cells change its behavior. If you copy trap.txt into your scenario's folder, you can place trap scripts and edit the memory cells to make the trap behave how you want. You can even do this without having any idea how to write a script.

If you go through the scenarios that come with Blades of Avernum, you will find a number of terrain scripts. Each begins with a simple description of what it does and how the Memory Cells make it work.

A number of interesting terrain scripts are listed in the appendices. You can have at most 100 terrain scripts in a town.

Setting the Difficulty for Doors/Traps

For each locked door you place (using the default locked door script), you will need to set a difficulty. This is the Strength that is needed to bash the door in, or the Tool Use skill required to pick the lock. The strength is set in memory cell 0.

In general, the difficulty has to have a minimum of 4 to count for anything at all (because every character will generally have a strength of at least 2 or 3). A difficulty of 5-6 is low, 7-10 is medium, and above that is high.

When picking locks, the character with the highest Tool Use skill will make an attempt. That character gets a bonus from the item in his or her inventory with the highest Pick Locks ability. For example, basic lockpicks give a bonus of 2 levels, and magic lockpicks give a bonus of 10.

Casting an Unlock spell near a door is equivalent to bashing it with a Strength of 5 + 3 * the level the character knows the spell at + the character's bonus with mage spells. The Unlock spell will usually be a player's most powerful way to unlock doors.

Setting the difficulty for traps (using the default trap script) is the same, except that only the Tool Use skill helps, there is no lockpick bonus, and the Unlock spell doesn't help. Generally, traps start to get tricky to disarm around a difficulty of 4-5.

Magical Doors (Advanced)

There is a mage spell called Unlock Doors. When cast near a terrain script, it calls state UNLOCK_SPELL_STATE in that script. The script can then process the Unlock spell's effect. The strength of the spell is 2 times the caster's skill with the spell, plus half the caster's spell bonus. The strength can be found by the script by using the get_unlock_spell_strength call.

The Unlock Doors spell can unlock regular doors (that use the default door script), though it is difficult. The spell's strength must be at least twice the door's lock strength.

In addition, the terrain scripts folder contains a very useful script called magicdoor.txt. This is the script for a door with a magic lock. The party must first unlock it with a magic spell, and then pick the lock normally. For adventures with high level parties, it's a good idea to use this script a few times to make the Unlock Doors spell useful.

Beam Projectors (Advanced)

Terrain types 380-389 are beam projectors. A projector fires out a harmful beam which continues until it hits an obstacle. The beam either blocks party movement or harms anyone who tries to walk through it. If a beam hits a mirror, it is deflected 90 degrees to the right or left.

Beam projectors aren't on by default. You need to activate them by entering a value in the correct field of the Town Details window.

Terrain types 380-383 are projectors, facing north, west, south, or east. A projector does not fire a beam unless it is next to an active power source (terrain type 384). Some power sources start out off (terrain type 388), but they turn on if a beam hits them and stay on until the party leaves the town. If a beam hits a projector, the projector is destroyed.

Terrain types 386 and 387 are immobile mirrors. You can create mirrors the party can push around by using the put_object_on_space script call.

Beams automatically destroy any terrain type that can be affected by a Move Mountains spell.

Lighting

You will probably want to put some lights in all of your towns and some of your dungeons. Some terrain types, like braziers, torches, and streetlights, cast light in a certain radius.

When you save your town, the game calculates in advance the amount of light each of these terrain types sheds on the spaces around it. This is not recalculated later in the game (to save CPU time). So if a special encounter deletes a torch, the light it cast will still be there.

Since the party might enter towns on the surface at night, they should probably have lights.

Section 2: Scripting

Chapter 2.1: Introduction to Scripting

All of the real power in Blades of Avernum comes from scripts. Scripts can be used to create custom terrain types, items, and creatures, make special encounters, control creature behavior, run dialogue, and much more.

Scripts are easily the most complicated thing about Blades of Avernum. Writing scripts for creatures and special encounters is, basically, computer programming. Once mastered, however, they enable you to do a phenomenal number of things.

Important Note For Beginners

If you do not want to learn how to write the more complicated scripts, it is still possible to create scenarios. You don't need to be able to program to write dialogue. Just read the chapter on making town dialogue. It is still a bit complicated, but you don't need to be able to program.

You can still have terrain and creatures that act in interesting ways. Use premade scripts, which come with the editor or will be available at the Scenario Workshop at http://www.spiderwebsoftware.com. You will need to understand how memory cells work, but then you can just place the creatures with the premade scripts, plug the appropriate numbers into the memory cells, and let them run.

To make special encounters, you need to understand Stuff Done Flags and a bit of programming. However, simple examples and templates will be available at the Scenario Workshop.

In short, don't panic. You will be able to do a lot without understanding the full complexity of the system.

What Is a Script?

A script is a text file. It always has a name at most 13 characters long (with only letters, numbers, and spaces) followed by ".txt". Examples: "fish.txt", "Valley Dying.txt", "o12dungeon.txt".

On a Macintosh, scripts can be edited using SimpleText, BBEdit, or any other text editor. On Windows, scripts can be edited using NotePad or WordPad. Make sure the script is saved as regular text (not, say, in Microsoft Word format) or Blades of Avernum won't be able to understand it.

The text file of the script itself contains a series of instructions. For scripts which contain custom terrains and objects or dialogue, the script has lines of data. For scripts which control the behavior of creatures or special encounters, the script contains program code written in a special, simple programming language, called Avernumscript.

All of the scripts for a scenario must be in the same folder as the scenario (not in any subfolders). The game will only be able to find the scripts if they are in the same folder as the actual scenario file.

As always, the best way to learn scripting is to look at examples. Explore freely the folders containing the scenarios that come with Blades of Avernum.

The Basics of Script Syntax

Now for some basic information about how scripts work. In a script, each line is either empty or contains a command. Every command must end with a semicolon (";"). Examples:

```
beginscendatascript;
begindefinefloor 135;
import = 95;
clear;
variables;
short i,target;
do_attack();
```

Semicolons are how the game knows that you have finished a command and it should be executed.

Every script begins with a line which tells the program what sort of script it is, such as:

```
beginscendatascript;
begincreaturescript;
begintalkscript;
```

Beyond that, every script has a different syntax, depending on what the script is for.

In a script, it is possible to comment out text. What this means is that all of that text is ignored by the game. It is invisible. You can put anything you want there. Comments are a way to let yourself leave notes about what a script does. To comment out something, type "//" in a line. Everything after the "//" on that line will be ignored by Blades of Avernum. For example,

```
beginscendatascript; // Write anything you want here.
```

Some Script Terminology

Finally, some terminology. This may be confusing at first for the many people who aren't used to computer code. Rest assured that it will all sink in in time.

A string is a bit of text. (Example strings are "Fred" or "1111211").

An integer is a round number. It can be negative. 10, 5, -31, and 0 are integers. 3.5, 4.7 and pi aren't integers. Integers can be referred to in Avernumscript interchangeably as a "short" or an "int". (Int is short for Integer). Some sample shorts (or ints) are 0, 51, and -3240. Numbers in scripts can have values from -32767 to 32768. If your numbers go out of those ranges, strange things will happen. That is just how computers store numbers.

A Note On Formatting

When the instructions contain samples of code or formats of commands, they are written in `Courier font`. Descriptions of the calls are given in the regular font.

Anything on a line after an "//" is ignored. The writing after an "//" is called a "comment" and can be used to explain what is going on. Many of the code samples ahead have comments that explain what is going on.

In some sample code, you will often see something included in brackets, such as

```
begindefinefloor [number of floor to define];
```

The thing in brackets is something the game is expecting you to provide. In this case, it is a number.

Random Numbers

Much of what happens in role-playing games is determined by random numbers. Blades of Avernum has its own terminology for describing random numbers. A "die" is a single random number (which will be from one to something else). The "size" of the die is the highest number it can be. So if a die is size 8, that die gets random numbers from 1 to 8.

d[number] is a random number from 1 to [number]. So, for example, d8 is a random number from 1 to 8, and d30 is a random number from 1-30. (d stands for die)

[number 1]d[number 2] is [number 1] different random numbers, each from 1 to [number 2]. So 2d6 is a random number from 2-12 (1 to 6 + 1 to 6) and 3d8 is a random number from 3 to 24 (1 to 8 + 1 to 8 + 1 to 8).

What does all this mean in practical terms? Well, when making a creature which, say, bites people, you will have to say the size of the die for its attack damage. For example, you might say the attack's die is size 8. That means that the creature will do damage in multiples of 1 to 8. Creature attacks tend to do a number of dice of damage equal to the creature's strength. So if the creature has strength 5, its attack will do 5d8 damage (or 5-40 points).

One more example. When you set a floor to do damage to people, it does 2 dice of damage. You will set the size of the dice. So if you set the dice size to be 20, stepping on that floor will do 2d20 damage (or 1 to 20 + 1 to 20, with a possible maximum of 40).

Chapter 2.2: Creating Custom Objects

There are four sorts of things in Blades of Avernum that are entirely defined by scripts:

Floors - There are 256 possible floor types in each scenario, numbered 0 .. 255. Floor types 130 to 254 are free and you can place custom floor types in them.
Terrain Types - There are 512 possible terrain types, numbered 0 .. 511. Terrain types 426 and up are free and you can place custom floor types in them. In addition, all of the already defined terrain types from 122 up can be replaced with your own custom terrains.
Items - There are 500 possible item types in each scenario, numbered 0 .. 499. Item types from 450 up start unused, and you can replace any item type with a custom item.
Creatures - There are 256 possible item types in each scenario, numbered 0 .. 255. Creature types from 234 up start unused, and you can replace any creature type with a creature item.

When you create a scenario, you will start out with a huge array of predefined, default floors, terrain types, items, and creatures. These are defined in the scripts corescendata.txt and corescendataw.txt in the folder Blades of Avernum Files. You are welcome to look at these files to see how things are defined, but you can not modify them. These scripts must be left alone, and can only be changed by Blades of Avernum updates released by Spiderweb Software. This makes sense. After all, all scenario designers share these files. If you change them, you may break someone else's scenario.

This is not a real limitation, however. You can massively customize your scenario by making a Custom Object Script. This script will contain descriptions of all your new and changed floors, terrains, and objects, which will be read when the scenario is entered or loaded.

This script, like all scripts, must be placed in the same folder as the scenario.
A Custom Object Script has the name of the scenario file, followed by "data.txt". Examples:

If the scenario file is stealth.cmg, the Custom Object Script is stealthdata.txt.
If the scenario file is valleydy.cmg, the Custom Object Script is valleydydata.txt.

Beginning a Script

To make your script, first make a text file of the appropriate name. Then start writing it. The first line of a Custom Object Script must be:

```
beginscendatascript;
```

Begindefine Statements

Next, you will define floors/terrain/items/creatures one at a time. To define something, you first write a line saying what you will be defining. This is canned a begindefine statement. The four possibilities are:

```
begindefinefloor [number];
begindefineterrain [number];
begindefinecreature [number];
begindefineitem [number];
```

The [number] is the number of the thing you will be editing (so "begindefinecreature 91;" means you're about to edit creature 91).

Issuing a begindefine command does not just tell the program you are starting to change a thing. It also makes that thing a copy of the thing of that type you last edited. In other words, if your script contained these commands:

```
begindefinefloor 135;
      (a bunch of stuff)
begindefinefloor 136;
      // no extra data
begindefinefloor 137;
```

```
// no extra data
```

The result of these commands would be that floor types 136 and 137 are copies of 135. "begindefinefloor 136;" has the result of copying all the information for floor type 135 into floor type 136. Note that the first floor, terrain type, etc. you start defining in any script doesn't get any data. If "begindefinefloor 135;" was the first time you started defining a floor in the script, no extra data would be copied into floor 135.

Clear and Import

After the begindefine statement, you have the option of using one of two commands. One is

```
clear;
```

This fills the current floor, terrain type, etc. with all of the default values (all of the values and their defaults are given in the following chapters).

The other command is

```
import = [number];
```

This makes the current floor, terrain type, etc. a copy of the thing of the same type with the given number. For example:

```
begindefinefloor 135;
	import = 91;
```

would make floor 135 a copy of floor 91.

Characteristics

The qualities of your floors, terrain types, etc. are called characteristics. The name, the number of the graphic, the level of the creature, each is a characteristic. There are 3 sorts of characteristics:

Integer characteristic - A number. Such as the level of a creature or the value of an item.
String characteristic - A bit of text. Such as the name of a creature or floor type or the name of a script.
Array characteristic - A list of numbers of a certain length. For example, each creature type can be set to have 8 items it can start with. These possible items are an array characteristic of length 8. Another example is the list of skills for a creature type.

After the begindefine and possible clear or import command, you define the characteristics of your floor, terrain type, creature, or object. Each line is the definition of one characteristic. These lines look something like

```
fl_name = "Slimy Cave Floor"; // string, changes a floor's name to "Slimy Cave Floor"

te_which_icon = 2; // integer

cr_name = "Wisp"; // string
cr_level = 1; // integer, changes a creature's level
cr_start_item 1 = 210; // array

it_full_name = "Lab Equipment"; // string
it_variety = 21; // integer
it_floor_which_sheet = 1033; // integer
```

Most lines have the name of the characteristic you are changing (like "it_full_name"), followed by an equals sign, followed by an integer or a string, followed by a semicolon. There is one exception - array characteristics.

An array characteristic is a list of characteristics you can set (like all of the skill types for a creature type, or the 8 items you can set a creature type to have.) A line defining an array characteristic has the format:

```
[name of array characteristic] [number of list entry to change] = [new value];
```

So all of the possible formats to edit characteristics are:

```
[name of integer characteristic] = [new value];
[name of string characteristic] = [new string];
[name of array characteristic] [number of list entry to change] = [new value];
```

Finally, when you are through defining your floor type, just proceed to the next begindefine statement.

The meanings of each of the characteristics you can define are described at length in the following chapters.

Trick To Change Default Things (Advanced)

Suppose, say, you want to change some characteristics of creature 91 (Black Shade) in the default list of creatures. You don't need to change the default scripts. You can change them in your Custom Object Script. You do this with lines like this:

```
begindefinecreature 255; // note this erases whatever was in creature 255
       import = 91;
begindefinecreature 91; // Now creature 91 is filled with a copy of creature 255
       [Make all changes you want here.]
```

Note that you can't just go and put the line

```
begindefinecreature 91;
```

in your Custom Object Script, because it will fill creature 91 with a copy of whatever creature you've just edited.

Just a useful trick.

Chapter 2.3: Quick Introduction to Graphics

Everything you define can have at least one graphic icon attached to it. For example, every item has two icons: the small icon drawn in the terrain when it is on the floor, and the larger icon used when the item is on the getting screen or in your pack.

Blades of Avernum comes with a very large assortment of icons you can use for your new floors, etc. If you are ambitious, you will learn much later how to put custom graphics into the scenario. Most of the time, though, you will want to use the graphics that come with the game.

Looking at Macintosh Graphics

In Macintosh, all of the graphics for the game are in what is called a Resource File, a package of individual graphics packed into one file. The graphics are in the folder Blades of Avernum Files, in the files:

Terrain Graphics - All of the terrain icons.
Items Graphics - All of the item icons.
Scen Icon Graphics - The 64 x 64 icons that represent scenarios are dialogue pictures for characters.
Character Graphics – The creature graphics.

To look at resource files, your best option is to use an old program by Apple called ResEdit. You can download a copy in the Scenario Workshop at Spiderweb's web site (http://www.spiderwebsoftware.com). Run ResEdit and open the resource file. ResEdit is old, clunky, and doesn't run native in OSX, the result of Apple's unfortunate decision to not continue supporting a free, easily accessed resource editor.

(You can also use an awkward, clunky, expensive program called Resorcerer to view and edit resources. It does have the advantage of being OSX native.)

To view the graphics, run ResEdit, open one of the graphics files, and double-click on the PICT icon. (The graphic resource type for the Macintosh is called PICT.) You can see a list of all of the graphics in the game, each with a number below it. Click on a graphic. Double-click on a graphic to see it at full size.

To add a graphic to a resource file, copy it in a graphic editing program and paste it onto ResEdit. To set its number, click on it and select Get Resource Info from the Resource menu. To delete a graphic (a very bad idea in the game's graphics files), select it and press the delete key.

In ResEdit, the number below each graphic is the graphic's "sheet number". This is the number the game uses to identify the graphic.

Looking at Windows Graphics

The Windows situation is much simpler than the Mac situation. Open the folder Blades of Avernum Files. You will see folders called

Terrain Graphics - Contains all of the terrain icons.
Item Graphics - Contains all of the item icons.
Scen Icon Graphics - Contains all of the 64 x 64 icons that represent scenarios are dialogue pictures for characters.
Character Graphics – The creature graphics.

You can open, view, and edit these images using MS Paint (comes with Windows), or any other graphics editing program.

The format of any graphic for Blades of Avernum is:

```
G[number].bmp
```

For example, G1003.bmp, G6.bmp, etc.

The number in the file name is the graphic's "sheet number". This is the number the game uses to identify the graphic.

Leave the Blades of Avernum Graphics Alone

The graphics that come with Blades of Avernum are shared by all scenarios, so you are not allowed to edit them. If you want your own graphics, you will need to read the section on custom graphics later on.

The Meaning of a Sheet

In Macintosh, every graphics containing a block of icons in ResEdit is called a sheet.

In Windows, every Bitmap file containing a block of icons is called a sheet.

Sheets all contain either 1 graphic or a block of icons (for terrain types, creatures, etc.). For terrain (or items), one sheet can contain icons for lots of different terrain types (or items). For creatures, one sheet contains all of the icons for that creature.

When identifying a graphic for a new floor, terrain type, etc., you will need to provide the sheet number and, sometimes, the number of an icon in the sheet.

Identifying Graphics For Floor and Terrain Types

Floor and Terrain graphics are in the Terrain Graphics folder (Windows) or Terrain Graphics resource file (Mac).

Floor tiles need to have a very specific diamond shape. Look at sheets 700-702 to see what floor tiles look like. Floors won't look right if their graphics aren't of exactly this shape.

There are two sorts of graphics for terrain types. The first are the icons drawn in the terrain areas. These are 46 x 56 pixels. A sheet of terrain icons can contain any number of terrain icons, arranged in rows of 10. (look at sheets 600 and 790, for example.) Icons are identified by their number in the sheet. The 10 icons in the first row are 0 .. 9, the 10 icons in the next row are 10 .. 19, and so on.

The second sort of icon is the small icon (16 x 16 pixels) used in the editor and for the automap. Once again, these are arranged in sheets in rows of 10 (see sheets 680 and 681 for example). Icons are identified by their number in the sheet. The 10 icons in the first row are 0 .. 9, the 10 icons in the next row are 10 .. 19, and so on.

You can't put large and small icons in the same sheet.

Identifying Graphics For Item Types

Item graphics are in the Item Graphics folder (Windows) or Item Graphics resource file (Mac).

All item graphics are 28x28. These are arranged in sheets in rows of 10 (see sheets 1000 to 1010 for examples). Icons are identified by their number in the sheet. The 10 icons in the first row are 0 .. 9, the 10 icons in the next row are 10 .. 19, and so on.

Identifying Graphics For Character Types

Character graphics are in the Character Graphics folder (Windows) or Character Graphics resource file (Mac). Each sheet contains the icons for one and only one character.

Adjusting the Icons

Every time you can select an icon for something, you can give an adjustment value to it. You can make color shifts in any icon to make it look a little different. You can make it lighter, darker, switch red and green pixels, and so on. With some practice and experimentation, you can get much more variety in your graphics.

Some of the characteristics for icon adjustment are fl_icon_adjust (floors), cr_icon_adjust (creatures), and it_icon_adjust (items).

The icon adjust values for anything defaults to 0. By adding numbers to it, you can get color shifts. The values to add to get effects are:

+1 - Swap red and blue.
+2 - Swap blue and green.
+4 - Swap green and red.
+8 - Tint graphic to a neutral shade.
+16 - Darken the graphic.
+32 - Lighten the graphic.
+64 - Invert all of the pixels.
+128 – Tint the graphic red
+256 – Tint the graphic green
+512 – Tint the graphic blue

You can use several of these modifiers. For example, if you give an icon adjust value of 18 (16 + 2), the game swaps blue and green in the icon and then darkens it.

Note that these adjustments are not drawn in the editor, only the game.

Chapter 2.4: Creating Custom Floor Types

This chapter describes all of the characteristics of floor types, and the values you can put in them. Remember, you should only edit floors in your scenario's Custom Objects Script. The format of such a script is described in the previous chapter.

The format for defining a new floor is:

```
begindefinefloor [number of floor];
	[optional clear or import command];
```

And then a number of lines, each of which has the following format:

```
[name of integer characteristic] = [new value];
[name of string characteristic] = [new string];
```

Remember, every line must end with a semicolon.

An Example

Here is a sample definition of two floors:

```
begindefinefloor 4;
	clear;
	fl_name = "Rough Stone Floor";
	fl_which_icon = 4;
	fl_ed_which_icon = 4;
	fl_is_ground = 0;
	fl_is_rough = 1;
	fl_out_fight_town_used = 1001;
begindefinefloor 5; // this line makes floor 5 a copy of floor 4
	fl_which_icon = 5;
	fl_ed_which_icon = 5;
```

This bit of script begins to define floor 4 and sets a half-dozen characteristics of the floor. The line "begindefinefloor 5;" starts defining floor 5 and makes floor 5 a copy of the most recently edited floor (floor 4). The final two lines change two characteristics for floor 5.

The characteristics of a floor are described below. Note that all floor characteristics begin with "fl_".

String Characteristics

fl_name -The name of the floor. At most 19 characters long.

Integer Characteristics

fl_which_sheet - The number of the sheet for this floor's icon. (For more about graphics characteristics, read the earlier Graphics Introduction chapter.)
fl_which_icon - The number of the floor type's icon in the sheet.
fl_icon_adjust - The icon adjustment for the floor type.
fl_ed_which_sheet - The sheet for the 16x16 icon used for the floor type in the editor and on the automap.
fl_ed_which_icon - The number of the editor icon in the sheet.
fl_blocked (Defaults to 0) - If set to 1, this terrain is impassable. Characters can't walk over it.
fl_step_sound (Defaults to -1) - The sound that plays when the space is stepped on. Sounds are listed in the appendices. If left at -1, plays the default step sound.
fl_light_radius (Defaults to 0) - The radius of light this floor throws out.
fl_floor_height_pixels (Defaults to 0) - The number of pixels up graphics and creatures drawn on this space are shifted.

fl_special_property (Defaults to 0), **fl_special_strength** (Defaults to 0) - Marks the floor type as having a special property. If fl_special_property is left at 0, there is nothing special. fl_special_property defines what happens, and fl_special_strength defines how intense the effect is. The possible values for fl_special_property are:

0 - Nothing.

1 - Does 2 dice of fire damage to a character who steps on it. fl_special_strength is the size of the dice.

2 - Does 2 dice of cold damage to a character who steps on it. fl_special_strength is the size of the dice.

3 - Does 2 dice of magic damage to a character who steps on it. fl_special_strength is the size of the dice.

4 - Inflicts 1 die levels of poison on a character who steps on it. fl_special_strength is the size of the die. There is a 50% chance of the poison being inflicted.

5 - Inflicts 1 die levels of disease on a character who steps on it. fl_special_strength is the size of the die. There is a 50% chance of the disease being inflicted.

6 - Blocked to NPCs. Creatures will not step on this space and the game will not place character on it (when ending combat for entering a town).

7 - No rest. When on this floor type outdoors, you can not rest.

8 - Call scenario script. When this floor is stepped on, calls state fl_special_strength in the scenario's special script.

fl_is_water (Defaults to 0) - If 1, this space is considered to be water. Boats can move over it and other water spaces will conform to it when painting water in the editor.

fl_is_floor (Defaults to 0) - If 1, this a stone floor. Horses can't step on it. Water, dirt, and other terrain types will conform to it when painting terrain in the editor.

fl_is_ground (Defaults to 0) - If 1, this is plain ground (cave floor or grass). Not currently used for anything.

fl_is_rough (Defaults to 0) - If this, this floor is considered rough cave floor (underground) or dry ground (surface). When painting those ground types in the editor, they will conform to this floor type.

fl_fly_over (Defaults to 0) - If this floor is blocked but the party is flying, they can pass over it.

fl_shortcut_key - The number of a letter of the alphabet ('a' is 0, 'z' is 25). When that letter is typed in the editor, this floor type is selected. If several floor/terrain types have the same shortcut key, typing it repeatedly cycles through them.

fl_anim_steps (Defaults to 0) - Lets you set this floor to be animated. If left at 0 or 1, no effect. If greater than 1, when drawing this floor, the icon cycles between its base icon (set above) and the fl_anim_steps next icons in the sheet. (Example: If the floor's set icon in the sheet is 5 and fl_anim_steps is 4, the game alternates between icons 5, 6, 7, and 8 to draw the floor.)

fl_shimmers (Defaults to 0) - If 0, nothing. It 1, the floor type is drawn shimmering from light to dark and back. If 2, the floor is drawn with the blue "water shimmer" effect (look at any water tile for an example).

fl_out_fight_town_used (Defaults to 1000) - The number of the 48x48 town terrain to use for outdoor fights taking place on this floor. For more on the numbers used to select fighting terrain, read the section Terrain For Outdoor Encounters in the chapter editing the outdoors.

Array Characteristics

None for floors.

Chapter 2.5: Creating Custom Terrain Types

This chapter describes all of the characteristics of terrain types, and the values you can put in them. Remember, you should only edit terrain in your scenario's Custom Objects Script.

The format for defining a new terrain is:

```
begindefineterrain [number of terrain];
       [optional clear or import command];
```

And then a number of lines, each of which has the following format:

```
[name of integer characteristic] = [new value];
[name of string characteristic] = [new string];
```

Remember, every line must end with a semicolon.

Terrain types can be drawn in cutaway view. This means that the terrain type is drawn with a second icon if the area behind it is seen by the party, so as not to cover that area. A great example of this is walls, which are drawn short if the party is behind them. This is called being drawn in cutaway view. For terrain types, you can specify alternate icons to be drawn in cutaway view.

The characteristics of a terrain type are described below. Note that all terrain type characteristics begin with "te_".

String Characteristics

te_name - The name of the terrain type. At most 19 characters long.
te_default_script (Defaults to "Unused") - The name of the default script for the terrain type. At most 13 characters long. If left at "Unused" (which you will want to do the vast majority of the time), no script is attached.

Integer Characteristics

te_which_sheet - The number of the sheet for this terrain type's icon. (For more about graphics characteristics, read the earlier Graphics Introduction chapter.)
te_which_icon - The number of the terrain type's icon in the sheet.
te_icon_adjust - The icon adjustment for the terrain type.
te_ed_which_sheet - The sheet for the 16x16 icon used for the floor type in the editor and on the automap.
te_ed_which_icon - The number of the editor icon in the sheet.
te_cutaway_which_sheet (Defaults to -1) - If this is left at -1, the terrain type has no cutaway view. Otherwise, this is the number of the sheet to get the cutaway view icon from.
te_cutaway_which_icon - The number of the terrain type's cutaway icon in the sheet. Only has an effect if te_cutaway_which_sheet is not -1.
te_cutaway_icon_adjust - The icon adjustment value for the cutaway icon. Only has an effect if te_cutaway_which_sheet is not -1.
te_icon_offset_x, te_icon_offset_y (Defaults to 0) - The pixel offset when the terrain type is drawn. Can be negative. For te_icon_offset_x, negative is left, and for te_icon_offset_y, negative is up. Example: if, for a terrain type, te_icon_offset_y is -10, the icon is drawn shifted up a little.
te_second_icon (Defaults to -1) - It is possible to have a terrain type be composed of two icons. Examples are large pillars and statues, where one half of the icon is drawn on top of the other half. If left at -1, no second icon. Otherwise, this is the number of the second icon for the terrain type must be in the same sheet as the first icon (i.e. the sheet given for te_which_sheet).
te_second_icon_offset_x, te_second_icon_offset_y (Defaults to 0) - The pixel offset for the second icon. Can be negative. For the x offset, negative is left, and for the y offset, negative is up. Example: if, for a terrain type, te_second_icon_offset_y is 15, the icon is drawn shifted down 15 pixels.
te_cutaway_second_icon (Defaults to -1) - The second icon to draw if the terrain is being drawn in cutaway view. If left at -1, no second icon.

te_anim_steps - Lets you set this terrain type to be animated. if left at 0 or 1, no effect. If greater than 1, when drawing this terrain type, the icon cycles between its base icon (set above) and the te_anim_steps next icons in the sheet. (Example: If the terrain type's set icon in the sheet is 5 and te_anim_steps is 4, the game alternates between icons 5, 6, 7, and 8 to draw the terrain.)

Note that, if the terrain type has a cutaway view or second icon (or both), you need to provide animation frames for them too.

te_move_block_n, te_move_block_w, te_move_block_s, te_move_block_e (Defaults to 0) - If set to 1, the terrain blocks movement through to the north, south, east, or west. If all are set to 1, the terrain spot is completely blocked and nothing can be put there.

te_look_block_n, te_look_block_w, te_look_block_s, te_look_block_e, (Defaults to 0) - If set to 1, the terrain blocks vision through to the north, south, east, or west.

te_blocks_view_n, te_blocks_view_w, te_blocks_view_s, te_blocks_view_e (Defaults to 0) - If set to 1, the terrain, when drawn in its regular (not cutaway) view, blocks the player's view of the terrain behind it. If te_blocks_view_n is 1, the terrain's blockage is at the north end of the space, if te_blocks_view_w is 1, the obstruction is at the west edge, and so on. This helps the game figure out when to draw a terrain in its cutaway view.

te_height_adj_pixels (Defaults to 0) - The number of pixels up to shift any creature or item drawn in this space. For example, if a table is on the space, everything on the table needs to be shifted up to look like they're on the table, so te_height_adj_pixels may be set to 14 or so.

te_suppress_floor (Defaults to 0) - If set to 1, the floor below this terrain is completely covered. It will not be drawn, and stepping on it will not trip any effects (like damage).

te_light_radius (Defaults to 0) - The radius of light this terrain type throws out.

te_step_sound (Defaults to -1) - The sound that plays when the terrain is stepped on. Sounds are listen in the appendices. If left at -1, plays the default step sound. Terrain sounds have precedence over floor sounds, so if there is a sound for the terrain, the sound for the floor below won't be played.

te_shortcut_key - The number of a letter of the alphabet ('a' is 0, 'z' is 25). When that letter is typed in the editor, this terrain type is selected. If several floor/terrain types have the same shortcut key, typing it repeatedly cycles through them.

te_crumble_type (Defaults to 0) - Terrain can be set to crumble to dust when a beam strikes it or a Move Mountains spell is cast near it. If left at 0, won't crumble. If 1-5, when a Move Mountains spell is cast nearby at at least that skill, the terrain crumbles. If 6 or above, the terrain can only ever crumble if it is struck by a beam.

Beams will not crumble this terrain unless te_beam_hit_type is set to 3. If it is and this value is not 0, any beam will always crumble this terrain.

te_terrain_to_crumble_to (Defaults to 0) - The terrain type to replace this terrain with when it crumbles.

te_beam_hit_type (Defaults to 0) - How the terrain reacts when struck with a beam. If left at 0, ordinary behavior: if the terrain can be seen and walked through, the beam passes through. Otherwise, it is blocked. If set to 1, the beam always passes through. If 3, the terrain can be crumbled by a beam (though it won't be unless te_crumble_type is set to something other than 0).

te_hidden_town_terrain (Defaults to -1) - If this value is at least 0 and this terrain is being drawn outdoors and the space is inside the entry rectangle of a town that is hidden, the terrain is drawn as if it was terrain te_hidden_town_terrain, rather than itself. This is used to conceal towns that are not visible to the player.

te_swap_terrain (Defaults to -1) - When the flip_terrain call is used on a space containing this terrain type, if this value is set to something besides -1, the terrain in this space is changed to te_swap_terrain. You set this for terrains that are often changed to another terrain type (for example, doors like to change to open doors, and levers in the left position like to change to levers in the right position).

te_is_bridge (Defaults to 0) - If 1, this terrain is a bridge. If over a blocked floor, the party can still walk over it.

te_is_road (Defaults to 0) - If 1, this terrain is a road. This is mainly used for outdoor groups of creatures that are set to walk along roads.

te_can_look_at (Defaults to 0) - If 1, when a player presses the look button in the game, this is one of the spaces he or she is given the option of searching. Boxes, special terrains that you want the player to be able to search, and interesting and unusual things should have this set to 1.

te_special_property, te_special_strength (Defaults to 0) - Marks the terrain type as having a special property. If te_special_property is left at 0, there is nothing special. te_special_property defines what happens, and te_special_strength defines how intense the effect is. The possible values for te_special_property are:

Basic Properties:

0 - Nothing.
1 - Does 2 dice of fire damage to a character who steps on it. te_special_strength is the size of the dice.
2 - Does 2 dice of cold damage to a character who steps on it. te_special_strength is the size of the dice.
3 - Does 2 dice of magic damage to a character who steps on it. te_special_strength is the size of the dice.
4 - Inflicts 1 die levels of poison on a character who steps on it. te_special_strength is the size of the die. There is a 50% chance of the poison being inflicted.
5 - Inflicts 1 die levels of disease on a character who steps on it. te_special_strength is the size of the die. There is a 50% chance of the disease being inflicted.
6 - Blocked to NPCs. Creatures will not step on this space and the game will not place characters on it (when ending combat or entering a town).
7 - No rest. When on this terrain type outdoors, you can not rest.
8 - Call scenario script. When this terrain is stepped on, calls state te_special_strength in the scenario's special script.

Hill Properties:

These indicate that the terrain is a hill, stairway, or other way for the party to move safely from one height level to another.

19 - Hill or stairway, up to west
20 - Hill or stairway, up to southwest
21 - Hill or stairway, up to south
22 - Hill or stairway, up to southeast
23 - Hill or stairway, up to east
24 - Hill or stairway, up to northeast
25 - Hill or stairway, up to north
26 - Hill or stairway, up to northwest
27 - Hill or stairway, down to southeast
28 - Hill or stairway, down to northeast
29 - Hill or stairway, down to northwest
30 - Hill or stairway, down to southwest

Beam Properties:

Beam properties are only used for beam projectors and other terrain types that interact with beams.

31 - This terrain switches to the terrain set in te_swap_terrain when it is hit by beam.
32 - Fires beam north, but only if a power source is next to it.
33 - Fires beam west, but only if a power source is next to it.
34 - Fires beam south, but only if a power source is next to it.
35 - Fires beam east, but only if a power source is next to it.
36 - Northwest/southeast angled mirror.
37 - Northeast/southwest angled mirror.
38 - Beam power source.

Other Sorts of Terrain:

39 - This terrain is a sign. When placed, the scenario designer is prompted for its text. When searched, the player sees it.
40 - This terrain is a container. When an item is placed on the same space, it is marked as Contained and is invisible. The party can get it when they search this space.

41 - Acts as table. This space is a table. Players can get items placed on it, even if the space is blocked. (Normally, items on blocked terrain can't be gotten.)
42 - The terrain's icon shimmers light and then dark.
43 - Waterfall - south. If a player is on a boat and moves to the space just north of this terrain, they are transported 2 spaces south and lose some food.
44 - Waterfall - east. Like the other waterfall, but takes the party west to east.
45 - Terrain type is destroyed by quickfire. When quickfire touches it, it is immediately turned to the terrain type defined in te_terrain_to_crumble_to.

te_draw_on_automap (Defaults to 0) - If 1, this terrain is drawn on the automap (the game selects 16 pixels from the editor graphic).
te_full_move_block - If set to 1, the terrain is immediately set to be blocked to movement in all directions. If set to 0, terrain is completely unblocked.
te_full_look_block - If set to 1, the terrain is immediately set to be blocked to vision in all directions. If set to 0, terrain is completely unblocked.
te_shimmers (Defaults to 0) - If set to 1, the terrain glows light and dark.
te_out_fight_town_used (Defaults to -1) - The number of the 48x48 town terrain to use for outdoor fights taking place on this terrain type. For more on the numbers used to select fighting terrain, read the section Terrain For Outdoor Encounters in the chapter editing the outdoors.

If left at -1, the terrain for the floor space underneath is used. If a battlefield for this terrain type is set, it takes precedence over the battlefield set for the floor underneath

Array Characteristics

None for terrain types.

Chapter 2.6: Creating Custom Creature Types

This chapter describes all of the characteristics of creature types, and the values you can put in them. Remember, you should only edit creatures in your scenario's Custom Objects Script.

The format for defining a new creature is:

```
begindefinecreature [number of creature];
       [optional clear or import command];
```

And then a number of lines, each of which has the following format:

```
[name of integer characteristic] = [new value];
[name of string characteristic] = [new string];
[name of array characteristic] [number of list entry to change] = [new value];
```

Remember, every line must end with a semicolon.

Creatures are much like characters. The have health, mana, exactly the same sorts of skills, and they can have items in their inventory. They equip and use whatever weapons and armor they start with. They can pick up and use items (like wands and scrolls).

A scenario can have up to 255 creature types, numbered 1 to 255. Creature type 0 is reserved to be a Null (empty) creature.

Level and Statistics

The most important characteristic for a creature is its level. That single number determines everything about its starting strength. A creature's power increases exponentially with its level. A level 20 creature is far more than twice as strong as a level 10 creature.

A creature's hit points and damage are determined using the same formulas as a character's. The creature's starting statistics are as follows:

Strength = Level / 3 + 1
Dexterity = Level / 3 + 1
Intelligence = Level / 2 + 1
Endurance = Level / 2 + 1
All Weapons Skills = Level / 4 + 1

If you want to change a creature very quickly and easily, use the set_name and set_level calls. You can instantly make a whole new creature type.

For high level enemies with no special abilities and only one attack, the base statistics will probably not be enough to give the party a challenge. You may need to use the cr_what_stat_adjust and cr_amount_stat_adjust fields to tweak the creature's strength and dexterity the precise amount to enable it to crack through strong armor without devastating the character.

The maximum level for any creature is 100.

About Creature Graphics

Look at sheets 1453 and 1500 in Character Graphics.

Sheet 1453 is the small sort of creature graphic. The first row contains the standing pose of the graphic, facing north, west, south, and east. The second row is the attacking pose. The third row is the death animation. Below that are the tiny outdoor icons.

Sheet 1500 is the second, larger sort. There are additional icons for being in combat mode and for the character sitting in a chair. If a creature does not have a sheet of icons of this sort and is in the same space as a chair, the character will just be drawn standing in front of it.

The characteristic cr_small_or_large_template sets whether the game should expect a small or large sheet of icons for it.

The characteristics of a creature type are described below. Note that all creature type characteristics begin with "cr_".

Giving Creatures Abilities

There are several ways to give creatures interesting abilities.

You give creatures the ability to cast spells by increasing their Mage Spells and Priest Spells skills. The creature will know any spell its skill enables it to cast. For example, the following lines in the definition of a Wizard makes it a powerful caster:

```
cr_what_stat_adjust 1 = 11; // Mage Spells is skill 11
cr_amount_stat_adjust 1 = 20; // Sets the skill to 20
```

Creatures start out knowing spells at a default skill of 2. If you want to change the skill a character has with a spell, use the call change_spell_level.

You can give creatures special abilities by setting the cr_special_abil characteristic for the creature.

Finally, you can hardcode the abilities into the creature's script. Look at the sample script abilnpc.txt or spawner.txt for examples of how this might work.

String Characteristics

cr_name - The name of the creature type. At most 19 characters long.

cr_default_script (Defaults to "basicnpc") - The name of the default script for the creature type. At most 13 characters long. If left at "basicnpc" (which you will want to do a lot of the time), uses the basic default creature script.

Integer Characteristics

cr_level (Defaults to 2) - The level of the creature.

cr_hp_bonus (Defaults to 0) - The number of extra hit points the creature gets.

cr_sp_bonus (Defaults to 0) - The number of extra spell points the creature gets. NPC casters drain spell points like spell casters in the party. Serious spell casters should probably get a good pool of spell points.

cr_special_abil (Defaults to 0) - The creature's special ability. Its intensity depends on the level of the character.

A note about melee abilities that function when this character attacks (like abilities 1 and 6). They only work for the creature's innate attacks (defined in cr_attack_1, cr_attack_2, and cr_attack_3), so if you give the creature a weapon, the melee ability won't work. The melee ability takes affect for every blow that does damage.

Most of the ranged attacks have a maximum range of 8. Exceptions are noted below.

0 - None.

1 - Melee attack poisons target.

2 - Fires a ray that curses and weakens the target.

3 - Stoning ray. Has a chance of turning the target to stone. This should, of course, be used very sparingly.

4 - Melee attack slows target.

5 - Throws webs at the target. Has a maximum range of 4.

6 - Melee attack diseases target.

7 - Fires charm ray at target.

8 - Melee attack puts target to sleep.

9 - Fires sleep ray at target.

10 - Melee attack paralyzes target.

11 - Fires paralysis ray at target.

12 - Melee attack covers target with acid.

13 - Fires dumbfounding/spell point draining ray at target.

14 - Fires confusion ray at target.

15 - Fires terrify/enfeeble ray at target.

16 - Throws rocks, which do a fair amount of damage. This ability is used less often than the others.

17 - Breathes fire at target. (Breath weapons are actually not very powerful. The most effective area effect attacks are created with scripts. See t13kimzahn.txt in Za-Khazi Run for an example.)

18 - Breathes cold at target.

19 - Breathes acid at target.

20 - Melee attack does extra fire damage.

21 - Melee attack does extra cold damage.

22 - Melee attack drains experience.

23 - Melee attack does extra cold damage and drains experience.

24 - Creature is invisible. Can only be hit by melee attacks.

26 - Radiates a fire field.

27 - Radiates a cold field.

28 - Radiates an antimagic field.

29 - Divides in two when struck. New creatures use the default script for the creature, don't drop treasure, and don't give experience. Creatures with this ability don't regenerate health.

30 - Fires a fire ray at target.

32 - Fires an energy ray (magic damage) at target.

33 - Breathes darkness (damage + enfeeblement)

34 - Throws spines at target. Does good damage and paralyzes.

35 - Creates forcecage around target.

36 - Melee attack webs target.

37 - Radiates a sleep cloud.

38 - Radiates a stinking cloud.

39 - Radiates a blade field.

cr_default_attitude (Defaults to 2) - Sets the attitude the creature has when first placed (which can be changed later). 2- friendly. 3 - neutral. 4 - hostile A. 5 - hostile B.

cr_species (Defaults to 0) - The species of the creature. The species can affect what attacks the creature is vulnerable to. Creatures with species 0-4 are vulnerable to attackers with Anatomy skill. The spells Divine Retribution and Ravage Life only affect creatures of species 0-6.

0 - Human.

1 - Humanoid.

2 - Nephil. Gets bonus with Missile Weapons and Dexterity.

3 - Slith. Gets bonus with Pole Weapons and fire resistance.

4 - Giant

5 - Reptile. Won't pick up or use items.

6 - Beast. Won't pick up or use items.

7 - Demon. Won't pick up or use items. Vulnerable to Repel Spirit at high level. Immune to Mental spells.

8 - Undead. Won't pick up or use items. Vulnerable to Repel Spirit. Immune to Mental spells.

9 - Insect. Won't pick up or use items. Immune to Mental spells. Immune to webs.

10 - Slime/plant. Immune to assassination. Won't pick up or use items. Immune to Mental spells.

11 - Stone/golem. Immune to assassination. Won't pick up or use items. Immune to beams.

12 - Special. Immune to assassination and Lethal Blow. Immune to Simulacrum. Immune to webs.

13 - Vahnatai.

14 - Other.

cr_natural_armor (Defaults to 0) - If greater than one, damage from any melee blow is reduced by one die of size cr_natural_armor. (So if this is 15, every blow is reduced by 1 to 15 points.)

cr_attack_1 (Defaults to 0) - Gives the creature an innate attack. If greater than 0, this creature has a natural melee attack whose base damage is a die of this size. The attack will do a number of dice equal to the creature's strength (before bonuses). So if this value is 7 and the creature's strength is 5, attacks to a base of 5d7 damage (5 to 35 points).

A character will only use its innate attacks if it does not have a weapon. Effectiveness of innate attacks depends on the Melee Weapons skill (so each additional level of Melee Attacks increases the hit change by 5% and damage by one die).

cr_attack_2, cr_attack_3 (Defaults to 0) - Two additional innate melee attacks.

cr_attack_1_type (Defaults to 0) - The text description of the attack defined in cr_attack_1. Possible values are:

0 - Strike
1 - Claw
2 - Bite
3 - Slimes
4 - Punches
5 - Stings
6 - Clubs
7 - Burns
8 - Harms
9 – Stabs
10 - Kicks

cr_attack_23_type (Defaults to 0) - Like cr_attack_1_type, but affects the text description of the attacks defined by cr_attack_2 and cr_attack_3.

cr_ap_bonus (Defaults to 0) - the number of bonus action points the character gets every turn.

cr_default_strategy (Defaults to 0) - The default strategy for the creature when it tries to attack.

0 - Default, regular behavior. Creature runs up to nearest foe and fights.

1 - Archer/caster, so maintains a safe distance when possible.

10 - Default, regular attack, this creature will never switch its target unless you change it in the script.

11 - Archer/caster, so maintain distance, this creature will never switch its target unless you change it in the script.

cr_default_aggression (Defaults to 100) - The percentage chance that a character takes a foe as a target at a distance. The creature might decide not to attack. Also the percentage to reduce the maximum range a creature can get targets at. (For example, if a script tries to get a target within 8 spaces and the aggression is at 75, the creature will only get targets within 6 spaces, and only has a 75% chance of taking a target at all).

A note about aggression. If a character tries to get a target and its aggression level prevents it, the creature's turn ends immediately.

cr_default_courage (Defaults to 100) - The percentage chance that a wounded creature has of not running away. The more damaged the character is, the more often it checks whether it flees. A character whose courage is 0 always flees.

cr_which_sheet - The number of the sheet for this creature's icon. (For more about graphics characteristics, read the earlier Graphics Introduction chapter.)

cr_icon_adjust (Defaults to 0) - The icon adjustment for the creature.

cr_small_or_large_template (Defaults to 0) - If 0, the game expects the smaller type of creature icon sheet (described at the beginning of this chapter). If 1, the game expects the bigger sheet.

cr_which_sheet_upper (Defaults to -1) - You can have a creature be two icons high. If this is left at -1, the creature is 1 icon high. Otherwise, this is the sheet of the upper half of the creature. (For an example, look at sheets 1608 and 1609 in Character Graphics.)

cr_summon_class (Defaults to -1) - Determines whether a creature can be summoned by summoning spells. If left at -1, no. Otherwise, the higher the number, the more powerful a spell it takes to summon it (5 and above means it is very, very difficult to summon).

Array Characteristics

Remember the format to change an Array Characteristic is:

```
[name of array characteristic] [number of list entry to change] = [new value];
```

For example, if you want to have the creature's third item slot have item 232, the line would look like

```
cr_start_item 2 = 232;
```

An array is a list. So to change an array characteristic, you have to give the number of the member of the list to change, and then what to change it to.

cr_what_stat_adjust, cr_amount_stat_adjust - Each creature type has a list of 6 different statistic adjustments it can have. You can change 6 of the creature's base statistics. This line will set what statistic this adjustment is for. Cr_amount_stat_adjust sets how much to change the statistic selected in cr_what_stat_adjust by. Example:

```
cr_what_stat_adjust 0 = 2;
cr_amount_stat_adjust 0 = 3;
cr_what_stat_adjust 1 = 4;
cr_amount_stat_adjust 1 = 8;
```

The first line says the first adjustment (0 in the list of statistic adjustments) is to statistic 2 (Intelligence). The next line says the adjustment is 3 (so Intelligence is 3 higher). The third line says the second adjustment is to statistic 4 (Melee Weapons) and the fourth line says the skill is increased by 8.

cr_start_item, cr_start_item_chance - Each creature has a list of 8 different items it can start out with. cr_start_item is the number of the item and cr_start_item_chance is the percentage chance that it can have it. Example:

```
cr_start_item 0 = 91;
cr_start_item_chance 0 = 75;
cr_start_item 1 = 135;
cr_start_item_chance 1 = 100;
```

This creature type would have a 75% chance of starting with item 91 and a 100% chance of having item 135.

A creature can only start out with one weapon and piece of armor of each type. So if you set these value so the creature is given 2 shields, only the second shield given ends up with the character.

Finally, when a creature dies, to prevent the town from being clogged up with too many items, there is a chance that any non-magical, low-value item will disappear.

cr_immunities - Each creature has a list of 6 different immunities. You can give a percentage chance that the effects from these attacks will be reduced by. The immunities are

0 - Fire
1 - Cold
2 - Magic
3 - Mental
4 - Poison/acid
5 - Melee

If you give an immunity of 100, the creature will be totally immune. Example:

```
cr_immunities 1 = 15;
cr_immunities 5 = 100;
```

All cold damage against this creature would be reduced by 15%. The creature is totally immune to melee damage (which may not be a good design decision).

Chapter 2.7: Creating Custom Item Types

This chapter describes all of the characteristics of new types of items, and the values you can put in them. Remember, you should only edit items in your scenario's Custom Objects Script.

The format for defining a new item type is:

```
begindefineitem [number of creature];
	[optional clear or import command];
```

And then a number of lines, each of which has the following format:

```
[name of integer characteristic] = [new value];
[name of string characteristic] = [new string];
[name of array characteristic] [number of list entry to change] = [new value];
```

Remember, every line must end with a semicolon.

A scenario can have up to 499 creature types, numbered 1 to 499. Item type 0 is reserved to be a Null (empty) item.

Constant Item Types

You should be careful about editing the default item types, as many creatures give them as rewards. In addition, some of the default item types must be left alone. The game expects them to be unchanged from scenario to scenario. They are:

Food – Items 4 to 12 and 397 to 400.
Javelins, Arrows, Bolts - Items 84 to 88 and 99 to 108.
Alchemy Ingredients, Basic Potions, Scrolls - Items 214 to 263.
All the Tools - Items 171 to 178.
Crystals and Dust - Items 326 to 329 and 435 to 439.

String Characteristics

it_name - The name of the item the player sees if it isn't identified. At most 19 characters long.
it_full_name - The name of the item when it is identified. At most 29 characters long.

Integer Characteristics

it_variety (Defaults to 0) - What sort of item it is (e.g. armor, shield, potion). Determines when the item can be used, what inventory slot it goes into, etc. Values are:

0 - Null item. Doesn't exist.
1 - 1-handed weapon. Weapon that goes in the weapon inventory slot.
2 - 2-handed Weapon. Weapon that goes in the weapon inventory slot and prevents a shield from being equipped.
3 - Gold. When player tries to get it, it doesn't enter inventory. Instead player gets coins equal to the number of charges of the item.
4 - Food. When player tries to rest, one item of this variety in the party's inventory disappears.
5 - Thrown Missile. Weapon that goes in the missile inventory slot. Normally has charges, and throwing a missile uses up one charge. However, if charges are left at 0, the thrown missile can be used an unlimited number of times (example: a sling).
6 - Bow. Weapon that goes in the missile inventory slot. Firing it consumes items of variety 23.
7 - Potion. This is an item that has charges, allows itself to be combined with other items of the same type (so two healing items with 1 charge each combine in a pack into 1 healing potion with 2 charges), and uses a charge when used.
8 - Scroll. Works the same as potions.
9 - Wand/Rod. This is an item that has charges, does not allow itself to be combined with other items of the same type, and uses a charge when used.

10 - Tool. An item that doesn't have an ability that requires it to be used but has charges, and combines itself with like items in a character's inventory (such as torches or lockpicks).
11 - Pants. Armor that goes in the pants inventory slot. When wearer is struck, has 25% chance of blocking some of the damage.
12 - Shield. Armor that goes in the shield inventory slot. When wearer is struck, has 80% chance of blocking some of the damage.
13 - Armor. Armor that goes in the armor inventory slot. Can't be changed in combat. When wearer is struck, has 90% chance of blocking some of the damage.
14 - Helm. Armor that goes in the helm inventory slot. When wearer is struck, has 35% chance of blocking some of the damage.
15 - Gloves. Armor that goes in the gloves inventory slot. When wearer is struck, has 30% chance of blocking some of the damage.
16 - Boots. Armor that goes in the boots inventory slot. When wearer is struck, has 30% chance of blocking some of the damage.
17 - Cloak. Armor that goes in the cloak inventory slot. When wearer is struck, has 40% chance of blocking some of the damage.
18 - Ring. Item that goes into the ring inventory slot.
19 - Necklace. Item that goes into the necklace inventory slot.
20 - Bracelet. Item that goes into the bracelet inventory slot.
21 - Object. Just a general object. It can have charges and, if it does, drains one when the item is used. Combines with like items in inventory.
22 - Crossbow. Weapon that goes in the Missile inventory slot. Firing it consumes items of variety 24.
23 - Arrows. Equipped in the ammo inventory slot.
24 - Bolts. Equipped in the ammo inventory slot.

it_damage_per_level (Defaults to 0) - Only important for weapons. The size of the die of damage the weapon does. A blow with a weapon does a number of dice of damage equal to the attacker's strength plus the skill in the weapon plus the bonus.
it_bonus (Defaults to 0) - For weapons, the number of extra levels of skill the wielder is given. For armor, the number of extra points of damage it absorbs.
it_weapon_skill_used (Defaults to 4) - The skill the weapon uses. Skill types are listed in the appendices. Most often used values: 4 is Melee Weapons, 5 is Pole Weapons, 6 is Bows, and 7 is Thrown Missiles. However, you can have a weapon use any skill.
it_protection (Defaults to 0) - If the item is equipped and this value is greater than 0, blocks a die of that size worth of damage from each melee attack. For armor, there is a chance the item won't block the damage. See the percentage chances in the description of it_variety.
it_charges (Defaults to 0) - For money, the number of coins getting the item adds. For anything else, the number of uses it has.
it_encumbrance (Defaults to 0) - How much the item slows actions when equipped. Each level of encumbrance reduces chance to hit in combat by 5%. Too much encumbrance drains action points and prevents casting of mage spells.
it_floor_which_sheet - The number of the sheet for this item's icon when it is drawn on the floor in the terrain area. (For more about graphics characteristics, read the earlier Graphics Introduction chapter.)
it_floor_which_icon - The number of the icon to use when drawn on the floor in the terrain area.
it_icon_adjust - The icon adjustment for the item.
it_inventory_icon - The number of the icon to use when drawn in the inventory area or getting window. Note that the floor and inventory icons must be in the same sheet.
it_ability_1 (Defaults to -1), **it_ability_str_1** (Defaults to 0) - An item can have up to 4 special abilities. These two characteristics define the first ability. If it_ability_1 is left at -1, this item has no ability. it_ability_1 is the number of the ability, and it_ability_str_1 is the ability's strength.

All special abilities and the meaning of the strength value are defined below.

it_ability_2, it_ability_3, it_ability_4 (all default to -1**), it_ability_str_2, it_ability_str_3, it_ability_str_4** (all default to 0) - The other 3 special abilities and their strengths.

it_special_class (Defaults to 0) - A special value used by the scripts to identify a certain class of items. Script calls can tell if the party has an item of a certain class value (using calls like has_item of_class) and take an item of that class value. So, for example, if you wanted to check if the party had a buckler, you could change all the bucklers so they had an it_special_class value of 5, and then use the has_item of_class call to see if the party has an item of class 5.
it_value (Defaults to 0) - The value in coins of an item. If left at 0, the item can't be sold.
it_weight (Defaults to 0) - The weight of the item. The weight in pounds is this value divided by 10 (so 35 means 3 and a half pounds).
it_identified (Defaults to 0) - If 1, this item is always identified.
it_magic (Defaults to 0) - If 1, this item is magic. It is less likely to be deleted if the game runs out of room in a town, and it looks different in the getting window.
it_cursed (Defaults to 0) - If 1, item starts out cursed. When equipped, can't be removed until the curse is removed.
it_once_per_day (Defaults to 0) - If 1 and this item has an active ability, using it doesn't drain charges. Instead, the item can't be used until the beginning of a new day. Items that can be used once per day should have 0 charges.
it_junk_item (Defaults to 0) - If 1, the item is useless. The game is free to delete it if there are too many items in an area.
it_missile_anim_type (Defaults to 0) - This is only used for missiles. This is the animation to use for missiles. For a list of different missile types, see the appendices.

Item Special Abilities

An item can have up to 4 special abilities. Special abilities come in two types:

Passive Abilities - These are effects that affect the character when the item is worn. Only items that can be equipped can have passive abilities.
Active Abilities - These are effects that happen when the item is used. An item can only be used when it is in the character's pack. No item that can be equipped can have active abilities.

Some active abilities are spells or require targeting a foe. An item can only have one such ability. If it has more than one, only the last one will take effect.

No item can have both active and passive abilities.

The ability strength effects how potent the item it. The strength can almost always be negative, giving the item a harmful effect. These are the abilities, and the effect of the ability strength value:

0 - Does nothing.

1 - 49 - Affects the statistic equal to this value, minus one. So if 3, affects statistic 2 (Intelligence). If 12, affects statistic 11 (Mage Spells). Statistics are listed in the appendices. The strength is the amount to change the statistic by (which can be negative).

Note that you can't change Health or Spell Energy with an item, and changes to Intelligence and Endurance won't change Health or Spell Energy.

50 - Affects melee to hit chance. Each level of strength gives 5% bonus to hit
51 - Affects melee damage. Each level of strength gives one extra die of damage.
52 - Affects missile to hit chance. Each level of strength gives 5% bonus to hit
53 - Affects missile damage. Each level of strength gives one extra die of damage.
54 - Resist all hostile effects (fire, cold, magic, etc.). Each level of strength gives 5% resistance.
55 - Resist fire. Each level of strength gives 5% resistance.
56 - Resist cold. Each level of strength gives 5% resistance.
57 - Resist magic. Each level of strength gives 5% resistance.
58 - Resist mental. Each level of strength gives 5% resistance.
59 - Resist poison. Each level of strength gives 5% resistance.
60 - Resist acid, Each level of strength gives 5% resistance.
61 - Affect action points. Each turn, get a random number of extra action points from 0 to the strength. (So strength 2 means each turn in combat get 0, 1, or 2 action points).

62 - Affect all melee statistics. Changes by amount equal to strength. A skill must be at least 1 to be improved.

63 - Affect magic statistics. Changes by amount equal to strength. A skill must be at least 1 to be improved.

64 - Affect all statistics. Changes by amount equal to strength. A skill must be at least 1 to be improved.

65 - Affect rune reading skill. Changes by amount equal to strength.

70 - Does extra fire damage to target (only has effect on a melee weapon or ammunition). All abilities that do extra damage do an amount equal to ability strength.

71 – Sprays acid on target (doesn't damage instantly, only has effect on a melee weapon or ammunition).

72 - Does extra poison damage (only has effect on a melee weapon or ammunition).

73 - Does extra damage to humanoids (only has effect on a melee weapon or ammunition).

74 - Does extra damage to undead (only has effect on a melee weapon or ammunition).

75 - Does extra damage to demons (only has effect on a melee weapon or ammunition).

76 - Extra damage to giants (only has effect on a melee weapon or ammunition)

77 - Drains life from target, adds to life of wielder (only has effect on a melee weapon). Strength determines how much is drained.

78 - Does extra damage to reptiles (only has effect on a melee weapon or ammunition).

79 - Does extra damage to vahnatai (only has effect on a melee weapon or ammunition).

80 - Encumbrance. Each level of strength is one extra level of encumbrance.

81 - Resist melee damage. Each level of strength reduces damage from melee attacks by 1.

82 - Resist all damage. Each level of strength reduces damage from all sources by 1.

83 - Affects chance to be hit in combat. Each level of strength reduces chance by 5%.

84 - Protects from petrification. Each level of strength gives 5% resistance.

85 - Extra armor. Reduces damage from melee attacks by a dice of size equal to strength.

86 - Regenerate. Slowly heals damage. The more strength, the more is healed.

87 - Does damage to person hitting wearer with a melee attack. Does amount of damage up to ability strength.

88 - Lifesaver. Each level of strength gives 5% chance of not being killed by a fatal blow in combat.

89 - Does extra cold damage (only has effect on a melee weapon or ammunition).

ACTIVE ABILITIES (All of these abilities happen when the item is used, except when noted.)

100-119 - Casts a mage spell when used. The number of the mage spell is equal to the ability number, minus 100 (so 106 casts mage spell 6, Ice Lances). The spell is cast at a level of 1 plus the ability strength divided by 10. The bonus the spell is cast at is equal to the ability strength. Mage spells are listed in the Appendices.

Note that an item can only cast one spell, and that spell should be the last ability you give it.

120-139 - Casts a priest spell when used. The number of the priest spell is equal to the ability number, minus 120 (so 134 casts priest spell 14, Control Foes). The spell is cast at a level of 1 plus the ability strength divided by 10. The bonus the spell is cast at is equal to the ability strength. Priest spells are listed in the Appendices.

Note that an item can only cast one spell, and that spell should be the last ability you give it.

150 - 199 - Changes a status for the user. The status changed is equal to the ability number, minus 150 (so 168 affects Status 18, Confusion). Changes the status by an amount equal to the ability strength (which can be negative). Statuses are listed in the Appendices.

200 - Heals 5 points damage for each level of strength.

201 - Cures poison and disease. The higher the strength, the more cured.

202 - Changes experience by 10 times the ability strength (which can be negative).

203 - Changes skill points by the ability strength.

204 - Restores 5 points spell energy for each level of strength.

205 - Cleanses webs and disease. The higher the strength, the more cured

207 - Calls a state in the scenario script. The strength is equal to the state that is called. Doesn't lose a charge when used. Items with this ability can't be carried between scenarios. The state in the script can get the number of the character who used the item by using the call who_used_custom_item.

208 - Exactly the same as ability 207, except that the item loses a charge when used.

209 - Removes harmful mental effects from the entire party. The higher the strength, the more is removed. Strength shouldn't be negative for this ability.

210 - Gives 75 turns of light for each level of strength.

211 - Is a lockpick. This ability doesn't happen when the item is used. Each level of strength gives the holder a bonus skill level when picking locks. Strength shouldn't be negative for this ability.

212 - Is a First Aid Kit. This ability doesn't happen when the item is used. Each level of strength gives the holder a bonus skill level when doing first aid. Strength shouldn't be negative for this ability.

213 - Gives whole party flight for a number of turns equal to the strength.

214 - Afflicts user with numerous bad effects. The higher the strength, the worse it is.

215 - Permanently increase 1 statistic by one. The strength value is the number of the statistic changed.

216 - Permanently decrease 1 statistic by one. The strength value is the number of the statistic changed.

219 - Special script ability used on foe. This item, when used, asks you to select a foe. When the target is selected, a state in the scenario script is called. The number of the state called is equal to the ability strength. The script can then do something interesting or affect the target in some way. The state in the script can get the number of the character who used the item by using the call who_used_custom_item. The script can get the target by using the call who_is_custom_item_target.

Note that an item can only have one ability targeted on a foe, and this ability should be the last ability you give the item.

220 - Exactly the same as ability 219, except that the item loses a charge when used.

Transferring Items Between Scenarios

When a party leaves a scenario, all of the items in their inventory come with them. The exception is that some items will be lost:

Items that call states in the scenario script when used.
Items that have custom graphics.

In addition, items that have special classes will have those classes be set to 0.

When items with charges are transferred from another scenario, they will not combine properly with items from the new scenario (so apples from Scenario 1 will not combine with apples from Scenario 2). The exception to this is all of the constant item types listed at the beginning of the chapter. These will always combine properly.

Chapter 2.8: Programming with Avernumscript

So far, we have covered the scripts for defining objects in the game. These scripts are inert things. The game loads them when the scenario is loaded, reads them, digests their information, and moves on.

The rest of the script types, however, are much more complicated. They are run repeatedly as the game progresses, affecting the party and the world they are in. The different sorts of scripts are:

Scenario Script - Each scenario can have one. It defines various things in the scenario and has chunks of commands that can be accessed from anywhere in the scenario.
Creature Scripts - Every creature has one. It defines what the creature does.
Dialogue Scripts - Any town with people to talk to has one. It has all of the dialogue in the town.
Town Scripts - Each town can have one. Bits of code that determine all special encounters in the town.
Outdoor Scripts - Each outdoor section can have one. Bits of code that determine all special encounters in the outdoor section.

These scripts are full of commands (called calls) that affects things in the game. Calls can spawn or kill characters, shift around terrain, give the party items (or take them away) or do a multitude of other things. There are hundreds of calls, affecting practically every aspect of the game. A series of calls which does things in the game is referred to as "code".

All calls are listed in the section of the appendices. It is a big section, but the most commonly used calls are listed at the beginning.

Scripts are power but, to write code, you need to learn Avernumscript. Avernumscript is a very, very simple programming language, loosely based on C (the most popular programming language). It is easy to learn, as these things go.

(Note to people with experience programming: Avernumscript is not C. It was never intended to be. You will find that it is missing a lot of things you are used to, like arrays and pointers. These features, while desirable, also slow down the loading and running of the scripts. You should find that while Avernumscript doesn't have every feature you want, it is powerful enough to do everything you want it to do.)

Before you can start writing scripts, you need to learn about Avernumscript. If you have programmed a computer before, this will be easy. If you haven't, it will be hard. These instructions are not the best introduction to programming imaginable, but they will give you a fair chance of figuring out what is going on.

First, though, we need to cover the basics. We need to learn the basic vocabulary of programming.

Variables

Anyone who ever took a class in algebra should know what a variable is. It is a number referred to by a name. For example, say I am thinking of a number. You don't know what it is. I will only tell you a name for that number: "x". If I told you that if I added two to x, it would become 5, a little bit of thought would probably tell you that x must be 3. And so it is.

In programming, you can make names for numbers when you don't know what they will be yet (or which you may want to change). Variables are storage bins for numbers. Each one contains one number.

So your script may have a variable called, again, x, and you may have these lines:

```
x = 10; // x now represents the number 10
x = 15; // x now represents the number 15
x = x + 7; // takes x's old value, adds 7 to it, and puts it in x
x = 0; // x is now 0
```

Note that every command ends in a semicolon.

Mathematical Expressions

The world in Blades of Avernum is made of numbers. Pretty much everything is represented by a number. In Avernumscript, you will have lots of numbers to kick around.

Any number, or any formula that you can work out to get a number, is called a Mathematical Expression. Here are some examples:

```
1
6
15 - 12
(1 + 3 + 5) * 7
x - 3 // x is an integer variable
x + (y * z) // x, y, and z are integer variables
```

Mathematical Expressions can include all of the standard math symbols you're used to, and a few you may not have seen:

+, -, *, / - Plus, Minus, Times, Divided By. When dividing, the program loses remainders.
% (Advanced) - Modulus. When you write a % b, when the game does is divides a by b, and returns the remainder. So 8 % 3 = 2, and 15 % 5 = 0.

Note that, in all of the expressions above, there are spaces between all of the terms (so it's "x + (y * z)", not "x+(y*z)"). Place spaces between the expression terms, or the parser may not read them correctly.

When calculating an expression, Blades of Avernum uses standard precedence rules. Multiplication and division are done before subtraction, which is done before addition (so 2 * 3 + 1 is 7, not 8). However, operations are done from right to left, instead of the usual left to right (so 9 – 3 – 3 is 0, not 3, because the subtraction on the right is done first).

Setting the Value Of Variables

In the script's code, you can change the value of an integer variable. To do this, have on a line the name of the variable, an equals sign, a mathematical expression, and a semicolon. For example:

```
x = 5; // x is now 5
y = 13; // y is now 13
z = x + y; // z is now 18
```

Strings

Strings are bits of text. Just letters and numbers. When a character says dialogue or a special description of an area comes up, those bits of text are strings. Strings are, unless said otherwise, a maximum of 255 characters long. Strings in Avernumscript are always given in quotes, like

```
"This is a short string."
"This is a very very very very very very long string."
"" // this is an empty string.
```

Quotes are used to mark the beginnings and endings of strings. So what if you want your string to contain quotes? Well, when the game is drawing text and it comes across an underscore character ('_'), it will draw quotes. So use underscores in your text to represent quotes. For example,

```
"Suzy said, _I am talking now._"
```

How a Script Is Begun

Every script begins with an identification line.

```
beginscenarioscript; // Scenario Script
begincreaturescript; // Creature Script
begintalkscript; // Dialogue Script
begintownscript; // Town Scripts
beginoutdoorscript; // Outdoor Script
```

Next, Define Variables

Next, you will define the variables used in the script. To indicate you are about to define variables, use the line

```
variables;
```

Then define the variables. You can define two sorts of variables: integer and string. You define integer variables by starting a line with "short", giving the names of the variables, separated by commas, and ending with a semicolon. Examples:

```
short x; // one integer variable, named x
short fred, bob; // two integer variables, named fred and bob
```

You can also optionally initialize an integer value by putting an equal sign and then a number after its name. Examples:

```
short x = 10; // one integer variable, named x, which starts out equal to 10.
short fred = -13, bob_two; // two integer variables, named fred and bob_two. fred starts at -
13
```

If you don't initialize an integer value, it starts out with a random mystery value.

About variable names. A variable name can be up to 19 characters long and can have letters, numbers, and underscores ('_') in it. A variable name can't have spaces. Spaces are best represented by underscores. So you can't have a variable called "goblin health" but you can have one called "goblin_health".

You can also have string variables. A string variable has a name and then a bit of text up to 255 character long. You can not later change a string variable, so you need to initialize it. You define string variables like you define integer variables: the word "string", then the name of the variable, then an equals sign, then the string in quotes, then a semicolon. Such as:

```
string talk_string = "Suzy said, _I am talking now._";
string short_string = "Short string.";
```

String variables are called, in the programming world, constants. You can set them, but you can't change them. They are just there if there is one fixed string you will want to access again and again. You won't use them often.

One script can have up to 20 integer and string variables.

The game gets rid of scripts it has loaded in memory when it no longer needs them. When the party leaves a town, for example, the scripts for that town and all creatures in it are discarded. Values of variables are not remembered when this happens. If you set a variable in a town script and the party leaves and reenters that town, the value of that variable is forgotten.

Predefined Constants

There are a few constants in Avernumscript. These are like predefined variables that you can't reassign. The most important constants are:

TRUE (equal to 1)
FALSE (equal to 0)
ME (equal to –1)

You can also use constants to identify the more commonly used states in scripts (much, much more on this later):

INIT_STATE (equal to 0)
DEAD_STATE (equal to 1)
START_STATE (equal to 2)

EXIT_STATE (equal to 1)
START_SCEN_STATE (equal to 111)
SEARCH_STATE (equal to 100)
TALKING_STATE (equal to 110)
UNLOCK_SPELL_STATE (equal to 101)
SANCTIFICATION_STATE (equal to 102)
LOAD_SCEN_STATE (equal to 0)
BLOCK_MOVE_STATE (equal to 112)
DISPEL_BARRIER_STATE (equal to 113)
STEP_INTO_SPOT_STATE (equal to 114)

So, for example, these two lines of code are equivalent:

```
short i = 1;
```

and

```
short i = TRUE;
```

Defining the Body Of the Script

After variable definitions, the body of the script begins. Indicate this with this line:

```
body;
```

What comes after this depends on what sort of script it is. What is very likely, though, is that it will involve calls and flow statements. Nothing we have done so far is actual programming code. Now we need to learn how to write code. First, the basic building block of scripts: calls.

Using Calls

Calls are commands your scripts uses to interact with the Avernum world. Calls change things in the world, get information from the world, or both at the same time. All calls are described in the Appendices. There are hundreds of them.

Here is example of a call: erase_char. This call is used to make a creature disappear. Supposed creature 27 is a goblin and you want to make it disappear (how creatures are numbered is described later). If a line in your script is

```
erase_char(27);
```

then poor goblin creature 27 would disappear.

Another example is print_str, which prints a bit of text in the game's text area. You give print_str a string, and it prints it. If your script had this line:

```
print_str("Eeek! A monster!");
```

then the text "Eeek! A monster!" would appear in the text area.

Calls almost always need to have information given to them, so they know what to do. For example, erase_char needs you to tell it the number of the creature to erase. We say that we are "passing" information to the call when we give it numbers or strings to do what it needs to do. So above, we pass to print_str the string "Eeek! A monster!".

When you pass an integer to a call, it can be a full mathematical expression.

When you use a call, this is the format: first, the name of the call. Then a left parentheses ("("). Then all of the integers and strings you need to pass to the call, separated by commas. Then a right parentheses (")"). You need these things every time you use a call. Here are some examples of calls:

```
end();
print_big_str("You take ",x + 5,"points of damage");
```

```
play_sound(y); // y is the number of the sound
change_coins(-500);
```

Note that if nothing is passed to the call, you still need the parenthesis.

To figure out what information is passed to a call, read its description. For example,

print_big_str(string str,short num_to_print,string str2) - Like print_str, but prints a more customized piece of text. Prints, on one line, the text str, followed by the number num_to_print, followed by the text str2.

In parenthesis after the name of the call, we see we are passing a string, followed by an integer, followed by a string. So one way to use this call would be:

```
print_big_str("You get hit for",900,"points of damage."); // right
```

But not

```
print_big_str("You get hit for",900); // wrong
print_big_str("You get hit for","900","points of damage."); // no
```

Procedures and Functions

There are two sorts of calls: calls that return a value and calls that don't. If you look at the list of calls in the appendices, some begin with the word void. What this means is that the call does not return a value. It is just a command. You use it on a line by itself, followed by a semicolon. A call like this is called a procedure.

If a call description begins with "short", the call is a request for information from the game, and returns an integer value, which can be used in a mathematical expression. A call that returns a value is called a "function". You can also use a function on a line by itself followed by a semicolon, though it may not do anything.

For example, here is the description of the function coins_amount:

```
short coins_amount() - Returns the number of coins the party has.
```

Suppose the party has 83 coins. Here are some lines you could use:

```
x = coins_amount(); // the value of x is now 83
y = coins_amount() * 2; // the value of y is now 166
coins_amount(); // this line does nothing at all. The value 83 comes back, but nothing is
here to accept it, so no effect.
change_coins(-1 * coins_amount()); // takes away all the party's coins
```

Normally, a section of code in an Avernum script is just a set of calls, one after the other, separated by semicolons. For example, from the script basicnpc.txt:

```
set_target(ME,who_hit_me());
do_attack();
set_state(3);
```

However, you may not want to just have a list of commands. You may want to have some commands run in one situation, and a few other commands run at a different time. This is done using what is called flow controllers. Before they can be described however, we have to learn about Boolean Operators.

Boolean Operators and Boolean Statements

Sometimes in a script, you will want to use certain calls when one condition is true and different calls when another condition is true. For example, suppose a party is trying to walk through a tollgate that costs 10 coins. You would want the script to check if the party has 10 coins (using the convenient

coins_amount() function described above). If the party had 10 coins, they would be taken and the party would be allowed past. If the party didn't have the coins, there would be calls to give them a nasty message and block their progress.

The question "Does the party have 10" coins?" is a Boolean statement. It is a question with the possible answers true and false. In Avernumscript, you will use calls to ask Boolean questions, and use different calls depending on what the answer is. So first we have to learn introductory logic. What are Boolean values and Boolean statements?

A Boolean value is a value that is TRUE or FALSE. Boolean operators compare two numbers (examples > and <, i.e. greater than and less than). A Boolean statement is two numbers with a Boolean operator between them, and can be true or false.

So a Boolean statement is something that can be true or false.

Now for Boolean operators. "Operator" is a fancy programming word. You can think of a Boolean operator as a question you are asking. For example, consider ">", greater than. This little symbol represents a question: is the number before it greater than the number after it. If you see "10 > 5", you think "That is true." If you see "4 > 17", you think "That is false." You will use Boolean operators frequently in the game to ask questions about things like if the party's number of coins is at least 10.

Here are the Boolean operators. Remember, each has a number (or mathematical expression) before and after it, and the statement is true or false:

> (greater than) - If the number to the left is bigger than the number to the right, the statement is true. Otherwise, false. Examples:

```
3 > 2 // true
5 > 5 // false
-15 > 20 // false
x > 5 // true if the value of the variable is greater than 5
```

< (less than) - If the number to the left is smaller than the number to the right, the statement is true. Otherwise, false.

>= (greater than or equals to) - If the number to the left is the same size or bigger than the number to the right, the statement is true. Otherwise, false. Examples:

```
3 >= 2 // true
5 >= 5 // true
-15 >= 20 // false
x >= 5 // true if the value of the variable is at least 5
```

<= (less than or equals to) - If the number to the left is the same size or smaller than the number to the right, the statement is true.

== (equals) - The statement is true if the numbers on each side are the same. False otherwise.

!= (not equal) - The statement is true if the numbers on each side are not the same. False otherwise.

You can use functions (calls that return an integer) in Boolean statements. Recall the call coins_amount() returns the number of coins the party has. Suppose the party has 50 coins. Here are some examples of Boolean statements and whether they are true or false:

```
coins_amount() > 40 // true
coins_amount() <= 50 // true
coins_amount() < 50 // false
coins_amount() == 50 // true
coins_amount() != 45 // true
coins_amount() !=50 // false
coins_amount() * 2 == 100 // true
```

You can also use variables in Boolean statements. Suppose x is an integer variable.

```
x = 15; // now x is 15
x < 20 // true
x != 8 // true
```

```
x == 16 // false
```

There are also Boolean operators that can look at two Boolean statements and evaluate whether both or true or whether at least one is true. These are called the AND and OR operators:

&& - AND operator. It has two Boolean statements to the left and right of it, in parenthesis. If both of these statements are true, the whole statement is true. Otherwise, the whole statement is false. Example:

```
(1 > 0) && (2 == 2) // true ... both the statement on the left and on the right are true, so the whole thing is true
(0 > 1) && (3 > 2) // false ... the statement on the left is false, so the whole thing is false
(0 > 1) && (2 > 3) // false ... both statements are false, so the whole thing is false
```

|| - OR operator. It has two Boolean statements to the left and right of it, in parenthesis. If at least one of these statements is true, the whole statement is true. Otherwise, the whole statement is false. Example:

```
(1 > 0) && (2 == 2) // true ... both the statement on the left and on the right are true, so the whole thing is true
(0 > 1) && (3 > 2) // true ... the statement on the right is true, so the whole thing is true
(0 > 1) && (2 > 3) // false ... both statements are false, so the whole thing is false
```

You can also use functions in statements like these. Suppose, again, the party has 50 coins. And suppose x is a integer variable with value 25.

```
(x == 25) && (coins_amount() < 40) // false ... the statement on the right is false
(x == 25) || (coins_amount() < 40) // true ... the statement on the left is false
```

Finally, you can use OR and AND to evaluate a large number of Boolean statements at the same time. You can do this by placing statements one after the other, or by nesting statements in pairs of parenthesis. If statements are in parenthesis, they are evaluated first. Suppose, again, the party has 50 coins, x is a integer variable with value 25, and y is an integer of value 0.

```
(coins_amount() == 50) && (x == 25) && (y == 0) // true ... all are true
(coins_amount() == 50) && (x == 25) && (y == 1) // false ... one of the 3 statements is false
(coins_amount() == 50) && (x < 20) && (y == 1) // false ... two of the 3 statements are false
(coins_amount() == 50) || (x < 20) || (y == 1) // true ... one of the 3 statements is true
and, for OR statements, only one has to be true for the whole thing to be true
(x == 25) || (y == 1) // true. now what if we put this in ( ) and put it in the statement below ...
(coins_amount() == 50) && ((x == 25) || (y == 1)) // true, as the statement on the left is true and the statement "(x == 25) || (y == 1)" on the right in parenthesis is true
(coins_amount() == 50) && ((x == 15) || (y == 1)) // false, because the statement "(x == 15) || (y == 1)" on the right is false
```

Why are true and false things important? Because sometimes in a script, you will want to use certain calls when one condition is true and different calls when another condition is true. For example, what we described at the beginning of this section: do one thing if the party has 10 coins, and a different thing if they don't. Now that we understand how to phrase our questions, we can understand Flow Controllers.

Flow Controllers

Flow Controller is a funny sounding term for something very simple. Flow Controllers control the flow of Blades of Avernum through your script. They can send the game to use a certain set of calls instead of a different set.

If Statements

The simplest controller is the If statement. It is used like this:

```
if ([Boolean statement])
       [some call]
```

In this case, if the Boolean statement is true, the call is used. Otherwise, it isn't. For example, consider this call:

void change_coins(short coin_amount) - Changes the number of coins the party has by coin_amount (which can be negative).

Now look at this If statement:

```
if (coins_amount() >= 10)
       change_coins(-5);
```

When these calls are reached, if the party has at least 10 coins, they lose 5. Otherwise, nothing happens. If the party has 10 coins, the flow of the script goes through change_coins. Otherwise, it flows past it.

The condition for an If statement must always be surrounded by parenthesis. So while the above code is correct, this is wrong:

```
if coins_amount() >= 10 // No!
       change_coins(-5);
```

Sometimes you will want the condition of an If statement to result in several calls being used. To do this, after the conditional for the If statement, put the calls in brackets ("{" and "}"). For example:

```
if (coins_amount() >= 10) {
       change_coins(-5);
       print_str("You lost 5 coins! Ha ha!"); // prints message in the text area
       }
```

For this code, if the party has at least 10 coins, they lose 5 and a message is printed. Otherwise, nothing happens.

Note that you only put a semicolon after calls, not after brackets.

Also, you can use much more complicated Boolean statements with If statements, using AND and OR operators. For example, suppose you have a variable x and, for whatever reason, you only want to take the party's 5 coins if the have 10 coins and x is 0. Then you would write this:

```
if ((coins_amount() >= 10) && (x == 0))
       change_coins(-5);
```

You can make these Boolean statements as complicated as you want. Now suppose you only wanted to take the 5 coins if the party had 10 coins and x was 0 or 3 or 8. Then you would write this:

```
if ((coins_amount() >= 10) && ((x == 0) || (x == 3) || (x == 8)))
       change_coins(-5);
```

If-Else Statements

Now suppose that if a condition is true you want some calls to be used, but if it isn't true, different calls were to be used. Well, you could use two If statements, like this:

```
// Take 5 coins if the party has 10 or more, mock them if they don't.
if (coins_amount() >= 10)
       change_coins(-5);
if (coins_amount() < 10)
       print_str("You are poor.");
```

But you can also use an Else statement. This is formatted like this:

```
if ([Boolean statement])
       [One call, or several calls inside brackets]
       else
       [A different call, or several calls inside brackets]
```

Then, if the Boolean statement is true, the first set of calls is used. If false, the second set of calls is used. As before, if you want more than one call in either set, enclose them in brackets. Examples:

```
// Take 5 coins if the party has 10 or more, mock them if they don't.
if (coins_amount() >= 10)
       change_coins(-5);
       else print_str("You are poor.");
```

or

```
if (coins_amount() >= 10) {
       change_coins(-5);
       print_str("You lost 5 coins! Ha ha!");
       }
       else {
              print_str("You are poor.");
              } // note you can choose to put a single call inside brackets
```

While Statements

The final sort of Flow Controller is the very powerful While statement. A While statement uses a series of calls again and again until a given Boolean statement becomes true. A While statement is formatted like this:

```
while ([Boolean statement])
       [one call, or several calls surrounded by brackets]
```

If the Boolean statement is true, then the game keeps using the calls until the statement is not true. For example, suppose i is an integer variable. Consider this code:

```
i = 0;
while (i < 10) {
       print_str("Counting");
       i = i + 1;
       }
```

This code would print the word "Counting" in the text area 10 times. Or consider this:

```
i = 0;
while (i < 10) {
       print_str("Counting");
       change_coins(-8);
       i = i + 1;
       }
```

This code would print the word "Counting" in the text area and take 8 coins away from the party 10 times.
Note that, when writing a While statement, you have to make sure that the Boolean statement eventually becomes false. If it doesn't, the loop will run forever and Blades of Avernum will return an error. Consider this ugly code:

```
i = 0; // begin some ugly code
while (i != 10)
       print_str("Writing unending words!");
```

Note that we never do anything to the value of i to make it 10. This is really bad. This means that the script will write the words “Writing unending words!” forever, while the poor player watches, helpless. Eventually, Blades of Avernum will give an error and keep the player from being trapped forever, but this is still bad.

Avernumscript is a powerful tool, but it will not prevent you from hurting yourself. A fork is nice when eating, but you shouldn’t jam it into your eye. Similarly, Avernumscript can do wonderful things, but you are responsible for not writing code that makes everything blow up.

Instead, you should correct the above statement by changing it to:

```
i = 0; // begin some less ugly code
while (i != 10) {
	print_str(“Writing ending words!”);
	i = i + 1;
	}
```

Much better.

Finally, this is the end of the description of the syntax of Avernumscript. If you aren’t too lost, we can begin to write scripts. First, a bit of explanation is in order for how Blades of Avernum keeps track of the many things in the game it keeps track of.

How Spaces Are identified

Each outdoor section is 48 x 48 spaces. The spaces in each dimension are numbered from 0 to 47. The coordinates of the spaces are written {x position, y position}, where the x position is the number of spaces from the west edge of the area and the y position is the number of spaces from the north edge. So {0,0} is the upper left corner (northwest), {47,0} is the upper right corner (northeast) and {47,47} is the bottom right corner (southeast).

The coordinates for towns work exactly the same way, except that towns can be 32 x 32, 48 x 48, or 64 x 64. The size of the current town can be found in your script by using the function call current_town_size().

How Characters Are Identified

A huge number of the calls have to do with getting information about and affecting characters and creatures. To use them, you need to know how characters and creatures are numbered.

Each town can have at most 120 creatures active in it at any one time. They are referred to by numbers 0 to 119. Different ranges of numbers are used for different sorts of creatures:

0-3: The party. The player’s characters have numbers 0 to 3.
4,5: Added characters. You can use calls to have a character travel with the party. That character will be put in one of these two slots. You can only add two characters.
6-85: Placed town characters. You can place up to 80 characters in a town. When you select a character in a town, you will see the character’s number to the lower left. That is the number the character will have in the game. Those preset characters are the only characters that can ever be in these slots. If you place an ogre in slot 43, slot 43 will only ever contain that ogre.
86-119: Added creatures. These slots are for wandering monsters, summoned creatures, and creatures that split.

Stuff Done Flags

The single most important concept to master in scenario design is the Stuff Done Flag. The Stuff Done Flags are numbers the game keeps track of, which are used by the game to remember what the party has done in the scenario so far.

For example, suppose you have a treasure hidden somewhere, like, say, a shield. You only want the party to be able to get that shield once, so the game needs a way to remember that the party has gotten it.

This is done with a Stuff Done Flag. It is a number that starts at 0, and is matched up with getting the shield. When the shield is reached, the game sees if the Stuff Done Flag is 0. If it is, the shield hasn't been taken yet, so the game gives the party the shield, and the Stuff Done Flag is set to 1. The Stuff Done Flag being 1 tells the game in the future that the shield has been taken.

A Stuff Done Flag is often referred to as an SDF for short.

Stuff Done Flags - the Specifics

But what are the Stuff Done Flags? Picture a grid of numbers, 300 wide and 30 high. Each of these numbers starts out at 0. These 9000 (300 x 30) numbers are your Stuff Done Flags, and they are all set to 0 when the party starts a scenario (they are written down in the save file, which is how the game remembers what you've done already when the save file is opened).

Stuff Done Flag are described by coordinates. Much as the coordinates of a spot of terrain are given by an X and Y value, a SDF has coordinates too. The first coordinate of a Stuff Done Flag is the column it is in (out of 300 columns, a number from 0 to 299), and the second coordinate of a Stuff Done Flag is the row it is in (out of 30 rows, a number from 0 to 29). For example, the taking of a shield may be attached to Stuff Done Flag X = 112, Y = 3, also written (112,3). These instructions often refer to the two coordinates of a Stuff Done Flag (the X coordinate is the first part and the Y coordinate is the second part). In the example, 112 is the first part, and 3 is the second part.

All Stuff Done Flags start as 0 when the scenario is started. For every event or thing that must be remembered, you will need to assign a Stuff Done Flag to it. What SDF goes with which event is chosen by you. When the party completes some mission, you may decide that (91,9) will become 1, and when some demon is killed, (94,2) will become 1. Later, you can have special encounters check a Stuff Done Flag, and do different things depending on what the value of the Stuff Done Flag is. For this reason, you will want to keep careful notes on your scenario, to make sure that no two events are linked to the same Stuff Done Flag.

Keeping Notes

One thing all of the Avernum games had in common is that, when they were done, dozens of pages of careful, intricate notes had been taken about them. For example, in the first Blades of Avernum scenario, Valley of Dying Things, when you take the opening stone in Avizo's shop, the Stuff Done Flag (1,2) is set to 250. Later, when the player tries to take the stone, the program will see that the Stuff Done Flag (1,2) is not 0, and will know that stone has been taken and should not be given again.

In the same scenario, when you kill the evil spirit in Blinlock, Stuff Done Flag (2,3) is set to 1. Suppose, however, that the designer has made a mistake and made the Stuff Done Flag (1,2) set to 1 when the spirit was slain. In this case, if the party killed the spirit first (causing (1,2) to be set to 1), and then went to get the opening stone, it wouldn't be there! (because (1,2) has been set to 1, making the game think that the item has been taken, even if it hasn't) Then the party wouldn't be able to finish the game. This is bad. However, careful notes reminded the designer that the Stuff Done Flag (1,2) had already been used, and thus it wasn't used in Blinlock.

One useful tip for making sure Stuff Done Flags don't get reused: in town 0, only use Stuff Done Flags with first coordinate 0 ((0,0), (0,1), (0,2), etc.). It town 1, only use Stuff Done Flags with first part 1, and so on. For outdoor sections, use the Stuff Done Flags with first part 200, 201, 202, and so on. So outdoor section 0 (the upper left section) gets flags (200,0), (200,1), (200,2), etc. This way, you can be sure when designing town 8, you won't use a Stuff Done Flag already needed for town 4.

Of course, in some towns you might need to use more than 30 Stuff Done Flags. When this happens, you can consult your notes to find Stuff Done Flags that aren't being used for anything yet.

What Is a State?

Before writing any scripts, it's important to understand roughly how they will be organized.

Scripts with code that affects the game (scenario, town, outdoor, creature, terrain scripts) are split up into little chunks of code called "states". You can direct the game to go to a chunk of code in a state and do what it says. When this happens, we say that you are "calling" the state.

So this is how the terminology works: if you want something to happen in the game, you make a state and write some code in it. Then you do what it takes to call that state.

States are numbers. You can have state 1, state 2, state 50, and so on.

For example, consider special encounters in towns and outdoor sections. The code for special encounters is split up into chunks, each in its own state. Each special encounter calls code in its own state.

In addition, you can have states call other states. Your special encounter may call state 47. Then, at the end of state 47, you can put a line of code calling state 81, which in turn calls state 13. So the code in states 47, 81, and 13 would be called in order.

For scenario, town, and outdoor scripts, you will just write a list of states, which will be called under certain circumstances (when the town is entered, when the party steps on a special encounter, etc.). States work a little differently for creatures and terrain scripts, but that is gone into in more detail in their respective chapters.

Drawing Dialog Boxes

When writing special encounters, you will frequently want to display a dialog box which describes a situation and asks the party what they want to do. This is slightly complicated, but very powerful and versatile. The way you do this is you use calls to build your dialog box, one line at a time.

All of the calls for building dialog boxes are listed in the section of appendices listing all the calls, in the section Basic Dialog Box Calls. This little section just gives the basics.

To tell Blades of Avernum that you are about to build a dialog box, use the line

```
reset_dialog();
```

Next, you can set up to six chunks of text in the box. Use the call

```
add_dialog_str([which of the 6 strings you are editing, numbered 0 to 5],[the string],[the
number of pixels the string is indented, probably 0);
```

For example:

```
add_dialog_str(0,"This is the first line!",0);
add_dialog_str(1,"This is the second line, indented a little!",20);
```

Next, you can set up to three choices for the player. This defaults to one choice that says "OK." The call to set choices is

```
add_dialog_choice([which choice, numbered 0 to 2],[the text of the choice]);
```

For example,

```
add_dialog_choice(0,"The first choice.");
add_dialog_choice(1,"The second choice.");
```

Finally, use a call to display the dialog and get the response.

```
The_choice = run_dialog(1);
```

The_choice is an integer variable. The 1 means that the dialog box will have a Record button.

The run_dialog function records which of the choices that were selected, either 1, 2, or 3. This is the sole exception to everything being numbered starting with 0. If the first choice is selected, 1 is returned. The second choice returns 2. The third choice returns 3.

Chapter 2.9: Creating Scenario Scripts

Every scenario can have one scenario script. It's file name is the same as the scenario's file name, but ending in .txt instead of .bas (so the scenario script for "valleydy.bas" has to be called "valleydy.txt").

The scenario script generally contains two things. First, there are states initializing things in the scenario (like the names and descriptions of the quests, and what is in the shops). Second, there are states that can be accessed from anywhere in the scenario (for example, the gate opening effect when someone used the Opening Stone in Valley of Dying Things).

How the Scenario Script Is Formatted

A scenario script has to begin with the line

```
beginscenarioscript;
```

Then you declare any string or integer variables. First, you write the line:

```
variables;
```

and then you declare the variables (if you have any). For example:

```
short some_number, some other_number = 5;
string some_text = "a few words";
```

Then you indicate the beginning of the actual script with this line:

```
body;
```

Then, finally, there are the states (states are defined at the end of the previous chapter). Each state is formatted like this:

```
beginstate [some number]; // this begins the state
      [your code here]
break; // this says that the state is done
```

As an example, here is a very simple state from Valley of Dying Things, which is called when a ruined book is used:

```
beginstate 44; // start
      message_dialog("You have no idea what this old book was about. Mold and rot have
consumed it. It is useless now.",""); // display text
break; // done
```

Each state always begins with "beginstate" and must always end with "break;". States are numbered. Each state must be assigned a number, from 0 up. However, some states are reserved for special uses, and you need to know what they are.

The Loading Scenario State

State 110 is automatically called by the game whenever the scenario is loaded from memory. When the party begins this scenario or you load a save file in this scenario, this state is called.

You can also write LOAD_SCEN_STATE as the number of this state. For example:

```
beginstate LOAD_SCEN_STATE;
      init_special_item(0,"Runed Stone","This is a black, smooth stone, about the size of
your fist. It is covered with small, finely etched runes and has been worn smooth by repeated
handling. It is slightly warm.");
      init_quest(0,"Deal with Skylark Vale Curse","Mayor Crouch in Sweetgrove has asked you
to investigate the curse that has afflicted the entire Vale.");
```

```
break;
```

What goes into the state 110? Anything you need to do to initialize the scenario, which needs to be done every time it is played. Most such calls are listed in the Scenario Initialization Calls section of the calls descriptions in the appendices. The most important ones are init_special_item and init_quest. Special item and quest descriptions need to be reset every time you enter a scenario.

The Starting Scenario State

State 111 is automatically called by the game whenever the scenario is begun. When the party begins this scenario, this state is called. It is only ever called once in the scenario, at the very beginning.

You can also write START_SCEN_STATE as the number of this state. For example:

```
beginstate START_SCEN_STATE;
	add_item_to_shop(0,4,500);
	create_boat(29,1,17,54,1);
	create_horse(0,1,6,24,1);
break;
```

What goes into the state 0? Anything you need to do to initialize the scenario, which you only ever want called once. For example, calls that create boats and horses, and calls that put items in the shop.

The Start State

State 2 is automatically called by the game every turn the party is in the scenario. It is called outdoors, in town, and in combat.

You can also write START_STATE as the number of this state. For example:

```
beginstate START_STATE;
	print_str(“A turn has passed.”);
break;
```

This example would make text appear in the text area every turn. This would, of course, be irritating.

The Rest of the States

Finally, there are states that do special things that you want. For example, you can have items call scenario script states when used. You can have floors and terrains that call scenario script states when stepped on. And you can call scenario script states from any other script.

These other states should be numbered from 10 to 100. For example, Valley of Dying Things contains a scepter that heals the entire party when used. This item is set to call state 18 when used. Here is the script:

```
beginstate 18;
	message_dialog("You wave the scepter, and a warm light emerges from its tip. As the
light washes over you, you feel a pleasant, tingling sensation. The rod cleanses you of
unpleasant afflictions.","The rod doesn't feel warm anymore. It probably will take some time
for it to regain its energy.");
	i = 0;
	play_sound(60); // play healing sound
	while (i < 6) { // loop through the 6 party members
		if (char_ok(i)) { // heal character if alive
			set_char_status(i,0,-10,1,0); // heal 4 nasty effects
			set_char_status(i,7,-10,1,0);
			set_char_status(i,12,-10,1,0);
			set_char_status(i,13,-10,1,0);
			}
		i = i + 1;
		}
```

```
break;
```

This item can be used anywhere (town, outdoors, combat), so the code for it needs to be reachable everywhere. Code that has to be accessible anywhere in the game needs to be in the scenario script.

Chapter 2.10: Creating Town Scripts

Every town can have one town script. It's file name can be up to 13 characters long, followed by .txt. You set a town's script in the Town Details window.

The town script generally contains three things. First, there is the state called when the party enters the town. Second, there are states called when the party leaves the town. Finally, there are states called when the party triggers a special encounter.

How the Town Script Is Formatted

A town script has to begin with the line

```
begintownscript;
```

Then you declare any string or integer variables. First, you write the line:

```
variables;
```

and then you declare the variables (if you have any).

Then you indicate the beginning of the actual script with this line:

```
body;
```

Then, finally, there are the states. Each state is formatted like this:

```
beginstate [some number]; // this begins the state
	[your code here]
break; // this says that the state is done
```

Each state always begins with "beginstate" and must always end with "break;". States are numbered. Each state must be assigned a number, from 0 up. However, some states are reserved for special uses, and you need to know what they are.

The Enter Town State

State 0 is automatically called by the game whenever the town is entered.

You can also write INIT_STATE as the number of this state. For example:

```
beginstate INIT_STATE;
	set_name(18,"Forbes");
	set_level(18,30);
break;
```

What goes into the state 0? Anything you need to do to initialize the town, which needs to be done every time it is entered.

The Leave Town State

State 1 is automatically called by the game whenever the town is exited (either by walking out or being taken out by a special encounter).

You can also write EXIT_STATE as the number of this state. For example:

```
beginstate EXIT_STATE;
	message_dialog("You flee the dungeon, screaming.","");
break;
```

The Start State

State 2 is automatically called by the game every turn the party is in the town. You can also write START_STATE as the number of this state.

The Rest of the States

The rest of the states contain bits of code to be called by a variety of circumstances.

Most commonly, you can use the Create Special Encounter button, select a rectangle, and give the number of a state. When the party enters that rectangle, that state is immediately called. One example of this is a state that prints a description of the entered area, such as:

```
beginstate 30;
	if (get_flag(0,21) > 0)
		end();
	message_dialog("You notice something unusual about the rear corner of this building.
There is a rectangular crack in the wall, about the size and shape of a small door.","");
	set_flag(0,21,1);
break;
```

This state prints the text description once, and then never again.

You can use the Town Details window to set special states that are called when the town is left in different directions (north, west, south, east).

Crimes

In towns, you don't want the party to be able to slaughter and rob indiscriminately. That is what crime levels are for.

The game remembers a crime level for the party for every town. It starts at 0. Each time they attack a friendly person, it goes up by 5. Each time they steal an item they don't own and a friendly character sees them, it goes up by 1.

Normally, the crime level doesn't do anything. If you use the set_crime_tolerance call, you can specify the crime level at which the whole town goes hostile. For example, if you use the call set_crime_tolerance (3), the town goes hostile permanently when the party attacks one friendly person or steals 3 items.

The Town Status

The game keeps track of a town status number for each town. Its values are:

```
0 - never been entered
1 - party has entered
2 - party has entered, town hates
3 - town has been destroyed
```

The call set_town_status can be used to manually change the settings for a town, and the town_status call can return the status of a given town.

Reloading the Town Script

When you load a saved game in a town, the script is reloaded. So, if you make a change in the script and save it while playing the town, saving and loading the saved game will reload the script. However, to be entirely safe when testing, you should probably reenter the town from the outside after changing the script.

Chapter 2.11: Creating Outdoor Scripts

Every outdoor section can have one outdoor script. It's file name can be up to 13 characters long, followed by .txt. You set an outdoor section's script in the Outdoor Details window.

Outdoor Scripts are much more limited than scenario and town scripts. You can't have a state called automatically when an outdoor section is loaded or exited.

Outdoor script states are mainly used for special encounters you place or called by groups of creatures (when they are met, defeated, or fled).

How the Outdoor Script Is Formatted

An outdoor section script has to begin with the line

```
beginoutdoorscript;
```

Then you declare any string or integer variables. First, you write the line:

```
variables;
```

and then you declare the variables (if you have any).

Then you indicate the beginning of the actual script with this line:

```
body;
```

Then, finally, there are the states. Each state is formatted like this:

```
beginstate [some number]; // this begins the state
      [your code here]
break; // this says that the state is done
```

Each state always begins with "beginstate" and must always end with "break;". States are numbered. Each state must be assigned a number, from 0 up. However, some states are reserved for special uses, and you need to know what they are.

The States

Most commonly, you can use the Create Special Encounter button, select a rectangle, and give the number of a state. When the party enters that rectangle, that state is immediately called.

In addition, when editing an outdoor encounter, you can set states to be called when the encounter is met, defeated, or fled. The section Outdoor Calls in the descriptions of the calls in the appendices gives some useful calls to use in this situation. outdoor_enc_result is especially useful.

Loading the Outdoor Script

The script for an outdoor section is loaded when the party gets within 48 spaces of the edge of the section. It will be reloaded when you load a saved game in which the party was near that section.

If you change the script for a section or assign a script to a section that didn't have one, you will need to move over 48 spaces from the edge of the outdoor section and then move back to get the script to load.

Also, importantly, when you load a saved game, the terrain for the outdoor sections is reset to what it was. If you made any changes to the terrain (with set_terrain), they will be erased. Bear this in mind when designing encounters.

Chapter 2.12: Creating Creature Scripts

This is roughly how the previous scripts worked: They were a collection of states. When something special happens, a state is called. A chunk of code is run by the game. And then the script is done and play continues.

Creature scripts are different.

Creature states contain calls that direct the creature around. These calls make it move around, choose targets, attack, and do all sorts of other things.

Creature scripts are what is called, in game design terminology, a Finite State Machine. What does that mean? Well, creature scripts are divided into states like the other scripts are. However, one of the states is the state the creature's brain is in. It keeps running that state again and again, every time it acts, until you use a call to switch the state it is in. Then it runs that state again and again.

For example, most creatures have two states: a peaceful state and an attacking state. The peaceful state contains the code that makes the creature wander around and look for targets. When it has a target, it switches to the hostile state. Then it runs the hostile state, with all the code for fighting, until it has no target. Then it switches back to the peaceful state.

The state that the creature's brain is currently in is called its current state. All creatures start with a current state of 2 (which can be written START_STATE).

To change a creature's state, use the set_state call.

Writing creature scripts can be very complicated. Most of the time, you will just want to use pre-written creature scripts and customize their behavior using Memory Cells.

How Often Does a Creature Think?

How does a creature use its script (i.e. think)? Normally, at first, every 8 turns, when the party is close, the creature will call its current state. The code there will generally make the creature walk around, look for targets, etc. When a creature has been alerted (either by seeing a hostile target or by the call alert_char), its current state will be called every turn, whether the party is close or not.

You can set a creature to run its script more often and at a distance from the party. You can change this using the Act At Distance field in the editing creature window. You can also use the set_char_script_mode call to make a creature act more often.

Characters that have been alerted call their current states every turn.

When a wandering creature is created, it starts with the default script for its creature type (usually basicnpc) and starts out alerted.

How the Creature Script Is Formatted

A creature script has to begin with the line

```
begincreaturescript;
```

Then you declare any string or integer variables. First, you write the line:

```
variables;
```

and then you declare the variables (if you have any).

Then you indicate the beginning of the actual script with this line:

```
body;
```

Then, finally, there are the states. Each state is formatted like this:

```
beginstate [some number]; // this begins the state
      [your code here]
break; // this says that the state is done
```

Each state always begins with “beginstate” and must always end with “break;”. States are numbered. Each state must be assigned a number, from 0 up. However, some states are reserved for special uses, and you need to know what they are.

Making the Creature Move, and Pathfinding

You can make characters move around by using calls like approach_char, fidget, return_to_start, and approach_waypoint. Each of these calls has a character start walking towards a given location. When you tell a character to move, it will keep moving closer to its destination each tick until it gets there, where it stops.

Characters remember their destinations. Frequently, movement calls will not do anything if the character currently has a destination. The stop_moving call erases a character’s current destination, enabling you to give it a new destination.

When you tell a creature to go somewhere and walls or other creatures are in the way, Blades of Avernum is smart enough to move your creature on a path around them. This is called pathfinding.

Blades of Avernum has a sophisticated pathfinding technique, which enables it to quickly and smoothly direct creatures through towns and dungeons. However, it has several limitations you should know about.

First, it can’t move characters through doors. If you want a character to move somewhere over a distance, you need to either not place any closed doors in its way or have a script change the closed doors in the way to open doors.

Second, it can be slow. The longer the distance the game is trying to make a path along, the longer it will take. Trying to make a path when no possible path exists (because the route is blocked by creatures or terrain) is slowest of all. You should try to avoid having large numbers of characters move long distances at the same time.

Third, it can only make paths about 40 or 50 steps long. If you want a character to move a long way, you may want to have it first move to an intermediate point (say, a waypoint) and then move the rest of the way.

Making the Creature Attack

The call do_attack has the creature attack a foe. It uses Blades of Avernum’s AI to choose the best action for its current situation. If the character has Mage or Priest spell skill, it may cast the most helpful spell that level of skill allows (an NPC starts out knowing all spells its skill enables it to cast). It may pick up a nearby useful item or use an item in its pack. And, of course, it may attack.

Blades of Avernum handles most of a character’s combat strategy, but you can use different calls to affect a creature’s action. The call do_attack_tactic enables you to give a creature a strong preference for a certain sort of attack. The set_strategy call enables you to dramatically affect how a creature acts. Same for set_aggression and set_courage.

If you do not want a character to ever cast a certain spell, you can use the change_spell_level call to change that character’s skill with that spell to 0. A character won’t cast a spell it doesn’t know.

Creature Health/Energy and Regeneration

A creature’s default starting health is roughly half its endurance times its level. It’s starting spell energy is 3 times the sum of its level, Mage Spells, and Priest spells skills.

Each turn, non-player characters regenerate some health and spell energy. The amount of health regained each turn is the maximum health divided by 25. The energy regained is the maximum energy divided by 25.

If you want to manually adjust the amount regained, you can use the **change_char_health and change_char_energy** calls in the creature’s script.

The Initialization State

State 0 is automatically called by the game whenever the creature is spawned (usually when the town is entered, but not necessarily).

You can also write INIT_STATE as the number of this state. For example:

```
beginstate INIT_STATE;
	set_name(my_number(),"Forbes");
	set_level(my_number(),30);
break;
```

The Death State

State 1 is automatically called by the game whenever the creature is killed in combat. This state is NOT called when the character disappears because the party leaves town or uses the erase_char call.

You can also write DEAD_STATE as the number of this state. For example:

```
beginstate DEAD_STATE;
	print_str("The giant bunny falls down, dead.");
break;
```

The death state of the script is called before the creature's death animation is played, experience is awarded, and loot is dropped. If you use erase_char to delete the character in this state, the character just disappears without any of the death effects.

The Start State

State 2 is the starting current state for every creature. This state is called again and again until use of the set_state call changes the current state. Then the new state is used again and again.

This state may have the character walk around and should probably have it watch for hostile targets.

Your Custom States

You can write states numbered from 3 to 100 for all of the possible other states the creature's brain can be in. The most common (and, most of the time, the only one you will use) is a state for what the creature is in combat. Here is the basicnpc combat state.

```
beginstate 3; // attacking
	if (target_ok() == FALSE)
		set_state(START_STATE);
	do_attack();
break;
```

If the creature doesn't have a target, it returns to its starting behavior. Otherwise, do_attack tells the game to have it attack in the best way.

The Talking State

You probably want something special to happen when the party talks to a character. The party can only talk to non-hostile characters. When the party talks to such a character, the game calls state 110 in the creature script (can be written TALKING_STATE).

For many creatures, you will want to start a conversation when this happens. This is done using the call begin_talk_mode. For example:

```
beginstate TALKING_STATE;
	begin_talk_mode(12); // begins a conversation at talking mode 12
break;
```

For much more on talking, read the chapter on Chapter 2.14: Creating Dialogue.

Messages (Advanced)

Sometimes, you will want creature scripts to be able to communicate with each other. You can send a message to a creature script, and the script can watch for incoming messages and act upon them.

A message is a number (which can be anything 0 and above). You can give a creature a message by using calls, such as give_char_message. A script can get the number of a message it has received by using the call my_current_message. When a creature script has been run, its current message is set back to –1. If you do not process the message when the script is next run, it is lost.

A creature can only remember one message at once. If, in the same round, it gets as a message the number 4, and then it gets as a message the number 6, the next time the script is run, my_current_message will return 6.

When a creature has a message, its script is run at the end of the current round, no matter what (even if it is far away from the party).

How might message be used? Well, for example, suppose that when a certain boss is killed, every NPC within 8 spaces dies. Then you would want to give each of those NPCs a script which watches every round for message 50 (or whatever number you pick). And, when the boss dies, you would use the broadcast_char_message call to give message 50 to every creature within 8 spaces.

Party members and NPCs who have joined the party can not get messages.

Chapter 2.13: Creating Terrain Scripts

Terrain scripts are scripts that live on one space in a town. Terrain scripts are used for terrain that acts in an interesting way (doors, levers that open and close gates, mines, traps).

They run every 8 turns and only when the party is close. This behavior can be changed using the set_script_mode call. Terrain scripts can also be triggered and instantly run when certain special circumstances happen (the space is walked into, the party tries to walk in this space and is blocked, an unlock spell is cast nearby, etc.).

Terrain Scripts are very similar to creature scripts. Each has a current state, which starts at 2 and can be set to a different state using set_state.

You can't create or delete terrain scripts after a town is entered. All terrain scripts that have been set will be there and initialized when the town is entered.

Remember that a terrain script sits on one space in the town. You can put any terrain script on any space. You do not have to put a "door.txt" script on the same place as a door. However, it won't work right if you put "door.txt" on the same space as, say a pillar.

Writing terrain scripts can be very complicated. Most of the time, you will just want to use pre-written terrain scripts and customize their behavior using Memory Cells.

How the Terrain Script Is Formatted

A creature script has to begin with the line

```
beginterrainscript;
```

Then you declare any string or integer variables. First, you write the line:

```
variables;
```

and then you declare the variables (if you have any).

Then you indicate the beginning of the actual script with this line:

```
body;
```

Then, finally, there are the states. Each state is formatted like this:

```
beginstate [some number]; // this begins the state
      [your code here]
break; // this says that the state is done
```

Each state always begins with "beginstate" and must always end with "break;". States are numbered. Each state must be assigned a number, from 0 up. However, some states are reserved for special uses, and you need to know what they are.

The Initialization State

State 0 is automatically called by the game when the terrain spot is initialized (always when the town is entered). State 0 can, as before, be written INIT_STATE.

The Start State

State 2 is the starting current state for every terrain script. This state is called again and again until use of the set_state call changes the current state. Then the new state is used again and again.

For inert terrain types that wait for a party to walk into it (doors, traps, etc.), you probably won't do anything in the start state.

Your Custom States

You can write states numbered from 3 to 99 for all of the possible other states the terrain type can be in. You probably won't do this very often.

Triggered States

Most of the action for terrain scripts takes place in the triggered states. These are states that are instantly called when something happens in or around the terrain spot. You will probably only want to use a few of these (if any) for any given terrain script. They are:

beginstate BLOCK_MOVE_STATE; (state 112) - This state is called when the terrain in the space the terrain script is sitting on blocks the party's progress.

The best example of this is doors. A door is essentially a wall. When the party walks into the door, it blocks their progress, the script notices it, and then the script opens the door for them.

beginstate STEP_INTO_SPOT_STATE; (state 114) - This state is called when a member of the party tries to walk into the space containing the terrain script. The call block_entry can be used to keep the character from entering.

The best example of this is traps. A trap activates when you try to enter its space. If the player refuses to try to disarm the trap, block_entry is used to keep him or her out.

beginstate SEARCH_STATE; (state 100) - This state is called when the party searches the location containing the script (by standing next to it and looking at it). For a terrain type to be searchable, its te_can_look_at characteristic needs to be set to 1.

Normally, if a terrain spot is a container, the party can get the items inside. If this state in a script is called and the block_entry call is used, you can keep the party from getting the items inside.

beginstate UNLOCK_SPELL_STATE; (state 101) - This state is called when an unlock spell is cast nearby. The get_unlock_spell_strength() call will return the strength of the spell.

beginstate SANCTIFICATION_STATE; (state 102) - This state is called when a Ritual of Sanctification is used next to the terrain spot.

Messages (Advanced)

Like creature scripts, terrain scripts can get messages. To learn the basics about messages, read the previous Section. You give a terrain script a message by using calls like broadcast_terrain_message.

Every turn, after the game runs the terrain scripts, it goes through again and runs every terrain script that has received a message. Thus, if a terrain script has a message, it can (and must) be processed immediately. Every time a terrain script is run, its message is set to –1.

Chapter 2.14: Creating Dialogue

No good scenario is complete without people to talk to. Dialogue advances the plot, gives quests, and lends pleasing context to the carnage.

The bad news is that dialogue scripts are formatted completely differently than the other sorts of scripts covered. (More to learn! Ack!) But the bright side is that dialogue scripts are fairly simple. Even if you don't want to learn anything about programming, you can create simple dialogue scripts and give people someone to talk to.

Naming Dialogue Scripts

To make the name for your dialogue script, take the name of the script for that town and add "dlg.txt" to it. For example, if the town's script is "ratpit.txt", the dialogue script will be called "ratpitdlg.txt". For a town to have dialogue, it has to have a town script.

Outdoor sections can't have dialogue.

Dialogue Terminology

Dialogue scripts are split up into nodes. What is a node? A node is one question that can be answered by one character. So if you ask someone "Where is the pit?" and the character says "West.", that is one node.

A dialogue script is simply a long list of nodes.

Each node has a question and an answer to that question. When the player is in a conversation, he or she will see a list of available questions and can select one to see one answer. The question is: how does the game know what questions to make available? The answer is: by using dialogue states.

The most important concept to understand for dialogue (really, the key concept) is the dialogue state. (This is different from the word "state" as used in the previous chapters. Try to forget about that one for a few minutes.) In dialogue, the state is a number. That number reflects where exactly the player is in a conversation.

Every dialogue node is available for only one dialogue state. When the party is in conversation and is at dialogue state 5, then only dialogue nodes available at state 5 can be seen.

Each dialogue node has a dialogue state that makes it available. Every dialogue node, in turn, can change the dialogue state when it is selected. Here is a short sample conversation to show how this works:

```
begintalknode 1; // this is the first node in the conversation
       state = -1; // the dialogue state that makes this node available, left at -1 for
states that start conversations
       nextstate = 1; // what to change the dialogue state to when this node is selected
       question = "Sunny Jim"; // For the first node in a conversation, the question is the
name of the character
       text1 = "You meet Sunny Jim. _What do you want, friend?_";
// When the conversation is started, nextstate sets the state to 1.

begintalknode 2;
       state = 1; // this question is available when the dialogue state is 1
       nextstate = 2; // when this question is asked, changes dialogue state to 2
       question = "Is there a dungeon nearby?";
       text1 = "_Yes._"; // Remember that an underscore means quotation marks.

begintalknode 3;
       state = 2; // this question is available when the dialogue state is 2
       nextstate = -1; // -1 means this is the end of a conversation thread
       question = "Where is it?";
       text1 = "_West._";

// this is the beginning of a different conversation thread
begintalknode 4;
       state = 1; // this question is available when the dialogue state is 1
       nextstate = 3; // when this question asked, changes dialogue state to 3
       question = "What do you do for a living?";
```

```
	text1 = "_I farm._";

begintalknode 5;
	state = 3; // this question is available when the dialogue state is 3
	nextstate = -1; // -1 means this is the end of a conversation thread
	question = "Farm what?";
	text1 = "_Oats._";
```

There is a lot more than this you can do with conversation nodes, but the states are the structure that makes the whole thing work.

When in conversation, the player wanders a path from node to node. Each question moves the player another step down that path. A question sets the dialogue state to a different number and makes a new set of questions available.

There are several rules about conversations. First, when one conversation node it visited, it can't be seen again. Second, if the player runs out of options (i.e. available questions), the player is moved back a step along the path and the previous set of questions (minus the ones asked before) becomes visible again. Third, if even then, there are no more questions available, the player is given a choice: start the conversation over or end it. If the player chooses to restart the conversation, he or she is returned to the beginning and all dialogue nodes become available again.

These rules might seem unusual. Just play the game for a while, talking to people and seeing how the conversations flow, and it should make more sense.

How To Start a Conversation

When you place a creature with the basic default script (basicnpc), open the Edit Creature window and set memory cell 3 to be the first node of the conversation. Then the conversation will start when the party talks to the character.

When writing a creature script from scratch, begin_talk_mode, and give the number of the first node in the conversation. You can also call begin_talk_mode from other sorts of scripts.

How the Dialogue Script Is Formatted

A dialogue script has to begin with the line

```
begintalkscript;
```

Then you declare any string or integer variables. First, you write the line:

```
variables;
```

and then you declare the variables (if you have any).

Then you write all of the dialogue nodes, one after the other.

How a Dialogue Node Is Structured

Each dialogue node begins with the line

```
begintalknode [number];
```

Each dialogue script can have up to 200 nodes, numbered 0 to 199.

After that, each node has a set of characteristics. On each line, you write the name of the characteristic, than an equals sign, then the desired value/string, and finally a semicolon to indicate that you are done. Such as:

```
begintalknode 5;
	state = 3;
	nextstate = -1;
	question = "Farm what?";
	text1 = "_Oats._";
```

The characteristics are:

state: This node only becomes available if the current state is equal to this value. If state is left at -1, no other state can make it available by the party asking a question.

If this node is the first node in a conversation, leave this at -1.

nextstate: What the dialogue state become when this question is asked. if left at -1, this question leads to a dead end in the conversation. In this case, the game goes back a step in the conversation path to see if any other questions are available.

personality: You can set this characteristic equal to the personality who says this dialogue. This does not actually have any effect on the dialogue and you don't have to do it. However, it does help you keep track of what dialogue node belongs to what character.

condition (Advanced): This characteristic is more advanced but incredibly powerful. You can use it to hide conversation options from the player.

Suppose, for example, that this node is for a character named Bob who says where a magic sword is, but the party can only ask about it if Fred has told the party Bob knows where the sword it. You can set Stuff Done Flag (3,5) to be 1 if the party knows to ask Bob about the sword, and have this node/question only be available if that Stuff Done Flag is 1.

To use this characteristic, set condition equal to a Boolean statement. If the statement is true, the question is available. If no, it isn't. Set condition to equal 1 if you want the node to always be available. For the example above,

```
begintalknode 185;
	state = 3;
	nextstate = -1;
	condition = get_flag(3,5) == 1; // is SDF(3,5) one?
	question = "Where is the sword?";
	text1 = "_It's in my hat._";
```

If this node is the first node in the conversation, condition should always be set to 1.

question: The question that the player sees. Selecting the question activates this node.

If this is the first node in the conversation, this question is never seen. Just set it to the name of the character.

text1, text2, text3, text4, text5, text6, text7, text8: The bits of text the game displays when the question is asked. Each can be at most 255 characters long. Be careful to not make the response so long it goes off the bottom of the window.

The default behavior is that all text is displayed when the question is asked. However, the code and action characteristics (below) can enable you to only show a few of the bits of text, based on circumstances you choose.

action (Advanced): You can have the game do one of several preprogrammed actions when a dialogue node is selected. This can be extremely useful. These actions can show a subset of the pieces of text available, change Stuff Done Flags, or do a few other things.

The format of an action line is:

```
action = [NAME OF ACTION] [from 0 to 4 numbers, based on the action];
```

The available actions are described later in this chapter.

code (Advanced): The code characteristic is the heart of the true power and versatility of the dialogue. You can have a chunk of code be run when a dialogue node is selected, doing anything the scripting engine is capable of doing. Format the code like this:

```
code =
	[Your code here]
break;
```

The code can do just about anything. Spawn creatures. Give items. Set flags. Whatever. Also, you can use the add_string, remove_string, and clear_strings calls to change which of the 8 bits of text for the node are selected.

Dialogue Action Types

The action is written in the form "action = [COMMAND] [Optional NUMBER A] [Optional NUMBER B] [Optional NUMBER C] [Optional NUMBER D];"

If a dialogue node has both an action and a bit of code attached to it, the action is done first, and then the code is run.

The values of COMMAND and the meanings of the following numbers follow:

DEP_ON_SDF: Text displayed depends on values of a SDF. If SDF (A,B) is less than or equal to C, shows only text1 and text2. Otherwise, shows only text3 and text4. Example:

```
action = DEP_ON_SDF 3 8 12; // If Stuff Done Flag (3,8) is at most 12, shows text1 and text2. Otherwise, text3 and text4.
```

SET_SDF: Sets SDF (A,B) to be C and shows all text. Example:

```
action = SET_SDF 3 8 12; // Sets Stuff Done Flag (3,8) to be 12
```

SET_TO_1: Text displayed depends on values of a SDF. If SDF (A,B) is 0, shows only text1 and text2 and sets SDF (A,B) to be 1. Otherwise, shows only text3 and text4.

PAY: Pay for response. Tries to take D coins and sets flag(A,B) to be C. If the flag is already that value, shows only text5 and text6. Otherwise, if party can pay, shows only text1 and text2. Otherwise, if can't pay, shows text3 and text4. If A or B are negative, no stuff done flag is set. Example:

```
action = PAY 3 8 12 100; // If SDF (3,8) is not 12 and party has 100 coins, sets SDF (3,8) to be 12, takes 100 coins, and shows text1 and text2. If flag isn't that value but party doesn't have money, shows text3 and text4. If flag is already that value, shows text5 and text6.
```

END_TALK: Immediately ends dialogue. Note that any code given for this dialogue node WILL still be run.

INTRO: If party has already talked to a character with this personality number (or a personality number is not given), shows only text5, text6, text7, and text8. Otherwise, shows text1, text2, text3, and text4.

INN: Tries to charge A coins to end dialogue, heals and recharges party, and teleports party to X = B, Y = C. If party can afford inn, shows text1 and text2. If party is on horseback, doesn't put party in inn and shows text5. Otherwise, shows text3 and text4. Example:

```
action = INN 20 40 43; // If party has 20 coins and isn't on horseback, moves it to space {40,43}, heals, and ends conversation. If party is on horseback, shows text5. Otherwise, underfunded party is shown text3 and text4.
```

ID: Starts item identification mode. The cost to identify each item is A.

Making Shops

You put the party into buying and selling mode using the call begin_shop_mode:

```
void begin_shop_mode(char shop_name,char shop_desc,short which_shop,short price_adjust,short sell_adjustment)
```

You will probably most often use this in dialogue, although you can use this call from a town or outdoor script.

Before this does anything useful, however, you need to stock the shop. The best way to do this is to use the call add_item_to_shop:

```
void add_item_to_shop(short which_shop,short which_item,short how_many)
```

Using this call, you can add items or alchemy recipes to a shop. You can add mage spells or priest spells, while specifying the highest level you can learn the spell at. And you can add the ability to learn a skill to the shop, while specifying the highest level you can train the skill to.

When you call begin_shop_mode, price_adjust determines how expensive the shop's wares are. The legal values are:

0 – 60% of regular cost
1 – 80% of regular cost
2 – 100% of regular cost
3 – 130% of regular cost
4 – 180% of regular cost
5 – 220% of regular cost
6 – 280% of regular cost

sell_adjustment determines whether the shop buys items and how much they are bought for. Items sell for a base value of 25% or their cost. The legal values for sell_adjustment are:

-1 – Shop doesn't buy items.
0 – Items sell for base value plus 40%
1 – Items sell for base value plus 30%
2 – Items sell for base value plus 20%
3 – Items sell for base value plus 10%
4 – Items sell for base value
5 – Items sell for base value minus 10%
6 – Items sell for base value minus 20%

Chapter 2.15: Dealing With Errors

One inevitable result of attempting to write scripts is errors. Avernumscript, like all programming languages, simple or complex, is very literal-minded. If you don't give it exactly what it expects, it will get upset. Forgetting a semicolon, misspelling a call name, or giving integers when a call expects strings will bring your script to a grinding halt.

When a script encounters an error, it immediately stops working and an error message appears in the text area. If you are getting weird behavior (creatures not moving or reacting, special encounters not being displayed), scroll up the text area and you will probably see a bright red error message.

When a town, outdoor, or scenario script meets an error, it will stop and not run again. When a creature script encounters an error, the creature stops moving and will be unable to do anything at all. A terrain script with an error will cease to do anything. If a dialogue script has an error, no character in its town will be able to talk. If a Scenario Custom Object script has an error, the scenario will not load at all.

Your only way out is to fix the errors. Fortunately, each error message will give a line and a clue about the problem.

Reloading Scripts.

All scripts for a scenario are reloaded when the scenario is restarted or a save file in the scenario is reloaded. If you are playing the game and you switch out to change a script (and save the change), the change will not immediately be seen in the game. However, if you save and load the save file, all scripts will be reloaded and all changes will be visible.

An exception to this is scripts that stop because of errors. If a script stops because of an error and you save the game and reload, the script will still not work, even if you corrected the problem. You probably shouldn't save the game after you've gotten an error. Otherwise, you will need to do something to force the script to reload (leave town and reenter for scripts in a town, walk a long way away and come back for scripts outdoors).

General Error Messages

The error messages don't always say exactly what is wrong, but they can give valuable clues (including the line number where the game ran into troubles). These errors can appear in any sort of script:

Bad SDF - One or both of the coordinates passed to this SDF are out of range. The first coordinate must be from 0 to 299 and the second must be from 0 to 29.
Bad term in expression - Blades of Avernum found something it didn't expect in a mathematical expression.
Creature script bad header. - Your creature script didn't begin with begincreaturescript.
Creature/terrain-only function - You can only use this call in a creature or terrain script.
Empty expression – The game did not find something in your script that it was expecting (like a right parenthesis).
Failed to load [script name] - You probably specified a script name that didn't exist.
Function error - Something unexpected went wrong when trying to use this function. Double-check it to make sure all the values given to it are correct.
Improper command - Something about your syntax confused the game. Recheck the line.
Internal Error - Some bug in the Blades of Avernum engine causes an internal error, preventing it from continuing. You should send a copy of this script to Spiderweb Software.
Invalid operator - You wrote something that is close to a mathematical operator, but isn't one (like, say, "++").
Invalid symbol - You used a character or word the game doesn't recognize.
Missing (, Missing) - Each "(" must have a matching ")".
Missing = - The game expected an "=" and didn't find one.
No body declared - You forgot the "body;" line before you began the body of your script.
Nonexistent state called- The game tried to access a state in the script that didn't exist.
Outdoor script bad header. - Your outdoor script didn't begin with beginoutdoorscript.

Overlong or unending string - You wrote an overlong string. The maximum length for strings is 255 characters.
Procedure call error - Something unexpected went wrong when trying to use this call. Double-check it to make sure all the values given to it are correct.
Right bracket without matching conditional - You used a "}" without an earlier If or While statement attached to it.
Scenario data file bad header. - Your scenario custom object script didn't begin with beginscendatascript.
Scenario script bad header. - Your scenario script didn't begin with beginscenarioscript.
Script name empty or too long. - Somewhere a script name was expected, you gave a script name 0 characters long or over 13 characters long.
State not ended properly - Your state didn't end with a "break".
State not found - The game tried to access a state in the script that didn't exist.
State with no number - You used a beginstate command without a number after it.
The script [name] is too long. - The program ran out of memory. Make your script shorter.
Variable name too long - Variable names can be at most 19 characters long.
Too many nodes used - Your script tried to run for too long. Cut down on the while loops and things you are trying to do.
Too many strings - You can only use 750 strings in any one script.
Too many variables defined - You can have at most 20 integer variables and 20 string variables in each script.
Town script bad header. - Your town script didn't begin with begintownscript.
Tried to call a non-existent [script type] state - The game tried to access a state in the script that didn't exist.
Tried to run invalid creature script. - The game tried to use a creature script which couldn't be found or had had an error.
Tried to run invalid outdoor script. - The game tried to use a outdoor script which couldn't be found or had had an error.
Tried to run invalid scenario script. - The game tried to use a scenario script which couldn't be found or had had an error.
Tried to run invalid terrain script. - The game tried to use a terrain script which couldn't be found or had had an error.
Tried to run invalid town script. - The game tried to use a town script which couldn't be found or had had an error.
Unexpected end of file - The game reached the end of your script and didn't find something that should have been there.
Unknown command [junk word] - The game encountered a bit of text it didn't recognize. Check to make sure you didn't misspell a sort of command (say, writing "txet1" instead of "text1" in a dialogue node) and that you didn't forget a quotation mark somewhere (writing print_str(You die!"); instead of print_str("You die!");).
Unmatched left bracket - You used a "{" without a corresponding later "}".
Unmatched right bracket - You used a "}" without a corresponding earlier "{".
While loop without left bracket - Every while loop must have a { after the Boolean statement.
Wrong number/type of parameters - You either gave the wrong number or wrong type of information to this call.

Dialogue Script Errors:

Errors particular to Dialogue scripts.

Dialogue script bad header. - Your dialogue script didn't begin with begintalkscript.
Improper block definer - You used a bit of text Blades of Avernum doesn't recognize.
Missing matching break - Blades of Avernum expected a "break;" and didn't find one. You used a "code =" and forgot the "break;".
Missing number - Blades of Avernum expected a number and didn't find one.
Dialogue node out of range - Dialogue nodes must be numbered 0 to 199.
Missing = - Blades of Avernum expected an "=" and didn't find one.

Missing ; - Blades of Avernum expected an ";" and didn't find one.
Missing string data - Blades of Avernum expected a string and didn't find one.
No dialogue node selected - You used a begintalknode without a number right after it.
Tried to process invalid dialogue script - The game tried to use a dialogue script which couldn't be found or had had an error.

Custom Object Script Errors

These errors are particular to custom object scripts.

Array Assignment Error, Assignment error - You tried to assign a number or string that was the wrong type or out of the legal range.
Edited creature out of range (0..255) - The thing you're trying to edit is out of the legal range,
Edited floor out of range (0..255) - The thing you're trying to edit is out of the legal range,
Edited item out of range (0..499) - The thing you're trying to edit is out of the legal range,
Edited terrain type out of range (0..511) - The thing you're trying to edit is out of the legal range.
Improper block definer - Blades of Avernum encountered some text it doesn't recognize. You might have misspelled the name of a characteristic.
Missing array member - You forgot to include the number of the member of the list you are editing.
Missing number - There wasn't a number where one was expected.
Missing string - There wasn't a string where one was expected.
Missing value - There wasn't a number where one was expected.
Out of place import command - You can't use an import command where you tried to use it.
[String] is too long - This string was too long.
Unexpected variable definer - Blades of Avernum encountered some text it doesn't recognize. You might have misspelled the name of a characteristic.
A value assigned to a [something] was out of range. - You tried to give a characteristic a value out of the legal range.

Chapter 2.16: Advanced Topics

Of course, some people will think that everything in this section is an Advanced Topic. But for people who really want to dig deep and do unusual things, Blades of Avernum has a few interesting features.

Groups

It can be extremely handy to clump creatures into groups. Then you can change and affect large numbers of creatures, all with one call.

There can be up to 8 groups of creatures (numbered 0 to 7), each of which can contain 20 creatures. Groups 1-7 are defined by you. Group 0 is special. Group 0 always contains all of the characters in your party, and any characters that have joined you (see below).

To learn about the calls to add and remove creatures in groups, go to the list of calls in the appendices and read the section entitled Grouping Calls.

There are many calls which normally affect single creatures but can affect every creature in a group instead. Consider

void set_name(short which_char_or_group,char new_name) - Sets the name of the character/group to new_name. Maximum name length is 19 characters.

The command

```
set_name(19,"Fred");
```

would change creature 19's name to Fred. But, instead of the number of a single creature, you could pass a group. You do this by passing the number of the group, plus 1000. For example,

```
set_name(1002,"Fred");
```

would change the name of every creature in group 2 to "Fred".

Custom Abilities

You can give characters custom abilities. Go to the list of calls in the appendices and read the section entitled Custom Character Abilities Calls.

Use the init_special_abil call in the scenario script to set the name of the ability and the node in the special script to call when the ability is used. Then use get_custom_abil_uses and change_custom_abil_uses to set the number of times the characters can use the abilities.

The player can reach the abilities by using the Use Abilities button (the same button they use to reach their other abilities).

Adding Characters to the Party

Having characters travel with the group is difficult to implement but interesting in practice. Go to the list of calls in the appendices and read the section entitled NPC Joining Party Calls. There are examples of characters joining the party in all of the scenarios that came with the game.

The call add_char_to_party adds the indicated character to the group in slots 4 or 5. This character should have its own script. The script should set some SDF when the character dies to indicate that the character is dead.

Once that character is in the group, it is looked for and referred to by its Character ID. You can use the character_in_party call to see if the party has the character.

There should always be a place the party can go to get rid of the extra character.

Extra characters use the default character AI for combat. Extra characters do not have their own inventories, do not take experience, and do not gain levels.

Cutscenes

At dramatic moments, you can have Blades of Avernum display an animation of some sort which advances the story. There are numerous examples of this in the scenarios that came with the game. These animations are called cutscenes.

Go to the list of calls in the appendices and read the section entitled Cutscene Calls. These calls shift the view, enable the player to see everything, and let you move around the characters for your little animation.

Section 3: Porting Blades of Exile Scenarios

Chapter 3.1: Porting Scenarios, the Basics

Blades of Avernum is a bottom-up rewrite of Blades of Exile (BoE for short), an earlier, much cruder scenario design system. There are lots of Blades of Exile scenarios out there, and we know that a lot of people will want to port their work to the new system.

The Blades of Avernum editor has a utility to port Blades of Exile scenarios to the new system. It is very important to understand, however, that this feature is quite limited. The resulting scenario will still need a bit of work before it is functional, and even more work if you want to take advantage of the many new features of Blades of Avernum. You will probably need to spend at least an hour for each town and outdoor section to get it into shape.

Alas, this is inevitable. Blades of Avernum is very, very different from Blades of Exile. It uses a completely different game system, a different system for dialogue and special encounters, and the terrain works in a completely different way. The porting tool will save you a lot of ugly work, but you will need to do a lot of polishing on your own.

So before we start, we should go over a few of the differences between the two systems.

Terrain

Terrain was very different in BoE. There was only one terrain type. Walls filled up the entire space. There were no different heights.

In Blades of Avernum, walls only occupy the edge of the space (so you can stand in the same space as a wall). This means that dungeons on Blades of Avernum are much roomier than dungeons in BoE.

Floors and terrain are different things in Blades of Avernum. You can put any terrain type on any floor (unlike BoE, where you had a terrain for altar on one floor, altar on a different floor, and so on).

The Blades of Avernum Editor takes its best guess of what the ported terrain should look like. You will need to go over all the terrain and get it into shape.

Since all of the terrain types have changes, special nodes that check for and change terrain will have to be changed.

Doors and Terrain Scripts

In BoE, you could set a terrain type to act like a door. The values in the terrain type determined how tight the door's lock was.

In Blades of Avernum, there is only one door terrain type. The Memory Cells you set tell the door script how hard the door is to unlock. Therefore, you will need to manually set the difficulty for each of the doors.

Also, you can set a Stuff Done Flag for each door, which is set to 1 when the door is unlocked. Thus, doors can remember that they have been unlocked and stay that way.

For more on doors and Memory Cells, see the sections on doors in the chapter on Editing Towns.

Spells and Recipes

All of the spells and recipes are different in Blades of Avernum. Thus, special encounters that give them will have to be adapted.

Dialogue

Dialogue in Blades of Avernum is completely different from BoE. Each talking node in BoE has two words you can ask about to get to that node, and you can get to any node from any point in the conversation by using the Ask About button.

In Blades of Avernum, conversations are a tree of branching choices. To port your dialogue, you will need to make up a question the party has to ask to see that bit of speech. You will also need to assign dialogue states to every node so that the conversation flows in a natural way.

Finally, there are no “Look”, “Name”, and “Job” nodes in Blades of Avernum. You will need to adapt the old answers to these stock questions to fit in the new dialogue system.

Shops

You can highly customize the items available in stores in Blades of Avernum, unlike in BoE. You use the add_item_to_shop call in the scenario script to put items in shops. This has to be done by hand.

Custom Terrain, Items, Creatures

The BoE and Blades of Avernum systems are too different to be able to meaningfully port custom terrain types, creatures, etc. Custom things have to be ported by hand.

When terrain is ported between the two scenario types, the editor acts as if you never changed any of the default BoE terrain types. Thus, it ports, say, the pillar terrain as if it was still the pillar terrain, even if you changed it to something else. Thus, you may have to do some repainting of custom terrains by hand (or change the pillar terrain in Blades of Avernum to be the custom terrain type).

Conveyor Belts

There are no conveyor belts in Blades of Avernum.

Special Nodes Called When Creatures Die

Special things that happen when a creature dies must be handled through custom terrain scripts. The exception to this is Stuff Done Flags being set when the creature dies. The basicnpc.txt script has this capability.

Traps

Traps are handled completely through terrain scripts in Blades of Avernum. Trap special nodes are not ported.

Quickfire

Quickfire can only be created in Blades of Avernum using the put_field_on_space call. Quickfire you placed yourself will need to be placed again in a script.

Secret Door Special Nodes

In BoE, you could place a special encounter on a space which turned that space into a secret door. This capability does not exist in the new system. You will need to create a custom terrain type that can be walked through.

Timers

Town and scenario timers no longer exist. To implement a timer, set a SDF to the number of turns for the timer, decrement the flag in the START_STATE state of the town or scenario script, and have the script do what you want when the flag reaches 0.

Better Capabilities

The scripting engine for Blades of Avernum is much more complicated that anything in BoE. However, it is infinitely more powerful. Better, once you are used to it, you can create special encounters and dialogue much faster in Blades of Avernum than BoE.

Chapter 3.2: How To Port Your Scenario

Run the editor. Select Import Blades of Exile Scenario from the File menu. Select the scenario, and it will be turned into a folder containing a Blades of Avernum scenario. The folder will contain the scenario file and ported scripts for the old scenario special encounters and dialogue.

Every town and outdoor section will get its own script. The town scripts all get the name "t[number of town][name of town].txt". the outdoor scripts all get names of the form "o[x coordinate of section][y coordinate of section][name of section].txt."

For example, in Valley of Dying things, town 0 is called Fort Talrus, so the ported script is called "t0FortTalrus.txt". The script for outdoor section x = 0, y = 0 is called "o00Northweste.txt". the name is truncated because the maximum length for script names is 13 characters.

The folder of the ported scenario will have the same name as the original scenario file. You can change this. The name of the scenario file will be the same, but with a .bas extension. You can't change this.

Once ported, you will need to pick through all of the scripts, towns, and outdoor sections to smooth out the many rough spots (like store inventories) and rework the dialogue. You will find, however, that the editor porting the scenario will make the job far faster than porting it from scratch.

Translating Creatures and Items

The editor will translate creatures from old to new scenarios as best it can. However, changes you made to creature types won't be translated. So, if you changed the creature type Nephil to Orc, all orcs in your old scenario will be changed to nephilim in Blades of Avernum.

Many items in Blades of Exile have matching items in Blades of Avernum. Those items will be translated. Some items with no corresponding types won't be translated at all. If the editor is unsure what to do with a custom item, it may create an item of type 499 (Untranslatable Item). These are just markers to tell you that there was originally an item there. Items of that type should be deleted.

Section 4: Custom Graphics

Chapter 4.1: Basics of Custom Graphics

We have already learned about the incredible number of ways to customize terrain types, floors, creatures, and items. Blades of Avernum comes with a large assortment of graphics for them, and the ability to play with the colors gives a lot of freedom to customize their appearance.

However, to make a truly custom scenario, you may want to provide your own graphics. This is complicated and tricky, but the result is the ability to make a scenario that truly has its own feel.

Before you begin to study this topic, go to the section on Scripting and read the chapter Quick Introduction to Graphics. Learn what a sheet is and what sort of graphics you need to assign to each of the different objects in the game.

Now then. How do you make a new graphic, how do you put it into the game, and where can those new graphics go?

How Do I Make a New Graphic?

Any graphics editing program. MS Paint. Photoshop. Appleworks. Paint Shop Pro. Just paint it.

Graphics intended for different purposes need to be arranged in the sheet in different ways (more on this later), but your graphics have to have these qualities:

They need to be at 72 dpi.
They need to be either 8 bit (a 256 color palette) or 16 bit.

Once your sheet is made (with new terrain icons, item icons, or whatever), you put it into the game. You can create up to 100 custom sheets. They can be numbered 500 to 599. How they are put into the game depends on whether you are using a Macintosh or Windows.

How Do I Put It Into the Game? (Macintosh)

Using ResEdit (discussed in the Quick Introduction to Graphics chapter), create a resource file which has the same name as the scenario file, but ends in .cmg instead of .bas. (Example: For "valleydy.bas", the custom graphics file is "valleydy.cmg").

Open the file, copy your image in the graphics program, and paste it into ResEdit. Change its resource number, and your sheet is ready to be accessed.

ResEdit is a very old program. Sadly, Apple made the questionable decision to stop supporting it. ResEdit doesn't work for a lot of the tasks it was originally designed for, though it should be adequate to make resource files full of PICTs. If you are using OS X and ResEdit refuses to work, there is a program called ResFool (at http://www.ljug.com/sw/resfool.html) that may meet your needs.

How Do I Put It Into the Game? (Windows)

Figure out what number your sheet will be. Save your image as a bitmap with file name "G[number].BMP". (Example: For sheet 539, save the file at "G539.BMP").

Copy the bitmap into your scenario folder. You're done.

How Do I Port My Graphics From Windows to Mac or Back?

If you want to do it yourself, make the graphics in Windows, copy them to a Mac, and convert them to PICT resources using a program like DeBabelizer or Photoshop.

Of course, you may not have access to such a program. Fortunately, if you submit your scenario to Spiderweb Software and we host it on our web site, we'll happily convert the graphics for you.

If you put both the resource file and all of the bitmaps into the same scenario folder, it can be used on either Mac or Windows without any extra adaptation.

Mac/Windows Conversions and Gamma Values

Gamma settings for Windows and Mac monitors tend to be different. Windows monitors display brighter than Mac monitors. Graphics that look normal on a PC will look too light on a Macintosh monitor, and icons from a Mac will be too dark on a PC monitor.

When porting your graphics from PC to Mac, you may need to make them a little darker in your graphics program, and you may need to make your icons lighter when porting from Mac to PC. To see whether this change is necessary and whether you've adjusted the icons the right amount, you'll need to see how they look in game.

But what can you use custom graphics for? And how should they be arranged in the sheet? And how are they accessed? Read on.

Chapter 4.2: Custom Floor and Terrain Graphics

To see the rough format of a custom terrain icon, look at terrain graphic sheets 600 and 680.

There are two sorts of icons associated with each terrain and floor type: the large icon which is drawn in the terrain area, and the small icon which is drawn in the automap and the editor.

The Large Icons

All large terrain icons are 46 pixels wide and 55 pixels high. They are arranged in sheets in rows of 10, with a black frame around each icon.

Floor tiles need to be drawn with exactly the same diamond shape and positioned as far down as possible in the frame. See sheet 700 for the shape and positioning.

For terrain types like pillars, table, etc., you may need to shift the icon around in the frame a bit to get it positioned exactly.

The Small Icons

All large terrain icons are 16 pixels wide and 16 pixels high. They are arranged in sheets in rows of 10, with a black frame around each icon.

In the editor, the icon is drawn as-is.

In the automap, the automap draws a 4x4 graphic for each space. It gets that graphic by taking every fourth pixel from the small graphic, starting at the upper left. So the upper left pixel of the 4x4 automap icon is the upper left pixel of the small graphic, the next pixel to the right on the automap is the pixel 4 to the right in the 16 x 16 icon, and so on.

Referencing the New Graphic In the Editor

To give a floor type one of your custom graphics, just give fl_which_sheet the number of the sheet (500 to 599) with the custom floor graphic. Same way for terrain types. Nothing special needs to be done otherwise.

Custom Walls

You can customize walls, but it is very difficult. Walls have to fit together very precisely, and there are many small pieces that need to be drawn and positioned very exactly.

The wall types, shapes, and positions for a wall type with all different icons (doors, windows, gates, etc.) can be seen in sheets 600 to 605.

The wall types, shapes, and positions for a wall type with basic walls and doors (and nothing else) can be seen in sheets 606 and 607.

The wall types, shapes, and positions for a wall type with only walls (no doors, windows, or anything else extra can be seen in sheets 610 to 618.

To use a custom wall in a town, just enter its sheet number in the Town Details window.

Custom Cliffs

Look at sheets 650 to 658 to see how cliff graphics are organized. The dark side of the cliff is in the first frame. The light side of the cliff is in the right side of the second frame. There is a two pixel wide strip of the dark side at the left edge of the second frame.

If you follow this format, you can do whatever you want to your cliffs.

To use a custom cliff in a town, just enter its sheet number in the Town Details window.

Chapter 4.3: Custom Item Graphics

Look in Item Graphics to see a wide assortment of sample item graphic sheets. All items graphics are 28 x 28, arranged in the sheet in rows of 10, with a black frame around them.

Each item has two graphics: first, the small icon for an item on the ground in the terrain area. Second, the large icon that appears in the getting, store, and inventory area. Both icons for any given item have to be in the same sheet.

Chapter 4.4: Custom Creature Graphics

Look in Character Graphics to see many sample sheets of creature graphics. For sheets with creature graphics, the icons are 46 pixels wide and 55 pixels high (like terrain). The icons are arranged in rows of four, with frames around them
The graphics in each row represent some pose or action for the creature. There are two sizes of sheet for creatures:

The Small Sheet

First Row - Creature just standing there.
Second Row - Creature attacking.
Third Row - Death animation.
Fourth Row - Four tiny (11 x 16) views of the creature facing in the four cardinal directions, used when drawing it outdoors.

For examples, look at sheets 1450 and 1452.

The Large Sheet

First Row - Creature just standing there.
Second Row - Creature with weapon out.
Third Row - Creature attacking.
Third Row - Creature sitting in chair.
Fifth Row - Death animation.
Sixth Row - Four tiny (11 x 16) views of the creature facing in the four cardinal directions, used when drawing it outdoors.

For examples, look at sheets 1500 and 1513.
To tell the game whether a creature uses the large or small sort of sheet, use cr_small_or_large_template.

Chapter 4.5: Custom Scenario Graphics

There are several other places where you can put your custom graphics in a scenario.

Introduction Pictures

Each of the three introduction screens can have a large custom graphic. Put it in a sheet by itself, with no black frame. Select Set Intro Text 1, Set Intro Text 2, or Set Intro Text 3 and put the number of the sheet in the field at the top. The graphic will be drawn with that text.

Though this graphic can be any size, it probably shouldn't be bigger than 450 x 450.

Custom Pictures in Dialogs

You can have a custom graphic up to 200 x 200 be displayed in a dialog box. Put the graphic in a sheet by itself with no black frame and use the call small_draw_pic_dialog to show it.

You can also have a custom graphic up to 400 x 400 be displayed in a dialog box. Put the graphic in a sheet by itself with no black frame and use the call large_draw_pic_dialog to show it.

Custom Dialogue Pictures

You can set a custom graphic to be displayed next to a character when you talk to him or her. A dialogue picture can be up to 64 x 64 pixels (it can be larger, but this isn't recommended) and should be a sheet by itself with no frame around it. Use the set_char_dialogue_pic call to give a character a dialogue picture.

Section 5: Testing and Distributing Your Scenario

So you've spent many hours learning to write your scenario, and even more of them writing one. It's almost done, and you want to share it with the world. First, however, you need to test it.

Keep Up To Date

Make a habit of visiting the Scenario Workshop at http://www.avernum.com. You can always get advice, useful information, and warnings about known bugs. If something refuses to work and you can't figure out why, it might be a bug and not your error. You might get a push in the right direction at the Workshop.

Playtesting

There are few tasks less welcome and more important than testing your scenario. Without testing, you are almost destined to release a scenario with serious bugs, which may not even be finishable at all. An untested scenario is worthless. Nobody will want it.

The best way to avoid this sad fate for your scenario is to play it. And give it to friends and let them play it. With time and careful play, the problems will become clear, and you can fix them.

Debugging Mode

Fortunately, debugging aids have been provided for you. If you place the call turn_on_debug_mode() somewhere in your scripts, debug mode will be enabled.

To turn on debugging mode, type "}", followed by "=" (so a right bracket, followed by an equals sign). To turn off debugging mode, type an equals sign. When debugging mode is on, combats and death animations will play more quickly. You will not get experience for killing things. Monsters will not drop treasure. Your blows in melee will do much more damage. And you can use a variety of useful commands while in town:

Shift-'k': Kills all hostile monsters in the whole town.
Shift-'m': Magic map. All creatures in the area will show up on the map.
Shift-'g': Enter ghost mode. You will be able to walk through walls, and even through blackness. Type Shift-'g' again to turn this off.
Shift-'h': Heals and restores spell energy for everyone in your group.
Shift-'l': Add light. Gives the party 100 more turns of light.
Shift-'i': Advances the time in the scenario one day.
Shift-'s': Set special items. Dialog boxes will come up asking you for two numbers. The first number you give is the number of a special item. The second number is the number of that special item the party would be set to have (so if you enter 12 and 4, the party will have 4 of special item 12).
Shift-'f': Set Stuff Done Flags. You will be asked for 3 numbers. The first two numbers are the coordinates of a stuff done flag. If the third number is –1, you will be told the value of that flag in the text area. Otherwise, that SDF will be set to the third number. (So if you enter 5, 23, and –1, then the text area will tell you the value of SDF(5,23). If you enter 5, 23, and 8, then SDF(5,23) will be set to 8.

Also, when debugging mode is enabled, looking at a creature gives you a lot of information about that creature.

Debug mode is automatically turned off whenever you load a saved game or enter a new scenario.

You can edit a scenario while playing through it. It is often convenient to keep the editor open while playing through a new town, so you can make fixes immediately. Just bear in mind that changes made to a town or outdoor section usually won't be visible in the game until they town or section is completely reloaded (by leaving and reentering the area, or by loading a game not saved inside the town and then entering it).

Have Others Test It

Game designers are infamous for having blind spots for flaws in their own work. Don't just test a scenario yourself. Get friends to play it. Or, even better, find volunteers over the Internet. Go to the Blades of Avernum forum on Spiderweb's web site (http://www.spiderwebsoftware.com) and offer your scenario for testing. You can get a lot of good help.

Balancing Your Scenario

The most important job when testing your scenario, besides getting it to actually work, is to make sure it is balanced. This means two things.

First, it has to be a proper difficulty. It can't be a pushover for its intended levels, but it can't slaughter them either. Make sure you have a good flow of monsters, usually in groups, but that you aren't overwhelming the party. Make sure that the party is warned before going in to tough situations. Putting 6 dragons in a room may give you a laugh, but it will make your player mad. Remember, your goal is for the player to have fun. Too hard or too easy equals not fun.

Second, it has to give proper rewards. When you give out a powerful item or spell, bear in mind the effects if the player takes his or her party through the scenario multiple times to keep getting that bonus. Don't put a nice reward or gift at the beginning of the scenario, so that the player can just start the scenario, collect the gift, leave the scenario, and repeat until rich. Force the player to at least spend a little time getting the power-ups.

Also, be sure to cap all bonuses. Only give experience with the award_party_xp function, and make sure the experience has a level cap, so that very high level characters can't just keep collecting piles of experience. Gaining levels at high level should take time. Also, make SURE that, when you increase skills and skills with spells, there is a cap. If you have an encounter that increases Bolt of Fire by 1 level, make sure it will only do it up to a certain maximum (3? 5?) Otherwise, a patient player can get the spell to have nearly infinite power.

When giving rewards to the party, show respect to the other scenario designers. Don't force them to balance their scenarios to take into account ultra-powerful players who have receive godlike powers from your scenario. It's a good thing to reward the player for time spent in your scenario. Just avoid huge rewards for minimal effort.

Getting Your Work Out There

Once you are sure your scenario works, you can place it onto the Internet and let people all over the world play it. To distribute a scenario, place it and its custom graphics files (if any) into a folder, compress it (use WinZip on Windows and StuffIt on Mac) and E-mail it to blades@spiderwebsoftware.com. Alternately, you can mail the scenario on a disk or CD to Spiderweb Software, Inc. (the address is in the order form).

Before sending your scenario out, you may want to take a look at the scenario legal stuff, elsewhere in this documentation. A few guidelines when writing your scenario:

i. Overt racism and other sorts of prejudice, as well as obscenity and explicit sexuality, are bad ideas. Such scenarios will probably not be kept on software sites, such as Spiderweb Software's.
ii. Play balance is important. Don't give low level scenarios too much gold, too much treasure, or too much magic.
iii. Don't make things too difficult. Keep the monsters not too tough, and make sure the puzzles are solvable and give at least a few hints. It's always better to be a little too easy than a lot too hard.

And with that, good luck! Tell everyone a story, and share the results of your imagination! After all, there is very little that is more satisfying than creating something truly great, and having it get the appreciation it deserves!