MODDING THE FALL

Disclaimer

Please note that any manipulation done to files or any other data is at your own risk. You should make a copy of every file you may work on. Any changes made to a file could destroy your *The Fall* installation.

Every information or tool contained in this tutorial is used at your own risk. There are no guarantees or warranties stated or implied by the distribution of this information or the corresponding tools. Use this tutorial and the tools at your own risk. Any damage or loss resulting from their use is the sole responsibility of the reader.

The tools and information may contain errors, problems, bugs or other limitations. We assume no liability or responsibility for any such errors, problems, bugs or limitations. We also assume no liability or responsibility for any errors, problems, bugs or limitations caused by using this tutorial.

In no event shall Silver Style be liable for any special, indirect, incidental or consequential damages arising out of the use of tools or information contained in this tutorial. In all other circumstances, Silver Style‘s total aggregate liability to you or any third party, however arising, shall be limited to $15.

“Come closer ... Yes you, Siv.

Ah, look at this posture, strength and spirit, excactly what I'm looking for. Let me ask you just one thing: Do you wanna feel like a god?

Create your own world controlled by your rules, your people, your own history and future? Do you wanna create virtual life? 'Cos that's what I'm offering; nothing more, nothing less.

Ever had a dream or an idea that you would like to share with others? Tell your story in a playable way? Then follow me into this humble tutorial on creating your game with The Fall*. Create a map with the built-in editor. Add your objects to this world and give them functions. Create your heroes, villains and normal folks, make them act, fight and tell your story.*

All it needs is a little (or all) of your time ...“

“If that's really all? Nah, no deal is that perfect. Of course it takes some more general tools and skills, but someone like you has them at his fingertips. Aeh?“

Table of contents

Introduction

The Fall offers many possibilities to create your own story. From changing the existing campaign to creating a completely new one. From maps over importing objects, textures, sounds to scripting NPCs, dialogs, story and action – everything can be done.

But such possibilities need a lot of explanation and skill. So do not let the size of this tutorial scare you and remember: to create a game it always takes a team. Learn and understand the part that most appeals to you and find friends who do other parts. Working as a team is always more fun.

Let's take a short look at the following chapters:

The first 3 subchapters contain real basics. You will need them for nearly every aspect of your creation. They will tell you how to unlock *The Fall*'s files and how to change entries in the Python scripts. You do not need to learn programming, you just need to know how to find the right spot and add an information field for your new map, object, sound or texture. If you are a programmer, these chapters might be too easy for you, but they are short, so it will not hurt to risk a look.

The next big chapter explains objects, textures, sounds and maps. You might say that this is up to the artist. But do not misunderstand me, I will only explain what can be imported into *The Fall* and how this can be done. How to create the material itself is covered by tutorials all over the web.

The second big chapter is about scripts. Scripts hold everything together and define the story. Many aspects of the game depend on scripts, like NPCs, dialogs, schedules, plot and action. You do not know Python? Don't worry; if you know any programming language, you will learn Python in a day.

The last chapter covers more complex topics and is for cracks or those who became ones while working with the other chapters. Here you will learn how to create a usable object or a mod package.

The Directories, the Files and the Extentions

The Directories

Most directories in *The Fall* are compressed into handy archives. You can easily indentify them by their extension '*.ubn*'. These are standard *ZIP*-archives, which can be decompressed using the corresponding tools.

For example the file *scripts.ubn*:

Open your ZIP-tool, select *scripts.ubn*, then hit '*decompress*', or select *scripts.ubn* and click on '*decompress*' in the context menu (you may need to register '*.ubn*' with your *ZIP*-tool). Now select the game root directory (e.g. 'The Fall Last Days Of Gaia') and decompress the files there. After this, there should be a new subdirectory in your game directory called '*scripts*'.

The Fall will now load the files in the uncompressed directory the next time the program is started. However, there are constellations where this might not work properly:

If the program has difficulties loading your changed files from the uncompressed directories, then replace the corresponding '*.ubn*'-file with an empty archive of the same name (as always, NEVER forget to back up before deleting or changing any game files).

The Files

Each directory contains a special set of information:

- '*data*': Mostly information about masks for panels and graphic effects.
- '*gui*': Textures for panels and controls (needs mask information from *data*).
- '*objects*': All models.
- '*scripts*': Global scripts and texts.
- '*sounds*': All sound effects (except speech).
- '*speech_english*': Global speech samples.
- '*terrain*': Textures for terrain and sky.
- '*textures*': Textures for models (objects).
- '*video*': The cinematics.
- '*zones*': Scripts and texts for zones.
- '*zone_speech_english*': Speech samples for zone dialogs.
- '*zone_speech_english2*': More speech samples for zones; keep the directory structure '*zone_speech_english2\zone_speech_english\....*' as it is.

The Extentions

'.ubn'

Look above in chapter: 'The Directories'.

'.py' and '.pyc'

These are python script files in plain text (*.py*) or as bytecode (*.pyc*). At every start, *The Fall* turns all changed python files into bytecode files. For more information on scripting go to chapter: 'Scripts and Basic Structure'.

'.fx'

Data on graphic effects.

'.gui' and '.lay'

Masks that define graphics, panels and buttons given in the textures found in the directory *gui*. They need special tools to modify them; look in chapter: 'Textures and UI Elements'.

'.png' and '.dds'

The texture formats used in *The Fall*. All textures in the game are provided as '*.png*' and '*.dds*'. The game loads only the '*.dds*'-versions, but can load '*.png*'-files if no '*.dds*'-file is found. Chapter 'Textures and UI Elements' covers this topic.

'.diff3d'

3D-models (except persons/animals); these hold the wireframe and the texture's path/name of an object. See chapter 'Adding a new Object' for more.

'.grid_map'

Defines the ground occupied by objects in the game. Also see chapter: 'Adding a new Object'.

'.ogg'

Sounds in OGG Vorbis. Chapter 'Sounds' covers this topic.

'.bik'

Video files.

'.zip'

The zone maps (of course only '*.zip*'-files in the game directory). See chapter 'Creating a Map' for more information.

'.gr2'

Objects animated with Granny 3D. An export filter is not freely available.

Snake Charmer: Scripts and Basic Structure

The Scripting Language Python

The Fall uses the established scripting language Python. Python blends in easily with C++ Code and is freely usable and distributable. You can find a lot of information about Python on: http://www.python.org/

I cannot explain here how to use Python in general, but for a programmer it should be easily comprehensible by reading and modifying existing *The Fall* scripts, as they are distributed in plain text. Use the documentation on the Python homepage and our API documentation if any questions arise.

Chapter 'Debugging' and chapter 'Scripting' give additional information about Python in *The Fall*.

The phrase 'Use the source Luke' can be found somewhere in the Python documentation and this pretty much sums it up: If you can read a programming language, you can learn from existing code. And Python is easy to read.

Plain Script and Bytecode

Python is a script language interpreted at runtime and not compilable into machine code. Every file in the *The Fall*'s installation that ends with '*.py*' is a Python plain text file. These Python scripts are pre-parsed and converted into a bytecode that can be interpreted faster. Python bytecodes are marked with a '*.pyc*'.

At starting *The Fall*, every Python text file is checked and, if newer than the bytecode file, compiled into bytecode. In the Python documentation this process and command is called '*compile*', but as mentioned before, it is not compiled into machine code, as C++ Code is.

You can compile Python scripts into bytecode by using '*py_compile.compile*' (see Python documentation), but for scripting in *The Fall* it should be enough to let the game do it for you.

Basic Structure

There are two main directories for Python scripts in *The Fall*:

- *scripts*: Contains all global scriptings.
- *zones*: Devided into different zones; contains the local scripts for each zone.

Modules found in *scripts* and *zones* can be imported into the game. This includes packages like the information for a complete zone. For instance, to address the main script in *zone_1* you need to import the package *zone_1* and then the module for that particular package: *zone_1.zone_1*.

Let me give you a short explanation of import and namespace. Let's say we want to use the functions defined in the module *global_defines*. One way is to use '*import global_defines*', then every function is addressed as '*global_defines.function*'. The other way is to use '*from global_defines import **', then a function there would be addressed as '*function*'. The same goes for packages like '*zone_1*'.

All commands provided by the engine are imported and addressed in the same way. Every command available and most modules and routines in the script directory are explained in the *API* documentation. Find the Application Programming Interface in this package or at: http://www.the-fall.com/mod-doc/

http://www.the-fall.com/mod-doc/mod-doc.zip

Nobody's Perfect: Debugging Python

The Console

During the game, the key '*F11*' opens the console. This is an input line for the Python command interpreter and above it you can find a text screen listing all prints, bug messages and command responses.

Every Python command and every function call to our engine can be given via this console. Start functions for a test run, check variables, cheat items, do everything normally done with script code instantly.

The Python command *print* can display strings and most variables in the console and tracefile. It is a good way to check code flow and variable values at critical game situations. Even exceptions (script bug) print a detailed report.

Tracefile

Every print and other system message is writen into the file *tracefile.log* in your *The Fall* configuration and savegame directory. To find it in Windows, use the following source name in the explorer: '*%APPDATA%\Silver Style Entertainment\The Fall - Last Days of Gaia*'.

At every start of *The Fall*, the tracefile is overwritten with a new one. So make copies if you need the information contained in it at a later time.

Bugs in objects, maps and textures can often cause a termination of the program. The only way to get information about these errors is the tracefile. But the game-exe is compiled without additional debug information, so do not hope to get much information there.

The Module Debug

The module debug consists of special commands for debugging *The Fall*. It offers cheats, faster gameplay, and information displays.

The function '*debug.cheat(keyword)*' reacts to many keywords, here some examples:

- '*immortal*': Makes PCs (player characters) invulnerable.
- '*pickinfo*': Lists information about an object when left-clicked (essential for scripting).
- '*speedhack*': Press key 'T' to speed up the game.
- '*teleport*': Activates context button '*teleport PCs to cursor*'.
- '*region*': Highlights all defined regions.
- '*help*': Lists all cheats (more than listed here).

The function '*debug.render_debug(object, keyword)*' also uses keywords:

- '*grid*': Shows a collision grid around the object; red tiles are buildings, NPCs have green tiles.
- '*delete*'**:** Deactivates *render_debug*.

See the API for additional functions.

The Command *Reload*

This Python command allows you to reload modules and thereby implement changes during runtime. Change or add scripts and just reload the module to test them. A full restart of the game is

not always necessary.

However, this command is not perfect. For one, it cannot implement new functions, only change their content code. Second, constants are not changed. And third, there is always the chance that it might fail without obvious reasons. Using *reload* will save a lot of time during development, but if something unusual happens, it will be better to restart the program before starting a wild goose chase after a non-existing bug.

Use *reload* as a tool but do not trust it.

Maps, Objects, Textures and Sounds

Welcome to the artist's way. If you did not read the introduction, now might be the time to read at least the chapters 'The Directories, the Files and the Extentions' to learn how to find the different files. But maybe you have already figured this out yourself, then just read on.

The first chapter explains the editor in greater detail. All tools are covered, plus some basic tips.

However, the next three chapters after that only explain how to implement material into *The Fall*. To find tutorials about creating objects, textures and sound, just search the web. There should be plenty of good ones around.

The chapter *Objects* covers objects for the editor, but no equipable items. Textures just need to be placed in the right directory; only UI-elements need additional information. The chapter *Sounds* is even shorter, because only the sound effects need a special list entry.

Creating a Map with the Built-in-Editor

Getting Started

In *The Fall* a region/level is called a '*zone*'. A zone contains a map, scripts and local dialog texts. To work properly in the game, the zone has to be placed in the directory *zones* inside the games directory. Create a directory there with the name of your project. For this tutorial we will just call it: *tutorial_map*.

Now the zone directory should look like this: '*The_Fall**zones**tutorial_map*'. Remember to save your zone in the directory under the same name: '*The_Fall**zones**tutorial_map**tutorial_map.zip*'.

The map must be created inside this directory, because the editor uses the directory structure to create the IDs for the objects placed on the map. Creating the map somewhere else and just moving it to the right directory later on, might cause problems and will make scripting somewhat more difficult.

For more information about directory structure, see chapter 'Introduction'.

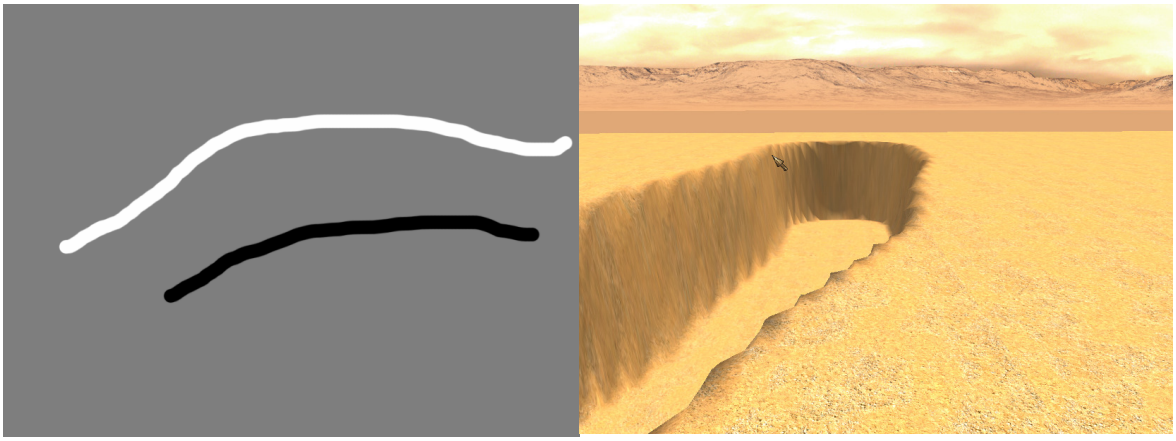
Creating a Height Map

To create the topography of a map, *The Fall* uses a simple grayscale image (*.png*). You can create and edit this picture with your favored graphic tool and save it in *PNG*-format (color).

Let's start with the size of the picture, because the size of the height map controls the size of the zone. For this tutorial we start small, so let's say we want a usable area of 400 x 200 m. Just using this size would make our zone look 'cut off', so every zone needs a designated border. In *The Fall* we have a set border of 200 m on each side. This gives ample room to look around and at the same time hide the 'end of the world'. Adding this figure leaves us with a final size of 800 x 600 m and that is also the picture size in pixel.

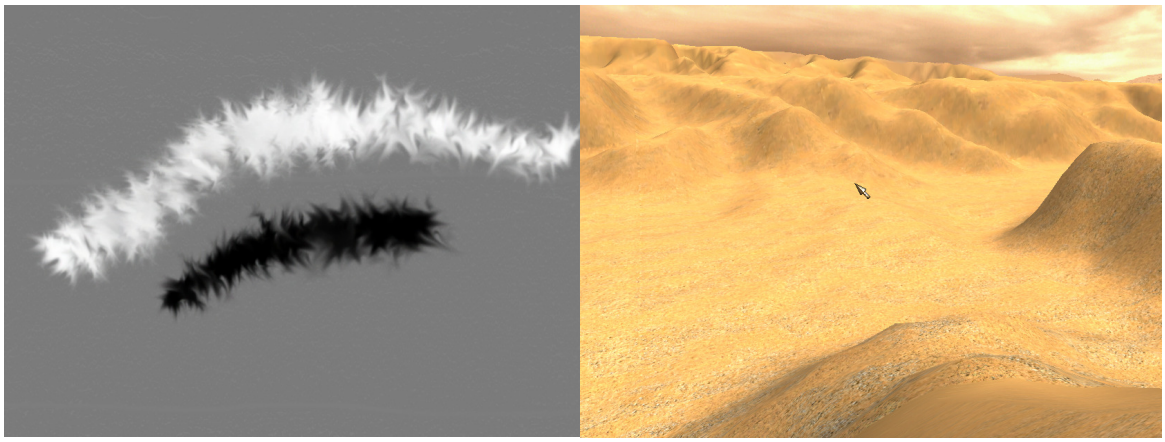
However, it is never that easy, because for technical reasons the picture size needs to be a multiple of 32 and increased by 1. If the image does not conform to that rule, it is cut to the next fitting size. For now, however, our map does not need to be that perfect.

It is time to design your height map. We want to create a hill and a canyon set on a natural flat area. The shades of grey on the map correspond to heights in the game; white is highest and black is lowest. Because we want to create a canyon, we need to raise the surrounding area: start by painting the whole picture in grey. Now add a white line marking the ridge of the hill and a black line for the canyon.



Yeah, not much, so on to the next step. Take your smudge tool and blur the lines with random

strokes. Start on the line and move out a small distance in random directions. Repeat till the lines have a smooth transition to the surrounding grey. But do not try to make it perfect – it needs this random touch to make it look natural.

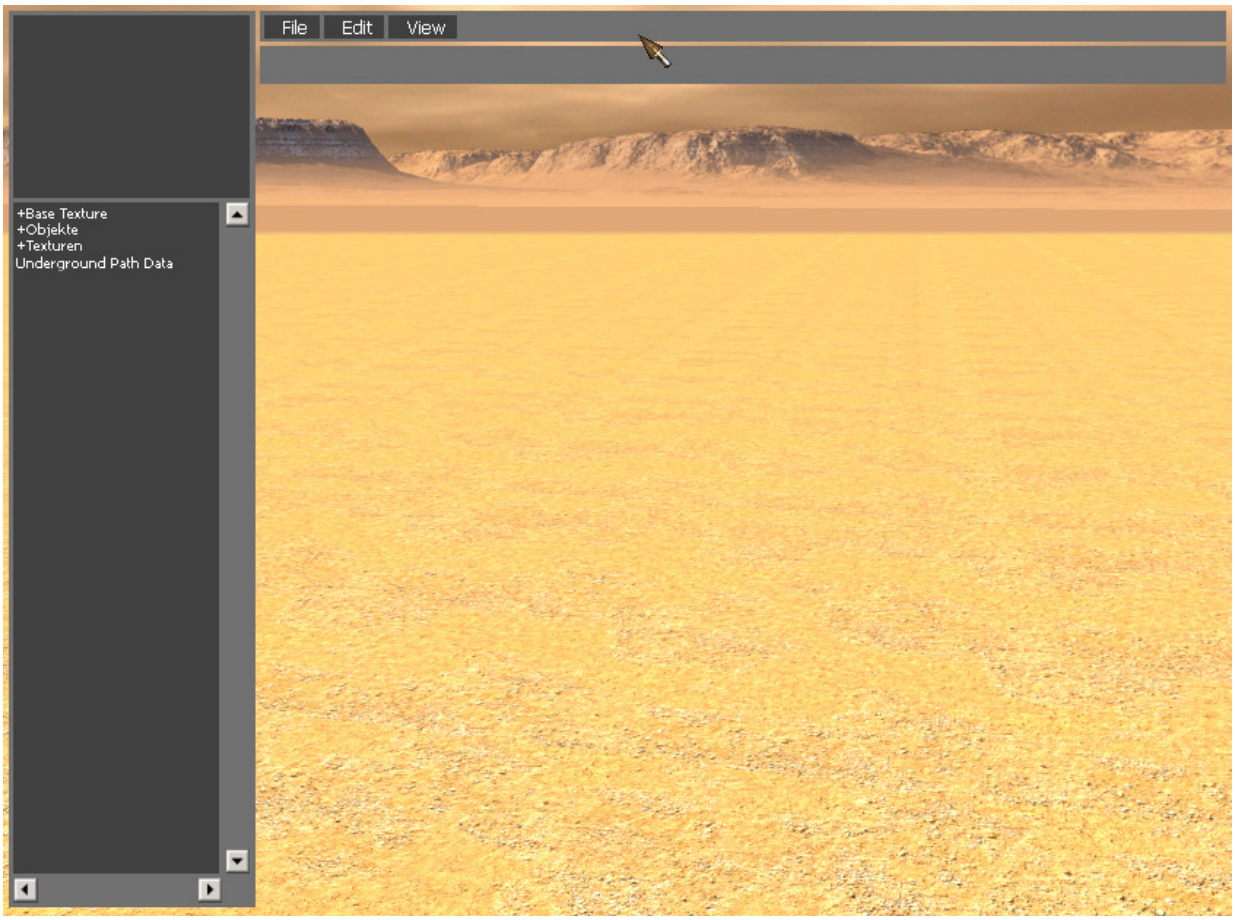


The result is still not the real thing, but good enough for 10 minutes work (the picture '*tutorial_heightmap2.png*' can be found in this package). It is essential to get a feeling for how color corresponds to height and how smooth but still rough it needs to be in order to look natural.

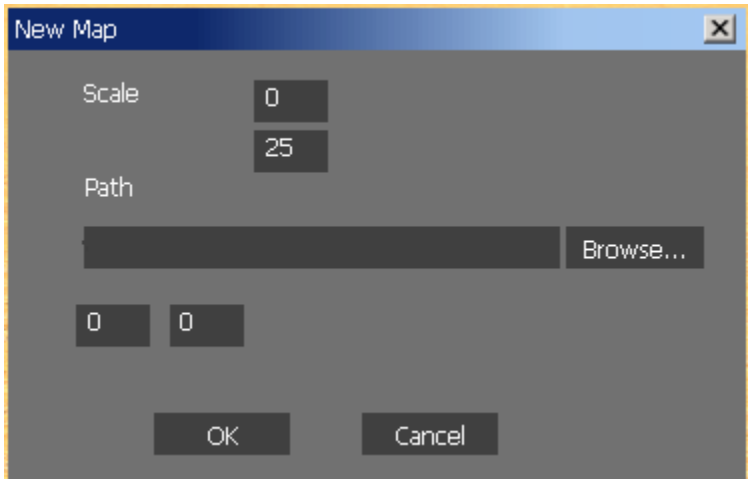
Also, this is only one way to create smooth but random transitions; try your own ideas.

Using the Height Map in the Editor

Now open the editor. Start *The Fall* and click on '*Editor*' in the start menu. The editor uploads with a clear flat ground:



On the left side you can select textures and objects, and the top holds the basic menus. Open '*files*' and select '*import height map*':



The upper two fields labeled '*Scale*' control how color is converted into height; the first field stands for black and the second for white color. Leave them at 0-25 m for the moment.

The field labeled '*Path*' takes the address of your image. You can select the image by clicking on the '*Browse*' button. Now click OK and look at the world you have created.

When you want to edit your height map again, just select '*export height map*' from the same menu to save a new *PNG*-picture representing the actual height map information. This is needed when the original picture is not available or when the height map is changed in the editor (see '*equalize*').

Objects

Now take a closer look at the left panel; in the upper left corner is a preview area that will show an image of the selected object, if available. Below it is the selection tree holding all available objects and textures divided into four groups: '*Base Texture*', '*Objects*', '*Textures*' and '*Underground Path Data*'.

It is time to place your first object: click on '*Objects*' in the selection tree. A list of object classes opens up. Also the editor is now set to '*Objects*' mode; this means that we can select objects, but cannot select textures. In order to change to '*Texture*' mode click on '*Textures*' (more information about textures later).

Now click on '*Village Houses*' and select a house you like. Then place it on the map by just clicking on the target area. Click on the placed house to select it. It should turn red. Click and hold again to move the object around.

Have you noticed the new button row now visible below the menus? This button row lists all available actions for objects:

To place the next object from the selection tree click on the '*Insert*' button and then place the object on the map with the next click. To select another object click on '*Select*' and then on the object on the map. Now move it around by clicking on '*Move*', click and hold and move it around. Next click on '*Rotate*', click and hold again to turn the object around.

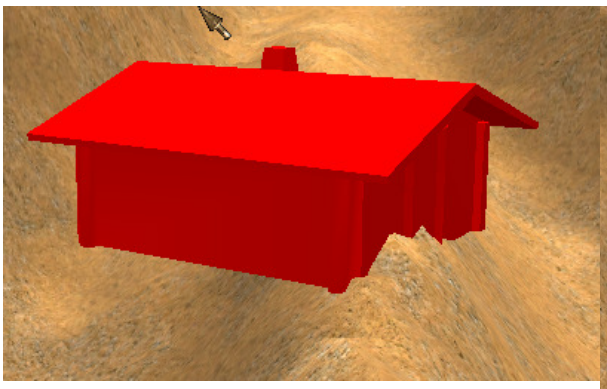
You can change the scale of an object by clicking on the '*Scale*' button, click and hold and move the mouse up and down. All objects that characters interact with like houses, furniture, etc. need a fixed size and cannot be scaled. However, trees, stones and other objects can be changed in size.

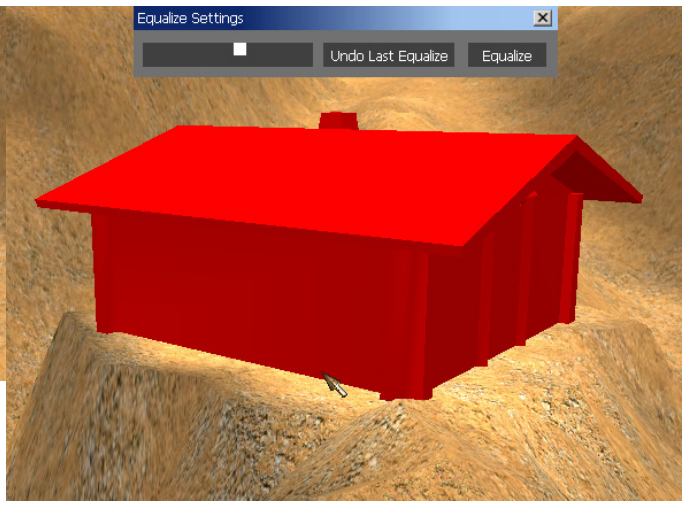
Big objects need a flat ground for placement. To level the ground use the '*Equalize*' function. Just select the object and hit the '*Equalize*' button. You get more control over this function by opening the '*Equalize*' window: select '*View*'->'*Show Equalize Settings*' from the menu.

The left slider controls the height of the levelled ground. Moving the slider to the left sets *Equalize* to the lowest occupied ground. Changes only take effect with the next hit of the '*Equalize*' button and you cannot equalize a changed ground again. So, always use '*Undo Last Equalize*' after every try that is not perfect.

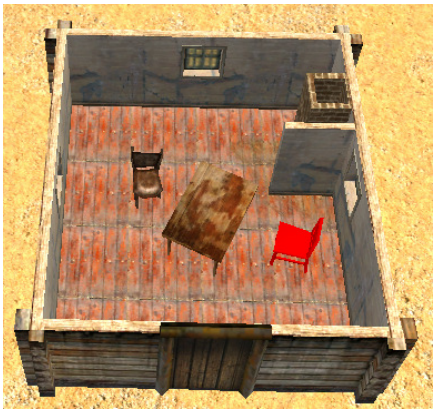
When more than one object is selected, all objects will be equalized to the same height, with *min* and *max* being extended to all objects. To equalize a group of objects separately hold '*Shift*' while equalizing.

Use '*Q*' and click somewhere to find nearby objects that still need equalizing.





Most houses have an interior, which you can access by clicking on '*Interior*'. This removes the roof of the house; all objects placed now become part of the house. Only place objects inside the house, it would look strange to have a table outside the house that is only visible when someone enters the house (which switches the house into *Interior* mode).



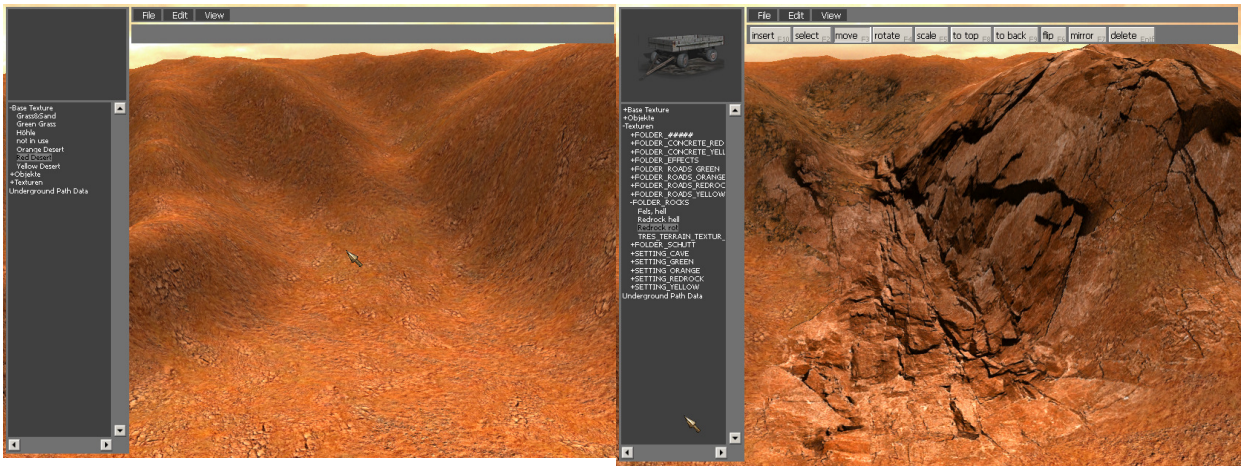
Objects that you do not need anymore can be deleted with '*Delete*'. If you want to replace an object that is used by script first write down its ID. Then delete it and replace it with the new object. Now give this new object the old ID by clicking on '*Identifier*' and entering the old ID.

It is up to you to experiment with these functions, and you will see that every info was worth reading it. But let me add some final tips:

- Too many objects can hamper performance; use enough objects for atmosphere, but do not overdo it.
- Objects cannot be placed 'inside' the ground. Do not place objects overlapping each other, as this may cause texture flicker.
- To search for objects select *'View'* under *'Menu'* or press '*CTRL + L*'. The first object found containing the searched word is selected. Go to all objects found with *'Next'* and *'Previous'*.

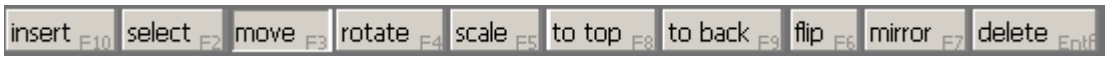
Textures

In *The Fall* all objects are fully textured. The only thing left to do is to texture the ground. Let's start with changing the base texture: Select '*Base Texture*' in the selection tree and pick '*Red Desert*'.



Next take a stone texture to cover the steep hillside. You will find it in '*Texturen*'->'*FOLDER ROCKS*' ->'*Redrock rot*'.

Now move it around and turn it like you did with the objects. Scale works on two axes by moving



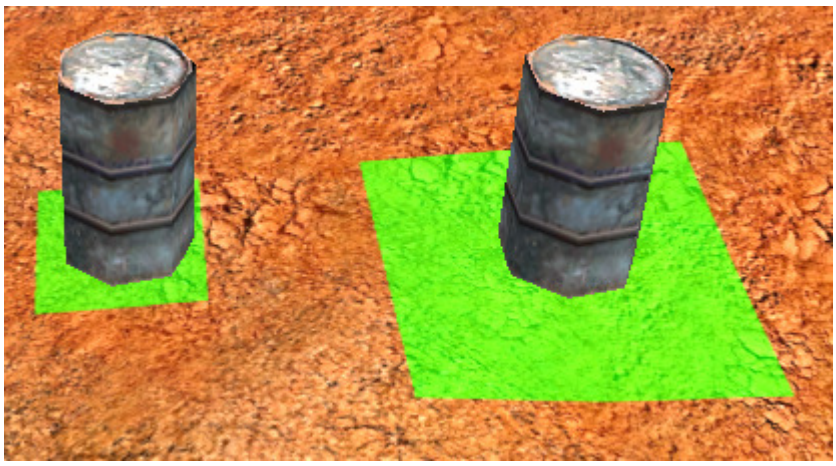
the mouse left/right or up/down. You can also '*Flip*' or '*Mirror*' the texture until it fits into place.

Place more texture around it to make it look like a broken, rocky hillside. When placing textures close or overlapping, you may need *'To top*' and *'To back*' to control which textures you see and which not.

There are no general rules to give here, just the advice to look at every creation from all sides and different distances. Scaling textures can be a tricky beast – better do not scale up; most times it will look grainy.

Ground Data

The Fall uses a grid structure to mark blocked ground. Press '*T*' and look around. Most objects have green tiles beneath them, showing the ground they block for movement. The space needed for an object is always rounded up to full tiles.



Try to place small objects in the middle of tiles, so that they block less tiles. And pay special attention to doors, fences and objects standing close together: Make sure that there is still a tile free between them, so that characters will be able to pass through there.

Objects that NPCs interact with have dummies marking the place the character will move to. Such

places should also be accessible, but do not worry about the ground covered by the object itself, the person send to the object will ignore this specific tile. Press 'M' and click on an object to view its dummies.

To intentionally block areas like steep mountainsides click on '*Underground Path Data*' in the selection tree. Select a brush and paint the blocked area with red tiles.

Experiment

Now it is up to you. Play around, use what you find and talk to others about it. A forum is always a good way to discuss, learn and teach. Have Fun.

Shortcuts

New map ('*KEY_CTRL*', '*KEY_N*')

Open map ('*KEY_CTRL*', '*KEY_O*')

Return to main menu ('*KEY_CTRL*', '*KEY_S*')

Import height map ('*KEY_CTRL*', '*KEY_I*')

Export height map ('*KEY_CTRL*', '*KEY_W*')

Redo texture ('*KEY_CTRL*', '*KEY_R*')

Copy ('*KEY_CTRL*', '*KEY_C*')

Insert ('*KEY_CTRL*', '*KEY_V*')

Hide object ('*KEY_CTRL*', '*KEY_H*')

Display object info ('*KEY_CTRL*', '*KEY_D*')

Search ('*KEY_CTRL*', '*KEY_L*')

Free camera ('*KEY_CTRL*', '*KEY_A*')

Fullscreen ('*KEY_F*')

Delete ('*KEY_DELETE*')

Identifier ('*KEY_I*')

Help ('*KEY_F3*')

Select ('*KEY_F2*')

Move ('*KEY_F3*')

Rotate ('*KEY_F4*')

Scale ('*KEY_F5*')

Flip texture ('*KEY_F6*')

Mirror texture ('*KEY_F7*')

Stack texture top ('*KEY_F8*')

Stack texture down ('*KEY_F9*')

Place object ('*KEY_F10*')

Switch texture/object mode ('*KEY_F12*')

Equalize ('*KEY_E*')

Editor menu ('*KEY_ESCAPE*')

Display ground data ('*KEY_T*')

Display interior ('*KEY_SPACE*')

Display grid ('*KEY_G*')

Display dummies ('*KEY_M*')

Display equalize window ('*KEY_Q*')

Adding a New Object: 'Hey, That's my Red Cube There!'

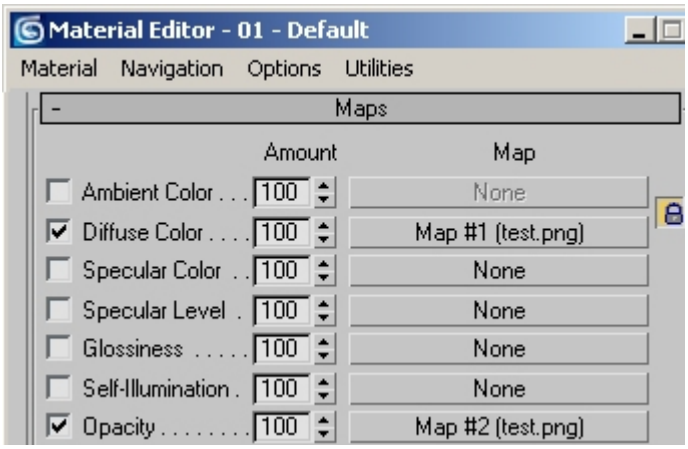
First, I need to say, that I cannot explain here how to build objects with 3D programs. That would be way too much for a specific modding tutorial.

We are using *3DS Max* here and the following shots were made with this program. As these pictures show only two configuration masks, it should not be too difficult to make the same adjustments in your favored 3D tool.

Building an Object

For start, the object has to be of the type '*editable poly*' or '*editable mesh*'. To control the size of the object in the game set the unit size to '*centimeter*'.

The texture is saved in a separate file and has to be set in the fields '*Diffuse Color*' and '*Opacity*'. Just select your texture; path information does not matter for now.



Now you are set to create your item. If you need help here, just search the internet; there will be plenty of tutorials, texts and forums about designing 3D-objects.

Make it an object to be placed on the map for now, because usable items will be explained later in the chapter Items: 'I need guns ...'

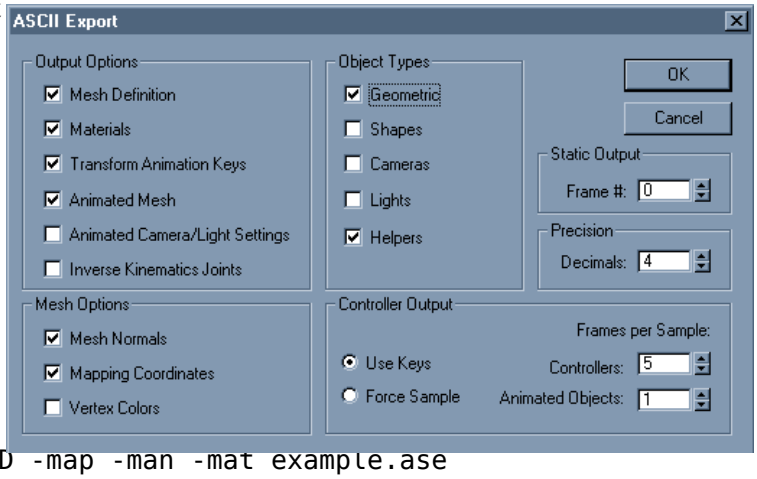
Meet me back in the next chapter for exporting the object into the game format.

Exporting an Object

We will need the Diff3D converter for this, so get it from this package or at:

http://www.the-fall.com/mod-doc/ConvertToDiff3D.exe

The first step is to export your new object into the ASCII Scene Export format (*.ASE*). Activate the following options (picture to the right):



'*Mesh Definition*', '*Materials*', '*Transform Animation Keys*', '*Animated Mesh*', '*Mesh Normals*', '*Mapping Coordinates*', '*Geometrics*', '*Helpers*'.

Then convert the '*.ASE*'-file with the converter ConvertToDiff3D.exe.

Use the following options:

```
ConvertToDiff3D -map -man -mat example.ase
```

Placing an Object in the Right Game Directory

All objects need to be placed in the directory '*objects*'. You can look up all existing objects in the file *objects.ubn*. As described in the previous chapters 'The Directories', 'the Files and the Extentions' this is a zip archive containing the directory *objects*.

You do not need to decompress this directory because *The Fall* checks the directory *objects* and the file *objects.ubn* for registered objects. But it should be easier to place your new object in the uncompressed game resources because you can see the existing directory structure and do not need to recreate it. Also, your object needs a unique name, so you will have to check the objects archive whether your desired name is still unused.

The directory '*objects*' holds the following sub-directories:

- '*additionals\thefall*': All weapons, items (directory '*misc*') and quest items (directory '*quest*') are placed here. You will learn more about these items in the chapter Items: 'I need guns ...'
- '*animals*': Contains all animals.
- '*buildings*': Here you can find all objects that can be placed on a map via the editor (except vegetation).
- '*characters*': Contains all character models (they cannot be created, because the Granny 3D exporter is not free).
- '*vegetations*': Same as '*buildings*' but for vegetation.
- '*vehicles*': Contains the vehicle models.

As your first item should be a map item, it goes into '*buildings*'. Create a new directory in there and place your object file in the directory (the converted Diff3D version of course).

Now for the texture of your new object. This is similar to placing the object but do not create a new directory in '*textures\buildings*', just place your texture in the existing directory. For more information about textures, see the next chapter 'Textures and UI Elements'.

Registering an Object in the Game

The game engine needs to know more about the object than is defined in the Diff3D file and that is where scripting starts. Hey, don't run, it's easy. Just copy the set of an existing object and adjust it to your object. But let me explain what is behind all this.

Your new object belongs to the group '*buildings*', so we have to enter this information in the file '*scripts\\objectsdata\\buildings.py*'. Find out more about Python in the chapters 'Scripts and Basic Structure' and 'First Steps'.

First we need to create a data set for your new object:

```
add_building_data(typeid='RES3D_TRES_OBJECTS_BUILDING_EXAMPLE')
```

The name does not really matter, but to identify its content at a glance it is better to start the ID with a special name. For buildings this is '*RES3D_TRES_OBJECTS_BUILDING_*' (or '*RES3D_*').

Now set the attributes for this data set:

```
objects.set_attributes( object_id='RES3D_TRES_OBJECTS_BUILDING_EXAMPLE',
        diff3d_file = "objects\\buildings\\furnishing\\example.diff3d",
                              height = 2.000,
                          transparency = 1.0,
            editor_directory = "TRES_OBJECTS_FOLDER_KLEINOBJEKTE",
           preview_file = "gui\\previews\\buildings\\example.png",
                          map_type = MT_NONE,
                                  )
```

These attributes stand for:

- '*diff3d_file*': The path identifying your new object (starting after main game directory).
- '*height*': This represents the height of the object as used for line-of-sight calculations. This is independent of the visual height of your object, which you set during object creation.
- '*transparency*': Also for line-of-sight; defines whether one can see through the object. 1.0 means solid, 0.5 is good for fences and 0.0 stands for invisibility.
- '*editor_directory*': For organizing objects in the editor's directory structure.
- '*preview_file*': Also for the editor: Defines a *PNG*-picture as preview when selecting this object in the editor.
- '*map_type*': *MT_NONE* states that no ground is blocked by this item, so that NPCs could walk through it. See next chapter below.

That's it, your very own object usable by the editor. OK, it is not yet in the campaign, but that is another topic. Start the editor and place your object somewhere or read about the editor in the chapter 'Creating a Map'.

Ground Data for Objects

The Fall needs the ground data (the ground blocked for movement by this object) of objects in a separate file. There are three main ways to set this ground data.

No Ground Data

To set no ground data use:

```
map_type = MT_NONE, # as part of the above set_attributes command
```

or:

```
objects.set_attribute(object='RES3D_TRES_OBJECTS_BUILDING _EXAMPLE',
                      attribute="map_type", value=MT_NONE)
# as stand-alone command
```

You can still paint the ground blocked in the editor. See chapter 'Ground Data'.

Generic Ground Data

There are generic ground data sets in the directory '*buildings**grounddata*'. Set one of these with the code:

```
objects.set_attributes( object_id='RES3D_TRES_OBJECTS_BUILDING_EXAMPLE',
          map_type = MT_FILE, # default value so not really needed
                                map_file =
      ("objects\\buildings\\grounddata\\4x4meter.grid_map","soa_map"),
                    ) # keep the form („file“,“soa_map“)
```

Or if you only need a cubic ground data set:

```
objects.set_attributes( object_id='RES3D_TRES_OBJECTS_BUILDING_EXAMPLE',
                           map_type = MT_QUADRATIC,
                     map_quadratic_size = 1, # for 1 m
                                   )
```

Special Ground Data

The tool with which to create the special ground data file is not published yet while writing this tutorial, but maybe it is when you read it. Just have a look in the official forum.

The file is set like the generic one:

```
objects.set_attributes( object_id='RES3D_TRES_OBJECTS_BUILDING_EXAMPLE',
           map_type = MT_FILE, # default value so not really needed
                                map_file =
                                     ("objects\\buildings\\example\\example.gric
                    ) # keep the form („file“,“soa_map“)
```

Textures and UI Elements

Everything displayed in *The Fall* is a texture. Even menus are simple objects clad in a texture. For objects, the way the texture is linked to the mesh is stored in the objects file, but for UI elements this data is saved in a special file created with tools described further down.

The Textures

Format

The base picture must be a 32 bit *PNG* picture (24 bit color, 8 bit alpha). Also, the dimensions of the picture need to be a power of two (..., 512, 1024, ...). No other specifications.

Convert to DDS

A new *DDS* file is only created, when there is no *DDS* file for an existing *PNG* picture. So delete the *DDS* file if you changed the *PNG* picture. Also, as described in the chapter 'The Directories' unpack the *textures.ubn* and *gui.ubn* und replace them with empty *ZIP* files, to make sure that the changed picture is used.

But do not convert the files to *DDS* yourself. *The Fall* can load the *PNG* files and use them. It can also convert and save the *DDS* version automatically if an additional tool is installed.

This tool can be found in this tutorial package or at:

http://www.the-fall.com/mod-doc/png2dds.zip

Unpack the file and place it in your *The Fall* main directory. At the launch of the game, all directories are checked for pictures to convert to *DDS*. Normally this is done only at first launch a day and takes quite some time.

The converter extracts the format from existing *DDS* files in a directory and transfers other *PNG* pictures into this *DDS* format. A *PNG* picture in a directory without any *DDS* picture might cause a convert error.

Registering UI Elements (Adding a New Location to the Map)

As you will most likely use the following tools to create a new icon on the travel map, all steps and examples will be about that topic. For other purposes, only the changing of position and size makes sense if you want to rearrange panels.

The textures and placement of UI elements are set with two tools. Find these tools in this tutorial package or at:

http://www.the-fall.com/mod-doc/GUIResourceData.exe

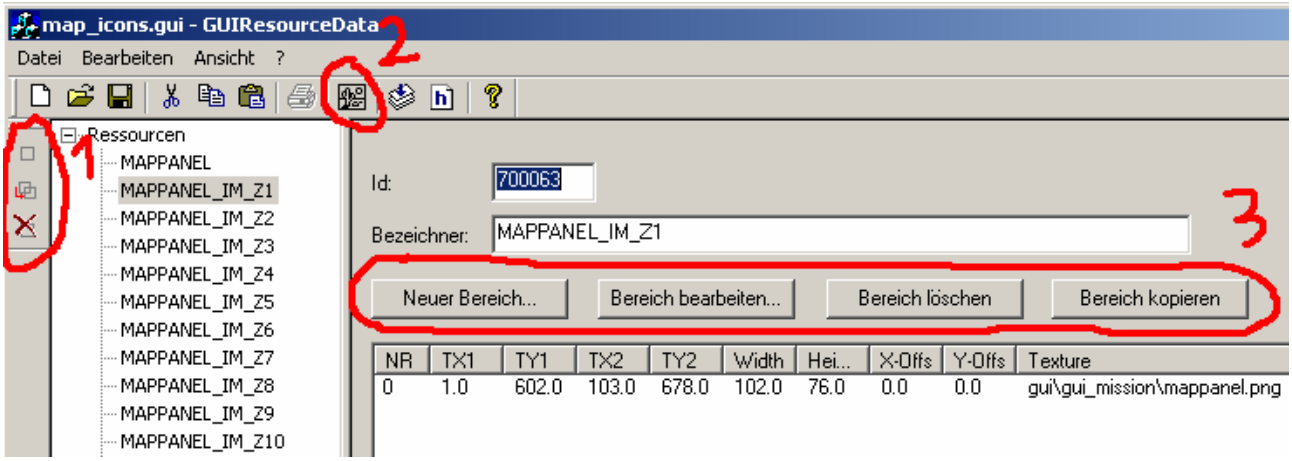
http://www.the-fall.com/mod-doc/LayoutManager.exe

However, these are only tools, which means that they are somewhat intricate to use and not fully translated.

To get started, add your new location icon for the new zone to the texture *'main_directory'\gui\gui_mission\mappanel.png*. Write down the position and size of this icon on the texture and its intended location on the part showing the map.

GUIResourceData

Now start the tool *GUIResourceData* and open the file *'main_directory'\data\guidata\layoutmission\mappanel.lay*. Then select an entry to get the following view:



The button marked with the red 2 imports the texture file. Only imported textures can be used in the element dialog. You could use an additional texture for your new elements, but it is easier to add your new icon to the existing map panel texture.

The button row marked with the red 1, controls the resources:

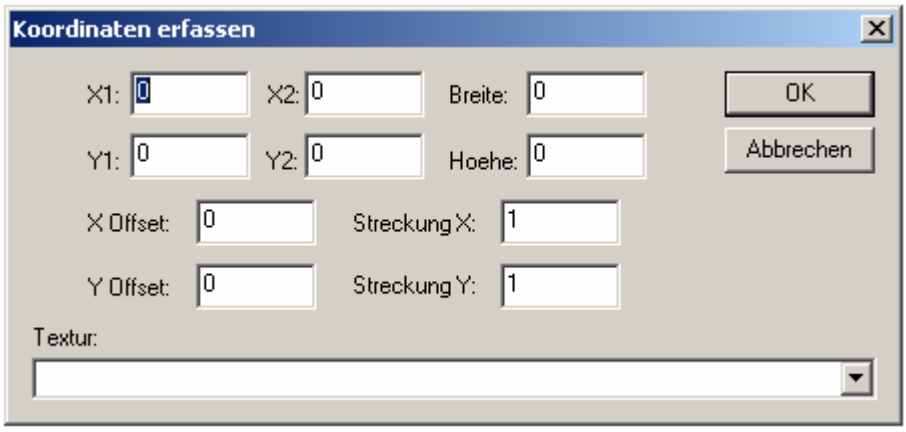
- *Upper button*: Creates a new resource entry.
- *Middle button*: Copies an existing entry.
- *Lower button*: Deletes an entry.

Press the upper button to create a new entry. Select it and change its ID and name ('*Bezeichner*') in the fields above button row three. Use a unique ID in the same range as the other entries; the name of your element should also be unique but is only used as description here.

Now define an element with the button row marked with the red 3:

- '*Neuer Bereich...*' - Creates a new entry.
- '*Bereich bearbeiten...*' - Changes an existing entry.
- '*Bereich löschen*' - Deletes an entry.
- '*Bereich kopieren*' – Copies an entry.

Click on the button '*Neuer Bereich*' and get the following dialog window:



The fields named '*X1*', '*Y1*', '*X2*', '*Y2*' take the coordinates of the icon on the texture. The fields '*width'* ('*Breite*') and '*height'* ('*Hoehe*') display the dimensions of the icon; use these sizes later in the layout manager.

Select the texture file under '*Textur*' and leave the other four fields as they are.

Take a note of ID, width and height to use in the layout manager.

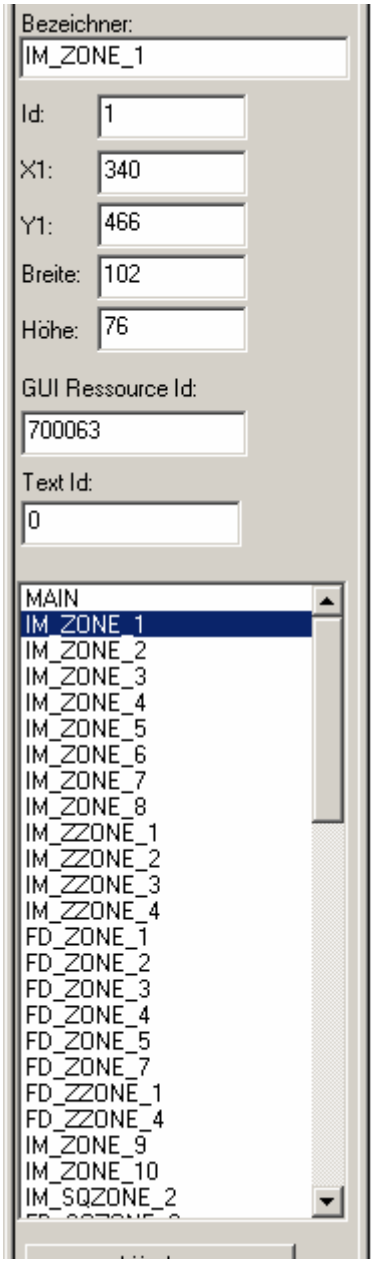
Layout Manager

Start the layout manager and open the file: *'main_directory'\gui\gui_mission\mappanel.lay*.

Start a new entry by clicking on and holding the white field and drawing a box.

Now edit the fields:

- '*Bezeichner*': Name of the entry (see below).
- '*ID*': Number of the element in this layout.
- '*X1,Y1*': Upperleft coordinates of the box.
- '*Breite, Höhe*': Dimensions of the box; need to be identical to the ones set in GUIResourceData.
- '*GUI Ressource Id*': ID set in GUIResourceData.
- '*Text Id*': Not used in *The Fall*.



Enter the name of the new zone in *The Fall*'s scripts and add a prefix:

- '*IM_*': Image; mark your icon this way.
- '*FD_*': Field; the click sensitive area for this icon.
- '*CP_*': Placeholder for texts (no new ones possible).
- '*BT_*': Button: Place of buttons (no new ones possible).

The image entry needs to have the exact dimensions and ID set in GUIResourceData. The click field only needs the right name, but do not overlap it with other click fields. The other two field types cannot

be added, at least there is no way to give them functionality.

Now visit chapter 'Activating a Zone' to learn more about the script part for a new zone.

Sounds

Format

The sound files need to be OGG Vorbis with reasonable sampling rates.

Directories and Files

Effect Sounds

The sounds for effects can be found in the directory *'main_directory'\sounds\ingame*. They have to be added to the sound list (see further down).

Speech Files

There are three main directories for dialogs:

- *'main_directory'\speech_english\globalsmp*: Global speech.
- '*main_directory*'*zone_speech_english*: Speech registered in zones.
- '*main_directory*'*zone_speech_english2*: As above, only divided for installer reasons.

Registering Sound Effects

Sound effects need an identifier and other settings defined in a dictionary in the module *sounds*:

```
sfx_list = {
 #sounds\ingame\animals
 "bear_1": ("sounds\\ingame\\animals\\bear_1.wav",
      DEFAULT_MIN_DISTANCE, DEFAULT_MAX_DISTANCE, DEFAULT_VOLUME),
 }
```

- '*Identifier*': A unique ID (e.g.: '*bear_1*').
- '*File*': Path and file name (e.g.: "*sounds**ingame**animals**bear_1.wav*").
- '*Hearing distance min*': Distance at which sound is played at full volume (e.g.: *DEFAULT_MIN_DISTANCE*).
- '*Hearing distance max*': Distance at which sound is not played (e.g.: *DEFAULT_MAX_DISTANCE*).
- '*Volume*': Volume of sound (e.g.: *DEFAULT_VOLUME*).

Scripting Sounds

Sound Effects

The commands to play sounds can be found in the module *sounds*.

Speech

The engine plays the speech with the same ID as the dialog. Female player characters have the prefix '*aef_*' and mercenaries the prefix '*mercenary_name_*'.

Scripting

In the following chapters I will explain how to create a new zone and fill it with content. The first chapter starts at zero and builds up example content. The other chapters are just collections of information. Maybe boring to read, but full of details.

Remember, most times it is easier to modify existing code than to build something from scratch. And here is where abstraction and autodidaction starts. Look at the scripts, existing objects, maps, etc.

Start with the chapter '*First Steps*', maybe even go through the topics that interest you most right now, but be sure to experiment with existing material while or even before trying to build something yourself.

Always remember: If you are stuck with your new zone, go back to experiment with modding/reading an existing zone. If you cannot get a hold on existing code, then (re)read that specific part of the tutorial and try to build it yourself.

And never forget the forum: Ask questions about what you cannot find out yourself and teach others the things you understand.

First Steps: 'Dear Diary: Today I Met ...'

This really is a 'work with me' part of the tutorial and it will guide you through the first steps of scripting. Let's start with some preparations:

The zone for this tutorial is provided as a mod. Look for the directory '*mods*' in your *The Fall* main directory. If you cannot find the mod *tutorial_mod* in there, just unpack the *tutorial_mod.zip* provided with this tutorial and place it in *The Fall*'s main directory (directory structure *mods\\tutorial_mod* saved in zip file).

Next, you will need your code editor. Visit www.python.org for a nice overview. Open the code editor and open all *.py*-files found in *mods\\tutorial_mod\\mod_1*. These are all script files that define the tutorial zone. To make it easier, I placed all scripting examples for this tutorial as comments into them. Just find the lines for the chapter you are currently reading and uncomment them. That is, if you do not want to write them yourself.

```
#This is a comment; just delete the '#' to make a normal command out of it
```

It is always handy to have additional code examples, so if this interests you just unpack the *scripts.ubn* and *zones.ubn* (see chapter 'The Directories') and you will have all Python code used in *The Fall* at hand.

And do not forget the API. It can be found in this package or at http://www.the-fall.com/mod-doc/mod-doc.zip

Also, it is easier to switch tasks between the running game and your code editor when the game is running in window mode. So change this in the external options and also set a low resolution.

The last step is to activate the mod in the ingame options. Open '*Options*' in the '*Startmenu*' and set '*Use modification 'tutorial_mod'* ' to *on*. You need to restart *The Fall* if you change mod options.

Now press '*Start*' (yes, the button to start a campaign) and see your Alter Ego standing in a desert.

The Meeting

It is really boring standing there alone in the desert, so let's script some company. Open the file *npc.py* (with your code editor) in the *mod_1* zone (*mods\\tutorial_mod\\mod_1\\npc.py*) and look at line 33ff:

Dictionaries and Lists

There you will find examples for NPC data sets. Copy one data set into the empty dictionary *npcs*. It should look like this now:

```
npcs = {
 'FEMALE':['female', 'NPCS', 300.00, 300.00, globaltext.M_NAME_FEMALE,
 180.00, '', 'VILLAGE_PEOPLE', 1, 'low', 'female', 'white',
 'ss_001', 'evil01', 0.00, ACTIVE],
 }
```

Now we have one dictionary *npcs = {'name' : [..]}* with the entry named '*FEMALE*' and a list *[..]* of information. The entry name will be the ID of the new NPC and both names have to be unique (in their namespace) so change '*FEMALE*' into '*NPC_1*'.

The NPC would be created on the same spot as your player character, so better place her a few meters away by changing her x,y position: ... *,300, 305,* ... Give her a new name: Just replace *globaltext.M_NAME_FEMALE* with '*girl*'.

```
Npcs = {
 'NPC_1':['female', 'NPCS', 300.00, 305.00, 'Girl',
 180.00, '', 'VILLAGE_PEOPLE', 1, 'low', 'female', 'white',
 'ss_001', 'evil01', 0.00, ACTIVE],}
```

That should be enough for now. As I explained above, you will find these lines as a comment at line 45ff. Just uncomment them to make them work. Do this now and then restart *The Fall* to visit your first NPC.

You will find more information about NPCs in the chapter 'NPCs'.

Her First Words

It would be nice to talk to her, so let's give her a small dialog. Dialogs are placed in the file *dialogs.py*, so open this file in your editor (again with the code editor and the directory *mods\\tutorial_mod\\mod_1*).

Define a Function

We start by defining a new function for your dialog:

```
def tutorial_dlg_1():
```

Now is a good time to open the Application Programming Interface (API). It contains descriptions and syntax for most functions defined in the engine and modules in the scripts directory. Open the module *dialog* and read the examples. With this information build the following dialog:

```
def tutorial_dlg_1():
 dialog.init_dialog(type=DIALOGSMALL)
 dialog.add_text(text='Hi, do you have some meat?', speaker='NPC_1',
                                    follower='ENDDIALOG')
 dialog.start()
```

Look at the indentation: The three dialog lines are indented to mark them as a code block for Python. In Python indentation is not only used for visibility but also to mark code blocks. In the declaration of NPCs above it was only used for visibility, though. So just remember, indent every code block and use additional indentations for visibility only for comments and lists/dictionaries (everything in brackets).

For more information on dialogs visit chapter 'All Talk'.

Understanding Events

We still have one problem to solve: The engine needs to know when to start our function *tutorial_dlg_1()*. All code in *The Fall* (at least all Python code) is called by events. Just think of it this way: The player does something and the game reacts to that action. All events that the engine can call are defined in *object_events* and *system_events*. All of these functions are also called in the module of the actual zone *mod_1.py*. Some of them are in the basic zone script, but you can add almost all other event functions to this module which are defined in *object_events* and *system_events* (for exceptions, see chapter 'Events').

Using the Event on_approach()

Let's use our problem as an example: 'When the player approaches the NPC, start function *tutorial_dlg_1()*'. The program reacts to the action 'approach', so let's search the API for something close to *on_approach()*. This time the right event is not hard to find because its name fits its function. However, it is not always that easy.

When you look at the module *mod_1.mod_1* (*mod_1.py* in the same directory) you will find a lot of events, some of them with basic code. The function *on_approach()* in there calls the *dialogs.on_approach()*, so you just need to call your function in the *dialogs.on_approach()*.

```
Def on_approach(target, operator):
 # If talked to 'NPC_1' call dlg_1
 if target == 'NPC_1':
       tutorial_dlg_1()
```

The Command reload()

Again you will find the new lines as comments in the module and only need to uncomment them. But do not close *The Fall* right now. There is another Python feature I want to introduce to you: '*reload()*'. The command *reload()* will load a module again and also all changes made to this module. You have changed the file *dialogs.py*. The Python name for this module is *mod_1.dialogs*, so try to enter this in the console (open with 'F11'):

```
reload(tutorial_mod.mod_1.dialogs)
```

You will find more information about the console and the command *reload()* in the chapter 'Debugging Python'. Just two things: The command *reload()* cannot add a new defined function. You will always need to restart *The Fall* when you add a new function to a module. But do not worry about that right now, I have placed all functions needed in this tutorial in the *mod_1* module so that you can play around with this command. Second, the command *reload()* is only a utility tool and definitely has some bugs: Always make a full restart if you encounter 'unusual' behavior! But still, the command saves more time than its problems should cost you.

Hunting

So she wants some meat. Could have been worse: Ever tried to find a rose in the desert? Just creating some meat would be easy, but for a programmer the easy way is no fun and that is why we will start with 'walking' meat.

Consulting the API turns up the following command:

```
system.create_animal(id, type, party, x, y, direction=180)
# inserting needed parameters makes it to
system.create_animal(id = 'ANIMAL_1', type = ANIMAL_WOLF,
       party = 'ANIMALS', x = 315.00, y = 300.00, direction=180)
```

The Event on_map_loaded

But where to put that line? This command should be executed at the initialization of the zone, so we need an event for this. The right event here is *on_map_loaded*. This event is the backbone for every initialization done in a zone at first start. All objects, variables and recurring actions are defined and started here. Remember that we created the NPC by adding the info to a list? The function using that list to create the NPC is also called by *on_map_loaded*.

Just add above command at the end of the event *on_map_loaded* in the module *mod_1* (line 86). But this time restart the game, because the reload command will not execute code and without it

there would be no new animal. Welcome back to the desert, the girl and the wolf. However, it is a stupid wolf and you could just beat it senseless, because it does not even know how to defend itself. But that would be cruel and we want a challenge, so let's teach the wolf a trick.

Attributes

Take a look at this code line:

```
objects.set_attribute(object = 'ANIMAL_1', attribute = 'foes',
       value = ['UISPIELER'])
```

Attributes link data to an object. In the scripting layer of *The Fall*, objects are no real programming objects but just addressed as data sets. The attribute system is the scripting equivalent to assigning variables to a code object. Attributes are explained in detail in the API in the module *attributes*.

The attribute *foes* controls which characters an NPC would attack. *UISPIELER* (UIPLAYER) is the group for all player controlled characters (PCs). Do not forget that the attribute *foes* needs a list *[..]*, even if only one enemy (or group) is set.

Another Event Example

So, your actual problem is: 'If the animal is attacked, set its foes attribute so that it fights back.' Again, you are looking for the right event to place the line in. Search the module *object_events* in the API for something like *on_attack*. The same function is defined in your *mod_1* module (one last time: module *mod_1* is file *mod_1.py*) and you can place the line there:

```
def on_attack( operator_id, target_id ):
 """Called when an object is attacked."""
 if target_id == 'ANIMAL_1': # If someone attacks ANIMAL_1 do...
       objects.set_attribute('ANIMAL_1','foes',['UISPIELER'])
       return
       # stops function, do not call global for this target_id
```

Passing Parameters to a Function

Did you notice that the command is a little different from the above? Python is flexible and you do not need to name parameters as long as you write them in their right order:

```
objects.set_attribute(object = 'ANIMAL_1', attribute = 'foes', value =
['UISPIELER'])
# is the same as
objects.set_attribute('ANIMAL_1','foes',['UISPIELER'])
```

Finally, use the command '*reload(tutorial_mod.mod_1.mod_1)*' in the console to import our changes and attack the wolf (remember: *reload* should work perfectly here, but restart if something goes wrong).

Using Debug Cheats

Ups, you died? Time to tell you about some cheats. Open the console and import the module debug *import debug*. Most cheats are entered like *debug.cheat('name')*, where name is replaced by:

- '*immortal*': Makes PCs (player characters) invulnerable.
- '*pickinfo*': Left-clicking on objects displays information (ID, Pos, ...).

- '*speedhack*': Hold key 'T' to speed up the game.
- '*teleport*': Activates context button '*teleport PCs to cursor*'.
- '*help*': Lists all cheats.

Load the autosave, activate the cheat '*immortal*' and try the wolf again.

Getting the Meat

The wolf is dead, but still the girl is not satisfied. You could script a knife and a high value in survival for your AE (ALTER_EGO; your character), but again we want to learn some things here.

So we define our problem like this: 'Use the wolf to get meat.' I mean the context button '*Use*'.

Finding Information About Items

Let's start by getting some information. You will need the unpacked directory scripts to have a look at the global modules. Or just read these lines here and forget about finding them yourself for the moment.

Open the file *scripts\\itemdata\\items.py* in your code editor. We want some meat, so search for it. You will get a lot of hits, because every line of a data set holds the object ID. Go to *SET_MEAT_RAW*: The first line creates the item type followed by lines setting the needed attributes to define this object. No ingame object is created here, just its type. Every item of this type will be a copy of the original data set and may contain additional or changed attributes.

Now we need the command to create an object in an inventory. Again take a look at the API, this time in the module *objects*, because the item has to be linked to an object (wolf, AE):

```
objects.create_item_in_inventory('ALTER_EGO','SET_MEAT_RAW')
```

If you want to create an item on the ground it would be *system.create_object*, but right now we need the above one.

Event Groups

Now the events for '*Use*': I have already told you how to search for something with '*on_*' and '*use*' and '*on_use*' is one event that we will need. However, it is a little more complex now, because we also need '*can_use*'. Some events have an '*on_*' and a '*can_*' (and even some more) variation. The '*on_*' event is triggered when the action is really executed and the '*can_*' decides whether the player can execute the action at the present moment.

First, allow AE to use the wolf:

```
def can_use( character_id, object_id ):
 """Returns True for items the character is able to use."""
 if character_id == 'ALTER_EGO' and object_id == 'ANIMAL_1':
       return True #allows action (on_use is started)
```

Now the code to create the meat. Again you should check which objects interact:

```
def on_use(character_id, object_id, item_id, item_type):
 """Is called when the player wants to use an item."""
 if object_id == 'ANIMAL_1':
       objects.create_item_in_inventory(character_id, 'SET_MEAT_RAW')
       return True #stop here, because event is handled
```

Updating Code to Full PC Group

I used the variable *character_id* instead of the ID '*ALTER_EGO*' and one might say that is a little dirty, but it just left me another option: All PCs should be able do do most ingame actions, so let's change the *can_use* to:

```
def can_use( character_id, object_id ):
 """Returns True for items the character is able to use."""
 if character_id in system.get_pcs() and object_id == 'ANIMAL_1':
       #system.get_pcs() gives a list of all pcs
       return True #tell engine to start on_use
```

So, enter, copy or uncomment the lines, reload the module *mod_1* and try it.

The Second Dimension: Time

Yes, second dimension: The code runs in one and the game moves on in time. That is the last concept I will explain in this chapter.

Animation Stack

First, we need a command that takes time to execute: Search the API for a command to play an animation for a character and find a nice animation in the scripts directory (try *scripts\\anidata\\male.py*):

```
def tutorial_animation():
 character.play_animation('ALTER_EGO',
       'CM_NPC_STEHEN_LIEGESTUETZE_START')
 character.play_animation('ALTER_EGO',
       'CM_NPC_STEHEN_LIEGESTUETZE_LOOP')
 character.play_animation('ALTER_EGO',
       'CM_NPC_STEHEN_LIEGESTUETZE_ENDE')
 print "finished tutorial animation"
```

The character should play through these animations one after the other, whereas code commands would be executed in one frame (with no difference in game time). But try it, start this function *tutorial_mod.mod_1.mod_1.tutorial_animation()* in the console. You will see the print right after executing. Close the console and watch how the AE plays through these three animations one after the other. But why?

Every object has a command stack for actions that take time. If one is finished, it will take a new one from the stack, or execute a waiting animation. This works for commands like *play_animation*, *move_to*, *turn_to*, *pick_up* and every other command that uses an animation.

Callbacks

You should ask yourself now: "But how do I execute new code after an action is finished?" Look up the *play_animation* command again in the API:

```
play_animation(char, animation, callback)
```

Yes, the parameter *callback* is able to call a function after the animation is done. But it needs a pointer to that function: In Python a function called by *function_1()* is executed instantly and just writing *function_1* gives the pointer to that same function to be executed another time.

Try the following in the console:

```
objects.move_to_object('ALTER_EGO','NPC_1', callback =
tutorial_mod.mod_1.mod_1.play_dialog)
```

Now the dialog starts when the AE reaches the girl. Whereas the following command would start

the dialog and the movement at the same time:

```
objects.move_to_object('ALTER_EGO','NPC_1')
tutorial_mod.mod_1.play_dialog() # or dialogs.tutorial_dlg_1()
```

On your first try with callbacks you might get an error, because the engine tries to pass the object ID as a parameter to the called function. That is why the function *play_dialog* holds a parameter:

```
def play_dialog(character_id = None):
 dialogs.tutorial_dlg_1()
```

You cannot give specific parameters to be passed to your pointed to function, but the engine automatically passes the object ID.

Lambda

I know this is hard to understand if you see time and code interacting for the first time, but still I have to throw in another: Lambda and lambda.

First, forget the Python lambda, because it cannot be saved and would cause a lot of trouble if used for zone scripting. You will find the Python lambda used in schedules, but they are not saved in the savegame but uploaded from script and therefore compatible. Everywhere else you should use our implemented function, the 'big' Lambda.

Lambda is a way to store a command with parameters for later execution. The pointer we used above in the callback could not take parameters, but often enough you will need to pass additional information.

```
lambda_string = "objects.move_to_object( 'ALTER_EGO', 'NPC_1') "
lambda_function = Lambda(lambda_string)
```

The function and parameters are passed as a string and stored in an executable code package that fits in a variable.

Wanna see what executable code package means? Just enter the following lines into the console:

```
print_message = Lambda("print 'HELLO' ")
print_message()
```

And that is where the fun begins, but we are a mile too far for “First Steps.“ Do not worry, if you cannot get callbacks and Lambda right in your first scripts, just copy and manipulate what you need until it becomes clear. See you in the following chapters.

Oh, want some homework? Try to write the following code:

- After getting the meat from the wolf, make AE walk to the girl.
- Then turn to her (use stack for move and turn to).
- When AE looks at her (callback to a function) start a dialog with two choices: 'Here some meat'/'Bye'. (Search the API for the commands and structure)
- Play an animation for the girl after choice 1 (store the *play_animation* in a Lambda for this callback).

You can find the code in the functions *callback_tests* and *callback_tests_dlg_2* at the end of module *mod_1.dialogs*.

```
# add following line in mod_1.on_use
       dialogs.callback_tests()
```

```
# and these are in mod_1.dialogs
def callback_tests():
 """ Move AE to Girl and faces her """
 objects.move_to_object('ALTER_EGO','NPC_1')
 # write both into stack
 objects.turn_to_object('ALTER_EGO','NPC_1', callback =
callback_tests_dlg_2)
 # calls next function when finished turning

def callback_tests_dlg_2(character_id = None):
 """ Initiate dialog and play dance for girl after choice 1 """
 dialog.init_dialog(DIALOGLARGE)
 dialog.add_choice('Hi, here is your meat.', speaker = 'ALTER_EGO',
       follower = 'ENDDIALOG', callback =
       Lambda("characters.play_animation('NPC_1','CM_NPC_TANZEN_1')"))
 # Above the full play_animation command as string in Lambda assigned
 # to the callback
 dialog.add_choice('Bye.', speaker = 'ALTER_EGO',
                   follower = 'ENDDIALOG')
 dialog.start(False)
```

Understanding the Zone Structure

A Zone‘s Directories and Files

Base Name, Directory and Module

Every zone needs its own base directory and the name of this directory sets the zone’s internal name. This directory is also the module containing all scripting modules. The main script module has to be named like the base module and together they are the identifier in all lists (it is a little different in the mod directory; the equivalent mod name follows in brackets):

```
'zone_X' ( 'mod_1' )# basic zone name; used in
global_defines.set_zone_enabled()
global_defines.set_zone_enabled('zone_X', True)

'zone_X.zone_X' ( 'tutorial_mod. mod_1.mod_1' ) # internal identifier for
data(), and most lists
data('zone_X.zone_X').variable = True
# if the data is used for current zone, it is enough to write:
data().variable = True

# addressing a zone function via console
# base module / main script module / function
zone_X.zone_X.function_1() ( 'tutorial_mod.mod_1.mod_1.function_1()' )
```

Files and Directories in the Zone Directory

- *zone_example.py*: The main script module (needed).
- *dialogs.py*: The module holding dialog scripts (changeable; only used by main script).
- *npc.py*: The module containing all NPC creations (changeable; only used by main script).
- *const.py*: The module setting all variables (changeable; used by all local modules).
- *schedules.py*: The module for schedules (changeable; only used by main script).
- *__init__.py*: This file turns the directory into a Python module (needed).
- *zone_example.zip*: The map created with the editor (needed).
- *zonetext*: Directory/module containing all modules for local texts (needed).
- *zonetext\\english.py*: The module containing all local texts (needed for current language).
- *map.png.dds*: The picture for the minimap (needed).

All modules marked with '*changeable*' are parts of the main script module and are only used to organize it in smaller and clearer chunks.

Events

All code in *The Fall* (at least all Python code) is called by events. Just think of it this way: The player does something and the game reacts to that action. All events that the engine can call are defined in *object_events* and *system_events*. All of these functions are also called in the module of the actual zone *mod_1.py*. Some of them are in the basic zone script, but you can add almost all other event functions to this module which are defined in *object_events* and *system_events*.

Exceptions to this rule are utility functions used by events defined in these modules. Check the API or read the function to find out, whether it is an event or a utility function. If unsure, try the function in the zone module and write a print in it.

Most events need to call the global function. Some only need to be executed. For others you have to return their value ('*return object_events.function()*'). Look at these functions in the events modules to find out how to connect local to global function. When your code solves the problem and you do not need or want the global behavior, just finish the function with *return*:

```
def function_1(): # return a value or variable
 if action == True:
       return True
       # the following code of this function would be ignored
 # link to global event
 return object_events.function_1()

def function_2(): # or return nothing and just finishing this function
 if action == True:
       return
       # following code ignored
 # link to global event
 object_events.function_2()
```

Event Prefixes and Types

Prefix 'on_': These events react to a situation. Most of them only execute code, need to call the global event (the local functions need to call their equivalent in *object_events/system_events*), and have no return value. But a few have a reaction to return like *on_wait_for_commando()*.

Prefix 'can_': These events check if a special action can be taken. Most are linked to the context menu. They need to return the return value of their global equivalent and their return value always needs to be *True* or *False*.

Prefix 'calc_': These events return a value to the engine. Normally only used on global scale.

Prefix 'get_': As above, these events return a value and are normally for global use.

Prefix 'post_': Called after an '*on_*' event is finished. E.g.: to add a comment after a special trunk was opened.

Prefix 'pre_': Called before an '*on_*' event. These events affect positioning of the acting character or move him/her to the right position to execute the '*on_*' event.

And some more events not fitting into this pattern:

Some Main Events

- *'on_map_loaded'*: Nearly all initialization of a zone is done in this event.
- *'on_map_reentry'*: Things that need to be done on reentering a zone, like initiating schedules and FX.
- *'on_post_load'*: Like above, but executed when a game is loaded in this zone.
- *'on_timer'*: Event to react to local set timers.
- *'on_enter_region'*: This event is called by region triggers.

```
def on_enter_region(id, objectid, partyid ):
 if id == 'region_1':
       function_1()
```

```
def on_map_loaded():
 system.add_region(id = 'region_1', x1 = 419.90, y1 = 348.00,
            x2 = 420.00, y2 = 370.90)
 system.notify_region_entered(region= 'region_1',
            partyid = 'UISPIELER', permanent=False)
```

- *'on_approach'*: To connect dialogs and schedule dialogs these events need the following local:

```
def on_approach(target,operator):
 if not object_events.on_approach(target, operator):
       dialogs.on_approach(target, operator)
# and in dialogs
def on_approach( target, operator ):
 if schedules.Dialog(target, operator):
       return
```

- *'on_wait_for_commando'*: Needs to call the schedule executor:

```
def on_wait_for_commando(id):

 return schedules.Activity(id)
```

- *'can_examine_object'*: The local event '*on_examine*' is only called, if the event returns *True* (normally the local '*on_*' event would be called, even if only the global '*can_*' event was *True*).

Activating a Zone

Preparing the Travel Map

Read chapter 'Textures and UI Elements' about how to register a new zone icon for the travel map. Then add the position for the new map to the list:

```
# in the module system_py
zones_list = {
 "zone_1":   ("zones\\zone_1\\zone_1", [391.00, 504.00]),
 # ....
 "zone_X":   ("zones\\zone_X\\zone_X", [341.00, 554.00])
 # or
 "mod_X":    ("new_mod\\mod_X\\mod_X", [391.00, 504.00])
```

Even if the zone is designed as a mod, the zone's name needs to be unique (like *mod_1*).

The position set here is not the same as in the layout manager, but the one the loading bar is attached to (the travel points displayed while travelling).

Setting Location Name

Also, the location needs a name to be displayed on the travel map:

```
text.add_global_text(id='ZONE_NAME_ZONE_X',  text="""Zone X""")
text.add_global_text(id='ZONE_NAME_ZONE_X_EXTENDED',
              text="""Zone X - Example""")

text.add_global_text(id='ZONE_NAME_MOD_X',  text="""Tutorial Mod""")
text.add_global_text(id='ZONE_NAME_MOD_X_EXTENDED',
              text="""Tutorial Mod – Under Construction""")
```

Other Global Lists

Sky Color

Add the sky color of your new zone to the list in the module *skies*:

```
skies = {
 "zone_1.zone_1":             "yellow",
 "zone_X.zone_X":             "yellow",
 # or
 'new_mod.mod_X.mod_X'        'yellow',
```

Resting Place

The resting place is defined in the module *rest*:

```
def init_data():
 if not hasattr(data("rest"), "legal_rest_areas"):
       data("rest").legal_rest_areas = {
             "zone_1": [  ( 566.57, 504.53 ) ],
             'zone_X' : [(500,400)],
             # or
             'mod_X': [(500,400)],
```

Activating Respawn of Animals

Dead animals will only be replaced if allowed in the module *spawnanimals*. Make sure that animals in your zone use the usual data structure:

```
SPAWN_ANIMAL_ZONES = [
             'zone_1.zone_1',
             'zone_X.zone_X',
             # or
             'new_mod.mod_X.mod_X',
```

Enabling a New Zone

The command (*global_defines.*) *set_zone_enabled()* activates the new zone on the map for travelling:

```
set_zone_enabled(zone='zone_X', enabled=True)
# or
set_zone_enabled(zone='mod_X', enabled=True)
```

To use the new map as the starting map for the campaign set in *global_defines*:

```
ENTRY_ZONE = "zones\\zone_X\\zone_X"
# or
ENTRY_ZONE = "new_mod\\mod_X\\mod_X"
```

NPCs: 'A New Star is Born'

Every character in a game is an NPC (None Player Character). Even PCs (Player Characters) are only NPCs whose control is transferred to the player. The few differences between these two types will be pointed out in the last section.

Animals are not that different from NPCs; just look up their types in the basic *npc.py*.

Basic Creation

Take a look at the *npc.py* of the empty or tutorial zone:

```
def init_npcs():
 for npc in npcs.keys():
       system.create_character(
             id                = npc,
             gender            = npcs[npc][0],
             party       = npcs[npc][1],
             x                 = npcs[npc][2],
             y                 = npcs[npc][3],
             name        = npcs[npc][4],
             direction   = npcs[npc][5],
             resourceui  = npcs[npc][6])
       objects.set_attributes(npc,
             faction           = npcs[npc][7],
             level       = npcs[npc][8],
             model       = npcs[npc][9],
             gender            = npcs[npc][10],
             skin        = npcs[npc][11],
             clothes           = npcs[npc][12],
             face        = npcs[npc][13])
       character.update_appearance(npc)
       if not npcs[npc][14] == 0.00:
             character.scale(npc, npcs[npc][14])
       objects.set_active_state(npc, npcs[npc][15])
```

It uses the dictionary *npcs* to create a character and set all necessary attributes. The variables are:

- '*id*': The ID of the new NPC; must be unique.
- '*gender*': *Male* or *female*; can be changed later.
- '*party*': Sets the party; see below.
- '*x,y*': Position.
- '*name*': A string is a variable of the module *globaltext* holding a localized name; can be changed later.
- '*direction*': Initial direction the character faces.
- '*resourceui*': Empty string or name of picture for large dialogs; can be changed later.
- '*faction*': Used for setting HP and XP.
- '*level*': The level; used for calculating skills and attributes.
- '*model*': *Low* for NPCs, *high* for PCs and *child* for children; controls model complexity.
- '*skin*': Color of skin: *black*, *white*, *red*, *grey*.
- '*clothes*': Sets the body texture for *low* and *child* models; directory *textures\\characters\\human*.
- '*face*': ID of the face.

Party

Every NPC needs to be part of a party. Three main parties are always defined, but you can create additional ones for different NPC groups in a zone:

```
system.create_party(id="NPCS",type=PT_COMPUTER)
system.create_party(id="UISPIELER",type=PT_HUMAN)
system.create_party(id="ANIMALS",type=PT_COMPUTER)
```

NPCs can switch groups (e.g.: when hiring a mercenary):

```
system.switch_party( id, party)
```

Clothes

All character textures are placed in the directory *textures\\characters\\human*, and in the file *scripts\\character_py.py* you will find how to address them.

```
"ll_001": ("ll", "body",
       baseres+"low\\male\\ll\\low_body_ll_001_%(skin)s.png")
```

Faces

Same directory as clothes and again the *scripts\\character_py.py* translates ID into file.

```
"evil01": baseres+"low\\faces\\female_low_face_evil01.png"
```

Faction and Level

Like PCs all NPCs have HPs, skills and attributes to define their combat skills. These are set based on their faction and level. The tables are defined in the module *object_events:*

- *npc_level_table*: Defines skills and attributes for level.
- *npc_animals_le_table*: Defines base HP and HP per level.
- *kill_exp*: XP for killing an NPC of this faction and level.

Special Commands to Control NPCs

Active

Like all objects NPCs can be turned on or off with:

```
objects.set_active_state( ID,  True )# or False
```

Immortal

To keep quests active some NPCs need to be immortal (they cannot be attacked, wounded or killed).

```
objects.set_immortal_state( ID, True)# or False
```

Approachable

NPCs are by default approachable (the player can speak to them) and show this on mouse over (dialog icon). Deactivate this attribute if the NPC has no dialog at the moment or no dialog at all (combat NPCs).

```
character.set_approachable_state( ID, True) # or False
```

Daily_routine commands

See chapter 'Daily Schedule' for more information on this.

Foes

The attribute *foes* controls what kind of characters an NPC will attack. It holds a list of IDs or parties (e.g.: *UISPIELER*).

```
objects.set_attribute( ID, 'foes', ['UISPIELER'] )
```

Scale

The scale of characters can be changed, but stay within the usual frame, because animations and objects only fit for normal scale NPCs.

```
character.scale(ID, height )
```

Equipping NPCs

Items

NPCs only need items as loot or for special animations (e.g.: chopping wood). During an active schedule the objects needed are generally created and removed afterwards. Here some commands to create, equip and remove items:

```
objects.create_item_in_inventory( ID, ITEM_TYPE)
character.equip( ID, ITEM_ID) # or ITEM_TYPE
character.equip_direct( ID, ITEM_ID) # doesn't play animation
character.un_equip( ID, ITEM_ID)# or ITEM_TYPE
objects.remove_item_from_inventory( ID, ITEM_ID)#or ITEM_TYPE
```

The item list is: *scripts\\itemdata\items.py*.

Weapons

NPCs do not need ammunition. They will always load empty weapons back to a full clip.

An unarmed NPC will equip himself with a weapon found in his inventory when he starts an attack.

The fastest way to arm an NPC is:

```
character:equip_direct( ID, ITEM_TYPE,
       creation=True, # Create a new weapon
       direct_ammo_load=Ture )# load it without animation
```

Useful lists are *scripts\\itemdata\weapons.py* and *scripts\\itemdata\ammo.py*.

Player Characters

Only PCs or possible PCs should have the attribute *model = high.* All *high* models need an individual face, hair and clothing.

Face and Hair

The faces can be found in *textures\\characters\\human\\high\\faces* and the ID in *scripts\\character_py.py.*

The hair uses the same ID as the face.

Clothing

There are four slots for clothing: clothes, boots, gloves and helms. All these things must be objects in the inventory of the character. Use '*equip*' to dress the PC. The parts will only fit into one specific slot and replace the formerly equipped item. Clothing parts are items defined in *scripts\\itemdata\items.py*.

Voice_id

The attribute *voice_id* is used to select the speech sample for a mercenary. The ID is the voice ID plus the text ID (e.g: *mercenary_carmen_examine_item_found_1.ogg*).

This also switches AE speech: Leave blank for male AE and set to 'AEF' for female.

ID Card

The attribute *passport = True* activates the ID card for PCs.

All Talk: Scripting Dialogs

The Fall uses two main dialog types:

Small Dialog

Small dialogs appear in a box above the speaker's head and the game continues while playing this dialog. Long small dialogs should be played in cutscenes, because you can never know when the player moves away from the speaker. The game can play multiple small dialogs but most times it is better to allow only one at a time.

Large Dialog

A large dialog opens a dialog box consisting of a picture of the actual speaker and a text field. The game pauses while a large dialog is active and it may contain choices. All active small dialogs are cancelled when a large dialog starts, and of course only one large dialog can be active.

I will start the following sections by explaining how to set up the text in localizable IDs. Probably you will not localize a mod, but working with text IDs will be easier in complex dialogs.

After that I will show you the basic structure of branching (choices) dialogs and multiple entry points.

The third section will point out some tricks to force-fit complex dialogs into our code structure, and the last section explains how to remember spoken parts.

Registering Text IDs

You do not need to use text IDs but can just enter the dialog text in the text parameters of the dialog. But *The Fall* will write an exception every time a non-registered text ID is used (and a dialog line would be interpreted as an unregistered text ID). This exception can be deactivated with the command:

```
import debug
debug.cheat("disable_textid_exception")
# this condition will not be saved, so it needs to be set at every start
```

All texts are initialized in modules named after their language (e.g: *english*). These modules exist for every zone (like *zone_1.zonetext.english*) and there is also one for global texts (*globaltext.english.english*). A normal text entry for a zone would be:

```
## Dialog 001: The player approaches the president's house unauthorized
add_zone_text(     id='D_001_T_001',
            text="""Stop! That's close enough. Now beat it! """,
            speaker="""Wache_1_Z1""",
            audiofile="",
            comment="""""""
            )
```

The text entry contains the ID of this dialog and the text string. The other parameters *speaker*, *audiofile* and *comment* are not used in the game and are only for creating the speech files.

All IDs are in capital letters for script use but the file names for speech are in lower case.

The text strings are marked with three " to allow most sign combinations in the text.

Global texts are nearly identical, but use the command *add_global_text* instead.

Dialog Choices and Branches

Besides the command *dialog.add_text()* described in the chapter 'Her First Words', large dialogs contain choices and are non-linear. Take a look at *dialog.add_choice()* in the API:

```
dialog.add_choice(text, follower, callback=None, order=0,
       auto_deactivation='normal')
```

A set of dialog choices needs to be declared successively and the IDs need to be of the form: 'D_xxx_C_yyy_z' (x = dialog nr., y = choice set nr., z choice nr.). Dialog choices are normally listed in order of appearance, but they are changeable with the parameter *order*.

In general, the dialog texts are played in the order they are coded, but this can be changed with the parameter *follower*. This parameter is needed in an *add_choice* and useful in *add_text* to build complex dialog.

```
dialog.add_choice('D_001_C_001_1', follower = 'D_001_T_001')
dialog.add_choice('D_001_C_001_2', follower = 'D_001_T_003')
dialog.add_choice('D_001_C_001_3', follower = 'ENDDIALOG')
dialog.add_text('D_001_T_001', 'NPC_1')
dialog.add_text('D_001_T_002', 'ALTER_EGO', follower = 'D_001_C_001')
dialog.add_text('D_001_T_003', 'NPC_1', follower = 'D_001_C_001')
```

Line *'D_001_T_002'* follows line *'D_001_T_001'* in code and dialog so the parameter *follower* is not needed here. The follower *'D_001_C_001'* sends the dialog back to the choice set.

Auto-Deactivation and New Choices

The command *add_choice* has another parameter: *auto_deactivation*. If you set it to '*normal*'(default), this choice is deactivated after first use, so when the dialog returns to this choice set, this specific choice is no longer displayed. To make a choice always available set the parameter *auto_deactivation* to '*None*'.

Now, let's say we want to deactivate a used choice for every restart of this dialog. This can be done by setting the parameter *auto_deactivation* to '*permament*', but it needs an ID to identify the dialog:

```
dialog.init_dialog(type=DIALOGLARGE)
dialog.set_dialog_identifier('D_001')
dialog.add_choice('D_001_C_001_1', 'NPC_1', 'D_001_T_001',
             auto_deactivation = 'permanent')
dialog.add_choice('D_001_C_001_2', 'ALTER_EGO', follower = 'D_001_T_003')
dialog.add_choice('D_001_C_001_3', 'NPC_1', follower = 'ENDDIALOG')
....
```

Next, we want choice two to be only available after choice one is played. Dialog lines can be added after a dialog is finished and started:

```
dialog.add_choice('D_001_C_001_1', follower = 'D_001_T_001',
       auto_deactivation = 'permanent',
       callback = Lambda("dialog.add_choice('D_001_C_001_2',
       follower = 'D_001_T_003')"))
dialog.add_choice('D_001_C_001_3', follower = 'ENDDIALOG')
```

Removing an Additional Dialog Choice from a Running Dialog

First try to kick your designer until he makes a better dialog, but if he is stronger ...

Dialog lines cannot be removed from a running dialog, so we have to trick here. Build a dialog with

an identifier and use *auto_deactivation* to remember settings for played choices. In the line that deactivates another choice close the dialog with '*follower* = "*ENDDIALOG*"' and restart it with the removed choice:

```
def dlg_001(restart = False):
 dialog.init_dialog(type=DIALOGLARGE)
 dialog.set_dialog_identifier('D_001')
 dialog.add_choice('D_001_C_001_1', follower = 'D_001_T_001',
            auto_deactivation = 'none')
 if not restart:
       dialog.add_choice('D_001_C_001_2',  'D_001_T_003',
            auto_deactivation = 'permanent')
 dialog.add_choice('D_001_C_001_3', follower = 'ENDDIALOG',
            auto_deactivation = 'none')
 dialog.add_text('D_001_T_001', 'NPC_1')
 dialog.add_text('D_001_T_002', 'ALTER_EGO', 'ENDDIALOG',
            callback = Lambda("dlg_001(True)"))
            # finish and call dialog again
 dialog.add_text('D_001_T_003', 'NPC_1', follower = 'D_001_C_001')
```

Through restart of a dialog you can build any form of dialog structure, but a dialog designed to work without this trick is always better.

Alternative Starting Line

A usual dialog starts with an introduction and is followed by some choice sets. Most of these dialogs need a second short introduction if the player returns:

'D_001_T_001': "Hi Civ, who are you?"

'D_001_T_002': "I'm Max, how can I help you?"

....

'D_001_T_010': "Hi Max, I have some more questions."

And as code:

```
dialog.init_dialog(type=DIALOGLARGE)
dialog.set_dialog_identifier('D_001')
dialog.add_text('D_001_T_001', 'ALTER_EGO')
dialog.add_text('D_001_T_002', 'NPC_1', follower = 'D_001_C_001' )
dialog.add_text('D_001_T_010', 'ALTER_EGO', follower = 'D_001_C_001')
....
if second_start:
      dialog.jump_to('D_001_T_010', 'D_001')
```

The command *jump_to* can also be used in a callback to override the follower.

Daily Schedule: 'Everyone Has to Make a Living'

Everyday behavior of NPCs is defined in their schedule. A schedule is a dictionary of persons, times and actions. The modules *daily_routines* and *npc_activities* hold all code and utility functions to make the schedules work. Additional local functions and the schedules for all NPCs in a zone are defined in the module *schedules* of the current zone.

You will now learn how a schedule is structured, how to deactivate or pass the schedule and some basic schedule sets.

The Dictionary Schedule

Each schedule is defined in the dictionary schedules in the module *schedules* in the current zone. It consists of 5 layers:

```
ID: # ID of the NPC
      (#list of schedules for this ID
            {# dictionary of timeframes
            time:
                  {# dictionary of keywords in timeframe
                  keyword:
                        [..] #list of functions
      }})
```

ID of NPC

The first layer is a dictionary with the ID of the NPC. The value is either a list of schedules or a single schedule (go to third layer: dictionary of timeframes).

List of Schedules

If a list of schedules is found, the first schedule in which the function marked by the keyword *S_CONDITION* is *True* is used. If no keyword *S_CONDITION* is defined, it is considered as *True*.

The keyword *S_CONDITION* can be found in the third layer dictionary (see below).

Dictionary of Timeframes

The third layer uses the starting time of timeframes as dictionary keywords. Also, the keyword *S_CONDITION* can define a condition for this schedule in this level. A second keyword accepted here is *S_SECONDARY_DIALOG*. This may contain a dialog ID or a function playing a dialog. The secondary dialog is used when the dialog defined for a timeframe was already played or is not defined.

Dictionary in a Timeframe

This dictionary can accept a number of keywords:

S_START: Functions executed at the start of this timeframe.

S_LOOP: Functions executed again and again in this timeframe (after *S_START*).

S_DIALOG: A dialog ID or function/code object playing a dialog. NPC ID is passed to the function.

S_LOCATION: Defines a number for this timeframe.

S_NOSTOP: List of location (frame) numbers; omit *S_START* for listed numbers.

List of Functions

All elements in this list need to be executable code objects (Lambda or lambda). The module *npc_activities* holds a lot of functions returning the right code objects for this purpose.

```
"14:00:00":{S_START:[npc_activities.goto(300,300)]},
# this function returns and inserts
"14:00:00":{S_START:[lambda char: objects.move_to(char, 300, 300)]}

# "14:00:00":{S_START:[objects.move_to(char, 300, 300)]} ! BUG ERROR !
```

Commands to Control Schedule Behavior

The General Work Flow of NPC Actions

Every NPC has a command stack and executes animations from this stack. When the stack is empty the function *on_wait_for_commando* is called to get a new animation. This function writes a new command in the stack or returns *False*, commanding the engine to play a waiting animation.

The function *on_wait_for_commando* delegates work to *npc_activities. npc_activity()*, which checks for schedules and executes the next schedule sequence.

'allow/prohibit activity'

Getting a new schedule action can be switched on and off with the commands *daily_routine.allow_activity()* and *daily_routine.prohibit_activity()*. When issuing commands to an NPC (e.g. for a cutscene) you should switch off schedules during these commands or get most unusual results.

'on wait for commandos'

The function *on_wait_for_commandos()* sometimes holds special code for some NPCs that would block general behavior. Be sure to check this function when changing behavior for NPCs.

Using this function for special control is always a tricky business. The code will be executed for every character (NPC and PC) whenever there is no following action defined, which mounts up to a lot of situations. In most cases it will be easier to write additional schedules for this special situation and switch them on/off via conditions. This way you can make sure that the code of two different layers will not collide.

Using Schedules

Basic Timeframe

In a basic timeframe an NPC moves to a new position, maybe executes a starting animation (both in *S_START*) and then a loop animation until the time is up. The timeframe is switched to the next frame and in between the *npc_activities.finish_activity* issues finishing animations if needed.

```
"14:00:00":{S_START:[npc_activities.goto(300,300),
```

```
            npc_activities.start_chop_wood()],
       S_LOOP:[npc_activities.chop_wood()]}
#there is no finish_chop_wood because this animation is issued
automatically when finishing a timeframe
```

Continue Loops

One never knows at what time a schedule for an NPC will start. The time can even jump, by revisiting a zone or taking a rest. Therefore, every timeframe should have the keyword *S_START* to make sure that the NPC moves to the right position. Take a look at this:

```
"14:00:00":{S_START:[goto_object('bench_1'),
            sit_down_on_bench()],
       S_LOOP:[sit_on_bench()]},
"14:30:00":{S_START:[goto_object('bench_1'),
            sit_down_on_bench()],
       S_LOOP:[read_book_on_bench()]}
```

Using this schedule would make the NPC stand up at 14:30 and then sit down again on the bench. It would be better to have the NPC sitting on the bench uninterrupted. He should only take out a book to read.

That is what the keyword *S_NOSTOP* does:

```
"14:00:00":{S_START:[goto_object('bench_1'),
            sit_down_on_bench()],
       S_LOOP:[sit_on_bench()],
       S_LOCATION:1},
"14:30:00":{S_START:[goto_object('bench_1'),
            sit_down_on_bench()],
       S_LOOP:[read_book_on_bench()],
       S_NOSTOP:[1]}
```

In the first frame a location number is set and the second frame omits the functions defined in *S_START* when the previous timeframe had the number *[1]*.

Start/Loop Sets and Intelligent Functions

The functions defined in the module *npc_activities* are built around two basic ideas. They are divided into a start and a loop function:

```
       S_START: [npc_activities.start_chop_wood()]
       S_LOOP: [npc_activities.chop_wood()]
```

Or as a single function deciding what animation to play:

```
def make_push_up():
 return start_and_loop( "CM_NPC_STEHEN_LIEGESTUETZE_START",
                   "CM_NPC_STEHEN_LIEGESTUETZE_LOOP")
```

However, the character still needs to be moved to the starting position before using the animating functions.

Dialogs Defined in Schedules

Basic Time-dependant Dialogs

To define a dialog for a whole schedule use the keyword *S_SECONDARY_DIALOG*. This secondary dialog is played when the dialog for the timeframe was already played or is not defined.

The dialog for the timeframe is assigned via the keyword *S_DIALOG* in the forth layer.

```
NPC_ID:{
 S_SECONDARY_DIALOG: 'NPC_ID_001',
 '00:00:00': [S_START:[],
              S_LOOP:[],
              S_DIALOG: 'NPC_ID_001']}
```

Gender-specific Dialogs

The command *npc_activities.gender_switch()* can be used to allocate different dialogs for gender:

```
S_DIALOG :gender_switch('NPC_MARIA_T_005F', 'NPC_MARIA_T_005')
```

Long Dialogs

Sometimes it is necessary to play more than one dialog line in a timeframe or the AE needs to comment on an NPC. Therefore the command *npc_activities.longdialog()* can be used in the schedule to play through a list of dialog lines:

```
S_DIALOG :(long_dialog([['ALTER_EGO', 'D_186_T_001']]))
S_DIALOG :(long_dialog([['ALTER_EGO', 'D_001_T_001'],['NPC_1',
'D_001_T_002']]))
```

Specials

Items: 'I Need Guns ...'

Creating an item is a good step up from normal world objects. It takes three different steps and it will be easier to try each of them separately first. You need to be familiar with chapter 'Objects', be able to add an icon as described in chapter 'Textures and UI Elements', and you need some experience with 'Scripting' to define useful parameters. Maybe your item even needs 'Sounds'?

Creating the 3D Object

This is basically the same as described in the chapter 'Objects'. Just remember that ingame size is controlled in Max, so set unit size to 'centimeter' and scale accordingly. And better read the following three subchapters, because you will need them.

Effect Dummy

If your item is going to be a firearm, it needs an effect dummy, that is a node defining position and direction for graphic effects. The dummy *'Dummy_muzzleflash'* is reserved for the fire effect at a weapon's muzzle. This dummy node is needed and must be linked to the mesh node for the equipable object.

You can define additional dummy nodes for your own scripting. And even some world objects have dummies. They are used to position NPCs for interaction with this object (bench, houses). However, there should be enough examples of scripted objects in *The Fall*'s code.

Two Objects in One Mesh

The Fall needs the objects in two states: 'hold in hands' and 'lying on the ground'. So, copy your finished object and define two mesh nodes. One for the equipped item and one for the item lying on the ground.

Positioning the Object

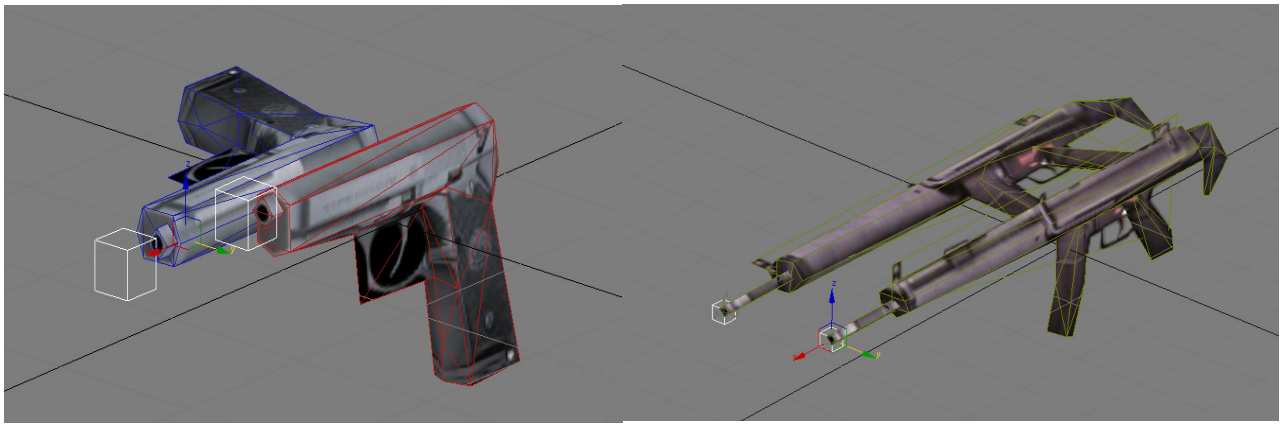
An object in *The Fall* is placed according to its point of origin. Even its orientation is taken as defined in 3D creation. That is why an equipable item consists of two mesh nodes. The ground object is easy to place: Its origin is equivalent to the point the object is linked to in the game world.

If the object should be held in hand, it gets a little more complicated. The character models have two hand dummies; the right hand holds one-hand-items and the left hand holds the both-hand-items.

For one-hand-items place the pivot point as if it were the holding hand on the grip. And for both-hand-items rearrange it so that the pivot point is where the left hand holds the item. For a rifle that would be the shaft. Then rotate the item or direction of the pivot point accordingly.

You can adjust the holding position with script later but not the alignment.

Take a look at the following pictures, which show examples for the placement of the mesh nodes:



You can find an example file for 3DS Max in this package: *m_16.max*

The texture for this item is: *'the_fall_main_folder'\textures\additionals\additionalweaponsa.png*

Adding the Icon

Nothing extraordinary here. The texture with room for items is *'maindirectory'\gui\gui_mission\items_26.png*. Then use GUIRsesourceData to add this item to *'maindirectory'\gui\gui_mission\fall_items.gui*. This time the identifier is used to address the icon ingame (see scripting below). Still the GUI ID should be unique.

The picture for the item must be 65X32 in size.

Scripting

'objectdata\items'

The first data set goes into *'maindirectory'\scripts\objectdata\items.py*.

```
# create a new data set
add_3d_item_data(typeid='RES3D_ITEMX')

# define object resource
objects.set_attribute(object='RES3D_ITEMX', attribute="diff3d_file",
            value="objects\\additionals\\thefall\\itemX.diff3d")

# the two mesh nodes: first for ground item, second for hand item
objects.set_attribute(object='RES3D_ITEMX', attribute="states",
                  value=['ITEMX_00','ITEMX_01'])
# unequipable items only need one mesh node for ground

# adjustments to hand-held in centimeters
objects.set_attribute(object='RES3D_ITEMX', attribute="offset",value=[
            ['male',    'high',     'node_rifle',     [-1,0,-0.5]],
            ['female',  'high',     'node_rifle',     [1,1,2]],
            ['male',    'low',      'node_rifle',     [-1,0,-0.5]],
            ['female',  'low',      'node_rifle',     [1,1,2]]])
```

The offset parameter defaults to zero for all possible models (*male_low*, *male_high*, *female_low*, *female_high*, *child_low*) and is not needed. Every entry needs the following elements:

- Model type (male, female, child)
- Model complexity (high, low)

- Connected node (*node_pistol*, *node_rifle*)
- Adjustment vector (x,y,z) in centimeters

'itemdata'

Three files define items: *weapons.py*, *items.py*, *ammo.py*. These files are in: *'maindirectory'\scripts\itemdata*.

The parameters for all item types are:

```
# create the data set: create_weapon_type, create_ammo_type,
create_item_type
create_weapon_type(typeid='SET_ITEMX')

# define the text resources
objects.set_attribute(object='SET_ITEMX', attribute="name",
value=globaltext.SET_ITEMX_NAME)
objects.set_attribute(object='SET_ITEMX', attribute="hint",
value=globaltext.SET_ITEMX_HINT)

# link the icon resource (name is defined in fall_items.gui)
objects.set_attribute(object='SET_ITEMX', attribute="resourceui",
                                value='RES_ITEM65X32_WEAPON_ITEMX')

# link the 3D mesh resource defined in objectdata\items.py
objects.set_attribute(object='SET_ITEMX', attribute="resource3d",
       value='RES3D_ITEMX')

# trading value of item
objects.set_attribute(object='SET_ITEMX', attribute="value", value=300.0)
# weight of item
objects.set_attribute(object='SET_ITEMX', attribute="weight", value=8.0)
# maximal stack size
objects.set_attribute(object='SET_ITEMX', attribute="stacking", value=1)
# forbids random placement in trader
objects.set_attribute(object='SET_ITEMX', attribute="not_tradable",
       value=True)
```

Equipable items need the following:

```
# occupied_slots can be: left_hand, right_hand, throw_weapon, helm,
# gloves, boots, clothes, vision
objects.set_attribute(object='SET_ITEMX', attribute="occupied_slots",
       value=["left_hand","right_hand"])
```

Some weapon parameters:

```
# Firing range
objects.set_attribute(object='SET_ITEMX',
       attribute="minimale_feuerreichweite", value=0)
objects.set_attribute(object='SET_ITEMX',
       attribute="maximale_feuerreichweite", value=60)
# Shots per salvo
objects.set_attribute(object='SET_ITEMX',
       attribute="salvenlaenge", value=1)
# Rounds per minute
objects.set_attribute(object='SET_ITEMX',
       attribute="feuergeschwindigkeit", value=0.00)
# Accuracy
objects.set_attribute(object='SET_ITEMX',
       attribute="genauigkeit", value=3.00)
```

```
# Loadable ammo types
objects.set_attribute(object='SET_ITEMX', attribute="munitionsarten",
 value=['SET_AMMOPACK_7_62NATO_MM','SET_AMMOPACK_7_62NATO_MM_SABOT])
# Fire animation type (see module global_defines)
objects.set_attribute(object='SET_ITEMX',
       attribute="fire_animation", value=FPA_SNIPER_SINGLE)
# Weapon type (mostly to select skill to use)
objects.set_attribute(object='SET_ITEMX',
       attribute="weapon_type", value="sniper")
# Rounds per clip
objects.set_attribute(object='SET_ITEMX',
       attribute="magazingroesse", value=10.0)
# Range at which the shot can be heard
objects.set_attribute(object='SET_ITEMX',
       attribute="hoerweite", value=40.0)
# Time to aim weapon
objects.set_attribute(object='SET_ITEMX',
       attribute="schusspausenzeit",      value=4.0)
# Accuracy against moving targets
objects.set_attribute(object='SET_ITEMX',
       attribute="genauigkeit_bewegte_ziele",   value=0.7)
# Probability of weapon jam
objects.set_attribute(object='SET_ITEMX',
       attribute="blockadewahrscheinlichkeit",  value=0.0)
```

The weapon parameter '*effects*':

```
objects.set_attribute(object='SET_ITEMX', attribute="effects", value={

 # defines weapon sound; directory : sounds\weapons
       "sound_shot"     : "shot_ITEMX",
       "sound_unload"   : "rifle_discharge",
       "sound_gunjam"   : "rifle_gunjam",
       "sound_reload"   : "rifle_charge",

 # defines the muzzle flash; compare weapons for examples
       "flash_type"     : 1,
       "flash_size"     : 1.0,
       "stream_type"    : 1,
       "stream_size"    : 1.0,
       "fade_out"  : 50})
```

Define item components:

```
objects.set_attribute(object='SET_HK_USP_45_W_SILENCER',
attribute="item_combination",  value={
 # Technical skills needed to build or disassemble
       "difficulty_assemble"   :     COMBO_EVERYBODY,
       "difficulty_disassemble":     COMBO_EVERYBODY,

 # Functions used for dis-/assemble
       "assemble_function": "assemble_weapon_silencer_laserpointer",
       "disassemble_function":
             "disassemble_weapon_silencer_laserpointer",

 # Component list
       "combination_list": [['SET_HK_USP_45','SET_SILENCER']]}
```

Ammo parameters:

```
# Area of effect; shotguns, explosives
objects.set_attribute(object='SET_AMMOPACK_7_65_MM',
       attribute="wirkungsbereich", value=0.00)
```

```
# Damage range
objects.set_attribute(object='SET_AMMOPACK_7_65_MM',
       attribute="schaden_organisch_min", value=1)
objects.set_attribute(object='SET_AMMOPACK_7_65_MM',
       attribute="schaden_organisch_max", value=2)
# Rounds per pack
objects.set_attribute(object='SET_AMMOPACK_7_65_MM',
       attribute="schuss", value=30)
# ballistic curve
objects.set_attribute(object='SET_AMMOPACK_7_65_MM',
       attribute="schussbahn", value=LINEAR)

# Linking text resource for caliber
objects.set_attribute(object='SET_AMMOPACK_7_65_MM', attribute="kaliber",
                        value=globaltext.CALIBER_7_65_MM_HINT)

# Factors changing weapon parameters
objects.set_attribute(object='SET_AMMOPACK_7_65_MM',
       attribute="hit_chance_factor", value=1.0)
objects.set_attribute(object='SET_AMMOPACK_7_65_MM',
       attribute="fire_range_factor", value=1.0)
```

Linking Special Scripts

Weapon Events

Just a list of possible events to link special functions for a weapon:

- *object_events.on_(un)equip*: When item changes something while being equipped.
- *object_events.get_hit_chance_factor*: The weapon might perform better while being in a prone position.
- *object_events.get_damage_factor*: Your weapon might use an ammo type that is more effective.
- *weapon_events.on_weapon_fire_single(multi)*: Add more than a flash when firing your weapon.

Special Functions for Using Items

You can link a special function for *on_use* to an item by just adding a function named like the *item_type* to the module *global_items*:

```
def ITEMX(item_id):
 commands
```

For this the item needs to be registered in *global_items.global_items_list*.

Creating a Mod Package

Most likely you will take your first steps in scripting by changing existing code. Scripting here and there, until your new script is worth sharing with others. Just copying the scripts and zones directory would be the easy way, but then users could play only one mod at a time. Not to mention the file size. Also, most users do not like to change their installed files.

Therefore, *The Fall* can access special mod directories and integrate everything that they contain into the game. Files found in the mod directory are used instead of existing files and modules. That is good for new files like graphics or a new zone, but bad for global modules.

Here the module *startup* comes in. Placed in the mod directory it is executed when launching *The Fall* with an active mod. You can use this file to redirect and insert functions into global script modules. This way you can leave the global scripts nearly untouched.

The Structure

Mods are placed in the directory *mods* in the *The Fall*'s main directory. All directories placed in there need to be Python packages. That means they need a file named '*__init__.py*'. This file is usually empty, but its existence tells Python to treat this directory as a package.

In addition to that the mod package needs the file *startup.py*. This file contains the initialization and is covered in the next chapter.

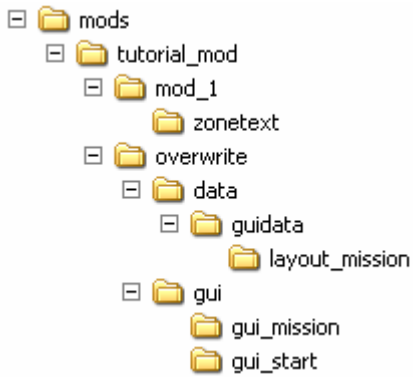
Also there should be a directory named '*overwrite*'. Every file placed in this directory will replace its equivalent game file. You need to **recreate** the game directory structure to add the files to the program structure at the right place.

Besides this, you can place additional files, modules and scripts in this directory to use in your mod. Now take a look at the tutorial mod:

In the mod's directory is the base directory (package) *tutorial_mod* of the tutorial mod.

This directory contains the files *__init__.py* and *startup.py* and the directories *overwrite* and *mod_1*.

The directory *mod_1* is a full zone package and the directory *overwrite* holds the files needed to integrate a new icon for the travel map. The directory structure is a copy of the game directory structure but only holds the necessary files.



Remember, every file in the main directory of your mod is used only when your scripts address it. And everything in the overwrite directory replaces existing game files, so be careful what to place in there, because only one mod at a time can replace a specific file.

Problems could arise with replaced script files. Replacing script files can be avoided by a clever use of the module *startup* (see below).

Convert PNG to DDS

Textures placed in the overwrite directory are only used in the game when they are in the format *DDS*. If you want to use the automatic converter *png2dds*, you just need to place your new texture in the normal game directory (the right subdirectory for that type of texture of course) and start *The Fall* until the automatic converter has created a *DDS* version of your texture. Now move this to your mod directory and return the game directory to its original state.

The Module *Startup*

Addressing Mod Files

The main directory of a new mod is a package and therefore every call to a module or function in this mod starts with '*mod_name*'. Here some naming examples borrowed from chaper ['Base Name, Directory and Module']:

```
'mod_1'
# basic zone name; used in global_defines.set_zone_enabled('mod_1', True)

'tutorial_mod.mod_1.mod_1'
# internal identifier for data(), and most lists
data('tutorial_mod.mod_1.mod_1').variable = True

# addressing a zone function from console
# base module / main script module / function
tutorial_mod.mod_1.mod_1.function_1()
```

The path for files placed in the mod directory also starts with this directory:

```
# adds coordinates for the travel map
system.zones_list['mod_1'] = ("tutorial_mod\\mod_1\\mod_1",
            [505.00, 477.00])
```

But remember that files placed in the overwrite directory are accessed as if they were in the original game directory. You could now use the overwrite function to integrate every mod file into the game directory structure, but that would make it more difficult to distinguish between mod and game files. So use overwrite only for replaced files and place new files outside the overwrite directory.

Startup: Place Statements Only in Functions

In Python a module is executed when imported. Global variables are initialized and functions declared. A statement which is not indented (not part of a function) in this file would also be executed while importing that module. That means that such a statement might be executed even if the mod is turned off. So do not use base level statements in the module *startup*.

```
# only new variable declarations: do not change original game variables
# here
mod_variable_1 = 'test'

# base level statement: Do not use!
print 'oh, that's wrong'

#only functions called by the function mod_init()
def mod_init():
 function_1()
```

The Function mod_init

This function is called when starting *The Fall* with an active mod and therefore the event for all initializations. Every change done to *The Fall*'s module and function structure should be started here. Let's see which tools Python offers for this.

Python can change the pointer to a function by a simple assignment:

```
def function():
 example_code()

system.function = function
# Function without brackets gives the pointer to that function
```

Now we use this to insert our new commands for an event in the global event:

```
# global variable for pointer to global event
global_event = None

def example_event():
 # inserted code
 example_code()

 # not really needed, because the value of this variable is not
 # changed
 global global_event

 # call old global event
 global_event()

def mod_init():
 # remember pointer to global event
 global global_event
 global_event = object_events.example_event

 # insert new event pointer in global event name
 object_events.example_event = example_event
```

And this is the right way to do it. Do not change the global function or overwrite it entirely! Always call the original global event, because that way the original code and other mods can work together with your mod (as long as the code you insert is compatible).

If you really need to pass existing global script, then make sure that you only omit the global call for those event calls that would not work together.

Or call your new global event only in your zone. Let's say you created a new zone and this zone cannot work together with the global *on_object_killed*. The local event is always called first (if it exists) and has to call the global event. Now insert a call to your global event instead of the call to the original global event. That way all other zones can still use the original global script, because it is not changed.