

AdonHELL
Engine Architecture

Kai Sterker

27.10.2004

ADONTHELL
ENGINE ARCHITECTURE

This document is part of the Adonhell Developer's documentation
<http://adonhell.linuxgames.com>

© 2004 Kai Sterker & The Adonhell Team
adonhell-project@linuxgames.com

Created with L^AT_EX_ε

Contents

1	Introduction	2
1.1	Abstract	2
1.2	Obtaining and Distributing	2
2	Overview	3
2.1	Design Principles	3
2.2	Architecture	4
2.3	Python on board	6
3	Main Module	7
3.1	Using the Engine from C++	7
3.2	Using the Engine from Python	7
3.3	Engine Startup	8
4	Base Module	9
4.1	Machine-Independence	9
4.2	Data Persistence	9
4.3	Configuration Files	10
4.4	Game Directories	11
5	Event Module	12
5.1	Event Handling	12
5.2	Date and Time	12
6	Python Module	13
6.1	SWIG support	13
6.2	Python Scripts and Methods	13
7	Rpg Module	14
7.1	Quests and Logs	14
7.2	Items and Inventories	15
7.3	Characters and Equipment	16
8	Map module	17
8.1	Maps and Views	17

1 Introduction

1.1 Abstract

This document is a supplement to the Adonhell API documentation. It is intended for new programmers and everyone else interested into the general concepts and design of the Adonhell game engine.

It is not meant to be a complete reference, but it should give the reader a good overview of the codebase and ideas behind various implementation details. It won't explain the code itself; that's what the API docs are meant for, but it might help to understand why certain things are the way they are.

In brief, this document attempts to provide a guided tour through select parts of Adonhell's source code, grouped by the different modules the engine is composed of. No more, no less.

1.2 Obtaining and Distributing

The Adonhell Architecture Documentation can be redistributed freely. The most recent version can be obtained from the Adonhell website¹. The \LaTeX sources are available via anonymous CVS. To retrieve them, issue the following commands:

```
export CVS_RSH=ssh
cvs -z3 -d:ext:anoncvs@savannah.gnu.org:/cvsroot/adonhell co -P doc
```

The first time you do this, you will be prompted by SSH to authenticate the server's key fingerprint. Answer with 'yes'. To update your copy of the code, run

```
cvs -z3 -update -dP
```

within the documentation directory from time to time.

¹<http://adonhell.linuxgames.com/download/documentation.shtml>

2 Overview

Before describing the different parts and layers of the engine in detail, lets have a look at the big picture.

2.1 Design Principles

As Adonthell's development progressed, a few fundamental design decisions have been made. These principles have to be taken into account, whenever a part of the engine is updated or new parts are added. So please read on carefully before you start coding; it may save you much trouble.

1. Code and data have to be strictly separated

This is probably the most important rule of all. The engine is not intended for a single RPG, but should instead be able to drive multiple games. For that reason, the code must not contain any game data. Only functionality needed for a RPG may be provided by the engine, not the games themselves.

2. The engine should be as flexible as possible

Creators of games should have the most possible freedom in designing their game. That is, the engine shall help people making their own RPGs without limiting their creativity. The game should control the engine, not the other way round. That means that the engine is no interpreter of game data, but rather a library the game may use for common and repetitive tasks.

3. The engine should be easily extendable

To allow people to make quite different types of games, with different rules, different items, different AI or just a different user interface, the engine needs to be customisable. Therefore, not only game data, but also most of the game mechanics have to be separated from the core engine.

4. The engine should be portable

Adonthell shall run on as many different operating systems and hardware architectures as possible. Right now the engine supports BeOS, BSD, Linux, Mac OS X, Solaris and Windows on hardware such as arm, alpha, ia64, ppc, sparc, x86 and possibly others. This has to be taken into account when writing code or utilising third-party libraries.

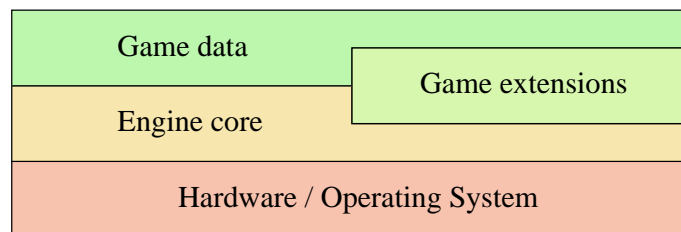


Figure 1: Adonthell engine and surrounding layers

When considering all these four points, we can draw a rough picture of the engine and its surrounding layers: the **engine core** is the only component that has direct access to the **operating system** and underlying hardware. It is written in C++ and provides common functionalities. **Game specific extensions** – written in Python – make use of the interface provided by the engine core to implement functionality specific to a game. Avoiding direct access to the OS ensures that games will run on all platforms supported by the Adonthell engine.

Both game engine and game extensions may access **game data**: maps, graphics, dialogues and so on.

2.2 Architecture

The engine core isn't a monolithic bloc, but split into a number of modules, each with its own namespace and each compiled into its own library. For each engine module, a corresponding extension module exists to allow access from Python.

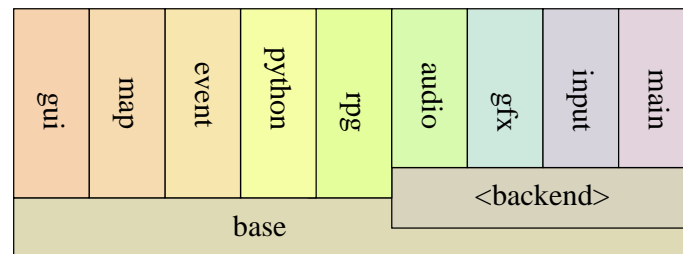


Figure 2: Adonthell engine architecture

- **Base**

The base module contains low level functionality used by most, if not all, the other modules. This includes I/O routines, i18n support, configuration file handling, a timer and more.

- **Backend**

A number of modules are accompanied by a backend library, providing the glue between the engine core and third-party code responsible for capturing input and handling audio and video output. Currently, SDL is the only backend supported, but support for other libraries may be added at need.

This backend mechanism serves two purposes: first, it should allow to use different, more suitable backends – like QT or GTK+ – for our editors while still having full access to the functionality provided by the engine. Second, it limits the dependency upon a single media library. Should SDL 1.2 become obsolete, a port to a different backend would be fairly simple.

- **Main**

The main module provides the entry point for applications using the Adonthell engine, both Python scripts or C++ programs. It takes care of initializing other modules as requested, parses commandline arguments and a generic configuration file. It then passes control to a user supplied callback (Python) respectively main method (C++).

- **Python**

The python module provides the C++ < -- > Python bindings. For one, it is a thin wrapper around the Python C API, allowing access to python scripts, classes and methods from within the engine. At the same time, it provides the functionality for accessing C++ objects from within Python scripts.

- **Gfx**

The gfx module provides graphic primitives, like surfaces to draw on and image loading/saving abilities. Any video output is routed through this module, meaning all other parts of the library that render something on the screen have no dependencies to the underlying graphic library.

- **Input**

The input module keeps track of keyboard, mouse and gamepad input. It also provides a configurable, virtual control device. As with gfx, all user interaction with the engine is routed through this module.

- **Audio**

The audio module allows playback of sound effects and background music and provides other sound related functionality to parts of the engine that need them. Primary audio format is Ogg Vorbis.

- **Event**

A great deal of gameplay is accomplished through Python methods triggered by various events the engine generates. The event module provides the core event system based on a publish/subscribe mechanism. It also implements a date class to keep track of time in the game world and time events based upon that class.

- **Rpg**

The rpg module provides core functionality required by every RPG: characters, quests, items, inventories and so forth. However, it does not contain complete implementations where this would limit flexibility, only interfaces. Those have to be extended by Python scripts in order to implement the rules of a specific game².

- **Gui**

The gui module provides a themeable widget library, suitable for modelling an unobtrusive game interface. It supports true type fonts through Freetype 2.

- **Map**

Low and high level classes for displaying the state of the gameworld to users.

²More on this topic can be found in the Adonhell Scripting Guide

2.3 Python on board

Python is an easy to learn, interpreted, object oriented programming language – and as such the perfect means to accomplish some of the design principles mentioned earlier.

Adonthell's integration with Python works in two ways. In one direction, Adonthell acts as a Python interpreter. Therefore, it can delegate parts of the work to user-supplied scripts, allowing for more flexibility. This can be compared to a plug-in mechanism. In the other direction, all of Adonthell's modules can be accessed from within Python scripts. If a C++ class *bar* exists in namespace *foo*, a Python script can import it and manipulate it as if it was accessed from C++ code. That way, objects created in Python scripts can be passed to C++ methods or vice versa, with practically now limitations.

This is accomplished by SWIG, which provides the necessary glue between C++ and Python. Given a C++ class, SWIG will generate a Python module with the same interface. Internally, all calls to methods of the Python object are routed to the underlying C++ implementation. The gory details of wrapping and unwrapping C++ objects in order to pass them from C++ to Python and back are covered by SWIG. The `pass_instance` and `retrieve_instance` template functions supplied by our python module further simplify this task. They are described in the API.

As a result, the engine can be seen as a library providing bits and pieces of RPG related functionality that can be used to write the actual game in Python. The benefit of that concept is that quite different games can be created without having to touch the underlying engine. One day, a community might form around the engine, providing players with mods and completely new games.

3 Main Module

Although the various Adonhell modules can be used from custom programs without utilizing the main module, this is highly discouraged. Not only provides the main module convenient initialization methods, it also contains code to prepare the environment for modules like gfx and input on different operating systems.

3.1 Using the Engine from C++

The main module provides a main function (hence the name), so your program will not need one when linking to the main module. Instead, it must provide a special class serving as entry point to your code once the startup process is complete. It should look as follows:

```
#include "main/adonhell.h"

class AdonhellApp : public adonhell::app
{
    // your application entry point
    int main ()
    {
        init_modules (GFX | ...);
        ...
    }
} theApp;
```

It must extend `adonhell::app` and implement the pure virtual method `int main()`. This is the method called after basic engine initialization. Exactly one instance of that class, called `theApp` must be provided, so that the main module can find it. `adonhell::app` gives you access to commandline arguments and further initialization methods, described in more detail in the API.

3.2 Using the Engine from Python

The main module provides roughly the same functionality when being used from within a python script, although a few more lines of code are required:

```
from adonhell import main

class App (main.AdonhellApp):
    def __init__ (self):
        main.AdonhellApp.__init__ (self)

    # -- your application entry point
    def main (self):
        self.init_modules (self.GFX | ...)
        ...

if __name__ == '__main__':
    theApp = App ()
    theApp.init (theApp.main)
```

The first difference to the C++ code is that we extend the class `Adonthe11App`, which is only available in the main module on Python side. It is in turn derived from the aforementioned class `adonthe11::app`, so the methods it provides are available to Python too. The second difference is the call to `theApp.init`, which makes the applications entry point known to `Adonthe11App`. It also triggers the startup procedure described below, which will finally call the user supplied method. Note that code placed after `theApp.init`, will not be executed until the engine quits – if it is executed at all! You have been warned ...

3.3 Engine Startup

After having seen how the main module can be used, lets see what happens when using it.

1. Parse command line parameters

As some parameters override defaults or supply otherwise important information, they are read first. The backend used or the configuration file loaded can be influenced that way.

2. Initialize base module

Amongst other things, the path to the configuration directory – required for the next step – will be set up. Other game related paths will be initialized too.

3. Read configuration file

Unless given on the command line, the backend to use is read from the configuration file *adonthe11.xml*. A different config file can be chosen via the command line.

4. Perform platform specific initialization

The main module's backend is loaded. If none was given via command line or configuration file, *sdl* is used as the fallback. Once loaded, it performs the operating system dependent initialization.

5. Call user-supplied main method

If all went well so far, the user supplied main method is executed. It should call the `init_modules` method to further initialize those parts of the engine it requires. Once the main method returns, the engine does some cleanup and finally exits.

4 Base Module

The most basic code is contained in this module, which means that you will definitely need to use it when writing code for Adonthell. As seen before, all the other modules depend on base.

4.1 Machine-Independence

One goal of the Adonthell engine is portability, and a number of services to aid this goal are implemented in the base module:

- **Primitive data types**

All primitive integer types are redefined by the base module. Using them instead of the data types provided by the underlying C library ensures that a short will always be 16 bit and integers 32 bit, regardless of the hardware architecture and operating system.

Note that this is not implemented yet, but by sticking to the types defined in base, it can be added easily. For example, the configure script could define types appropriately at compile time.

- **Endianness**

The same data files should be usable on both little and big endian hardware. Also, save games should be endian independent and both system types should be able to exchange data over a network. That means that there must be a way to convert data from little to big endianness and back. This is done through macros defined in the base module. Make sure to use them whenever data is read or written. The I/O methods described later are already endian independent – sticking to them where possible will avoid problems too. Btw., data is always stored in little endian format and converted at need.

- **Game speed**

Since computers tend to get faster and faster, it is important to make the engine independent from CPU speed. The base module allows to specify the duration of a single game cycle. When calling `base::Timer.update` after each cycle, the engine ensures that each cycle consumes exactly that timespan. That way, games will run at constant speed, given a fast enough CPU.

There's also limited support for slower CPUs. If computation of a cycle takes longer than allowed, the engine could react by skipping frames or by disabling costly, but otherwise unnecessary graphic or sound effects.

4.2 Data Persistence

The engine needs to be able to save the state of the gameworld and restore it at a later date. For that purpose, most objects must be serialized and written to disk. Since objects will change as development progresses, a mechanism is required to keep serialized data at least partially compatible with newer versions of the engine.

This can be achieved by storing each variable with an id and type information. That way, an old engine can skip over new fields in the serialized data or provide defaults for missing

fields if possible. Even if an object can't restore its state from such a data file, it will at least notice that fields are missing and can abort loading in a safe way.

Internally, an id, size, type and of course the value of a variable are written to a byte stream that can be later saved or transferred over the network. Multiple variables can be grouped together in a block to avoid confusing variables with the same id. It is therefore recommended to put the contents of each object into a separate block.

Blocks are also useful for storing array data with empty ids, thus saving space and time. These array elements can be easily restored by iterating over the contents of their block.

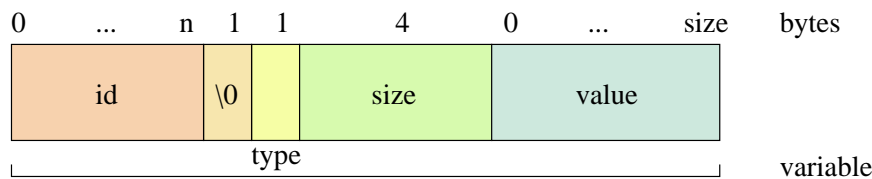


Figure 3: Variable Bytestream Format

The figure above shows the bytestream format of a single variable. Multiple variables are concatenated without delimiter. Blocks are treated like variables, with their own type code and contents stored in the value area.

A checksum can be calculated for the whole stream, so data corruption can be detected, although no error correction is possible.

As mentioned above, the bytestream is kept in memory. To save it to disk, a wrapper around zlib is provided by the base class that can be used to write it into a compressed file. That reduces the overhead somewhat and keeps file size small. This zlib wrapper can also be used independently from the serialization, to write primitive types or blocks of memory to compressed files. This should be avoided, however, as no compatibility is guaranteed that way.

4.3 Configuration Files

The base module provides a way to store configuration parameters in an XML file. The format of the file is very simple. The first level consists of *sections*, each of which can contain *options* with a value each. To retrieve a certain parameter, the name of the section and option it is kept in needs to be known by interested modules. A default value can be passed – if an option does not exist it is created with that default value. That way, the engine can start without configuration as long as it can provide a sensible default value for each parameter.

As discussed before, a default configuration is read during engine startup. It is automatically written back to disk when the engine quits.

With respect to a graphical configuration system, different option types are defined. One of them allows any type of value, others are limited to boolean values or a integer range. A last type allows only selection from a predefined list of values. The parameter type isn't stored

in the configuration file, but must be explicitly set by the module requesting the parameter. That way, the configuration is kept free of metadata and easily editable by hand.

4.4 Game Directories

Usually, there will be one directory for static gamedata shared by all users and a data directory in the users \$HOME, where saved games and configurations reside. That way, the engine can be installed by a privileged user and later be used by everyone else. When developing a new game, one might not want to install newly created game data, so a user defined data directory exists as well.

To ease the handling of data files, the base module provides a search strategy that operates on those three directories (which will differ for different games, of course). The method `path::open` tries to locate a file in one of the directories in the following order:

1. Saved game directory (if specified)
2. User supplied data directory (if specified)
3. Game specific data directory

The first matching file is opened. Data files should always be opened this way instead of requiring them to sit in a fixed location. Later on, this mechanism might be expanded by add-on or patch directories that will have to be included in the search path above.

5 Event Module

For efficiency, all game related tasks performed by scripts or the engine itself should be event-driven. That way, actions – which are usually written in Python – are only executed when needed.. Given that it is rather costly to call a Python method from C++ only to check a condition or to increase a counter, an integrated event dispatcher is a must to improve overall performance.

5.1 Event Handling

To make event handling even more efficient, there isn't a single event manager to take care of all events. Instead, one can write specialized manager classes for each type of event. These must be derived from `manager_base` and registered with the generic `manager` that will pass on events accordingly. This generic manager also provides the interface other modules require to dispatch events, completely hiding the specialized managers to the outside.

In order to get notified of events, a `listener` needs to be created and registered with the manager. If an event occurs, all matching listeners are activated, resulting in the execution of the Python callbacks they provide. Each callback will receive a pointer to the listener itself, to the triggering event and, optionally, further user defined arguments. Latter are specified when connecting the callback to the listener, but can be easily changed from within the callback – as the callback itself. User defined arguments are currently limited to integers and strings, as more complex objects cannot be saved when serializing a listener.

If an object does not want to receive any events temporarily, it can pause its listeners, thus avoiding the need to destroy them completely. Listeners can further have a repeat count, meaning they will destroy themselves after receiving a certain number of events.

Since objects will often have several listeners for different events, a `factory` can be used to group them together. Via the factory, all listeners can be paused or resumed, saved and loaded, thus removing the need to keep track of each individual listener.

5.2 Date and Time

Closely related to the event system is the game time, because many events in the game will be time based – especially those required for implementing NPC behaviour.

Time in the game will usually differ from real world time and is measured by game cycles. As the number of cycles per second is constant, game time will progress at a constant rate too. For that, a call to `date::update` after each cycle is required. Whenever a second of game time has passed (which is currently defined as five game cycles), a time event is dispatched.

The date class also provides fixed conversion methods to get the current weekday, day, hour, minute and second, based on a 7 day week, 24 hour day. In future, custom conversion rules might be made available to allow more unusual in-game date and time.

6 Python Module

This module includes the basic functionality essential for those objects that want to call Python methods or need to be accessible within Python scripts.

6.1 SWIG support

To allow passing an object from C++ to Python and vice versa, it must be wrapped in a `PyObject` by SWIG and unwrapped again later. Basically, SWIG stores a string representation of an object's pointer and the type information necessary to call its methods. Therefore, each class that needs to be passed between the engine and Python scripts has to provide this type information, i.e. its class name. This is done by the method `get_type_name`, which can be generated by the `GET_TYPE_NAME` macro. Once it is present, wrapping and unwrapping can be handled automatically by `pass_instance` and `retrieve_instance`.

In order to make parts of the engine *known* to Python in the first place, they have to be added to the appropriate SWIG interface file residing in the `py-wrappers/adonthell/` directory. Usually, including the header is enough, but in special cases, extra care has to be taken to avoid memory leaks or allow for in/out parameters. Details on this can be found in the SWIG manual.

6.2 Python Scripts and Methods

A wrapper around the Python C API is provided, to instantiate classes defined in Python scripts and call the methods they provide. If exclusive access to a python object is required, the `script` class can be used directly. It will always instantiate a new object. If that is not required, Python objects should be accessed via the `pool`. Each python object in the pool is only loaded once, thus reducing access time and memory footprint. Python objects that need their own context shouldn't be used that way, however, to avoid the side effects of multiple consumers accessing the same instance.

It should be noted that Python callback used by event listeners do use the pool. Extra care needs to be taken when writing those callbacks. As a result, it is possible that way to pass data between different events, as long as their callbacks are methods of the same Python class.

7 Rpg Module

This is one of the core modules, providing the low level classes essential for role playing games.

7.1 Quests and Logs

To represent the plot and record the player's progress in the game, the quest system has been written. It allows to define milestones of quests that can be queried and changed by various Python scripts, like dialogues, event callbacks or character schedules. Whenever the state of a quest changes, these scripts can make the necessary changes to the gameworld. (*Todo: generate quest event whenever a quest or part of a quest changes its state.*)

As quests are usually not linear and often solvable in multiple ways, each quest is represented by a tree, whose root is the quest itself and whose leaves represent its individual milestones. These milestones are the only part of a quest that can be directly influenced by scripts – they can be set to completed. All the quest nodes below the leaves have a rule how to determine their state of completion from the state of their children. Scripts that only need to know whether a quest as whole is completed can query the state of the quest root, while scripts that need more detailed information can query nodes higher up the tree.

Queries are not limited to the two states *complete* and *incomplete* though. The engine distinguishes between the following:

- **started** A quest is started as soon as its first milestone is set to completed. (As milestones are atomic, starting them means to complete them)
- **in progress** A quest that has been started but not yet completed.
- **completed** Once the player has accomplished a quest or part thereof, it is set to completed (in which case it is no longer in progress).

What has been described so far allows the engine to keep track of progress through the game. To help players keeping track, a quest log has been implemented too. This is closely tied to the quest tree, as each node of the tree can have a log entry attached that will be copied into the quest log whenever that node is set to completed.

The log system itself supports several *log books*; but so far, the quest log is the only one integrated into the engine. Others may be implemented by individual games. In order to ease navigation, the engine provides support for automatic index generation: a list of keywords can be specified and whenever an entry is copied into a log book, its text is searched for those keywords and an index entry is created under each keyword found in the text. Additionally, an entry can provide additional keywords, so that index entries can also be created for keys not found literally in the text.

On top of the indexing mechanism, a fulltext search could be easily implemented too. All it takes is creating a new index with the search terms as its only keywords. Then all entries of the desired log books need to be processed by that index, resulting in a list of index entries for each keyword. By limiting the search result to entries appearing in all lists, one will find

the log entries containing all the search terms. More complex boolean search operators could be supported by processing the resulting lists accordingly.

Even though the index is optimized for comparing text against a long list of keywords, a fulltext search done that way should still be pretty fast.

7.2 Items and Inventories

In this section, we will explore how items are modelled and kept. The main goal was to allow for all kinds of items, with many different abilities. Creation of new items should be fast and simple, without having to modify the engine. And last but not least, the item system as a whole should be efficient and convenient for both designers and players.

Therefore, items have been splitted into three layers.

1. **Engine layer.** Functionality required to manage items is implemented by the engine. This includes item instantiation, inventory handling and access to advanced item functionality.
2. **Extension layer.** The item system itself – properties and abilities as well as game rules concerning items – is implemented in Python on top of these basic functions. That way, different games can have totally different item systems.
3. **Data layer.** On top of the item system sits the item data, i.e. all the individual items actually available in a game.

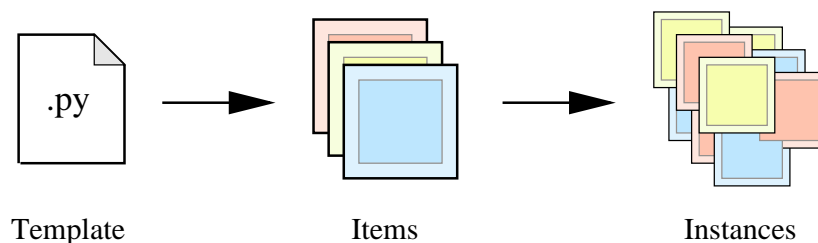


Figure 4: Item hierarchy

The figure shows the relation between extension, data and engine layer. For each “class” of items³, a *template* needs to be written in Python. By initialising attributes of the same template with different values, different *items* can be defined. At runtime, multiple *instances* of every such item can be created and used by the engine.

For efficiency reasons, the engine distinguishes between two different kinds of items: *mutable* items require a new instance for every occurrence in the game. As a result, they can be easily changed throughout the game. *Immutable* items on the other hand are instantiated only once. They are kept in a central `item_storage` and reference counts are used to allow multiple occurrences in the game.

³Like weapons, armour, potions, lightsources and so on. Subclassing is also possible, of course. Imagine ranged weapons, healing potions, etc.

Despite the fact that large parts of the item system are not implemented by the game engine, items are always stored on engine level, in so called `slots`. A slot may contain several items of the same kind, if the item is stackable. Immutable items are considered to be of a kind when their instance pointers are equal. For mutable items, item names have to match.

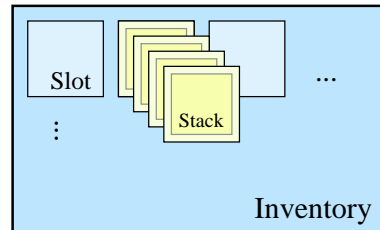


Figure 5: Inventory, slots and stacks

With the above in mind, an `inventory` is easily implemented. It is little more than a list of slots. To remain flexible, only the most basic inventory operations have been implemented on engine level, like adding new items. For convenience, the inventory tries to group items of same kind together in a *stack*. Furthermore, removal of items happens automatically as they are moved from one slot to another.

Inventories are not limited to a certain number of slots. If desired, they can even grow or shrink at runtime. Therefore, inventories can be used whenever large quantities of items need to be stored, not only to represent a character's backpack.

7.3 Characters and Equipment

8 Map module

There is no map module in v0.4 at present. What is described below refers to the v0.3 map engine, but chances are that a new implementation will work in similar ways.

8.1 Maps and Views

One prominent component of the engine is the renderer. It produces the graphical representation of a scene, and that about 40 times per second. (A new scene – or internal engine state – is computed about 75 times per second, btw.)

More important than those numbers or the fact that drawing and update operations are disjunct is the implementation. As the heading suggests, a model-view-architecture is used.

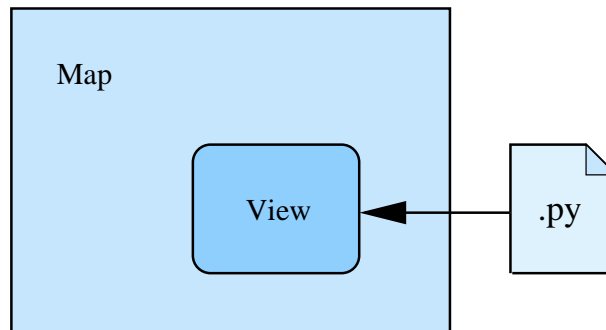


Figure 6: Architecture of the renderer

The part of the map being displayed is determined by so-called *mapviews*. A mapview can be of any size and can direct output to any *surface*. Multiple mapviews can be active at the same time, although at present they can't show different maps, just different areas of the same map.

The question that remains is how a mapview knows what part of a map to display. For that purpose, each mapview may have a schedule script assigned, which is executed before rendering takes place. The script in turn can use the methods provided by the mapview class for all kinds of effects. Usually, it will want to center the view on a certain character, but it is not limited to this.