General information

Plugins enhance the game with optional features.

They allow for special behaviours that can't be done through simple mapping, or require complex scripting actions - the only thing limiting is the imagination of the coder!

Current examples include:

- message boards, on which players can leave messages
- Imperial Post Office, that lets players exchange ingame mail
- playing slot machines at casinos
- improved guild system - currently being tested

Existing plugins

Currently existing plugins are :

Name	Description	Status
cfanim	animate objects	build, but untested
cfpython	run Python scripts	working
cflogger	logs events to a SQLITE database	experimental
cfnewspaper	newspaper generation	experimental
template	not a real plugin, but a skeleton to create new ones	up-to-date

Hooks

Plugin system works through *hooks*. A hook is merely an event taking place, to which plugins can respond. Default server behavior can sometimes be overridden totally.

The full list of events corresponds to the *event_xxx* archetypes, found in the /system folder of the archetypes [http://crossfire.svn.sourceforge.net/viewvc/crossfire/arch/trunk/system/].

Events are either *global* or *object-specific*. Examples of global events include a player joining or exiting the game, a map being loaded. Examples of object-specific events include an item being applied, a player attacking a monster.

Fix Me! check parameters and such :)

Hooking to an object-specific event

Just add an *event_xxx* object into your object's inventory, and fill the `title` field with the plugin name. See the specific plugin documentation for optional arguments that need to be set to that object. Note that the `slaying` field must be set too.

When an object-specific event is raised, affected object is sent to the plugin.

For some events, returning a non-zero value will prevent the default server processing to take place, allowing to override things. Fix Me! be more specific :)

The plugin registered function will get the following parameters. See specific events for the meaning of the field.

- op: `object*` which is the current object containing the event.
- activator: `object*`, can be NULL.
- third: `object*`, can be NULL.
- message: `const char*`, can be NULL.
- fix: `int` {Fix Me!} meaning?
- event: `object*` representing the actual event object linking to plugin. Will be `NULL` for global events. Its fields mean:
  - `subtype`: event code
  - `title`: plugin name
  - `slaying`: plugin specific value

Follows the full list of object-specific events.

Apply

Archetype: event_apply

This event is generated whenever the object is applied or unapplied.

Attack

Archetype: event_attack

This event is used in three cases:

- bound to a (hand) weapon, it is triggered each time the weapon is used to slay something; this can typically be used to generate special effects when you hit a monster. In this case, "op" is the weapon, "activator" the player and "third" the monster attacked. Returning a non-zero value cancels the attack
- bound to a monster, it is triggered when the monster is attacked. In this case, "op" is the monster, "activator" is the player, "third" is the weapon (can be equal to "activator"). Returning a non-zero value cancels the attack
- bound to an arrow, it is triggered when the arrow hits something. "op" is the arrow, "hitter" what really hits (arrow can have stuff inside), "third" is the victim

Close

Archetype: event_close

Generated when a container is closed.

Death

Archetype: event_death

Generated when the object dies.

Drop

Archetype: event_drop

Generated when the object is dropped, either on the floor or in a container.

Pickup

Archetype: event_pickup

`WhoAmI` is about to be picked up by `WhoIsActivator` and put into `WhoIsOther` (will be WhoIsActivator if put into inventory, or container else). Event is called when all checks (weight, container, levitation, ...) are done, creature picking up can really pick up.

Return 0 to allow to pickup, non zero to prevent from picking up.

Say

Archetype: event_say

Generated when someone says something around the object.

Stop

Archetype: event_stop

Generated for a thrown object, when the object is stopped for some reason (wall, max distance, ...). Will not be called when hitting a living creature, but `Attack` is called.

Time

Archetype: event_time

Generated each time the object gets an opportunity to move.

Return non zero value to prevent the regular processing to occur.

Throw

Archetype: event_throw

Generated when the object is thrown.

Trigger

Archetype: event_trigger

Used for various objects, like traps, teleporters or triggers. Generated when those objects are used (for example, when a player passes through a teleporter).

- for a scroll/book/tome, generated when player writes into the item. `op` is the item, `activator` is the player, `message` is what player is trying to write. Return non zero to prevent writing. Event is generated before length is checked for overflow.
- for a magical scroll, generated when player inscribes a spell in the item. `op` is the item, `activator` is the player, `third` is the spell being inscribed. Return non zero to prevent writing. Event is generated after all checks are done (player can really write the scroll, he didn't read it accidentally) but before sp/gr are decreased.

Timer

Archetype: event_timer

Generated when the timer connected triggered.

Hooking to global events

Those concern the game as a whole or can't be bound to a specific object. Those events may be "registered" by a plugin (it means that the plugin requests to get a message each time one of those events happens).

Plugin should use provided server callbacks to register itself. See the specific plugin documentation.

Event	Description	Parameters	Note
Born	Generated when a new character is created.	`object*` pointing to player	
Clock	Generated at each game loop.	(none)	When no player is logged, the loop "stops", meaning that clock events are not generated anymore!
Crash	Generated when a server crash does occur. It is not a recursive event, so if a crash occur from *inside* the crash event handling, it is not called a second time, preventing infinite loops to occur.	(none)	This event is not implemented for now.
Gdeath	Generated whenever a player dies.	`object*` pointing to player	
Gkill	Generated whenever something/someone is killed.	`object*` pointing to dead object, `object*` pointing to killer.	
Kick	Generated when a player was kicked by a DM.	`object*` pointing to kicked player, `const char*` containing the parameter the DM used to kick	
Login	Generated whenever a player logs into the game.	`player*` pointing to player, `const char*` containing the hostname of the client.	
Logout	Generated whenever a player logs out of the game.	`player*` pointing to player, `const char*` containing the hostname of the client.	
Mapenter	Generated whenever someone enters a map.	`object*` pointing to the player, `map*` pointing to the map	
Mapleave	Generated whenever someone leaves a map.	`object*` pointing to the player, `map*` pointing to the map	
Mapload	Generated when a map is loaded in memory.	`map*` pointing to the map	
Mapreset	Generated each time a map is reset.	`map*` pointing to the map	
Mapunload	Generated when a map is being unloaded from memory.	`map*` pointing to the map	
Muzzle	Generated when a player was muzzled by a DM.	`object*` pointing to muzzled player, `const char*` containing the parameter the DM used to muzzle	
Remove	Generated when a player character is removed from the game ("quit" command).	`object*` pointing to the player	
Shout	Generated whenever someone shouts something.	`object*` pointing to talking player, `const char*` containing the message, `int` containing the priority	
Tell	Generated whenever someone tells something.	`object*` pointing to talking player, `const char*` containing the message, `object*` containing the recipient of the message	

Registering a command

A plugin can register a custom game command. This command will work exactly like other commands from the player's point of view, accepting arguments and such.

A plugin can override an existing Crossfire command simply by declaring a command with the same name. Server command will be ignored.

Creating a plugin

Coding

Creating a plugin requires some knowledge of Crossfire's internals.

Plugins should use the `common` plugin helper library (located in `plugins/common` [http://crossfire.svn.sourceforge.net/viewvc/crossfire/server/trunk/plugins/common/] directory), and use provided functions to manipulate Crossfire data. This library handles the actual communication with the server, and does some runtime checks on values.

Warning: a plugin can easily crash the server and/or corrupt files if care is not taken in data manipulation. Some checks are done through `assert`, but there are times checks can't be done, thus it's the responsability of the plugin writer to take care of the plugin logic.

Some rules for plugin writing:

- a plugin should never use `free` or equivalent on data the server allocated. Use provided free functions.
- a plugin should never directly link to the Crossfire libraries (`common`, `socket`, ...) and call functions. This may work on some platforms (Linux seems to handle that just fine), but will not work on others (example is Windows). Thus always use provided common plugin interface
- a plugin should never change directly a value in a Crossfire object (player structure, ...), but use provided wrapping functions. Ideally, a plugin should not access directly that structure's memory to read values, either.

Client updating

Fix Me! expand/check: is an object removed automatically from player's inventory? is view updated? is fix_object() called?

Creating plugin skeleton

The simplest way is to look at the template plugin, available in plugins/template directory of the server sources [http://crossfire.svn.sourceforge.net/viewvc/crossfire/server/trunk/plugins/template/]

In trunk, there now is a script, `plugins/template/create_plugin.sh`, that will automate all steps below, excluding running `configure && make && make install`. The syntax is: `create_plugin.sh cftest "Test plugin"` from the `plugins/template` directory.

In case you want to create a plugin manually, here are step by step instructions to create a basic plugin. Plugin name will be assumed to be `cftest`. Paths are relative to server root.

- create your plugin directory `cftest`, in `plugins`
- copy `plugins/template/plugin_template.c` to `plugins/template/cftest.c`
- copy `plugins/template/include/plugin_template.h` to `plugins/cftest/include/cftest.h`
- modify `plugins/cftest/cftest.c` and `plugins/cftest/include/cftest.h` to correct the file names in the includes and such. You may also want to change the plugin's name
  - in particular, `PLUGIN_NAME` should be set to a unique identifier that will be your plugin's name for the `slaying` field of events.
- copy `plugins/template/Makefile.am` to `plugins/cftest/Makefile.am`
- edit this file, to change all occurrences of `plugin_template` to `cftest` (that fixes paths and a few macros)
- edit `plugins/Makefile.am`, add `cftest` to the `SUBDIRS` line
- edit `configure.ac`
  - find the line `AC_OUTPUT([Makefile`
  - somewhere in between other lines, add `plugins/cftest/Makefile`

Whether you used the script or manually created a plugin, you need to do the following steps:

- run `autoconf && automake && configure && make && make install`

- your plugin should be installed and ready to run

Now you need to write your actual plugin logic.

Fix Me! link to common plugin documentation (generated from doxygen hopefully)

Fix Me! specific Windows stuff / workspace/project issue

Plugin internals

This paragraph only concerns people wishing to write a plugin without using the `common` interface, or extending it. It is not intended for everyday use.

Important note: the API, and other parts, are valid for the `trunk`. While it may still be correct for the `branch`, one should ensure it works the same way. In particular, `branch` uses returned `void*` value to return values to plugin, whereas `trunk` uses an additional pointer parameter.

All functions available to plugins share the same prototype, `f_plug_api`, defined in `include/plugin.h` [http://crossfire.svn.sourceforge.net/viewvc/crossfire/server/trunk/include/plugin.h?view=markup] .

Parameters sending

All parameters are sent through the use of variable argument lists, using the macros `va_start`, `va_arg` and `va_end`.

Data type

All functions accept as first parameter an `int* type`. This parameter is used by server and plugin to exchange the type of a modified/returned value. Apart its presence for the variable argument list handling, it is used to check data coherence.

Data is exchanged as either a value or a pointer to a Crossfire structure. When the server needs to return a value to the plugin (for function wrapping, properties getting, ...), it expects the last argument to be a pointer to a variable of the returned value's type.

Data types available are defined in `include/plugin.h` [http://crossfire.svn.sourceforge.net/viewvc/crossfire/server/trunk/include/plugin.h?view=markup] .

Special case: when a `CFAPI_STRING` needs to be transferred, 2 parameters are expected, a pointer to the buffer and an integer to the buffer's size (which can't be always be determined automatically).

Note that signed/unsigned variables are transferred as signed, and should be cast appropriately when needed.

Basic data access

Access (get/set) to properties of the Crossfire objects, maps, structures is done through property wrappers.

Syntax for wrapper is eg `cf_object_get_property(int* type, object* ob, int propcode, (property type)* value)`.

Fix Me! expand

Function wrapping

This wraps specific Crossfire functions. The calling convention is to send parameters in the same order as the wrapped function, and add as the last parameter a pointer to a variable of the same type as the return value which will receive the actual function return value. The hook return value will be a constant indicating whether the function was called or another error occurred (Fix Me! expand/check/make that true in the code ;p)

Let's take for example the wrapper for `get_ob_key_value`.

The function prototype is:

```
const char *get_ob_key_value(const object *op, const char *const key)
```

(the return value is a shared string)

The `server-side` [http://crossfire.svn.sourceforge.net/viewvc/crossfire/server/trunk/server/plugins.c?view=markup]

[http://crossfire.svn.sourceforge.net/viewvc/crossfire/server/trunk/server/plugins.c?view=markup]
hook is (note the NULL return value, since this is being worked on Fix Me! remove when fixed):

```
void* cfapi_object_get_key(int* type, ...)
{
    va_list args;
    const char* keyname;
    const char** value;
    object* op;

    va_start(args, type);
    op = va_arg(args, object*);
    keyname = va_arg(args, const char*);
    value = va_arg(args, const char**);
    va_end(args);

    *value = get_ob_key_value(op, keyname);
    *type = CFAPI_SSTRING;
    return NULL;
}
```

The plugin wrapper function, defined in the common library [http://crossfire.svn.sourceforge.net/viewvc/crossfire/server/trunk/plugins/common/plugin_common.c?view=markup], is:

```
const char* cf_object_get_key(object* op, const char* keyname)
{
    int type;
    const char* value;
    cfapiObject_get_key(&type, op, keyname, &value);
    return value;
}
```

where `cfapiObject_get_key` is the matching hook.

Linked list handling

In some cases, objects are linked through the use of a `next` field, with a `first_` pointer somewhere (includes objects, friendly list, maps, archetypes). Since the `next` field will certainly be a property for the object, a convention for getting the `first_` item is to call this property getter with a NULL value for the object.

Thus, for the `partylist` linked list, the server-side functions looks like (parts edited out):

```
void* cfapi_party_get_property(int* type, ...)
{
[snipped]

    case CFAPI_PARTY_PROP_NEXT:
        rparty = va_arg(args, partylist**);
        *rparty = (party ? party->next : get_firstparty());
        *type = CFAPI_PPARTY;
        break;
```

The common library's function for first party is:

```
partylist* cf_party_get_first(void)
{
    int type;
    partylist* value;
    cfapiParty_get_property(&type, NULL, CFAPI_PARTY_PROP_NEXT, &value);
    assert(type == CFAPI_PPARTY);
    return value;
}
```

whereas access to the next party is done through:

```
partylist* cf_party_get_next(partylist* party)
{
    int type;
    partylist* value;
    cfapiParty_get_property(&type, party, CFAPI_PARTY_PROP_NEXT, &value);
    assert(type == CFAPI_PPARTY);
    return value;
}
```