Roger Firth's IF pages

InfAct -- about Inform NPCs

Like its associate InfLight and InFancy, InfAct is an Inform overview site, this time covering Non-Player Characters (NPCs). As before, this is just a sampler of the tools and techniques available, more a kit of parts than a ready-to-run solution; the work of breathing life into your creation is a long hard slog, but very rewarding if you make a success of it.

The material is loosely organised as follows:

Conventions

To clarify where the various example displays come from, a little colour-coding is used:

```
This is a sample of text in an Inform source file.
```

```
This is from a Z-machine interpreter at run-time.
```

Acknowledgements

Various modules are mentioned along the way; my gratitude, as ever, to the authors concerned:

- Start at the Inform home page for details of the environment in which we're operating.
- Be sure to read Emily Short's fine essay on NPC characterization.
- Pick up Irene Callaci's **AskTellOrder.h**, L Ross Raszewski's **Converse.h**, Adam Cadre's **Flags.h**, Jesse Burneko's **Info.h**, Neil Brown and Alan Trewartha's **MoveClass.h**, Adam Cadre and David Glasser's **PhTalkOO.h**, Volker Lanz's **NPC_Engine.zip** and David Cornelson's **WhoWhat.h** from the archives at if-archive/infocom/compilers/inform6/library/contributions.
- You can get the Cloak of Darkness example game from this site.

We'll begin by creating a minimal NPC object, then provoking a few reactions.

Roger Firth's IF pages

InfAct -- about Inform NPCs

An NPC -- Non-Player Character -- is just another Inform object. Well, ok, that "just" is maybe a little glib; it's probably truer to say that creating good NPCs is one of the hardest part of any game. Tricky, certainly; impossible, no. And many players would agree that the ability to interact with a well-written NPC is one of the characteristics making a memorable and satisfying game. So let's get started.

The features that distinguish an NPC from any other Inform object are easily listed:

- the attribute **animate** (or just possibly **talkable**) is key.
- You'll also need one of the attributes **male**, **female** or **neuter** (to ensure that pronouns like his/her/its are used correctly), and maybe **proper** (to avoid things like "You see an Aunt Bessie here.").
- the properties **react_before** and **react_after** enable your NPC to respond to nearby activities.
- the **life** and **orders** properties handle direct interaction.
- there's also a **grammar** property which may be needed in certain circumstances.

You may also need other properties like **daemon** and **each_turn**, but they aren't so closely tied to NPC-specific behaviour.

Starting from scratch

Creating the bare bones of an NPC is really easy.

```
Object  usher "gentleman usher" cloakroom
  with  name 'usher' 'gentleman' 'gentle' 'man',
        description "The usher is smartly uniformed.",
  has   animate male;
```

From these humble beginnings, anything is possible. What currently happens is fairly modest, but still a tribute to the power of the Inform library.

```
>SHOW CLOAK TO THE USHER
(the velvet cloak to the usher)
The usher is unimpressed.

>ASK USHER ABOUT THE THEATRE
There is no reply.
```

Simple reactions

One easy way to inject some life is to have the NPC notice your comings and goings.

```
Object  usher "gentleman usher" cloakroom
  with  name 'usher' 'gentleman' 'gentle' 'man',
        description "The usher is smartly uniformed.",
        react_before [;
            Go: print "The usher nods politely as you leave.^"; rfalse;
            Examine: if (noun == player)
                "~If I may say so, as good-looking as ever.~";
            ],
        react_after [;
            Go: print "~Good evening.~^"; rfalse;
            Drop: if (noun == cloak) {
                move cloak to player;
                "~Oops - let me pick that up for you.~";
                }
            ],
  has   animate male;
```

The **react_** properties enable the NPC to passively comment on (with the **rfalse**) or actively affect (with the **rtrue**) actions that would happen normally in another location.

```
>WEST
"Good evening."

Cloakroom
The walls of this small room were clearly once lined with hooks, though now only
one remains. The exit is a door to the east.

You can see a gentleman usher here.

>EXAMINE USHER
The usher is smartly uniformed.

>EXAMINE ME
"If I may say so, as good-looking as ever."
```

```
>REMOVE CLOAK. DROP IT
(the velvet cloak)
You take off the velvet cloak.
"Oops - let me pick that up for you."

>EAST
The usher nods politely as you leave.
```

Unsolicited interjections

Another straightforward approach is to use a daemon or timer of some sort which causes the NPC to make his presence felt. Here's a first attempt, using the **each_turn** property:

```
Object  usher "gentleman usher" cloakroom
  with  name 'usher' 'gentleman' 'gentle' 'man',
        description "The usher is smartly uniformed.",
        each_turn [; switch(random(10)) {
            1: "^~Welcome to the Opera.~";
            2: "^~Shall I hang your cloak on this hook?~";
            3: "^~Still raining, I see?~";
            4: "^The usher clears his throat apologetically.~";
            5: "^The usher shifts position slightly.";
            default: ;      ! values 6-10 are ignored
            }
            ],
  has   animate male;
```

Using this simple method, the NPC will make a random remark roughly five turns out of ten; you can easily add to the repertoire of remarks and adjust the probability of one being made. However, hearing the same remark more than once soon gives the game away, so a better idea uses an array of logical flags to prevent this:

```
Array   UsherSaid -> 6;     ! 0=unused, 1-5=true/false flags

[ UsherSays num str;
        if (UsherSaid->num) rfalse;
        UsherSaid->num = true;
        print_ret (string) str;
        ];

Object  usher "gentleman usher" cloakroom
  with  name 'usher' 'gentleman' 'gentle' 'man',
        description "The usher is smartly uniformed.",
        each_turn [; switch(random(10)) {
            1: UsherSays(1, "^~Welcome to the Opera.~");
            2: UsherSays(2, "^~Shall I hang your cloak on this hook?~");
            3: UsherSays(3, "^~Still raining, I see?~");
            4: UsherSays(4, "^The usher clears his throat apologetically.");
            5: UsherSays(5, "^The usher shifts position slightly.");
            default: ;      ! values 6-10 are ignored
            }
            ],
  has   animate male;
```

You could use the library package **`Flags.h`** to manage the logical flags as bits rather than bytes. Also, a more sophisticated implementation would permit random NPC remarks only in turns when the player *hasn't* already interacted with the NPC.

Next, other forms of silent interaction.

InfAct -- about Inform NPCs

You probably know that the standard library verbs (in **Grammar.h**) include a number of grammars which include the **creature** token specifically targetted at NPCs.

```
Verb 'give' 'pay' 'offer' 'feed'
        * held 'to' creature        -> Give
        * creature held             -> Give reverse
        * 'over' held 'to' creature -> Give;
Verb 'show' 'present' 'display'
        * creature held             -> Show reverse
        * held 'to' creature        -> Show;
Verb 'wake' 'awake' 'awaken'
        * creature                  -> WakeOther
        * creature 'up'             -> WakeOther
        * 'up' creature             -> WakeOther;
Verb 'kiss' 'embrace' 'hug'
        * creature                  -> Kiss;

Verb 'ask'
        * creature 'about' topic    -> Ask
        * creature 'for' noun       -> AskFor;
Verb 'tell'
        * creature 'about' topic    -> Tell;
Verb 'answer' 'say' 'shout' 'speak'
        * topic 'to' creature       -> Answer;
```

In this segment, we'll talk about the first group of actions -- Give, Show, WakeOther and Kiss -- which cover forms of interaction where you don't need to say anything. (A later segment will look at Ask, Tell and Answer.) These four actions, plus Attack and ThrowAt, are passed to an NPC's **life** property.

```
Object  usher "gentleman usher" cloakroom
  with  name 'usher' 'gentleman' 'gentle' 'man',
        description "The usher is smartly uniformed.",
        life [;
        Give:
            if (noun ~= cloak) print_ret (The) self, " courteously refuses ",
              (the) noun, ".";
            move cloak to hook; give cloak ~worn;
            "~I'll just hang it on here for you.~";
        Show:
            if (noun ~= cloak) print_ret (The) self, " smiles politely.";
            "~I don't think I've ever seen one as dark as that!~";
        WakeOther:
            "~Sorry, I must have dozed off for a minute.~";
        Kiss:
            print_ret "On second thoughts, you don't know ", (ItOrThem) self,
              " all that well.";
        Attack:
            print_ret (The) self,
              " looks startled, but doesn't otherwise respond.";
        ThrowAt:
            print (The) self, " deftly catches ", (the) noun;
            if (noun ~= cloak) " and returns it to you.";
            move cloak to hook; give cloak ~worn;
            " and carefully places it on the hook.^
              ~Things not been going too well today?~";
            ],
  has   animate male;
```

That's enough to handle some simple actions involving you and the NPC.

```
>SHOW THE CLOAK TO THE USHER
"I don't think I've ever seen one as dark as that!"

>THROW IT AT HIM
(first taking the velvet cloak off)
You take off the velvet cloak.
The gentleman usher deftly catches the velvet cloak and carefully places it on
the hook.
"Things not been going too well today?"

>KISS THE USHER
On second thoughts, you don't know him all that well.
```

Extending the life property

Suppose you wish to add some extra forms of interaction. For example, you might fancy getting up close and personal with your NPC, specifically distinguishing between hug/embrace, touch/feel/pat, stroke/caress, fondle/grope, and kiss. Some appropriate verbs are already defined, but their behaviour isn't as well-tuned as you'd like. Also, whereas you can trap the **Kiss** action in your **life** property, the same isn't by default true for the **Touch** action. These lines are from **`Grammar.h`**.

```
        Verb 'kiss' 'embrace' 'hug'
                * creature                  -> Kiss;

        Verb 'touch' 'fondle' 'feel' 'grope'
                * noun                      -> Touch;
```

The matching action routines are in **`VerblibM.h`** (for clarity, I've change **`L__M()`** calls into plain print statements):

```
        [ KissSub;
          if (ObjectIsUntouchable(noun)) return;
          if (RunLife(noun,##Kiss)~=0) rfalse;
          if (noun==player) "If you think that'll help.";
          "Keep your mind on the game."; ];

        [ TouchSub;
          if (noun==player) "If you think that'll help.";
          if (ObjectIsUntouchable(noun)) return;
          if (noun has animate) "Keep your hands to yourself!";
          "You feel nothing unexpected."; ];
```

To achieve the desired effect, we need to do three things:

1. change the existing library grammar to match the extended verbs
2. create appropriate new action routines; from examining the **`KissSub()`** routine, it's clear that **`RunLife()`** is the key to invoking the **Life** property
3. extend our NPC's **life** property to handle the new actions

First, here's a possible grammar:

```
        Extend only 'embrace' 'hug' replace
                * creature                  -> Hug;

        Extend only 'touch' replace
                * creature                  -> Pat
                * noun                      -> Touch;

        Verb 'pat'
                * creature                  -> Pat;

        Extend only 'feel' replace
                * creature                  -> Stroke
                * noun                      -> Touch;

        Verb 'stroke' 'caress'
                * creature                  -> Stroke;

        Extend only 'fondle' 'grope' replace
                * creature                  -> Fondle
                * noun                      -> Touch;
```

Second, we need action routine which mimic the default behaviours of **`KissSub()`** and **`TouchSub()`**:

```
        [ HugSub;
                if (ObjectIsUntouchable(noun)) return;
                if (RunLife(noun,##Hug) ~= 0) rfalse;
                if (noun == player) return L__M(##Touch,3,noun);
                L__M(##Kiss,1,noun);
                ];
        [ PatSub;
                if (ObjectIsUntouchable(noun)) return;
                if (RunLife(noun,##Pat) ~= 0) rfalse;
                if (noun == player) return L__M(##Touch,3,noun);
                L__M(##Touch,1,noun);
                ];
        [ StrokeSub;
                if (ObjectIsUntouchable(noun)) return;
                if (RunLife(noun,##Stroke) ~= 0) rfalse;
                if (noun == player) return L__M(##Touch,3,noun);
                L__M(##Touch,1,noun);
                ];
        [ FondleSub;
                if (ObjectIsUntouchable(noun)) return;
                if (RunLife(noun,##Fondle) ~= 0) rfalse;
                if (noun == player) return L__M(##Touch,3,noun);
                L__M(##Touch,1,noun);
                ];
```

Finally, the extended **life** property:

```
life [;
    ⋮
Hug:
    print_ret "You clasp ", (ItOrThem) self, " in a comradely embrace.";
Pat:
    print_ret (CThatOrThose) self,
        " steps slightly back, eyeing you cautiously.";
Stroke:
    "Warm, firm, smoothly curvaceous yet pulsing with life.";
Fondle:
    print_ret "You run your hand through ", (the) self,
        "'s long dark hair.";
Kiss:
    print_ret "On second thoughts, you don't know ", (ItOrThem) self,
        " all that well.";
    ⋮
    ],
```

I'm sure your imagination is sufficient to tell you how this will perform at run-time.

(You could probably achieve much the same effect by using a **before** property instead of **life**. However, I'd recommend following in Graham's footsteps and use **life** for NPC interactions, unless you're sure you know what you're doing.)

Time to start getting vocal.

Intro ▸ Basics ▸ **Silent interaction** ▸ Conversation ▸ Conversation packages ▸ Orders ▸ Movement

InfAct -- about Inform NPCs

When describing the verb grammars using the **creature** token, we deferred discussion of the Ask, Tell and Answer actions until later. This is because these actions also use a **topic** token, which is treated by the parser as a bit of a special case.

```
Verb 'ask'
        * creature 'about' topic    -> Ask;
Verb 'tell'
        * creature 'about' topic    -> Tell;
Verb 'answer' 'say' 'shout' 'speak'
        * topic 'to' creature       -> Answer;
```

Whereas the parser happily maps a command like SHOW THE BLACK VELVET CLOAK TO THE USHER into the variables **action**=Show, **noun**=cloak object and **second**=usher object, it's rather less helpful with the similar SAY THE BLACK VELVET CLOAK TO THE USHER. You still get **action**=Show and **second**=usher, but **noun** just points to the first dictionary word which isn't THE; in our example it holds 'black', which isn't all that informative. Two additional variables are set -- **consult_from**=2 and **consult_words**=4 -- so that effectively the parser is saying: the topic of this conversation occupies four words starting at the second word in the input buffer; making some real sense of those words is down to you. The handling of Ask and Tell is similar, except that in these cases it's **second** which points to the first dictionary word which isn't THE.

So, if you plan to allow conversation in your game -- and there's only so many deaf/indifferent/uncomprehending NPCs you can get away with -- you need to devise a strategy for dealing with topics. The rest of this segment looks at some of the possibilities.

Using the first parsed word

The first approach, viable only if you anticipate very limited conversation, is simply to use the pointer to the first non-THE word. For example, we could code the Ask action as follows (remember, you could replace the rfalse with a print statement of your own to provide exception handling):

```
life [;
Ask: switch(second) {
    'black','velvet','cloak': "~It'll be quite safe in here.~";
    'smart','uniform': "~The management here is very particular.~";
    'hook','hooks': "~Yes, a few more would be useful.~";
    }
    rfalse;
Give:
    ⋮
    ],
```

It works, up to a point. Note how the perfectly sensible question about HIS UNIFORM isn't handled properly, because we didn't allow for 'his' in the switch statement:

```
>ASK THE USHER ABOUT THE VELVET CLOAK
"It'll be quite safe in here."

>ASK HIM ABOUT THE HOOKS
"Yes, a few more would be useful."

>ASK HIM ABOUT HIS UNIFORM
There is no reply.
```

Looking at all the words

Slightly more ambitiously, you can look at all the topic words before deciding what to do. Here's a routine **`ScanTopic()`** which uses the **consult_from** and **consult_words** values to fetch all the topic, discarding minor noise words and words not in the dictionary, and storing the remainder in a **`topicWord`**table.

```
Constant topicLimit 7;
Array   topicWord table topicLimit;

[ ScanTopic x y z;
        y = z = 0;
        wn = consult_from;
        while (consult_words--) {
            x = NextWord();
            if (x == 0) ++z;
            else if (x ~= 'the' or 'a' or 'an' or 'some'
                    or 'this' or 'that' or 'these' or 'those'
                    or 'in' or 'of' or 'at' or 'on' or 'with'
                    or 'my' or 'his' or 'her' or 'its' or 'their'
                    or 'is' or 'are' or 'has' or 'have'
                    ) if (y < topicLimit) topicWord-->(++y) = x; else ++z;
            }
        @storew topicWord 0 y;  ! -S doesn't allow topicWord-->0 = y;
        return z;
        ];
```

`ScanTopic()` returns the number of topic words which weren't in the dictionary, so that you can estimate how reliably the words in the table match what the user intended. Here's an Ask handler which uses **`ScanTopic()`**:

```
        life [;
        Ask:
            if (ScanTopic()<2) for ( : topicWord-->0 : (topicWord-->0)--)
                switch(topicWord-->(topicWord-->0)) {
                'black','velvet','cloak': "~It'll be quite safe in here.~";
                'smart','uniform': "~The management here is very particular.~";
                'hook','hooks': "~Yes, a few more would be useful.~";
                }
            "~I'm not sure about that.~";
        Give:
            ⋮
            ],
```

In this code, up to one unknown word is acceptable in the topic text. The words in the table are tested in reverse order, though you could go forwards if you wished.

```
>ASK THE USHER ABOUT THE VELVET CLOAK
"It'll be quite safe in here."

>ASK HIM ABOUT HIS UNIFORM
"The management here is very particular."

>ASK HIM ABOUT THE OPERA HOUSE
"I'm not sure about that."
```

Parsing as an object

One problem with the previous approaches is that we're trying to parse the input text ourselves; what we'd rather do is have the much clever library parser do it for us. Here's a routine **`ParseTopic()`** which parses the topic using the library routine **`NounDomain()`**:

```
[ ParseTopic x;
        wn = consult_from;
        x = NextWord();
        if (x == 'the') {consult_from++; consult_words--; } else wn--;
        x = NounDomain(actor, actors_location, NOUN_TOKEN);
        if (x == REPARSE_CODE) {
            Tokenise__(buffer, parse);
            wn = consult_from;
            x = NounDomain(actor, actors_location, NOUN_TOKEN);
            if (x == REPARSE_CODE) x = 0;
            }
        return x;
        ];
```

The strength of this approach is that we're applying the full leverage of the browser to resolve the topic into a reference to an object rather than to a dictionary word. Sadly, this is also its weakness, since we need to provide a scope (here, **actor** and **actor_location**) where that object may be found. This works well for the cloak and the hook, but doesn't work at all for the usher's uniform, which is merely mentioned, not instantiated as an object. Here's the new simplified Ask handler:

```
        life [;
        Ask: switch(ParseTopic()) {
            cloak: "~It'll be quite safe in here.~";
            hook: "~Yes, a few more would be useful.~";
            }
            "~I'm not sure about that.~";
        Give:
            ⋮
            ],
```

And this all behaves as you'd expect.

```
>ASK THE USHER ABOUT THE VELVET CLOAK
"It'll be quite safe in here."

>ASK HIM ABOUT THE HOOK
"Yes, a few more would be useful."

>ASK HIM ABOUT HIS UNIFORM
"I'm not sure about that."
```

Parsing as a topic

A good way forward would be to combine the strengths of the two previous approaches: use the power of the parser, but reset its scope to embrace a family of 'topic' objects. This turns out to be surprisingly easy. First we define those objects:

```
Object  Topics "conversational topics";
Object  -> t_cloak    with name 'handsome' 'dark' 'black' 'velvet' 'cloak';
Object  -> t_uniform  with name 'smart' 'uniform';
Object  -> t_hook     with name 'small' 'brass' 'hook' 'peg' 'hooks';
Object  -> t_yes      with name 'yes' 'please' 'ok';
Object  -> t_no       with name 'no' 'thanks';
Object  -> t_hello    with name 'hello' 'hi';
Object  -> t_goodbye  with name 'goodbye' 'good-bye' 'good' 'bye' 'cheerio';
```

Next, we adjust the grammar to bring the objects into scope:

```
[ TopicScope; switch(scope_stage) {
        1: rfalse;
        2: ScopeWithin(Topics); rtrue;
        3: "At the moment, even the simplest questions are confusing.";
        } ];

Extend 'look' first
        *                                    -> Look
        * 'up' scope=TopicScope 'in' noun    -> Consult reverse;
Extend 'consult' first
        * noun 'about' scope=TopicScope      -> Consult;
Extend 'ask' first
        * creature 'about' scope=TopicScope -> Ask;
Extend 'tell' first
        * creature 'about' scope=TopicScope -> Tell;
Extend 'answer' first
        * scope=TopicScope 'to' creature    -> Answer;
```

And finally we set up handlers in the NPC's **life** property as usual:

```
        life [;
        Ask: switch(second) {
            t_cloak: "~It'll be quite safe in here.~";
            t_uniform: "~The management here is very particular.~";
            t_hook: "~Yes, a few more would be useful.~";
            }
            "~I'm not sure about that.~";
        Tell: switch(second) {
            t_cloak: "~That's absolutely fascinating.~";
            }
            print_ret (The) self, " expresses mild interest.";
        Answer: switch(noun) {
            t_yes,t_no: "~I'm sure you're right.~";
            t_hello: print_ret (CTheyreorThats) self, " pleased to see you.";
            t_goodbye: "~Come again soon.~";
            }
            print_ret (The) self, " doesn't respond.";
        Give:
            ⋮
            ],
```

And this time we've got a conversational system that performs fairly well.

```
>SAY HELLO TO THE USHER
He's pleased to see you.

>SHOW THE CLOAK TO THE USHER
"I don't think I've ever seen one as dark as that!"

>TELL USHER ABOUT CLOAK
"That's absolutely fascinating."

>ASK USHER ABOUT CLOAK
"It'll be quite safe in here."

>ANSWER YES TO USHER
"I'm sure you know best."
```

In fact, there's a library package **`Info.h`** which does exactly this.

Canned conversations

As an aside, an alternative approach which is sometimes appropriate is to replace the ASK/TELL/ANSWER mechanism by one which simply replays a pre-programmed speech or dialogue. Here's a fairly crude example, showing a new TALK verb:

```
Object  usher "gentleman usher" cloakroom
  with  name 'usher' 'gentleman' 'gentle' 'man',
        description "The usher is smartly uniformed.",
        talk_number 0,
        life [;
        Order,Ask,Tell,Answer: print_ret "Just use T[ALK] [TO ", (the) self, "].";
        Talk:
        if (self.talk_number == 0) {
            if (cloak in player)            self.talk_number = 10;
            else if (self hasnt general)    self.talk_number = 20;
            else if (message.number > 0)    self.talk_number = 30;
            }
        switch (self.talk_number++) {
            10: move cloak to hook; give cloak ~worn;
                "~Would you mind hanging up my cloak?~^
                ^~With pleasure, sir. There you are now.~";
            11:
                "~Thank you very much.~^
                ^~You're most welcome. Looks like a bad night out there?~";
            12: self.talk_number = 20;
                "~Yes, it's raining quite hard at the moment.~^
                ^~Well, you'll be warm and dry in here. Enjoy the opera.~";
            20:
                "~What is tonight's performance?~^
                ^~It's Don Giovanni.~";
            21:
                "~Ah, Verdi!~^
                ^~Indeed, sir. And tomorrow night we're doing Figaro,
                    another, ahem, Mozart gem.~";
            22: give self general;
                "You decide a dignified silence is your best response.";
            30:
                "~It's very dark in the bar!~^
                ^~I'm sorry to hear that, sir.
                    I believe there was a problem earlier.~";
            31:
                "~Yes, I almost stumbled over something on the floor.~^
                ^~I think you'll find things are now back to normal.~";
            default: self.talk_number = 0;
            }
        ],
  has   animate male;
⋮
[ TalkSub;
        if (noun == player) "Nothing you hear surprises you.";
        if (RunLife(noun,##Talk) ~= 0) rfalse;
        "At the moment, you can't think of anything to say.";
        ];

Verb    'talk' 't//' 'converse' 'chat' 'gossip'
        * 'to'/'with' creature       -> Talk
        * creature                   -> Talk;
```

Here's the first part in action:

```
>TALK TO THE USHER
"Would you mind hanging up my cloak?"

"With pleasure, sir. There you are now."

>AGAIN
"Thank you very much."

"You're most welcome. Looks like a bad night out there?"

>AGAIN
"Yes, it's raining quite hard at the moment."

"Well, you'll be warm and dry in here. Enjoy the opera."
```

There's more on conversation to follow.

Intro ▸ Basics ▸ Silent interaction ▸ **Conversation** ▸ Conversation packages ▸ Orders ▸ Movement

Roger Firth's IF pages

InfAct -- about Inform NPCs

It's probably occurred to you by now that this conversation stuff is tricky. The occasional two-line interchange under controlled circumstances isn't too bad, but if you're contemplating anything approaching free-flowing give'n'take, you're in for a hard slog. Here are some library packages which may slightly ease the burden.

PhTalkOO.h

The **PhTalkOO.h** package gives you the simple menu-based conversational system which was at the heart of Adam Cadre's award-winning Photopia. It's quite easy to use; your NPC must inherit from the **Character** class and three new properties are needed, with most of the effort going into **Respond**:

```
Object  usher "gentleman usher" cloakroom
  class Character
  with  name 'usher' 'gentleman' 'gentle' 'man',
        description "The usher is smartly uniformed.",
        InitQuips [; self.QuipsOn(3, 0,1,2); ],
        SayQ [ q; switch (q) {
            0:  "Hello.";
            1:  "Nice uniform, mate. Just out the navy, are you?";
            2:  "Blimey, only one hook?";

            3:  "Please hang up my cloak.";
            4:  "Thanks very much.";

            5:  "What's tonight's opera?";
            6:  "Ah, Verdi.";

            7:  "Not many people around?";
            8:  "True enough - it's a nasty night to be out.";
            } ],
        Respond [ q; switch (q) {
            0:  self.QuipsOff(3, 0,1,2); self.QuipsOn(3, 3,5,7);
                "~Good evening.~";
            1:  self.QuipsOff(3, 0,1,2); self.QuipsOn(3, 3,5,7);
                "The usher smiles politely but doesn't reply.";
            2:  self.QuipsOff(3, 0,1,2); self.QuipsOn(3, 3,5,7);
                "~I'm afraid the others have been mislaid.~";

            3:  self.QuipOff(q); self.QuipOn(q+1);
                move cloak to hook; give cloak ~worn;
                "~Certainly.~";
            4:  self.QuipOff(q);
                "~Think nothing of it - it's what I'm here for.~";

            5:  self.QuipOff(q); self.QuipOn(q+1);
                "~It's Don Giovanni.~";
            6:  self.QuipOff(q);
                "~We're doing Figaro tomorrow - another, ahem, Mozart gem.~";

            7:  self.QuipOff(q); self.QuipOn(q+1);
                "~Indeed not - it must be the weather.~";
            8:  self.QuipOff(q);
                "~I can see that from your cloak.~";
            } ],
  has   animate male;
```

That's virtually all you need do, apart from invoking **InitQuips** from your **Initialize()** routine. Here's what it looks like:

```
>TALK TO THE USHER
What would you like to say?

[1] Hello.
[2] Nice uniform, mate. Just out the navy, are you?
[3] Blimey, only one hook?

Select an option or 0 to say nothing >> 3

"I'm afraid the others have been mislaid."

>AGAIN
What would you like to say?

[1] Please hang up my cloak.
[2] What's tonight's opera?
[3] Not many people around?

Select an option or 0 to say nothing >> 2

"It's Don Giovanni."
```

```
>AGAIN
What would you like to say?

[1] Please hang up my cloak.
[2] Ah, Verdi.
[3] Not many people around?

Select an option or 0 to say nothing >> 2

"We're doing Figaro tomorrow - another, ahem, Mozart gem."

>AGAIN
What would you like to say?

[1] Please hang up my cloak.
[2] Not many people around?

Select an option or 0 to say nothing >> 1

"Certainly."

>AGAIN
What would you like to say?

[1] Thanks very much.
[2] Not many people around?

Select an option or 0 to say nothing >> 1

"Think nothing of it - it's what I'm here for."

>AGAIN
What would you like to say?

[1] Not many people around?

Select an option or 0 to say nothing >> 1

"Indeed not - it must be the weather."

>AGAIN
What would you like to say?

[1] True enough - it's a nasty night to be out.

Select an option or 0 to say nothing >> 1

"I can see that from your cloak."

>AGAIN
You can't think of anything in particular to say.
```

Converse.h

Converse.h is another menu-based system, but in this case the menus are more formal (and, I think, more intrusive). There'a quite a bit to set up, at the head and foot of the game (including three other packages to be included first):

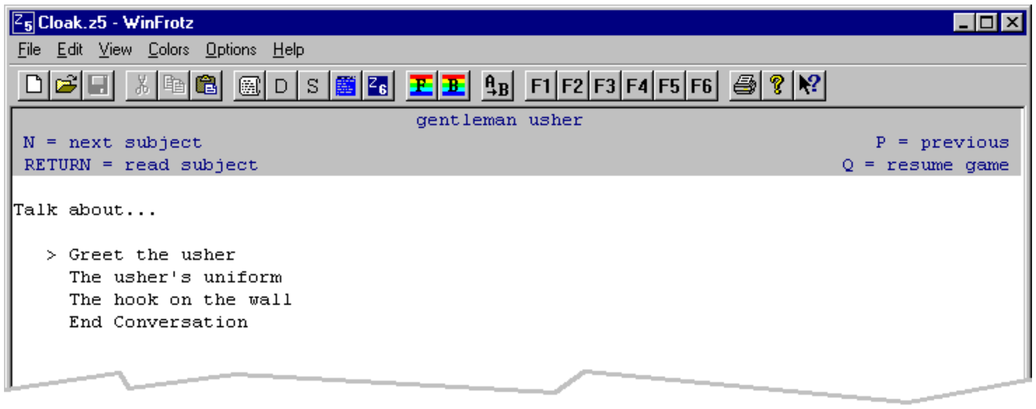
```
Constant Story      "Cloak of Darkness";
Constant Headline   "^A basic IF demonstration.^";
Constant MANUAL_PRONOUNS;
Constant MAX_SCORE  2;

Replace  DoMenu;
Replace  LowKey_Menu;

Include "Parser";
Include "VerbLib";

Constant ALTMENU_PROP_STUBS;
Constant CONVERSE_NO_GRAMMAR;
Constant CONVERSE__TX = "Talk about...";
Include "Utility";
Include "DoMenu";
Include "AltMenu";
Include "Converse";
⋮
Verb 'talk' 'converse' 'interview'
        * creature                  -> Converse
        * 'with'/'to' creature      -> Converse;
```

and for the NPC himself:

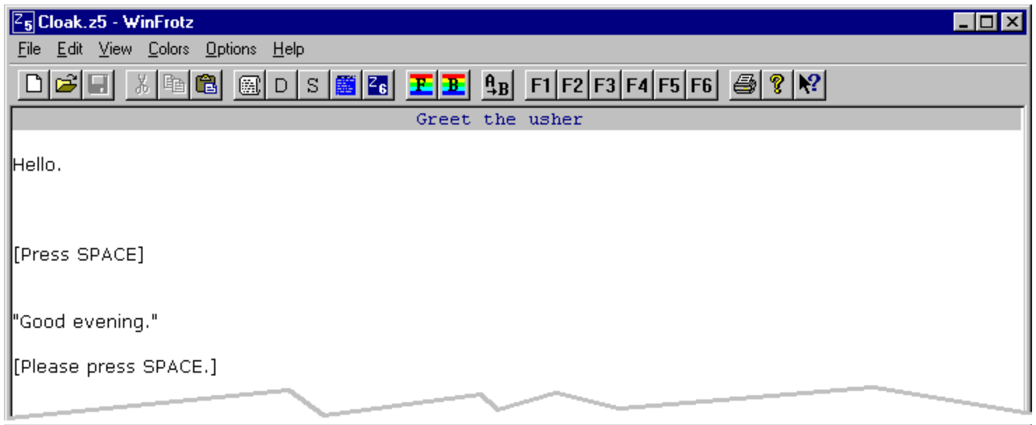
```
Object  usher "gentleman usher" cloakroom
  class Conversor
  with  name 'usher' 'gentleman' 'gentle' 'man',
        description "The usher is smartly uniformed.",
        conversation cc_usher,
  has   animate male;

Conversation_controller cc_usher;

Topic   -> t0 "Greet the usher"
  with  playerpart "Hello.",
        convpart [;
            usher.remconv(t0); usher.remconv(t1); usher.remconv(t2);
            usher.putconv(t3); usher.putconv(t5); usher.putconv(t7);
            "~Good evening.~"; ];
Topic   -> t1 "The usher's uniform"
  with  playerpart "Nice uniform, mate. Just out the navy, are you?",
        convpart [;
            usher.remconv(t0); usher.remconv(t1); usher.remconv(t2);
            usher.putconv(t3); usher.putconv(t5); usher.putconv(t7);
            "The usher smiles politely but doesn't reply."; ];
Topic   -> t2 "The hook on the wall"
  with  playerpart "Blimey, only one hook?",
        convpart [;
            usher.remconv(t0); usher.remconv(t1); usher.remconv(t2);
            usher.putconv(t3); usher.putconv(t5); usher.putconv(t7);
            "~I'm afraid the others have been mislaid.~"; ];

Topic   t3 "Your cloak"
  with  playerpart "Please hang up my cloak.",
        convpart [; usher.remconv(self); usher.putconv(t4);
            move cloak to hook; give cloak ~worn;
            "~Certainly.~"; ];
Topic   t4 "His reply"
  with  playerpart "Thanks very much.",
        convpart [; usher.remconv(self);
            "~Think nothing of it - it's what I'm here for.~"; ];

Topic   t5 "The opera"
  with  playerpart "What's tonight's opera?",
        convpart [; usher.remconv(self); usher.putconv(t6);
            "~It's Don Giovanni.~"; ];
Topic   t6 "His reply"
  with  playerpart "Ah, Verdi.",
        convpart [; usher.remconv(self);
            "~We're doing Figaro tomorrow - another, ahem, Mozart gem.~"; ];

Topic   t7 "The lack of people"
  with  playerpart "Not many people around?",
        convpart [; usher.remconv(self); usher.putconv(t8);
            "~Indeed not - it must be the weather.~"; ];
Topic   t8 "His reply"
  with  playerpart "True enough - it's a nasty night to be out.",
        convpart [; usher.remconv(self);
            "~I can see that from your cloak.~"; ];
```

At run-time, the TALK verb switches from normal display to a full screen menu; you remain within the menu interface until you end the conversation, whereupon the normal screen returns (with no evidence of what has taken place). This doesn't show up well on a normal transcript, so I'll give a couple of screen shots:

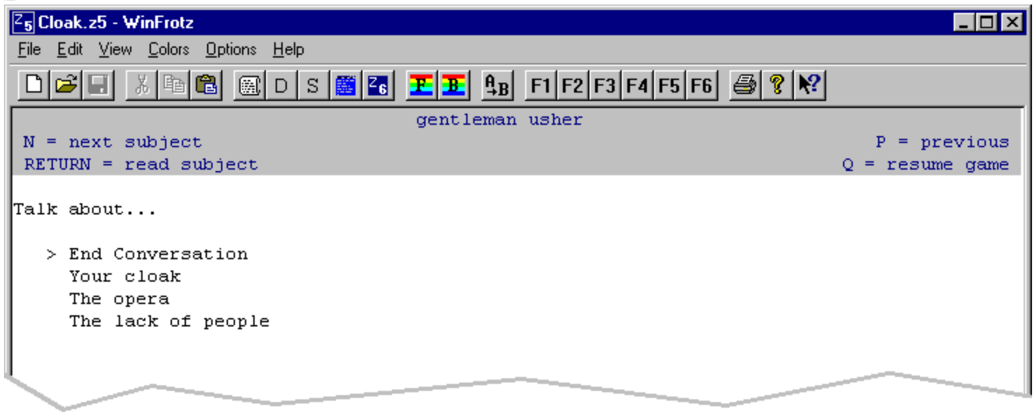
```
You can see a gentleman usher here.

>TALK TO THE USHER
```



```
                          gentleman usher
 N = next subject                                      P = previous
 RETURN = read subject                              Q = resume game

Talk about...

   > Greet the usher
     The usher's uniform
     The hook on the wall
     End Conversation
```



```
                    Greet the usher

Hello.


[Press SPACE]


"Good evening."

[Please press SPACE.]
```



```
                    gentleman usher
 N = next subject                              P = previous
 RETURN = read subject                    Q = resume game

Talk about...

   > End Conversation
     Your cloak
     The opera
     The lack of people
```

```
You can see a gentleman usher here.

>
```

On to giving orders.

[Intro](#) ▸ [Basics](#) ▸ [Silent interaction](#) ▸ [Conversation](#) ▸ **Conversation packages** ▸ [Orders](#) ▸ [Movement](#)

Roger Firth's IF pages

InfAct -- about Inform NPCs

This casual chatting with your NPC is all very well, but it's a bit, well, *passive*. There are times when you'd like to have him do as he's told.

Direct orders

The conventional way of getting an NPC to perform some action on your bidding is to use a form like USHER, GO EAST. The NPC's **orders** property is designed to handle this.

```
Object  usher "gentleman usher" cloakroom
  with  name 'usher' 'gentleman' 'gentle' 'man',
        description "The usher is smartly uniformed.",
        orders [;
        Go:
            "~Sorry - I mustn't leave the cloakroom.~";
        Take:
            if (noun ~= cloak) "~Thank you, but no.~";
            move cloak to hook;  give cloak ~worn;
            "~I'll just hang it on here for you.~";
        Drop:
            move noun to location;
            print_ret (The) self, " drops ", (the) noun, " on the floor.";
        Give:
            move noun to second;
            print_ret (The) self, " hands over ", (the) noun, ".";
        NotUnderstood:
            "~Pardon?~";
        default:
            "~I think not.~";
            ],
  has   animate male;
```

This code is really a bit too simplistic; for example, there ought to be a check that the usher is actually holding the object before he Drops or Gives it. (The Library doesn't do much validation here; ensuring sensible behaviour is down to you.) However, for our purposes, it's enough to support a little give and take.

```
>USHER,TAKE THE CLOAK
"I'll just hang it on here for you."

>INVENTORY
You are carrying nothing.

>USHER,DROP THE CLOAK
The gentleman usher drops the velvet cloak on the floor.

>ASK USHER FOR THE CLOAK
The gentleman usher hands over the velvet cloak.
```

If your NPC doesn't have an **orders** property, or has one which returns false, then the library passes the order to the **life** property. This is only for compatibility with older games; you're recommended always to deal with orders by providing a comprehensive **orders** property.

AskTellOrder

The useful package AskTellOrder.h extends the library syntax to encompass orders like ASK USHER TO TAKE THE CLOAK or TELL USHER TO DROP THE CLOAK. Cleverly, it does this by modifying the input buffer so that these two orders become USHER,TAKE THE CLOAK and USHER,DROP THE CLOAK; all you need do is Include the package.

```
>ASK USHER TO TAKE THE CLOAK
"I'll just hang it on here for you."

>TELL USHER TO DROP THE CLOAK
The gentleman usher drops the velvet cloak on the floor.

>USHER,GIVE ME THE CLOAK
The gentleman usher hands over the velvet cloak.
```

Questions

A natural extension of the **orders** syntax outlined here is the handling of questions: USHER, WHAT|WHERE|WHO IS A|THE ... One approach to this follows that suggested for conversational topics on a previous segment:

```
Object  Queries "conversational queries";
Object  -> q_cloak      with name 'handsome' 'dark' 'black' 'velvet' 'cloak';
Object  -> q_opera      with name 'opera' 'show' 'performance';
Object  -> q_time       with name 'time';
Object  -> q_bar        with name 'bar';
Object  -> q_foyer      with name 'foyer';

[ QueryScope; switch(scope_stage) {
        1: rfalse;
        2: ScopeWithin(Queries); rtrue;
        3: "At the moment, even the simplest questions are confusing.";
        } ];

[ WhatIsSub; "You seem to be talking to yourself."; ];

[ WhereIsSub; "You seem to be talking to yourelf."; ];

Verb    'what' 'who'
        * 'is'/'are' scope=QueryScope       -> WhatIs;
Verb    'where'
        * 'is'/'are' scope=QueryScope       -> WhereIs;
```

With the query objects and the grammar in place, the **orders** handlers are straightforward:

```
Object  usher "gentleman usher" cloakroom
  with  name 'usher' 'gentleman' 'gentle' 'man',
        description "The usher is smartly uniformed.",
        orders [;
            ⋮
        WhatIs:
            switch(noun) {
            q_opera: "~Don Giovanni tonight.~";
            q_time: print "~It's about ";
                LanguageTimeOfDay(the_time/60, the_time%60); ".~";
            }
            "~I'm afraid you've got me there.~";
        WhereIs:
            switch(noun) {
            q_cloak: if (IndirectlyContains(location,cloak))
                "~Why, here it is!~";
            q_bar,q_foyer: "~It's through there to the east.~";
            }
            "~I'm sorry, but I don't know.~";
            ⋮
        ],
  has   animate male;
```

Look at the `WhoWhat.h` package for further ideas.

In the last segment, we take off the brakes.

Intro ▸ Basics ▸ Silent interaction ▸ Conversation ▸ Conversation packages ▸ **Orders** ▸ Movement

InfAct -- about Inform NPCs

Some NPCs are intrinsically rooted to the spot; perhaps their role is such that this follows naturally ("Sorry, guv, I've gotta stay guardin' this 'ere priceless jewel." or "You want fries with that?"), or maybe their situation requires it ("I say, what beastly rotten luck to be stuck in bed with a broken leg, right in the middle of the bally old tennis season!"). More often than not, though, the game would appear more realistic if your NPC had the semblance of independent mobility. Like conversation, this is hard to do well, so there are a couple of packages which you should consider. Neither, by the way, is here shown working at full capacity -- they both support important features like arrival/departure processing and the handling of doors which are beyond our current scope.

MoveClass.h

The `MoveClass.h` package is pretty easy to set up. Your rooms must be of class **Room** and your NPC of class **MoveClass**; beyond that, your basic choices are about movement types:

- **NO_MOVE** -- the NPC stays where he is
- **RANDOM_MOVE** -- he roams at random
- **AIMED_MOVE** -- he heads towards a named room
- **PRESET_MOVE** -- he follows a preset path (N, N, E, N, W...)

Here's what it takes to set up random movement; this for the NPC:

```
Object  usher "gentleman usher" cloakroom
  class MoveClass
  with  name 'usher' 'gentleman' 'gentle' 'man',
        description "The usher is smartly uniformed.",
        walkon "arrives",
        walkoff "walks off",
  has   animate male;
```

together with this in your `Initialise()` routine:

```
[ Initialise;
    location = foyer;
    move cloak to player;
    give cloak worn;
    StartDaemon(usher);
    NPC_Path(usher, RANDOM_MOVE, 80);
   "^^Hurrying through the rainswept November night, you're glad to see the
    bright lights of the Opera House. It's surprising that there aren't more
    people about but, hey, what do you expect in a cheap demo game...?^^";
    ];
```

And here he comes:

```
Foyer of the Opera House
You are standing in a spacious hall, splendidly decorated in red and gold, with
glittering chandeliers overhead. The entrance from the street is to the north,
and there are doorways south and west.

The gentleman usher arrives.

>WAIT
Time passes.

>WAIT
Time passes.

The gentleman usher walks off to the south.
```

Here, instead, he's moving to and fro between the cloakroom and the bar, with a short pause at each end:

```
Object  usher "gentleman usher" cloakroom
  class MoveClass
  with  name 'usher' 'gentleman' 'gentle' 'man',
        description "The usher is smartly uniformed.",
        walkon "arrives",
        walkoff "walks off",
        npc_arrived [;
            StartTimer(self, 2);
            NPC_Path(self, NO_MOVE);
            ],
        time_left 0,
        time_out [;
            if (parent(self) == bar) NPC_Path(self, AIMED_MOVE, cloakroom);
            else NPC_Path(self, AIMED_MOVE, bar);
            ],
      has   animate male;
```

Just hang around and you'll see him:

```
Foyer of the Opera House
You are standing in a spacious hall, splendidly decorated in red and gold, with
glittering chandeliers overhead. The entrance from the street is to the north,
and there are doorways south and west.

>WAIT
Time passes.

The gentleman usher arrives.

>WAIT
Time passes.

The gentleman usher walks off to the south.

>WAIT
Time passes.

>WAIT
Time passes.

>WAIT
Time passes.

The gentleman usher arrives.

>WAIT
Time passes.

The gentleman usher walks off to the west.
```

Follower.h

The **`Follower.h`** package enables you to FOLLOW an NPC who's moving around the game. It's nicely compatible with **`MoveClass.h`**; hardly any extra code is needed:

```
Include "Follower";
⋮
Object  usher "gentleman usher" cloakroom
  class MoveClass FollowClass
  with  name 'usher' 'gentleman' 'gentle' 'man',
        ⋮
```

NPC_Engine.h

The **`NPC_Engine.h`** package is even easier to get going. Its capabilities match those of **`MoveClass.h`** (except that there seems to be no support for randomly wandering around), and it includes some tasty extras listed below. Your rooms must be of class **NPC_Room** and your NPC of class **NPC_Engine**, and you get to choose between moving to a room, moving along a path, or standing still. So, here's more shuttling to the bar and back, again with a small pause at each end:

```
Object  usher "gentleman usher" cloakroom
  class NPC_Engine
  with  name 'usher' 'gentleman' 'gentle' 'man',
        description "The usher is smartly uniformed.",
        npc_arrived [ x;
            if (x == bar) NPC_Schedule(self, cloakroom, 2);
            else NPC_Schedule(self, bar, 2);
            ],
  has   animate male;
⋮
[ Initialise;
    location = foyer;
    move cloak to player;
    give cloak worn;
    NPC_Initialise();
    NPC_Path(usher, bar);
   "^^Hurrying through the rainswept November night, you're glad to see the
    bright lights of the Opera House. It's surprising that there aren't more
    people about but, hey, what do you expect in a cheap demo game...?^^";
    ];
```

The NPC behaves much as he did before:

```
Foyer of the Opera House
You are standing in a spacious hall, splendidly decorated in red and gold, with
glittering chandeliers overhead. The entrance from the street is to the north,
and there are doorways south and west.

>WAIT
Time passes.
The gentleman usher is walking past you.

>WAIT
Time passes.
The gentleman usher heads off to the south.

>WAIT
Time passes.

>WAIT
Time passes.

>WAIT
Time passes.
The gentleman usher is walking past you.

>WAIT
Time passes.
The gentleman usher heads off to the west.
```

In addition to its basic movement capabilities, this package also comes with

- built-in support for the FOLLOW verb
- being able to ask one NPC about the location of another
- complex descriptions of NPCs moving in the distance
- watching NPCs visible through windows

which make it well worth a serious look.

Would you believe it? we've run out of time. Tune in again next week for...

Intro ▸ Basics ▸ Silent interaction ▸ Conversation ▸ Conversation packages ▸ Orders ▸ **Movement**