Chapter 8: Ticks, scenes, and progress

(Timed code • Cutscenes • The intro • Keeping score • The status bar • Game over • Choice mode)

Timed code

As we've seen earlier (Stopping and ticking), the standard library measures time in *ticks*. One tick corresponds to one action. The library makes a multi-query to the predicate (on every tick) on every tick. By default, this predicate contains a rule that in turn makes a multi-query to (on every tick in $), with the current room as parameter.

To add flavour text to a location, you can combine this mechanism with a select statement:

```
(on every tick in
#library)
```

```
(on every tick in #library)
 (select)
 Somebody tries to hold back a sneeze.
 (or)
 You hear the rustle of pages turned.
 (or)
 The librarian gives you a stern look.
 (or)
 (or)    %% Don't print on every single turn.
 (or)
 (or)
 (at random)
```

[Copy to clipboard]

For fine-grained control, you can use a global variable to implement *timed events*:

```
(global variable
(dragon's anger
```

```
(global variable (dragon's anger level 0))

(narrate entering #lair)
 (now) (dragon's anger level 0)
 (fail)    %% Proceed with the default '(narrate entering $)' rule.

(on every tick in #lair)
 (#dragon is #in #lair)    %% Only proceed if the dragon is here.
 (dragon's anger level $Anger)
 (narrate dragon's anger $Anger)
 ($Anger plus 1 into $NewAnger)
 (now) (dragon's anger level $NewAnger)

(narrate dragon's anger 0)
 The dragon gives you a skeptical look.
(narrate dragon's anger 1)
 The dragon huffs and puffs.
(narrate dragon's anger 2)
 The dragon looks at you with narrowed eyes.
(narrate dragon's anger 3)
 The dragon roars! You'd better get out.
(narrate dragon's anger 4)
 The dragon almost hits you with a burst of flame. You flee.
 (enter #outsideLair)
```

[Copy to clipboard]

To model a scene that plays out in the background for several moves, use a global flag and a tick handler:

```
(perform [read
#notice])
```

```
(perform [read #notice])
 Auction! Today! In the marketplace!
 (now) (auction scene is running)
```

```
(on every tick in #marketplace)
 (auction scene is running)
 "Who can give me five dollars for this lovely
 (select) spatula (or) glass bead (or) stuffed wombat (at random)?"
 shouts the auctioneer at the top of his lungs.

(perform [wave])
 (current room #marketplace)
 (auction scene is running)
 Just as you are about to place a bid, an unexpected thunderstorm
 emerges from a gaping plot hole. "Auction suspended", says the
 auctioneer.
 (now) ~(auction scene is running)
```
[Copy to clipboard]

Cutscenes

In their simplest form, cutscenes are just large blocks of text and perhaps a couple of modifications to the object tree. As such, they can appear anywhere in the program. If a cutscene is triggered by an action handler or a tick callback, it is customary to end the scene with (stop) or (tick) (stop), to inhibit any subsequent actions mentioned in the player's input. For instance:

```
(perform [pull #handle])

(perform [pull #handle])
 You grab the handle of the machine, and hesitate for a moment. Is
 this really safe?

 (par)

 But you have no choice. You pull the handle. Sparks hum in the air
 as you are sucked into the vortex of the machine.

 (par)

 You find yourself in a barn.

 (move player to #on #haystack)
 (try [look])
 (tick) (stop)
```
[Copy to clipboard]

If the cutscene can be triggered in multiple ways, put it in a separate predicate and query that from as many places in the code as you wish.

To prevent a cutscene from occurring twice, use a global flag:

```
(perform [pull #handle])

(perform [pull #handle])
 (if) (have teleported to barn) (then)
 Nothing happens when you pull the handle.
 (else)
 (teleport to barn cutscene)
 (endif)

(teleport to barn cutscene)
 ...
 (now) (have teleported to barn)

 (stop)
```
[Copy to clipboard]

The intro

When a story begins, the standard library queries the (intro) predicate, which story authors are encouraged to override.

The default implementation of (intro) just prints the *story banner* by querying (banner). The banner includes version information for the compiler and the standard library. By convention, stories should print the banner at some point during play. With Dialog, there is no formal requirement to print the banner at all, but it is helpful to the community and to your future self, and it looks professional.

```
(intro)
	"In medias
```

```
(intro)
 "In medias res?" exclaimed the broad-shouldered Baron furiously.
 "We find it preposterously cliché!"

 (banner)

 (try [look])
```

[Copy to clipboard]

The (banner) predicate calls out to (additional banner text), which is empty by default but can be overridden to include e.g. a subtitle, co-credit, or dedication.

Keeping score

The player's progress can be tracked by a global score variable. This feature needs to be enabled, by including the following rule definition somewhere in the story:

```
(scoring enabled)
```

```
(scoring enabled)
```

[Copy to clipboard]

For scored games, the current score is displayed in the status bar.

The global variable is called (current score $).

Points can be added to the score with (increase score by $), and subtracted with (decrease score by $). These predicates fail if the score would end up outside the valid range of integers in Dialog, which is 0–16383 inclusive.

After every move, the standard library will mention if the score has gone up or down, and by how much, unless the player has disabled this feature using NOTIFY OFF.

If you know what the maximum score is, you can declare it:

```
(maximum score 30)
```

```
(maximum score 30)
```

[Copy to clipboard]

When declared, the maximum score is mentioned by the default implementation of the SCORE command, in the status bar, as well as by the (game over $) predicate. It does not affect the operation of (increase score by $).

The status bar

It is straightforward to supply your own, custom status bar. Define a rule for the predicate (redraw status bar), and make use of the status area functionality built into the Dialog programming language.

The standard library defines (status headline), which can be used to print the location of the player character in the usual way. That would normally be the current room header, followed by something like “(on the chair)” if the player character is the child of a non-room object. But if the player character is in a dark location, control is instead passed to (darkness headline), which usually prints “In the dark”.

```
%% A thicker status
bar with the name of
```

```
%% A thicker status bar with the name of the current player in the upper right corner.

(style class @status)
  height: 3em;

(style class @playername)
  float: right;
  width: 20ch;
  margin-top: 1em;

(style class @roomname)
  margin-top: 1em;

(redraw status bar)
  (status bar @status) {
  (div @playername) {
```

```
  (current player $Player)
  (name $Player)
  }
  (div @roomname) {
  (space 1) (status headline)
  }
  }
```
[Copy to clipboard]

Game over

The library provides a predicate called (game over $). Its parameter is a closure containing a final message, which the library will print in bold text, enclosed by asterisks. Then it will:

- Invoke (game over status bar) which sets the status bar to “Game over”, unless you override it.
- Report the final score (if scoring is enabled), and the maximum score (if one has been declared).
- Enter an infinite loop where the player is asked if they wish to RESTART, RESTORE, UNDO the last move, or QUIT.

Here is an example of a (very small) cutscene that ends the game:

```
(perform [eat
#apple])
```

```
(perform [eat #apple])
 The apple is yummy. You feel that your mission has come to an end.
 (game over { You are no longer hungry. })
```
[Copy to clipboard]

A fifth option can be added to the game-over menu: AMUSING. First, add the option to the menu with the following rule definition:

```
(amusing enabled)
```

```
(amusing enabled)
```
[Copy to clipboard]

Then, implement a predicate called (amusing) that prints a list of amusing things the player might want to try:

```
(amusing)
      (par)
```

```
(amusing)
 (par)
 Have you tried...
 (par)
 (space 10) ...eating the transmogrifier? (line)
 (space 10) ...xyzzy? (line)
```
[Copy to clipboard]

Custom options can be added to the menu by defining rules for (game over option). A multi-query will be made to this predicate, and the output is supposed to end with a comma. For instance:

```
(game over option)
      read a NOTE by
```

```
(game over option)
 read a NOTE by the author,
```
[Copy to clipboard]

And here is how to specify what happens when the user types the given word:

```
(parse game over
[note])
```

```
(parse game over [note])
 (par)
 Thanks for playing!
 (line)
 -- (space) The Author
```

(par)
[Copy to clipboard]

Choice mode

As a complement to the parser, the Dialog standard library offers a *choice mode*, where the player navigates a set of *nodes* (text passages) by choosing from explicit lists of options. Choice mode can be used for interactive cutscenes, conversations, mini-games, or even as the primary mode of interaction of a game. The author is free to switch between parser-based and choice-based interaction at any time, as behoves the story.

Nodes are represented by ordinary Dialog objects. In the simplest mode of operation, each node has a label and some display-text, and offers a set of links to other nodes:

```
(intro)
(activate node
```

```
(intro)   (activate node #start)

#start
(disp *)  You extend your wings. A warm, buzzy feeling spreads through your body
          as you leave the hive.
(* offers #rosebush)
(* offers #poppies)

#rosebush
(label *) Follow a scent of roses.
(disp *)  You hover for a while near the pink rosebush.
(* offers #poppies)

#poppies
(label *) Follow a scent of poppies.
(disp *)  You circle a patch of poppies by the pondside.
(* offers #rosebush/#pond)

#pond
(label *) Approach the pond.
(disp *)  You flutter across the water, enjoying the sweet bouquet of water lilies.
(* offers #poppies)

#land
(label *) Return back home.
(disp *)  After an impeccable landing, you find yourself back at the hive.
          (game over { A day well spent! })
(#rosebush/#poppies/#pond offers *)
```

[Copy to clipboard]

To select choice mode—or remain in choice mode but force a transition to a different node—make a query to (activate node $) with the desired node object as parameter. To select parser mode, make a query to (activate parser). Be aware that both of these predicates invoke (stop), thereby effecting an immediate return to the main loop.

In the main loop, if choice mode is on, the library determines what nodes are reachable from the currently active node, and prints a numbered list of their labels. If the player types one of the numbers (or clicks one of the labels, if library links are enabled and the interpreter supports them) then the corresponding node is activated. Otherwise, the input is parsed in the usual way. By default, all in-world actions are disabled in choice mode; only commands (e.g. UNDO, SAVE) work. Here is an example session:

```
You extend your wings. A warm, buzzy feeling spreads through your body as you leave the hive.

1. Follow a scent of roses.
2. Follow a scent of poppies.
> 2
Follow a scent of poppies.

You circle a patch of poppies by the pondside.

1. Follow a scent of roses.
2. Approach the pond.
3. Return back home.
> undo
Undoing the last turn.

1. Follow a scent of roses.
```

```
2. Follow a scent of poppies.
> look
(That action is currently disabled.)

1. Follow a scent of roses.
2. Follow a scent of poppies.
>
```

Exposed and unexposed nodes

When a node has been activated at least once, it is considered *exposed*, and the flag ($ is exposed) is set. This is handy for putting conditions on the links between nodes:

```
...
```

```
...

#land
(label *) Return back home.
(disp *)  After an impeccable landing, you find yourself back at the hive.
          (game over { A day well spent! })
(#rosebush/#poppies/#pond offers *)
          (#rosebush is exposed)
          (#poppies is exposed)
```

[Copy to clipboard]

In the above example, the “Return back home” option will only show up after the player has visited both the poppies and the rosebush.

The access predicate ($ is unexposed) is the negation of ($ is exposed).

A node can have an *initial label* which is shown instead of the regular label while the node is unexposed:

```
#poppies
(initial label *)
```

```
#poppies
(initial label *) Follow a scent of poppies.
(label *)         Return to the patch of poppies.
(disp *)          You circle a patch of poppies by the pondside.
(* offers #rosebush/#pond)
```

[Copy to clipboard]

The default implementation of (initial label $) simply passes control to (label $).

Conditional labels

The currently active node is indicated by the global variable (current node $). In parser mode, (current node $) is unset. You shouldn't update this variable directly—use (activate node $) and (activate parser)—but you may query it.

Labels can have conditions. In particular, they can depend on the current node:

```
(label #poppies)
(current node
```

```
(label #poppies) (current node #rosebush)
                 Leave the rosebush and follow a scent of poppies.
(label #poppies) (current node #pond)
                 Leave the pond and return to the poppies.
(label #poppies) Follow a scent of poppies.
```

[Copy to clipboard]

Dead ends and sticky nodes

A *dead end* is a node that doesn't offer any further choices. When the current node is a dead end, control flows back to the most recent node by default. In addition, the dead-end node becomes *unavailable*. Unavailable nodes do not show up in option lists, even if they are declared using ($ offers $).

```
#poppies-collect
(#poppies offers *)
```

```
#poppies-collect
(#poppies offers *)
(label *) Collect nectar from the poppies.
(disp *)  Yum!
```

[Copy to clipboard]

```
You extend your wings. A warm, buzzy feeling spreads through your body as you
leave the hive.

1. Follow a scent of roses.
2. Follow a scent of poppies.
> 2
Follow a scent of poppies.

You circle a patch of poppies by the pondside.

1. Follow a scent of roses.
2. Approach the pond.
3. Collect nectar from the poppies.
> 3
Collect nectar from the poppies.

Yum!

You circle a patch of poppies by the pondside.

1. Follow a scent of roses.
2. Approach the pond.
>
```

Sometimes you'll want a dead-end node that remains available in choice-listings even after it has been exposed. Just mark the node as *sticky*:

```
#poppies-collect
(sticky *)
```

```
#poppies-collect
(sticky *)
```

[Copy to clipboard]

Flow and converging paths

It is possible to specify a different target for a dead-end node using ($ flows to $):

```
#poppies-collect
(#poppies offers *)
```

```
#poppies-collect
(#poppies offers *)
(label *) Collect nectar from the poppies.
(disp *)  Yum!
          (par)
          A sudden gust of wind throws you in the direction of the pond.
(* flows to #pond)
```

[Copy to clipboard]

This mechanism can be used to implement *converging paths*, where a single node can be reached in several ways, with different display-text for every path. The common node itself can have a blank display-text, and only serve as an anonymous bag of subsequent choices:

Note: In this example, #poppies-collect and #rosebush-collect are declared sticky. This prevents the game from becoming unwinnable if the player revisits a plant after collecting both kinds of nectar.

```
(intro)
(activate node
```

```
(intro)          (activate node #start)

#start
(disp *)         You extend your wings. A warm, buzzy feeling spreads through your body
                 as you leave the hive.
(* offers #rosebush)
(* offers #poppies)

#rosebush
(initial label *) Follow a scent of roses.
(label *)        Return to the rosebush.
(disp *)         You hover for a while near the pink rosebush.
(* offers #poppies)

#poppies
(initial label *) Follow a scent of poppies.
(label *)        Return to the patch of poppies.
(disp *)         You circle a fragrant patch of poppies.
(* offers #rosebush)

#rosebush-collect
(#rosebush offers *)
(sticky *)
(label *)        Collect nectar from the rosebush.
(disp *)         Yum! Rose nectar!
(* flows to #collect-done)

#poppies-collect
(#poppies offers *)
(sticky *)
(label *)        Collect nectar from the poppies.
(disp *)         Yum! Poppy nectar!
(* flows to #collect-done)

#collect-done
(* offers #rosebush/#poppies/#land)

#land
(label *)        Return back home.
(disp *)         After an impeccable landing, you find yourself back at the hive.
                 (game over { A day well spent! })
```

[Copy to clipboard]

The default behaviour of automatically returning to the previous node is actually implemented as a fallback rule for the ($ flows to $) predicate.

Only a single level of node history is recorded, so automatic backtracking only works once. If the previous node also offers no choices, then the library dumps the player back into parser mode. If this is undesirable, always provide explicit ($ flows to $) links for nodes that might *turn into* dead ends. That is, nodes that offer choices initially, but where all of those choices may eventually go away.

The hub pattern

Another way to organize a choice-based sequence is to have a central hub node, and to selectively offer links based on which peripheral nodes have been exposed so far.

In the following example, the player can collect nectar once from each plant, and needs at least some nectar in order to proceed with the landing:

```
(intro)          You
extend your wings. A
```

```
(intro)          You extend your wings. A warm, buzzy feeling spreads through your body
                 as you leave the hive.
                 (activate node #hub)

#hub
(* offers #rosebush)
(* offers #poppies)

#rosebush
(sticky *)
```

```
(initial label *)  Follow a scent of roses.
(label *)          Return to the rosebush.
(disp *)           You (select) discover a (or) revisit the (stopping) pink rosebush.

#poppies
(sticky *)
(initial label *)  Follow a scent of poppies.
(label *)          Return to the patch of poppies.
(disp *)           You (select) find a (or) circle the (stopping) fragrant patch of poppies.

#rosebush-collect
(#hub offers *) (#rosebush is exposed)
(label *)          Collect nectar from the rosebush.
(disp *)           Yum! Rose nectar!

#poppies-collect
(#hub offers *) (#poppies is exposed)
(label *)          Collect nectar from the poppies.
(disp *)           Yum! Poppy nectar!

#land
(#hub offers *) (#rosebush-collect is exposed) (or) (#poppies-collect is exposed)
(label *)          Return back home.
(disp *)           After an impeccable landing, you find yourself back at the hive.
                   (game over { A day well spent! })
```

[Copy to clipboard]

Choice/parser integration

A *terminating* dead-end node has the side-effect of leaving choice mode. This is handy when integrating choice-based sequences into a larger game:

```
(current player
#player)

(current player #player)
(#player is #in #beehive)

#beehive
(room *)
(look *)  Honeycombs line every wall. The exit is due east.

(instead of [leave * #east])
          (activate node #start)

#start
(disp *)  You extend your wings. A warm, buzzy feeling spreads through your body
          as you leave the hive.
(* offers #rosebush)
(* offers #poppies)

#rosebush
(label *) Follow a scent of roses.
(disp *)  You hover for a while near the pink rosebush.

(* offers #poppies/#land)

#poppies
(label *) Follow a scent of poppies.
(disp *)  You circle a patch of poppies by the pondside.
(* offers #rosebush/#land)

#land
(label *) Return back home.
(disp *)  After an impeccable landing, you find yourself back at the hive.
(terminating *)
```

[Copy to clipboard]

Terminating nodes are implicitly sticky.

Nothing prevents you from letting arbitrary game objects double as nodes in choice mode. For instance, in the above example we could have used #beehive as the starting node, instead of introducing a separate #start object. This is

particularly handy when a game contains multiple choice-based sequences that are triggered in a similar way. For instance, conversations with non-player characters could be launched with a generic rule, such as:

```
(perform [talk to
(animate $NPC)])
```

```
(perform [talk to (animate $NPC)])
  (activate node $NPC)
```

[Copy to clipboard]

When the player types a number during choice mode, a query is made to (choose $). The default behaviour of this predicate—feel free to override it!—is to print the label of the object followed by a paragraph break, and then make a query to (activate node $). This predicate, in turn, performs some internal housekeeping, and *displays* the node using a predicate called (display $):

```
(display $Obj)
      (exhaust) { *
```

```
(display $Obj)
  (exhaust) { *(before disp $Obj) }
  (disp $Obj)
  (exhaust) { *(after disp $Obj) }
```

[Copy to clipboard]

So as a complement to the normal (disp $) rules, story authors can put header and footer material in (before disp $) and (after disp $), respectively.

After querying (display $), (activate node $) marks the node as exposed, and invokes (stop). But no query is made to (tick). Hence, by default, no in-game time passes in choice mode.

To change this, just add an (after disp $) rule with an explicit query to (tick):

```
(after disp $)   (tick)
```

```
(after disp $) (tick)
```

[Copy to clipboard]

Sometimes, it is more natural to advance time only at terminating nodes, i.e. just before the game transitions from choice mode to parser mode:

```
(after disp
(terminating $))
```

```
(after disp (terminating $))
    (tick)
```

[Copy to clipboard]

And here is a variant that also prints the current room description, sending a signal to the player that the game is now in parser mode:

```
(after disp
(terminating $))
```

```
(after disp (terminating $))
  (par)
  (try [look])
  (tick)
```

[Copy to clipboard]

Hybrid modes

The library also supports parser/choice hybrid modes, where certain actions are available in addition to the numbered choices. For instance, it could be useful to allow SHOW X TO Y from within a choice-based conversation.

By default, all actions are allowed in parser mode, while only commands are allowed in choice mode. Change this by adding rules to (allowed action $), e.g.:

```
(allowed action [look])
```

```
(allowed action [look])
```

[Copy to clipboard]

As an arbitrary example, you could allow LOOK from a subset of the nodes:

```
(allowed action
[look])
```

```
(allowed action [look])
 (current node $Node)
 ($Node is one of [#rosebush #poppies])
```

[Copy to clipboard]

Onwards to “Chapter 9: Non-player characters” • Back to the Table of Contents