Universitat de Girona
Escola Politècnica Superior

Treball final de màster

Estudi: Màster en Enginyeria Informàtica

Títol:

SISTEMA DE NARRATIVA PROCEDURAL PER A FICCIONS INTERACTIVES

Document: Memòria

Alumne: Ricard Galvany Méndez

Tutors: Gustavo Patow (UdG) i António Coelho (UPorto)
Departament: INFORMÀTICA, MATEMÀTICA APLICADA I ESTADÍSTICA
Àrea: LLENGUATGES I SISTEMES INFORMÀTICS

Convocatòria (mes/any): 09/2019

ÍNDICE

1 INTRODUCCIÓN

1.1 MOTIVACIÓN

En la industria del desarrollo de viejuegos, hasta hace muy poco tiempo, mayoritariamente se han utilizado mecanismos que prefijan el contenido narrativo. Esto hace que varios jugadores utilizando el mismo videojuego obtengan siempre el mismo resultado. O que un mismo jugador descarte rejugar un videojuego a causa de conocer como trascurrirá.

Por ejemplo, la popular aventura gráfica “The secret of the Monkey Island” de la compañía “Lucasfilm Games” (ver Figura 1.1), que basa gran parte de su éxito en un elaborado guión. A pesar de cierta variabilidad en lo referente a los diálogos con los personajes no jugables, no suele ser rejugada por perder el interés del jugador tras finalizarla por primera vez [SECRETMONKEY19].

Figura 1.1 - Inicio del videojuego “The secret of the Monkey Island”

Existen casos muy populares de sistemas que presentan narrativas que varia en función de las acciones del jugador, dentro de un contexto limitado, como “Dwarf Fortress” o “No Man's Sky”.

El primero de estos, “Dwarf Fortress”, es un juego para ordenadores sin coste alguno desarrollado por la compañía “Bay 12 Games”. Situado en un ambiente fantástico, tiene como objetivo explorar mazmorras y realizar construcciones. Dispone de un subsistema para la generación y distribución de manera procedural de los elementos básicos de las mazmorras [DWARF19] (ver Figuras 1.2 y 1.3).

Figura 1.2 – Versión para la tienda digial Steam de “Dwarf Fortress” con gráficos mejorados

Figura 1.3 – Una de las zonas del mapeado de “Dwarf Fortress” con gráficos ASCII

El segundo, denominado “No Man's Sky”, un videjuego de exploración espacial que dispone de un sistema procedural para la creación de mundos, con su flora y fauna, para que cada jugador desarrolle diferentes experiencias [NOMANSSKY19] (ver Figuras 1.4 y 1.5).

Figura 1.4 – Captura de una partida de “No Man's Sky” donde el personaje recorre un mundo generado proceduralmente

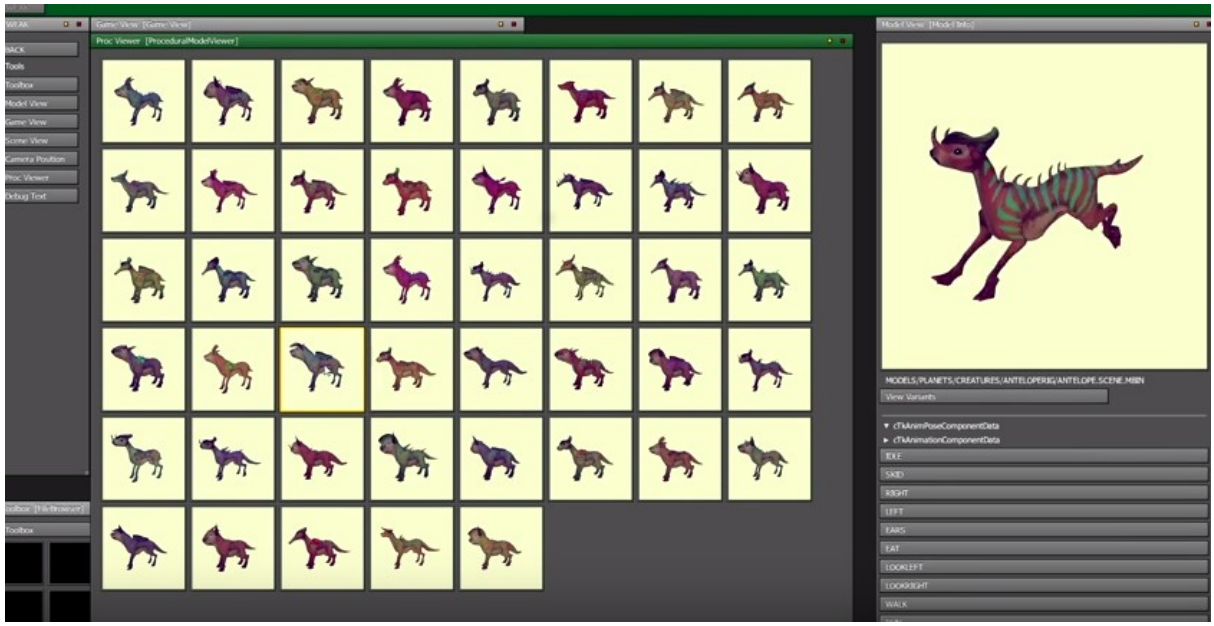

Figura 1.5 – Variaciones procedurales de la fauna disponible en el videojuego “No Man's Sky”

Desafortunamdamente generar contenido que pueda varir requiere de un tipo de sistema generador de narrativa que aún está en pañales en lo referente a su desarrollo [SHORTADAMS17].

1.2 OBJETIVO

El objetivo de este trabajo consistide en investigar sobre mecanismos para establecer un sistema en el que la narrativa contenga componentes dinámicos que permitan diferentes desarrollos narrativos en respuesta a las interacciones con el usuario, generando de esta manera historias diferentes en cada partida del juego.

1.3 PLANIFICACIÓN Y EJECUCIÓN

Partiendo de la idea inicial de desarrollar un sistema para implementar narrativa de forma procedural, en primer lugar se examinaron diferentes plataformas de desarrollo de ficciones interactivas en busca de una que se aproximara a las bases del proyecto.

Posteriormente se examinaron diferentes planificadores para encontrar uno que se adaptara al proyecto, y una vez escogido, se valoraron diferentes estrategias para incluirlo en la plataforma de desarrollo de ficciones interactivas.

Finalmente se generó una capa que envolvía todos los componentes y la controlaba, de forma que el desarrollador pudiera abstraerse de estos detalles y pudiera trabajar con la herramienta final.

2 PREVIOS

2.1 AVENTURAS CONVERSACIONALES

La aventura conversacional es un género de videojuegos, más común de ordenadores que de consola o arcades, en el que la descripción de la situación en la que se encuentra el jugador proviene principalmente de un texto. A su vez, el jugador debe teclear la acción a realizar. El juego interpreta la entrada -normalmente- en lenguaje natural, lo cual provoca una nueva situación y así sucesivamente. A veces existen gráficos en estos juegos, que sin embargo son tan sólo situacionales o que ofrecen ayuda complementaria en algunos casos, al estilo de las ilustraciones de un libro. El género de las aventuras gráficas surgió como evolución de las videoaventuras y las aventuras conversacionales, dejando estas últimas 'pasadas de moda' en Occidente. En Japón siguen estando muy presentes en la forma de novelas visuales, un género que se puede considerar sucesor de las aventuras conversacionales, aunque con características propias [AVENCON19].

La primera aventura conversacional se denominó *Adventure* [COLCAVE19], fue creada en el año 1975 por Will Crowther y Don Woods mediante el lenguaje de programación Fortran en un PDP-10 de la compañía DEC (ver figura 2.1). En la aventura el jugador toma el control del personaje principal mediante la introducción de órdenes a través del teclado para explorar una gran cueva, eludiendo peligros y buscando tesoros, para conseguir puntos.

Figura 2.1 - Adventure ejecutándose en un PDP-11/34 mediante una consola VT100

Esta aventura está considerada como uno de los videojuegos más influyentes de la historia, existiendo versiones para todos los ordenadores e influyendo a multitud de videojuegos modernos (ver Figura 2.2).

Figura 2.2 - Adventura en su versió para MS-DOS desarrollado por la compañía Level 9 Computing

La primera aventura conversacional en tener un gran éxito comercial fue la denomanida *Zork* [ZORK19], desarrollada por Infocom en 1980 (ver Figura 2.3). Empezó siendo una implementación particular de *Adventure* para el ordenador PDP-10 para luego, tras fundar los programadores su propia compañía, portarla a todos los ordenadores domésticos de la época, dividiéndola en varias entregas.

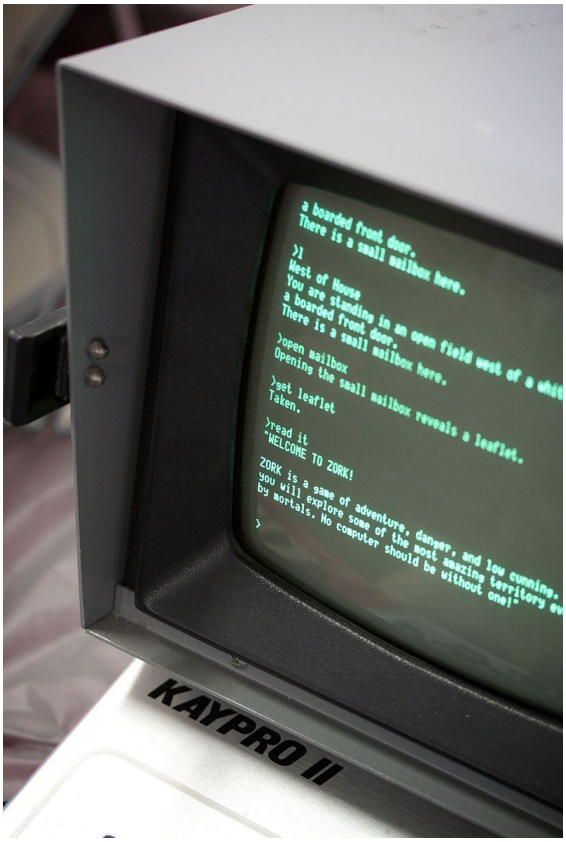

Figura 2.3 - Zork ejecutándose en un ordenador Kaypro

Otra aventura de gran importacia es la denominada *The Hobbit* [HOBBIT19], basada en la novela del mismo nombre de J. R. R. Tolkien, desarrollada por la compañía Beam Software y distribuida por la compañía Melbourne House para la mayoría de ordenadores domésticos disponibles en el momento (BBC Micro, ZX Spectrum, Commodore 64, Amstrad CPC 464, Dragon 64, Oric, MSX y Apple II) (ver Figura 2.4).

Figura 2.4 - The Hobbit ejecutándose en un ordenador Commodore 64

A nivel nacional cabe destacar las aventuras desarrolladas por la empresa Aventuras AD, de diversa índole, que fueron realizadas a finales de los años ochenta y principios de los años noventa del siglo pasado y que tuvieron una buena aceptación en el mercado nacional [AVAD19] (ver Figura 2.5).

Figura 2.5 - La aventura original en su versió IBM PC desarrollada por Aventuras AD

2.2 GENERACIÓN DE NARRATIVA PROCEDURAL

Existen muchas estrategías y herramientas que, con mayor o menor éxito, han pretendido atacar el problema de la generación de una narrativa procedural o con un aparente desarrolo espontáneo [KYBI16] (ver Figura 2.6).

Básicamente se basan en establecer en qué aspecto del espacio o guión se trabaja para alcanzar el objetivo de generar una narrativa procedural.

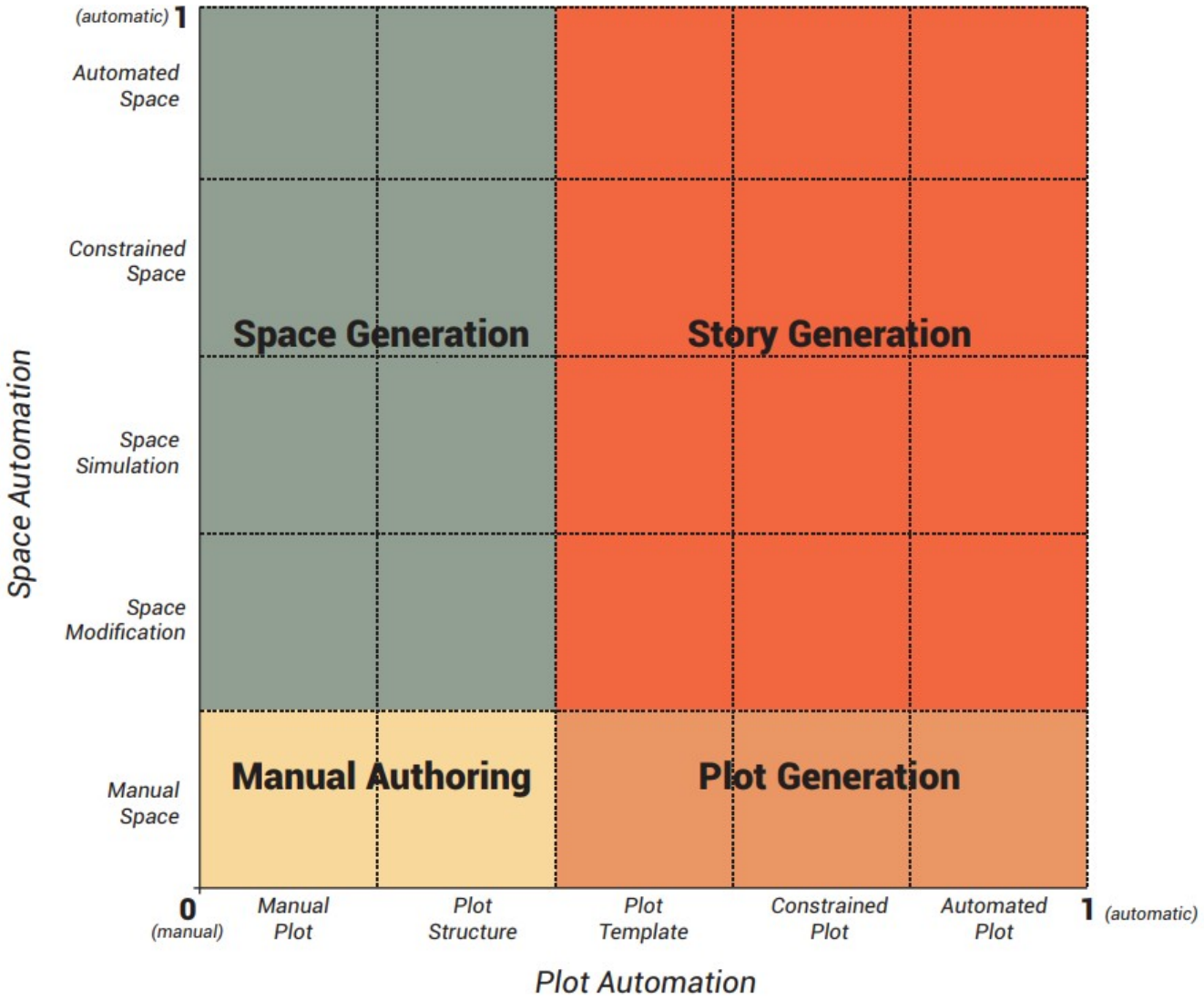

Figura 2.6 - Gama de automatización de historias, expresada en términos de los grados de automatización para la generación de guiones y espacios

El m*anual authoring* es el paradigma tradicional en el que la generación de todo el contenido recae en el equipo de personas que se encarga del proceso narrativo. Todos los aspectos del mismo se establecen en el momento de su creación, generando una narrativa estática sin ningún tipo de variación en sus diferentes ejecuciones.

Respecto el p*lot generation*, es un paradigma que se basa en construir un esquema básico de los eventos de la narración y definir posibles variaciones que se establecen de manera aleatoria o condicionada en las diferentes ejecuciones de la narración. Aunque introduce variaciones, estan acotadas a la estructura predefinida de eventos y pueden llegar a ser predecibles por parte del usuario final.

El Paradigma s*pace generation* hace referencia al hecho de construir un esquema básico de los personajes y objectos de la narración y la definición de posibles variaciones que se establecen de manera aleatoria o condicionada en las diferentes ejecuciones de la narración. Del mismo modo que el *Plot generation*, las posibles variaciones estan acotadas y pueden llegar a ser predecibles por parte del usuario final.

Finalmente, el paradigma de *Story generation*, es el más rico y el más costoso, pues es una combinación de los paradigmas de p*lot generation* y s*pace generation* que puede llegar a exigir un grado mayor de refinamiento en su desarrollo por parte del equipo encargado del proceso narrativo.

Al enforcar su atención en la automatización de uno más aspectos en particular, se han descuidado o ignorado otros, haciendo que sean soluciones potentes en algun aspecto en particular pero estáticas en el resto.

3 HERRAMIENTAS BASE

Además del sistema operativo, tres han sido las herramientas sobre las cuales se ha construido el sistema, bien modificándolas o incrustándolas en el sistema final: los programas Inform y Frotz, además del planificador Pyhop.

3.1 SISTEMA OPERATIVO Y PAQUETES

El sistema operativo escogido para albergar las herramientas y el sistema final ha sido Debian [DEBIAN19] en su versión 9 (con su actualización de abril de 2019) con la interfaz gráfica de usuario por defecto. Debian es un sistema operativo libre, basado en Linux, desarrollado por voluntarios de todo el mundo que colaboran a través de la red de redes. Los motivos de su elección han sido su facilidad a la hora de instalarlo y usarlo, además de su gran estabilidad.

Posteriormente se han añadido los siguientes paquetes para dar soporte a dos de las herramientas utilizadas (Inform, Frotz):

- build-essential: contiene una lista informativa de los paquetes considerados esenciales para la creación de paquetes Debian.
- libncurses: contiene el paquete ncurses, una biblioteca de programación que provee una API que permite al programador escribir interfaces basadas en texto.
- libao-dev: contiene una biblioteca de programación que provee una API para la gestió del audio.
- libmodplug-dev: contienen una biblioteca para la reproducción de audio.
- libsamplerate-dev: contienen una biblioteca para la conversión de audio.
- libsndfile-dev: contienen una biblioteca para leer y escribir ficheros de audio.
- libvorbis-dev: contienen una biblioteca para gestionar los ficheros de audio en format Ogg.

3.2 INFORM

Una máquina-Z (Z-machine) es una máquina virtual desarrollada por la compañía Infocom en 1979 para ejecutar sus aventuras conversacionales (ficciones interactivas).

Arquitectónicamente consiste en un procesador que se comunica con una memoria, un contador de programa y una pila. El contador de programa le permite al procesador ir ejecutando las instrucciones de la aventura que estan contenidads en rutinas. Respecto los datos de la aventura las tres estructuras más importantes son la cabecera (que contiene detalles sobre la aventura y una mapa del resto de la memoria de la aventura), el diccionario (una lista de palabras en inglés aceptadas por la aventura) y el árbol de objetos (que contiene todos los objetos con sus relaciones paterno-filiales en forma de árbol) [NELSON19].

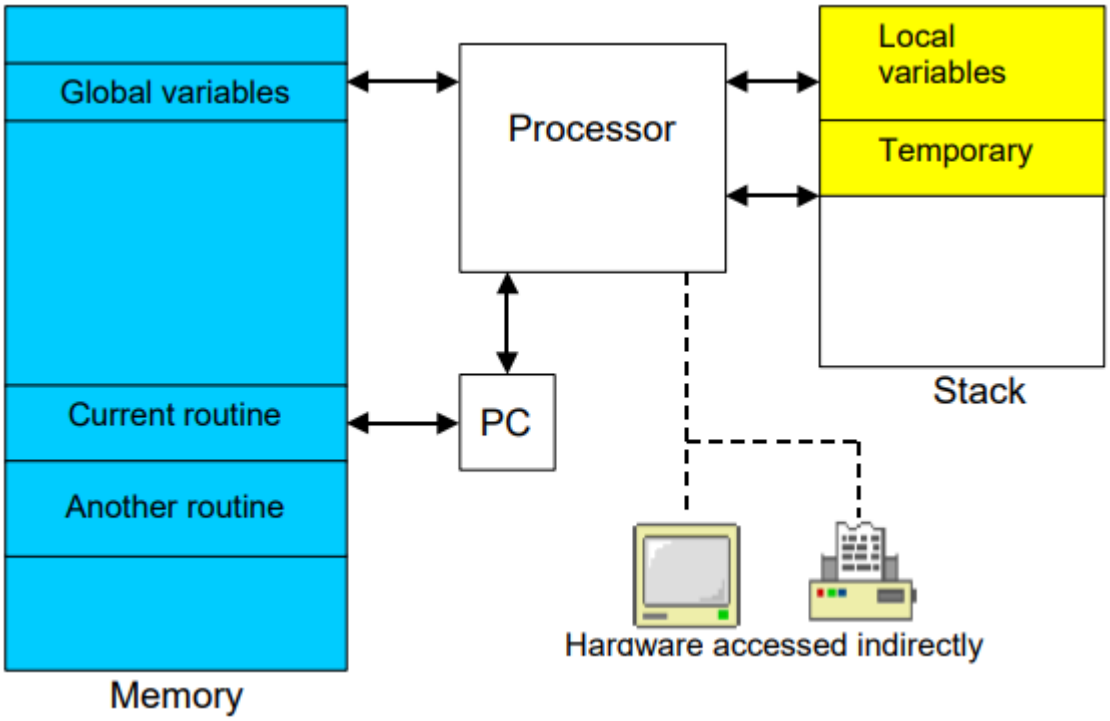

Figura 3.1 - Descripción general de la arquitectura de la máquina Z

Infocom desarrollaba sus aventuras conversacionales en un lenguaje de marcas propio, que luego convertía a instrucciones de la máquina-Z (llamadas ficheros de historias o fichero Z-Code) con un programa denominado Zilch, de manera que podía portar automáticamente sus aventuras conversacionales a cualquier máquina simplemente programando un intérprete de máquina-Z. Existieron intérpretes para todos los ordenadores domésticos de los años ochenta y noventa del siglo pasado y actualmente se han programado intérpretes para todas las arquitecturas existentes.

Inform es un paquete de herramientas para el desarrollo de aventuras conversacionales (ficciones interactivas) para la máquina-Z. Fue creado por en 1993 por Graham Nelson. Inform incluye como herramientas un lenguaje de marcas para el desarrollo de aventuras, una biblioteca y un compilador [INFORM19].

El lenguaje de marcas tiene como finalidad representar la aventura en alto nivel para facilitar su desarrollo. El compilador se encargar de convertir la aventura del lenguaje de marcas a código de la máquina-Z. Las bibliotecas de soporte incluyen elementos comunes en las aventuras codificados para que puedan ser usuados en cualquier momento.

La gestión que hace Inform es codificar la aventura en binario para la máquina-Z, incluyendo las estructuras de datos necesarias como el árbol de objetos, de forma que cualquier intérprete la podrá ejecutar sin importar la arquitectura donde se hace.

La estructura más importante es el árbol de objetos, que jerarquiza en forma de árbol la relación de los objetos respecto las localizaciones donde estan situados. Por ejemplo, en una aventura en la que un personaje denominado Heidi, que vive en un cabaña cerca de un bosque, encuentra un pequeño pájaro que ha caido de su nido y lo devuelve al mismo, su árbol de objetos se irá transformando como indica la Figura 3.2.

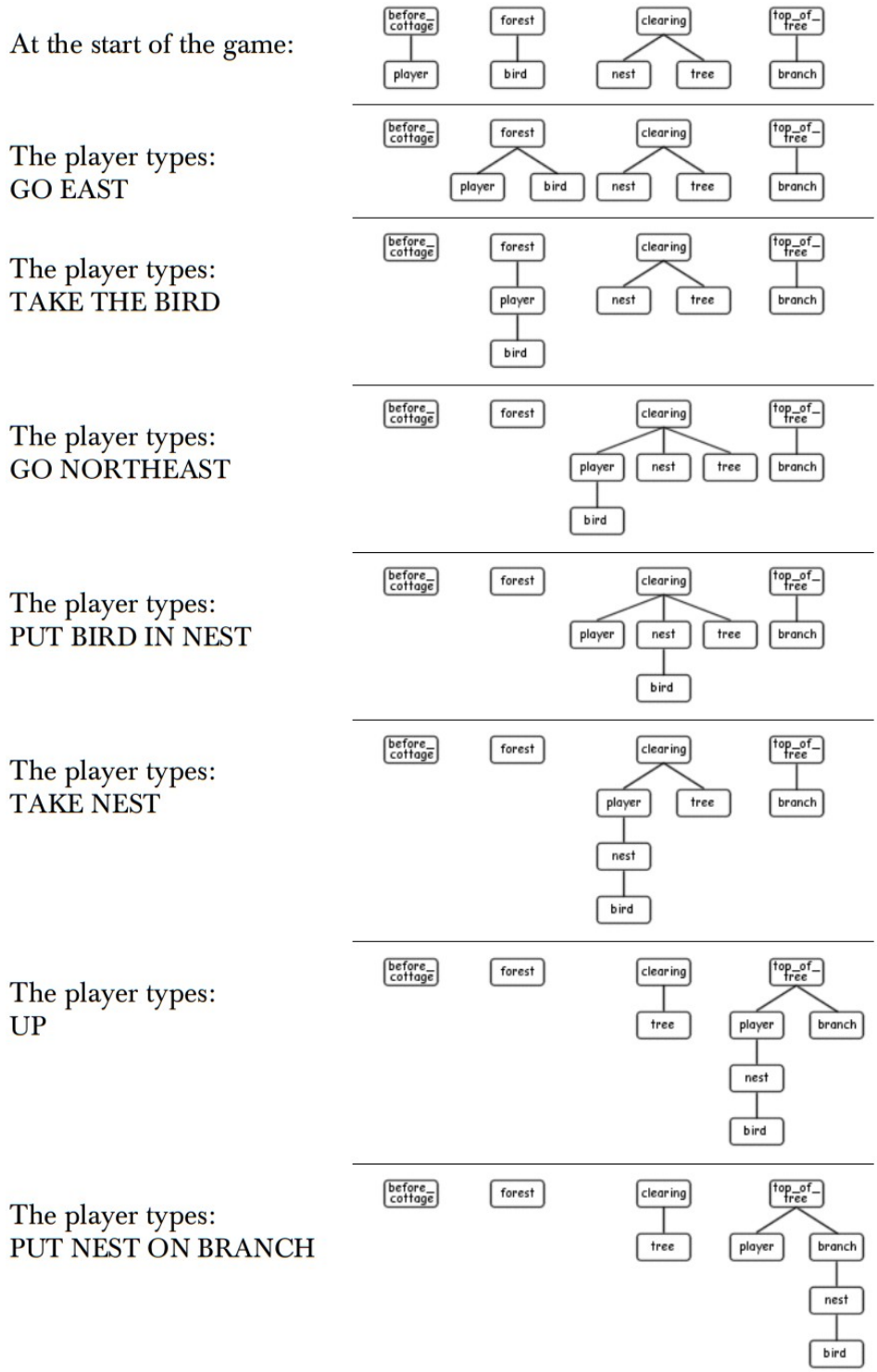

Figura 3.2 - Árbol de objetos de una aventura de ejemplo

La Figura muestra el árbol de objetos del capítulo uno de la aventura de ejemplo del sistema.

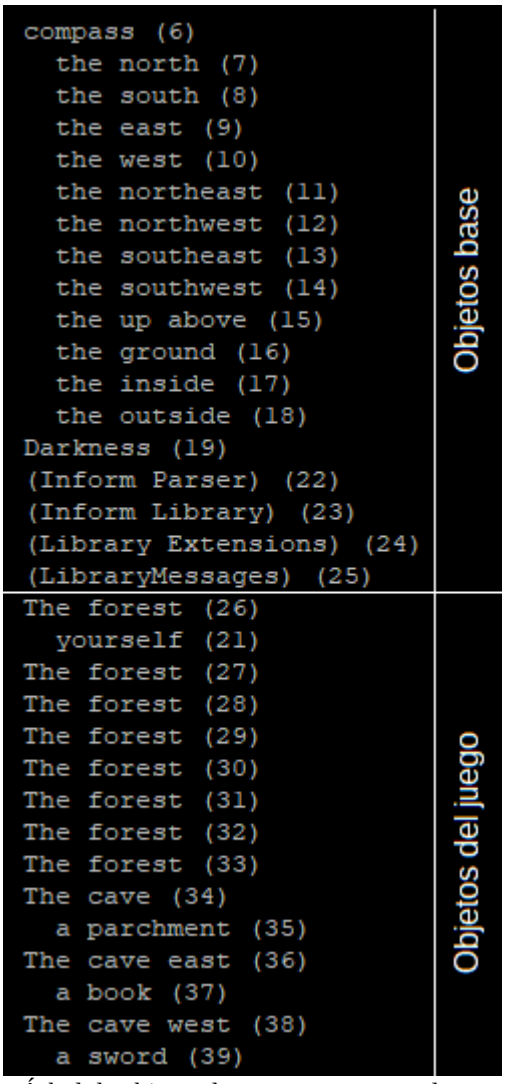

Figura 3.3 - Árbol de objetos de una aventura creada para el sistema

Los objetos base del juegos (como las direcciones de navegación o las bibliotecas) son comunes a todas las aventuras mientras que los objetos del juego son los que podrá interactuar el jugador (localizaciones, objetos, el jugador, etc) en una aventura en particular. La estructura de árbol representa las relaciones parteno-filiales.

3.3 FROTZ

Frotz es un intérprete de aventuras conversacionales de Infocom y otras basadas en la máquina-Z. Existen versiones para todo tipo de plataformas, incluyendo dispositivos móviles, y es considerado un estándard a la hora de reproducir aventuras conversacionales [FROTZ19] (ver Figura 3.4).

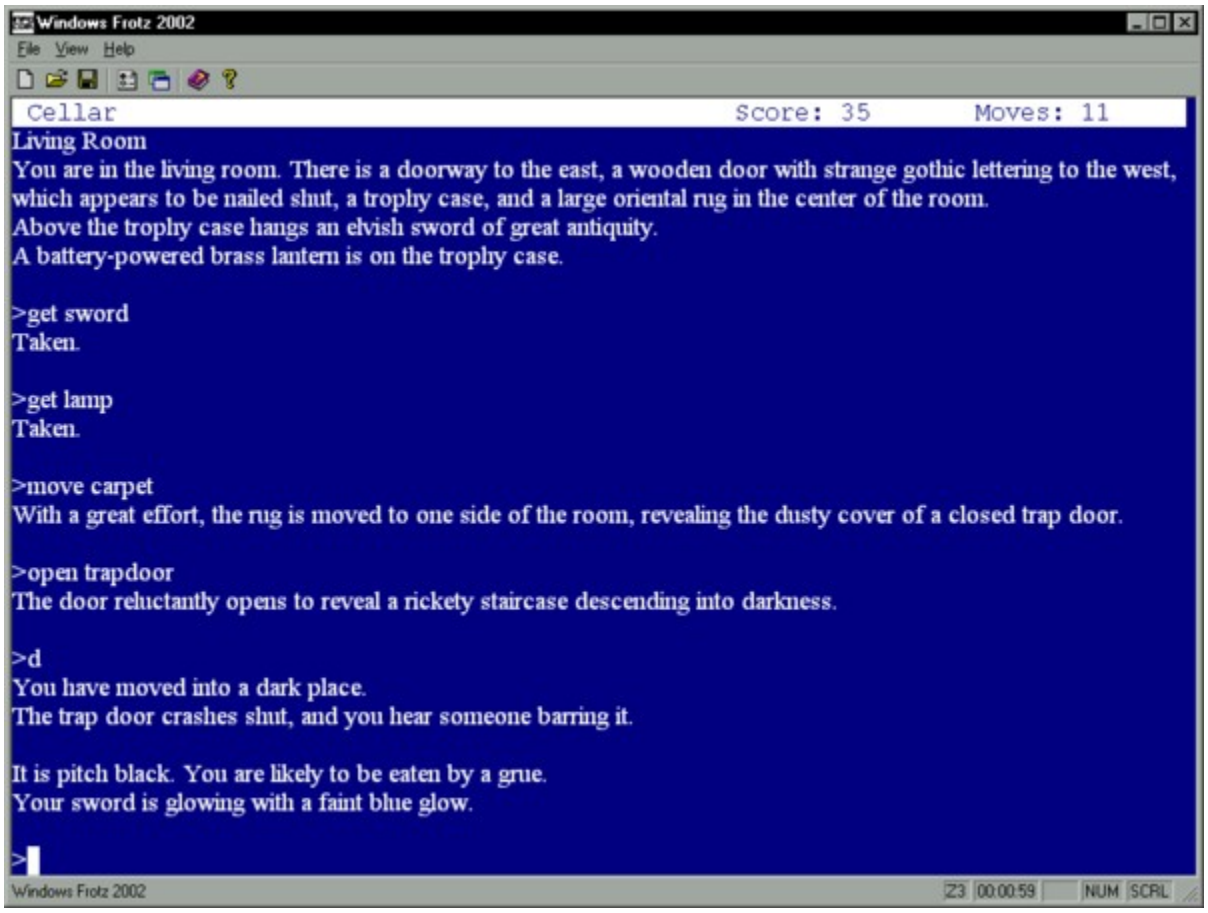

Figura 3.4 . Frotz ejecutándose sobre “Windows XP”

3.4 PYHOP

Pyhop es un planificador jerárquico (hierarchical task network planning) [PYHOP19] desarrollado en Python por Dana S. Nau que, mediante la definición de métodos y operaciones, permite obtener respuestas a preguntas referentes a la planificación de alguna tarea o propósito. Destacan su tamaño (ocupa menos de 150 líneas) y que es funcional con las dos versiones principales de Python (2.7 y 3.2). El algoritmo del panificador es similar a otro desarrollador por el mismo autor denominado SHOP [SHOP19].

Como ejemplo base se puede estudiar el problema del trayecto para ir de casa al parque, donde se debe valorar si se deber ir andando o en taxi, o en ambos. El problema consiste en que disponemos de recursos limitados y que la distancia no es corta.

Para ello se describe:

- Estado: Estoy en casa, tengo 20 € y el parque esta a 8 kilómetros.
- Objetivo: Ir al parque.
- Dos tipos tareas: primitivas (acciones base) y compuestas (formadas por tareas primtivas).

También se deberan tener en cuenta los operadores, pequeñas tareas acotadas que serviran de soporte a los métodos:

- Caminar de un lugar x hacia otro lugar y,
  - Precondición: el agente está en el lugar x.
  - Efecto: el agente está en el lugar y.
- Llamar a un taxi para situarlo en el lugar x,
  - Precondición: ninguna.
  - Efecto: el taxi está en el lugar x.
- Ir en taxi del lugar x hacia el lugar y,
  - Precondición: el agente y el taxi están en el lugar x.
  - Efecto: el agente y el taxi están en el lugar x y el el agente debe (1,5 * ½ distancia (x,y)) € al taxi.
- Pagar al taxista,
  - Precondición: el agente dispone de r €, donde r es mayor o igual al precio del trayecto en taxi.
  - Efecto: el agente no debe dinero al taxista y dispone del dinero que tenia al inicio menos lo que ha pagado al taxista.

Se deberan definir unos métodos:

- Ir andando de x a y,
  - Tarea: ir de x a y.
  - Precondición: el agente está en x, la distancia hacia y es menor o igual a 4 Km.
  - Subtarea: andar de x a y.
- Ir en taxi de x a y,
  - Tarea: ir de x a y.
  - Precondición: el agente está en x y tiene el dinero necesario para ir en taxi.
  - Subtareas: Llamar a un taxi, ir en taxi hacia y para finalmente pagar al taxista.

Gráficamente se puede interpretar como se puede ver en la Figura 3.5.

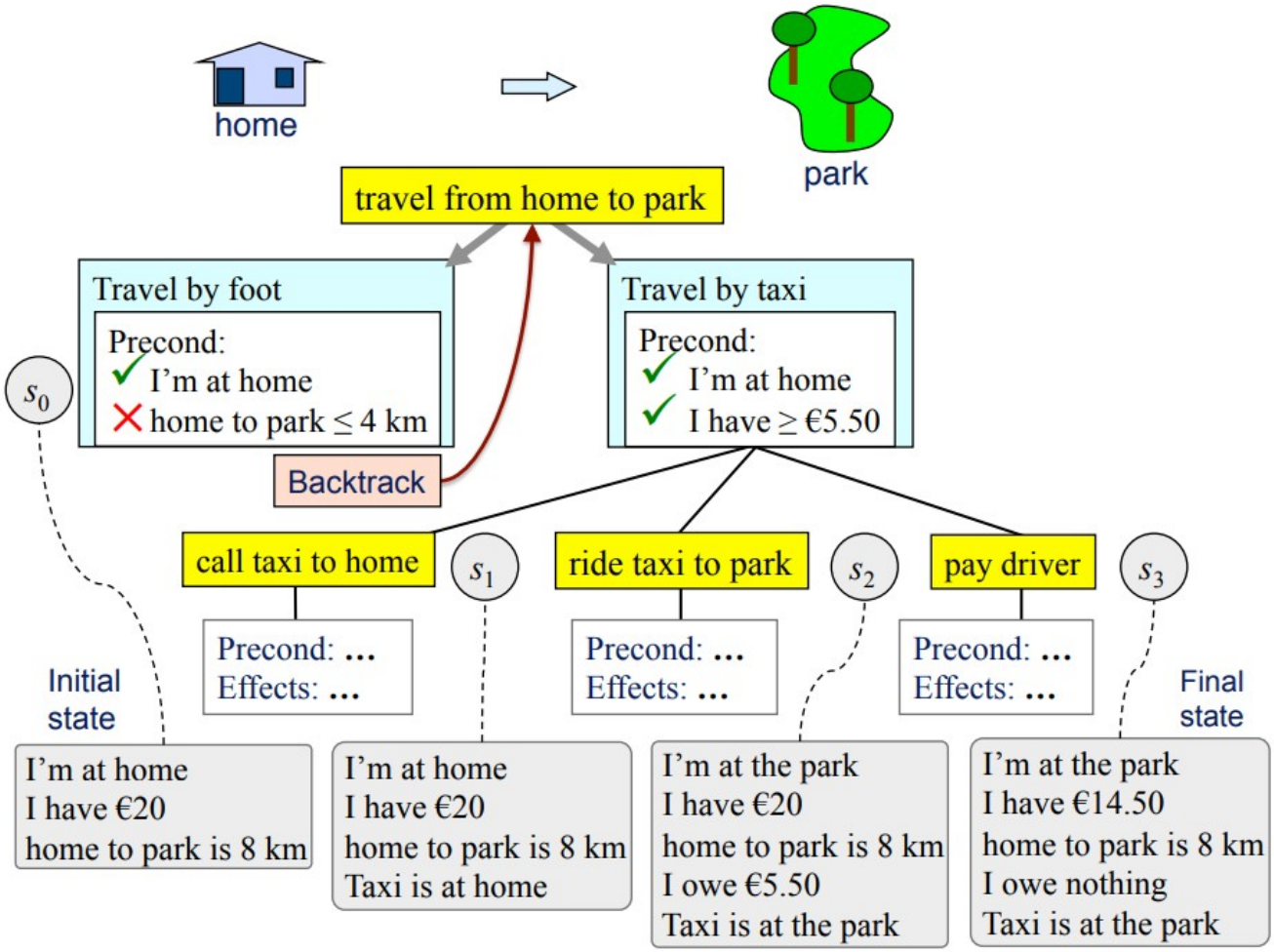

Figura 3.5 - Representación del problema del trayecto al parque en función de los recursos disponibles y la distancia

Siendo el código de Python, utilizando la biblioteca Pyhop, que planifica el trayecto el siguiente:

```
"""
The "travel from home to the park" example from my lectures.
Author: Dana Nau <nau@cs.umd.edu>, May 31, 2013
This file should work correctly in both Python 2.7 and Python 3.2.
"""

import pyhop
```

El primer bloque contiene los comentarios que describen el código y la sentencia para instanciar la biblioteca.

```
def taxi_rate(dist):
    return (1.5 + 0.5 * dist)

def walk(state,a,x,y):
    if state.loc[a] == x:
        state.loc[a] = y
        return state
    else: return False

def call_taxi(state,a,x):
    state.loc['taxi'] = x
    return state

def ride_taxi(state,a,x,y):
    if state.loc['taxi']==x and state.loc[a]==x:
        state.loc['taxi'] = y
        state.loc[a] = y
        state.owe[a] = taxi_rate(state.dist[x][y])
        return state
    else: return False

def pay_driver(state,a):
    if state.cash[a] >= state.owe[a]:
        state.cash[a] = state.cash[a] - state.owe[a]
        state.owe[a] = 0
        return state
    else: return False

pyhop.declare_operators(walk, call_taxi, ride_taxi, pay_driver)
```

El segundo bloque define y declara los operadores, pequeñas funciones que dan soporte a los métodos, que se encargan de realitzar evaluciones o cálculos.

- El primer operador, taxi_rate, calcula el precio de realitzar un viaje en taxi de una distancia determinada (el argumento que se indica al ser llamado el operador).
- El operador walk se encarga de mover a la persona que desea realizar el trayecto de la posició x a la y.
- El operador call_taxi se encarga de posicionar taxi en la misma posición de la persona que desea realizar el trayecto.
- El operador ride_taxi se encarga de, si el taxi y la persona que desea realizar el trayecto estan en la misma posición, desplazarlos hasta otra posición además de calcular el precio de ese trayecto, utilizando para esto último el operador taxi_rate.
- Finalmente el operador pay_driver se encarga de gestionar la transación económica entre la persona y el taxi.

Cabe destacar que los operadores utilizan una estructura de datos para almacenar todo la información necesaria para resolver el problema, denominada estado. Esta estructura contiene la localización inicial de la persona que desea realizar el trayecto, la cantidad de dinero que dispone, el dinero que debe al taxista y la distancia que desea recorrer. A medida que el planificador realiza operaciones a través de llamadas de los métodos, esta estructura va variando.

```
def travel_by_foot(state,a,x,y):
    if state.dist[x][y] <= 2:
        return [('walk',a,x,y)]
    return False

def travel_by_taxi(state,a,x,y):
    if state.cash[a] >= taxi_rate(state.dist[x][y]):
        return [('call_taxi',a,x), ('ride_taxi',a,x,y), ('pay_driver',a)]
    return False

pyhop.declare_methods('travel',travel_by_foot,travel_by_taxi)
```

El tercer bloque define y declara los métodos, funciones que evaluan a alto nivel las posibilidades del problema y que se sustentan sobre los operadores.

El primer método se denomina travel_by_foot y se encarga de evaluar si la distancia que se desea recorreger es inferior a una distancia determinada (operador dist). La idea de este método es determinar si la distancia es pequeña y si es así se puede recorrer a pie (operador walk).

El segundo método se denomina travel_by_taxi y es ejecutado si el método anterior da como resultado falso, indicativo que la distancia es grande y no se puede recorrer a pie. Este segundo método, cuando es ejecutado, evaluará si la persona dispone de suficiente dinero para realitzar un trayecto en taxi (operador taxi_rate) e indicará qué pasos ha de seguir la persona para realitzar el trayecto (operadores call_taxi, ride_taxi y pay_driver).

```
state1 = pyhop.State('state1')
state1.loc = {'me':'home'}
state1.cash = {'me':20}
state1.owe = {'me':0}
state1.dist = {'home':{'park':8}, 'park':{'home':8}}
```

Finalmente se debe definir el estado del que parte el problema, para que los operadores y métodos puedan ir almacenando la variación de los datos base, y realizar la llamada a Pyhop,

```
regular@debian-9-gui:~$ cd /opt/pyhop/
regular@debian-9-gui:/opt/pyhop$ python3 simple_travel_example.py
** pyhop, verbose=1: **
   state = state1
   tasks = [('travel', 'me', 'home', 'park')]
** result = [('call_taxi', 'me', 'home'), ('ride_taxi', 'me', 'home', 'park'), ('pay_driver', 'me')]
```

Figura 3.6 - Solución del problema de trayecto al parque mediante Pyhop con verbose = 1

Las llamadas se pueden modificar para mostrar más información respecto al desarrollo de la solución, añadiendo un valor a un parámetro denominado verbose, que se encargará de detallar los pasos que realizar el planificador para resolver el problema.

```
state1 = pyhop.State('state1')
state1.loc = {'me':'home'}
state1.cash = {'me':20}
state1.owe = {'me':0}
state1.dist = {'home':{'park':8}, 'park':{'home':8}}

#Call pyhop.pyhop(state1,[('travel','me','home','park')]) with different verbosity levels

#print('- If verbose=2, Pyhop also prints a note at each recursive call:')
pyhop.pyhop(state1,[('travel','me','home','park')],verbose=2)

#print('- If verbose=3, Pyhop also prints the intermediate states:')
#pyhop.pyhop(state1,[('travel','me','home','park')],verbose=3)
```

```
regular@debian-9-gui:/opt/pyhop$ python3 simple_travel_example.py
** pyhop, verbose=2: **
   state = state1
   tasks = [('travel', 'me', 'home', 'park')]
depth 0 tasks [('travel', 'me', 'home', 'park')]
depth 1 tasks [('call_taxi', 'me', 'home'), ('ride_taxi', 'me', 'home', 'park'), ('pay_driver', 'me')]
depth 2 tasks [('ride_taxi', 'me', 'home', 'park'), ('pay_driver', 'me')]
depth 3 tasks [('pay_driver', 'me')]
depth 4 tasks []
** result = [('call_taxi', 'me', 'home'), ('ride_taxi', 'me', 'home', 'park'), ('pay_driver', 'me')]
```

Figura 3.7 - Solución del problema de trayecto al parque mediante Pyhop con verbose = 2

4 SISTEMA DE NARRATIVA PROCEDURAL

El sistema de narrativa procedural está compuesto por una aventura conversacional prediseñada. Parte de su contendio se puede alterar mediante código Python, para variar la linea narrativa. Una vez generada la aventura, mediante una llamada a la aplicación Inform se compila para que puede ser interpretada por Frotz.

Habitualmente, una aventura conversacional está contenida en un sólo fichero, pero para el propósito de este sistema se ha dividio en varios ficheros que representan diversos capítulos. Esta división permite que, antes de empezar un capítulo, el sistema pueda consultar cómo han transcurrido los anteriores y contruir en consecuencia los siguientes capítulos en función de criterios narrativos y de jugabilidad. Estos criterios los pueden establecer los miembros del equipo de desarrollo.

El encapsulado del sistema se puede ver en la Figura 4.1.

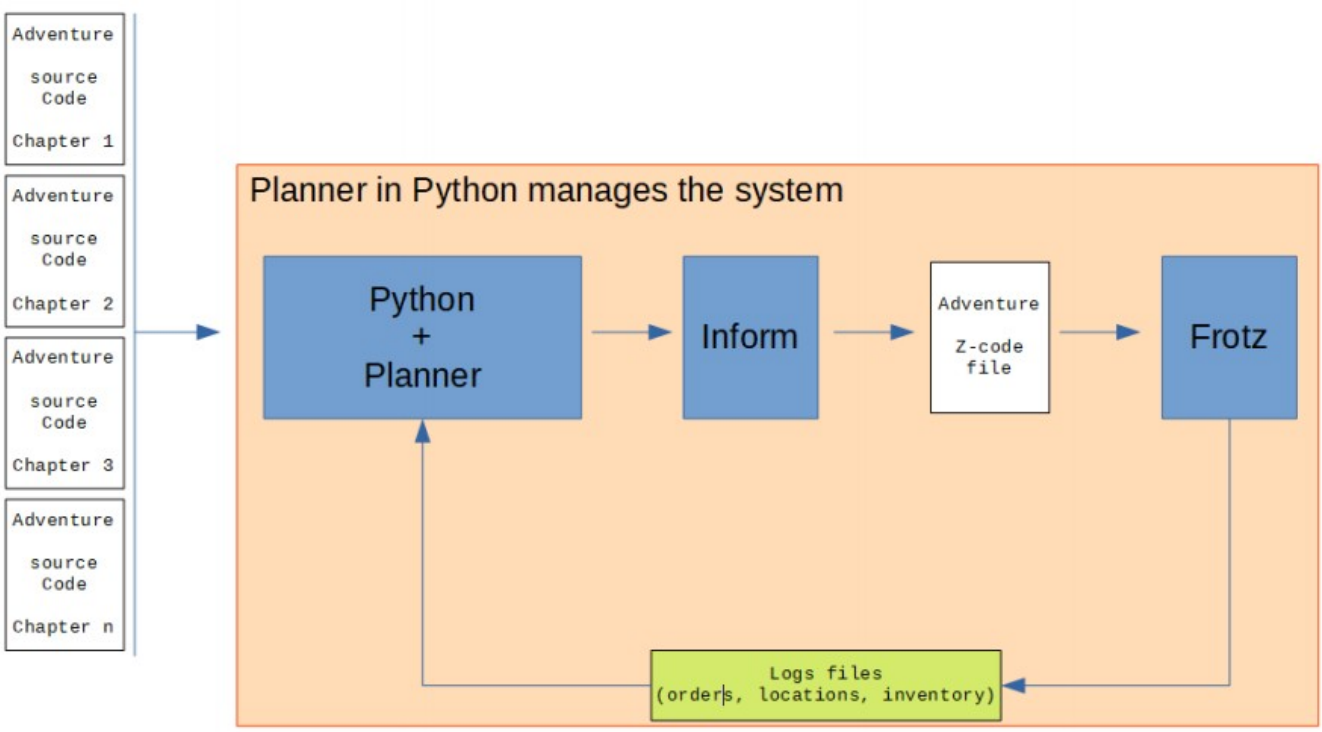

Figura 4.1 - Componentes del sistema y su encapsulado

El encapsulado representa el funcionamiento del sistema, un programa en Python que incorpora el planificador y que se encarga de generar los diferentes capítulos de la aventura conversacional (crear el código fuente y compilarlo para la máquina-Z) para ejecutarlos a través del programa Frotz. Esto se reliza por cada capítulo que forma parte de la aventura conversacional. Se fragmenta la aventura conversacional en capítulos para poder “detener” el transcurso de la partida e ir modificando el contendio de los siguientes capítulos.

4.1 PYTHON + PLANNER

Como se puede observar en la Figura 4.1, el conjunto está gobernado por una serie de ficheros en código Python que se encargan de gestionar los diferentes capitulos. Cada fichero realiza, a alto nivel, las siguientes tareas:

- Compilación del código fuente del capítulo de la aventura conversacional mediante una llamada a Inform.
- Ejecución de la aventura conversacional mediante una llamada a Frotz.
- Tratamiento de los ficheros log (registros de Frotz en los que se almacena información de juego de la aventura conversacional).
- Generación del código fuente del siguiente capítulo consultando el Planificador.

Un ejemplo de un fichero Python, el que gestiona el capítulo 3, se puede observar a continuación. El código en Python se encarga de ejecutar el capítulo y, una vez finalizado, realitzar variaciones de contenido para el siguiente capítulo.

```
import os, import csv, pyhop

res = 0
```

El primer bloque se encarga de contener referencias a las bibliotecas utilizadas, además de la variable que contendrá el resultado final, una variable que le transmitirá al código Python la decisión del planificador.

```
def sel_jump(state,p):
    global res
    res = 50
    return state

def sel_after(state,p):
    global res
    res = 20
    return state

def sel_middle(state,p):
    global res
    res = 10
    return state

def sel_before(state,p):
    global res
    res = 5
    return state

pyhop.declare_operators(sel_jump,sel_after,sel_middle,sel_before)
```

Se definen los operadores que deberá utilizar Pyhop, que se encargaran de determinar el valor de la variable que contendrá el resultado que situará al jugador en una localización en particular del mapeado. Cada operador es llamado por uno de los métodos. Estos operadores se encargan de asignar a la variable final el resultado del planificador, un valor que le indica al código Python qué decisión tomó el planificador.

```
def loc_jump_chapter(state,p):
    if int(state.oro[p]) > 1 and int(state.plata[p]) > 0 and int(state.bronce[p]) > 0:
       return [('sel_jump',p)]
    return False

def loc_after_maze(state,p):
    if int(state.oro[p]) > 0 and int(state.plata[p]) > 0:
       return [('sel_after',p)]
    return False

def loc_middle_maze(state,p):
    if int(state.plata[p]) > 0:
       return [('sel_middle',p)]
    return False

def loc_before_maze(state,p):
    if int(state.bronce[p]) > 0:
       return [('sel_before',p)]
    return False

pyhop.declare_methods('player_loc_ini',loc_jump_chapter,loc_after_maze,loc_middle_maze,loc_before_maze)
```

Posterirmente se definen los métodos utilizados por Pyhop. En este caso se encargan de determinar cómo construir el cuarto capítulo y, en función de como finaliza el tercer capítulo, posicionar al jugador en el siguiente. Los métodos simplemente ejecutan sentencias en las que mediante casuísticas determinan diferentes estados y en consecuencia realizan llamadas a los operadores oportunos.

Por ejemplo, el método loc_jump_chapter examina si el juegador a recolectado todos los objetos posibles, en casa afirmativo llama al operador sel_jump que se encargará de especificar que el jugador deberá aparecer en el quinto capítulo, saltando el cuarto.

```
def build_chapter_four(fi,val):
    os.system('cat the-adventure_4p.inf > the-adventure_4.inf')

    f = open(fi,'a+')

    f.write('!============================================================================='+'\n')
    f.write('! Entry point routines'+'\n')
    f.write('\n')

    if (val == 5):
        f.write('[ Initialise; location = p_river_01; ];'+'\n')

    if (val == 10):
        f.write('[ Initialise; location = p_river_06; ];'+'\n')

    if (val == 20):
        f.write('[ Initialise; location = p_river_09; ];'+'\n')

    f.write('\n')
    f.write('!============================================================================='+'\n')
    f.write('! Standard and extended grammar'+'\n')
    f.write('\n')
    f.write('Include "Grammar";'+'\n')
    f.write('\n')
    f.write('!============================================================================='+'\n')
    f.close()
```

El anterior bloque contiene la función que se encargará de construir el cuarto capítulo, tal y como lo haya determinado el planificador. Parte del cuarto capítulo está preconstruido y esta función añade a la preconstrución la situación del jugador al inicio del mismo. Dicho proceso consiste en añadir el código Inform (Initialise; location) que especifica en qué posición empieza el jugador:

```
filelogsorders = "orders.log"
filelogstree = "objectstree.log"

os.system('/opt/inform/inform +include_path=/opt/inform/lib/ the-adventure_3.inf -D')
os.system('/opt/frotz/frotz the-adventure_3.z5')

penultimalinea = ""
ultimalinea = ""

f = open(filelogsorders, "r")

for x in f:
        penultimalinea = ultimalinea
        ultimalinea = x

f.close()

if penultimalinea.upper() == "QUIT\n":
               quit()
```

El anterior bloque contiene las llamadas al sistema operativo (comandos Linux de ejecución de Inform y Frotz) para construir y ejecutar el tercer capítulo, además de la casuística para determinar si el jugador a introducido la orden para finalizar el juego.

```
lugar="0"
oro=0
plata=0
bronce=0

with open('objectstree.log', newline='') as File:
    reader = csv.reader(File)
    for row in reader:
        if row[0] == "21": lugar = row[1]
        if row[0] == "27" and row[1] == "21": bronce = 1
        if row[0] == "29" and row[1] == "21": plata = 1
        if row[0] == "32" and row[1] == "21": oro = oro + 1
        if row[0] == "38" and row[1] == "21": oro = oro + 1

state1 = pyhop.State('state1')
state1.location = {'player':lugar}
state1.oro = {'player':oro}
state1.plata = {'player':plata}
state1.bronce = {'player':bronce}

pyhop.pyhop(state1,[('player_loc_ini','player')])

if res == 0: os.system('python3 the-adventure_3.py')

if res == 5:
    build_chapter_four("the-adventure_4.inf",res)
    os.system('python3 the-adventure_4.py')

if res == 10:
    build_chapter_four("the-adventure_4.inf",res)
    os.system('python3 the-adventure_4.py')

if res == 20:
    build_chapter_four("the-adventure_4.inf",res)
    os.system('python3 the-adventure_4.py')

if res == 50: os.system('python3 the-adventure_5.py')
```

Este último bloque se encarga de verificar como ha terminado el tercer capítulo, realizar la llamada al planificador y realizar la llamada al sistema del siguiente fichero Python a ejectuar.

4.2 FROTZ

Por su parte Frotz ha sido modificado para generar ficheros de sucesos con información de la aventura conversacional (órdenes introducidas por el usuario, localizaciones recorridas, inventario, objectos, etc). Al finalizar un capítulo, el código Python que lo ha llamado examinarà estos ficheros para que, según los criterios de generación de cada capítulo, se planifique la construcción del siguiente. Estas construcciones se basan en partes de los capítulos construidos a los que se le pueden añadir o modificar localizaciones, objetos o determinar en qué localización se situará el jugador al principio.

La estructura de directorios del código de Frotz es la que muestra la Figura 4.2.

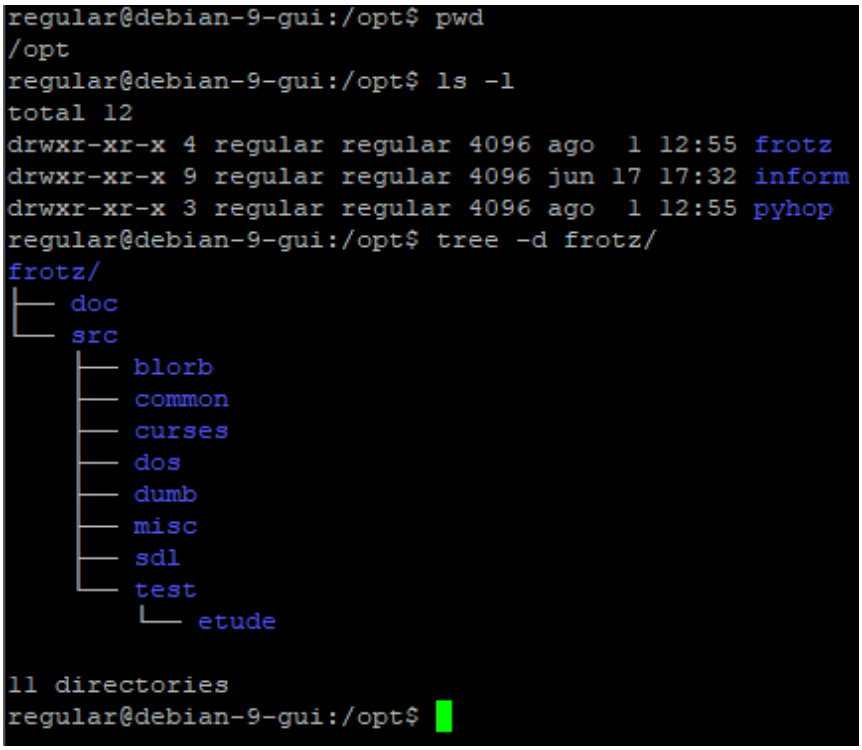

Figura 4.2 - Estructura de directorios de Frotz

Teniendo en cuenta que Frotz es una aplicación multiplataforma, dispone de diversos directorios para realizar compilaciones en consecuencia a los diferentes entornos (frotz/scr/curses/, frotz/scr/dos/, frotz/scr/sdl/, etc). El código genérico, independiente de la plataforma, está dentro del directorio /frotz/src/common/.

Los ficheros modificados de Frotz son los siguientes:

frotz/src/curses/ux_init.c

```
void os_reset_screen (void)
{
    os_stop_sample(0);
    os_set_text_style(0);
    /*
       os_display_string((zchar *)"[Hit any key to exit.]\n");
    os_read_key(0, FALSE);
       */
    os_quit();
}/* os_reset_screen */
```

Este fichero forma parte de la interfaz gráfica para sistema Linux y se ha comentado el mensaje de texto que muestra antes de salir para que se pueda gestionar desde el código Python. Este mensaje se ha tenido que ocultar porque, como el juego esta dividido en capítulos, se utiliza como función de salida cuando el jugador ha terminado uno, y se oculta la confirmación para generar una correcta transición de un capítulo al siguiente.

Si se quisiera compilar Frotz para otro sistema operativo, se debería modificar la función equivalente contenida en el directorio correspondiente (‘sdl’, ‘dos’, etc).

frotz/src/common/object.c

Este fichero contiene todas las funciones que se encargan de gestionar los objetos de la aventura conversacional. Se ha añadido una función que, dado el identificador de un objeto, obtiene el identificador de su objeto padre.

```
zword z_get_parent_an_obj (int id)
{
    zword obj_addr;

       obj_addr = object_address (id);

       zword parent;

       obj_addr += O4_PARENT;
       LOW_WORD (obj_addr, parent)


       return parent;

}/* MEINF z_get_parent_an_obj */
```

Teniendo en cuenta que los objetos se almacenan dentro de un árbol, esta función permite descubrir si un objeto está en una localización o si un objeto es propiedad del jugador. Por ejemplo, si una localización contiene un objeto, el padre de éste será la localización, pero en cuanto el objeto es recogido por el jugador, éste pasará a ser su padre (ver Figura 4.3).

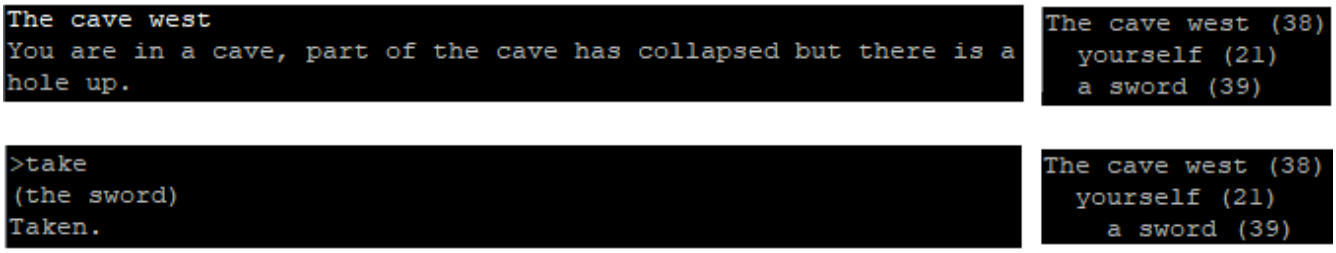

Figura 4.3 - Orden de recogida de un objeto y repercusión en el árbol de objetos

frotz/src/common/input.c

Este fichero contiene todas las funciones que permiten gestionar la entrada de comandos en la aventura conversacional por parte del jugador. Las modificaciones se han insertado en la función z_read, que es la que se encarga de capturar los comandos que introduce el jugador. Por lo tanto, se realizan las modificaciones cada vez que el jugador introduce un comando y pulsa enter. El juego se ejecuta por turnos, por lo que se deben hacer las comprobaciones después de cada turno.

La primera modificación consiste en añadir la casuística que gestiona el fin de los capítulos, ya que Frotz está diseñado para que una aventura conversacional esté contenida en un único fichero.

```
zword location_player;

location_player = z_get_parent_an_obj(21); // 21 es la direccion en la memoria del jugador

if (strcmp(f_setup.story_name,"the-adventure_1") == 0 && location_player == 36 &&
strcmp(buffer,"abracadabra") == 0) z_quit();
if (strcmp(f_setup.story_name,"the-adventure_1") == 0 && location_player == 38 && strcmp(buffer,"up")
== 0) z_quit();
if (strcmp(f_setup.story_name,"the-adventure_2") == 0 && location_player == 35 &&
strcmp(buffer,"thunder") == 0) z_quit();
if (strcmp(f_setup.story_name,"the-adventure_2") == 0 && location_player == 34 &&
strcmp(buffer,"fight") == 0) z_quit();
if (strcmp(f_setup.story_name,"the-adventure_3") == 0 && location_player == 40 &&
strcmp(buffer,"offer") == 0) z_quit();
if (strcmp(f_setup.story_name,"the-adventure_4") == 0 && location_player == 38 && strcmp(buffer,"e")
== 0) z_quit();
```

La segunda modificación consiste en añadir el código para la creación de los ficheros de registros (logs, un fichero de texto en formato CSV) de la información del juego. Se crean dos ficheros para almacenar esta información.

El primero fichero, denominado orders.log, contendrá todas las órdenes que va introduciendo el usuario, como se puede observar en la Figura 4.4.

```
 1  n
 2  n
 3  n
 4  n
 5  take
 6  w
 7  take
 8  up
 9  n
10  n
11  n
12  n
13  fight
14  nw
15  take
16  up
17  take
18  down
19  n
20  ne
21  n
22  offer
23  e
24  n
25  e
```

Figura 4.4 - Contenido del fichero orders.log de una ejecución de la aventura

El segundo fichero, denominado objectstree.log, contendrá el árbol de objetos en un formato especial: por cada objeto existirá una línea que empezará con el identificador del objeto, y a continuación contendrá el padre del objeto, a continuación el padre del padre, y así hasta que se llegue a la raiz del árbol de objetos. Se puede ver un ejemplo en la Figura 4.5.

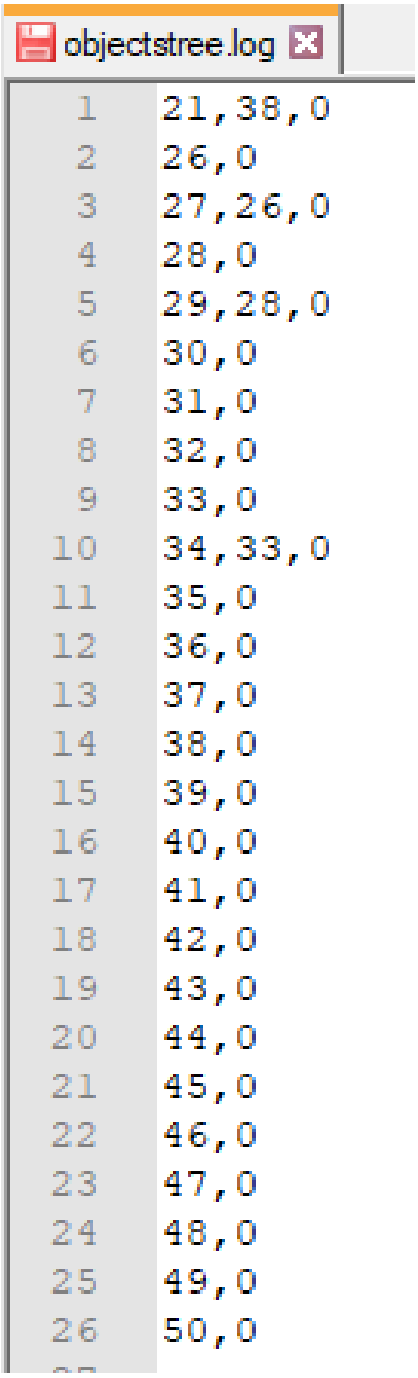

Figura 4.5 -Contenido del fichero objectstree.log de una ejecución de la aventura

El código que genera estos ficheros es el siguiente.

```
FILE * logs_orders;

logs_orders = fopen("orders.log","a");
fprintf(logs_orders, "%s\n", buffer);
fclose(logs_orders);

FILE * logs_objectstree;

logs_objectstree = fopen("objectstree.log","w");

id_objeto = 21;
fprintf(logs_objectstree, "%d", id_objeto);
do
{
	padre_objeto = z_get_parent_an_obj(id_objeto);
	fprintf(logs_objectstree, ",%d", padre_objeto);
	if (padre_objeto != 0) id_objeto = padre_objeto;
}
while (padre_objeto != 0);
fprintf(logs_objectstree, "\n");

for (i=26; i <= 50; i++) {
	id_objeto = i;
	fprintf(logs_objectstree, "%d", id_objeto);
	do
	{
		padre_objeto = z_get_parent_an_obj(id_objeto);
		fprintf(logs_objectstree, ",%d", padre_objeto);
		if (padre_objeto != 0) id_objeto = padre_objeto;
	}
	while (padre_objeto != 0);
	fprintf(logs_objectstree, "\n");
}

fclose(logs_objectstree);
```

Este fragmento de código se encarga de abrir dos ficheros, el de las órdenes y el de los objetos. Respecto las órdenes, por cada orden que introduce el jugador, es añadida en el fichero de registros de las órdenes. Este fichero contendrá todas las órdenes que ha introducido el jugador durante la patida. En el caso de los objetos, cada vez que el jugador introduce una orden, se elimina el contenido del fichero de registros y se añade al árbol de objetos con el formato comentado anteriormente, de manera que, por cada objeto, se puede rastrear la relación parternal hasta la raíz del árbol.

4.3 ELEMENTOS DE LA NARRATIVA PROCEDURAL

4.3.1 Ejemplo de transición (del capítulo 1 al 2)

En el primer capítulo, tras una serie de localizaciones que sirven para que el usuario se familiarice con las mecánicas de la aventura, se llega a una localización que contiene dos caminos, uno al este y otro al oeste, y que una vez escogido uno, sea cual sea, no se puede retroceder. La localización con la bifurcación contiene un texto informativo indicando al jugador que en función de su elección el juego tomará diferentes sentidos (ver Figura 4.6).

El que se dirige el oeste contiene un libro mágico, que si el juegador recoge y utiliza, determinará que el segundo capítulo tenga un desarrollo orientado a la magia. Por otro lado el camino que se dirige al este, llevará al jugador a una localización que contiene una espada, que si el jugador recoge determinará que el segundo capítulo tenga un desarrollo orientado a la lucha. En una tercera rama del desarrollo consiste en que, si el jugador no recoge ningún objeto, accederá al segundo capítulo sin objetos y sin la posibilidad de finalizarlo..

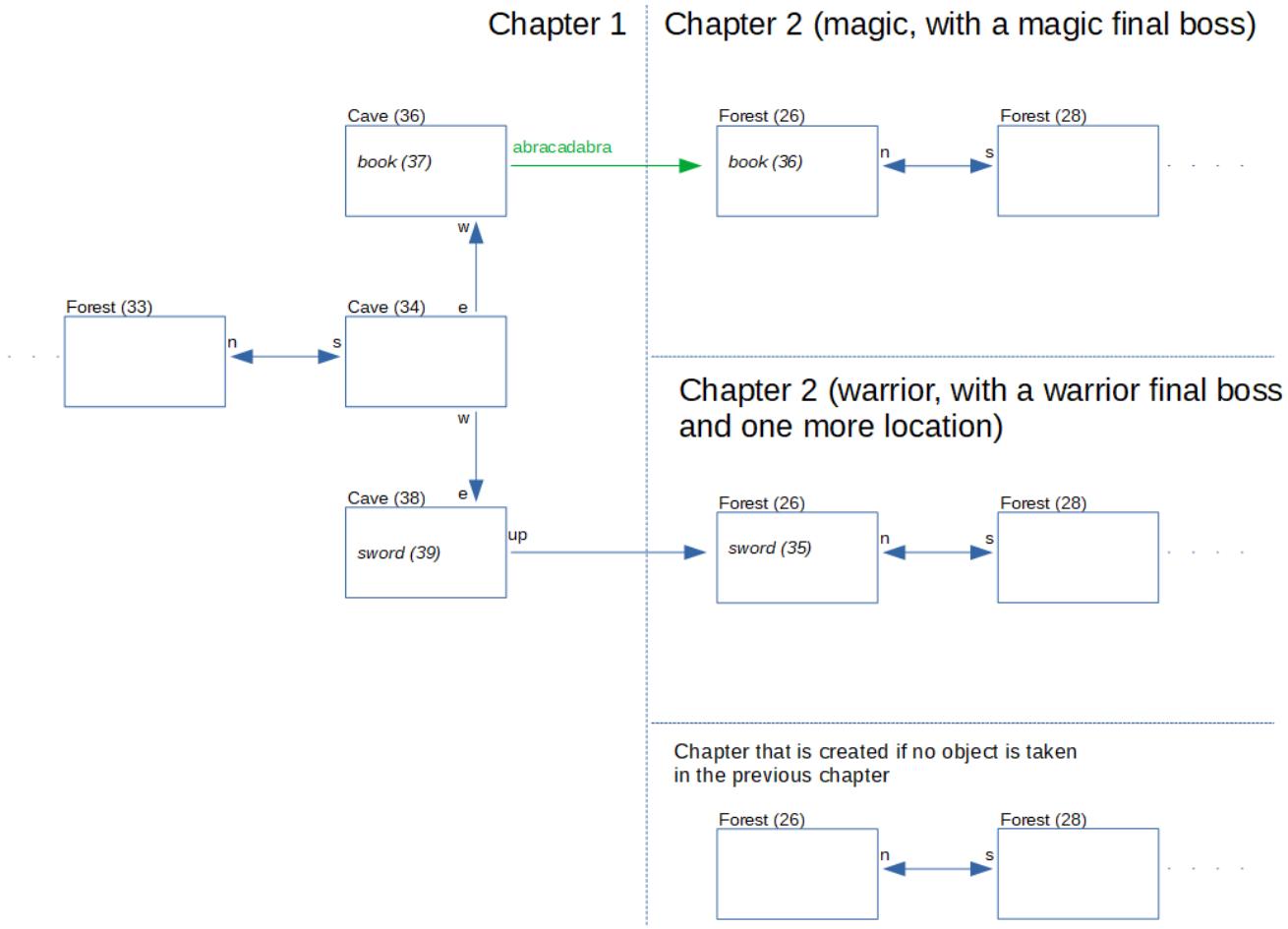

Figura 4.6 - Esquema de la transición entre el capítulo 1 al 2

4.3.2 Ejemplo de transición (del capítulo 2 al 3)

El segundo capítulo está condicionado por los objetos recogidos en el primer capítulo (ver Figura 4.7). Si el jugador recoge el libro mágico en el primer capítulo, el segundo capítulo tendrá como enemigo final un terrorífico ser situado en una localización que los otros hilos narrativos no tendrán. En cambio, si el jugador recoge la espada en el primer capítulo, tendrá como enemigo final en el segundo capítulo un guerrero. Finalmente se debe destacar que si el jugador no recoge ningún objeto en el primer capítulo, el segundo no contendrá objetos ni enemigos y tampoco una manera de acceder al tercero.

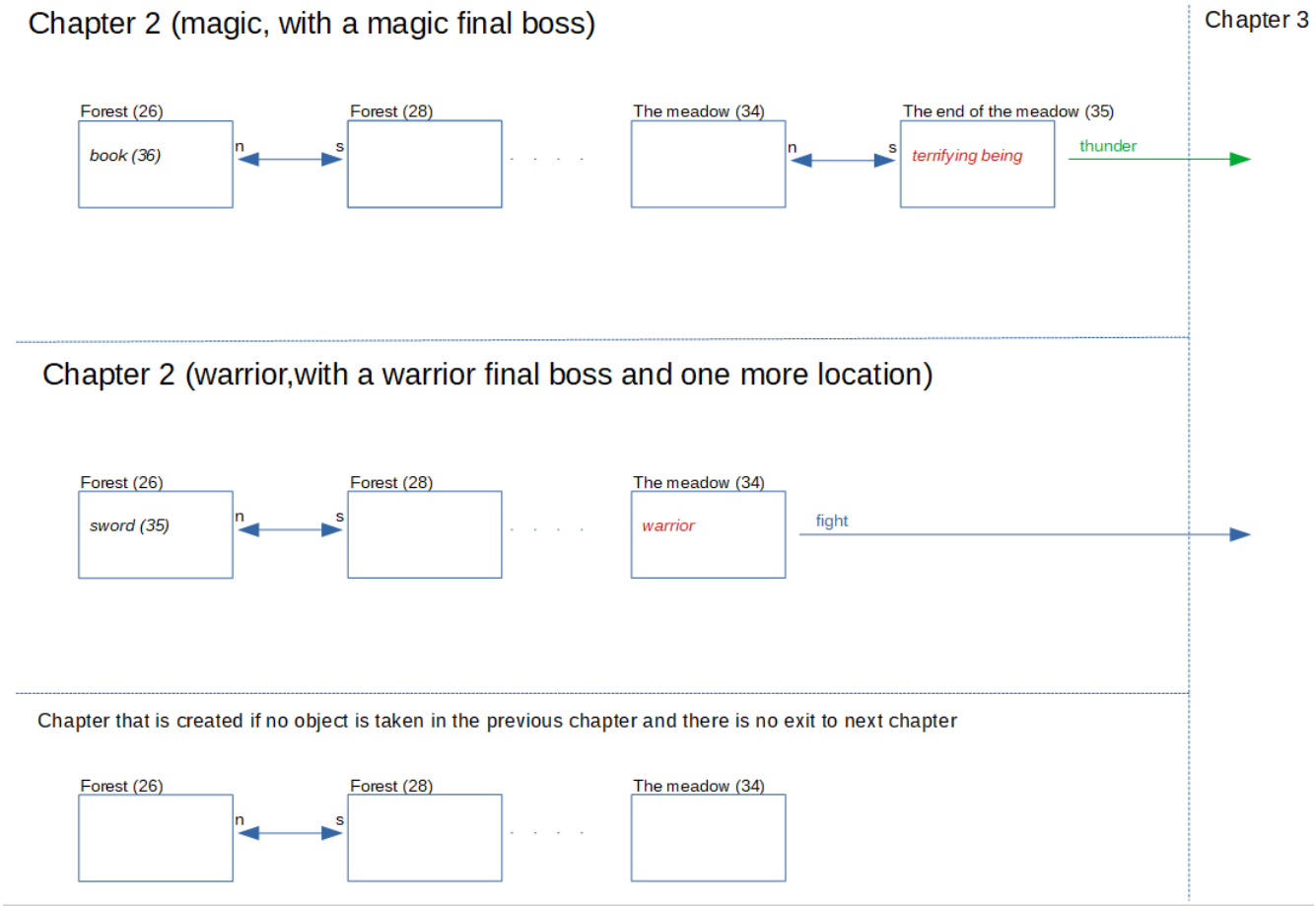

Figura 4.7 - Esquema de la transición entre el capítulo 2 al 3

4.3.3 Ejemplo de transición (del capítulo 3 al 4)

El tercer capítulo consiste en un laberinto boscoso en el que hay repartidos objetos de metal (oro, plata y bronce) que el jugador de recopilar para ofrecerlos a un tótem situado en un embarcadero al final del laberinto. Al ofrecer los objetos al tótem, el sistema calculará el valor de los mismos y, en función de ese valor, construirá el cuarto capítulo situando al jugador en otro laberinto, de modo que cuantos más y mejores objetos haya ofrecido, mejor será la posición donde se situará en el laberinto. Se debe destacar que si el jugador consigue todos los objetos, saltará del tercer capítulo al quinto (ver Figura 4.8).

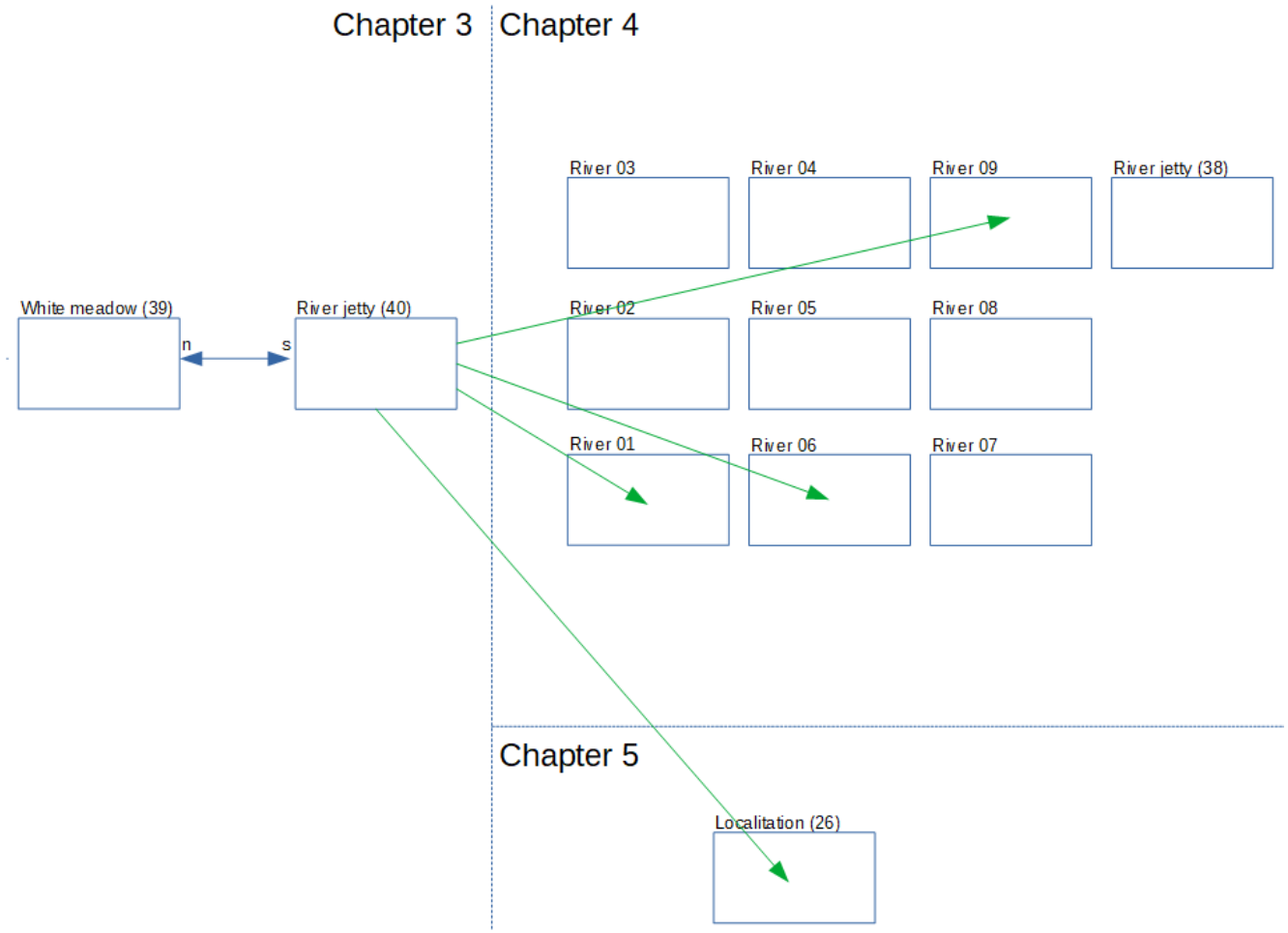

Figura 4.8 - Esquema de la transición entre el capítulo 2 al 3

El mecanismo que utiliza el sistema para determinar la posición del personaje en la transición del capítulo 3 al siguiente consiste en examinar el árbol de objetos resultante para contar los objetos que se han obtenido (ver Figura 4.9) y asignarles un valor que será evaluado por los métodos del planificador (ver Figura 4.10) conjuntamente con los operadores (ver Figura 4.11).

```
with open('objectstree.log', newline='') as File:
    reader = csv.reader(File)
    for row in reader:
        if row[0] == "21":
            lugar = row[1]
        if row[0] == "27" and row[1] == "21": bronce = 1
        if row[0] == "29" and row[1] == "21": plata = 1
        if row[0] == "32" and row[1] == "21": oro = oro + 1
        if row[0] == "38" and row[1] == "21": oro = oro + 1
```

Figura 4.9 – Código en Python que examina el árbol de objetos trás finalizar el capítulo 3 y su casuística para determinar los objetos recogidos por el jugador

```
def loc_jump_chapter(state,p):
    if int(state.oro[p]) > 1 and int(state.plata[p]) > 0 and int(state.bronce[p]) > 0:
       return [('sel_jump',p)]
    return False

def loc_after_maze(state,p):
    if int(state.oro[p]) > 0 and int(state.plata[p]) > 0:
       return [('sel_after',p)]
    return False

def loc_middle_maze(state,p):
    if int(state.plata[p]) > 0:
       return [('sel_middle',p)]
    return False

def loc_before_maze(state,p):
    if int(state.bronce[p]) > 0:
       return [('sel_before',p)]
    return False

pyhop.declare_methods('player_loc_ini',loc_jump_chapter,loc_after_maze,loc_middle_maze,loc_before_maze)
```

Figura 4.10 - Métodos del planificador del capítulo 3 para condicionar el 4

```
def sel_jump(state,p):
    global res
    res = 50
    return state

def sel_after(state,p):
    global res
    res = 20
    return state

def sel_middle(state,p):
    global res
    res = 10
    return state

def sel_before(state,p):
    global res
    res = 5
    return state

pyhop.declare_operators(sel_jump,sel_after,sel_middle,sel_before)
```

Figura 4.11 - Operadores del planificador del capítulo 3 para condicionar el 4

5 CONCLUSIONES

Podemos afirmar que hemos establecido las bases para desarrollar sistemas en los que la narrativa pueda contener elementos dinámicos que permitan un alto grado de variabilidad en lo que se refiere al desarrollo narrativo. Los mecanismos del sistema se pueden añadir a cualquier motor de juego generalista actual. y a gran parte de los géneros de videojuegos con los que se trabaja en el mercado.

Como propuesta de un futuro desarrollo, se propone aplicar los mecanismos de este trabajo sobre un motor de videjuegos actual, como Unity, que utilice el género de sandbox [SANDBOX19]. Esto permitirá explorar y expandir todos los elementos narrativos sin estar condicionado por el género de la ficción interactiva.

6 REFERENCIAS

[SECRETMONKEY19] Wikipedia, “The Secret of the Monkey Island”. https://es.wikipedia.org/wiki/The_Secret_of_Monkey_Island. (Agosto/2019).

[DWARF19] Wikipedia, “Dwart Fortress”. https://es.wikipedia.org/wiki/Dwarf_Fortress. (Agosto/2019).

[NOMANSSKY19] Wikipedia “No Man's Sky”. https://es.wikipedia.org/wiki/No_Man%27s_Sky. (Agosto/2019).

[SHORTADAMS17] Short, T., Adams, T., 2017, Procedural Generation in Game Design, Chapter 1, CRC Press.

[AVENCON19] Wikipedia, “Aventura conversacional”. https://en.wikipedia.org/wiki/Interactive_fiction. (Agosto/2019).

[COLCAVE19] Wikipedia, “Colossal Cave Adventure”. https://en.wikipedia.org/wiki/Colossal_Cave_Adventure. (Agosto/2019).

[ZORK19] Wikipedia, “Zork”. https://en.wikipedia.org/wiki/Zork. (Agosto/2019).

[HOBBIT19] Wikipedia, “The Hobbit”. https://en.wikipedia.org/wiki/The_Hobbit_(1982_video_game). (Agosto/2019).

[AVAD19] Wikipedia, “Aventuras AD”, https://es.wikipedia.org/wiki/Aventuras_AD. (Agosto/2019).

[KYBI16] B. Kybartas, R. Bidarra, “A survey on story generation techniques for authoring computational narratives”, 2016. https://ieeexplore.ieee.org/abstract/document/7439785. (Agosto/2019).

[NELSON19] G. Nelson, “The Z-Machine Standards”. https://www.inform-fiction.org/zmachine/standards/z1point1/index.html. (Agosto/2019).

[PYHOP19] D. Nau, “Pyhop”. https://bitbucket.org/dananau/pyhop/src/default/. (Agosto/2019).

[SHOP19] D. Nau, “Simple Hierarchical Ordered Planner”. http://www.cs.umd.edu/projects/shop/. (Agosto/2019).

[SANDBOX19] Wikipedia, “Sandbox”. https://es.wikipedia.org/wiki/Videojuego_no_lineal. (Agosto/2019).

[DEBIAN19] Debian. https://www.debian.org/. (Agosto/2019)

[INFORM19] Inform 6. http://www.ifarchive.org/indexes/if-archiveXinfocomXcompilersXinform6Xsource.html. (Agosto/2019)

[FROTZ19] Frotz. https://gitlab.com/DavidGriffith/frotz. (Agosto/2019)

7 APÉNDICE - INSTALACIÓN DE LAS HERRAMIENTAS

Desde la consola del sistema operativo Debian, siendo el usuario root, ejecutar:

```
su

apt-get update
apt-get install build-essential

apt-get install libncurses5-dev
apt-get install libao-dev
apt-get install libmodplug-dev
apt-get install libsamplerate-dev
apt-get install libsndfile-dev
apt-get install libvorbis-dev

exit

cd /tmp
wget http://www.ifarchive.org/if-archive/infocom/compilers/inform6/source/inform-
6.34-6.12.2.tar.gz
gunzip inform-6.34-6.12.2.tar.gz
tar xf inform-6.34-6.12.2.tar
mv inform-6.34-6.12.2 /pot/
cd /opt/inform-6.34-6.12.2/
make

cd /tmp
wget
https://gitlab.com/DavidGriffith/frotz/-/archive/4ba715ef0d56bce6981448ceb1ac3bf6f5
138726/frotz-4ba715ef0d56bce6981448ceb1ac3bf6f5138726.tar.gz
gunzip frotz-master.tar.gz
tar xf frotz-master.tar
mv frotz-master /opt/
cd /opt/frotz-master/
make
```