Gull

Adam Cadre

0.1. Cómo leer este documento

Las tres secciones de Gull son bastante independientes. Si no planeas programar en Inform en el futuro, pero tienes curiosidad sobre qué es eso de Glk del que has oido hablar a la gente, por ejemplo, puedes leer simplemente la primera parte. Si ya sabes exactamente qué es Inform Glulx, y solo quieres saber cómo tocar música con él, puedes comenzar por la sección 2. Y si todo lo que quieres es ver algunas de las cosas que Glulx puede hacer, puedes ir directamente a los juegos de ejemplo. Tú mismo.

Sin embargo, dentro de cada una de estas secciones, es una buena idea comenzar a leerlas desde el principio, y continuar en orden hasta el final. Algunos de los apartados asumen que ya estás familarizado con los apartados previos, de modo que si saltas directamente a la parte sobre la entrada desde ratón, por ejemplo, podrás sentirte confuso por las menciones a la rutina HandleGlkEvent() – que había sido explicada en apartados previos, en la parte sobre manejo de ventanas gráficas. A este nivel, Gull puede verse más bien como un tutorial. También es un suministro de código que puedes usar y personalizar: siéntete libre de cortar y pegar el código de IdentifyGlkObject() y trastear con él, por ejemplo, en lugar de tratar de memorizarlo y reconstruirlo por ti mismo, línea a línea.

Algo que Gull no es, sin embargo, es una referencia completa. Aunque una de las principales razones de ser de Gull es proporcionar una fuente de información, sobre todo más amigable que la especificación Glk, ciertamente no suplanta a ese documento. Gull no intenta ser ni la última palabra, ni la más detallada, ni siquiera pretende estar bien orginazado. Es probable que a veces necesites consultar la `http://www.eblong.com/zarf/glk`especificación Glk (en inglés), la especificación de Glulx, y la Guía para autores de juegos de Glulx Inform antes que Gull. Pero esos documentos no son especialmente comprensibles para el programador aficionado sin tener unos conocimientos básicos en el campo que abarcan. El propósito de Gull es proporcionar esos conocimientos.

0.2. Créditos

Gull fue escrito por mi, Adam Cadre. Sin embargo, lo único que hice fue escribir lo iba aprendiendo sobre estas cosas de otra gente. Cantidad de gente me explicó cosas para la sección uno, entre los que se cuentan John Cater, David Glasser, Stephen Granade, Iain Merric, Dan Shiovitz y sin duda otros muchos que ahora olvido. El material de la sección dos, sin embargo, lo aprendí casi en su totalidad de Andrew Plotkin, ya en forma directa (preguntandole cosas a través de una Wiki Web que montó David Glasser) o indirectamente leyendo su especificación de Glk. Otros comentarios me fueron suministrados en la etapa de edición por Ross Raszewski y Evin Robertson.

Gull trata de Inform Glulx. Glulx es de Andrew Plotkin; Inform es de Graham Nelson. Esto se menciona de pasada en algunos lugares de Gull, pero me pareció un buen lugar para llamar la atención sobre ello.

Capítulo 1

Una introducción a Glulx Inform

1.1. ¿Qué es una "Aventura Conversacional"?

Las aventuras conversacionales, o AC (Interactive Fiction en inglés), son un tipo de relato en el cual el oyente (lector) es un participante activo. Una aventura conversacional (generalmente llamada un "juego", incluso si no hay nada similar a un juego en ella) comienza típicamente presentando al jugador un breve escenario ("Aqui hay un estrecho callejón sin salida, con muros que se elevan opresivamente altos en tres direcciones. Una puerta de metal liso se enfrenta a tí al este, cerca del final del callejón. Está firmemente cerrada.") Entonces, el jugador decide qué quiere que el personaje a quien encarna en el juego haga – abandonar el callejón, quizás, o llamar a la puerta. Se entera de lo que pasa, dice qué hacer después, y así sigue este toma y daca hasta que el juego llega a su fin (a menudo con una horrible muerte del personaje). ¡Es divertido!

Si el autor estuviera realmente ahí, como el master de un juego de rol, el jugador podría decirle qué quiere hacer, y el autor podría pensar sobre ello un rato, quizás lanzando algunos dados, y responderle cuál fue el resultado. Pero si los cientos de personas que visitan regularmente el IF-archive (depósito de aventuras inglesas) necesitaran que los autores fueran a sus casas para poder jugar los juegos, seguro que tendrían para una larga espera. Por suerte, el medio no es tan dependiente de un contacto cara a cara. En lugar de ello, el autor codifica el juego de antemano – cada descripción de cada localidad, cada posible respuesta a cada petición del jugador – en un fichero. Los jugadores copian el fichero a su propio ordenador, leen el texto que fluye en sus pantallas, esperan a que aparezca un *prompt* (normalmente un "¿"), escriben su orden, leen la respuesta, obtienen otro *prompt*, escriben otra orden, etcétera. El apartado siguiente detalla el proceso por el cual una AC va desde la cabeza del autor hasta la pantalla del computador del jugador.

1.2. AC, de la granja al mercado

1.2.1. De la idea al código fuente

Tras jugar algunas aventuras, mucha gente decide que le gustaría intentar escribir la suya propia. El primer paso es traducir tus ideas en código fuente. El código fuente es un conjunto de instrucciones, escritas en un lenguaje de computador, que definen cosas como qué objetos hay en el juego, qué hacen, cómo reaccionan si el jugador intenta hacerles cosas, etc. Hay muchos lenguajes de computador diseñados específicamente para escribir aventuras (de ahora en adelante ACs), como TADS, Hugo o ALAN. Pero el lenguaje para IF más ampliamente usado, en el momento de escribir esto, es el llamado Inform.

Inform es un lenguaje de bastante alto nivel, lo que significa que está más cerca de la forma en que los humanos procesamos la información ("Si el jugador intenta alejar la galletita del loro, el loro debe morderle") de la forma en la que lo hacen los computadores ("001001101110010"). Un fragmento del código Inform que representaría el ejemplo del loro sería este:

```
Object loro "el loro Polly"
   with nombre 'loro' 'pajaro' 'polly',
      antes [;
         DejarSalir:
            if (uno == galletita)
              "Polly te muerde la mano cuando intentas mangarle la
              galletita ~<Groook! <Esa maano!~ declara.";
      ],
   has animado propio;
```

Incluso si no sabes Inform, puedes hacerte una buena idea de lo que hace el código anterior sólo mirándolo. Ciertamente es mucho más fácil convertir las ideas a esta forma, que directamente a una secuencia de números.

(Y si quieres aprender Inform – lo que necesitarás hacer antes de entrar en la sección dos – un buen sitio para comenzar es la página web de Inform, mantenida por el creador del lenguaje, Graham Nelson.)

Algunas piezas de un juego Inform, serán específicas para ese juego (no todos los juegos tienen un loro, por ejemplo). Pero un buen número de piezas serán bastante similares de un juego a otro. Por ejemplo, prácticamente todas las ACs necesitan un parser, la parte del juego que lee complejas frases como "PON TODO EXCEPTO LA GALLETITA EN LA JAULA DEL LORO", y lo rompe en una lista de acciones más sencilla de manejar para el programa ("ejeuta la acción Insertar con los objetos loro, alpiste y periodico, pero no con el objeto galletita, y con la jaula como destino"). Escribir un parser desde cero es extremadamente complejo – la mayoría de los parsers "cocidos en casa" (y unos pocos que suelen aparecer en cada competición de ACs) son terribles. También es una pérdida de tiempo: si una persona ha codificado un parser realmente bueno, y tú quieres que tu parser haga prácticamente lo mismo ¿por qué no puedes usar el código de esa otra persona? Respuesta: sí puedes. El parser, así como al sistema de manejo de objetos y prácticamente cada uno de los elementos típicos de las ACs que reaparecen de un juego a otro, están incluidos en la librería InformATE.

Una librería es un grupo de ficheros, escritos en el mismo lenguaje en que tú estás programando, que pueden ser incluídos en tu programa para realizar algunas funciones, sin que tengas que codificarlas por tí mismo. En inform, la cosa es tan fácil como escribir:

```
Include "EParser";
Include "Acciones";
Include "Gramatica";
```

Programa los objetos y rutinas específicos de tu juego, añade esas tres líneas en los lugares adecuados, y tu programa está completo. Sin embargo, ningún ordenador puede ejecutar código fuente Inform directamente: el código fuente debe ser traducido a código máquina.

1.3. Del código fuente al código de máquina virtual

En el corazón de cada ordenador, está la unidad central de proceso, o CPU. El trabajo de la CPU es ejecutar instrucciones, que le llegan en código máquina, que simplemente es una secuencia de dígitos binarios. Por ejemplo, "1101011010010001" `nota1.html`[nota] puede significar "vete a la posición de memoria número 106 y súmale 1 al número que encuentres allí". ¿Cómo sabe la CPU lo que significa esa secuencia de dígitos? Porque los chips de silicio de los que está hecha, están construidos de tal forma que al enviarles una secuencia de impulsos que representan ese número, producen el efecto deseado. ¿Cómo sabes tú, el programador, lo que esa secuencia de dígitos significa? A menos que estés trabajando con un lenguaje de muy bajo nivel, no necesitas saberlo. Para eso están los compiladores.

Un compilador es un programa que toma el código fuente escrito en un lenguaje de alto nivel, asequible a las personas (como Inform), y lo convierte en código máquina. Sin embargo, aquí aparece un problema. Cada tipo de máquina es diferente. Una secuencia de bits que para una máquina (como un Pentium) signfique "suma estos dos números", puede significar algo completamente diferente – o no significar nada de nada – para un computador Apple. Una solución a este problema sería coger tu código fuente, buscar una máquina que ejecute Windows, y compilarlo para obtener un fichero ejecutable para Windows – esto es, un programa que puede ejecutarse por si solo, como cualquier otro programa que puedas usar. Después, buscar un Mac y compilar un ejecutable para el Mac. Después pillar una Palm Pilot y compilar una versión para ella. Y después otra para el Acorn, y otra para el Amiga, y después otra cada vez que aparezca un nuevo tipo de ordenador... es fácil ver por qué esta no es realmente la mejor solución, si quieres que gente con diferentes tipos de máquina puedan jugar tu juego.

En cambio, la solución usada por la comunidad de ACs, es compilar sus programas para máquinas virtuales. En lugar de compilar para el Mac, o el DOS, los usuarios de TADS compilan para la "máquina virtual TADS". Los usuarios de Hugo compilan para la "máquina virtual Hugo", y los usuarios de Inform compilan para la "máquina Z". Los documentos con la especificación de la máquina Z son como los documentos que especifican cualquier otra máquina real: explican a los creadores de compiladores cómo es el código máquina que la máquina Z es capaz de comprender. La única diferencia es que nunca se han construido realmente máquinas Z: ninguna fábrica conecta chipZ de ZiliZio a una placa baZe, y no encontrarás portatileZ en venta en las tiendas de ordenadores. Pero aún así, puedes compilar programas para ella.

Así que has tenido una idea para un juego, has aprendido Inform, has escrito el código fuente, y lo has compilado. Lo que tienes ahora es un fichero con el juego, diseñado para ser ejecutado en una máquina que no existe. ¿Dónde está la gracia? Sigue leyendo.

1.3.1. Del código máquina virtual a tu pantalla

Puede que te sea familiar el concepto de **emulador**. Un emulador es un programa que te permite ejecutar programas compilados para un ordenador diferente del tuyo, mediante la traducción del código máquina extranjero en el código máquina nativo de tu ordenador. Muchos usuarios de Mac tienen un emulador llamado "Virtual

PC" que les permite ejecutar Microsoft Windows, por ejemplo. Las ACs (sobre todo las inglesas) hacen un intenso uso de este concepto: cada máquina virtual, tiene una serie de **intérpretes** asociados, que son programas que permiten que los ficheros compilados para esa máquina virtual puedan ser jugados en un tipo concreto de ordenador. MaxTADS, por ejemplo, es un intérprete que toma los juegos hechos para la máquina virtual TADS y los ejecuta en el Macintosh. WinFrotz es un intérprete que toma juegos en código para la máquina Z y los ejecuta en máquinas Windows. La máquina Z en particular tiene intérpretes para un asombroso número de plataformas, incluyendo la GameBoy (!). Así que, aunque algunos juegos para máquina Z se distribuyan en forma de ejecutables (usando un programa como JZEXE que junta el fichero del juego con un intérprete específico para una plataforma), la forma más habitual de jugar ACs es bajarse por separado el fichero con el juego que quieres jugar, y el intérprete que puede ejecutarlo en tu máquina, arrancar el intérprete, cargar el juego en él, y empezar a jugar. Y así es como el juego llega desde la cabeza del autor a tu pantalla.

Puede que te preguntes ¿pero el paso intermedio de especificar una máquina virtual, no ocasiona simplemente un trabajo extra? Cierto, significa que no tendrás que compilar tu juego para docenas de plataformas diferentes – pero tienes que compilar intérpretes para todas estas plataformas diferentes, de modo que ¿es esto más eficiente? La respuesta es: no lo es, si sólo se va a escribir un juego para la máquina virtual en cuestión. Pero tan pronto como escribas un segundo juego, resulta obvio que usar una máquina virtual merece la pena: compilas tu nuevo juego para la máquina virtual, y el fichero resultante puede ser jugado en todos los intérpretes que ya habían sido creados. Así que si haces un juego con Inform, no necesitas molestarte en compilarlo para Windows, para Macs, para Acorns y para GameBoys – simplemente lo compilas para la máquina Z, y toda la gente con Windows y Mac y Acorn y GameBoy pueden instantáneamente jugarlo, a través de sus intérpretes escritos hace años.

Y hasta hace poco, esto habría sido el final del cuento. Pero recientemente, ha sido introducido un nuevo elemento para ahorrar esfuerzos, que recibió el muy denigrado nombre de Glk.

Capítulo 2

En sus pantallas gracias a la letra “G”

2.1. Glk

Hay una tremenda cantidad de lenguajes para ACs, cada uno con su propia máquina virtual, y aún más están siendo creados. Hay también un buen montón de plataformas ahí afuera, y nuevo hardware y nuevos sistemas operativos están apareciendo también a un ritmo nada desdeñable. Esto ha traido un problema que debería sonar bastante familiar si has leído los apartados previos. Cada vez que un nuevo sistema para ACs aparece, es necesario escribir un intérprete para cada una de las plataformas en las que se quieran jugar los juegos creados con ese sistema: Windows, DOS, Mac, Acorn, Amiga, BeOS, Unix, y muchos muchos más. Y cada vez que una nueva plataforma en la que podrían jugarse ACs aparece –digamos que la gente quiere jugarlas desde dentro de un MUD, o a través de un teléfono móvil conectado a Internet– la nueva plataforma necesita un intérprete Z, y un íntérprete TADS, y un intérprete Hugo, y un intérprete ALAN, y así seguiríamos.

Más aún, una buena parte del trabajo necesario para escribir un intérprete, es esfuerzo malgastado. Mira, un intérprete tiene que hacer dos cosas más o menos independientes: procesar datos – parsear, mantener el estado de los objetos del juego, cosas así – y entrada/salida (abreviado como E/S, o en inglés I/O), que se ocupa de cómo debe mostrarse el texto que el programa ordena imprimir. Y salvo pocas excepciones (de las que nos ocuparemos en breve), la parte de proceso de datos no se preocupa de la parte de E/S. Simplemente la dirá a la parte de E/S “eh, imprime estas tres frases” (bueno, se lo dirá en código máquina), pero a la parte que procesa datos no le importa si esas frases aparecen con un tipo de letra caprichoso en una ventana gráfica con barras de scroll. De igual forma, la parte que se ocupa de la E/S sólo debe preocuparse de mostrar el material que le han ordenado mostrar. No le interesan los algoritmos por los que la parte de proceso de datos ha llegado a su decisión de qué imprimir. Sólo se interesa de imprimirlo realmente. Esto significa que, mientras que la parte de proceso de datos de todos los intérpretes de máquina Z será prácticamente idéntica, la parte de E/S variará drásticamente: el Frotz de DOS, diseñado para ser usado en un entorno en el que no hay posibilidad de varios tipos de letra, requiere unas rutinas para imprimir texto muy diferentes de las de WinFrotz. Y de forma similar, mientras que la parte de proceso de datos de diferentes intérpretes para Mac será bastante diferente – TADS e Inform y AGT y otros, simplemente no manejan los datos de la misma forma – en cambio la parte del intérprete que maneja el “look and feel”, es decir la E/S, será muy similar, con independencia de qué máquina virtual esté siendo usada. Pero ya que hay que crear cada posible par, el trabajo acaba siendo repetido muchas veces.

Permitanme presentarles Glk. Glk es un intento de hacer todo este trabajo más eficiente, separando las dos funciones del intérprete de ACs tradicional. Lo hace definiendo unas cuantas rutinas, escritas en el lenguaje C de programación, y recopiladas en la librería Glk. Los programadores familiarizados con el C pueden escribir **aplicaciones Glk**, que hagan sus cálculos y proceso de datos, y generen sus resultados en una forma que Glk pueda comprender, y **librerías Glk** (también llamadas “implementaciones de Glk”) que adapten las funciones Glk para una plataforma concreta: WinGlk, XGkl, MacGlk, etc. El escritor de intérpretes ve reducido su trabajo a juntar las aplicaciones con las librerías.

Si no está aún clara la utilidad de todo esto, miralo de esta otra forma. Digamos que en lugar de intentar llevar los lenguajes de ACs a las diferentes plataformas, lo que estamos intentando construir es un sistema ferroviario para llevar gente desde varios lugares de la costa oeste de los Estados Unidos – Seattle, Portland, San Francisco, Los Angeles, etc. – a ciudades en la costa este: Boston, Nueva York, Filadelfia, Washington, etc. El método anterior a Glk de líneas directas significaría construir vías desde Seattle a Boston, de Seattle a Nueva York, de Seattle a Filadelfia, etc. y después de Portland a Boston, de Portland a Nueva York, etc. Si quisieramos añadir una nueva ciudad de partida, sería necesario construir un conjunto nuevo de vias: San Diego a Boston, San Diego a Nueva York, etc. Y lo mismo ocurriría si añadimos un destino nuevo: tendríamos que construir de Seattle a Miami, de Portland a Miami, de San Francisco a Miami, etc. Lo que hace Glk es crear un centro. En nuestro ejemplo, digamos que es Omaha (Nebraska). Así, en lugar de construir vías directamente desde cada origen a cada destino, construimos de Seattle a Omaha, Portland a Omaha, San Franciso a Omaha, etc. y después de Omaha a Boston, Omaha a Nueva York, etc. Esto reduce drásticamente el número de vías que

tenemos que construir. Y si aparece una nueva ciudad de origen, sólo tenemos que construir una vía: San Diego a Omaha. ¿Un nuevo destino? Una vía: Omaha a Miami. Y la gente que venga desde San Diego, hará transbordo en Omaha y llegará a cualquier destino, y la gente que venga desde cualquier origen en la costa oeste, puede hacer transbordo en Omaha y llegar a Miami.

Así que pongamos que alguien escribe un nuevo lenguaje para hacer ACs, llamado ABC. No se necesita escribir intérpretes para ABC-Windows, ABC-Mac, ABC-Palm; en lugar de esto, simplemente escribe un intérprete ABC-Glk y todo el mundo en todas las plataformas para las que existan librerías Glk, podrán jugar juegos ABC. De forma análoga, si Toaster Technologias construyera el Cybertoaster, basta escribir una librería Glk-para-Cybertoaster, y ya tenemos instantáneamente al Cybertoaster preparado para jugar juegos Inform, TADS, Hugo o juegos de cualquier otro sistema para el cual exista un intérprete basado en Glk. Todo lo que los usuarios de Cybertoaster necesitan hacer es enlazar esos intérpretes con las librerías Glk para Cybertoaster.

Puede que te preguntes: los intérpretes basados en Glk, toman la salida de las máquinas virtuales y la convierten en algo que Glk pueda comprender. ¿Hay alguna máquina virtual cuya salida ya esté preparada para Glk sin requerir conversiones? La respuesta: puedes apostar que sí. Eso es Glulx.

2.2. Glulx e Inform Glulx

Hay, como se ha dicho en los apartados previos, muchas máquinas virtuales diseñadas para ACs. Algunas de ellas – la máquina virtual TADS y la máquina virtual Hugo, por ejemplo – pueden hacer cosas que la máquina Z no puede. Pueden manejar juegos grandes (más de 512k), por ejemplo, y hacer que muestren dibujos o que toquen sonidos, es mucho más simple que lograr lo mismo en la máquina Z. Sin embargo, para usar una de ellas, tienes que aprenderte el lenguaje asociado: TADS o Hugo. Muchos programadores Infrm han invertido mucho tiempo y energía para aprender Inform, y preferirían no tener que empezar a aprender un nuevo lenguaje desde cero.

Glulx fue diseñado para resolver este problema. Glulx es una máquina virtual de 32 bits `nota2.html`[nota] cuyos compiladores están diseñados para tomar código fuente Inform y convertirlo en código máquina Glulx. Así que los programadores Inform pueden liberarse de las limitaciones de la máquina Z, y aprovecharse de la nuevas capacidades de la máquina Glulx, sin tener que aprender un lenguaje nuevo. Al menos en su mayor parte.

Recordarás de la sección sobre Glk que había una excepción notable a la regla de que el proceso de datos y la entrada/salida son cosas separadas en la programación de ACs. La excepción ocurre cuando los programadores insertan *opcodes* en sus programas.

Un *opcode* es una instrucción que se salta algunos de los pasos habituales en el proceso de programación. Imagina este proceso como una torre de robots, cada uno sobre los hombros de otro. Cada robot, sólo habla con el que tiene debajo. Así, un robot de la parte alta, le dice al robot que tiene debajo "Eh, imprime esta frase". Y este robot le dice al siguiente robot de debajo (el robot de E/S), "Vale, imprime la letra A en esta posición de la pantalla, en este color, y después imprime la letra B en esta otra posición, en este color..." Y ese robot lo traduce en "Pon puntos blancos aqui, y aqui, y aqui, y aqui..." Al final, verás unos puntitos blancos en tu pantalla, que parecen formar letras, que forman palabras que lees como parte de una historia.

Pero ¿y si no quieres dejar a los robots la decisión de dónde deben ir las letras y qué aspecto deben tener? ¿y si quieres quitar de enmedio uno de los robots y hablar directamente con el robot de E/S y decirle "Vale, quiero la letra Q, en negrita en la posición (1,5) de la pantalla, y en color verde brillante"? Aquí es donde aparecen los opcodes. Normalmente, si tu juego está en Inform, tu escribirás todas tus instrucciones en Inform, y después dejarás al compilador que las convierta en código máquina. Pero a veces, cuando quieres un control más preciso – como cuando estás tratando de personalizar tu línea de estado, o si quieres algunos efectos especiales, como letras bailando en la pantalla de presentación– necesitas ponerlo en opcodes. Si estás compilando para la máquina Z, estos opcodes estarán escritos en ensamblador de la máquina Z, y tendrán un aspecto como `@set_colour 2 4;` (que significa "imprime a partir de ahora con texto negro sobre fondo verde") o `@read_char 1 8 dummy i;` (que significa "Espera hasta que se haya pulsado una tecla, o hasta que hayan transcurrido 8 décimas de segundo, lo que ocurra antes"). Los opcodes del ensamblador-Z están explicados en el "Inform Designer's Manual" y durante mucho tiempo se les consideró simplemente otra parte de Inform.

Sin embargo, la llegada de Glulx cambió las cosas. Mira, el compilador clásico de Inform sabía cómo convertir el código fuente Inform en código máquina-Z y como cambiar los opcodes del ensamblador-Z en código máquina-Z. Pero mientras que el compilador de Glulx Inform sabe convertir el código fuente Inform en código máquina-Glulx, en cambio el ensamblador-Z le suena a chino. El compilador de Inform Glulx necesita que los opcodes estén escritos para Glk (en ensamblador de la máquina Glulx, en realidad). La librería estándar InformATE (la versión 6/7) incluye opcodes en ensamblador-Z, así que no compilará con Inform Glulx: necesitarás usar la librería InformATE biplataforma (versión 6/10), que está diseñada para que pueda usarse con juegos para la máquina Glulx o para la máquina Z (incluye opcodes para ambas, y elige unos u otros según cuál sea la máquina elegida como destino).

Esto quiere decir que todos los truquitos que los autores de Inform habían aprendido relacionados con opcodes de máquina-Z, son inútiles para crear juegos Glulx. Si quieren que su juego compile para Glulx, necesitarán

aprender todo un nuevo sistema para especificar el estilo del texto, crear efectos especiales, mostrar imágenes, tocar música, etc. Y de esto trata el capítulo dos.

Capítulo 3

Cómo hacer cosas con Glulx Inform

3.1. Arrancando

3.1.1. Lo que necesitarás para usar Glulx Inform

Para compilar un juego con Inform Glulx, necesitarás lo siguiente:

- El código fuente que va a ser compilado. Puedes crearlo tú mismo, en un editor de textos (*no* en un procesador de textos).
- La librería InformATE 6/10 biplataforma.
- Infglk, un suplemento para la librería biplataforma, que proporciona un buen número de rutinas, que permiten a los programadores de Inform Glulx sacar partido de las capacidades multimedia de Glk, sin tener que usar el opcode `@glk` ni la tabla que relaciona los números con las funciones. Simplemente, descarga la última versión del paquete `Infglk` de la sección Glulx del if-archive, extrae los ficheros en la misma carpeta donde trabajarás con Inform Glulx, e incluye la siguiente línea en tu código fuente (justo después del punto donde incluyes `Acciones`):

  ```
  Include "infglk";
  ```

  Todas las secciones siguientes suponen que usas `infglk`, así que consideralo obligatorio.
- Un compilador de Inform Glulx para tu plataforma.

Y para ejecutar el programa que has compilado, necesitarás:

- Un intérprete Glulx para tu plataforma, cuyo nombre será probablemente una variación de "glulxe" (la "e" final es de "ejecutar").
- Una versión de las librerías Glk para tu plataforma, que proporciona al intérprete los recursos que necesita para correr.

Y si quieres incluir sonido o gráficos en tu juego, aún necesitas otra cosa más: **Blorb** ¿Qué es eso? Sigue leyendo...

3.1.2. Blorb

Como ya hemos señalado en secciones previas, la comunidad (inglesa) de programadores de ACs, tiende a dar una prioridad muy alta al hecho de que los juegos y herramientas para ACs sean utilizables por usuarios del mayor número posible de plataformas. Pero esta accesibilidad tiene un precio. Incluso el sólo hecho de jugar una AC generalmente requiere descargar una pareja de piezas diferentes: el intérprete apropiado para la máquina de uno, y el fichero con el juego; ahora imagina lo que ocurrirá si metemos la multimedia en la mezcla. Ya sería lo bastante malo si simplemente fuera cosa de coger unos cientos de imágenes y un generoso suministro de ficheros musicales y meterlos todos en un archivo zip junto con el fichero del juego, pero es más lioso que eso. Incluso si consigues que todos tus ficheros vayan del ordenador A al ordenador B, sin perder nada por el camino ¿quién dice que el ordenador B será capaz de comprenderlos? ¡Hasta los nombres de fichero pueden ser un problema cuando se intentan compartir datos entre diferentes tipos de plataforma! Las máquinas DOS y Windows requieren que los ficheros tengan una extensión de tres letras apropiada, para poder usarlos; las máquinas Acorn (como la que usó Graham para crear Inform) ni siquiera permiten extensiones en los nombres. Ser capaz de meter en un solo fichero todos los recursos que un juego necesita, junto con el propio juego, de

forma que pueda ser leido por diferentes intérpretes en muchas plataformas diferentes, sería un gran avance. Para esto es Blorb.

Para crear un fichero Blorb, necesitarás lo siguiente:

- Un fichero de "recursos blorb" (básicamente es una lista de qué ficheros quieres empaquetar juntos)
- Las imágenes y/o sonidos que usas
- El código fuente de tu juego Inform y las librerías que requiera.
- Un compilador de Inform Glulx para tu plataforma
- Un blorbificador para tu plataforma

Por desgracia, Blorb es muy nuevo, y las herramientas "blorbificadoras" aún no han sido estandarizadas, ni está muy ampliamente soportado de momento. Hasta que esto sea así, tenemos que apañarnos con las herramientas específicas disponibles para cada plataforma. Elige de la lista siguiente (aunque de momento, tus posibles elecciones son, ejem, limitadas):

- Blorb para DOS/Windows

Para más información sobre Blorb, mira en la página de Blorb de Graham Nelson y en la página de Blorb de Andrew Plotkin.

3.2. Test de capacidades

Imagina que tienes un juego que transcurre en una ciudad del salvaje Oeste, y que cuando el protagonista entra a la ciudad, pasa junto a un cartel que dice "Perro Muerto, Kansas, Población 213" con la pintura gastada sobre un madero estropeado y acribillado a balazos. El jugador sin embargo no sabe lo que pone hasta que no ordena "EXAMINA CARTEL", la descripción del lugar simplemente tiene la línea "Aquí ves un cartel" o algo así, para indicar que está ahí. Y digamos que tienes una magnífica imagen de un estropeado cartel con balazos. Lo ideal sería que pudieras mostrar la imagen si el intérprete que usa el jugador puede manejar gráficos, o imprimir un párrafo describiendo el cartel si no. Para esto sirve `glk_gestalt()`

Hay como una docena de cosas que puedes testear. En este ejemplo, la constante a usar sería `gestalt_Graphics`, y el código del cartel sería más o menos así:

```
Object  cartel "cartel erosionado"
 with   nombre 'cartel' 'erosionado',
        descripcion [;
            if (glk_gestalt(gestalt_Graphics, 0))
            {
                ! Aqui iría el código para mostrar el gráfico
            } else
                "El cartel dice ~PERRO MUERTO, KANSAS, POBL. 213.~ O
                quizás ponía 2130, o 21300, alguien ha volado la
                esquina del cartel de un disparo.";
        ],
 has    estatico;
```

Observa el `0` en la llamada a `glk_gestalt` – es obligatorio, porque `glk_gestalt` siempre requiere dos parámetros, incluso si el segundo no se necesita como en este caso.

He aqui algunas constantes que puedes usar con `glk_gestalt`:

Constante	**Segundo parámetro**	**Qué retorna**
`gestalt_Version`	0	Un número de 32 bits que lleva codificado el número de versión de Glk (los 16 primeros bits, el número de versión mayor, los 8 siguientes el número menor y los 8 últimos el número sub-menor, de modo que la versión 0.6.1 se codificaría como `00000601`

`gestalt_CharOuput`	código de un carácter	Esto es para comprobar si el intérprete es capaz de mostrar un carácter determinado (como la `ñ`). El resultado es uno de tres posibles valores: `gestalt_CharOuput_ExactPrint` si puede mostrarlo, `gestalt_CharOutput_CannotPrint` si no puede, o `gestalt_CharOutput_ApproxPrint` si lo cambia por algo parecido (por ejemplo `n` en vez de `ñ`)
`gestalt_LineInput`	código de un carácter	Como el anterior, pero para comprobar si ese carácter es admitido durante la lectura de una línea (o sea, la entrada que es procesada después de haber pulsado Intro)
`gestalt_CharInput`	código de un carácter	Como el anterior, pero para comprobar si ese carácter es admitido durante la lectura de una tecla (un tipo de entrada en el que sólo se procesa un carácter de cada vez)
`gestalt_MouseInput`	tipo de ventana	1 si se soporta la entrada desde ratón, 0 si no
`gestalt_Timer`	0	1 si se soporta el tiempo real, 0 si no
`gestalt_Graphics`	0	1 si se soportan gráficos, 0 si no
`gestalt_DrawImage`	tipo de ventana	Este es para probar específicamente si se pueden colocar imágenes en un tipo concreto de ventana (`wintype_TextBuffer` o `wintype_Graphics`): 1 si se puede, 0 si no
`gestalt_GraphicsTransparency`	0	Esta es para comprobar si las imágenes PNG con zonas transparentes aparecerán realmente con la transparencia funcionando como debe. 1 si es así, 0 si no
`gestalt_Sound`	0	1 si hay sonido disponible, 0 si no
`gestalt_SoundMusic`	0	1 si se puede interpretar música MOD, 0 si no (esto puede retornar cero incluso si `glk_gestalt(gestalt_Sound, 0)` retornó 1 – en este caso sólo se soportan efectos de sonido AIFF
`gestalt_SoundVolume`	0	1 si funciona la función `glk_schannel_set_volume()`, 0 si no
`gestalt_SoundNotify`	0	1 si se puede hacer que `HandleGlkEvent()` ejecute el código que el programador desee en el momento que un sonido finaliza, 0 si no
`gestalt_Hyperlinks`	0	1 si se soportan los hipervínculos, 0 si no
`gestalt_HyperlinkInput`	tipo de ventana	1 si los hipervínculos están soportados en ese tipo concreto de ventana, 0 si no.

Las siguientes secciones explicarán cómo sacar partido de estas capacidades – pero a cuando leas esas secciones, recuerda que antes de mostrar una imagen, o tocar música, o declarar un hipervínculo, o lo que sea, debes antes usar `glk_gestalt()` para comprobar si el intérprete del jugador puede hacer esas cosas. Si no, necesitarás preparar las cosas de forma alternativa – descripciones textuales de las imágenes, ese tipo de cosas – o al menos mostrar un mensaje advirtiendo al jugador de qué capacidades necesitaría en su intérprete para poder jugar tu juego. Idealmente, deberías hacer una llamada a `glk_gestalt` cada vez que fueras a usar una de las características opcionales de Glk – si solo haces la comprobación en `Inicializar`, te estás buscando problemas, si por ejemplo, el jugador comienza un juego en su lujoso ordenador de sobremesa, después guarda la partida y la carga de nuevo en su Palm para jugar en el avión más tarde. (Esto no implica necesariamente tener que escrbir un montón de `glk_gestalt()` – puedes por ejemplo crear una rutina `MiDibujarGlk()`, que haga las comprobaciones necesarias y después muestre la imagen en la forma que quieras)

3.3. Ventanas Glk

Cuando arrancas un intérprete Glulx, que llamaremos "Glulxe" de ahora en adelante, el sistema operativo de tu ordenador reservará un trozo de la pantalla para él. Esto puede ser simplemente una ventana más entre otras, como en Windows o en el Mac; o alternativamente puede ser la pantalla completa, como en DOS, que sólo puede ejecutar un programa de cada vez. Sea como sea, en lo que concierne al juego, el espacio reservado

para Glulxe es el mundo completo. El juego no sabe de y ni puede afectar a nada de lo que está fuera del espacio de pantalla de Glulxe.

Dentro del espacio de pantalla de Glulxe, al principio, no hay nada. Y no puede ponerse nada allí hasta haber creado una ventana Glk. Una ventana Glk es una porción del espacio de pantalla de Glulxe, que puede mostrar cosas y a veces aceptar input. Hay tres tipos de ventanas que el programador de juegos debe conocer:

- **Ventanas de buffer de texto**. Este tipo de ventanas muestra texto, e incluso imágenes, pero lo hace en una forma de "flujo" (*stream*). Esto es, cuando el juego le dice a la ventana "imprime esta frase" o "muestra esta imagen", la frase o la imagen son colocados justo después del texto o imágenes que hubiera ya en esa ventana. Cuando la ventana de buffer de texto se llena, el material más antiguo se desliza hacia arriba y desaparece de la vista; en algunos intérpretes este material desaparece de la pantalla para siempre, mientras que otros pueden proporcionar una barra de desplazamiento que te permita retroceder páginas y ver parte de lo que ha desaparecido. La ventana donde los jugadores teclean sus comandos será normalmente una ventana de tipo buffer de texto, salvo en los casos más excéntricos.

- **Ventanas de rejilla de texto** Este tipo de ventanas también muestran texto, pero lo hacen en una forma diferente a las ventanas de buffer de texto. Las ventanas de rejilla de texto permiten al programador colocar los caracteres donde él quiera, como si colocara fichas de Scrabble en el tablero. Esto las hace ideales para el caso en que quieras sobreescribir información vieja, como en la línea de estado que usan la mayoría de los juegos. Las ventanas de rejilla de texto pueden aceptar entrada desde el teclado– flechas arriba y abajo para mover la elección en un menú, por ejemplo – pero intentar algo más avanzado requeriría manipular un poco la librería.

- **Ventanas gráficas**. Este tipo de ventana no puede mostrar texto en absoluto, pero a cambio permiten al programador mostrar imágenes (en formato JPEG o PNG) que no "deslizarán" hacia arriba a medida que el juego progrese. El programador puede incluso crear sus propias imágenes usando los comandos de dibujo de Inform Glulx, si tiene la paciencia necesaria. Las ventanas gráficas no pueden aceptar entrada desde el teclado, pero pueden reconocer las pulsaciones del ratón (una interesante capacidad, que comparten con las ventanas de tipo rejilla de texto).

Los detalles de cómo usar cada uno de estos tipos de ventana serán cubiertos en las secciones siguientes. El resto de esta sección tratará de cómo crear y construir una disposición de ventanas personalizada para tu juego.

Hay dos ventanas que son cradas por la librería InformATE automáticamente (aunque esto puede evitarse programando una función llamada `InitGlkWindow()`, enseguida veremos más sobre esto). La primera ventana es una ventana de buffer de texto, llamada `gg_mainwin`, que llena todo el espacio de pantalla de Glulxe. Esta ventana es inmediatamente dividida en dos, con una ventana de rejilla de texto llamada `gg_statuswin` ocupando una línea en la parte superior del espacio de pantalla de Glulxe, y `gg_mainwin` ocupando el resto.

Si quieres añadir más ventanas, lo que tienes que hacer es esto. Primero, decide qué tipo de ventana deberá ser. Quizás quieras un dibujo de un mapa del mundo del juego, visible permanentemente en la pantalla. Esto requeriría una ventana gráfica. Después, piensa un nombre para la ventana. Puedes ponerle cualquiera, pero ya que la librería empieza todos los nombres de ventana con `gg_`, puedes hacer lo mismo tú tambien. De modo que la llamaremos `gg_mapawin`.

Ahora tienes que elegir un "valor roca" para esta ventana. Este ha de ser un número cualquiera mayor o igual a 210. (Si no sabes qué es un "valor roca", no te desanimes – simplemente elige un número mayor o igual a 210, que no hayas usado ya para otra ventana, y prosigue). Para este ejemplo, elegiremos el 210. El siguiente paso es añadir un par de líneas a tu código. En la zona donde defines constantes, pon esta línea:

```
Constant GG_MAPAWIN_ROCA 210;
```

y en tu zona de variables globales, pon esta línea:

```
Global gg_mapawin = 0;
```

Con estas líneas añadidas al código, estamos preparados para crear realmente nuestra nueva ventana. La instrucción clave aquí será `glk_window_open()`. Dentro de los paréntesis van cinco parámetros:

1. Primero, el nombre de la ventana que hay que partir en dos. En este caso partiremos la ventana principal, así que pondremos `gg_mainwin` aquí.

2. Después, el método que queremos usar para partir la ventana. Este es un proceso en dos partes. Primero, necesitarás elegir uno de los siguientes cuatro métodos:

   - `winmethod_Above`: divide la ventana en dos partes, una encima de otra, y la nueva ventana será la parte superior (en inglés, *above* es "encima")
   - `winmethod_Below`: divide la ventana en dos partes, una encima de otra, y la nueva ventana será la parte inferior (en inglés, *below* es "debajo")

- `winmethod_Left`: divide la ventana en dos pates, una al lado de la otra, y la nueva ventna será la parte de la izquierda (*left* en inglés)
- `winmethod_Right`:divide la ventana en dos pates, una al lado de la otra, y la nueva ventna será la parte de la derecha (*right* en inglés)

Después tienes que elegir una de las opciones siguientes:

- `winmethod_Fixed`: reserva un cierto número de líneas o columnas (para las ventanas tipo texto), o de pixels (para las ventanas gráficas) para la nueva ventana.
- `winmethod_Proportional`: reserva un cierto porcentaja de la ventana original para la nueva ventana.

Junta estas dos elecciones con un signo `+`. Así, por ejemplo, si queremos que el mapa del juego aparezca a la izquierda del espacio de pantalla de Glulxe, y sabemos que el mapa tiene 240 pixels de ancho, probablemente estaremos inclinados a usar `(winmethod_Left+winmethod_Fixed)`. ¿Dónde va el 240? Vaya, es el...

3. ...tercer argumento, donde podemos poner: bien el número de líneas o columnas que deben reservarse para la nueva ventana (si es una ventana de texto y se había elegido `winmethod_Fixed`); o el número de pixels que deben reservarse para la nueva ventana (si es una ventana gráfica y se había elegido `winmethod_Fixed`); o bien, si habíamos elegido `winmethod_Proportional`, el porcentaje de la ventana original que queremos reservar para la nueva ventana (por ejemplo, pondríamos 50 si quisiéramos dividir la ventana a la mitad, o 33 si quisiéramos reservar 1/3 del espacio que ocupaba la ventana original para la nueva ventana).
4. A continuación, ponemos qué tipo de ventana queremos: `wintype_TextBuffer` para una ventana de tipo buffer de texto, `wintype_TextGrid` para una ventana de tipo rejilla de texto, o `wintype_Graphics` para una ventana gráfica. En este ejemplo, queremos el último caso.
5. El último elemento de la lista es la constante que has declarado antes. En nuestro ejemplo, sería `GG_MAPAWIN_ROCA`

Esto nos deja la siguiente llamada:

```
gg_mapawin = glk_window_open(gg_mainwin,
                (winmethod_Left+winmethod_Fixed), 240,
                 wintype_Graphics, GG_MAPAWIN_ROCA);
```

Por supuesto, soltar una línea como esta en mitad de tu programa, es una mala idea. Hay que comprobar una serie de cosas antes. Para empezar, antes de abrir una ventana debes estar seguro de que la ventana no está ya abierta. ¿Cómo podría estarlo? Quizás el jugador acaba de restaurar una partida guardada (lo que no restaura automáticamente la configuración de ventanas que había en el momento de guardar la partida, esto tenes que hacerlo tú. Cómo hacer esto se explicará en breve). Así que envolvamos la línea en una condición:

```
if (gg_mapawin == 0) {
  gg_mapawin = glk_window_open(gg_mainwin,
                  (winmethod_Left+winmethod_Fixed), 240,
                   wintype_Graphics, GG_MAPAWIN_ROCA);
  }
```

Otra cosa a tener en cuenta, es que sólo porque hayas pedido que se abra una ventna, eso no significa que realmente se haya abierto. El jugador podría estar usando un intérprete que no soporta gráficos, por ejemplo, en cuyo caso tu intento de crear una ventana gráfica estará condenado al fracaso. O puede que el intérprete soporte gráficos, pero que el jugador haya reducido la zona de pantalla de Glulxe a menos de 240 pixels de ancho, por lo que la ventana que obtienes no es exactamente la que habías pedido. Deberías comprobar estas cosas; pero lo que hagas con los resultados de las comprobaciones queda a tu criterio. Por ejemplo, si tu juego es incomprensible sin el gráfico, entonces deberías comprobar si tu ventana gráfica realmente se ha abierto (con un test del tipo "`if (gg_mapawin == 0)`" y, si no se abrió, imprimir un mensaje como "Este juego necesita un intérprete capaz de mostrar gráficos. ¡Lo siento!" y finalizar el juego. Pero si el gráfico no es más que un añadido sin importancia, entonces podrías querer que el juego prosiguiera, pero en ese caso asegúrate de que no intentas nunca hacer un `glk_image_draw()` para poner una imagen en esa (inexistente) ventana – comprueba si la ventana es cero cada vez.

Para cerrar una ventana, usa el comando `glk_window_close()`. Esta rutina necesita en realidad dos argumentos. El primero es el nombre de la ventana a cerrar. El segundo es extremadamente esotérico; si simplemente pones 0, todo irá bien.

3.4. Estilos de texto

Como hemos dicho antes, aquellos que quieran compilar sus juegos (preexistentes en forma de código fuente Inform) para GLulx, tienen que asegurarse de haber reemplazado cualquier instrucción de ensamblador Z, por ensamblador Glulx (o llamar a rutinas de Inform Glulx que hagan lo que antes hacía con el ensamblador Z). El otro cambio importante que los programadores de juegos necesitarán hacer es reemplazar cualquier comando como `style bold;` y `style underline;` – Glulx no los reconoce. En su lugar, tienes que poner llamadas a los cambios de estilo Inform Glulx.

Las llamadas Inform Glulx para cambios de estilo son muy similares a los cambios de estilo del Inform clásico. Usas el comando `glk_set_style()` con el estilo que quieres aplicar dentro de los paréntesis, y todo el texto que imprimas después de eso aparecerá en el estilo especificado. No hay forma de "desactivar" un estilo, como se haría en HTML con sus tags `</estilo>` – para volver a imprimir texto normal, simplemente inserta la línea `glk_set_style(style_Normal);`. Otras posibilidades son:

- `style_Emphasized`: para texto resaltado.
- `style_Preformatted`: para dibujar cosas a base de letras, como tablas o cualquier otra cosa que requiera una disposición de caracteres de ancho fijo (courier).
- `style_Header`: para dar título a secciones largas (por ejemplo, "ACTO PRIMERO")
- `style_Subheader`: para dar título a secciones más pequeñas, dentro de las secciones largas antes mencionadas (ej, "Escena I")
- `style_Alert`: para anuncios muy importantes (ej, "*** Has muerto ***")
- `style_Note`: para anuncios menos importantes (ej, "[Tu puntuación ha aumentado en 4 puntos.]")
- `style_BlockQuote`: para citas y similares
- `style_Input`: para el texto que el jugador escriba.

Pero exactamente ¿qué aspecto tienen todos estos estilos? Varía con cada intérprete. `style_Emphasized` puede ser negrita, o cursiva, o simplemente de un color más brillante. Sin embargo el programador puede hacer sugerencias con respecto a varios aspectos de un estilo dado (incluyendo dos estilos que se dejan para que el propio programador los defina: `style_User1` y `style_User2`). Algunos intérpretes harán caso de estas sugerencias; otros no. (Y algunos incluso les harán caso pero de forma incorrecta: la versión de Glulxe para DOS basada en Allegro, actuelmente pone los colores de fondo y letra al revés) De momento, no hay un `gestalt_StyleHints` o algo asi que pueda servir para saber si nuestras sugerencias son atendidas o ignoradas. Pero si crees que tendrás suerte, usa la rutina `glk_stylehint_set()`, que requiere los siguientes cuatro parámetros:

1. El tipo de ventana sobre la que debe aplicarse esta sugerencia de estilo. Las opciones son `wintype_TextBuffer`, `wintype_TextGrid` o `wintype_AllTypes` (pero no `wintype_Graphics`, puesto que no puedes imprimir texto en una ventana gráfica).
2. El estilo al que se aplica esta sugerencia de estilo.
3. El aspecto del estilo que se quiere cambiar (la lista de ellos se verá en breve).
4. El número o constante que indica qué se cambia en ese estilo y cómo.

El significado del cuarto argumento varía drásticamente dependiendo de lo que el tercer argumento indique. Así, "1" puede significar: "indentar el texto un poco", "negrita", "prácticamente negro" u otras cosas, según el contexto. Por tanto, el tercer y el cuarto argumento no deben ser tratados separadamente, sino en tándem. He aquí una tabla de las combinaciones que se admiten:

Aspecto (3er argumento)	Valor (4o argumento)
`stylehint_Indentation`	Aquí debe ir un número que significa "mueve el siguiente trozo de texto tantas unidades a la derecha" (un valor negativo lo mueve hacia la izquierda). ¿Cuánto es una unidad? Depende del intérprete.
`stylehint_ParaIndentation`	Esto es como el anterior, pero sólo afecta a la primera línea de cada párrafo.
`stylehint_Justification`	Aquí puede ir una de las siguientes constantes: `stylehint_just_LeftFlush` (justificación a la izquierda), `stylehint_just_RightFlush` (justificación a la derecha), `stylehint_just_LeftRight` (justificación completa) o `stylehint_just_Centered` (centrada).
`stylehint_Size`	Aquí hay que poner un número, pero no un número absoluto, como el tamaño en puntos, sino relativo. `0` significa "el tamaño por defecto", los valores positivos incrementan el tamaño una cierta cantidad, y los negativos lo decrementan. Los incrementos no son necesariamente de la misma magnitud: si `0` es 12 puntos, y `+1` es 14 puntos, `+2` puede ser 18 puntos.
`stylehint_Weight`	`0`=normal, `1`=negrita, `-1` = ligera.
`stylehint_Oblique`	`0`=no cursiva, `1`=cursiva.
`stylehint_Proportional`	`0`=usar fuente de ancho fijo (tipo courier), `1`=fuente de ancho proporcional
`stylehint_TextColor`	Este debe ser un número de 32 bits que representa el color usado. Es mucho más fácil, con diferencia, si escribimos el número en hexadecimal `nota3.html`(nota), lo que se hace en la forma siguiente. Primero, escribe el signo dolar (que indica a Inform que lo que sigue es un número hexadecimal). Después escribe un número hexadecimal de dos cifras, de `00` a `FF`, que representa la cantidad de rojo presente en el color. Después viene otro número de dos dígitos hexadecimales que representa la cantidad de verde, y finalmente otro número de dos dígitos hexadecimales que representa la cantidad de azul. Así, `$000000` sería el negro, y `$FFFFFF` sería el blanco, `$FF0000` sería un rojo intenso, `$FFC000` sería un bonito dorado, `$C0C0FF` un azul bebé, etcétera.
`stylehint_BackColor`	Aquí va un número de 32 bits, exactamente como el anterior, sólo que en esta ocasión estás seleccionando el color del "papel" detrás del texto. Sin embargo, éste no es el color de fondo de la ventana, y los resultados serán extremadamente horribles en algunos intérpretes (`screen.gif`aqui hay una captura de pantalla). Actualmente no hay sugerencia de estilo para cambiar el color de fondo la ventana; con suerte, se le añadirá sin tardar mucho.
`stylehint_Reverse`	`0`=imprimir normalmente, `1`=imprimir el texto usando el color del fondo, y viceversa.

Así que, juntándolo todo, pongamos que quieres definir el estilo "User1" como texto rojo sobre un fondo negro. Esto se lograría con el código siguiente:

```
glk_stylehint_set(wintype_TextBuffer, style_User1,
                  stylehint_TextColor, $FF0000);
glk_stylehint_set(wintype_TextBuffer, style_User1,
                  stylehint_BackColor, $000000);
```

Sin embargo, las sugerencias de estilo sólo afectan a las ventanas que se crean a partir de entonces. Esto significa que si quieres usarlas, debes hacerlo antes de crear la ventana en la que quieres que se usen. Y puesto que la ventana `gg_mainwin` es creada por la libreria, si quieres que las sugerencias de estilo tengan efecto sobre la ventana principal del juego, ponerlas en `Inicializar` sería demasiado tarde – necesitas programar una rutina llamada `InitGlkWindow()`. Esta rutina es llamada desde la librería varias veces, y le pasa diferentes valores cada vez. En este caso, tienes que ejecutar tus sugerencias de estilo cuando el valor recibido sea igual al valor "roca" de `gg_mainwin`, así:

```
[ InitGlkWindow winrock;
  switch (winrock) {
     GG_MAINWIN_ROCK:
         glk_stylehint_set(wintype_TextBuffer, style_User1,
                           stylehint_TextColor, $FF0000);
         glk_stylehint_set(wintype_TextBuffer, style_User1,
```

```
                            stylehint_BackColor, $000000);
    }
  rfalse; ! si te olvidas esta linea, el juego no funcionará bien
];
```

Una última observación antes de proseguir. Podría parecerte que un código como este:

```
print "Te he dicho que la violencia ";
glk_set_style(style_Emphasized);
print "no";
glk_set_style(style_Normal);
print " es la solución en este caso.";
```

es poco manejable. Lo es. Por otro lado no es mucho peor que:

```
print "Te he dicho que la violencia ";
style bold;
print "no";
style roman;
print " es la solución en este caso.";
```

Por esto muchos programadores usan pequeñas rutinas para hacer todo esto más amigable. No es necesario escribir `style_Emphasized` un ciento de veces; algo como esto, lo haría perfectamente:

```
[ b texto;
  glk_set_style(style_Emphasized);
  print (string) texto;
  glk_set_style(style_Normal);
]
```

Y una vez que has creado esa rutina, puedes escribir el ejemplo de antes de esta otra forma:

```
print "Te he dicho que la violencia ", (b) "no",
      " es la solución en este caso.";
```

3.5. Líneas de estado

Uno de los usos más comunes del ensamblador-Z dentro de un juego Inform es el de crear líneas de estado personalizadas. Éstas no funcionarán con Inform Glulx, puesto que no reconoce el ensamblador-Z. En su lugar, tendrás que usar las llamadas a Glk equivalentes – que no son opcodes, sino más bien rutinas como cualquier otra.

La línea de estado es una tradición tan arraigada dentro de las ACs (sobre todo inglesas) que cuesta realmente un poco deshacerse de ella. Un juego Inform estándar reservará una parte de la pantalla para mostrar el nombre de la localidad en la que se halla el jugador, la puntuación actual, y el número de movimientos realizados. Para cambiar esto, el programador debe insertar la línea "`Replace DibujarLineaEstado;`" al comienzo del programa (también vale si lo pones justo después de las constantes, en realidad, es suficiente que aparezca antes del primer `Include`). Después, el programador debe proporcionar una rutina `DibujarLineaEstado` alternativa. Pero aquí es donde las cosas divergen un poco.

En Inform estándar, el comando para crear la ventana de la línea de estado (`@split_line`) se hacía dentro de la rutina `DibujarLineaEstado` de la librería. Si quieres reemplazar ésta, tienes que poner el comando `@split_window` en la tuya. Si quieres eliminar la línea de estado, todo lo que necesitarás hacer es reemplazar `DibujarLineaEstado` por una rutina vacía, como esta:

```
[ DibujarLineaEstado; ];
```

Esto ya no es suficiente en Glulx Inform. En la sección Glulx de InformATE biplataforma, la ventana para la línea de estado es creada en la rutina `GGInitialise`, no en `DibujarLineaEstado`. Esto significa que si te limitas a usar una `DibujarLineaEstado` vacía, la línea de estado saldrá vacía, pero aún estará ahi, burlándose de tí. De modo que lo que tienes que hacer es interceptar el comando que crea la ventana de estado, mediante una programación de la rutina `InitGlkWindow` como la siguiente:

```
[ InitGlkWindow winrock;
   switch (winrock) {
      GG_STATUSWIN_ROCK: rtrue;
   }
   rfalse; ! si olvidas esta línea el programa no funcionará
];
```

Si quieres mantener la línea de estado, pero configurar a tu gusto la información que se muestra en ella, he aqui algunos consejos. Lo primero, si quieres reservar más de una línea para la ventana de estado, debes hacerlo en `InitGlkWindow`, como aqui:

```
[ InitGlkWindow winrock;
   switch (winrock) {
      GG_STATUSWIN_ROCK:
         gg_statuswin_size = 2; ! el número de líneas que desees
   }
   rfalse; ! si olvidas esta línea el programa no funcionará
];
```

Después, al escribir tu propia `DibujarLineaEstado`, debes comenzar con el código siguiente:

```
   ! Si no hay ventana de estado, no hay que redibujarla
   if (gg_statuswin == 0)
      return;    ! Si no hay localización, tampoco
   if (localizacion == nothing || parent(jugador) == nothing)
      return;
```

La primera línea después de eso, debe ser `glk_set_window(gg_statuswin);` para cambiar a la ventana en la cual estás a punto de imprimir información de estado, y la última línea debe ser `glk_set_window(gg_mainwin);` para asegurarte de que el programa continuará imprimiendo el texto del juego en el lugar correcto. En el medio, puedes imprimir cualquier cosa, si bien el procedimiento para ello es ligeramente diferente al usado en una ventana de tipo buffer de texto.

Recuerda que una ventana tipo rejilla de texto es una especie de tablero de Scrabble: aunque no se ven las líneas separando las casillas, la ventana consiste en una rejilla de casillas, cada una de las cuales puede contener un carácter – letra, número o puntuación. Cada casilla tiene un par de coordenadas asociadas, una coordenada X y una coordenada Y. La coordenada X es cuántas casillas hay antes de ella contando desde el borde izquierdo; la coordenada X de la primera casilla (junto al borde izquierdo) es 0. La coordenada Y es cuántas casillas hay por encima de la casilla actual, hasta el borde superior; la coordenada Y de las casillas superiores es 0. (Esto es diferente al Inform estándar, en el que la esquina superior izquierda no era (0,0), sino (1,1)). Para seleccionar el lugar donde quieres comenzar a escribir, usa la orden `glk_window_move_cursor()`, que necesita tres argumentos: el nombre de la ventana donde imprimirás, la coordenada X de la casilla en la que quieres que comience a imprimir, y la coordenada Y de esa casilla. Cuando imprimas, el cursor irá avanzando automáticamente hacia la derecha para cada letra del texto impreso – no necesitas moverlo para imprimir la siguiente letra de una palabra. Así que el código siguiente:

```
   glk_window_move_cursor(gg_statuswin, 3, 0);
   print "Hora: ";
```

colocará la letra “H” en la posición (3,0), la “o” en (4,0), “r” en (5,0), “a” en (6,0), dos puntos en (7,0) y un espacio en (8,0), dejando el cursor en (9,0). Presumiblemente después iría el código para imprimir la hora.

Una cosa a tener en cuenta es que no todo el mundo que juegue tu juego tendrá la misma cantidad de espacio de pantalla reservado para Glulxe: la línea de estado puede tener espacio para 120 letras en el ordenador de una persona, y 40 en el de otra. O alguien puede cambiar de tamaño la ventana Glulxe, y el ancho cambiar de 120 a 40 en mitad del juego. Puedes programar la rutina `DibujarLineaEstado` para que tenga esto en cuenta (como de hecho se hace en la `DibujarLineaEstado` que viene con InformATE). Simplemente añade variables llamadas `ancho` y `alto` en la declaración de `DibujarLineaEstado`, e inserta el siguiente fragmento de código en tu rutina:

```
glk_window_get_size(gg_statuswin, gg_arguments, gg_arguments+4);
ancho = gg_arguments-->0;
alto = gg_arguments-->1;
```

Y después puedes hacer los ajustes que estimes oportuno. Digamos que estás haciendo un juego llamado Tritura, Machaca y Pisotea y quieres poner un indicador de hambre en la línea de estado, comenzando en la columna 40, que debe mostrar una de las palabras “HARTO”, “SATISFECHO”, “HAMBRIENTO”, “FAMELICO”. Esto podría llevarte hasta la columna 50. Puede que algunos jugadores estén jugando con menos de 50 columnas, y en ese caso prefieres que no se muestre este indicador (porque lo que desde luego no quieres es que aparezca incompleto, a mitad de una palabra). Usarías un código como el siguiente:

```
   if (ancho > 50) { ! Nos aseguramos de tener sitio
      glk_window_move_cursor(gg_statuswin, 40, 0);
      switch (gojira.hambre) {
```

```
            0: print "HARTO";
            1: print "SATISFECHO";
            2: print "HAMBRIENTO";
            3: print "FAMELICO";
        }
    }
```

Observa que las cuatro palabras anteriores tienen diferentes longitudes. Si intentas sobreescribir "SATISFECHO" con "HARTO", obtendrás "HARTOFECHO", a menos que, o bien rellenes con espacios a la derecha cada palabra, o pongas un `glk_window_clear(gg_statuswin);` justo después de tu llamada a `glk_set_window(gg_statuswin);`. La librería InformATE usa este segundo método.

Se pueden hacer acomodos más inteligentes que simplemente el no mostrar los elementos que no caben – podíamos haber puesto algún código elaborado para abreviar las cosas a medida que la línea de estado es más pequeña, por ejemplo. Lo bueno es que, ya que `DibujarLineaEstado` es llamada en cada turno, no necesitas andar tocando `HandleGlkEvent` para manejar los cambios de tamaño, como es necesario hacer con los gráficos. Más sobre esto en las secciones siguientes.

Capítulo 4

Gráficos

4.1. Mostrando gráficos PNG y JPEG

Si quieres que tu juego muestre un gráfico, lo primero tienes que preparar algunas cosas. El nombre del fichero que contiene la imágen debe ester listado en el fichero de recursos Blorb asociado con tu programa (ver `blorb.html`Blorb) y haberle dado un nombre para referirte a él desde el juego. Glk soporta dos formatos de imágen: PNG y JPEG. JPEG es un formato con pérdida, utiliza un algoritmo de compresión que reduce considerablemente el tamaño del fichero, a cambio de perder un poco de fidelidad a la imagen original (a menudo la pérdida de fidelidad no es detectable por el ojo). PNG es un formato sin pérdida que tiene soporte para pixels transparentes, en una forma muy similar al GIF, solo que sin los potenciales problemas legales que podrían derivarse de usar GIF (pues el algoritmo de compresión que usa GIF está patentado). JPEG se considera mejor para fotos, PNG para cosas como arte dibujado "a mano" (con un programa de dibujo), imágenes de texto, y similares. Si tu imagen no está en ninguno de esos formatos, tendrás que usar un programa gráfico para convertirlo antes de poder usarlo en tu juego Inform Glulx.

Lo siguiente, tienes que decidir dónde mostrarás la imagen. El trozo pantalla que ocupa el intérprete Glulx puede ser dividido en ventanas (ver `VentanasGlk`ventanas.html, y tienes que decidir cuál de ellas será el destino de esta imagen concreta. Lo que hay que hacer a partir de aquí, depende del tipo de ventana al que pertenezca la ventana destino.

- Si quieres salpicar de imágenes el texto de tu juego, entonces querrás poner gráficos en una ventana de tipo buffer de texto.
- Si quieres mostrar la imagen en un área específicamente diseñada para ello de la pantalla, entonces querrás poner gráficos en una ventana gráfica.

4.1.1. Gráficos en una ventana de buffer de texto

Bien, has decidido que quieres meter una imagen directamente en medio del texto de tu aventura conversacional. Esto es, realmente, bastante sencillo: simplemente llama a `glk_image_draw()` con cuatro argumentos:

1. El nombre de la ventana a la que vas a enviar la imagen. A menos que estés haciendo cosas realmente inusuales con tu disposición de ventanas, el nombre de esta ventana será `gg_mainwin`
2. El nombre de la imagen a mostrar, tal como la has nombrado en tu fichero de recursos Blorb. Si la imagen es de un mono, por ejemplo, podrías llamarla `Img_Mono`
3. La forma en la que quieres que el texto fluya alrededor de la imagen. Aquí tienes cinco posibilidades. Las tres primeras son para el caso de que quieras que tu imagen sea colocada como si simplemente fuera otra palabra más en la línea de texto que aparecerá. Quizás la mejor forma de mostrar cada opción sea con algunos diagramas:

   - `imagealign_InlineUp`

   ```
   El veloz murciélago hindú comía feliz cardillo
           ################
           #              #
           #  (Img_mono)  #
           #              #
   y kiwi. ################ La cigüeña tocaba el
   saxofón detrás del palenque de paja.
   ```

- imagealign_InlineDown

```
El veloz murciélago hindú comía feliz cardillo
y kiwi. ################ La cigüeña tocaba el
        #              #
        #  (Img_mono)  #
        #              #
        ################
saxofón detrás del palenque de paja.
```

- imagealign_InlineCenter

```
El veloz murciélago hindú comía feliz cardillo
        ################
        #              #
y kiwi. #  (Img_mono)  # La cigüeña tocaba el
        #              #
        ################
saxofón detrás del palenque de paja.
```

¡

Naturalmente, si la imágen es la única cosa que va a aparecer en ese párrafo línea puedes elegir cualquiera de los tres, pues todos saldrán igual.

Las otras dos opciones sólo funcionarán si la imagen es la primera cosa que sale en la línea (es decir, si va después de un carácter ^ en una cadena, o después de una llamada a `new_line;`) Cualquier texto que vaya después de la imagen, será plegado alrededor de la imagen en una de estas dos formas:

- imagealign_MarginLeft

```
################ El veloz murciélago
#              # hindú comía feliz
#  (Img_mono)  # cardillo y kiwi. La
#              # cigüeña tocaba el
################ saxofón detrás del
palenque de paja. El veloz murciélago
hindú comíafeliz cardillo y kiwi. La
```

- imagealign_MarginRight

```
El veloz murciélago  ################
hindú comía feliz    #              #
cardillo y kiwi. La  #  (Img_mono)  #
cigüeña tocaba el    #              #
saxofón detrás del   ################
palenque de paja. El veloz murciélago
hindú comía feliz cardillo y kiwi. La
```

4. El cuarto argumento vale 0. Este argumento sólo tiene algún significado si la imagen va a ser dibujada en una ventana gráfica; sin embargo, no podemos omitirlo, debido a que, a diferencia de las funciones Inform normales, las funciones de `Infglk` no pueden recibir menos parámetros de los que están esperando.

Así que supongamos que queremos que esta imagen del mono aparezca cuando el jugador decide llevar al protagonista al zoo y `> EXAMINAR MONO`. Deberíamos programar el mono como sigue:

```
Object  mono "el mono Bubu"
 with   nombre 'mono' 'bubu',
        descripcion [;
            print "El mono tiene esta pinta:^";
            glk_image_draw(gg_mainwin, Img_mono,
                           imagealign_InlineUp, 0);
            print "^"; rtrue;
        ],
 has    animado propio;
```

Por supuesto, siempre cabe la posibilidad de que el jugador esté usando un intérprete que no soporte gráficos, en cuyo caso la imagen no podrá cargarse y el jugador sólo verá una línea en blanco después de la frase "esta pinta:". Para saber cómo habérselas con esta posibilidad, ver la sección sobre test de capacidades.

4.1.2. Imágenes en una ventana gráfica

Al igual que en las ventanas de buffer de texto, la función para colocar imágenes en una ventana gráfica es `glk_image_draw()`, pero en esta ocasión sus cuatro argumentos son los siguientes:

1. El nombre de la ventana en la que se va a colocar el gráfico (`gg_mapawin`, por ejemplo).
2. La imagen que se va a colocar (pero mira la explicación que va después de la lista de argumentos).
3. y 4. Las coordenadas en las que quieres que se sitúe la esquina superior izquierda de la imagen. El tercer argumento indica cuántos pixels hay desde el borde izquierdo de la ventana, y el cuarto cuántos hay desde el borde superior de la ventana, así que si quieres que la imagen salga pegada a estos bordes, pon 0 para ambos. (¿Por qué querrías usar coordenadas distintas de 0,0? Quizás quieras antes poner una imagen de un borde decorativo, y cargar pequeñas imágenes en ese borde, quizas en 10,l0.) No te preocupes si la imagen sobrepasa el margen derecho o inferior de la ventana gráfica – cualquier exceso será recortado.

La parte complicada es el segundo argumento. Cuando pones imágenes en una ventana de tipo buffer de texto, esta imagen pasa a ser simplemente un elemento más del flujo que el programa envía a la ventana, de modo que cuando una operación `undo` o una restauración de un juego salvado cambia la localización del protagonista dentro del juego, la librería manejará estas imágenes automáticamente. No ocurre lo mismo cuando estás poniendo las imágenes en otras ventanas. Veamos un ejemplo. Supongamos que tienes un juego con una ventana gráfica de tamaño fijo, donde muestras un gráfico de la localidad en la que se halla el jugador. Y digamos que has decidido hacer esto insertando directamente la imagen que quieres mostrar cuando el jugador se mueve a esa habitación. Así, si la cocina está al sur del comedor, podrías tener un trozo de código en el objeto comedor con un aspecto como este:

```
al_s [;
    if (gg_picwin) { ! comprobación de que existe la ventana
       glk_image_draw(gg_picwin, Img_Cocina, 0, 0);
    }
    JugadorA(Kitchen);
],
```

Esto, realmente, causará que la imagen de la cocina aparezca en la ventana correcta cuando el jugador entra en la cocina. Pero ¿y si el jugador después pone `undo` o carga una partida salvada en la cual estaba en el comedor? En la ventana principal (de texto), se verá retornar al comedor, ¡pero la ventana gráfica aún estará mostrando una imagen de la cocina! La librería no actualiza las ventanas gráficas automáticamente, tienes que hacerlo tú.

Manejo de ventanas gráficas

La librería biplataforma proporciona unos cuantos "puntos de entrada" para hacer un poco más sencillo el manejo de las ventanas. Un punto de entrada es una especie de rutina opcional, que los programadores pueden escribir en su código fuente. Los programadores de librerías, de vez en cuando encontrarán un punto en el código de la librería donde sospechan que los programadores de juegos querrían añadir código personalizado, o quizás no. Entonces el programador de la librería hace una rutina "tonta" (vacía) que no hace nada, y la llama desde ese punto. Entonces, el programador de juegos puede escribir una rutina con el mismo nombre, y cuando la librería llega al punto en cuestión, ejecutará el código de la rutina que ha proporcionado el programador del juego, que hace algo, en lugar de la que venía en la librería que no hacía nada. O el programador de juegos puede no escribir esa rutina, y entonces la librería ejecutará la rutina "tonta" que no hace nada y proseguirá.

De modo que, si en tu juego no vas a usar ventanas especiales (como ventanas gráficas), o canales de sonido, o referencias a ficheros externos, no necesitarás preocuparte por los puntos de entrada que se van a explicar ahora. La librería se ocupará de `gg_mainwin` y `gg_statuswin` por ti, así como de cualquier fichero de "partida guardada" que puedas generar. Pero si creas tus propios objetos glk, como una ventana gráfica, es tu responsabilidad atenderla. He aquí cómo.

Lo primero, tienes que crear una rutina para el punto de entrada que es llamado tras cada reinicio, restauración de partida guardada y cada `undo`. Esta es `IdentifyGlkObject()`. La razón por la que debes tener una rutina `IdentifyGlkObject()` si creas tu propia ventana gráfica es porque, después de un reinicio, una restauración de partida o un `undo`, las variables globales que manejan esas ventanas pueden tener valores incorrectos, y las cosas pueden ponerse feas si no les devuelves los valores correctos.

`IdentifyGlkObject()` en realidad es llamada tres veces, y recibe un parámetro llamado `fase` que indica qué llamada es la que se ha producido (toma los valores 0, 1 y 2). En cada una de esas llamadas, la misión de `IdentifyGlkObject()` es diferente. En la fase 0, tu misión es poner a cero todas las variables que uses para manejar objetos Glk. En la fase 1, tienes que restaurar los valores correctos a todas las variables que uses para manejar ventanas, flujos de datos y referencias a ficheros. Finalmente, en la fase 2 debes restaurar los valores correctos de los restantes objetos Glk que hayas creado (canales de sonido, por ejemplo). En la fase 2 es también

donde debes actualizar tus ventanas para que muestren las imágenes correctas, y los canales de sonido para que toquen la música adecuada, etc.

Así que, continuando con el ejemplo de la sección pevia, digamos que tenemos una ventana gráfica, `gg_picwin`, en la que mostramos la imagen de la localidad donde está el protagonista. Pero a diferencia de lo que hicimos en la sección previa, no dibujaremos las imágenes directamente en esa ventana, con llamadas como `glk_image_draw(gg_picwin, Img_Cocina, 0, 0)`. En vez de eso, tendremos una variable global llamada `imagen_actual`, y una rutina como la siguiente:

```
[ RedibujarVentanaGrafica ;
   glk_image_draw(gg_picwin, imagen_actual, 0, 0);
];
```

Ahora, la rutina para mover al jugador desde el comedor a la cocina hacia el sur, tendría este aspecto:

```
  al_s [;
     imagen_actual=Img_Cocina;
     RedibujarVentanaGrafica();
     JugadorA(cocina);
  ],
```

Y por último, tendremos nuestro punto de entrada `IdentifyGlkObject()`:

```
[ IdentifyGlkObject fase tipo ref rock;
   if (fase == 0) { ! Poner cero en todos nuestros objetos glk
      gg_picwin = 0;
      return;
   }
   if (fase == 1) { ! Reiniciar correctamente las variables glk
      switch (tipo) {
         0: ! es una ventana
            switch (rock) {
               GG_PICWIN_ROCK: gg_picwin = ref;
            }
         1: ! es un flujo
            ! pero no hay flujos en este ejemplo
         2: ! es una referencia a fichero
            ! pero no hay ficheros en este ejemplo
      }
      return;
   }
   if (fase == 2) { ! Actualizar nuestras ventanas
      RedibujarVentanaGrafica();
   }
];
```

¿Qué es todo eso? Primero, cogemos la variable global en la cual guardábamos la ventana gráfica que habíamos creado antes (pero que ahora solo contiene basura, debido al reinicio) y la ponemos a 0 – esto es la fase 0. Después, la librería encuentra este objeto Glk creado y no sabe lo que es, así que llama de nuevo al punto de entrada para que determine lo que era. Como parámetros le pasa el identificador de este objeto (en `ref`) y el "valor roca" del objeto en `rock` (el cual no se pierde durante el reset). En nuestra rutina decimos "Si el valor roca de este objeto es igual al valor roca que habíamos puesto a nuestra ventana gráfica, entonces ¡tiene que ser nuestra ventana gráfica!" Por tanto, actualizamos el valor de nuestra variable con el valor que recibimos en `ref`. Esto fue la fase 1. Finalmente en la fase 2 hacemos un redibujo: ahora que ya sabemos a dónde enviar la imagen que queremos mostrar, podemos mostrarla.

La ventana gráfica ahora se comportará de forma correcta ante restauraciones de partidas grabadas y "undos" y similares: si el jugador mueve al protagonista a la cocina, y después pone UNDO, no sólo el protagonista volverá al comedor, sino que la ventana gráfica reemplazará además la imagen de la cocina con la imagen del comedor.

¿Y qué hay de los eventos externos que pueden afectar al juego? Por ejemplo el jugador puede cambiar de tamaño la ventana de Glulxe, o cambiar la resolución del monitor – ¿cómo podemos manejar estas cosas? Respuesta: la librería tiene un bucle en el que comprueba constantemente estas cosas, y este bucle también contiene un punto de entrada llamado `HandleGlkEvent()`. `HandleGlkEvent()` recibe dos argumentos: "`ev`" es un array que contiene información sobre lo que acaba de ocurrir (¿un cambio de tamaño? ¿una pulsación del ratón? ¿el final de un efecto de sonido?), y "`contexto`" que puede valer 0 si el evento ocurrió mientras la librería esperaba una entrada (como cuando espera por un comando normal, o en un prompt como el de la

rutina `SiONo()` – en cualquier momento en que el juego está esperando a que el jugador pulse la tecla Intro para interpretar la entrada) o 1 si el evento ocurrió durante una espera de carácter (como en los menús y similares, en los que el juego espera por cualquier tecla). Para el ejemplo que estamos tratando, todo lo que necesitamos es esta breve rutina:

```
[ HandleGlkEvent ev contexto;
   contexto = 0; ! Esta linea solo está para evitar un warning
   switch (ev-->0) {
      evtype_Redraw, evtype_Arrange:
         RedibujarVentanaGrafica();
   }
];
```

Este código, esencialmente se resume en "Si ocurre algo que implique que hay que redibujar la ventana gráfica, pues adelante, redibujala usando las instrucciones que hemos escrito antes."

Y esto debería ser todo lo que necesitas para hacer tus ventanas gráficas lo bastante robustas como para manjear cualquier contingencia que se les venga encima.

4.1.3. Solución de problemas con los gráficos

Pregunta: ¡Estoy bastante seguro de haberlo hecho todo bien, pero mi juego se niega a mostrar este JPEG! No se cuelga, y pasa el test de `gestalt_Graphics`, pero no sale el gráfico. ¿Qué pasa?

Respuesta: El problema puede estar en el propio fichero JPEG. Realmente, JPEG es un nombre que se da una variedad de formatos de imagen diferentes, y la implementación de Glk que estás usando puede estar equipada sólo para decodificar agunos de ellos. Volver a guardar el JPEG con otros settings, o con otro programa de tratamiento de gráficos, puede arreglar la cosa. Si aún no funciona, puede que tengas que usar PNG en su lugar.

4.2. Dibujando tus propias imágenes

Puedes hacer más que colocar imágenes PNG y JPEG en tus ventanas gráficas; puedes dibujar las tuyas propias. El principal comando para hacer esto es `glk_window_fill_rect()`, que necesita seis argumentos:

1. El nombre de la ventana en la que quieres dibujar
2. El color del que quieres que sea el rectángulo. Éste está codificado como un número hexadecimal en la forma siguiente. Primero, escribe un signo dólar (que indica que lo que va después es un número hexadecimal). Después, ecribe un número hexadecimal de dos dígitos, desde `00` a `FF`, que representa la cantidad de rojo que forma parte del color. Después viene un número exadecimal de dos dígitos que representa la cantidad de verde, y después un número hexadecimal de dos dígitos que representa la cantidad de azul. Así, `$000000` sería el negro, `$FFFFFF` sería blanco, `$FF0000` sería un rojo intenso, `$FFC000` sería un bonito dorado, `$C0C0FF` sería un azul bebé, etc...
3. La coordenada X de la esquina superior izquierda del rectángulo.
4. La coordenada Y de la esquina superior izquierda del rectángulo.
5. El ancho del rectángulo, si quieres poner un solo pixel o una línea vertical, éste debe ser 1.
6. El alto del rectángulo. Si quieres dibujar un solo pixel o una línea horizontal, éste debe ser 1.

Y esto prácticamente es todo. Otro truco que puedes hacer es llenar toda una ventana con un color sólido, usando `glk_window_set_background_color()` (que requiere dos argumentos: la ventana en cuestión y el color que quieres darle al fondo, codificado como se ha explicado antes) y a continuación hacer un `glk_window_clear()` (que requiere un argumento, la ventana a borrar). Pero si quieres dibujar cosas más complejas que puntos, líneas y rectángulos, tendrás que, al menos de momento, construirlas tu mismo a base de puntos, líneas y rectángulos. En la mayoría de los casos será mucha mejor decisión usar un programa gráfico para crear una imagen PNG o JPEG.

Capítulo 5

Música y sonido

El proceso de crear un canal de sonido, lanzar efectos de sonido y música en él, y usar puntos de entrada para manejarlos, debería resultar muy familiar a quienes han leido ya la sección sobre implementación de ventanas gráficas. A quienes se hayan saltado ese capítulo para venir directamente aqui, se les recomienda muy fervientemente que lean antes la sección sobre gráficos, especialmente la parte sobre manejo de ventanas gráficas.

Otra lectura necesaria es la sección sobre Blorb, ya que usarás Blorb para hacer que los ficheros de sonido estén disponibles para tu juego, en la misma forma que lo hiciste para los ficheros de gráficos. Los formatos de sonido que actualmente soporta Glk, y por tanto Inform Glulx, son AIFF y MOD. El primero es un formato de sonido sin compresión, una especie de equivalente multiplataforma del más conocido `WAV` de Microsoft, que produce ficheros que, con los parámetros adecuados, son de calidad suficiente como para grabar con ellos un CD de audio. (El inconveniente es que comparados con un formato con pérdida de calidad como el MP3, son bastante más enormes). MOD es bastante diferente. Los ficheros MOD se crean con programas llamados *trackers*, en los cuales tú cargas algún tipo de muestra de sonido breve (un golpe de tambor, un acorde de guitarra, un ladrido de un perro, lo que quieras) y despues compones una especie de partitura, colocando notas en una rejilla con el tono adecuado, seleccionando un *tempo*, aplicando efectos especiales a las muestras, etc. Puesto que el fichero no almacena el sonido final de la música, sino sólo las piezas de las que se forma junto con instrucciones de cómo tocarlas, los ficheros MOD son bastante pequeños. (Además es muy divertidos de coleccionar y retocar. Si usas una máquina Windows, te recomiendo encarecidamente el Tracker Modplug).

Así que pongamos que estás programando un ascensor, y que decides que cuando el protagonista entre en él, debe sonar música ambiente de ascensor. El proceso sería como sigue. Primero, necesitas crear un canal de sonido, y darle un número-roca de 410 o superior (en este caso, elegiremos 410). Escribimos las líneas siguientes en el programa:

```
Constant GG_CANALMUSICA_ROCK 410;
Global   gg_canalmusica = 0;
```

Y ya que estamos, añadamos una variable global para almacenar un identificador de qué música está sonando en cada momento del juego:

```
Global musica_actual = 0;
```

Y naturalmente, en algún momento necesitaremos abrir el fichero de recursos Blorb y darle al fichero de sonido un nombre que podamos usar en el código fuente. Digamos que te apetece mofarte de las leyes del mercado y llamar al fichero `Muzak`. Cómo indicar esto puede variar; si estás usando `iblorb`, la línea será algo como:

```
SOUND Muzak musicascensor.mod
```

Una vez liquidada la preparación anterior, podemos dedicarnos al negocio de abrir el canal de sonido y enviarle música. Abrirlo es fácil: basta añadir lo siguiente en `Inicializar`:

```
  if (gg_canalmusica == 0)
    gg_canalmusica = glk_schannel_create(GG_CANALMUSICA_ROCK);
  }
```

(Personaliza lo anterior en tus propios programas en la forma obvia.) Con esto, el canal está ahora abierto. Lo siguiente: tocar sonidos en él. Esta vez el comando clave es `glk_schannel_play_ext()` que necesita cuatro argumentos:

1. El nombre del canal de sonido. Observa que puedes tener múltiples canales de sonido – digamos, uno para efectos especiales y otro para música – pero en cualquier plataforma llegarás eventualmente a un límite. Tocar múltiples MODs a la vez es una perspectiva especialmente dudosa.

2. El nombre del sonido a tocar (pero lee más abajo)

3. El número de veces que el sonido debe ser repetido. Si quieres que se repita por siempre, hasta que le digas explícitamente que pare (o hasta que el juego acabe), pon -1.

4. Este debe ser 0, a menos que quieras recibir eventos del tipo `evtype_SounNotify` en tu rutina `HandleGlkEvent()` cuando el sonido finalice su ejecución. Por ejemplo, podrías querer programar una mansión encantada, en la que suenen al azar espectrales ruidos de monstruos. El código podría ser así:

```
[ HandleGlkEvent ev context sonido_nuevo;
   context = 0; ! evitar el warning de variable no usada
   switch (ev-->0) {
      evtype_SoundNotify:
         sonido_nuevo = random(Aullido, Chirrido, Grunyido);
         glk_schannel_play_ext(gg_canalmusica, sonido_nuevo, 1, 1);
   }
];
```

Pero nos estamos saliendo un poco de madre. Volvamos con lo nuestro.

Al igual que ocurría con las ventanas gráficas, no queremos simplemente lanzar directamente una música a un canal de sonido, especialmente si va a repetirse en bucle. Si lo hiciéramos, el jugador podría hacer que el protagonista entrara en el ascensor, después restaurar un juego salvado en un punto en que el protagonista estaba en medio de un campo de batalla – ¡y la música del ascensor aún seguiría sonando! Para evitar estas potenciales pifias, usaremos la variable que hemos creado antes, y escribiremos una rutina como esta:

```
[ ReiniciarCanalMusica;
   if (gg_canalmusica) {
      if (musica_actual == 0) glk_schannel_stop(gg_canalmusica);
      else glk_schannel_play_ext(gg_canalmusica, musica_actual, -1, 0);
   }
];
```

El código dentro de las llaves hace sonar la música que debería estar sonando, indicada por la variable `musica_actual`, o detiene la musica completamente si esa variable está a 0 [el valor de esa variable sí se recupera al cargar una partida]. Por supuesto, si el canal de sonido no está abierto, no intentará tocar nada. Ahora podemos crear un objeto ascensor, con una rutina `antes` como esta:

```
antes [;
   Meterse: musica_actual = Muzak;
          ReiniciarCanalDeMusica();
          print "Entras en el ascensor. La selección musical de
                 hoy: arreglos de Korn para xilófono y flauta de
                 pan.^";
          JugadorA(Dentro_Ascensor);
],
```

Y finalmente, una rutina `IdentifyGlkObject()` para asegurarse de que todo es restaurado correctamente tras un REINICIAR o UNDO. Si ya tienes una, mézclala con esta otra:

```
[ IdentifyGlkObject fase tipo ref rock res id;
   if (fase == 0) { ! Poner a cero nuestras variables de glk
      gg_canalmusica = 0;
      return;
   }
   if (fase == 1) { ! En la fase 1 no se hace nada con los canales
                    ! de sonido.
      return;
   }
   if (fase == 2) {
      ! Iterar sobre todos los canales de sonido existentes,
      ! e identificar el nuestro.
      id = glk_schannel_iterate(0, gg_arguments);
      while (id) {
         switch (gg_arguments-->0) {
```

```
            GG_CANALMUSICA_ROCK: gg_canalmusica = id;
          }
          id = glk_schannel_iterate(id, gg_arguments);
        }
        ! Ahora, que ya tenemos inicializada la variable
        ! del canal, necesitamos actualizar la musica que
        ! suena, o desconectarla
        ReiniciarCanalDeMusica();
    }
];
```

El código de la fase 2 hace práctiamente lo mismo que lo que hacía el código de la fase 1 para las ventanas gráficas; la diferencia es que un canal de sonido no es una ventana, ni un *stream* (flujo) ni una referencia a fichero, de modo que la fase 1 no ha de tratar con ellos.

¡Y esto debería ser todo! Por supuesto, si intentas esto en casa, puede que te lleves una desilusión ya que, en el momento en que se escribe esto, sólo un par de plataformas tienen soporte para MOD (Windows y DOS, y encima el caso DOS tiene bugs). Pero otras plataformas se subirán al tren enseguida.

Capítulo 6

Otros trucos

6.1. Entrada por ratón

Las ventanas de tipo texto y las de tipo rejilla tienen la interesante capacidad de reconocer cuándo se pulsa un ratón sobre ellas, y de retornar las coordenadas de la letra o pixel que se ha seleccionado. Para sacar partido de esta capacidad, primero debes alertar al programa de que tiene que estar alerta para detectar las pulsaciones de ratón en la ventana o ventanas que te interesen. Por ejemplo, digamos que tienes una ventana gráfica, `gg_ventana_brujula`, en la que has puesto una imagen de una rosa de los vientos de ocho puntas. Quieres que el jugador pueda clicar sobre cualquiera de las puntas y que eso se convierta en un comando para ir en la dirección apropiada. Lo primero, necesitas añadir la siguiente línea a la rutina `Inicializar()`:

```
glk_request_mouse_event(gg_ventana_brujula);
```

El juego estará alerta para detectar una pulsación, y sólo una, del ratón en la ventana que tiene la rosa de los vientos. Si no ocurre ninguna, no pasa nada; el programa aceptará igualmente la entrada habitual por el teclado. Pero si ocurre una, se disparará la rutina `HandleGlkEvent()`.

`HandleGlkEvent()` ya ha aparecido en secciones previas (mira "Imágenes en una ventana gráfica", por ejemplo, o en "Música y sonido") pero este es un caso especialmente claro de cómo funciona la rutina. Como siempre, la rutina `HandleGlkEvent()` recibe los parámetros `ev` y `contexto`; esta vez `ev`, que es un array, traerá bastante información. Cuando el programa detecta una pulsación de ratón, rellena el array con la siguiente información:

- `ev-->0` Esto recibe la constante `evtype_MouseInput`, para indicar que lo que ha ocurrido ha sido una pulsación del ratón (y no, por ejemplo, un redimensionado de la ventana).
- `ev-->1` toma el valor de la ventana que ha sido "pinchada" (en este ejemplo, tomaría el valor `gg_ventana_brujula`)
- `ev-->2` la coordenada X del pixel en que han pinchado (si la ventana fuera de texto, rejilla, aquí tendríamos la coordenada X del carácter sobre el que se ha pinchado)
- `ev-->3` la coordenada Y.

Tu rutina `HandleGlkEvent`, entonces, tendría el siguiente aspecto (al menos en parte):

```
[ HandleGlkEvent ev contexto;
   switch (ev-->0) {
      evtype_MouseInput:
         glk_request_mouse_event(gg_ventana_brujula);
         ! Aqui pondriamos codigo para analizar los resultados y
         ! generar instrucciones nuevas. Por ejemplo, podriamos
         ! hacer que si ev-->2 esta entre 50 y 100, mientras
         ! ev-->3 esta entre 0 y 50, entonces se habria pinchado
         ! en la flecha norte de la brujula, y entonces
         ! lanzariamos el comando para ir al norte
   }
];
```

Obsérvese que, una vez que el juego ha detectado una pulsación del ratón, dejará de detectar más pulsaciones de ratón en esa ventana, a menos que le indiquemos que espere por otra. De ahí la llamada a a `glk_request_mouse_event()` que aparece en el código anterior.

Ahora, debido a que el juego está esperando a que el jugador complete su línea de comandos, no podemos simplemente lanzar un comando interno en el juego del tipo `<<Ir Obj_n>>`. En vez de eso, tendremos que

hacer lo siguiente: Primero, vete hasta la línea en que comienza la rutina `HandleGlkEvent` y declara tres nuevas variables locales: `abortres`, `comandonuevo` y `longcomando`. Después, tras aceptar el click de ratón, cancelaremos la entrada de línea del teclado, poniendo:

```
glk_cancel_line_event(gg_mainwin, 0);
```

Ahora, imagina un comando que, tecleado por el jugador, haría lo que quieres que ocurra. En este ejemplo, el comando podría ser "ir norte", o "norte". Haz que éste sea el nuevo comando en la forma siguiente:

```
nuevocomando = "ir norte";
```

Y después copia el siguiente código tal cual:

```
longcomando = PrintAnyToArray(abortres+WORDSIZE,
   INPUT_BUFFER_LEN-WORDSIZE, comandonuevo);
abortres-->0 = longcomando;
```

Lo que hace lo anterior es completar el proceso para decirle al computador que en este caso particular, pinchar el ratón en ese área concreta de la ventana gráfica, equivale exactamente a teclear "ir norte" y pulsar Intro. Ahora, puedes hacer que ese texto aparezca tras el prompt, para que el jugador vea lo que ha ocurrido, o simplemente hacer algo como:

```
glk_set_style(style_Input);
print "(click de ratón)";
glk_set_style(style_Normal);
new_line;
```

Qué método es el mejor depende de los gustos personales, y del caso particular que tengamos entre manos. En cualquier caso, la última línea de estos bloques de código ha de ser `return 2;`, para indicar que este turno ha finalizado. Et voilà – ya has manejado la entrada por ratón.

6.2. Hiperenlaces

Hiperenlaces son imágenes o trozos de texto que, cuando el usuario pincha con el ratón sobre ellos, enviarán un mensaje a `HandleGlkEvent()`, indicando que han sido pinchados, lo que te permite ejecutar código especial si quieres. Por ejemplo, podrías insertar notas al pie "clickables", que muestren su texto si el jugado pincha en ellas. O permitir a los jugadores que se muevan simplemente pinchando sobre los nombres de las salidas en las descripciones de las localidades... hay infinitas posibilidades. Los hiperenlaces se preparan de forma muy similar a los eventos de ratón para ventanas gráficas o de rejilla. Lo primero que hay que hacer es alertar al programa de que debe estar atento a las pulsaciones de ratón sobre un hiperenlace:

```
glk_request_hyperlink_event(gg_mainwin);
```

(Cambia el nombre de la ventana en la que aparecerán los hiperenlaces, si no es `gg_mainwin`. Si quieres enlaces en muchas ventanas, tendrás que repetir una línea como la anterior para cada una de ellas.) Ahora el programa está listo para responder a un enlace, sólo tienes que construir los enlaces propiamente dichos. Esto se hace con la llamada a `glk_set_hyperlink()`. Esta función recibe un parámetro, que debe ser un número distinto de cero que identifique de forma única a tu enlace. Probablemente querrás crear constantes para referirte a los números que hayas elegido para los enlaces. En tu código, algo como `ENLACE_IR_NORTE` es bastante más claro que algo como, por ejemplo, `67`. Cualquier texto o imagen que se muestre en pantalla después de la llamada a `glk_set_hyperlink()` será parte del enlace, hasta que ejecutes una llamada a `glk_set_hyperlink(0)`. Si dos enlaces van seguidos, puedes omitir esta llamada entre ambos. Tan pronto como comienza un enlace nuevo, se da por terminado el anterior.

Ahora tus enlaces ya están listos, y todo lo que necesitas es programar `HandleGlkEvent()` para que responda a ellos cuando el jugador pinche en uno. Por ejemplo, podría ser así:

```
[ HandleGlkEvent ev contexto abortres comandonuevo longcomando;
   switch (ev-->0) {
      evtype_Hyperlink:
         glk_request_hyperlink_event(gg_mainwin);
         ! Pon codigo aqui analogo al que vimos para manejar
         ! el raton. La unica diferencia es que ahora ev-->1
         ! contiene la ventana, y ev-->2 el numero del link
         ! sobre el que el jugador ha pinchado
   }
];
```

Y al igual que en la sección previa, recuerda que tras responder a un click sobre un hiperenlace, el programa no responderá a otro, a menos que explícitamente le digas que lo haga.

6.3. Pausas y tiempo real

Aunque las aventuras, generalmente, están basadas en turnos, es posible –y en realidad bastante sencillo, una vez que te has familiarizado con la entrada por ratón y por hiperenlaces– incorprar eventos de tiempo real en tu juego. Esto puede estar bien para, por ejemplo, mover a los personajes, que entren y salgan de la localidad mientras el jugador está pensando qué comando dar después, o para implementar una bomba que el jugador realmente deba desactivar en unos segundos. También pueden usarse en conjunción con gráficos para crear animaciones.

El comando para lanzar un evento de tiempo real es `glk_request_timer_events()`, con un número entre los paréntesis, que representa el número de milisegundos que transcurrirán entre los sucesivos eventos de tiempo real. Es decir, cuando el programa llega a una línea que dice `glk_request_timer_events(1000)`, se ejecutará el código apropiado de `HandleGlkEvent()` cada 1000 milisegundos, o sea, cada segundo. Detectar este tipo de evento es muy similar a los del ratón o los hiperenlaces, símplemente equipa a tu rutina `HandleGlkEvent()` con algo como lo siguiente:

```
[ HandleGlkEvent ev contexto;
   switch (ev-->0) {
      evtype_Timer:
         ! Aqui pones el codigo a ejecutar cada cierto tiempo
   }
];
```

La principal diferencia entre estos eventos y los de raton o de hiperenlace, es que no necesitas solicitar de nuevo un evento de tiempo real cada vez que has consumido uno: el programa seguirá generandolos automáticamente hasta que le digas explícitamente que ya no genere más, con la línea `glk_request_timer_events(0);`

Así que, ¿cómo harías una animación? La clave es sacar partido del hecho de que, si enumeras cada cuadro de la animación de forma correlativa en tu fichero de recursos Blorb, las constantes que se les asignarán serán números secuenciales. Así que si tienes una animación con siete cuadros, cada cuadro podría recibir los números 3, 4, 5, 6, 7, 8 y 9, por ejemplo. Pero no necesitas saber qué números son, basta que crees un contador global para los cuadros y le asignes como valor inicial el del primer cuadro:

```
cuenta_cuadros = Primer_Cuadro;
```

Después, pon un código como el siguiente en la rutina `HandleGlkEvent()`:

```
switch (ev-->0) {
   evtype_Timer:
      if (cuenta_cuadros > Ultimo_Cuadro) glk_request_timer_events(0);
      else {
         glk_image_draw(gg_ventana_anim, cuenta_cuadros, 0, 0);
         cuenta_cuadros++;
      }
}
```

Este código mostrará primero el primer cuadro, después el segundo, después el tercero, etc. hasta llegar al séptimo, y después parará. Es cosa tuya asegurarte de que esto ocurra a una velocidad razonable – intenta diferente número de milisegundos desde tu `glk_request_timer_events()` y mira cuál queda mejor.

Ahora, si lo único que quieres es una pausa hasta que el jugador pulse una tecla, la cosa es mucho más sencilla. Hay un comando llamado `KeyCharPrimitive()` que espera por una tecla. Si pones en tu código la línea `x = KeyCharPrimitive();` el juego esperará a que el jugador pulse una tecla, y después pondrá el valor de la tecla pulsada en la variable `x`. Un interesante efecto secundario es que puedes usar `KeyCharPrimitive()` a solas para causar una pausa en la salida, y esperar hasta que el jugador pulse una tecla antes de continuar. Por ejemplo, supón que quieres hacer una pausa antes de una revelación dramática:

```
print "~He considerado todas las pruebas~, dice el detective,
       ~y todo apunta en una dirección. El asesino fue...";
KeyCharPrimitive();
print " <yo! Lo siento, no volverá a ocurrir.~";
```

6.4. Entrada/Salida a fichero

La mayoría de los sistemas de programación de aventuras tienen previsto algún tipo de entrada/salida a fichero, aunque sólo sea para crear y leer partidas guardadas en disco. Inform Glulx permite al programador crear y leer ficheros con cualquier tipo de contenido. Por ejemplo, puedes escribir una serie de juegos con el

mismo protagonista, y transferir información sobre el personaje de un juego al siguiente. Esto puede hacerse si el primer juego crea un fichero que contiene la información pertinente, y el segundo lo lee. He aquí cómo.

Para crear un fichero, lo primero de todo, debes crear un objeto conocido como `fileref`, de modo que el programa tenga una forma de referirse al fichero. Esto se puede hacer de dos formas. Si quieres que el nombre del fichero sea fijo, siempre el mismo, elegido por el programa, el comando es `glk_fileref_create_by_name;` si prefieres que sea el jugador el que elija el nombre del fichero, usarás `glk_fileref_create_by_prompt`. Ambos necesitan tres parámetros.

El primer parámetro indica qué tipo de fichero queremos crear. Consiste en dos constantes, unidas por un signo más. La primera constante será `fileusage_Data` (otras posibilidades se usan sólo para ficheros con partidas grabadas y similares). La segunda constante será o bien `fileusage_BinaryMode` o `fileusage_TextMode`; el modo Binary se usa para ficheros que sólo serán leídos desde otros juegos Glulx (como en el ejemplo de antes que almacenaba datos del personaje), mientras que el modo Text es para ficheros que puedan ser leidos por el jugador (como una lista de cosas divertidas para probar en el juego, un certificado de haber finalizado, o lo que se te ocurra).

El tercer parámetro (veremos el segundo en un momento) es la ROCA del fichero que vamos a crear. Las rocas para filerefs son análogas a las de las ventanas o los canales de sonido. Por supuesto, si estás creando un fichero simplemente para escribir información en él y cerrarlo a continuación, no necesitarás una roca, ya que las rocas son para seguir la pista a los objetos Glk entre diferentes sesiones de juego, y no va a ocurrir un cambio de sesión a mitad de la escritura del fichero. Así que en este caso puedes poner un 0 para este parámetro. Si necesitas mantener el fichero abierto durante varios turnos, en cambio, necesitarás crear una roca para él.

El segundo parámetro depende del método de creación del fichero. Si lo has creado en la forma en la que el programador especifica el nombre del fichero, aquí sería donde iría ese nombre. Sin embargo, hay dos complicaciones. En primer lugar, querrás que el nombre de fichero funcione en diferentes ordenadores y plataformas, así que lo más fiable es que te limites a nombres con ocho caracteres o menos (ya que algunas plataformas no pueden manejar nombres más largos). En segundo lugar, no puedes simplemente poner el nombre del fichero en este punto, sino que tienes que envolverlo en una llamada especial: `ConvertAnyToCString()`. Por ejemplo, para crear un fichero llamado `INFO` en el que poder escribir después, habría que poner:

```
fref = glk_fileref_create_by_name(fileusage_Data+fileusage_BinaryMode,
   ConvertAnyToCString("INFO"), 0);
```

Si vas a dejar que el jugador elija el nombre del fichero, el segundo parámetro contiene información sobre el tipo de fichero que se va a crear. Ya que se usa el mismo comando para crear ficheros de lectura o de escritura, aqui hay cuatro opciones:

- `filemode_Read`: El fichero tiene que existir ya previamente. Al jugador se le pedirá que elija uno de los ficheros ya existentes para ese uso concreto.
- `filemode_Write`: El fichero no debe existir; si el jugador elige un fichero ya existente, se le avisará de que los contenidos serán reemplazados.
- `filemode_ReadWrite`: El fichero puede existir o no, si ya existe se avisará al jugador de que será modificado.
- `filemode_WriteAppend`: El comportamiento es el mismo que el anterior, pero debe usarse para añadir información al fichero en el caso de que éste ya exista.

En este caso podríamos terminar con una llamada como:

```
fref = glk_fileref_create_by_prompt(fileusage_Data+fileusage_BinaryMode,
   filemode_Write, 0);
```

Una vez que has obtenido el `fileref`, puedes abrir el fichero:

```
str = glk_stream_open_file(fref, filemode_Write, 0);
```

El segundo parámetro es uno de los `filemode`s que hemos listado antes. Si has pedido el nombre del fichero al jugador, debe ser el mismo `filemode` que usaste antes.

Una vez que el fichero ha sido abierto, podemos deshacernos del `fileref`:

```
glk_fileref_destroy(fref);
```

Para escribir, hacemos que el stream actual sea ese fichero, y después imprimimos texto normalmente (como si fuera para la pantalla):

```
glk_stream_set_current(str);
print "PUNTOS DE EXPERIENCIA: ", jugador.exp, "^";
```

Una vez finalizado, volvemos a la ventana principal, y cerramos el stream:

```
glk_set_window(gg_mainwin);
glk_stream_close(str, 0);
```

El segundo parámetro de `glk_stream_close()`, al igual que en `glk_window_close()`, sólo tiene interés si quieres conocer cuántos caracteres han sido escritos o leídos. En cualquier otro caso, usa cero.

Para leer de un fichero, el proceso es muy similar, solo que usando `filemode_Read`. Las funciones para leer los datos son `glk_get_char_stream()` para leer un único carácter, `glk_get_buffer_stream()` para leer un array de bytes, y `glk_get_line_stream()` para leer una línea de bytes, hasta el siguiente carácter de línea nueva.

Mira el paquete `Ork` para más detalles. (Ork 1 genera un fichero con información sobre el jugador; Ork 2 permite al jugador cargar el personaje que quieran, o empezar de cero.)