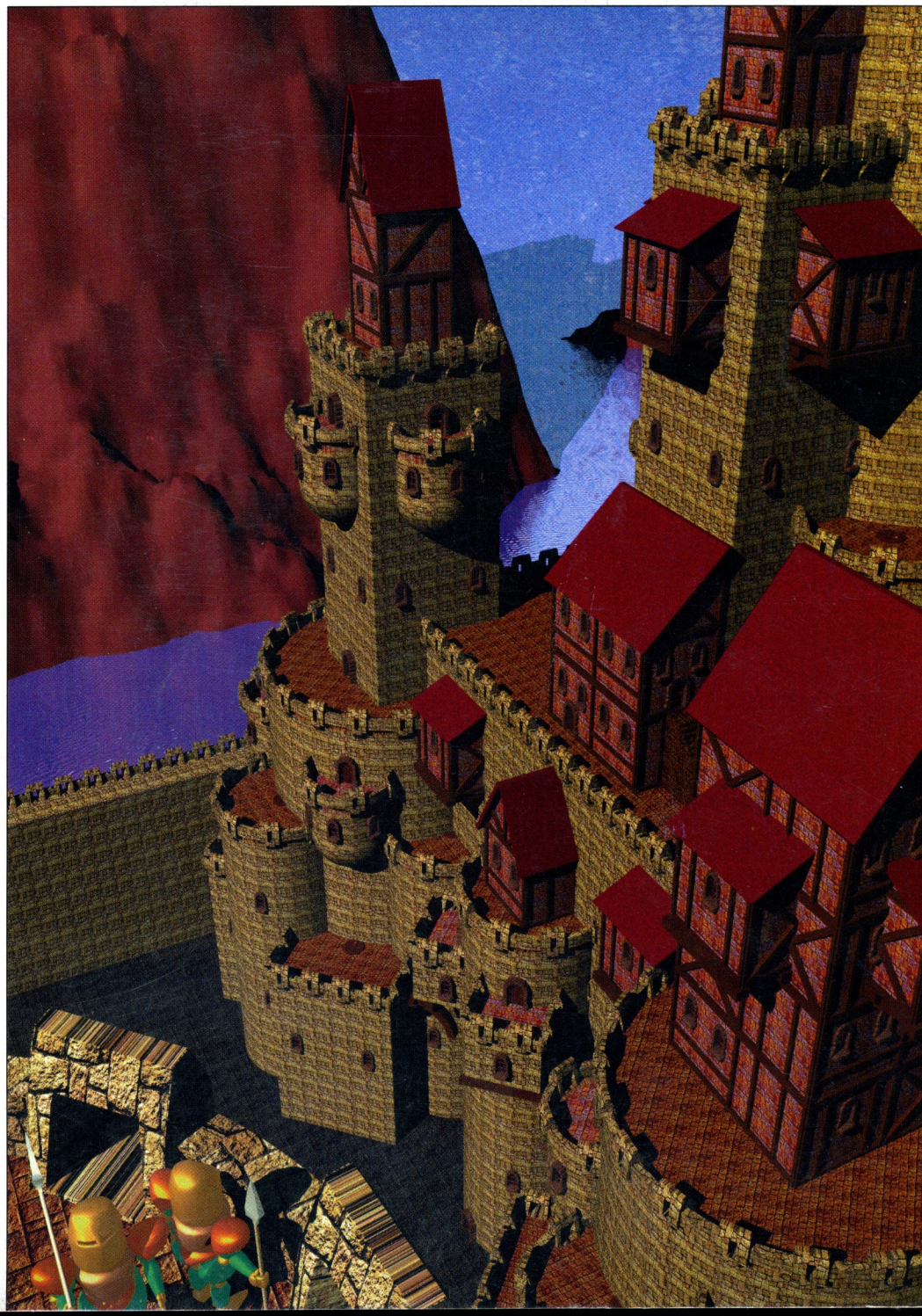
SUPLEMENTO MENSUAL DEDICADO A LA IMAGEN SINTÉTICA

Rendermanía

Número 1



BIBLIOTECA 3D

Modelos POV para castillos y ciudades virtuales

EN EL CD-ROM

Imagine 4.0, versión Try-Out

FORO DEL LECTOR

Los mejores trabajos de los rendermaníacos

CÓDIGO & PIXELS

Programación con #while, #if y rnd

pcmanía

La Ventana



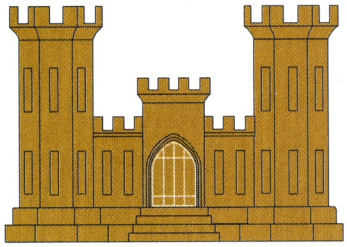
Todos los ficheros de inclusión y los ejemplos podéis encontrarlos en el directorio **PCMANIA/RENDER51** del Cd-Rom

PRESENTACIÓN

Saludo a los viejos amigos

Este es el primer número de la nueva Rendermanía. Sus objetivos no difieren demasiado de los de la antigua sección del mismo nombre. Desde estas páginas confiamos en poder presentaros mes a mes las novedades del mundillo infográfico, en exhibir las escenas creadas por los mejores artistas y en hablar de las técnicas utilizadas para crear dichas escenas.

Aquí se hablará de POV, Polyray, Imagine y de otros programas ya conocidos por el lector. Estudiaremos el funcionamiento de programas nuevos, de herramientas diversas como modeladores, generadores de fractales, conversores de formatos, etc. y crearemos nuestras propias imágenes con dichas tools.



Todos estos contenidos se expondrán en varias secciones. Estas serán bastante flexibles, con un número variable de páginas y la posibilidad de irse alternando unas a otras en cada número. Además ocasionalmente se preparará algún que otro número monográfico. Ahora veamos lo que el lector puede esperar de la nueva Rendermanía.

La primera página será siempre una imagen creada por técnicas infográficas. Normalmente esta imagen habrá sido creada ex-profeso para ser utilizada como portada y los entresijos de su diseño y creación serán desvelados en una sección llamada precisamente así: "La Portada". Para realizar esta portada se empleará cualquiera de los programas de generación de imagen sintética tratados en Rendermanía. De esta manera dicha portada nos servirá para ilustrar un caso práctico de planificación y desarrollo de una escena; aparte de para estudiar los detalles propios del trabajo con las herramientas empleadas. La página o páginas siguientes estarán ocupadas por "La ventana", una sección dedicada a presentar noticias, rumores y novedades.

Otra sección clave es "Cómo...". Esta frase se completará con textos tales como "...exportar modelos para POV", "...crear superficies de revolución con Imagine", etc. En cada número de Rendermanía habrá una o dos secciones encabezadas de esta manera y dedicadas a tratar en detalle diversas cuestiones prácticas.

"El taller virtual", por otra parte, será una sección dedicada a tratar aspectos más especulativos y algo más teóricos. Aquí podremos, por ejemplo, hablar sobre cómo solventar las limitaciones de tal o cual herramienta para crear algún objeto o efecto dado. También pueden compararse aquí los distintos métodos para crear texturas procedurales con programas diferentes, etc.

Los casos prácticos de creación y/o uso de modelos serán tratados en "Biblioteca 3D". Estos modelos generalmente serán los creados para componer muchas de las escenas y portadas de Rendermanía.

Los aspectos de la infografía relacionados con la programación estarán en la sección: "Código & pixels", en la cual los adeptos a la programación gráfica encontrarán artículos relacionados con el desarrollo de utilidades para POV y otros programas. También se tratarán aquí temas diversos tales como la aplicación de texturas, los mapas de voxels, los arboles Bsp, etc. Por último también se hablará de la programación orientada a la creación de modelos o de escenas. Aparte de todo esto aparecerán de tanto en tanto informes y nuevas secciones sobre las que, por ahora, no vamos a dar detalles. Confiamos en que este primer número -casi enteramente dedicado a POV puesto que en el anterior falto espacio para abordar algunos temas-, sea de vuestro agrado.

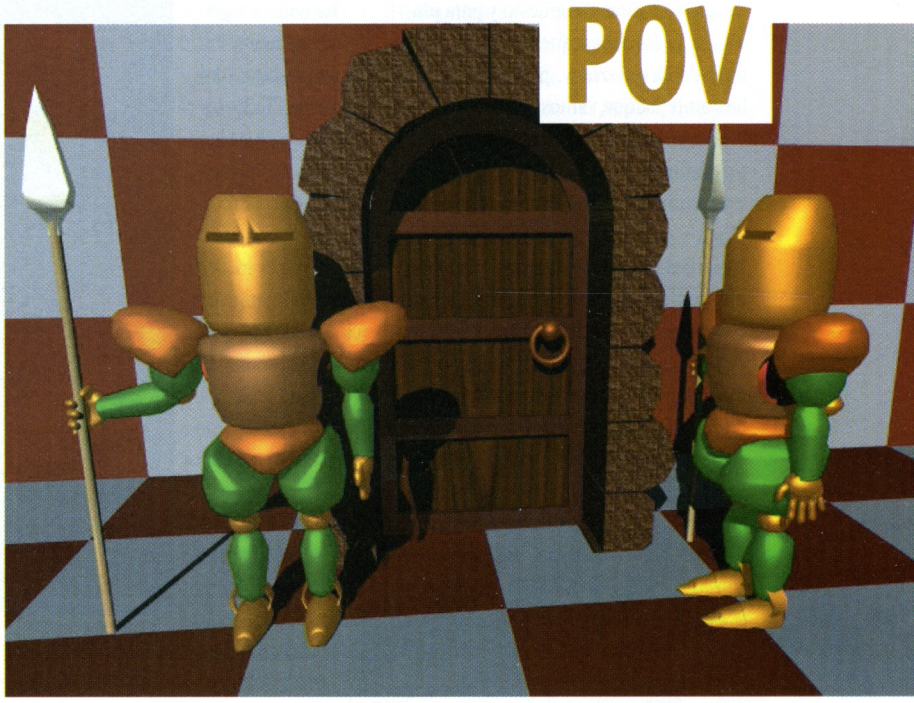
Además, todos vosotros tendréis un espacio permanente en Rendermanía dentro del Foro del lector, apartado donde aparecerán vuestras creaciones, preguntas, sugerencias y críticas.

José Manuel Muñoz



Exportar modelos para POV

En las escenas medievales del número anterior de Rendermanía se echaba en falta un importante detalle, la gente. En efecto, en una escena espacial o en una imagen de una fábrica automatizada las personas no son imprescindibles. En una ciudad medieval, en cambio, el espectador puede acabar preguntándose si la escena no habrá sufrido los efectos de algún tipo de peste negra virtual.



Desgraciadamente el lenguaje escénico de POV no ha sido diseñado para crear formas orgánicas. En teoría puede resultar posible crear formas vivas a golpe de CSGs, pero, en la práctica, esto es inviable. Otra posibilidad radica en el uso de las blobs. En efecto, existe un modelador que, en teoría, es a POV lo que Metareyes es a 3D Studio. Sin embargo esta idea tampoco parece muy práctica ya que las blobs están entre los objetos más lentos de calcular en el lenguaje escénico de POV y una forma humana requeriría docenas, quizá cientos de blobs.

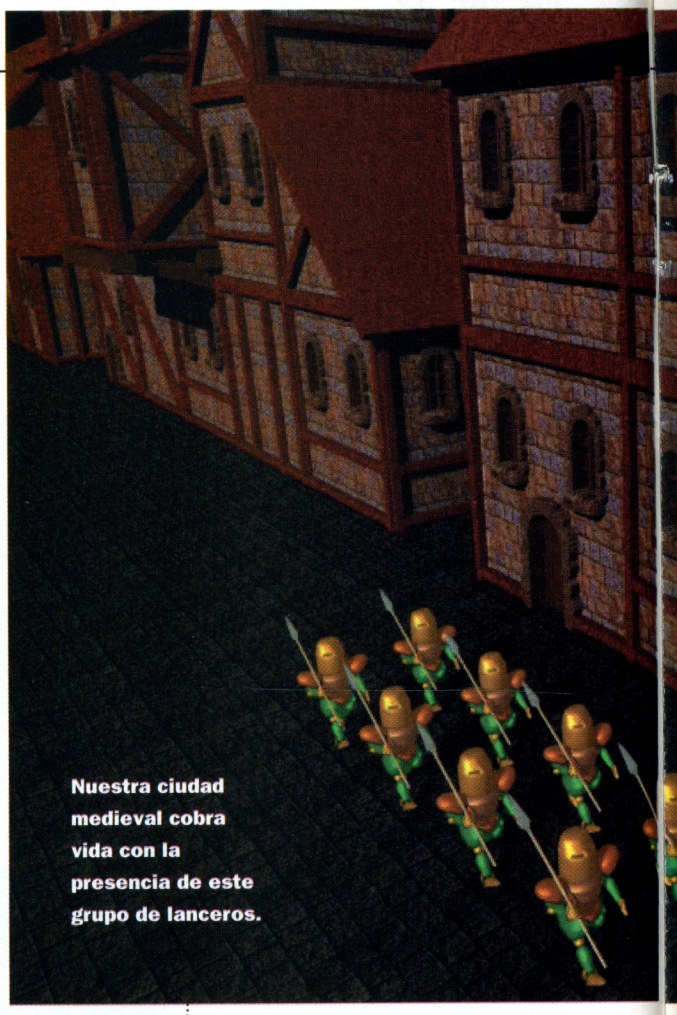
Decisiones

Para poblar las ciudades virtuales de estas escenas, el autor de estas líneas tomó tres decisiones: crear un caballero acorazado –para ahorrarse el engorroso trabajo de modelar ciertas partes complejas como el rostro–, caricaturizar al personaje –así puede tener las proporciones que nos venga en gana– y emplear un modelador para usar herramientas de diseño de las que POV aún no dispone.

La caricatura

El primer paso para crear cualquier modelo es siempre realizar un par de esbozos en papel para conseguir una idea clara de su forma y proporciones.

Cómo...



Nuestra ciudad medieval cobra vida con la presencia de este grupo de lanceros.

Hay que dibujar al modelo en dos vistas: una frontal y otra de perfil. Los esbozos también nos servirán para dividir el modelo en las diferentes piezas de que va a estar compuesto y para planificar la forma en que vamos a construir dichas piezas y, por supuesto, la herramienta que vamos a emplear.

Hecho esto el siguiente paso era el propio modelado del personaje. Para esto, aunque la versión 3.0 puede crear objetos extrude y superficies de revolución, el lenguaje escénico de POV no era demasiado práctico. Téngase en cuenta que muchas de las piezas del lancero son objetos que debían construirse mediante deformaciones o extrusiones de dos vistas distintas; algo que puede hacerse con programas como 3D Studio, Imagine, Amapi, Caligary y otros, pero que POV aún no puede hacer, –el lenguaje escénico de POV está más orientado a la construcción de objetos como naves o edificios–. Hoy no vamos a explicar el proceso de modelado de nuestro personaje. Esto quedará para el futuro, cuando expliquemos el funcionamiento de algún buen modelador. Hoy tan solo nos interesa estudiar los pasos precisos para exportar un modelo creado con alguna de estas herramientas a POV.

Colocación espacial

El primer problema con el que debe enfrentarse el artista que quiere exportar a POV modelos creados desde un modelador, es que la orientación de los ejes en el modelador no tiene por qué ser la misma que emplea POV. Y las dimensiones del personaje tampoco tienen por qué coincidir con las de los modelos creados con el lenguaje escénico. Siendo así, ¿cómo nos las apañaremos para situar a nuestros personajes sobre el suelo de nuestras pov-escenas? ¿Y cómo les daremos las dimensiones adecuadas?

Estudiemos primero la orientación y colocación del personaje. Todos los modeladores suelen usarr varios planos de construcción con distintas vistas para el proceso de modelado. Si la orientación de los ejes en el modelador coincide con la de POV, no tendremos que preocuparnos de reorientar al personaje y bastará con situar a éste sobre el suelo de POV y escalarlo. En POV, la mayoría de los usuarios aceptamos como suelo el plano formado por los ejes X-Z. Este suelo está colocado a una altura 0 en el eje Y en nuestras escenas medievales. Queremos colocar al personaje con los pies sobre este suelo, colocando su centro de gravedad en el eje Y, y situándolo de frente a la cámara, a la que suponemos en algún punto del lado negativo del eje Z. Llamaremos a esto la pov-orientación.

Si en el modelador la orientación de ejes y el plano de construcción usado es el mismo no habrá ningún problema, pero veamos un caso algo más complejo: el de 3D Studio, donde los ejes Z e Y están intercambiados respecto a los de POV. En este caso, para conseguir dar al personaje la pov-orientación adecuada, habremos de colocar al lancero de pie y mirando hacia nosotros en la ventana Top. Además, en la ventana Left, el lancero estará colocado de perfil, con los pies y la cabeza formando una línea horizontal paralela a la de la ventana, con el casco apuntando hacia el lado izquierdo de la ventana y los ojos mirando hacia abajo.

Una vez lograda la pov-orientación hay que colocar los pies del lancero en el suelo y centrarlo en las coordenadas <0,0> del eje (sea cual sea) que abandona perpendicularmente dicho suelo. (Aquí ya el caso es general, para cualquier modelador). Para ello lo mejor es dibujar en el modelador una superficie que sirva como pov-suelo. O sea, crearemos una superficie perpendicular al lancero y la situaremos en el punto 0 del eje en que la línea pies_cabeza es paralela. Esta superficie –que puede ser uno de los lados de una caja– representa



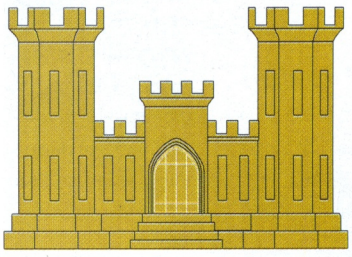
el suelo de POV. Cuando tengamos la caja arrastraremos al lancero hasta depositar sus pies sobre el "pov-suelo". Hecho esto debemos centrar al personaje con el eje en que la línea pies-cabeza de éste es paralela. Para ello, si es necesario, podemos crear otra caja. Hecho todo esto, tan sólo resta escalar al personaje.

Escalas

En nuestras escenas medievales la unidad de medida elegida ha sido el decímetro. Las puertas empleadas para las casas y las torres, por ejemplo, tienen una altura de unas 20 unidades incluyendo el marco. Esto significa que, desde el modelador, habremos de escalar al personaje de modo que la distancia pies-casco sea holgadamente inferior a 20 unidades. Como los pies del personaje están a 0 unidades del plano del suelo, podremos escalarlo tranquilamente. El cambio de escala se efectuará dejando al modelo en el suelo y centrado. Ni que decir tiene que la operación deberá ser global, en los tres ejes. Hecho todo esto, tan sólo nos resta eliminar las cajas y grabar un fichero con el modelo.

Conversión de formatos

Ya tenemos un archivo con un modelo con la escala y orientación adecuadas para incluirlo en nuestras escenas medievales. ¿Y ahora qué? El formato del fichero no será, de seguro, comprendido por POV, ya que éste precisa que los modelos se almacenen en un fichero ascii empleando sentencias del lenguaje escénico. Afortunadamente POV deja una puerta abierta a la importación de modelos poligonales. Esta puerta son las sentencias triangle y smooth_triangle que pueden utilizar los programas de conversión para crear ficheros "traducidos" partiendo de archivos en otros formatos. Herramientas como Wcvt2pov ó 3ds2pov generan ficheros ascii para POV partiendo de archivos que emplean el formato de 3D Studio, Imagine y otros programas. Estas tools pueden encontrarse fácilmente (han sido distribuidas por Pcmanía) y son de un uso muy sencillo.



Una vez que tenemos el fichero traducido hay que tomar nota de algunos detalles. El primero es que lo que tenemos es una "estatua" con una postura determinada del personaje. Si deseamos más estatuas, habremos de regresar al modelador, hacer adoptar al personaje la postura deseada y repetir todo el proceso utilizando otro nombre para el fichero. Otra cuestión importante es que las herramientas de traducción de formatos 3D no suelen traducir las texturas y, de ser éste el caso, no suelen aplicar bitmaps. Por ello lo mejor será usar colores para los personajes y aplicar las texturas desde POV.

Por último debe existir un identificador que sea igual a la unión de todas los objetos que componen al personaje. Si éste identificador no existe deberemos crearlo nosotros mismos trasteando en el archivo traducido con un editor de texto. ¡Ojo! Los ficheros con modelos poligonales suelen ser bastante grandes, de 1 megabyte o más. Habrá que usar un editor que pueda manejar archivos de este tamaño (el edit de Dos no sirve). Una vez que tengamos un identificador para designar al modelo importado podremos emplear a éste en la escena tantas veces como queramos (véase el ejemplo de desfile.pov).

Problemas con Bitmaps

Ya hemos estudiado, en el pasado número, un caso concreto de aplicación de texturas bitmap sobre los modelos usados para componer las pov-ciudades medievales. Hoy profundizamos en el mismo tema, estudiando diversos problemas relacionados con el uso de bitmaps en los objetos creados con POV.



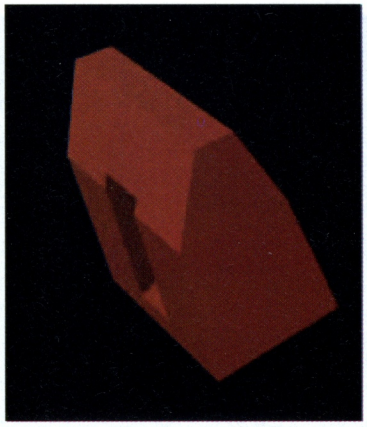
Aunque las texturas procedurales resultan ideales para crear objetos con apariencia de madera, piedra, metal y de otros tipos, tendremos serias dificultades en otros casos. Así, aunque en la versión 3.0 de POV se incluye un nuevo patrón llamado brick (ladrillo), nos resultaría muy difícil construir con él una textura de apariencia similar a las empleadas para las paredes de las casas y muros del castillo.

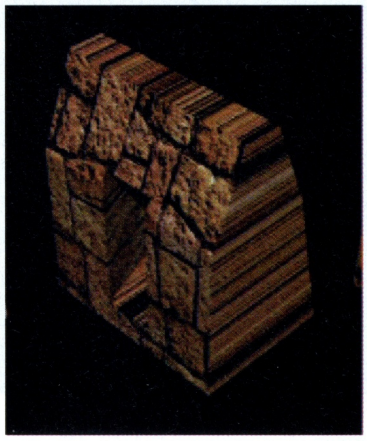
El problema fundamental

Existen algunas diferencias fundamentales entre las texturas procedurales y los bitmaps. Las primeras son construidas por el usuario usando las sentencias disponibles para ello en el lenguaje escénico mientras que los bit-

maps son mapas de imagen, en formato tga, gif u otros, con los que recubriremos los objetos de la misma forma que si los envolviéramos con un papel pintado. Las texturas procedurales, por el contrario, basándose en los parámetros suministrados por el usuario en las sentencias de patrones, distorsión de tramas, pigmentación, etc., emplean métodos algorítmicos para construir invisibles bloques tridimensionales que impregnan todo el volumen espacial de los objetos sobre los que se aplican. Así, si aplicamos una textura algorítmica de madera sobre un cubo y realizamos después una operación CSG de recorte sobre él, observaremos que el interior del objeto también tiene los anillos y la apariencia de la madera.

Esta aplicación tridimensional de las texturas procedurales es una característica muy útil, ya que nos deja las manos libres para concentrarnos en la construcción de los objetos. Sin embargo, aunque POV dispone de una buena librería de texturas de este tipo, puede ocurrir que no exista ninguna similar a la que precisamos en ese momento. En este caso sólo tendremos dos alternativas: crear nuestra propia textura algorítmica usando las sentencias de POV o bien utilizar un bitmap. La primera alternativa puede ser o muy sencilla o bien extraordinariamente difícil, dependiendo de lo que pretendamos lograr. En el caso de los muros de las casas y torres medievales, el que esto escribe optó por utilizar un par de bitmaps.







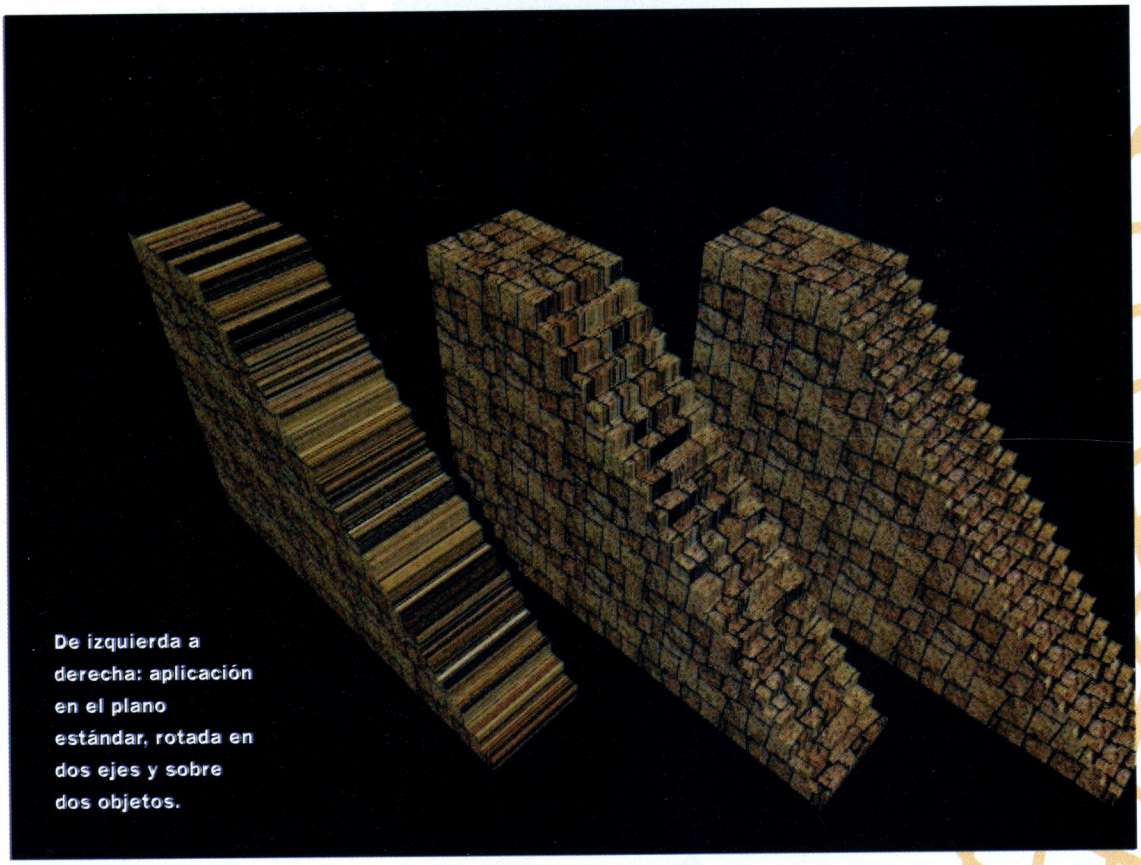
Distintos ejemplos de aplicación de bitmap sobre un mismo elemento.

Aplicación de bitmaps

Aunque ya hemos comentado los aspectos básicos de la aplicación de bitmaps en POV, volveremos hoy sobre el tema ya que ha afectado de modo directo a la forma en que se han construido los objetos sobre los que habían de aplicarse estas texturas. También veremos algunos fallos de aplicación que quizá no fueron advertidos en las escenas del número anterior, ya que la cámara había sido colocada a una cierta (y prudente) distancia. En el número anterior de Rendermanía se explicó el formato de la sentencia image_map, con la que el usuario puede aplicar un bitmap sobre el objeto eligiendo el tipo de aplicación a que va a recurrirse para efectuar la "envoltura". Recordemos que podíamos escoger entre 4 posibles tipos de recubrimiento dando diferentes valores al modificador map_type; mapeado plano (0), esférico (1), cilíndrico (2) y toroidal (5). Así, en las sentencias…

```
pigment{image_map{tga "text1.tga" map_type 0} scale <14,14,1>}
```

…se especifica que la textura "text1", en formato tga, recubrirá al objeto con un mapeado de tipo plano. Por defecto la textura se extiende siempre sobre un área que va desde las coordenadas <0,0> hasta <1,1> en el plano X-Y y se repite cuantas veces sean precisas para cubrir totalmente el objeto (a menos que incluyamos la palabra "once"). Como este área de aplicación suele resultar inadecuada con respecto al tamaño del objeto, suele indicarse también un factor de escala para la textura en los ejes



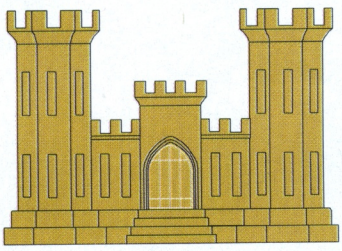
De izquierda a derecha: aplicación en el plano estándar, rotada en dos ejes y sobre dos objetos.

X e Y sobre los que el bitmap, por defecto, se aplica.

Los dos bitmaps empleados se aplicaron en estas escenas sobre todos los objetos que habían de exhibir una apariencia de roca, como paredes, escaleras, torreones, etc. En todos los objetos -incluso en los de forma esférica-, el tipo de aplicación elegida fue la planar (luego veremos por qué). Comenzaremos estudiando un caso de mala aplicación de uno de estos bitmaps sobre un objeto.

El bitmap determina la forma

En castlib (la librería escrita para crear escenas de construcciones medievales) hay varias escaleras de distintos tamaños. Algunas se emplean para acceder a las murallas y otras sirven para llegar hasta la puerta del segundo piso de algunas casas. Todas las escaleras están construidas de la misma forma; mediante una unión de cajas que hacen las veces de peldaños. La escalera está orientada lateralmente con respecto al



eje Z. Mirando desde el lado negativo de este eje veremos como los peldaños ascienden hacia el lado izquierdo de la pantalla. En este objeto el bitmap se aplicó inicialmente en el plano por defecto con lo que la aplicación resulta correcta en las caras laterales de la escalera. No ocurre así en cambio con la parte superior, donde parece como si los pixels correctamente aplicados en el lateral se estirasen a lo largo del objeto, en el plano de aplicación, lo cual es precisamente lo que sucede en el mapeado plano (véase la escena). Si para solucionar esto rotamos 90 grados el plano de aplicación para que la textura se aplique sobre el plano X-Z, no

ganaremos gran cosa ya que en dicho caso el bitmap quedará bien aplicado en la parte superior de la escalera y mal en los laterales de la misma. De esto deducimos que si el plano de aplicación tiene unos 90 grados con respecto a una superficie del objeto, los pixels que correspondan en la textura se "estirarán" para cubrir toda esa superficie.

Una solución para esto residiría en rotar el plano de aplicación de modo que haga ángulo con todas las superficies del modelo. Sin embargo esto también tiene sus pegas. Si deseamos que la aplicación de la textura resulte sin inclinaciones en todas las superficies del modelo, no tendremos más remedio que aplicar la textura más de una vez. Sin embargo esto, colocar dos o más sentencias pigment}, no dará el resultado esperado.

La única salida que nos queda es reescribir el modelo de forma que se tengan en cuenta las peculiaridades del mapeado plano. Para ello dividiremos el objeto en tantas piezas como planos de mapeado vayamos a aplicar sobre él. En el caso de las escaleras estableceremos, en principio dos planos de aplicación del bitmap; uno vertical, siguiendo el eje Z, y otro horizontal, siguiendo el eje Y. Crearemos un nuevo modelo que será la unión de dos operaciones CSG; una diferencia y una intersección. Para ello emplearemos una pieza, a la que llamaremos escal5m, que será la antigua escalera.

En la primera operación restaremos a escal5m una caja que dividirá la antigua escalera en dos bandas muy delgadas. Sobre estas bandas aplicaremos el mapeado estándar a lo largo del eje Z. La siguiente operación de la unión es una intersección sobre escal5m empleando una caja de idénticas dimensiones a la de la caja de la operación anterior. Sobre esta última pieza resultante de la intersección aplicaremos la misma textura rotada 90 grados en el eje X (aplicación a lo largo del eje Y). Así pues la nueva escalera será idéntica en su forma a la antigua, diferenciándose tan sólo en que el bitmap está mejor colocado. Esta descripción puede resultar un tanto liosa pero el nuevo modelo sigue siendo muy sencillo ya que no es sino la unión de dos escaleras: una en la que se ha aplicado el mapeado primitivo a lo largo de Z, y otra (el centro de la escalera) sobre la que hemos cuidado la aplicación en las superficies horizontales de la escalera.

El resultado es superior al de la escalera inicial de la cual se ha partido pero aún está lejos de ser perfecto. Para empezar no hemos cuidado la aplicación en el lado trasero de la escalera y en los lados verticales de los peldaños. Esto podría ser remediado rotando el plano de aplicación 45 grados en el eje Z. Sin embargo el resultado –que podemos ver en la última escalera de la imagen– queda algo extraño. Ello se debe a que, al haber inclinado el plano de aplicación, las piedras de la textura son más alargadas en los peldaños y la cima, que en los laterales.

Podríamos efectuar aún un nuevo intento en el cual colocásemos la textura en cada caja-peldaño de la escalera pero esto resultaría contraproducente, puesto que el modelo resultante requeriría mucha más memoria de cálculo que el antiguo. De hecho, la escalera central de la imagen precisa también más memoria que la escalera inicial y por ello es adecuado guardar dos modelos: el antiguo se empleará en escenas "a vista de pajaro" donde la cámara abarque un amplio espacio y el nuevo en imágenes donde la cámara esté cerca de la escalera. Esta es la razon de que algunas líneas o bloques de sentencias estén englobadas por comentarios en los ficheros que componen Castlib. ¡Hay que intentar ahorrar memoria de cálculo a toda costa!



Una de las primeras pruebas de aplicación de los bitmaps.

Más ejemplos

Como hemos visto en el ejemplo anterior, el uso de bitmaps puede afectar a la forma en que se construyen los objetos. Otro caso similar al anterior es el de las torres cuadradas, las cuales han debido construirse empleando 4 paneles-caja en cada piso para evitar el efecto de estiramiento. También sucede lo mismo con los objetos que hacen las veces de aspilleras en los muros y torres, con los cuales se ha usado un truco muy similar al descrito para la escalera. Hay, además, otras muchas piezas que han precisado alteraciones para que la aplicación del bitmap no resultase catastrófica pero nada ha sido tan problemático como la creación de las piezas para las torres circulares.

¿Aplicación cilíndrica?

En una librería de formas para escenas medievales no podían faltar las torres de forma cilíndrica. En principio una torre de este tipo debería ser un cilindro sobre el que se ha efectuado un mapeado del tipo 2 (cilíndrico) con el bitmap usado para representar los muros. Sin embargo algo tan aparentemente sencillo resultó un verdadero suplicio para el autor de estas líneas. ¿Deficiencia del POV? ¿Fallo garrafal del autor de castlib?

En fin, repasemos la teoría de la aplicación cilíndrica. En este tipo de mapeado se asume que un cilindro de cualquier diámetro se extiende a lo largo del eje Y, quedando sus dos bases paralelas al plano X-Z. El bitmap se enrolla alrededor del cilindro de la misma forma que lo haría un papel pintado en una lata de conservas. La envoltura se realiza una sola vez, no siendo posible repetir la textura en torno al cilindro (o al menos éste parece ser el caso). Esta banda de envoltura va desde el punto 0 al 1 del eje Y, por lo que el bitmap se repetirá cuantas veces sea preciso a lo largo de este eje hasta recubrir completamente el objeto.

Este es, en definitiva, el problema, ya que si se pretende crear una torre cuyo diámetro sea de 10 ó 20 metros, precisaremos que la banda de envoltura se repita bastantes veces en torno al cilindro, lo cual no parece estar contemplado por POV. Podemos escalar la banda a lo largo del eje Y, pero, de cualquier modo, ésta se enrollará para cubrir una sola vez todo el cilindro.

La chapuza

Teniendo en cuenta esta extraña limitación del mapeado cilíndrico, el aplicar el bitmap a la torre se convirtió en una frustración para su autor. Finalmente, después de numerosas pruebas, se optó por una idea que solucionaba este problema aunque, técnicamente hablando, se trata de una verdadera chapuza. La idea consistía, lisa y llanamente, en olvidar el mapeado cilíndrico y usar el mapeado planar. Para ello, en lugar de emplear un cilindro para el cuerpo de la torre, se creó una nueva pieza cuya forma es la del típico trozo que nos entregan al cortar en pedazos una tarta de forma cilíndrica. De esta manera el cilindro empleado para la torre se construyó con la unión de varios objetos de este tipo, en cada uno de los cuales se aplicó el bitmap usando la aplicación planar. Cada uno de estos objetos se rotó en torno al eje Y, hasta conseguir la forma del cilindro.

Después de realizar varias pruebas se determinó que construir el cilindro con menos de 8 de estas piezas no era conveniente. Naturalmente a mayor número de trozos de tarta, menor era la curvatura de cada pieza con lo que la aplicación resultaba más perfecta. Los mejores resultados se obtuvieron con 12 y 16 piezas por cilindro. (De nuevo aquí hay que hacer notar que esto requiere mucha más memoria que el empleo de un único cilindro pero…)

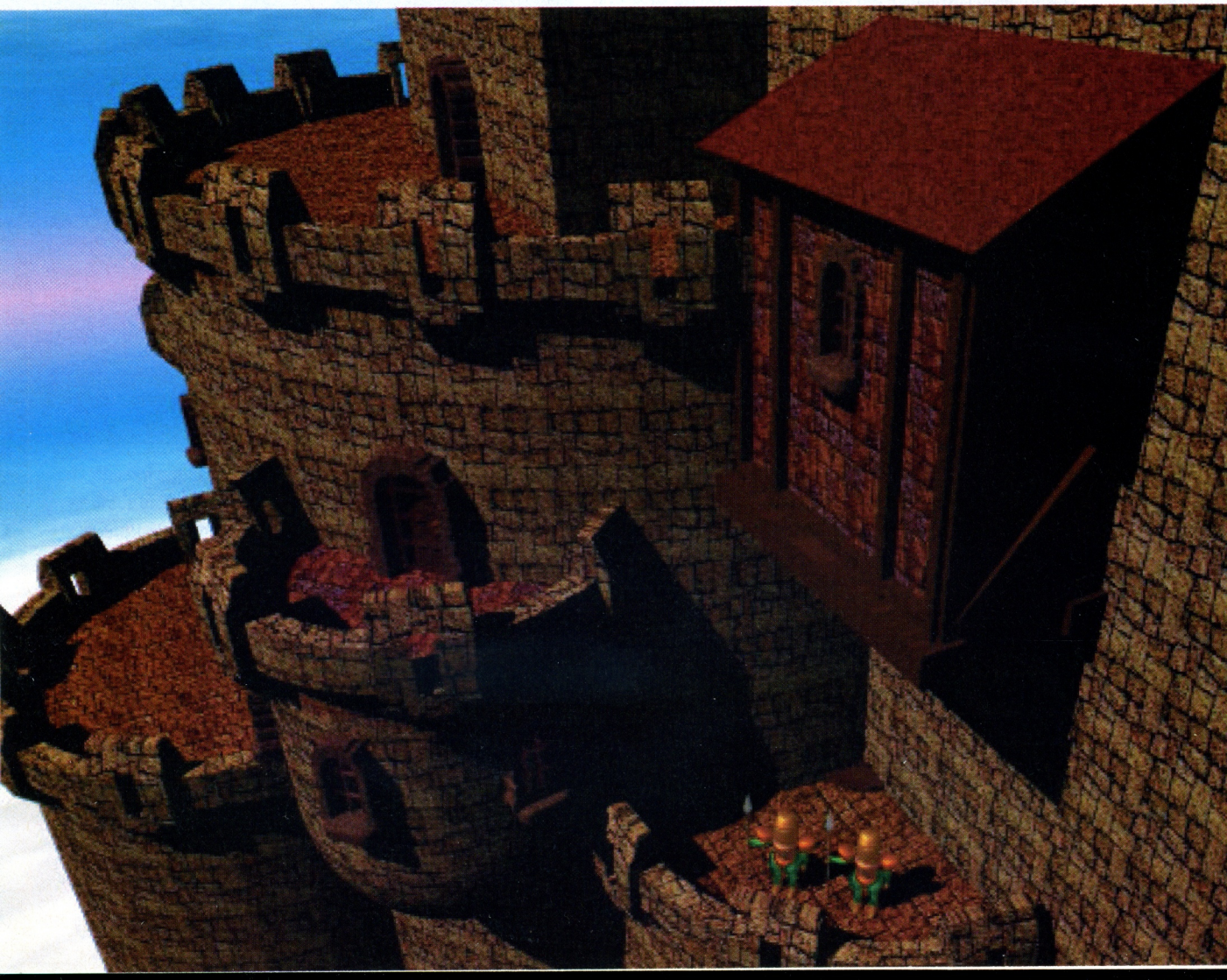
Otros problemas

Otro problema de difícil solución es el hecho de que un mismo bitmap puede dar diferentes resultados dependiendo de la distancia a la que enfoquemos el objeto sobre el que se efectúe la aplicación. Un bitmap con ladrillos de distintos colores puede quedar muy bien sobre la pared de una casa enfocada a corta distancia. Desde lejos, sin embargo, puede suceder que los tonos se mezclen y obtengamos un resultado, cuanto menos, extraño. Por ello puede ser buena idea emplear alguna herramienta como Photoshop para obtener varias versiones de la textura.

Por último hay que resaltar un detalle de suma importancia. Podría pensarse que en la escena generada por POV la apariencia de los objetos sobre los que hemos aplicado la textura ha de quedar idéntica a la del bitmap original. Sin embargo esto no tiene por qué ser así. A menos que se le indique otra cosa, POV aplica un valor por defecto de 0.6 para diffuse, con lo que la textura aplicada resultará un 40% más oscura en la escena generada que en el bitmap. Para remediar esto basta con añadir un valor de 1 para diffuse. Esta sentencia especifica la cantidad de luz procedente de las fuentes que se refleja sobre el objeto. (Véanse las declaraciones de las texturas de muros en castlib).

Modelos POV
para ciudades y castillos medievales

Esta sección, cuyo propósito es describir los pormenores del diseño de los modelos creados para Rendermanía, se inaugura hoy con el análisis de los modelos que componen la librería de formas medievales.

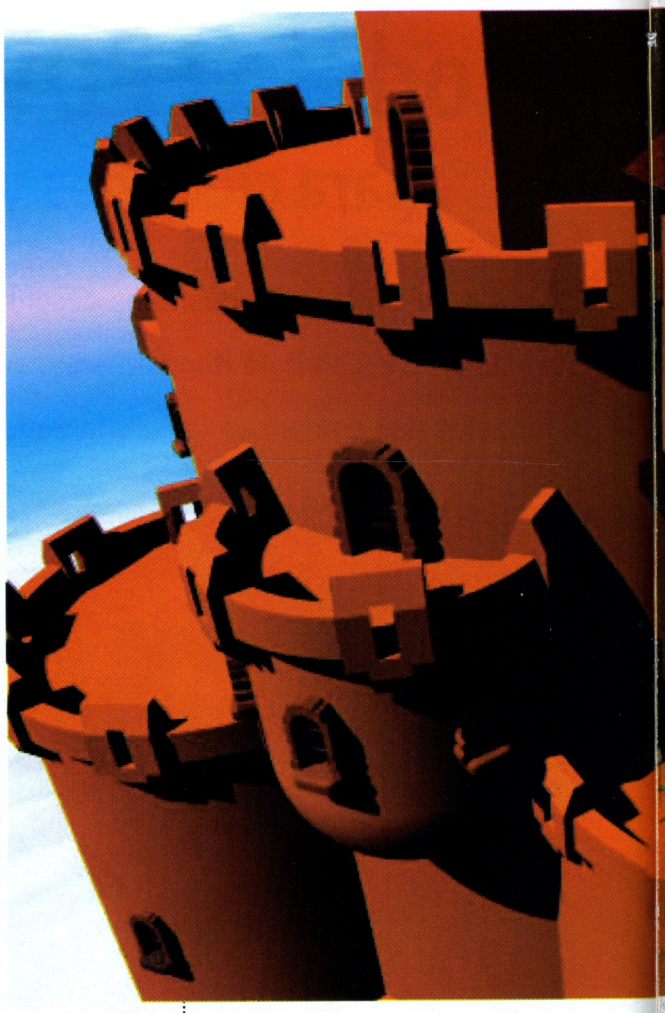


Biblioteca 3D

La última versión de la librería de formas medievales llamada Castlib está compuesta por tres ficheros: Castle20.inc, town.inc y lateral.inc. Para utilizar Castlib, el usuario deberá crear un fichero.pov donde dispondrá convenientemente cámara y luces y donde referenciará y situará los modelos de la librería. Únicamente será preciso sumar una sentencia #include "castle20.inc" en el fichero .pov. (Castle20.inc ya tiene, a su vez, sentencias include para leer town.inc o lateral.inc). Sin embargo, al renderizar cualquier escena, es probable que el disco duro empiece a ronronear y que, según sea el caso, el tiempo de parsing (preparación y compilación de la escena) se dispare. ¿Por qué? Al concluir la generación de la imagen, POV 3.0 exhibe una pantalla informativa con diversos detalles de interés. Entre estos se halla el apartado "Peak memory used". Junto a él veremos una cifra que indica la cantidad total de memoria empleada por POV para construir un modelo interno de la escena. Esta cifra puede referirse no sólo a la Ram sino también, en caso de que nuestro ordenador no disponga de la suficiente memoria, a memoria virtual de disco duro. Naturalmente, cuanto más supere esta cifra al número de Megabytes de RAM del ordenador, más trabajo extra tendrá que hacer el disco duro. ¿Por qué ocurre esto? ¿Existe algún modo de reducir el consumo de memoria?

Niveles de detalle y problemas

Al proyectar Castlib, el objetivo de su creador era preparar escenas con castillos enormes compuestos por multitud de torres, puentes y murallas de distintas formas y tamaños. Naturalmente, a fin de disponer del tiempo suficiente para crear tantos objetos, ello implicaba que estas estructuras debían ser forzosamente bastante sencillas. La idea era que las estructuras de mayor tamaño se recargarían con otras de menores dimensiones las cuales, a su vez, estarían rodeadas de otras de menor tamaño. De esta manera se confiaba en obtener castillos de una aparente complejidad a cambio de un mínimo esfuerzo en el modelado. Para reforzar esta impresión se dedicó más tiempo al modelado en detalle de los objetos pequeños; ventanas, puertas, aspilleras, etc. De esta forma, al incluir estos objetos en la composición de las estructuras de mayor tamaño, estas ganaban en realismo. Así pues puede hablarse de niveles en estos objetos; una ventana es un objeto de "bajo nivel", una casa un objeto de un nivel mayor y un castillo o una ciudad tendrían el nivel más alto.



Esta era, al menos, la idea inicial. No obstante, a medida que se iban agregando nuevos objetos a la librería, algo iba quedando más y más claro: aunque los ficheros ascii de Castlib no ocupan demasiado, el modelo interno que POV prepara a partir de dichos ficheros si exige una cantidad de memoria considerable, –en el momento de escribir estas líneas la simple inclusión de castle20.inc, ya requería una peak memory de unos 55 Megabytes–. Para ahorrar memoria y



Una prueba rápida de la escena sin texturas.

por consiguiente ahorrar tiempo de cálculo, son recomendables las siguientes medidas:

-En primer lugar podemos hacer una copia de los ficheros originales. Hecho esto desactivaremos a todos aquellos objetos o estructuras que no vayan a participar en la escena. Esto podemos hacerlo marcándolos como comentarios (con "/*" y "*/").

-En las pruebas intermedias podemos redefinir las texturas y usar un pigmento simple. Al emplear un color en vez de un bitmap, ahorraremos mucha RAM.

-Si la escena va a prescindir de casas y de estructuras laterales (para muros) y de algún elemento suelto, es conveniente prescindir de town.inc, para lo cual podemos eliminar la correspondiente sentencia #include en castle20.inc. Esto permitirá ahorrar 30 megabytes o más. Aunque, si el castillo va a emplear estructuras laterales y colgantes, habrá que activar el include a lateral.inc.

Como los objetos creados en POV no están compuestos por polígonos resulta difícil estimar a priori la memoria que puede demandar un proyecto dado. Casi todos los modeladores crean objetos a base de polígonos y suelen incluir herramientas para reducir el número de caras sin alterar excesivamente a los objetos. En POV esto (¡Ay!) no es posible. El único camino es sustituir los objetos más complejos por otros más simples. Teniendo esto en cuenta el autor de estas líneas probó a sustituir ciertos objetos complejos que se repiten mucho –las aspilleras, los marcos de puertas y ventanas– por otros más simples pero, sorprendentemente, esto no ahorró demasiada memoria dentro del conjunto.

Composición de las torres

Al realizar los primeros diseños cada torre se construía con un bloque propio de sentencias y tenía dimensiones puestas a capricho. Esto pronto se reveló como algo ineficaz: las torres debían conectarse entre sí con murallas, debían existir escaleras para acceder a ellas y tenían que adornarse con otras estructuras de menor tamaño –estructuras colgantes– para lograr una mayor vistosidad.

Pronto quedó patente, pues, que las dimensiones debían ser normalizadas. Además, a fin de obtener mayor variedad, cada torre no fue definida aisladamente. En lugar de esto se creó una colección de secciones que podían combinarse entre si para formar torres de distinta altura y forma.

Esta lista de secciones se dividió en varios grupos puesto que se deseaba crear torres de diferentes anchuras y de dos formas básicas: redonda y cuadrada. Dentro de dichos grupos, cada sección pertenece a un tipo determinado. Puede ser de tipo base, media o final. Las primeras son empleadas como base de las torres y suelen tener una o más puertas. Las secciones medias son las más numerosas y pueden incluir ventanas, torres colgantes y puertas (para acceder a las murallas).

En cuanto a las secciones finales, todas tienen 5 metros de altura (mientras que las demás suelen tener 10 metros de alto) y con ellas se construyen los techos de las torres.

Los nombres de las secciones parecen bastante crípticos pero no ofrecen dificultades. Así "secb10_10r01" quiere decir sección básica de tipo redondo (torres circulares), de 10 metros de diámetro y 10 de alto y número 1. Otro ejemplo puede ser "secf20_5c02", lo cual significa que se trata de una sección de tipo final y de forma cuadrada, con 20 metros de diámetro y 5 de alto. El número 2 significa que podemos escoger entre varias secciones de este tipo para rematar nuestra torre.

Todas las secciones están definidas de modo que descansan sobre el suelo, o sea que su base está en el punto 0 del eje Y. Veamos cómo crear una torre usando algunas de estas secciones:

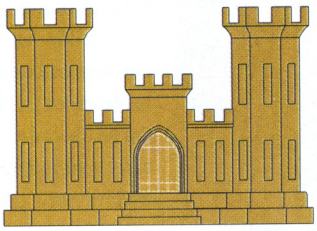
```
#declare TORRE25_10v=union{
  object{secb10_10r}
  object{secm10_10r translate<0,100,0>}
  object{secf10_5r translate<0,200,0>}
```

Esta torre tiene 25 metros de diámetro y forma circular. Al comienzo de cada sección hay un breve comentario indicando sus particularidades por lo que las tareas de construcción no deberían ser demasiado difíciles para los aspirantes a maestro cantero.

En cuanto al diseño de las secciones propiamente dichas, éste ha estado fuertemente determinado por el hecho de que se deseaba aplicar texturas bitmap. Por esta razón las secciones de las torres circulares son "trozos de pastel" y las secciones de las torres cuadradas están formadas por paredes similares a los paneles de las casas en vez de por una única caja. Por último hay que señalar que todas las secciones están huecas y, cuando hay ventanas, tienen practicados los correspondientes agujeros de resta. Esto ha sido hecho así en previsión de que se prepare alguna escena nocturna con fuentes de luz situadas dentro de las torres. (Ojo: las secciones, salvo las de cima, no tienen suelo ni techo).

Estructuras de nivel medio

Para que la apariencia de las torres y muros no resultase demasiado sencilla se añadieron algunos modelos extra. Este es el caso de las torres colgantes y de las estructuras laterales.

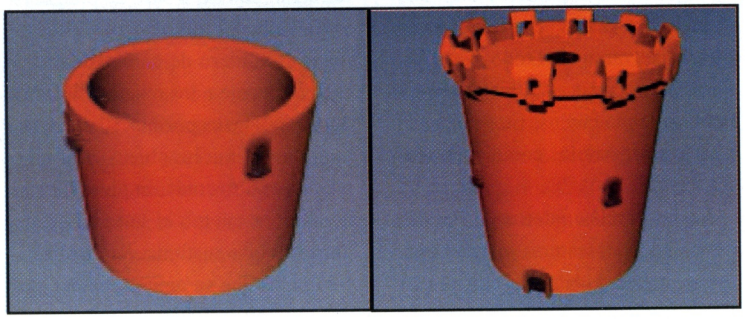


Las primeras son pequeñas torres de reducido tamaño adosadas a los muros laterales de las secciones medias. Las hay de forma circular y cuadrada y carecen de la portilla superior que tienen todas las torres normales. (Para acceder a ellas, los habitantes del castillo deberán usar las puertas previstas para ello en la sección inmediatamente superior a la que incorpora estas estructuras). Las torres colgantes están pegadas a los muros por su centro de gravedad y tienen unos diez metros de alto. Dado que las puertas de acceso están colocadas en la sección de arriba no podemos colocar otra sección de torres colgantes encima; a menos que creemos una nueva sección con puertas y torres colgantes intercaladas. La excepción a esto son las estructuras "colgantes".

En cuanto a las estructuras laterales, son casas colocadas en town.inc y pensadas para adornar los muros interiores de los castillos. Realmente town.inc acabó convirtiéndose en una ciudad por culpa de estos adornos iniciales. Algunas de estas estructuras no pueden ser colocadas en los muros más bajos, de 10 metros del alto.

Detalles y carencias

Cuando se diseñaron las aspilleras, las puertas, las ventanas, etc., se intentó tener en cuenta el tamaño de los posibles habitantes virtuales del castillo. Las aspilleras tienen una abertura interior pensada para que los posibles arqueros disparen al enemigo con como-



Elementos en orden de complejidad creciente.

didad y seguridad, las almenas tienen una anchura mínima adecuada para colocar las puertas de las torres y para que se paseen por ellas los guerreros. Incluso se intentó que el tamaño de los peldaños en las escaleras no resultase excesivo para las sufridas piernas de nuestros guerreros virtuales. Como los personajes están centrados en el eje Y y colocados a la altura del suelo resulta posible colocarlos fácilmente en cualquier punto del castillo (véanse las fotos).

A pesar de todo, sin embargo, los escenarios construidos a partir de castlib son como decorados de cartón piedra; los pisos de las torres no tienen techos, ni suelos, ni, por supuesto, muebles. No existe tampoco una arquitectura interna del castillo ni hay puente levadizo ni rastrillo ni las típicas escaleras de caracol. Es posible que en un futuro se añadan nuevos detalles pero lo cierto es que las exigencias de memoria de POV pesan ya como una losa sobre este proyecto.

Realmente es una pena que POV no tenga una sentencia equivalente al Grid del Polyray ya que, parece ser, esta sentencia coloca copias de objetos originales sin más gasto que el necesario para indicar su posición. Por otro lado, Polyray carece de las nuevas sentencias de programación de POV 3.0.

Estructuras de mayor nivel

En castlib realmente no se han definido las estructuras de mayor nivel. Hay ejemplos de definiciones de torres y, al final de castle20.inc, un ejemplo con un castillo completo. Estos ejemplos, sin embargo, están marcados como comentarios a fin de ahorrar memoria. Ahora examinaremos el castillo incluido como ejemplo para estudiar la filosofía a la que deben atenerse los constructores de castillos; una filosofía que debería ser válida para cualquier proyecto arquitectónico con POV.

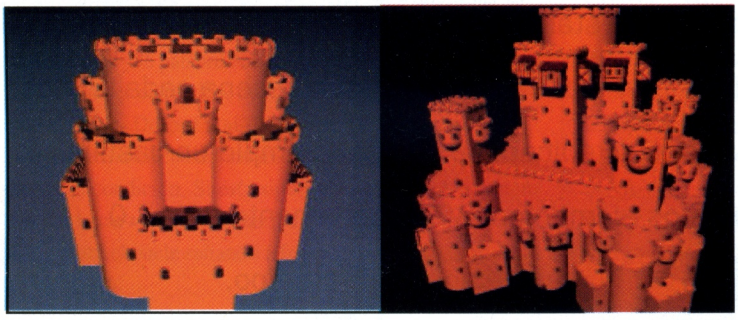
En el texto que define al castillo de nuestro ejemplo lo primero es la definición de las torres simples que utilizaremos para su edificación. Dichas torres son simples uniones de secciones colocadas a distintas alturas. Con dichas torres seguidamente se definen las nuevas uniones que forman los dos grupos de torres utilizados en el ejemplo. El primer grupo, llamado torbala, es el empleado para adornar la estructura principal del castillo (llamada body50_35). El objeto principal de torbala es una torre redonda de 20 metros de ancho por 35 de alto. A pesar de sus torrecillas colgantes, esta torre resultaría excesivamente sencilla de no estar rodeada por otras 8 torres redondas y cuadradas de 25 y 15 metros de altura que se intercalan entre si. La idea inicial era recargar también a estas torres secundarias con estructuras colgantes pero el resultado era de un recargamiento excesivo por lo que al final se prescindió de estos objetos.

El cuerpo principal del castillo –body50_35– son 4 paredes que forman una caja de 50 metros de ancho y largo por 40 metros de alto. En sus 4 esquinas hay 4 grupos de torres torbala y en su parte superior otro grupo similar de torres. Además se emplean 4 torres centinelas de gran altura dispuestas entre los grupos laterales y la caja.

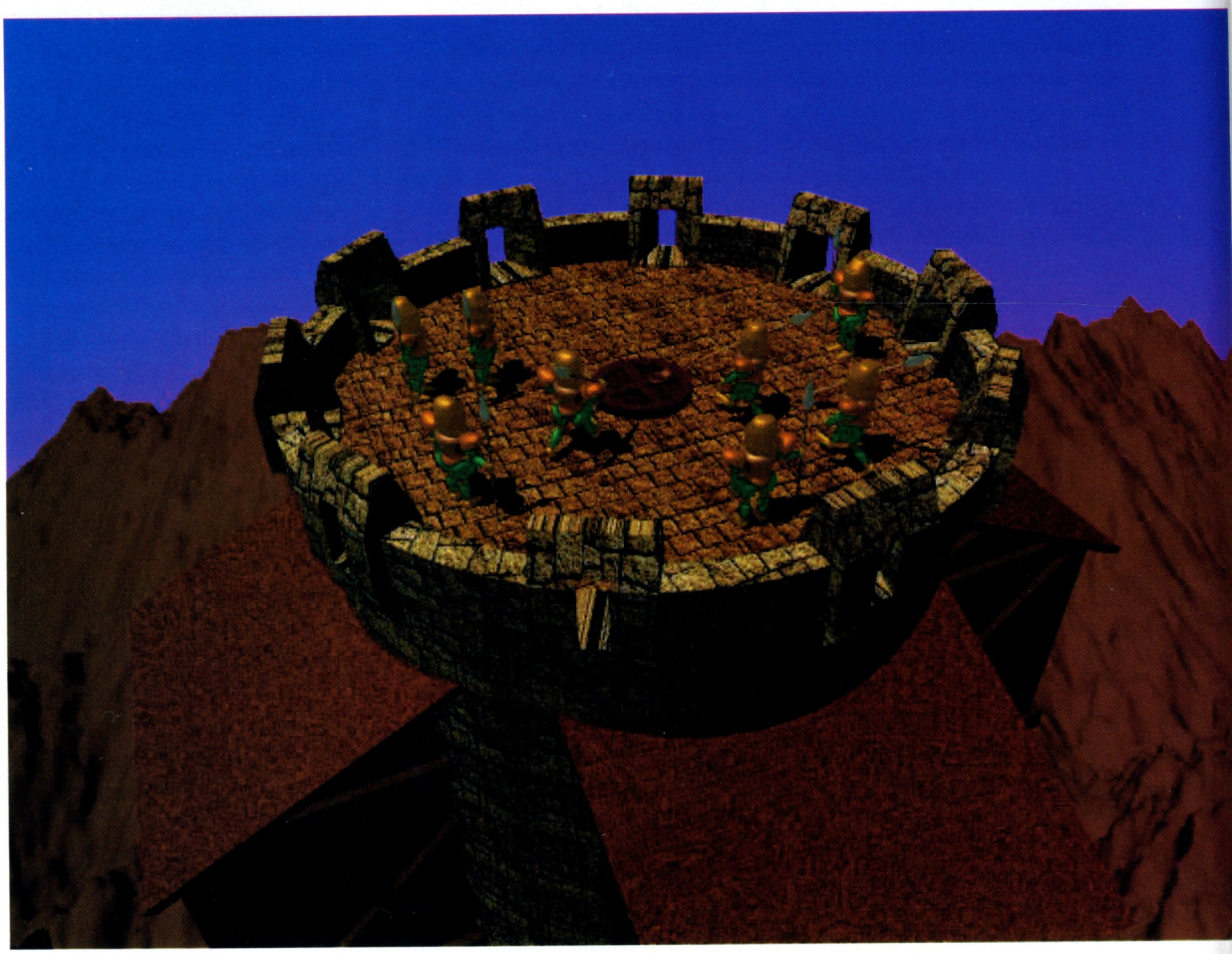
Para crear este castillo se estudió primero la altura que había de darse a cada torre para crear los grupos. Esto es necesario para impedir que algunas estructuras queden parcialmente encajonadas entre otras; hay algunas torres que en realidad quedan dentro de otras estructuras y no son visibles.

Una vez creados los dos grupos y las torres principales únicamente hubo que colocar referencias a estas estructuras en los puntos adecuados del cuerpo principal. En el resultado se aprecia una cierta falta de variedad, algo que podría haberse remediado en parte colocando algunas de las estructuras laterales de town.inc pero, a fin de ahorrar memoria, esto no se hizo.

Durante las pruebas necesarias para construir este ejemplo también se verificaron otros dos puntos. A saber, que resulta posible colocar casas (de town.inc) sobre estructuras de castle20.inc con un buen resultado y que no hubiera sido mala idea diseñar unas torres cuadradas de aspecto "rústico" empleando algunos de los paneles usados para las casas de town.inc.



Programación



Los modelos arquitectónicos suelen estar construidos por muchos elementos que se repiten con frecuencia. Situar manualmente en una escena cada uno de estos elementos podría ser algo muy tedioso, ya que por ejemplo un castillo puede constar de cientos de estos elementos. Afortunadamente POV 3.0 parece especialmente diseñado para crear escenas de este tipo.

orientada a escena

Estas nuevas sentencias de POV guardan un fuerte parecido con otras de nombre similar de lenguajes como BASIC o C. De hecho el funcionamiento de los comentarios en el lenguaje escénico de POV es idéntico al de C++ y los operadores utilizados para las sentencias #if y #while son también muy parecidos a los de C++. Es de esperar que estas nuevas pov-sentencias cambien enormemente la forma de trabajo de los usuarios de POV y por ello, a pesar de que ya hemos hablado sobre ellas en alguna que otra ocasión, estudiaremos hoy algunos casos prácticos de su uso en la creación de escenas.

Bucles

La nueva sentencia #while de POV 3.0 funciona exactamente como lo esperaría cualquier programador: haciendo que las líneas englobadas dentro del #while se repitan una y otra vez en un bucle mientras la condición del mismo resulte cierta. Veamos un ejemplo;

```
union{
  #declare nalm=0
  #while (nalm!=6)
      object{aspilleraGt translate<0,100,-31> rotate y*(nalm*60)}
      #declare nalm=nalm+1
  #end
  rotate y*30
  }
```

En este ejemplo, tomado de castle20.inc, se crea una unión de objetos usando #while. La condición del bucle está siempre enmarcada entre paréntesis, siguiendo al while, y las sentencias afectadas por el #while son las que se hallan entre dicha sentencia y la palabra #end. Así, en el ejemplo anterior, si la condición es cierta, Pov interpretará las dos líneas que forman el bucle. Después, al llegar a #end, POV regresará al #while que marca el extremo inicial del bucle y volverá a considerar la condición. Si ésta sigue siendo cierta, las sentencias del bucle volverán a interpretarse. De lo contrario POV abandonará el bucle saltando a la sentencia que sigue a #end (rotate y*30).

Para comprender esto, el usuario de POV debe recordar que la sentencia #declare permite asignar valores a variables y alterar dichos valores posteriormente. En el ejemplo, la ejecución del bucle depende del valor de la variable nalm. Este valor se inicializa a 0 antes de entrar en el bucle y va incrementándose en una unidad con cada pasada del mismo. Cuando el valor de la variable llega a 6, la condición es falsa y POV abandona el bucle. Ahora bien; ¿cómo se determina la verdad o falsedad de la condición del bucle? Pues de la misma manera que en C. O sea: el valor cero es falso y cualquier otro es verdadero. Aunque hay que precisar que valores sumamente pequeños y próximos a 0 pueden ser asumidos como falsos por POV.

Para determinar la veracidad o falsedad de las condiciones, POV dispone del siguiente juego de operadores relacionales:

(A<B) A es menor que B.
(A<=B) A es menor o igual que B.
(A=B) A es igual a B.
(A!=B) A es distinto de B.
(A>B) A es mayor que B.
(A>=B) A es mayor o igual que B.

También podemos emplear operadores lógicos para escribir condiciones complejas. POV dispone de…

(A&B) And La condición es cierta si A y B son verdaderos (con valores diferentes a cero).

(A | B) Or La condición es cierta si A o B o ambos son verdaderos.

Teniendo esto en cuenta, ahora podemos entender la condición del ejemplo anterior, a la que podemos traducir como "mientras nalm sea distinto de 6". (Por tanto mientras la variable no alcance dicho valor, el bucle se cumplirá). De esto deducimos que el bucle del ejemplo sirve para crear una unión de 6 objetos aspilleraGt. Además el valor de la variable empleada para controlar el número de pasos del bucle nos sirve también para preparar la orientación de cada objeto gracias a la sentencia "rotate y*(nalm*60)". Ahora veamos algunos ejemplos más de condiciones.

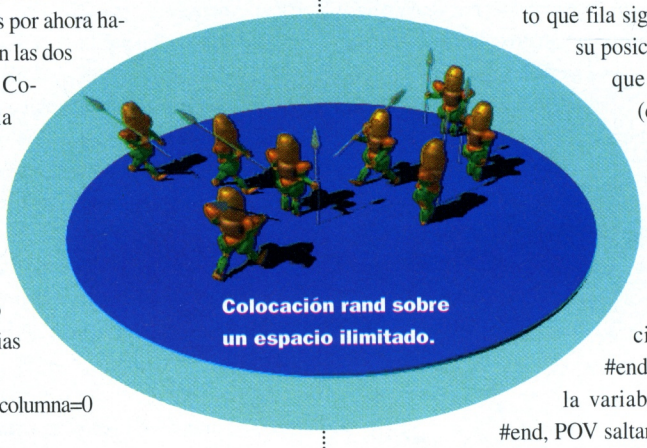
((A<B) & (A>=C)) Si A es menor que B y mayor o igual que C, la condición es cierta

(((A*500)=12500) | (A=C)) Si A * 500 es igual a 12500 ó A es igual a C, la condición es cierta.

Aparte de todo esto, POV emplea también operadores para trabajar con vectores y expresiones como "?" cuyo estudio dejaremos para otro día. Por ahora lo que sabemos ya resulta suficiente para empezar a hacer experimentos.

¡A mí la guardia!

Entre los ejemplos preparados para el presente número, el más interesante es el de la generación de ciudades medievales sin embargo comenzaremos por algo más sencillo: la colocación de grupos de lanceros virtuales. Ni que decir tiene que tendremos que ser bastante modestos en cuanto al número de personajes a emplear en cada escena (a causa del consumo de memoria). Además por ahora habremos de conformarnos con las dos únicas poses del lancero. Comenzaremos estudiando la escena del desfile de la sección "Cómo...". En esta escena un pelotón de lanceros circula por las calles de una ciudad. Dicho pelotón se crea, en el fichero desfile.pov, con las sentencias


Colocación rand sobre un espacio ilimitado.

```
#declare fila=0   #declare columna=0
#while (fila<4)
  #declare columna=0
  #while (columna<2)
      object{lananda rotate y*-90
             translate<fila*-20,0,columna*15>}
      #declare columna=columna+1
  #end
  #declare fila=fila+1
#end
```

Este bloque de sentencias utiliza un bucle #while anidado dentro de otro. Veamos cómo interpreta POV todo esto. Los povmaniacos con conocimientos de programación pueden saltarse tranquilamente la siguiente explicación. En primer lugar se inicializarán a cero las variables de control de ambos bucles y se comprobará la condición del #while más exterior. Al cumplirse ésta se pasará a la siguiente sentencia, otra #while, cuya condición también se cumplirá. Entonces se colocará un lancero andando (lananda) en las coordenadas X=fila*-20 y Z=columna*15. O sea en la posición <0,0> ya que las variables fila y columna están con valor cero al llegar a esta referencia al objeto lancero. Seguidamente se incrementa en uno la variable columna y el #end nos devuelve nuevamente al #while más interior. Aquí hay que recordar dos reglas. Primera: cada #while debe cerrarse con un #end. Segunda: en los bucles anidados el primer #end corresponde al último #while, el siguiente corresponde al siguiente #while más exterior y así sucesivamente. Pero sigamos con nuestro examen; la condición del while más interno volverá a examinarse y de nuevo se verificará su veracidad ya que la variable columna sigue siendo inferior a 2. Por ello POV colocará otro lancero cuya posición en X seguira siendo la misma, puesto que fila sigue valiendo 0, pero ahora su posición Z será igual a 15 dado que columna tiene el valor 1 (columna*15). La variable columna volverá a incrementarse con lo que la siguiente vez ya no se cumplirá la condición del bucle interno y POV continuará con las sentencias que siguen al primer #end. En ellas se incrementará la variable fila y, al encontrar el #end, POV saltara a la línea donde se halla el primer #while. Como la condición de dicho #while sigue siendo cierta se ejecutarán las sentencias comprendidas dentro del mismo. Así volverá a crearse un nuevo par de lanceros (esta vez en la posición X=-20) y nuevamente se incrementará fila y se reinicializará columna.

La ventaja que tiene este método anidado radica, por supuesto, en que podríamos crear un pelotón mucho más numeroso o cambiar la forma del mismo simplemente modificando las dos cifras de las condiciones de las sentencias #while. También resulta muy fácil cambiar las distancias entre las filas o las columnas de lanceros cambiando los valores arbitrarios que hemos colocado dentro de "translate" para multiplicar la columna y fila de cada lancero.

Guardando las torres

Utilizando provechosamente las sentencias translate y rotate, y teniendo presentes las dimensiones de nuestros personajes y el área donde deben ser colocados, podemos situar


Otra prueba aleatoria.

fácilmente a estos en disposiciones más complejas que un simple cuadro de lanceros desfilando. (Por supuesto un numeroso grupo de guerreros sería muy costoso en términos de memoria y por tanto de tiempo, pero el lector debe comprender que cuanto aquí se está exponiendo también puede ser aplicable a objetos menos gravosos). Podemos considerar muchos posibles ejemplos. Podríamos desear, por ejemplo, disponer un grupo de centinelas en una de las torres del castillo. Si dicha torre es redonda y tiene 20 metros de diámetro podríamos crear un grupo de 14 centinelas con las siguientes líneas

```
#declare nlan=0
#while (nlan<14)
  object{lananda translate<0,altura,-85>
        rotate y*(nlan*25.7)}
  #declare nlan=nlan+1
#end
```

Los 14 lanceros estarían todos situados cerca de las aspilleras de la torre a distancias equidistantes entre sí, y mirando hacia el exterior. Podriamos cambiar fácilmente el número o la colocación de los personajes pero hay un detalle que ya debe estar empezando a fastidiar al povmaniaco; la excesiva regularidad en las formaciones creadas hasta ahora. Afortunadamente POV 3.0 incorpora algunas nuevas características que nos permitirán solventar este problema. Entre ellas están las funciones para trabajar con vectores y con valores en flotante. Veamos algunas de ellas.

Rand(A) Retorna el próximo número pseudoaleatorio del generador de idem, atendiendo al entero positivo A dado como parámetro a la función. Previamente deberemos haber inicializado al generador con una llamada a la función seed(). Los números devueltos por rand() tienen valores comprendidos entre 0.0 y 1. Hay que recordar que estos valores no son verdaderamente aleatorios, sino que son el resultado de la aplicación de unas fórmulas. Esto quiere decir que para un código dado y una semilla elegida los valores devueltos serán siempre los mismos, lo cual es algo muy útil (luego veremos por qué).

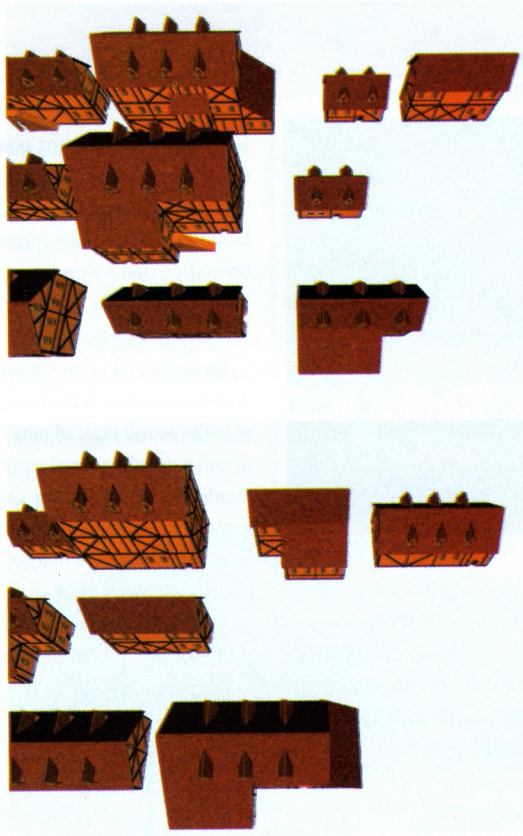
En cuanto a la función seed antes mencionada inicializa lo que el manual de POV llama un canal con un valor que se usará como semilla inicial para las fórmulas empleadas por el generador de números aleatorios. La función rand() posteriormente nos devolverá un valor del canal especificado como parámetro. Esto quiere decir que podemos abrir varios canales en el generador para obtener las mismas secuencias de numeros, lo cual puede tener su utilidad en la colocación de objetos.

Volviendo al ejemplo que nos interesaba, examinemos el siguiente trozo de código:

```
#declare R1=seed(717)
#declare numla=0
#while (numla<12)
  object{lanvigi translate<0, altura, rand(R1)*90>
        rotate y* rand(R1)*360}
  #declare numla=numla+1
#end
```

En estas líneas, primero inicializamos el generador con el canal R1 y después establecemos un bucle para situar 12 lanceros. El proceso para cada uno consistirá en obtener un valor pseudoaleatorio que oscile entre 0 y 90 para la coordenada Z del guerrero y luego otro valor del mismo tipo que oscile entre 0 y 360 para la rotación en grados del eje Y. De esta manera, aunque los lanceros seguirán orientados mirando hacia afuera, su colocación será más natural e irregular. (Recordemos que el lancero está con los pies en Y=0, centrado en este eje, y mirando hacia -Z). Para conseguir valores que oscilen entre 0 y 90 basta con multiplicar el valor devuelto por rand() por 90.

Si queremos que la pequeña guarnición de la torre tenga una disposición aún más irregular podemos empezar por colocar otra rotación aleatoria de 0 a 360 antes de efectuar el translate. De esta manera los lanceros mirarán en todas direcciones. Otra posible mejora puede consistir en emplear a rand() para decidir en cada pasada del bucle cuál de las dos posturas de lancero vamos a situar; si lananda o lanvigi. Para ello deberemos emplear una sentencia #if.



Dos matrices de casas creadas con un doble bucle y dos valores de semilla distintos.

Las sentencias #if y #else

La sentencia #if es muy similar en su funcionamiento a #while, aunque no implica bucle ninguno. Su estructura es:

```
#if (condición)
bloque de sentencias
#end
```

Cuando POV encuentra una sentencia #if comprueba la veracidad o falsedad de la condición que le acompaña entre paréntesis. Si esta es cierta, POV procede a interpretar el bloque de sentencias comprendidas entre el #if y la sentencia #end. De lo contrario salta a procesar las sentencias que siguen al #end. Una sentencia #if puede tener otras sentencias #if incluidas dentro de su bloque de sentencias. Así, en las siguientes líneas…

```
#if (A>B)
  #if (A+15=C)
    object{lananda translate<12,50,0>}
  #end
#declare c=c+1
#end
```

…el lancero se crearía sólo si las condiciones de ambos #ifs fueran ciertas. El segundo #if sólo se consideraría si se cumple la condición del primero. En dicho caso, la variable c se incrementaria aunque el segundo #if resultase falso ya que el #declare con la operación forma parte del bloque afectado por el primer #if. (Cuando hay muchos #ifs o #while, es probable que el autor del codigo acabe por perder de vista las sentencias condicionales que controlan cada trozo de codigo y por ello es aconsejable atenerse a un sangrado en el texto). Por último, una sentencia #if puede ir complementada con #else según el siguiente esquema:

```
#if (condición)
bloque de sentencias
#else
bloque de sentencias
#end
```

Aquí, si la condición es cierta POV procesará el primer bloque de sentencias y, al llegar al #else, saltará a procesar las sentencias que siguen al #end. Por el contrario, si la condición es falsa, se saltará directamente a procesar las sentencias que componen el segundo bloque (después de lo cual se continuará con las siguientes al #end).

Ahora ya estamos en condiciones de reanudar el estudio del ejemplo anterior. Queríamos usar la función rand() para decidir la colocación de una u otra postura del lancero para cada guerrero del grupo a crear. Para ello podríamos modificar el bloque de código del ejemplo de modo que quedase así:

```
#declare R1=seed(717)
#declare numla=0
#while (numla<12)
  #declare cual=rand(R1)*2
  #if(cual<1)
    object{lanvigi translate<0, altura, rand(R1)*90>
          rotate y* rand(R1)*360}
  #else
    object{lananda translate<0, altura, rand(R1)*90>
          rotate y* rand(R1)*360}
  #end
  #declare numla=numla+1
#end
```

Rand() dejaría en la variable "cual" un valor flotante oscilante entre 0 y 2, con lo que el asunto funcionaría. Ahora bien, imaginemos que disponemos de 4 posturas para el lancero. En dicho caso y obteniendo valores oscilantes entre 0 y 4 pa-

ra "cual", tendríamos que escribir un #if como el que sigue para representar la 3ª postura:

```
#if((cual>2) & (cual<3))
  object{postura3 ....}
#end
```

Esto es engorroso y por ello lo mejor que podemos hacer es truncar la parte decimal que no necesitamos. De este modo el #if anterior quedará como…

```
#if(cual=2)
object{postura3 ....}
#end
```

Para truncar la parte decimal de un número flotante emplearemos otra nueva función de POV 3.0; la función int(). En el ejemplo de las 4 posturas podríamos usarla así…

```
#declare cual=rand(R1)*4
#declare cual=int(cual)
```

…o bien así…

```
#declare cual=int(rand(R1)*4)
```

Para la primera escena de esta sección el grupo de centinelas se ha creado con un bucle while y usando rand de modo que la postura lanvigi tuviera una probabilidad de aparición mayor que lananda.

Al llegar a este punto seguro que más de un povmaniaco estará empezando a imaginar posibles aplicaciones para combinar rand() con #while.e #if. Casi cualquier idea es posible desde la creación de un bosque a la colocación aleatoria de una flota de naves espaciales o la disposición de un hormiguero. Sin embargo el empleo de rand() para colocar objetos tiene un pequeño problema.

Problema de rand y trucos para ahorrar tiempo

El problema está en que sencillamente no tenemos ninguna manera de saber si el espacio asignado a un nuevo objeto está ya ocupado o no. Si lo está es seguro que los objetos se superpondrán en mayor o menor medida. Aquí hay que mencionar que es una pena que los señores del pov-team no hayan contemplado la posibilidad de que el usuario pueda crear arrays. Si POV 3.0 permitiera esto, bastaría con ir guardando los datos de posición de cada personaje en un elemento del array y compararlos para cada nuevo lancero a crear.

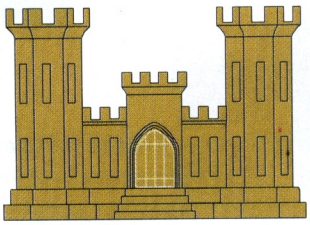
Sin embargo, puesto que esto no es posible, veamos algunos trucos para optimizar el uso de rand(). Cuanto más densamente poblada vaya a a estar el área donde pensamos diseminar los modelos más posibilidades habrá de que dos o mas de ellos aparezcan superpuestos en la escena. Como la generación de la misma puede llevarse algún tiempo (sobre todo si los modelos son complejos), lo mejor es sustituir a los objetos por cajas de color con las mismas dimensiones de los personajes que aparecerán en la escena final. También es conveniente eliminar al resto de los elementos de la escena y marcar el espacio donde los objetos deben diseminarse. En las pruebas para la escena de los lanceros de la torre, cada pose del lancero fue sustituida por una caja de un color y el área circular del piso donde estos iban a ir, se marcó con un cilindro de las mismas dimensiones del piso. De este modo las pruebas fueron bastante rápidas.

En posteriores pruebas para este ejemplo se comprobó que el sitio donde más tendían a amontonarse los lanceros era el centro de la torre. Por ello se decidió restringir el rango aleatorio para zpos con el fin de añadir un número fijo al valor devuelto por rand(). Con ello nos asegurabamos de que los lanceros nunca apareciesen en el centro de la torre.

Aún hay, además, otros métodos para rebajar las probabilidades de superposición e incluso de reducirlas a cero. Una posible manera sería dividir el área de aparición de los personajes en la torre a x anillos concéntricos y asegurarnos con un bucle de que sólo uno aparece en cada anillo. Como el ancho empleado para el anillo sería igual o superior al ancho del personaje no existiría posibilidad de superposición. Para concluir añadiremos que es posible diseñar otros trucos similares para áreas y distribuciones distintas de objetos.

Ciudades aleatorias

Al llegar a este punto el método usado para crear las ciudades medievales queda claro. Únicamente hay que crear un doble bucle anidado (como el de antes) para generar una matriz de nX*nZ casas. Usamos rand para decidir qué casa se va a dibujar en cada punto dado de la matriz y, para evitar la superposición, añadimos un valor a la posición según las dimensiones de cada casa. Este método, sin embargo, puede aún admitir muchas mejoras. Se podría cambiar la condición del bucle para que no tuviese en cuenta el número de casas de cada fila o columna, sino el largo de la calle. También se podría dar una distribución algo más aleatoria al trazado o emplear una utilidad para adaptar las alturas de cada casa a los valores de una malla montañosa o…



Castillos de

Desde su puesto, en una estratégica torre, dos solitarios guerreros de doradas armaduras otean sin descanso el horizonte. A sus pies, sobre un acantilado, un enorme castillo se alza frente a un mar infinito.

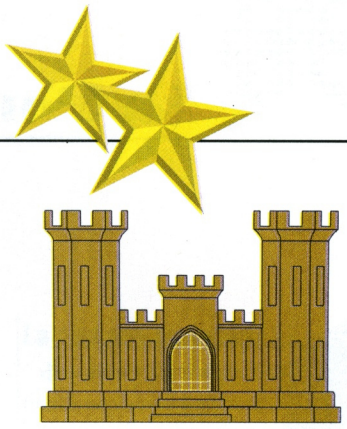


Esto es, al menos, lo que la portada de este número pretendía representar. Desgraciadamente, y por razones que ya se han explicado, no ha sido posible guarnecer debidamente los muros de nuestro castillo virtual, lo cual ha pesado bastante sobre el resultado que se buscaba. En efecto, utilizando las nuevas sentencias de POV 3.0 no hubiese resultado difícil situar a cientos de lanceros sobre torres y almenas... excepto por el hecho de que habríamos necesitado cientos de megabytes de Ram para evitar el continuo acceso de POV al disco duro.

Posibles portadas

Al crear una escena sintética el artista ha de materializar una posibilidad entre un número infinito de opciones. Este proceso es la primera dificultad con la que debemos lidiar. Para empezar no es raro que uno desee componer una escena a base de ideas o elementos que se excluyen mutuamente. Al cabo esto acaba rebelándose como algo imposible y no queda más remedio que empezar (¡ay!) a tachar ideas de la lista. Por ello, a fin de sopesar las ventajas e inconvenientes de cada idea, lo mejor es empezar por realizar unos cuantos bocetos en papel. Así, para la portada del presente número, se prepararon varios bocetos iniciales.

En el primer boceto el castillo se alzaba sobre una solitaria y empinadísima montaña a la que sólo se podía acceder siguiendo un largo camino serpenteante. Esta es una imagen clásica que se ha usado infinidad de veces para ilustrar cuentos igualmente clásicos pero que no por ello deja de resultar atractiva. En este dibujo el camino comenzaba casi a los pies de la cámara y se extendía dibujando numerosas curvas hasta llegar a las puertas del castillo situado casi al fondo de la imagen. El camino era estrecho, con profundos precipicios a ambos lados y la escena estaba dominada por las montañas. Como detalle adicional, el dibujo incluía unos soldados situados al principio del camino, de espaldas a la cámara y dirigiéndose al castillo. Esta idea presentaba diversos problemas. En el cuadro clásico el castillo queda demasiado lejos de la cámara -apenas es un borrón de tinta- con lo



cartón piedra



cual no resulta posible apreciar los detalles. Este problema se agrava si intentamos ser fieles a la escena clásica enfocando el castillo desde abajo.

Teniendo esto en cuenta se realizó un nuevo boceto en el que la cámara enfocaba al castillo desde arriba a una distancia muy inferior. Aquí todavía eran visibles algunas curvas del camino serpenteante pero ya no era posible tener en primer plano a los soldados. Se perdía el efecto del fondo montañoso y, por si esto fuera poco, se agravaban los problemas con el terreno (hablaremos de estos problemas después).

Desechadas las ideas iniciales se dibujaron nuevos bocetos. En uno de ellos se representaba la típica ciudadela medieval; la cámara enfocaba en primer plano la parte superior de los muros de una ciudad. Tras las almenas, algunos soldados paseaban sobre la muralla y más allá, rodeada por la muralla, se extendía una ciudadela en cuyo centro se alzaba un castillo. Esta idea tampoco duró mucho ya que nuevamente el castillo quedaba demasiado alejado de la cámara y además ya se había usado una ciudad como portada del número 0 de Rendermanía.

El siguiente boceto representaba algunos de los detalles de un castillo gigantesco. Toda la escena estaba llena de conjuntos de torres y murallas y no se veía suelo, ni cielo. Esta escena, que hubiera costado cara en memoria por su excesivo número de componentes, fue modificada lentamente hasta llegar al boceto final. En éste no se ve un castillo completo pero la cámara está lo suficientemente alejada como para darnos una idea general de la estructura. Hay una torre centinela que sirve para colocar dos soldados en un punto donde no restan espacio al protagonista del cuadro, el castillo. En cuanto a los problemas con el terreno se han minimizado colocando la cámara en un punto desde el que es imposible ver la unión del castillo con el suelo. La textura del Heightfield es francamente mala (de hecho se reduce a un simple pigmento marrón) pero contrasta con el color medio de los bitmaps del castillo y con el color del agua empleada con el mismo fin (además se calcula rápidamente).

Nuevos objetos para la portada

Al optar por desarrollar la idea anterior se comenzó por hacer pruebas con el mismo castillo empleado como ejemplo en el fichero castle20.inc. Los resultados, no obstante, eran poco atractivos debido a la excesiva uniformidad de las estructuras empleadas. En principio se pensó en escribir nuevas grupos de torres para obtener una mayor variedad pero casi enseguida quedó claro que el mayor problema no estaba en la poca variación de estas estructuras, ni en el cambio de sus dimensiones, sino en la excesiva monotonía de la construcción resultante. Por esta razón se decidió construir un nuevo tipo de torre cuadrada. Esta torre, de aspecto algo rústico, se construye con los mismos paneles y objetos empleados para el montaje de

las casas. La única diferencia estriba en que los nuevos objetos son mucho más altos que anchos o largos y en que el tejado ha sido escalado. Estas torres se crean por secciones -como sucede con las demás-. La diferencia está en que aquí todas las secciones tienen 5 metros de alto y en que debe colocarse un tejado al final. Hay dos clases de torres de este tipo: las de 5 metros de ancho y las de 10 metros. Las primeras pueden colocarse sobre las torres convencionales para rematarlas o pueden ser usadas como torres colgantes. En cuanto a las torres de 10 metros de ancho, es fácil emplearlas para diseñar estructuras complejas (por ejemplo añadiéndoles torres laterales adosadas, como en la portada). Estas nuevas estructuras añaden un adecuado constraste con las torres primitivas y son, además, fáciles de construir.

Problemas

La falta de variación del castillo original se remedió parcialmente añadiendo algunas de estas nuevas estructuras e incluyendo además estructuras laterales y colgantes en distintos puntos de las murallas y torres. El siguiente factor a considerar era el gasto de memoria. Para evitar un gasto inútil, y dado que no iban a emplearse casas sino tan sólo estructuras laterales y colgantes, se activo el include de "lateral.inc" en vez del correspondiente a "town.inc". (Recordemos que lateral.inc es un subconjunto de town.inc creado únicamente para ahorrar memoria). También se eliminaron muchos de los elementos del castillo que no iban a aparecer en la escena final (aunque no todos ya que no se conocía con precisión la distancia final a que iba a dejarse la cámara). Esto implica que el castillo no es en realidad una construcción completa sino un decorado de cartón piedra; algunas torres y estructuras no visibles fueron eliminadas, el muro que delimita el castillo sólo existe en las zonas visibles de la escena y hasta la torre donde van colocados los lanceros no es más que una sección final que está flotando en el aire. (Todo esto recuerda a una antigua novela de Dick llamada Ubik. En ella los humanos vagaban por un escenario virtual que abarcaba únicamente lo que podía alcanzar su capacidad de percepción. ¿Por qué malgastar memoria con objetos no visibles?). Naturalmente no podríamos crear una animación con este universo virtual incompleto pero si no pretendemos esto y no vamos a mover demasiado la cámara... (Sin embargo, y a pesar de todo lo dicho, la cantidad de memoria requerida para generar nuestra portada es de algo más de 75 megabytes).



El siguiente problema a considerar es el del terreno. Existen muchas maneras de generar terrenos sobre los que colocar nuestras construcciones virtuales. Podríamos, por ejemplo, emplear tools que generan ficheros con mallas de paisajes fractales. (Algunas de estas herramientas, como Frgen de Steve Anger, generan archivos directamente legibles por POV. En otros casos habrá que utilizar herramientas de conversión de formatos). Sin embargo la mayoría de estos programas no nos permiten tener ningún control sobre la forma final de la malla. Este detalle puede suponer un verdadero problema ya que la colocación de un castillo (o cualquier otro objeto) sobre un punto dado de una fractal requiere entonces un proceso de posicionamiento por tanteo (ya que no conocemos la altura del terreno en un punto dado y también puede ocurrir que la malla no disponga de una meseta donde colocar nuestro objeto). El problema es aún mayor si queremos que parte del terreno se adapte a cierta forma (por ejemplo al camino retorcido del primer boceto antes comentado). Afortunadamente estos problemas pueden ser soslayados empleando algun programa de dibujo 2D para "dibujar" el terreno sobre el que irá la construcción.

Dibujando terrenos

En esencia el proceso necesario para hacer esto es sencillo. Usando una herramienta de dibujo y procesado de imagen 2D -como por ejemplo Photoshop- dibujaremos una imagen empleando una paleta de grises. En esta imagen los pixels negros corresponderán a los vértices de menor altura y los de blanco a los de mayor altura en la malla a generar. Para crear el bitmap usaremos las herramientas de que disponga el programa que estemos usando; un generador de fractal de plasma (para crear la imagen inicial sobre la que vamos a trabajar), un "aerógrafo o un lápiz para crear las "mesetas" donde irán los objetos y los caminos para llegar a dichas mesetas, etc. También podemos partir de una imagen dada a la que retocaremos. Una vez que tengamos la imagen tga, gif o png, la incluiremos como fuente para una sentencia height_field de POV. Veremos



Todas las pruebas intermedias se tiran sin texturas para agilizar el render.

entonces como las zonas con el blanco más intenso generan las zonas más altas de la malla.

Para el primer boceto se preparó un bitmap con una única montaña rematada por una gran meseta en la que se pensaba colocar el castillo. De dicha meseta (dibujada en el bitmap con el color blanco más intenso) partía una serpenteante línea blanca dibujada con "aerógrafo" de tal modo que sus bordes quedaban difuminados. Esta línea era, por supuesto, el estrecho camino que se pretendía representar. Como los objetos height_field se extienden inicialmente sobre un área que abarca desde el punto <0,0,0> hasta el valor 1 de todos los ejes, resulta posible determinar el punto exacto del objeto donde está el centro de la meseta en el bitmap. Para ello lo ideal es centrar el height-field en los ejes X y Z. Así, al escalar el objeto, seguiremos sabiendo dónde está la meseta. (Basta con conocer la resolución del bitmap y el valor X e Y donde se halla el centro de la meseta dentro de dicho bitmap).

Nuestro posible control sobre la malla montañosa resultante no es perfecto, aunque es fácil conseguir el efecto buscado de una manera aproximada. Para nuestro boceto, por ejemplo, se deseaba que el camino serpenteante no tuviese la misma altura en todos sus puntos y por ello se retocó la línea serpenteante de modo que algunas zonas tuviesen distintos tonos de gris. El resultado, no obstante, no fue muy alentador ya que era difícil evitar que el camino obtenido en la malla presentase cambios de alturas abruptos. (Recordemos que, aunque POV puede representar 65536 niveles de altitud, una gama de grises dibujada con una utilidad de dibujo sólo nos permitirá 256 tonos de gris con los que, por tanto, sólo podremos obtener 256 alturas diferentes desde POV).



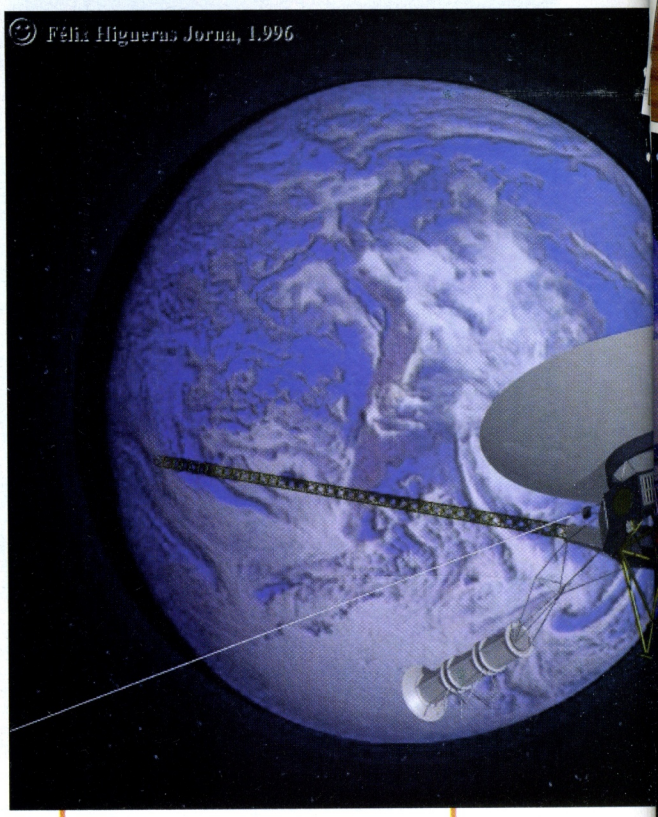
La portada escogida

Para la imagen final, sin embargo, la malla montañosa no era estrictamente precisa puesto que la cámara no abarca al castillo completo y el único lado donde podría verse tierra queda oculto tras un muro más alla del cual comienza nuestro mar virtual. Sin embargo -y por puro capricho- se decidió crear una malla para dar algo más de realismo a la costa sobre la que se alza el castillo. Esta malla se creó empleando un método de POV bastante original (y que el autor de estas líneas descubrió en el manual de la nueva versión 3.0). En dicho método POV se usa en primer lugar para construir un bitmap y luego, en la escena propiamente dicha, se da como entrada este bitmap al heightfield. Aquí la única novedad radica en el primer paso, en la forma en que POV es usado para generar dicho bitmap: con la sentencia global_settings (usada para indicar parámetros por defecto para muchas cosas distintas) se emplea la palabra hf_gray 16 para indicar que el fichero de salida usará un formato de 16 bits en escala de grises digerible por la sentencia height_field de POV (lo que nos dará 65536 posibles valores de alturas). Luego se crea una cámara que enfoca un plano sobre el que va una textura que normalmente usa el patrón wrinkles (originalmente creado para generar apariencias de celofan o telas transparentes). El resultado puede verse en la malla parcialmente sumergida de la escena.

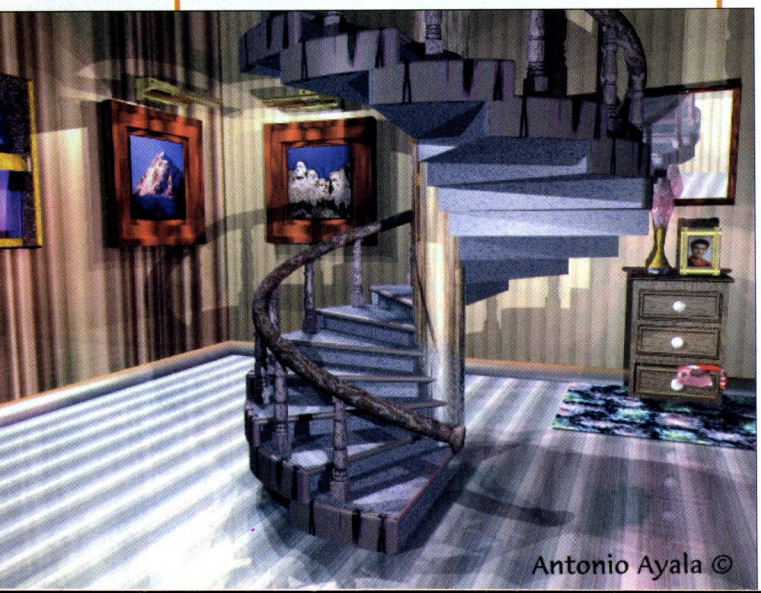
En cuanto al mar, a fin de darle mayor luminosidad se creó una esfera celeste (aunque el cielo en si no es visible). Las texturas se retocaron para dar mayor contraste a la portada y se generó la imagen a la resolución adecuada, lo que costó días al ordenador y sudores al usuario.

Galería de Artistas

Aquí tenéis una muestra de la paciencia, habilidad e imaginación que derrochan algunos autores que nos remiten sus trabajos. Como siempre en el CD recopilamos las obras de todos ellos y de los que por falta de espacio, que no de talento, no pueden estar aquí.



Comenzamos hablando de escaleras, **Antonio Ayala** nos envía esta de caracol y con barandilla hecha con Moray (leed su carta en el foro). Amigo Antonio, espero que estés satisfecho con la ampliación de Rendermanía (de la cual tú, con tus cartas eres uno de los culpables). Sin embargo aunque parezca increíble no tenemos hoy espacio para contestar a las cuestiones que planteas y que tendrán que esperar a algún número próximo. Perdona y gracias por tus cartas pidiendo la ampliación de Rendermanía. (Y que hicieron que me recordases a aquel romano que siempre acababa sus discursos diciendo "…Y hay que destruir Cartago").



¿qué decir de las magníficas escenas espaciales de **Félix Higueras Jorna**? En ellas Felix reproduce el vuelo de la sonda espacial Voyager 2 por el sistema Solar. La sonda construida a partir de un dibujo de una revista es de un gran realismo y está íntegramente creada con POV 3.0 (#while). Hay que hacer notar que los instrumentos de la sonda están articulados (#if, rotate) y encuentro verdaderamente admirable la escena de las nebulosas donde Félix ha empleado magistralmente los nuevos halos de POV 3.0. También hay que destacar otras escenas igualmente extraordinarias; unas latas de refresco y otras donde se mezcla sutilmente la imagen sintética de POV con fondos reales. En una escala de 0 a 10 anótate un 14. En resumen: ha nacido un nuevo POV-maestro. Las mejores pov-escenas que he visto en mucho, mucho tiempo.





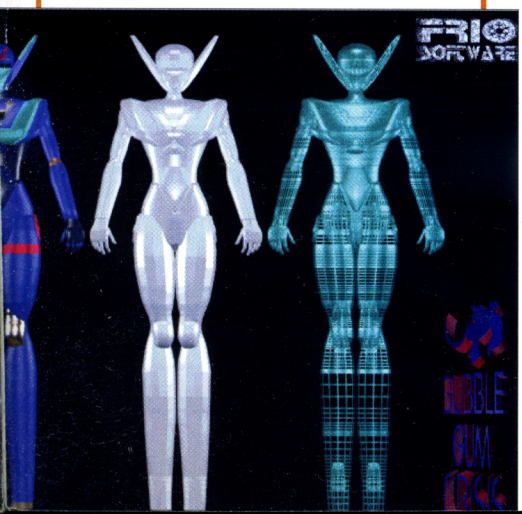




Desde Oviedo **Carlos Fernández Alonso**, nos envía una animación… de Mazinger Z. Carlos, a quien le parece increíble que Alberto Sendin se le adelantara en la idea, envía los archivos de generación de su animación y solicita una critica. Pues bien la animación -al igual que el Mazinger- está bastante bien, aunque debo confesar que me gustó más el trabajo de Alberto por dos razones: la primera porque su animación resultaba más dinámica que la tuya (la cámara se movía mejor) y por otro lado, Alberto modeló la base de Mazinger y eso siempre tiene más trabajo que incluir un bitmap detrás (aunque has tenido el acierto de alejar un poco el bitmap del fondo para que se aprecie algo de profundidad). También es una pena que, ya que te molestaste en articular los puños, esto no se aprecie en la animación. De todos modos, bien y adelante con tus siguientes animaciones.

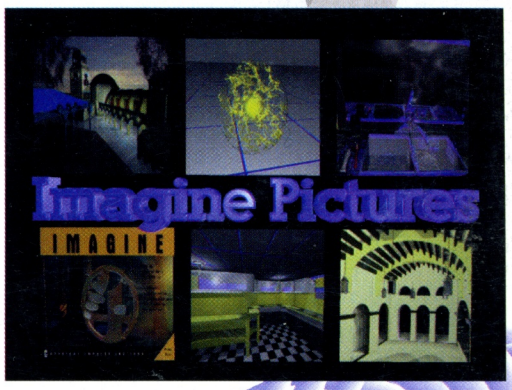


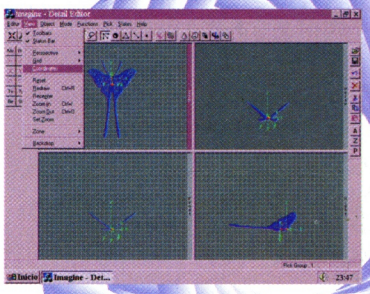
Nuestro amigo **Juan Miguel González Cortinas** de Frio Software nos envía unas escenas construidas con modelos de 3D Studio que ha preparado basándose en los caballeros sable de Bubble Gum Crisis, un popular manga del que tan sólo conozco "Demencia Mortal" de Adam Warren. Las chicas de Juan Miguel, aunque virtuales, son atractivas y están bien armadas. ¡Ah, y echad un vistazo a su carta en el foro! En Frio Soft se buscan nuevos miembros.



David Belinchón Domínguez nos ha enviado algunas de las escenas que pudo rescatar del desastre acaecido en su disco duro. Su escena principal no incluye archivos de generación pero la largísima explicación del proceso de modelado que adjunta en su carta demuestra que la escena es suya. Así pues, por esta vez, incluyo aquí esta escena basada en la escalera de la Biblioteca Laurentina de la iglesia de San Lorenzo, en Florencia. Esta magnífica escalera virtual de David sólo difiere del original de Miguel Angel en el número de escalones. David envía sus felicitaciones a Socrates y sugiere la creación de un concurso de infografía arquitectonica. Respecto a esto…¿Qué dicen los rendermaniacos?

Imagine 4.0



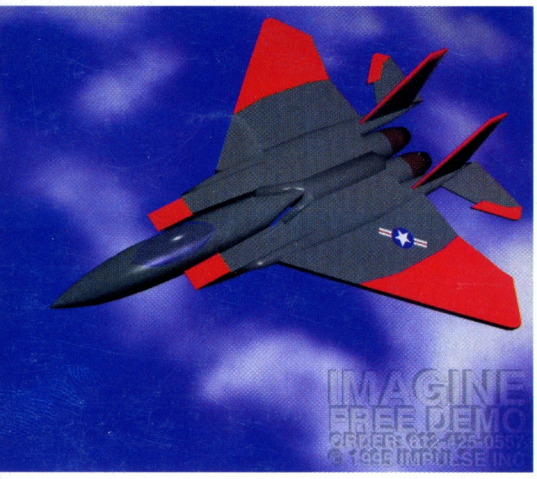


Hace meses hablamos de un programa procedente del entorno Amiga. Se trataba de Imagine 2.0, un programa con el que resulta posible modelar, renderizar y animar complejos modelos 3d. Hoy incluimos en el CD las demos de la versión 4.0 (para DOS y Windows 95) de este magnífico programa.

Imagine 4.0 presenta diferentes módulos, cada uno de los cuales cumple una función concreta. Nos ofrece un editor de formas, con el que resulta posible crear formas complejas utilizando lo que el programa llama secciones cruzadas, un editor de detalles, con el que podremos depurar los objetos creados en el módulo anterior utilizando una serie de poderosas funciones tales como magnetismo, deformaciones varias (taper, bend, twist y otras), extrusiones, superficies de revolución, particulas, blobs, etc. Desde este potente módulo podremos trabajar sobre el objeto como tal o sobre sus caras o sus vértices.

Otros módulos son el de preferencias, usado para customizar el funcionamiento de Imagine, el editor de splines, usado para crear objetos bidimensionales, el editor de ciclos, empleado para trabajar con animaciones, el editor de "stages", usado para potenciar la animación y el editor de acciones usado conjuntamente con el anterior.

Naturalmente, y dado que Imagine es un programa comercial, lo que hoy incluimos son demos utilizables de este programa. Esto quiere decir que las prestaciones de estas demos han sido recortadas lo suficiente como para impedir un uso comercial de las mismas, aunque sí podremos trastear con los programas y tomar nota de sus características.La versión de DOS incluye el editor de detalles mientras que la versión de Windows 95 incluye el editor de detalles, el de splines y el de stages. Por contra la versión de DOS no tiene las limitaciones de render de su hermana de Windows, la cual está restringida a 320*240 pixels (aunque, eso sí, todas las escenas grabadas llevarán el nombre del programa y de la compañía). Con los programas vienen objetos de ejemplo que nos servirán para probar las funciones disponibles. ¡Qué lo disfrutéis!



NOTAS IMPORTANTE:

1) Los bitmaps con los que han sido generadas la mayoría de las imagenes no son de libre distribución. Por ello el lector habrá de sustituir las texturas que hay al principio de castle20.inc por otros bitmaps, por texturas procedurales o por pigmentos simples.

2) Si alguien se atreve a intentar renderizar la portada, que tome nota de que antes hay que renderizar la malla para la tierra de la misma (echad un vistazo a portada.bat).

ADVERTENCIA: Algunas de estas escenas (sobre todo la portada) requieren una gran cantidad de memoria y son de muy lenta generación.

DISCULPA: En el número 0 del magazine Rendermania no fue publicado el fichero town.inc en el CD-ROM. Para renderizar las escenas de ese número incluimos la antigua versión de dicho fichero en el directorio "anticua".