Manual del usuario de la librería

POVRayWriter

1. Introducción

1.1. ¿Qué es POV-Ray?

POV-Ray es un software trazador de rayos; es capaz de generar imágenes solo a partir de la posición y las propiedades de los objetos y las fuentes de luz, dando imágenes 3D de gran calidad. Para dibujar las imágenes, se utiliza un código de programación bastante sencillo, por el cual se indican las propiedades de cada uno de los objetos. Además, soporta estructuras de vectores, bucles y decisiones lógicas, lo cual permite realizar programas que dan diferentes resultados dependiendo de las condiciones iniciales.

También se pueden realizar animaciones, mediante el uso de la variable *clock*. Esta va aumentando su valor dependiendo del número de fotogramas y los valores mínimo y máximo estipulado para ella. En la animación, simplemente se redibuja la pantalla, cambiando el valor de clock.

Hay que hacer notar que POV-Ray no es un programa de diseño 3D, como *Blender* o *SketchUp*. En estos, los elementos se presentan de forma dinámica en la pantalla: el usuario puede estirarlos, moverlos, pegarlos, etc. para luego realizar su animación o imagen. En el caso de la animación, normalmente incluso es posible ir moviendo la imagen fotograma a fotograma para obtener el resultado deseado. En POV-Ray, eso no es posible. Para modificar una imagen, es necesario cambiar el código. Igual pasa con las animaciones. La compilación del código y su transformación en imagen se llama renderizado (del inglés, *render*), y por cada imagen, puede llevar desde un momento a varios segundos, o minutos, dependiendo de su complejidad. Las animaciones han de renderizarse fotograma a fotograma de modo que, si una sola de nuestras imágenes tarda 30 s, una animación de 1 segundo (24 fotogramas) tardará 12 minutos en renderizarse. Por tanto, es necesario buscar un equilibrio aceptable entre calidad – tiempo. Una imagen media, con dos o tres objetos, una luz, y un plano, suele tardar unos 2 segundos en renderizarse (lo que hace un total de 48 s segundos por segundo de animación).

1.2. POV-Ray para para cálculo científico

A menudo, es útil en una simulación por ordenador visualizar resultados. La forma más directa de hacerlo es mediante la librería gráfica del lenguaje en cuestión, pero generalmente en cálculo intensivo esto no es buena idea. El motivo es que mostrar un objeto en pantalla (un círculo, o una línea) necesita bastantes recursos informáticos. Si estamos calculando, por ejemplo, diferentes valores de posición para varios cuerpos mediante la resolución de sistemas de ecuaciones o ecuaciones diferenciales complicadas, no estaría mal obtener 1000 valores, por cada segundo, para obtener una gran precisión. Sin embargo, esto ya supone tener que mostrar, para la librería gráfica, 1000 fotogramas. Un rendimiento aceptable para una librería de este estilo es de unos 40 a 100 fps (estamos suponiendo gran cantidad de cálculos entre fotograma y fotograma). Eso quiere decir, que, en el mejor de los casos, tardará 10 segundos en computar esos 1000 valores, y en el peor, 50 segundos. Si prescindiéramos de los resultados gráficos, es casi seguro que llevaría menos de tres o cuatro segundos realizar ese número de iteraciones. Además, la mayoría de estas librerías trabajan exclusivamente en dos dimensiones.

La otra opción es exportar los resultados a un fichero de texto y representarlos con un programa especializado. Por ejemplo ,el software VMD representa moléculas y forma animaciones con ellas a

partir de unos datos importados en fichero de texto. En mecánica también existen distintos simuladores especializados. Para propósito general, también se suele usar el lenguaje de programación gráfica OpenGL. Estos programas nos dan más calidad de imagen, en tres dimensiones. El único problema va a ser el tiempo necesario para renderizar, pero podemos hacerlo por partes. Es decir, puedo poner a mi ordenador a renderizar una hora, obtener 300 imágenes para mi animación y seguir al día siguiente, o bien formar una animación cogiendo solo 1 de cada 5 fotogramas. Si bien el movimiento no será continuo, se podrá ver la evolución del proceso, que la mayoría de las veces es lo único que nos interesa en una simulación.

1.3. POV-Ray y C++

El programa POV-Ray funciona con archivos de código, como ya hemos dicho, en formato .pov. Esto es una ventaja, porque para representar los resultados de una simulación, podemos usar la clase *ofstream* de C++ para escribir un fichero .pov que nos represente nuestros resultados. Si bien esto no es muy complicado, puede ser un engorro escribir en C++ toda la arquitectura necesaria para un fichero de POV-Ray. Por ejemplo, si quiero crear una bola en POV-Ray utilizo el código:

```
sphere {
<1,2.5,-3> 2
pigment { color Green }
finish { phong 0.7 }
}
```

Como se puede ver, es bastante código sólo para una esfera, que además no tiene demasiadas propiedades. Crear archivos de texto con estos formatos puede suponer cierta pérdida de tiempo a un programador, en un detalle que es simplemente técnico. Y eso, además, considerando que el programador en cuestión conoce la sintaxis propia de POV-Ray. De lo contrario, tendría que aprenderla antes.

Este es el motivo de la creación de la biblioteca POV-RayWriter, dar una clase de C++ que permita escribir archivos **.pov** representables por POV-Ray sin tener que aprender a manejar de cero la aplicación y con mayor rapidez para incluir elementos básicos en el código.

2. Dibujando formas.

2.1. Primeros pasos

Lo primero que tenemos que hacer es incluir la librería en nuestro código. Para ello copiamos nuestro archivo POVRayWriter.cpp al directorio donde se encuetra nuestra código fuente. Ya en el código, realizaremos la siguiente directiva include:

```
#include “POVRayWriter.cpp”
```

Esto ya incluye todos los elementos que necesitamos para trabajar. Después, tenemos que inicializar un objeto de tipo POVRayWriter. Eso se hace mediante la inicialización, y, entre paréntesis, la ruta del archivo con la extensión correspondiente. No es necesario que el archivo exista.

```
POVRayWriter pov ("C:\\Users\\Victor\\Ejemplo.pov");
```

Es **muy importante**, una vez hemos acabado de utilizar nuestro archivo, **cerrarlo**. Si no se cierra podrían producirse problemas. El cierre lo haremos mediante la función:

```
pov.closePOVRayWriter();
```

Entre ambas sentencias, debemos de colocar todas nuestras opciones de dibujo.

2.2. Dibujo básico. Objetos estándar

Lo siguiente es comenzar a dibujar objetos. Lo primero es que la clase contiene múltiples funciones para dibujar el mismo objeto, en función de unos parámetros u otros. Aquí vamos a ver algunos de los parámetros básico y el dibujo de los que la clase se utilizan como "objetos estándar", es decir, objetos con un formato predeterminado.

Lo primero es dibujar la cámara y la luz. Un foco de luz para nuestro código se escribiría como

```
pov.createLight(double x, double y, double z);
```

Que creará una luz de color estándar. Si queremos que la luz ilumine en cierto color dado en RGB, debemos crear la luz de esta manera

```
pov.createLight(double x, double y, double z, double r, double g, double b);
```

Donde las x, y, z son las coordenadas del punto y r, g, b, los valores de rojo, verde y azul respectivamente. Una vez tenemos luz (sin la cual es imposible que sea vea algo) debemos de crear la cámara. La cámara es el observador. Debemos de indicarle dónde está y hacia dónde está mirando. Una cámara básica se inicia así:

```
pov.createCamera(x1,y1,z1,x2,y2,z2);
```

Donde x1, y1, z1 son las coordenadas de posición de la cámara y x2, y2, z1, son las coordenadas del punto al que mira.

Lo siguiente es añadir formas básicas, como cubos, cilindros o esferas. Vamos a comenzar por las esferas. La primitiva de dibujo básica para una esfera es:

```
pov.createSphere(double x, double y, double z, double r);
```

Donde las números x, y, z representan las coordenadas del centro de la esfera, y r, su radio.Esto creará el siguiente código en POV-Ray:

```
sphere {
<x,y,z>  r
}
```

Como podemos ver, es un código absolutamente básico. Si POV-Ray representa este código, la esfera se verá como un círculo negro, al carecer de color. Tampoco tendrá efecto de ambiente, ni reflejo; simplemente será una bola negra y oscura del radio indicado. Para arreglar esto utilizamos el objeto estándar de esfera. Es exactamente igual, solo que le podemos añadir un color y vendrá con un valor de reflejo predeterminado. Se inicializa así:

```
pov.createStandardSphere(double x, double y, double z, double r, string color);
```

Es igual que antes, pero se puede ver que contiene una cadena de caracteres para almacenar color, que puede ser "Red", "Blue", etc. Se puede consultar la lista de colores predefinidos existentes en la documentación POV-Ray.

En la próxima sección explicaremos la utilidad de la "bola negra", que no es otra que dar una bola sin propiedades para poder personalizarla con todos los parámetros que uno quiera. La bola estándar no se puede personalizar, pero es una forma rápida de crear un objeto con una apariencia agradable.

Ahora ya sabemos crear una esfera. No estaría mal si la colocamos encima de un plano. El plano podemos inicializarlo de dos formas. La primera, mediante un vector normal a su superficie. Esto se hace mediante la siguiente función:

```
pov.createStandardPlane(double n1, double n2, double n3, double d, string c1, string
c2);
```

Los tres primeros argumentos son las componentes del vector perpendicular al plano, mientras que d es la distancia desde el origen a este vector. Generalmente, d=0. En su versión estándar, el plano está formado por cuadritos de colores, para poder diferenciarlo del cielo. Esos colores son c1 y c2. Normalmente, sin embargo, lo que queremos son planos verticales, uno que haga de suelo, o una combinación simple de ellos. Por ello, se puede escribir el plano de esta otra forma,

```
pov.createStandardPlane(string plane, double d, string c1, string c2);
```

Donde *plane* va a ser "x", "y", "z" o cualquier combinación como "3*y" o "x+y". El plano z=0, por ejemplo, sería:

```
pov.createStandardPlane("z", 0, "Red", "Blue");
```

Así, por ejemplo, con lo que ya sabemos ya podemos crear una escena simple. Un programa que crea una escena con una esfera sobre un plano podría ser este:

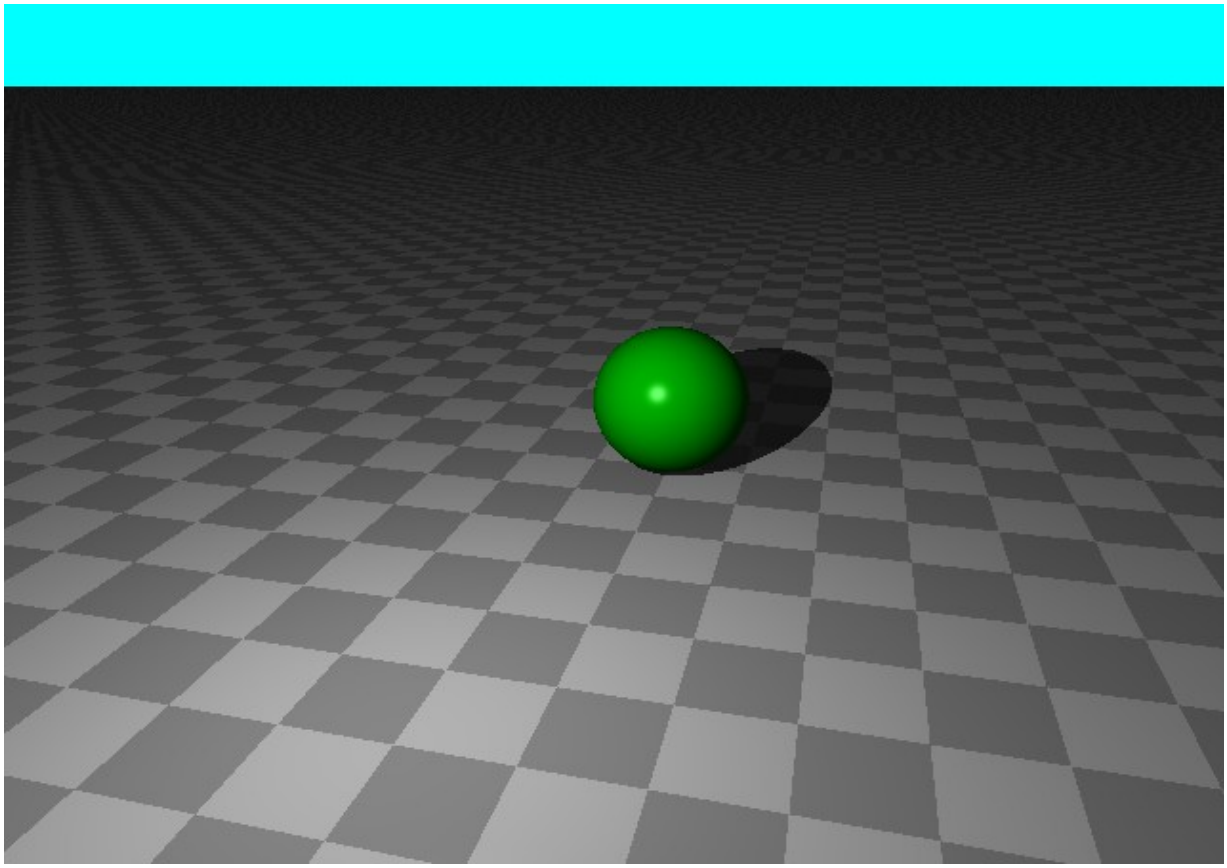
```
int main(void)
{
POVRayWriter pov ("C:\\Users\\Victor\\output.pov");

pov.createLight(-5,0,4,1,1,1);
pov.createCamera(-10,3,5,2,0,0);

pov.createStandardPlane("z",0,"Grey","White");
```

```
pov.createStandardSphere(1, 1,1,1, "Green");

pov.closePOVRayWriter();
return 0;
}
```

Este código nos genera un código de POV-Ray como este:

```
// This file have been generated with
// Victor Buendia's POVRAY WRITER
#include "colors.inc"

background { color Cyan }

light_source {
<-5, 0, 4>
color rgb <1, 1, 1>
}

camera {
location <-10, 3, 5>
look_at <2, 0, 0>
sky <0,0,1>
}

plane {
z  0
finish { phong 0.5  }
pigment { checker color Grey color White  }
}

sphere {
<1, 1, 1> 1
finish { phong 0.5  }
pigment { color Green  }
}
```

Crear este código pudiera haber requerido consultar más de una vez la documentación de POV-Ray si la sintaxis no es conocida. Y aun conocida, nuestro código en C++ contiene exactamente 6 líneas, mientras que el de POV-Ray contiene más del doble. Si hubiéramos intentado escribir este código mediante una clase *ofstream*, habríamos tardado una cantidad considerable de tiempo, frente a los pocos segundos necesarios para crear una imagen como esta:



2.3. Personalizando objetos básicos

Volvamos ahora a nuestra bola negra, vacía, básica, esa que dejamos de lado en la sección anterior por los objetos estándar:

```
pov.createSphere(double x, double y, double z, double r);
```

Como habíamos dicho antes, esta esfera no es útil de esta forma. No tiene color ni propiedades y aparece como una bola oscura en la pantalla. Esta va a ser nuestra materia prima para añadir propiedades. Las propiedades son de dos tipos: ***finish***, ***texture*** y ***pigment***.

Finish se ocupa del acabado final de la bola, como el reflejo metálico, representar una superficie más real mediante rugosidades o bien dar el reflejo ambiente. Estas tres propiedades se conocen en POV-Ray con el nombre de *phong*, *diffuse*, y *ambient*, respectivamente.

Para añadir estos efectos, vamos a utilizar la función *addFinishToNextObject*, cuya sintaxis es:

```
pov.addFinishToNextObject(double phong, double ambient, double diffuse);
```

Si no queremos añadir las tres propiedades a la cláusula, simplemente le daremos un valor -1. Esta sentencia añade el acabado **al siguiente objeto creado.** Esto es muy importante, ya que no permitirá que todos los objetos aparezcan por defecto, sino solo el siguiente. Por, ejemplo, podemos escribir el siguiente código:

```
pov.addFinishToNextObject(0,75, -1, 0,2);
pov.createSphere(1, 0, 1, 0,5);
```

Que nos creará una esfera en la posición (1,0,1), con un radio de 0.5, un reflejo (*phong*) de 0.75, y un nivel de reflexión difusa (*diffuse*) del 0.2. Como hemos puesto el ambiente con el valor -1, la esfera no tendrá ningún valor especial de esta propiedad.

La siguiente propiedad es ***pigment***, que permite ajustar el color y la transparencia de nuestro objeto. Podemos encontrarlo en dos versiones, según si queremos utilizar el color mediante el nombre o dándoselo en RGB:

```
pov.addPigmentToNextObject(string color, double transmit, double filter);
pov.addPigmentToNextObject(int r, int g, int b, double transmit, double filter);
```

Funciona exactamente igual que la función anterior: hay que colocarlo justo antes del objeto al cual queremos modificar. Junto a esta última función, hay otra complementaria, que es *setObjectColorChecker*. "*Checker*" se refiere al tipo de color que habíamos utilizado en los planos, que estaban puestos en cuadraditos. Cualquier cosa que contenga esta propiedad se dibujará de esa forma en lugar de con un único color. Eso sí, esta función hay que usarla **de manera conjunta** con *addPigmentToNextObject*. La sintaxis es la siguiente, según si damos los dos colores de forma explícita o en RGB:

```
void setObjectColorChecker(string c1, string c2);
void setObjectColorChecker(int r1, int g1, int b1, int r2, int g2, int b2);
```

A la hora de utilizar esta función, no importa si se hace antes o después de añadir los colores, siempre y cuando se coloque *antes* de crear el objeto. Como las anteriores, solo funciona para el objeto siguiente. En este caso, se puede obviar el indicar el color en la función *addPigmentToNextObject:*

```
pov.setObjectColorChecker("Red", "Blue");
pov.addPigmentToNextObject(0,5, -1);
pov.createSphere(1, 0, 1, 0,5);
```

Por ejemplo ,este código genera una esfera dibujada a cuadros rojos y azules, semitransparente.

Nota: todos estos valores de propiedades (phong, transmit, filter...) deben de estar comprendidos entre 0 (mínimo) y 1 (máximo). Para más información, consultar la documentación de POV-Ray

Por último, hay que decir que estas funciones **no tienen efecto sobre los objetos estándar** que explicamos en la anterior sección. Esos objetos son objetos predefinidos, que no se pueden cambiar, y cuya utilidad simplemente es el dibujo rápido de objetos. En el momento que queramos algo especial, tendremos que echar mano de estos objetos primitivos junto con las funciones que acabamos de ver.

Por último, las texturas. Estas permiten dar más realismo a nuestros objetos. Muchas veces esto no será necesario, dado el carácte práctico de nuestra simulación pero conviene saber utilizarlas. Para emplear las textuas, primero le tenemos que indicar a POV-Ray que utilice la directiva include que incluye las texturas. Esto se hace con el código

```
pov.useTextures();
```

Después, podemos aplicar las texturas de la misma manera que el finish y el pigment. La diferencia es que la librería solo admite las texturas preestablecidas del programa. Para poner una textura a un objeto, usaremos, pues,

```
pov.addTextureToNextObject(string textureName);
```

También, dijimos que las funciones aquí explicadas afectan solo al siguiente objeto. Existen unas funciones que permiten fijar un estilo concreto y aplicarlo a todos los objetos. Esas funciones son las siguientes:

```
pov.setDefaultPigment();
pov.setDefaultFinish();
```

Estas, acompañadas de las funciones *addPigmentToNextObject* y *addFinishToNextObject* permiten seleccionarr un estilo predefinido. El estilo se mantendrá igual hasta la siguiente vez que aparezca *addPigmentToNextObject* (o *addFinishToNextObject*); si vuelve a aparecer, se impondrá el nuevo estilo como predefinido. Para acabar de usar estilos predefinidos, se utiliza

```
pov.unsetDefaults();
```

Por ejemplo, si tenemos

```
pov.setDefaultFinish();
pov.addFinishToNextObject(1, 0.5, 0.5);
//Todo tendrá las propiedades phong = 1, diffuse = 0,5, ambient = 0,5.
// (…)
pov.addFinishToNextObject(-1, -1,0.7);
//A partir de ahora todos los objetos saldrán sin phong, sin finish, y ambient =0.7.
pov.unsetDefaults();
//Todo objeto que se escriba a partir de ahora no tendrá propiedades a menos
//que se indique lo contrario.
```

Nota: no es posible poner una textura por defecto.

Por último, queda decir que aquí hemos presentado únicamente esferas y planos, pero también se pueden construir paralelepípedos, cilindros y toros. Se hace de una forma completamente análoga a los objetos ya comentados, de modo que no lo mostramos de forma explícita. De todas formas, la forma de crearlos viene en la documentación de funciones, donde se encuentran todos los prototipos de funciones.

3. Creación y uso de arrays de vectores

3.1. Arrays en POV-Ray. Declaración.

Lo siguiente es aprender a utilizar los arrays* en POV-Ray. El concepto es el mismo que el de array de C++, un vector o matriz que contiene datos. En POV-Ray, se puede hacer un array con cualquier tipo de dato. A nosotros nos interesan especialmente los arrays de vectores. El motivo es que es la manera más útil de realiza animaciones: en cada fotograma, la posición del objeto será la del vector n-ésimo, contenido en un array de vectores.

Bien, vamos con la declaración de arrays. Para utilizar una array, primero ha de ser declarado. La biblioteca de funciones nos ofrece varias alternativas para realizar esta operación. La más sencilla e intuitiva es:

```
pov.createFullVectorArray(int n, string name, double x[], double y[], double z[]);
```

Esta función crea un array de vectores de longitud *n*, con el nombre indicado en *name*. El vector i-ésimo tendrá las coordenadas x[i], y[i], z[i]. La ventaja de esta función es que es la más rápida para crear un vector.

Sin embargo, aún tiene un problema. Supongamos que necesitamos crear nuestro array de vectores a partir de dos, o tres arrays distintos donde tenemos datos. No podremos usar esta función, puesto que solamente almacena los datos de *un* array, y nosotros tenemos *varios*. O bien, si no tengo los datos en un array, sino sueltos o guardados en fichero de texto de una anterior simulación.

La solución pasa por las siguientes funciones:

```
pov.declareVectorArray(int n, string name);
pov.addElementToVectorArray(double x, double y, double z);
pov.closeVectorArray();
```

Es muy intuitivo utilizar estas funciones. Primero, declaramos un array con el nombre *name* y longitud *n* mediante la función *declareVectorArray*. Después, añadimos todos los elementos que queramos, que van a ser vectores. Si son muchos, basta combinarlo con un bucle for.

Por último, cerramos el array de vectores mediante *closeVectorArray*. Así podemos crear arrays con la estructura que nosotros queramos, y no solo mediante el array completo que tenemos en nuestro código de C++.

De hecho, como ejemplo de código sirve muy bien la implementación de la función *createFullVectorArray*:

```
int i;
declareVectorArray(n, name);
for (i=0; i < n; i++)
{
addElementToVectorArray(x[i], y[i], z[i]);
}
closeVectorArray();
```

3.2. Utilizando los vectores.

Por último, una vez sabemos cómo crear vectores, nos queda aprender es utilizarlos. Para ello,

existe una sobrecarga de la mayoría de las funciones para admitir el array en lugar de usar el vector manualmente. Para que la biblioteca sepa qué vector vamos a utilizar, debemos de indicárselo con la función

```
pov.setNextObjectPosFromVector(string name, string index);
```

Al igual que decíamos con los decoradores (*finish, pigment, texture*) esto sólo tiene efecto para el siguiente objeto. Es muy importante fijarse en que index está definido como string y no como int. El motivo es que la variable que se utiliza para animaciones es *clock*, aparte de las variables que puede definir el usuario, de modo que se deja como *string* para poder añadir nombres de variables, y no solo una posición concreta, como número.

En cuanto a la sobrecarga de funciones, se encuentran todas detalladas en la documentación. En nuestro caso, vamos a poner un ejemplo con la sobrecarga de la esfera:

```
pov.setNextObjectPosFromVector("array", "0");
pov.createSphere(2);
```

Podemos ver que en la sobrecarga no tenemos que poner la posición, simplemente basta con el radio (en el caso de la esfera). Si es posible sustituir dos o más parámetros por vectores, hay más sobrecargas de la función (o incluso cambia el nombre para poder diferenciarlas) de modo que se puede controlar completamente la posición de los diferentes objetos por vectores.

4. Creando Animaciones

Una vez controlamos los vectores, el siguiente paso es utilizarlos para crear animaciones. Debemos de tener claro que POV-Ray, para crear una animación, simplemente realiza renderiza las imágenes fotograma a fotograma, que nosotros juntamos mediante una aplicación externa. Para ello, necesitamos conocer los siguientes elementos:

- El código de POV-Ray en sí. Para crear una animación, simplemente crearemos un array de vectores con las posiciones sucesivas de un objeto. La posición del objeto tendrá como índice la variable “clock” que va cambiando de valor según su fotograma.
- Un archivo .ini donde vienen todas las propiedades necesarias para la animación; básicamente, el número de fotogramas empleados y los valores inicial y final de la variable clock.

Bien, vamos a comenzar por el código de POV-Ray. Hagamos un ejemplo sencillo: una bola que se mueve en línea recta, a lo largo de la trayectoria $y = x$. En C++ escribiremos el siguiente código:

```
int main(void)
{
	POVRayWriter pov ("C:\\output.pov");
	double t;

	pov.createCamera(0,5,20,0,0,0);
	pov.createLight(0,0,5);
```

```
	pov.declareVectorArray(30, "pos");
	for(t=0;t<3;t+=0.1)
	{
		pov.addElementToVectorArray(t,0,1);
	}
	pov.closeVectorArray();pov.createStandardPlane("z",0,"Blue", "Grey");
	pov.setNextObjectPosFromVector("pos", pov.Clock());
	pov.createStandardSphere(0.6, "Red");

	pov.closePOVRayWriter();

	return 0;

}
```

Esto nos generará un fichero con una cámara que mirará hacia abajo, una luz estándar superior; estamos mirando hacia una bola roja sobre un plano azul y gris situado en z=0. La posición de la bola se coge del array "pos", que hemos declarado previamente, añadiendo elemento a elemento, con las coordenadas (t,0,1). Ahora bien, queremos que nuestro fichero, para realizar la animación, vaya cogiendo pos[0], pos[1], pos[2], etc, de modo que la posición de la bola vaya variando. Para ello, simplemente usamos esta línea de código:

```
pov.createIniFile(string path, int frames, int initClock, int endClock);
```

Dado que esta función utiliza un escritor de archivos paralelo, es recomendable utilizarlo tras invocar a la función *closePovRayWriter()*. No hay que preocuparse por cerrar el fichero, la función *createIniFile* se ocupa ella sola de abrirlo, escribirlo y cerrarlo posteriormente.

Un detalle importante es el valor de *clock*. POV-Ray le asigna automáticamente los valores a esta variable mediante un cálculo simple. Por ejemplo, una animación de 100 fotogramas, que comience en initClock = 20 y acabe en endClock = 70. La variación de la variable clock será 70 – 20 / 100 = 0.5. Así,

```
clock[0] = initClock = 20,
clock[1] = 20.5,
clock[2] = 21,0,
( … ),
clock[99] = 69.5,
clock[100] = finalClock = 70.
```

Podemos ver que la variable clock, para empezar puede tomar valores decimales, lo cual no nos conviene. Esto es así porque además de como índice el array, puede usarse para indicar, por ejemplo el radio de una esfera. En general, nos va a interesar initClock = 0 y finalClock = frames, el número de fotogramas. Si queremos que la animación se salte algún vector, por ejemplo, queremos ver solo 1 de cada 5 fotogramas para hacernos una idea del movimiento y pasar a cámara rápida la animación, simplemente lo que haremos será que nuestro objeto, en vez de tomar la posición de pos[clock], la tomaremos de pos[5*clock], por ejemplo. Esto se puede hacer, dado que el argumento de índice para vectores es de cadena de texto.

Para indicar la posición del vector, hemos usado el código

```
pov.setNextObjectPosFromVector("pos", pov.Clock());
pov.createStandardSphere(0.6, "Red");
```

donde pov.Clock() es una función que devuelve la cadena “clock”. Esto se hace así para facilitar la escritura simple de esta variable. Si queremos “5*clock” simplemente utilizaremos la operación de concatenación de cadenas de texto.

También existe la sobrecarga

```
pov.createIniFile(string path, int frames);
```

que directamente ajusta las propiedades initClock = 0 y endClock = frames, para mayor comodidad.

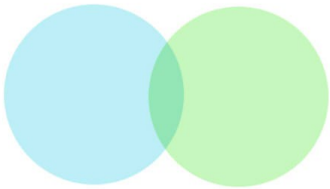
5. Geometrías compuestas. CSG.

Esta sección es complementaria para el estudio de la librería. En general, basta de sobra con los elementos presentados hasta ahora. Esos elementos nos van a permitir crear cualquier animación simple de mecánica.

Sin embargo, a veces, para nuestras representaciones querremos usar elementos compuestos o más elaborados. Estos se pueden crear en POV-Ray de varias maneras. Esta librería incluye dos: burbujas y geometría de sólidos constructiva.

5.1. Burbujas

Las burbujas son elementos formados por otros objetos. La burbuja tiene un valor umbral característico. A cada uno de los objetos, además, se le añade una característica de potencia. Cuando la potencia coincide con el umbral de la burbuja, se muestra la superficie del objeto. Supongamos que tenemos una esfera. Entonces, en la superficie de esa esfera, el radio tiene un valor *r*. Si *r* es igual al umbral, entonces se dibuja la superficie de la esfera. Por tanto, una burbuja formada por una sola esfera es una esfera de radio *r* igual al umbral. Ahora bien, supongamos que tenemos dos esferas de la misma potencia. En ese caso, ocurre lo siguiente:

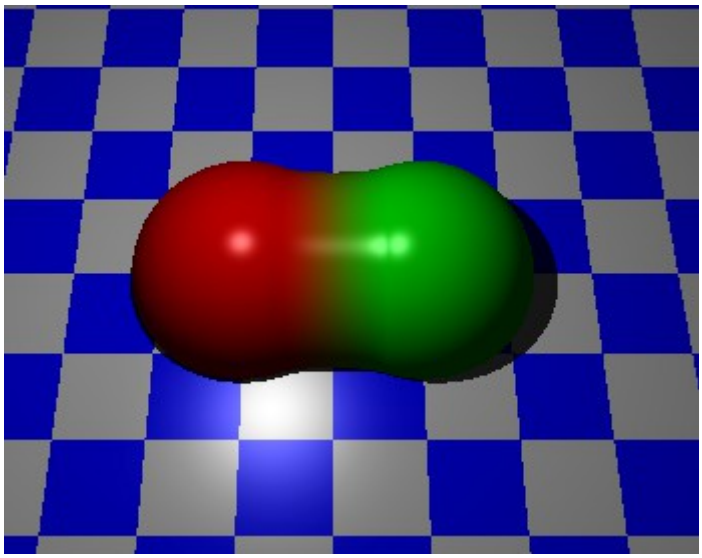


Ambas tienen la misma potencia. El valor del umbral de corresponde al borde de cada una. Con el siguiente código, que es el código base para crear una esfera:

```
pov.startBlob(0.7);

pov.addPowerToNextObjects(0.6);
pov.createStandardSphere(2,0,0,0, "Red");
pov.createStandardSphere(2,2,0,0, "Green");

pov.finishBlob();
```

Observemos algunos detalles. El argumento que se pasa a la función *startBlob* es el umbral. La función *addPowerToNextObjects* afecta a todos los objetos siguientes, de modo que en este caso solo hay que ponerla una vez. Después creamos los objetos que queramos que vayan dentro de la burbuja, y la cerramos.

Una vez hecho esto, y representando con POV-Ray, obtenemos la siguiente imagen:



Que es parecida al diagrama anterior, con la diferencia de que las esferas quedan "pegadas" por la parte en la que la suma de sus potencias es igual al umbral. Es de importancia resaltar que es posible añadir una potencia negativa a un objeto. Además, podemos añadir tantos cuantos queramos, siempre y cuando sean **cilindros o esferas.**

Por último, aunque podemos darle color a cada uno de nuestros objetos por separado, o incluso distintos *finish*, o *pigment*, mediante lo que hemos visto ya, también podemos añadir para todos los objetos simplemente añadiendo las sentencias de formato justo antes de la función *finishBlob*.

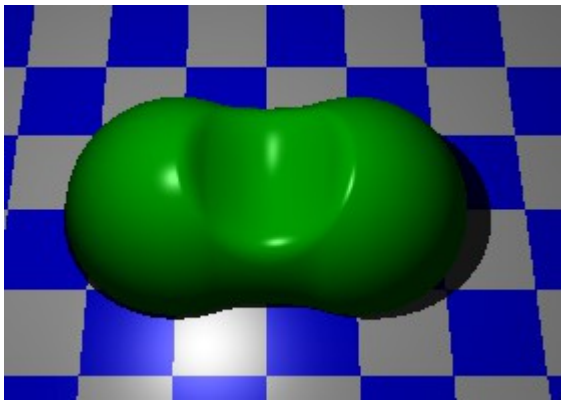
NOTA: este formato **NO** es afectado por los formatos estándar asignados mediante *setDefaultFinish* ó *setDefaultPigment*.

Esto es, mediante este código:

```
pov.startBlob(0.2);

pov.addPowerToNextObjects(0.6);
pov.createSphere(2,0,0,0);
pov.createSphere(2,2,0,0);

pov.addPowerToNextObjects(-0.3);
pov.createCylinder(1,0,-1,1,0,1,1);

pov.addPigmentToNextObject("Green",-1,-1);
pov.addFinishToNextObject(0.5,-1,0.5);

pov.finishBlob();
```

creamos las dos esferas de antes, y le añadimos un cilindro en medio con fuerza negativa. Después, añadimos color y acabado a la burbuja completa, a todos los objetos que hay dentro, no importa que no lleven un decorador individualmente.

Se obtiene esta imagen:



5.2. Geometría constructiva

La geometría constructiva (CSG por el inglés *Constructive Solid Geometry*) permite también construir objetos compuestos. Para ello, tenemos las operaciones usuales de álgebra de conjuntos. Hay más, pero la librería solo soporta estas:

- Unión: simplemente une varios objetos.
- Intersección: el objeto resultante es la intersección de todos los objetos datos.
- Diferencia: es la diferencia entre el primer objeto dado y los demás
- Fusión: funde varios objetos en uno solo. Es my parecido a la unión, solo que el primero simplemente convierte todos los objetos en un bloque y este los funde en un solo objeto.

La CSG funciona de forma muy parecida a las burbujas que hemos visto anteriormente. Se le puede añadir formato de la misma manera, hay que iniciarla y cerrar, etc. Las diferencias son que aquí no hay que darle potencia a cada objeto y que debemos especificar el tipo de CSG que vamos a realizar. Además, este admite cualquier tipo de objeto, no solo esferas y cilindros. Este es un ejemplo de diseño de CSG:

```
pov.startCSG(pov.CSGDifference());

pov.createSphere(3,0,0,0);
pov.createBox(-2,-2,-3,2,2,3);

pov.addPigmentToNextObject("Blue", -1,-1);

pov.finishCSG();
```

Como se puede ver, la esctructura es la misma de los objetos anteriores. Para indicar el tipo de CSG utilizado, la función CSGDifference simplemente devuelve la cadena de texto “difference”. De hecho, se pueden utilizar los modos “no implementados” simplemente escribiendo su cadena de texto correspondiente. No se han incluido como funciones porque no son tan usuales y con estas funciones debería ser suficiente para todo uso normal.

6. Listado de funciones

Aquí aparecen listadas todas las funciones de POV-Ray, por categorías.

6.1. Funciones fundamentales

```
1.   POVRayWriter(string path);
```

Crea un nuevo objeto de tipo POVRayWriter, inicializando un lector de ficheros que deberá cerrarse. Crea o edita un fichero nuevo en la ruta establecida.

```
2. void createIniFile(string path, int frames, int initClock, int endClock);
```

Crea un fichero .ini en el la ruta indicada, con el número de fotogramas establecido y los valores de reloj inicial y final seleccionados.

```
3. void createIniFile(string path, int frames);
```

Crea un fichero ini en la ruta indicada, con el número de fotogramas establecido. Los valores de reloj se ajustan al número de fotogramas para poder usarlo como índice de vectores.

```
4. string Clock();
```

Devuelve el valor de la variable *clock* de POV-Ray.

```
5. void write(string str);
```

Permite que el usuario añada el contenido que desee al fichero. Esto esto es útil para añadir funciones que no se encuentran en la librería y no necesitan de demasiadas instrucciones para escribirse.

```
6. void closePOVRayWriter();
```

Cierra el lector de ficheros de la librería. Es absolutamente obligatorio ejecutar esta función tras acabar de utilizar la librería.

6.2. Primitivas

```
7. void createCamera(double xi, double yi, double zi, double xf, double yf, double zf);
```

Crea una cámara en la posición (xi, yi, zi) mirando al punto (xf, yf, zf).

```
8. void createCameraLocationFromVectorArray(double xf, double yf, double zf);
```

Crea una cámara mirando a (xf, yf, zf) y con posición en el vector indicado mediante la función *setNextObjectPosFromVector*

```
9. void createCameraLookingAtFromVectorArray(double xi, double yi, double zi);
```

Crea una cámara en la posición (xi, yi, zi) mirando a la posición indicada mediante la función *setNextObjectPosFromVector*

```
10. void createCameraFromVectorArray(string vectorLocation, string vectorLookAt, string
    indexPos, string indexLookingAt);
```

Crea una cámara con posición en el vector vectorLocation[indexPos] y mirando al vector vectorLookAt[indexLookingAt]. **No** necesita el uso de *setNextObjectPosFromVector*.

```
11. void createCameraFromVectorArray(string vectorToUse, string indexPos, string
    indexLookingAt);
```

Crea una cámara con posición en vectorToUse[indexPos] y mirando al vector vectorToUse[indexLookingAt].

```
12. void createLight(double x, double y, double z, double r, double g, double b);
```

Crea una luz en la posición (x,y,z) con el color RGB indicado.

```
13. void createLight(double x, double y, double z);
```

Crea una luz en la posición (x,y,z) con el color RGB = (1,1,1), que es el color estándar de una luz.

```
14. void createSphere(double r, double x, double y, double z);
```

Crea una esfera de radio r en la posición (x,y,z)

```
15. void createSphere(double r);
```

Crea una esfera de radio r en la posición definida mediante la función *setNextObjectPosFromVector.*

```
16. void createPlane(double n1, double n2, double n2, double d);
```

Crea un plano mediante su vector perpendicular, usando su ecuación característica: n1 x + n2 y + n3 z = d

```
17. void createPlane(double d);
```

Crea un plano mediante su ecuación característica, tomando su vector perpendicular del vector indicado mediante *setNextObjectPosFromVector*

```
18. void createPlane(string plane, double d);
```

Crea el plano mediante su ecuación plane escrita de forma explícita a una distancia d, es decir, plane = d.

```
19. void createBox(double xi, double yi, double zi, double xf, double yf, double zf);
```

Crea un paralelogramo con su esquina inferior izquierda en (xi, yi, zi) y su esquina superior derecha en (xf, yf, zf).

```
20. void createBoxInitFromVectorArray(double xf, double yf, double zf);
```

Crea un paralelogramo con su esquina inferior izquierda en el vector indicado mediante *setNextObjectPosFromVector* y su esquina superior derecha en (xf, yf, zf).

```
21. void createBoxLastFromVectorArray(double xi, double yi, double zi);
```

Crea un paralelogramo con su esquina inferior izquierda en (xi, yi, zi) y su esquina superior derecha en la posición indicada por *setNextObjectPosFromVector*

```
22. void createBoxFromVectorArray(string initVector, string lastVector, string
    initIndex, string lastIndex);
```

Crea un paralelogramo con su esquina inferior izquierda en el vector initVector[initIndex] y su esquina superior derecha en el vector lastVector[lastIndex].

```
23. void createCylinder(double xi, double yi, double zi, double xf, double yf, double
    zf, double r);
```

Crea un cilindro de radio r con su cara inferior centrada en (xi,yi,zi) y su cara superior centrada en (xf, yf, zf).

```
24. void createCylinderInitFromVectorArray(double xf, double yf, double zf, double r);
```

Crea un cilindro de radio r con su cara inferior tomada de *setNextObjectPosFromVector* y su cara superior centrada en (xf, yf, zf).

```
25. void createCylinderLastFromVectorArray(double xi, double yi, double zi, double r);
```

Crea un cilindro de radio r con su cara inferior centrada en (xi,yi,zi) y su cara superior tomada de *setNextObjectPosFromVector*.

```
26. void createCylinderFromVectorArray(string initVector, string lastVector, string
    initIndex, string lastIndex, double r);
```

Crea un cilindro de radio r con su cara inferior centrada en initVector[initIndex] y cara superior centrada en lastVector[lastIndex].

```
27. void createTorus(double x, double y, double z, double middleR, double minorR);
```

Crea un toro en la posición (x,y,z), con el radio medio middleR y el radio interior minorR.

```
28. void createTorus(double middleR, double minorR);
```

Crea un toro en la posición especificada por *setNextObjectPosFromVector*, con un radio medio middleR y un radio interior minorR.

6.3. Primitivas estándar

```
29. void createStandardSphere(double r, double x, double y, double z, string color);


30. void createStandardSphere(double r, string color);


31. void createStandardPlane(double n1, double n2, double n2, double d, string c1,
    string c2);


32. void createStandardPlane(double d, string c1, string c2);


33. void createStandardPlane(string plane, double d, string c1, string c2);
```

```
34. void createStandardBox(double xi, double yi, double zi, double xf, double yf,
    double zf, string color);


35. void createStandardBoxInitFromVectorArray(double xf, double yf, double zf, string
    color);


36. void createStandardBoxLastFromVectorArray(double xi, double yi, double zi, string
    color);


37. void createStandardBoxFromVectorArray(string initVector, string lastVector, string
    initIndex, string lastIndex, string color);


38. void createStandardCylinder(double xi, double yi, double zi, double xf, double yf,
    double zf, double r, string color);


39. void createStandardCylinderInitFromVectorArray(double xf, double yf, double zf,
    double r, string color);


40. void createStandardCylinderLastFromVectorArray(double xi, double yi, double
    zi,double r, string color);


41. void createStandardCylinderFromVectorArray(string initVector, string lastVector,
    string initIndex, string lastIndex, double r, string color);


42. void createStandardTorus(double middleR, double minorR, string color);


43. void createStandardTorus(double x, double y, double z, double middleR, double
    minorR, string color);
```

Todas estas funciones son exactamente a sus correspondientes primitivas, simplemente contienen un valor de *phong* = 0.5 y se dibujan con el color o colores indicados en el último argumento.

6.4. Decoradores

```
44. void addFinishToNextObject(double phong, double ambient, double diffuse);
```

Añade la etiqueta *finish* al siguiente objeto, con los valores indicados en los argumentos. Si no queremos que se añada alguno de ellos, le pasaremos argumento -1.

```
45. void addPigmentToNextObject(string color, double transmit, double filter);
```

Añade la etiqueta *pigment* al siguiente objeto. Si no queremos que se añada *transmit* o *filter*, le pasaremos argumento -1. El color es obligatorio, y se escribirá uno de los colores preestablecidos.

```
46. void addPigmentToNextObject(double transmit, double filter);
```

Añade la etiqueta *pigment* al siguiente objeto. Si no queremos que se añada *transmit* o *filter*, le pasaremos argumento -1. El color del objeto será blanco por defecto si no se indica otro con setObjectColorChecker.

```
47. void addPigmentToNextObject(int r, int g, int b, double transmit, double filter);
```

Añade la etiqueta *pigment* al siguiente objeto. Si no queremos que se añada *transmit* o *filter*, le pasaremos argumento -1. El color del objeto será el color RGB indicado, y es obligatorio.

```
48. void setObjectColorChecker(string c1, string c2);
```

Se usa conjuntamente con *addPigmentToNextObject*, usándose igual, antes de crear el objeto. Al añadir esta función, el objeto será dibujado a cuadros con los dos colores predeterminados indicados.

```
49. void setObjectColorChecker(int r1, int g1, int b1, int r2, int g2, int b2);
```

Se usa conjuntamente con *addPigmentToNextObject*, usándose igual, antes de crear el objeto. Al añadir esta función, el objeto será dibujado a cuadros con los dos colores indicados mediante RGB.

```
50. void useTextures();
```

Se utiliza para poder utilizar texturas en el resto del fichero. Debe llamarse a esta función al principio.

```
51. void addTextureToNextObject(string texture);
```

Añade una textura predefinida al siguiente objeto.

```
52. void setDefaultPigment();
```

Permite establecer una configuración por defecto de *pigment* para todos los objetos que se declaren a partir de esta función. La etiqueta *pigment* que tendrán se especifica con *addPigmentToNextObject.*

```
53. void setDefaultFinish();
```

Permite establecer una configuración por defecto de *finish* para todos los objetos que se declaren a partir de esta función. La etiqueta *finish* que tendrán se especifica con *addFinishToNextObject.*

```
54. void unsetDefaults();
```

Elimina el uso de las configuraciones por defecto a partir de esta función.

```
55. void addRotationToNextObject(double x, double y, double z);
```

Rota el siguiente objeto una cantidad x respecto al eje X, y respecto al eje Y, y z respecto al eje Z. Por defecto, se usan grados, pero se pueden seleccionar radianes.

```
56. void addRotationToNextObject(string name, string index);
```

Añade una rotación al siguiente objeto mediante el vector name[index]. Solo se permite usar grados en este caso. Si se quiere usar radianes, se debe hacer la transformación radianes – grados al crear el vector.

```
57. void useRadians();
```

Selecciona los radianes como una unidad de rotación.

```
58. void useDegrees();
```

Selecciona los grados como unidad de rotación.

6.5. Arrays de vectores

```
59. void declareVectorArray(int n, string name);
```

Declara un nuevo array de vectores de n elementos con el nombre name.

```
60. void addElementToVectorArray(double x, double y, double z);
```

Añade un elemento (x,y,z) al vector abierto actualmente.

```
61. void closeVectorArray();
```

Cierra el vector abierto actualmente.

```
62. void createFullVectorArray(int n, string name, double x[], double y[], double z[]);
```

Crea un array de vectores de n elementos con el nombre name, con las posiciones (x[i], y[i], z[i]). Esta función declara, añade todos los elementos indicados en el vector y lo cierra, de modo que no es necesaria ninguna función más.

```
63. void setNextObjectPosFromVector(string name, string index);
```

Indica que el siguiente objeto utilizará el elemento index del vector con nombre name.

6.6. Geometrías compuestas

```
64. void startBlob(double threshold);
```

Inicia una burbuja con un umbral threshold.

```
65. void addPowerToNextObjects(double power);
```

Indica la potencia que deben tener los siguientes objetos dentro de la burbuja. Es posible añadirlo varias veces para dar valores distintos a diferentes objetos.

```
66. void finishBlob();
```

Finaliza la burbuja. Puede ir precedido de un decorador para darle un formato global a la burbuja. No es afectado por los formatos predefinidos.

```
67. void startCSG(string CSG);
```

Inicia una geometría CSG del tipo indicado en el argumento.

```
68. void finishCSG();
```

Finaliza la CSG. Puede ir precedido de un decorador para darle un formato global a la CSG. No es afectado por los formatos predefinidos.

```
69. string CSGUnion();
```

Devuelve la cadena correspondiente a la CSG de unión. Debe ir en el argumento de *startCSG*.

```
70. string CSGIntersection();
```

Devuelve la cadena correspondiente a la CSG de intersección. Debe ir en el argumento de *startCSG*.

```
71. string CSGDifference();
```

Devuelve la cadena correspondiente a la CSG de diferencia. Debe ir en el argumento de *startCSG*.

```
72. string CSGMerge();
```

Devuelve la cadena correspondiente a la CSG de fusión. Debe ir en el argumento de *startCSG*.

7. Recursos útiles

- Lista de colores predefinidos en POV-Ray: http://www.f-lohmueller.de/pov_tut/tex/tex_150e.htm
- Lista de texturas predefinidas en POV-Ray: http://www.f-lohmueller.de/pov_tut/tex/tex_160e.htm
- Tutoriales de POV-Ray: http://www.f-lohmueller.de/pov_tut/pov__eng.htm#textures