Construcción de imágenes de estructuras simétricas mediante programación funcional

Carlos Velarde V.
Laboratorio de Simulación de Materiales
Unidad de Cómputo para la Investigación, DGSCA, UNAM
México
velarde@servidor.unam.mx

Resumen

Se muestran las bondades de la programación funcional para construir imágenes geométricas de estructuras simétricas, que se usan para ilustrar diversos conceptos matemáticos, algunos pertenecientes a la programación y a la geometría mismas, otros al álgebra combinatoria, grupos y gráficas. Para visualizar y examinar las figuras se producen programas en VRML, lenguaje adecuado para estos fines.

1 Introducción

En este trabajo se explora el uso de la programación funcional [Bird and Wadler, 1987] con un lenguaje fuertemente tipificado y no estricto, como *Haskell* [Hudak and Fasel, 1992; Peterson *et al.*, 1997] y *Gofer* [Jones, 1991], para el cálculo de estructuras simétricas y la producción de programas en VRML [VRML97, 1997] que permitan ver y examinar imágenes geométricas relacionadas con dichas estructuras. Así, se producen imágenes del árbol generado al calcular los elementos de una estructura y del bosque de expansión de la gráfica que una simetría determina en forma natural. Los dos lenguajes mencionados son iguales en esencia y aquí usaremos el segundo.

En lo que se refiere al cálculo de las estructuras simétricas, el presente trabajo está inspirado y se apoya en el de Strooper y van Emden [1992], quienes presentan un método para describir estructuras simétricas basándose en el grupo de simetrías correspondiente. Mediante un programa lógico conciso y claro, definen la estructura en términos de un conjunto *soporte*, formado por unos cuantos de los elementos que la componen, y un conjunto de *generadores* de algún subgrupo de simetrías de la estructura. Ambos conjuntos se escogen procurando que tengan pocos elementos, pero de tal suerte que todo elemento de la estructura se pueda obtener mediante aplicaciones de los generadores a los miembros del conjunto soporte. Para generar sin repeticiones los elementos de la estructura se requiere programación adicional, tarea que se simplifica representando en forma canónica las simetrías del subgrupo en juego, como producto de potencias de los generadores (aquí “producto” significa composición). Sin embargo, cada caso requiere un trato particular, dependiendo del conjunto soporte, del conjunto de generadores y de la forma canónica escogida. Ilustran su método agregando predicados para definir la aplicación repetida de un generador y para calcular los vértices de una esfera geodésica.

Aquí se darán pasos similares para calcular la misma esfera geodésica, pero mediante programación funcional. Los programas lógicos ahí presentados se pueden ver como una especificación formal de los que aquí se dan.

La estructura del artículo es como sigue. La segunda sección trata de la construcción del dodecaedro a partir de la proporción áurea, se presentan funciones para construir la permutación asociada a una rotación y se ilustran algunos algoritmos de gráficas; la tercera sección versa sobre la construcción de la esfera geodésica; la última sección está dedicada al cálculo de la geometría de la molécula de carbono C_{60}. En general, los programas (definiciones de funciones) se presentan conforme se van necesitando, aunque algunas funciones se dan en el Apéndice A. En otro apéndice se muestra un ejemplo de programa en VRML.

2 El dodecaedro

En esta sección, partiendo de la proporción áurea, se construye un dodecaedro regular inscrito en la esfera unitaria con centro en el origen, como se ilustra en la Figura 1. Se encuentra colocado con dos caras paralelas al plano xy y con el vértice `ptoB` sobre el plano xz. Siguiendo a Strooper y van Emden [1992], los demás vértices se obtienen aplicando a `ptoB` composiciones apropiadas de tres generadores del grupo de rotaciones del dodecaedro. Estas rotaciones son: la de un quinto de vuelta alrededor del eje `ejeAlfa`, la de un quinto de vuelta en torno al eje `ejeBeta` y la de media vuelta alrededor del eje `ejeGama`. Las rotaciones se efectúan en sentido positivo o de “mano derecha”, es decir, si el dedo pulgar de la mano derecha apunta en la dirección del eje, el giro se da hacia donde apuntan los otros dedos.

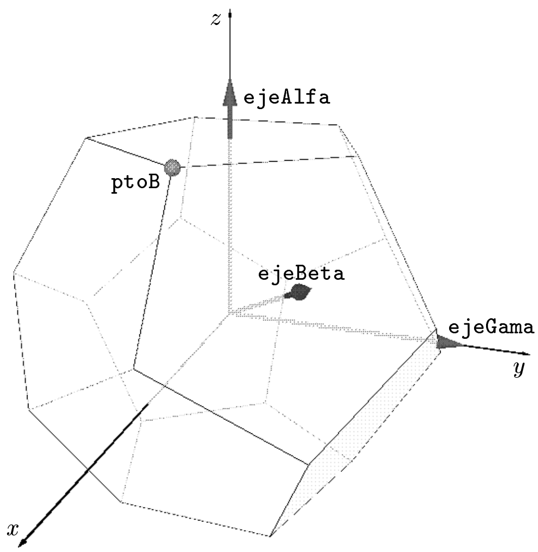

Figura 1: El dodecaedro.

La programación en lenguajes como *Haskell* o *Gofer* consiste en escribir definiciones de funciones, a la vez que se presenta una nomenclatura según el gusto y estilo de cada quién, en forma parecida a la acostumbrada en matemáticas. Así, para definir las rotaciones mencionadas, se nombran convenientemente algunos tipos de datos:

```
type R = Float
type R3 = (R,R,R)
type Rotacion = R3 -> R3
```

De esta manera, de aquí en adelante se puede usar indistintamente `R` o `Float` como nombre del conjunto de los números de punto flotante, `R3` también denotará al producto cartesiano R^3 y `Rotacion` se puede usar como nombre del conjunto de funciones de `R3` en `R3`. Las rotaciones antes mencionadas se definen así:

```
rotAlfa, rotBeta, rotGama :: Rotacion
rotAlfa = rota ((2.0*pi)/5.0) ejeAlfa
rotBeta = rota ((2.0*pi)/5.0) ejeBeta
rotGama = rota pi ejeGama
```

En estas expresiones aparecen términos que a su vez deben definirse. En cuanto a `pi`, se trata de un nombre para el número π y el lenguaje de programación incluye su definición. La función `rota` se define así:

```
type Angulo = R
type Eje = R3

rota :: Angulo -> Eje -> Rotacion
rota ang eje = ap_rot (matRot ang eje)
```

Por conveniencia se introducen nuevos nombres para `R` y `R3`. La función `rota` recibe dos argumentos, correspondientes al ángulo y al eje de rotación, y produce una función de tipo `Rotacion`. En general, en *Haskell* los argumentos y los valores de una función pueden ser de *cualquier* tipo. Aquí no se incluye la definición de las funciones que aparecen en la parte derecha de la última igualdad, `matRot` construye la matriz de rotación y `ap_rot` efectúa la multiplicación de esta matriz por un elemento de `R3`. Como el lenguaje de programación es fuertemente tipificado, las declaraciones de tipos, por ejemplo `rota :: Angulo -> Eje -> Rotacion`, pueden omitirse, pues son deducibles a partir de las igualdades que definen funciones. Sin embargo, son recomendables para facilitar la lectura y, mas importante, porque si el sistema de deducción de tipos infiere algo distinto a lo declarado, tal conflicto se informará al programador, para que corrija. Acerca de los ejes de rotación, se tiene:

```
ejeAlfa, ejeBeta, ejeGama :: Eje
ejeAlfa = (0.0, 0.0, 1.0)
ejeGama = (0.0, 1.0, 0.0)
```

y enseguida se define `ejeBeta`. Este eje y `ejeAlfa` pasan, respectivamente, por los centros de dos caras contiguas del dodecaedro. Ahora se calculará el ángulo β que forman dichos ejes. Para ello, téngase presente que los centros de las caras del dodecaedro son los vértices de un icosaedro (Figura 3) y que éstos a su vez son los 12 vértices de tres rectángulos áureos, colocados como se muestra en la Figura 2.

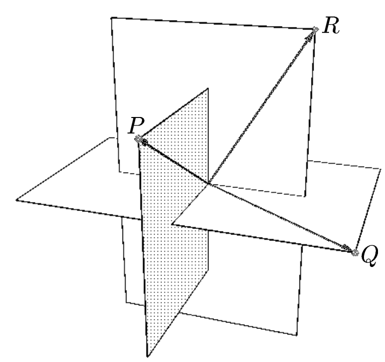

Figura 2: $P = (k, 0, 1)$, $Q = (1, k, 0)$, $R = (0, 1, k)$ rectsAu-1dit .

La proporción áurea $k = (1 + \sqrt{5})/2$ es igual a la solución positiva de la ecuación

$$k^2 = k + 1 \qquad (1)$$

Como los puntos $P = (k, 0, 1)$ y $R = (0, 1, k)$ son centros de dos caras contiguas de un dodecaedro, usando la fórmula que define al coseno en términos del producto interior y simplificando con ayuda de (1), se obtiene:

$$\cos\beta = k/(2 + k) \qquad (2)$$

Ahora se puede obtener `ejeBeta` aplicando a `ejeAlfa` una rotación de ángulo β alrededor de un eje paralelo a la arista común a las caras del dodecaedro correspondientes a dichos ejes. Escribiendo esto en fórmulas, que además constituyen definiciones ejecutables, se tiene:

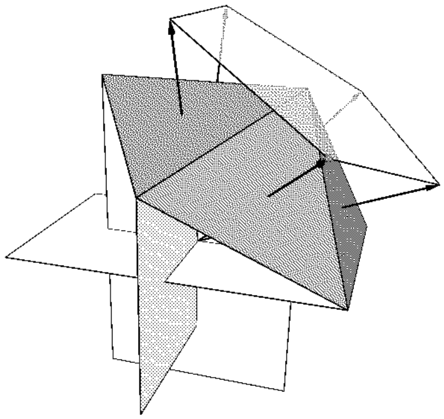

Figura 3: Cada vértice del icosaedro es centro de una cara de dodecaedro.

```
k = (1.0+sqrt 5.0)/2.0

ejeBeta = rota angulo ejeRot ejeAlfa
  where angulo = (acos(k/(2.0+k)))
        ejeRot = (-sin(pi/5.0),cos(pi/5.0),0.0)
```

El punto ptoB se calcula normalizando la suma de los centros de las tres caras de las que es vértice común. Dichos centros son ejeAlfa, ejeBeta y la reflexión de éste sobre el plano xz:

```
type Punto = R3

ptoB :: Punto
ptoB = direccion (ejeAlfa+ejeBeta+ejeBeta')
       where ejeBeta' = (eB1,-eB2,eB3)
             (eB1,eB2,eB3) = ejeBeta
```

en que direccion, definida en el Apéndice A, es una función que da la *dirección* (vector unitario) de su argumento.

2.1 Generación de los vértices

Los vértices del dodecaedro se computan aplicando a ptoB las 20 simetrías siguientes, pertenecientes al grupo de 60 rotaciones del dodecaedro y expresadas como composiciones de los tres generadores:

$$\mathtt{rotAlfa}^i \circ \mathtt{rotBeta}^j \circ \mathtt{rotGama}^k$$

($i{=}0,\ldots,4$; $j,k{=}0,1$), en que f^n denota la composición repetida de una función consigo misma, definida así: $f^0 = id$ (la función identidad), $f^{n+1} = f^n \circ f$. El lenguaje de programación proporciona la siguiente definición de la función iterate, que a una función f y un elemento x asocia la lista infinita [x, f x,..., f^n x,...]:

```
iterate :: (a -> a) -> a -> [a]
iterate f x = x : iterate f (f x)
```

Es un buen momento para observar que no es lo mismo "definir" que "computar". Por ejemplo, la definición del conjunto de los números primos es bien conocida, pero calcularlos todos es tarea imposible, pues constituyen un conjunto infinito. Sin embargo, la misma definición es usada para construir conjuntos finitos de números primos. En forma parecida, iterate f x denota una lista infinita, imposible de calcular completamente, pero la definición puede usarse para construir partes finitas de la lista. Este es el propósito de la siguiente función, que sirve para extraer los primeros n elementos de la lista:

```
iteraN :: Int -> (a -> a) -> a -> [a]
iteraN n f x = take n (iterate f x)
```

Si esta función se programa de la misma manera en algún lenguaje imperativo como *Pascal*, *Fortran* o *C*, su ejecución ocasiona una computación infinita, debido a que en estos lenguajes la aplicación de una función se realiza siguiendo un orden conocido como *aplicativo*, en el que la evaluación de los argumentos se debe realizar antes del inicio de la ejecución ("llamada") de la función. Así, para la ejecución de take se requeriría la previa e imposible computación de su argumento (iterate f x).

En contraste, en lenguajes como *Haskell* y *Gofer* la aplicación de una función se efectúa en orden *normal*, en que la evaluación de la función se inicia antes de la de sus argumentos. Además, tales lenguajes incorporan la evaluación en orden normal de una manera eficiente, llamada evaluación *retardada* o *floja* ("lazy" en inglés) porque nada se evalúa sino hasta que se necesita y sólo se evalúan las partes necesarias [Davie, 1992]. Así, la evaluación de take n (iterate f x) sólo necesitará la computación de los primeros n elementos de la lista infinita iterate f x, conforme los vaya requiriendo la evaluación de take.

Considerando a las listas infinitas como valores *indefinidos*, tenemos que la función take puede dar un valor definido a pesar de que su segundo argumento sea de valor indefinido. A las funciones que tienen esta propiedad se les llama *no estrictas*. Dicho de otra manera, una función es *estricta* si resulta indefinida para argumentos indefinidos. En los lenguajes imperativos arriba mencionados, take es estricta. La evaluación en orden normal de manera natural permite la definición de funciones no estrictas, por eso se dice que los lenguajes como *Haskell* son *no estrictos*.

Ahora se puede obtener la lista de vértices del dodecaedro evaluando la siguiente expresión:

```
iteraN' 5 rotAlfa
  (iteraN' 2 rotBeta
    (iteraN' 2 rotGama [ptoB]))
where
  iteraN' n f xs = concat (map (iteraN n f) xs)
```

Sin embargo, interesa producir imágenes no sólo de los vértices, sino también de aristas y caras de los poliedros, así como figuras que ilustren algoritmos que versan sobre árboles y gráficas. Para ello convienen las estructuras arborescentes, como los árboles *generales* o *etiquetados*, en que cada nodo se etiqueta con un valor de tipo genérico `e` y tiene un número arbitrario de subárboles:

```
data Arbol e = Nodo e [Arbol e]
```

Ahora se pueden definir funciones como la siguiente, que sirve para enumerar las etiquetas de los nodos, listadas en el orden determinado al visitar los nodos en *preorden*.

```
preordArb :: Arbol e -> [e]
preordArb (Nodo p hs) =
  [p] ++ concat (map preordArb hs)
```

En estas definiciones, `e` denota un tipo cualquiera, es una variable sobre el conjunto de tipos y puede ser sustituida por cualquier tipo particular. Por ejemplo, para construir el árbol de vértices del dodecaedro, convienen árboles etiquetados con valores del tipo `Punto`:

```
type ArbolVerts = Arbol Punto

arbol_vertsDodec :: Punto -> ArbolVerts
arbol_vertsDodec p =
  iteraArbN 5 rotAlfa
    (iteraArbN 2 rotBeta
       (iteraArbN 2 rotGama (Nodo p []) ))
```

en que `iteraArbN n f` (ver Apéndice A) como argumento recibe un árbol, al que transforma como si se aplicara `(iteraN n f)` a cada nodo, es decir, lo hace crecer sustituyendo cada nodo por una rama de nodos etiquetados por los valores que generaría `(iteraN n f)` al aplicarse al nodo reemplazado. La aplicación de la función `arbol_vertsDodec` al punto `ptoB` da el árbol de vértices del dodecaedro:

```
arbDodec = arbol_vertsDodec ptoB
```

En la Figura 4 se muestra una representación geométrica de este árbol, cuyos nodos son precisamente los vértices del dodecaedro y las trayectorias que parten de la raíz `ptoB` indican la manera en que se generaron los nodos, cada flecha corresponde a la aplicación de una de las tres rotaciones.

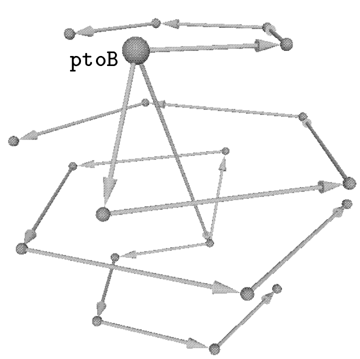

Figura 4: Árbol de generación de vértices del dodecaedro.

Para construir la figura, primero se extraen del árbol dos listas, la de sus nodos y la de sus lados, aplicándole respectivamente la función `preordArb` arriba presentada y la función `ladosArb` que aparece en el Apéndice A:

```
vertsDodec = preordArb arbDodec
ladosArbDodec = ladosArb arbDodec
```

Estas listas sirven de argumentos a respectivas funciones —`nodos` y `flechas`— encargadas de producir textos como los siguientes:

```
Nodo{radio_nodo 0.06
     centro 0.607062 0.0 0.794655}
Nodo{radio_nodo 0.03
     centro 0.187592 0.57735 0.794655}

Flecha{rot_flecha 0.0 0.0 1.0 0.628318
       trans_flecha 0.187592 0.57735 0.794655
       radio_punta 0.03
       altura_punta 0.12
       radio_arista 0.01
       altura_arista 0.533644
       trans_punta 0 -0.09 0
       trans_arista 0 -0.326822 0
       color 0.8 0.4 0.4}
```

Estos textos forman parte de un programa en lenguaje VRML, *Nodo* y *Flecha* son los nombres de dos *prototipos* o *clases* previamente definidos —también mediante funciones escritas en el lenguaje de programación funcional— y su uso en expresiones como las anteriores indica la producción de *objetos* (*nodos*, en la terminología de VRML) derivados de los respectivos prototipos. En este caso, se trata de objetos geométricos. *Nodo* sirve para construir esferas, sus *campos* *radio_nodo* y *centro* sirven para especificar el radio de la esfera y la posición en que ésta ha de colocarse dentro de la imagen. Sobre la figura, al primer *Nodo* antes listado le corresponde la esfera marcada `ptoB`. En forma parecida, al objeto de tipo *Flecha* mostrado le corresponde la flecha que en la figura apunta de `ptoB` hacia la derecha. El prototipo *Flecha* es más elaborado, cada flecha se compone de un cilindro y un cono (objetos que VRML proporciona) y sus campos, como en el caso de la esfera, sirven para indicar las dimensiones y el posicionamiento de la flecha en la imagen.

2.2 Rotaciones, permutaciones y gráficas

Numerados los vértices de acuerdo a su posición en la lista `vertsDodec`, todo elemento del grupo de simetrías del dodecaedro puede representarse como una permutación del conjunto {0,1,..,19}. En la misma forma se pueden tratar las simetrías de toda estructura determinada por un conjunto finito de elementos, numerados de 0 a n. La siguiente función recibe como argumentos una simetría y la lista de vértices de la estructura, como valor da la permutación correspondiente.

```
type Permutacion = Int -> Int

permutacion :: Rotacion->[Punto]->Permutacion
permutacion f vs =
  ([posicion vs fv | fv<-map f vs]!!)
```

Ahora se definen las permutaciones asociadas a los generadores:

```
permAlfa, permBeta, permGama :: Permutacion
permAlfa = permutacion rotAlfa vertsDodec
permBeta = permutacion rotBeta vertsDodec
permGama = permutacion rotGama vertsDodec
```

Así, la expresión `[(i,permAlfa i)|i<-[0..19]]` será reducida por el mecanismo de evaluación a la lista siguiente, que es la representación de `permAlfa` en forma de parejas ordenadas:

```
[(0,1), (1,2), (2,3), (3,4), (4,0),
 (5,6), (6,7), (7,8), (8,9), (9,5),
 (10,11), (11,12), (12,13), (13,14), (14,10),
 (15,16), (16,17), (17,18), (18,19), (19,15)]
```

La siguiente función, similar a `permutacion`, sirve para construir directamente este tipo de representaciones:

```
type PermParejas = [(Int,Int)]

permParejas :: Rotacion->[Punto]->PermParejas
permParejas f vs =
  [(i, posicion vs (fvs!!i)) | i<-[0..n]]
  where fvs = map f vs
        n   = length vs - 1
```

Otra representación de una permutación se da listando sus ciclos, a continuación se presenta un par de funciones para construirlos, aprovechando un magnífico algoritmo publicado en [Launchbury, 1995] para generar un bosque de expansión a una gráfica dada. Una permutación p puede verse como una gráfica dirigida g, con $\{0, \ldots, i, \ldots, n\}$ como conjunto de nodos y cuyos arcos son las parejas de la forma $(i, p(i))$. Cada ciclo de p determina un ciclo en g, el que a su vez es una componente fuertemente conexa (subgráfica máxima para la que existe algún camino entre cualesquiera dos de sus vértices) de g y, por tratarse de una componente cíclica, también es componente conexa máxima de g (considerando no dirigidos los arcos). Además la gráfica no tiene otras componentes conexas máximas. Hechas estas consideraciones, las componentes fuertemente conexas de g se pueden obtener simplificando el Algoritmo 3 dado en la misma publicación, para dejarlo en la forma:

```
componentes_ciclicas :: Graph -> Forest Vertex
componentes_ciclicas = dff
```

en que `dff` ("depth-first spanning forest") es una función que construye un bosque de expansión a una gráfica dada. Los árboles del bosque de g resultan lineales, cada uno corresponde a un ciclo de la gráfica y al recorrerlos en preorden se obtienen los ciclos de p. Escrito ésto en fórmulas:

```
Ciclos = [[Int]]

ciclos :: PermParejas -> Ciclos
ciclos p =
  map preorder (componentes_ciclicas g)
  where g =
    buildG (0,length p - 1) [x:=y | (x,y)<-p]
```

en que `preorder` y `buildG` son funciones que aparecen definidas en el artículo antes mencionado. Por ejemplo, la evaluación de las dos expresiones siguientes:

```
ciclos (permParejas rotAlfa vertsDodec)
ciclos (permParejas rotBeta vertsDodec)
```

da lugar, respectivamente, a las listas mostradas enseguida, de las cuales la primera se usa para construir la Figura 5:

```
[[0,1,2,3,4], [5,6,7,8,9], [10,11,12,13,14],
 [15,16,17,18,19]]
[[0,5,18,6,1], [2,4,17,13,19], [3,9,12,14,7],
 [8,16,11,10,15]]
```

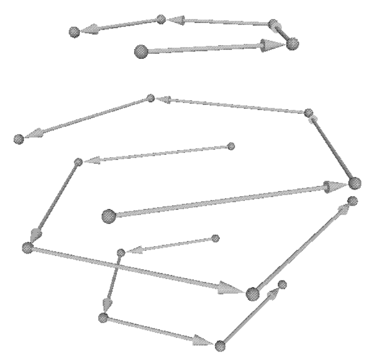

Figura 5: Bosque de expansión de la gráfica asociada a `permAlfa`.

2.3 Construcción de las caras

Una vez construidas las permutaciones que las simetrías determinan, se pueden usar para computar otros objetos geométricos de interés, como son las caras y aristas de poliedros. Se ilustra ésto calculando las caras del dodecaedro. Con la numeración asignada a los vértices del poliedro, un polígono se puede representar listando los números correspondientes a sus propios vértices, éste es el propósito del primero de los siguientes tipos:

```
type Poligono = [Int]
type ArbolPoligonos = Arbol Poligono
```

Ahora se define una función para construir las caras del dodecaedro a partir de la cara superior (a cuyo primer vértice, ptoB, le corresponde 0 en la numeración de vertsDodec antes presentada):

```
type Pentagono = Poligono
type ArbolPents = ArbolPoligonos

arbol_carasDodec :: Pentagono -> ArbolPents
arbol_carasDodec p =
  Nodo p ([Nodo (pG p) []] ++
          [iteraArbN 2 pG
           (iteraArbN 5 pA (Nodo (pB p) []))])
  where pA = map permAlfa
        pB = map permBeta
        pG = map permGama

carasDodec :: [Pentagono]
carasDodec =
  preordArb (arbol_carasDodec caraSup)
  where caraSup = iteraN 5 permAlfa 0
```

Precisamente ésta y vertsDodec son las listas que sirven para generar el programa mostrado en el Apéndice B, donde en lenguaje VRML se define el dodecaedro prototipo, usado en la construcción de figuras como la 1, la 7 y la 14.

3 Esfera geodésica

Aquí se da una manera de construir la esfera geodésica mostrada en la Figura 6, de nuevo siguiendo a Strooper y van Emden [1992]. Este poliedro irregular, de 240 caras triangulares y 122 vértices colocados sobre una esfera, se construye a partir del dodecaedro, dando lugar cada cara de éste a 20 caras del primero. Los vértices —de los cuales aparecen nombrados cuatro en la Figura 7— son los 20 vértices del dodecaedro base, los 12 puntos similares a ptoA, proyecciones de los centros de las caras del dodecaedro sobre la esfera que lo circunscribe, los 30 puntos similares a ptoD, proyecciones de los puntos medios de las aristas del dodecaedro, y los 60 puntos similares a ptoE, siendo este último el punto de intersección de dos círculos máximos, uno el que determinan ptoB y ptoA y el otro el que en ptoD forma un ángulo de 60° con el que pasa por ptoB.

Los generadores usados en el caso del dodecaedro también sirven para la esfera geodésica, pues sin dificultad se puede ver que el grupo de rotaciones es el mismo para ambos poliedros. El conjunto soporte para generar los vértices será el que tiene a ptoA, ptoB, ptoD y ptoE por elementos. En la sección anterior se calculó ptoB. El vértice ptoA coincide con ejeAlfa y ptoD se construye normalizando la suma de ejeAlfa y ejeBeta (ver Figuras 1 y 7):

```
ptoA = ejeAlfa
ptoD = direccion (ejeAlfa + ejeBeta)
```

El punto ptoE, intersección de dos geodésicas, se puede calcular así:

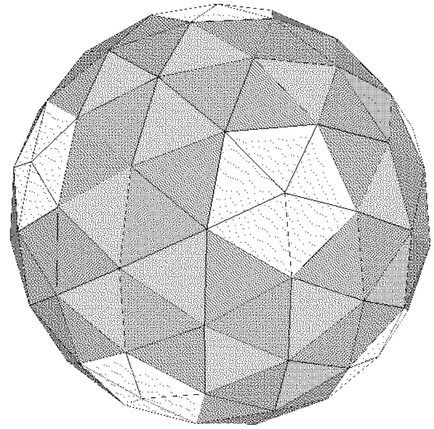

Figura 6: Esfera geodésica.

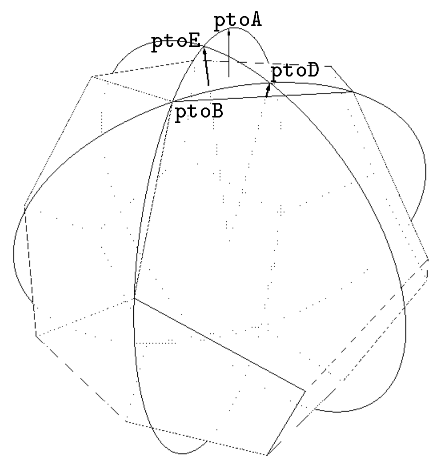

Figura 7: Definición de los vértices de la esfera geodésica.

```
ptoE = direccion (nDE*ejeGama)
  where nDE = rota (-pi/3.0) ptoD (ptoD*ptoB)
```

En esta definición el operador * denota al producto vectorial, por lo que (ptoD*ptoB) es un vector ortogonal a la geodésica que pasa por ptoD y ptoB (Figura 8). Al girar dicho vector un ángulo (-pi/3.0) alrededor del eje ptoD, se obtiene el vector nDE, ortogonal a la otra geodésica mostrada en la figura. Pero el ángulo que forman en ptoD los dos círculos es igual al que forman sus respectivos vectores ortogonales, por lo que la segunda geodésica es la que ha de intersectarse con aquélla que pasa por ptoA y ptoB, para determinar ptoE. Como ejeGama es un vector ortogonal a esta última geodésica, el vector direccion (nDE*ejeGama) es radio común de ambos círculos, igual a ptoE (Figura 9).

Los 12 vértices similares a ptoA se pueden calcular mediante una función parecida a la usada para generar las caras del dodecaedro, pero aplicando las rotaciones (en vez de las permutaciones) a ptoA. La siguiente función sirve para calcular los 30 vértices similares a ptoD:

```
arbol_vertsDs :: Punto -> ArbolVerts
```

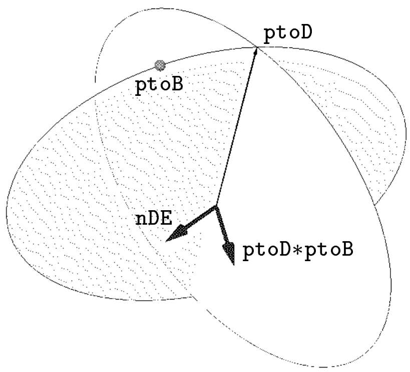

Figura 8: La geodésica ortogonal a `nDE` forma en `ptoD` un ángulo de 60° con aquella que pasa por `ptoB`.

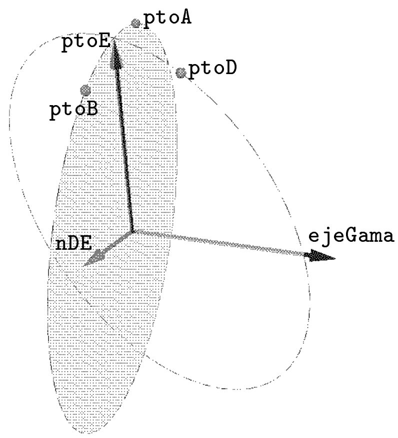

Figura 9: ptoE es la intersección de las geodésicas de respectivos vectores normales `nDE` y `ejeGama`.

```
arbol_vertsDs p =
  iteraArbN 5 rotAlfa
    (Nodo p [iteraArbN 3 rotBeta
               (Nodo (rotBeta p) []),
             Nodo (rotGama p) [],
             Nodo ((rotGama.rotBeta) p) []])
```

En la última línea, el punto denota la composición de funciones. Al aplicarse `arbol_vertsDs` a `ptoD`, la iteración `iteraArbN 5 rotAlfa` recibe como argumento un árbol de seis nodos: `ptoD` como raíz, tres correspondientes a aristas de la cara centrada en `ejeBeta`, otro correspondiente a una arista de la base y el sexto correspondiente a una de las aristas que unen la parte media con la base. La Figura 10 ilustra el árbol así generado.

Los 60 vértices similares a `ptoE` se obtienen mediante una función parecida a la que enseguida se presenta para generar las caras, o bien como se indica adelante, en la subsección 4.1. Una vez calculados los 122 vértices de la esfera geodésica, se juntan en la lista `vertsEG`. Procediendo como en el caso del dodecaedro, para construir las caras de la esfera geodésica primero se definen las permutaciones correspondientes a los generadores:

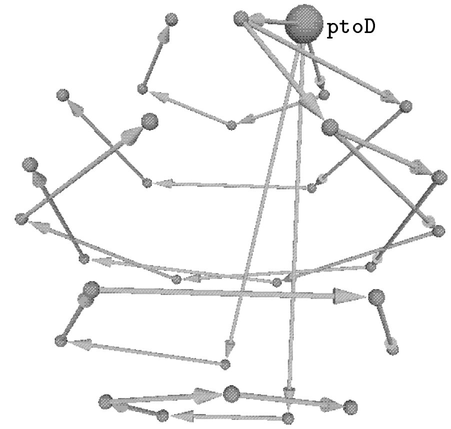

Figura 10: `arbol_vertsDs ptoD`.

```
permAlfaEG,permBetaEG,permGamaEG:: Permutacion
permAlfaEG = permutacion rotAlfa vertsEG
permBetaEG = permutacion rotBeta vertsEG
permGamaEG = permutacion rotGama vertsEG
```

Como se trata de caras triangulares, conviene la nomenclatura siguiente:

```
type Triangulo = Poligono
type ArbolTriangs = ArbolPoligonos
```

Las 240 caras se generan a partir de cuatro triángulos que se definen en términos de los vértices del conjunto soporte. Por cada uno de estos triángulos se construyen 60 caras, mediante una aplicación de la siguiente función:

```
arbol_triangsEG :: Triangulo ->  ArbolTriangs
arbol_triangsEG p =
  iteraArbN 2 pG
    (iteraArbN 5 pA
       (Nodo p [iteraArbN 5 pB
                  (Nodo ((pA.pB) p) [])]))
  where
    pA = map permAlfaEG
    pB = map permBetaEG
    pG = map permGamaEG
```

Así, los triángulos mostrados en la Figura 11 corresponden a los nodos del siguiente árbol, generado a partir del triángulo de vértices `ptoA`, `ptoE` y (`rotAlfa ptoE`):

```
arbol_triangsEG [iA,iE,iE']
  where
    iA = posicion vertsEG ptoA
    iE = posicion vertsEG ptoE
    iE'= posicion vertsEG (rotAlfa ptoE)
```

En la Figura 12 también se pueden apreciar las caras generadas a partir de los triángulos $\triangle$(`ptoE`, `ptoB`, `ptoD`) y $\triangle$(`ptoE`, `ptoD`, `rotAlfa ptoE`).

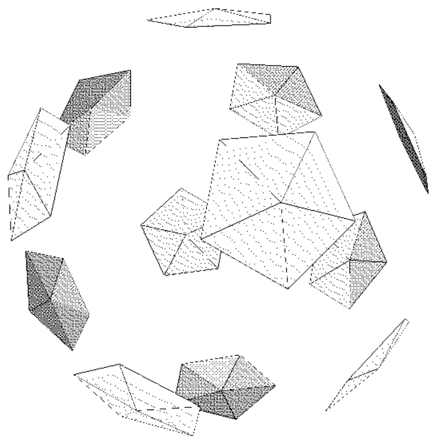

Figura 11: Caras similares al triángulo de vértices `ptoA`, `ptoE` y (`rotAlfa ptoE`).

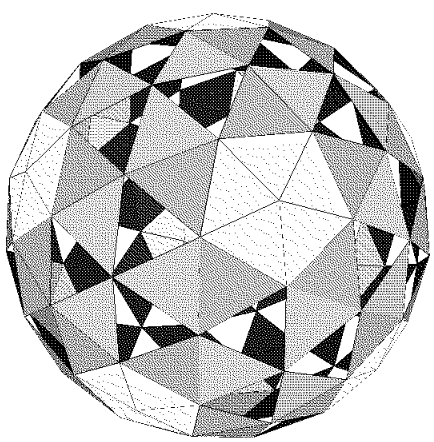

Figura 12: Construcción de las caras de la esfera geodésica.

4 Molécula C_{60}

En la Figura 13 se muestra una representación geométrica de la molécula C_{60}, conocida como fulereno *buckyball* [Curl and Smalley, 1991].

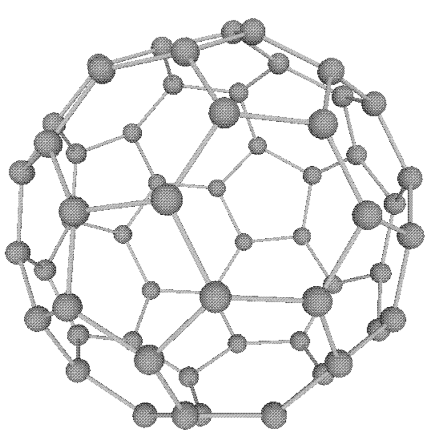

Figura 13: Molécula de C_{60}.

Los 60 átomos de carbono se encuentran distribuidos en una superficie esférica, en las posiciones de los vértices de un poliedro de 12 caras pentagonales y 20 caras hexagonales. Los pentágonos son respectivamente coplanares y concéntricos con las caras de un dodecaedro regular, localizándose los vértices de los primeros sobre las apotemas de las segundas, como se ilustra en la Figura 14.

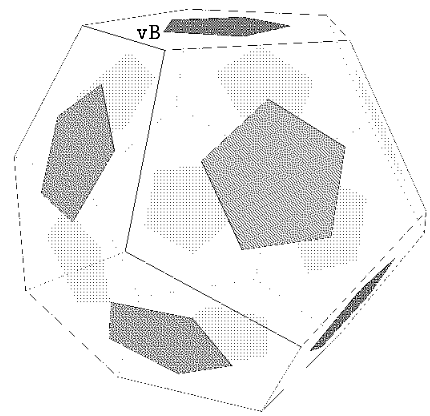

Figura 14: Caras pentagonales del C_{60} y dodecaedro base.

Cada hexágono tiene su centro sobre uno de los segmentos que unen los vértices del dodecaedro con el origen. Esta distribución de los vértices es la misma que tienen los puntos de unión de las costuras de los gajos de un balón de futbol. La misma figura evidencia que el grupo de rotaciones del C_{60} es igual al del dodecaedro, por lo que los vértices del C_{60} también pueden calcularse recurriendo a los generadores usados en los casos anteriores. Un conjunto soporte es el que tiene como único elemento al punto `vB` indicado en la figura, que de los vértices del pentágono superior es el más cercano a la arista frontal izquierda del dodecaedro. Para calcular las posiciones que los vértices del C_{60} ocupan sobre la superficie del dodecaedro de referencia, éste se orienta de la misma manera que el construido en la sección 2. Sin embargo, como dichos vértices también se colocan sobre la esfera unitaria, el dodecaedro aquí usado es de radio mayor que 1.

Dos pentágonos del C_{60} correspondientes a caras contiguas del dodecaedro serán llamados *vecinos*. Sobre las Figuras 15 y 16 se muestra la circunferencia máxima que pasa por `ejeAlfa` y `vB`. Sea ρ el ángulo que estos dos vectores forman. El punto `vB` se obtiene aplicando a `ejeAlfa` una rotación de ángulo ρ, para lo cual primero debe calcularse ρ. Este ángulo es igual a la proyección de un radio del pentágono superior sobre el círculo unitario. Denotando por γ a la correspondiente proyección de una apotema, considerando además que la proporción $\tan\gamma/\tan\rho$ resulta igual a la que guardan la apotema y el radio del pentágono —cociente que sobre un pentágono de radio unitario inmediatamente se muestra igual a $\cos(\pi/5)$— se tiene que

$$\tan\gamma/\tan\rho = \cos(\pi/5)$$

y por lo tanto

$$\rho = \arctan(\tan\gamma/\cos(\pi/5)) \tag{3}$$

Ahora, para calcular γ, sobre las mismas figuras se observa que la circunferencia mostrada puede descomponerse en 10 arcos: cuatro proyecciones (α) de alturas de hexágono, dos proyecciones (β) de segmentos con centros de pentágonos vecinos por extremos y cuatro proyecciones (γ) de apotemas de pentágono. De aquí que

$$2\pi = 4\alpha + 2\beta + 4\gamma$$

de donde se obtiene:

$$\gamma = (\pi - 2\alpha - \beta)/2 \tag{4}$$

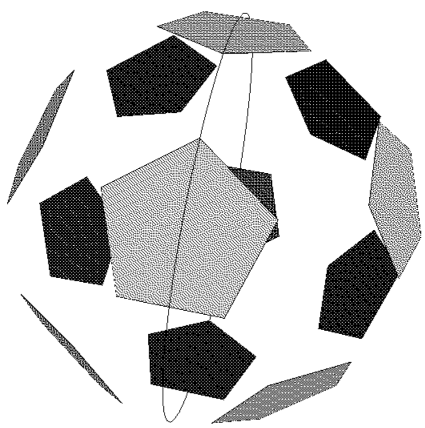

Figura 15: Caras pentagonales del C_{60}.

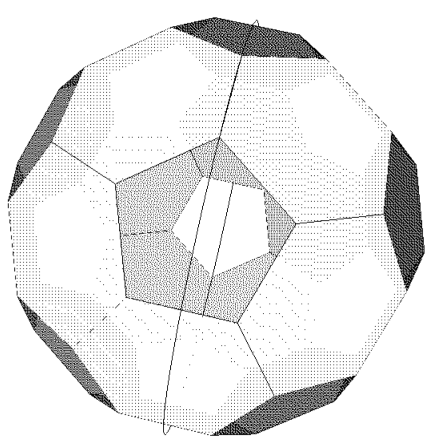

Figura 16: Caras hexagonales del C_{60}.

Sin dificultad se ve que el ángulo α es igual al ángulo entre dos centros de hexágonos vecinos, por lo tanto igual al ángulo entre dos vértices contiguos del dodecaedro y al que forman los centros de dos caras contiguas del icosaedro. Como se puede apreciar sobre las Figuras 2 y 3, una cara de icosaedro es la de vértices P, Q y R, por lo que tiene su centro sobre el rayo $(1+k, 1+k, 1+k)$. Una cara vecina es la de vértices P, R y $(0, -1, k)$, con centro sobre $(k, 0, 1+2k)$. De aquí se obtiene:

$$\cos\alpha = (1+3k)/(3\sqrt{2+3k}). \tag{5}$$

El ángulo β entre dos centros de pentágonos vecinos es el mismo que aparece en (1). Transcribiendo (3), (4), (1) y (5) a fórmulas ejecutables, se obtiene:

```
ro :: Angulo
ro = atan(tan gama/cos(pi/5.0))
  where
    gama = (pi-2.0*alfa-beta)/2.0
    beta = acos(k/(2.0+k))
    alfa =
      acos((1.0+3.0*k)/(3.0*sqrt(2.0+3.0*k)))
```

Por fin se puede calcular `vB`, rotando `ejeAlfa` un ángulo `ro` alrededor de un eje paralelo a la arista del dodecaedro cercana a `vB`:

```
vB = rota ro ejeRot ejeAlfa
  where ejeRot = (sin(pi/5.0),cos(pi/5.0),0.0)
```

Ahora se computan los 60 vértices, dejándolos en la lista `vertsC60`:

```
arbol_vertsC60 :: Punto -> ArbolVerts
arbol_vertsC60 p =
  iteraArbN 2 rotGama
    (iteraArbN 5 rotAlfa
       (Nodo p [iteraArbN 5 rotBeta
          (Nodo ((rotAlfa.rotBeta) p) []) ] ))

vertsC60 = preordArb (arbol_vertsC60 vB)
```

Para generar las caras mostradas en las Figuras 15 y 16, primero se definen las permutaciones del caso:

```
permAlfaC60, permBetaC60, permGamaC60
                                  :: Permutacion
permAlfaC60 = permutacion rotAlfa vertsC60
permBetaC60 = permutacion rotBeta vertsC60
permGamaC60 = permutacion rotGama vertsC60
```

a partir del pentágono superior se construyen las demás caras pentagonales:

```
arbol_pentsC60 :: Pentagono -> ArbolPents
arbol_pentsC60 p =
  Nodo p ([Nodo (pG p) []] ++
          [iteraArbN 2 pG
            (iteraArbN 5 pA (Nodo (pB p) []))])
  where pA = map permAlfaC60
        pB = map permBetaC60
        pG = map permGamaC60

pentsC60 :: [Pentagono]
```

```
pentsC60 = preordArb (arbol_pentsC60 pentSup)
           where
             pentSup = iteraN 5 permAlfaC60 0
```

y las caras hexagonales se generan a partir de la que tiene su centro sobre el radio correspondiente al vértice `ptoB` del dodecaedro:

```
type Hexagono = Poligono
type ArbolHexs = ArbolPoligonos

arbol_hexsC60 :: Hexagono -> ArbolHexs
arbol_hexsC60 p =
  iteraArbN 5 pA
    (iteraArbN 2 pB
       (iteraArbN 2 pG (Nodo p []) ))
  where pA = map permAlfaC60
        pB = map permBetaC60
        pG = map permGamaC60

hexsC60 :: [Hexagono]
hexsC60 = preordArb (arbol_hexsC60 hexB)
          where
            hexB = map (posicion vertsC60)
                       (iteraN 6 rotB vB)
            rotB = rota (pi/3.0) ptoB
```

4.1 Generalización

La comparación de la función generadora de los vértices, `arbol_vertsC60`, con la que en la sección anterior sirve para obtener las caras de la esfera geodésica, `arbol_triangsEG`, sugiere la construcción de la siguiente función, que las generaliza "parametrizando" las funciones usadas para efectuar las rotaciones:

```
arb_p :: (e->e)->(e->e)->(e->e)-> e -> Arbol e
arb_p fA fB fG p =
  iteraArbN 2 fG
    (iteraArbN 5 fA
       (Nodo p [iteraArbN 5 fB
                  (Nodo ((fA.fB) p) [])]))
```

Los tres primeros argumentos de `arb_p` son funciones de tipo genérico `e->e`, el cuarto argumento es de tipo `e`, y sus valores son de tipo `Arbol e`. Las funciones mencionadas se podrían definir en términos de `arb_p` así:

```
arbol_vertsC60 = arb_p rotAlfa rotBeta rotGama
arbol_triangsEG = arb_p (map permAlfaEG)
                        (map permBetaEG)
                        (map permGamaEG)
```

En forma parecida, la siguiente expresión da el árbol de 60 vértices de la esfera geodésica similares a `ptoE`:

```
arb_p rotAlfa rotBeta rotGama ptoE
```

5 Agradecimientos

Agradezco a la DGSCA la hospitalidad brindada para la realización del presente, al IIMAS por el permiso otorgado para trabajar en ella, a Alberto Alonso por animarme a presentar el trabajo en la Conferencia Visual 98 y a Luis Javier Álvarez por la revisión del mismo.

A Algunas funciones

La siguiente función, a la manera de `iterate`, itera `f` en cada nodo de un árbol. Por cada aplicación de `f` a un nodo `x`, como descendiente inmediato de éste se agrega el nodo `(f x)`. Todas las ramas del árbol resultan infinitas.

```
iterateArb :: (e -> e) -> Arbol e -> Arbol e
iterateArb f (Nodo p hs) =
  Nodo p ([iterateArb f (Nodo (f p) [])]
          ++ (map (iterateArb f) hs)  )
```

La siguiente, definida en términos de `iterateArb`, funciona como si se aplicara `(iteraN n f)` a cada nodo de un árbol `a`. La función auxiliar `takeArb` se encarga de podar las ramas infinitas definidas por `iterateArb`, dejando cada una con `n` nodos.

```
iteraArbN :: Int->(e->e)->Arbol e -> Arbol e
iteraArbN n f a = takeArb n (iterateArb f a)
  where takeArb 1 (Nodo p (h:hs)) =
             Nodo p (map (takeArb 1) hs)
        takeArb (n+1) (Nodo p (h:hs)) =
             Nodo p ([takeArb n h]
                     ++(map (takeArb (n+1)) hs))
```

La siguiente función genera la lista de lados de un árbol, recorriéndolo en preorden.

```
ladosArb :: Arbol e -> [[e]]
ladosArb (Nodo p []) = []
ladosArb (Nodo p ((Nodo h hhs):hs)) =
  [ [p,h] ] ++ ladosArb (Nodo h hhs)
          ++ ladosArb (Nodo p hs)
```

Las siguientes definen, respectivamente, el producto vectorial, la norma y la dirección de vectores en `R3`.

```
instance Num R3 where
 (*) (x1,y1,z1) (x2,y2,z2) =
        (y1*z2-y2*z1,z1*x2-z2*x1,x1*y2-x2*y1)

norma :: R3 -> R
norma (x,y,z) = sqrt (x^2+y^2+z^2)

direccion :: R3 -> R3
direccion p = k ** p
              where k=1.0/(norma p)
```

B Ejemplo de archivo VRML

Para graficar dodecaedros se produjo un archivo de contenido esencialmente igual al siguiente:

```
#VRML V2.0 utf8
  #Dodecaedro prototipo
  #Generado por dodecaedro.gs
PROTO Vertices []
```

```
  {Coordinate
    {point
      [
       0.607062 0.0 0.794655
       0.187592 0.57735 0.794655
       -0.491123 0.356822 0.794655
       -0.491124 -0.356822 0.794655
       0.187592 -0.57735 0.794655
       0.982247 2.98023e-08 0.187593
       0.303531 0.934172 0.187593
       -0.794655 0.57735 0.187593
       -0.794655 -0.57735 0.187593
       0.303531 -0.934173 0.187593
       -0.607062 0.0 -0.794654
       -0.187593 -0.57735 -0.794654
       0.491123 -0.356822 -0.794654
       0.491124 0.356822 -0.794654
       -0.187592 0.57735 -0.794654
       -0.982247 -1.19209e-07 -0.187593
       -0.303531 -0.934173 -0.187593
       0.794655 -0.57735 -0.187593
       0.794654 0.57735 -0.187593
       -0.303531 0.934172 -0.187593
      ]}}
PROTO Caras
  [field MFInt32 indices []
   exposedField SFColor color 1 1 1
   exposedField SFFloat transparencia 0
  ]
  {Shape {
     geometry IndexedFaceSet {
       coord Vertices{}
       coordIndex IS indices}
     appearance Appearance {material Material{
       diffuseColor IS color
       transparency IS transparencia}}
  }
}
  ##### El dodecaedro #####
PROTO Dodecaedro
  [exposedField SFRotation eje_angulo 0 1 0 0
   exposedField SFVec3f escala 1 1 1
   exposedField SFColor colorAnv 0.2 0.7 0.8
   exposedField SFColor colorRev 0.0 0.2 0.4
   exposedField SFFloat transpAnv 0
   exposedField SFFloat transpRev 0]
{Transform{rotation IS eje_angulo
           scale IS escala
children[
  Caras{
    indices[
      0 1 2 3 4 -1
      10 14 13 12 11 -1
      5 0 4 9 17 -1
      15 10 11 16 8 -1
      6 1 0 5 18 -1
      19 14 10 15 7 -1
      7 2 1 6 19 -1
      18 13 14 19 6 -1
      8 3 2 7 15 -1
      17 12 13 18 5 -1
      9 4 3 8 16 -1
      16 11 12 17 9 -1
    ]
    color IS colorAnv
    transparencia IS transpAnv
  }
]}}
  ## Ejemplo de uso del prototipo ##
Dodecaedro{colorAnv 1.0 1.0 1.0
           transpAnv 0.18}
```

Referencias

[Bird and Wadler, 1987] R. Bird and P. Wadler. *Introduction to Functional Programming*. Prentice Hall, 1987.

[Curl and Smalley, 1991] R.F. Curl and R.E. Smalley. Fullerenes. *Scientific American*, 265(4):32–41, October 1991.

[Davie, 1992] A.J.T. Davie. *An Introduction to Functional Programming Systems Using Haskell*. Cambridge University Press, 1992.

[Hudak and Fasel, 1992] P. Hudak and J.H. Fasel. A gentle introduction to Haskell. *ACM SIGPLAN Notices*, 27(5):T1–T53, May 1992.

[Jones, 1991] M.P. Jones. Gofer. Functional programming environment (v. 2.20). Technical report, 1991.

[Launchbury, 1995] John Launchbury. Graph algorithms with a functional flavour. In J. Jeuring and E. Meijer, editors, *Advanced Functional Programming*, number 925 in Lecture Notes in Computer Science, pages 308–331. Springer-Verlag, 1995.

[Peterson *et al.*, 1997] J. Peterson, K.H̃ammond (editors), and other 13 authors. Report on the programming language Haskell, a non-strict purely functional language (version 1.4). Technical report, Yale University and University of St. Andrews, April 1997. http://haskell.org.

[Strooper and van Emden, 1992] P.A. Strooper and M.H. van Emden. Describing symmetrical structures in logic. Technical report, Department of Computer Science, University of Victoria, February 1992. Report LP-20.

[VRML97, 1997] VRML97. The VRML97 specification, VRML/ISO draft for internationals standard. http://www.vrml.org/Specifications/VRML97/index.html, 4 April 1997.