

# Computación Gráfica - TP: Subdivision Surfaces

## 1. Resumen de tareas

1. Entender la estructura de datos utilizada para representar la malla (clases `Nodo`, `Elemento` y `SubDivMesh`, definidas en `SubDivMesh.hpp`).
2. Implementar el algoritmo de *Catmull-Clark* de subdivisión de superficies (función `subdivide` en `main.cpp`).

## 2. Consigna detallada

### 2.1. Estructuras de datos

El código inicial del TP carga una malla de triángulos y/o cuadriláteros en una estructura de datos ad-hoc, representada por la clase `SubDivMesh`. A diferencia de las estructuras más simples que se venía utilizando en prácticos anteriores, en esta estructura hay información adicional sobre la topología y las conectividades:

- La malla (`class SubDivMesh`) contiene una lista de nodos (`std::vector<Nodo> n`) y una lista de elementos (`std::vector<Elemento> e`).
- Cada elemento (`class Elemento`) puede ser un triángulo o un cuadrilátero (lo cual se define con el atributo `nv`, "número de vértices", según su valor sea 3 o 4).
- Además de los índices<sup>1</sup> de los 3 o 4 vértices que forman el elemento (`int n[]`), el elemento tiene también información sobre cuales son los 3 o 4 elementos "vecinos" (`int v[]`).
  - Un elemento A es vecino de B si comparten una arista. El arreglo `v` está ordenado de igual forma que `n`. Esto quiere decir que `v[i]` tiene el índice del vecino que comparte la arista que va de `n[i]` a `n[i+1]`<sup>2</sup>.
- Cada nodo (`class Nodo`) tiene, además de sus coordenadas (`glm::vec3 p`), una lista índices de elementos a los que pertenece (`std::vector<int> e`), y un booleana (`bool es_frontera`) que indica si está en una frontera.
  - Un nodo frontera es un nodo que está en una arista del borde<sup>3</sup> de la malla (que no tiene vecino).

Al modificar los elementos, se debe mantener en todo momento actualizado el arreglo `e` de cada nodo. Por ej, si se agrega un nuevo elemento, hay que registrar su índice en todos los nodos que lo componen. Para asegurarse de no omitir este paso y dejar la malla en un estado inconsistente, evite modificar directamente el arreglo de elementos de la malla, y utilice en cambio los métodos `agregarElemento` y `reemplazarElemento` que se aseguran de hacer ese "mantenimiento".

El otro "mantenimiento" que podría ser necesario hacer al modificar los elementos es el de las relaciones de vecindad. En general es más complicado contemplarlo en cada modificación; por ello la clase permite no hacerlo con cada pequeña modificación, sino que provee un método `makeVecinos` que recalcula todas las vecindades









<sup>1</sup>Notar que el elemento guarda los "índices" de los nodos, y no a los nodos mismos (sus coordenadas). Entonces, para acceder a un nodo (las coordenadas y su info adicional) se debe utilizar el índice con el vector de nodos de la malla. Por ej, si la malla es `m`, y tenemos un elemento `e1`, el primer `Nodo` del elemento sería `m.n[e1.n[0]]`

<sup>2</sup>El índice en realidad debería ser `(i+1)%nv`, para que luego el último nodo el siguiente sea otra vez el primero. La clase elemento cuenta con sobrecargas para el operador `[]` que realizan esta operación (en realidad `(i+nv+1)%nv`). Entonces, suponiendo que `e1` es un cuadrilátero (los índices válidos serían 0, 1, 2 y 3), se puede acceder por ej. a `e1[3]` (retornaría lo mismo que `e1[0]`) o a `e1[-1]` (retornaría lo mismo que `e1[4]`).







<sup>3</sup>Las mallas cerradas (como un cubo a una esfera) no tienen frontera (todas sus aristas son compartidas por 2 elementos). Entre los ejemplos del tp, la plano formado por un triángulo y un cuadrilátero tiene inicialmente todos sus nodos en la frontera; y la mona *Suzanne* tiene vértices de frontera en los ojos (los ojos propiamente dichos son fragmentos de mallas separados del resto de la cabeza).

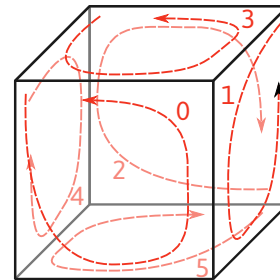
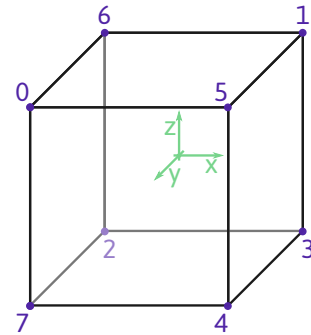
actualizando todos los arreglos `v` de todos los elementos. Lo mismo ocurre con el flag de frontera, y es el mismo método, `make_vecino`, el que actualiza el valor de `es_frontera` de todos los nodos.

### 2.1.1. Ejemplos

	x	e	frontera
	p[0] = { {-1, +1, +1},	{0, 3, 4},	false }
	p[1] = { {+1, -1, +1},	{1, 2, 3},	false }
	p[2] = { {-1, -1, -1},	{2, 4, 5},	false }
	p[3] = { {+1, -1, -1},	{1, 2, 5},	false }
	p[4] = { {+1, +1, -1},	{0, 1, 5},	false }
	p[5] = { {+1, +1, +1},	{0, 1, 3},	false }
	p[6] = { {-1, -1, +1},	{2, 3, 4},	false }
	p[7] = { {-1, +1, -1},	{0, 4, 7},	false }

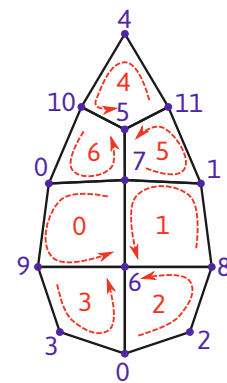
MakeVecinos()

	nv	n	v
	e[0] = { 4, {0, 7, 4, 5}, {4, 5, 1, 3} }		
	e[1] = { 4, {1, 5, 4, 3}, {3, 0, 5, 2} }		
	e[2] = { 4, {3, 2, 6, 1}, {5, 4, 3, 1} }		
	e[3] = { 4, {6, 0, 5, 1}, {4, 0, 1, 2} }		
	e[4] = { 4, {0, 6, 2, 7}, {3, 2, 5, 0} }		
	e[5] = { 4, {3, 4, 7, 2}, {1, 0, 4, 2} }		



	nv	n	v
e[0]	= { 4, {0, 7, 4, 5}, {4, 5, 1, 3} }		
e[1]	= { 4, {1, 5, 4, 3}, {3, 0, 5, 2} }		
e[2]	= { 4, {3, 2, 6, 1}, {5, 4, 3, 1} }		
e[3]	= { 4, {6, 0, 5, 1}, {4, 0, 1, 2} }		
e[4]	= { 4, {0, 6, 2, 7}, {3, 2, 5, 0} }		
e[5]	= { 4, {3, 4, 7, 2}, {1, 0, 4, 2} }		

	x	e	frontera
p[ 0 ]	= { { ... }, { 0, 3, 4 },	true	}
p[ 1 ]	= { { ... }, { 1, 2, 3 },	true	}
p[ 2 ]	= { { ... }, { 2, 4, 5 },	true	}
p[ 3 ]	= { { ... }, { 1, 2, 5 },	true	}
p[ 4 ]	= { { ... }, { 0, 1, 5 },	true	}
p[ 5 ]	= { { ... }, { 0, 1, 3 },	false	}
p[ 6 ]	= { { ... }, { 2, 3, 4 },	false	}
p[ 7 ]	= { { ... }, { 0, 4, 7 },	false	}
p[ 8 ]	= { { ... }, { 0, 1, 3 },	true	}
p[ 9 ]	= { { ... }, { 2, 3, 4 },	true	}
p[10]	= { { ... }, { 0, 4, 7 },	true	}
p[11]	= { { ... }, { 0, 1, 3 },	true	}

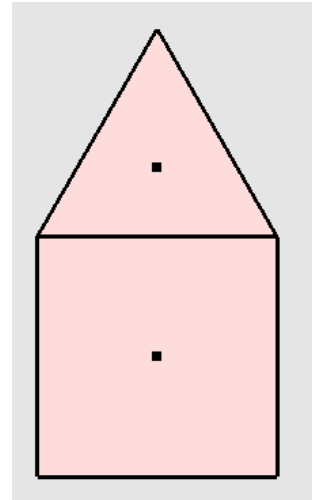
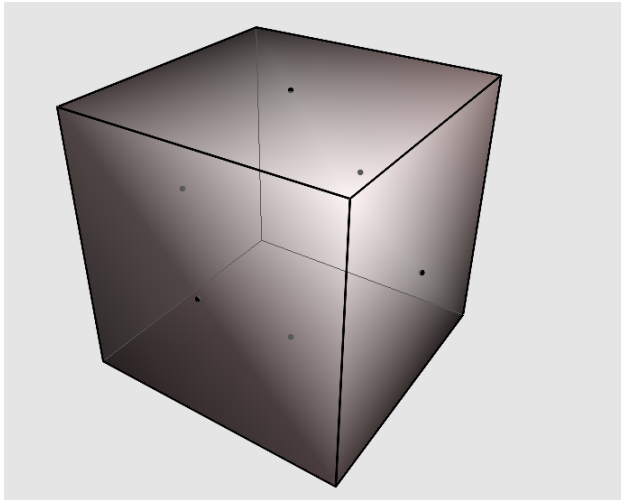


## 2.2. Algoritmo de *Catmull-Clark*

Debe implementar el algoritmo de subdivisión de *Catmull-Clark* en el método `void subdivide(SubDivMesh &mesh)` (en `main.cpp`).

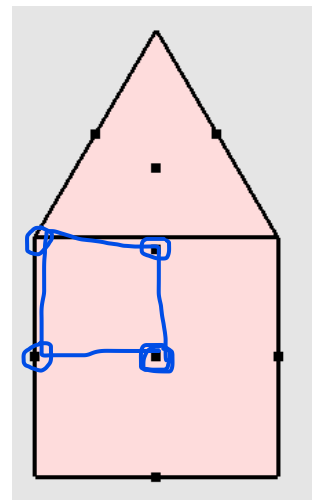
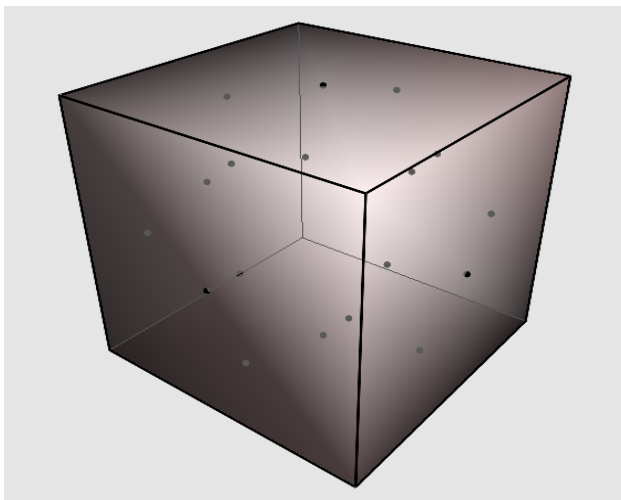
Se sugiere implementar el algoritmo en 4 pasos:

1. Por cada elemento original, generar un nuevo nodo en la posición del centroide (promedio de los nodos del elemento).



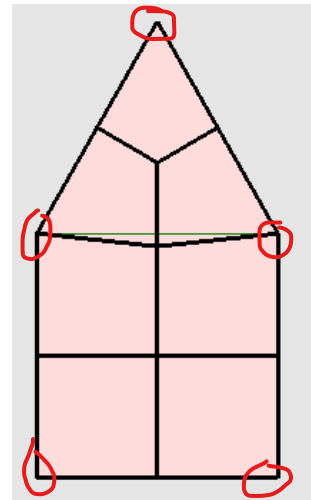
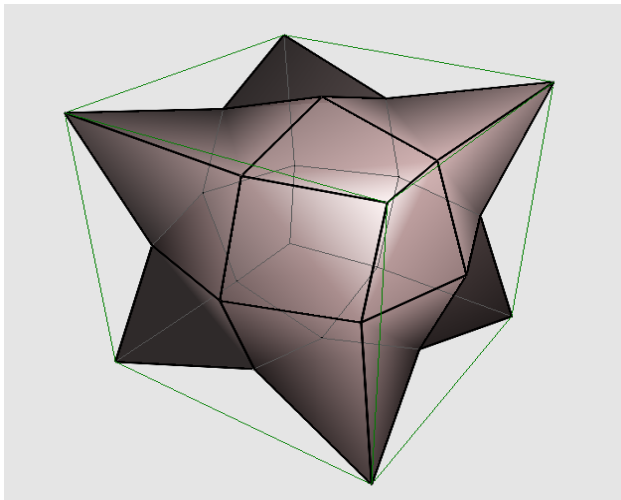
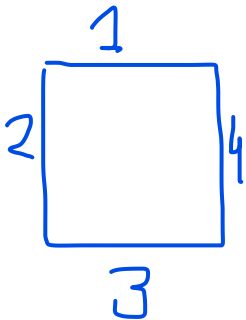
2. Por cada arista, agregar un nodo asociado a la misma.

- No hay un vector de aristas. La forma de recorrer todas las aristas es recorrer todas las caras, y por cada cara recorrer sus aristas. Pero entonces se pasa 2 veces por cada arista interior (que la comparten dos caras), por lo que se necesita algún mecanismo para detectar cuando esto ocurre (para ello el comentario del código sugiere utilizar un mapa).
- El nodo a agregar será:
  - si es frontera (si el elemento tiene vecino -1 para esa arista): el promedio de los nodos de la arista,
  - si es interior: el promedio entre los vértices de la arista y los centroides de sus dos elementos .



3. Generar los nuevos elementos. Por cada elemento original, la nueva malla tendrá  $n_v$  (3 si es triángulo, 4 si es cuadrilátero) cuadriláteros nuevos<sup>4</sup>.

- No agregue directamente todos los elementos nuevos a la malla. Utilice uno de ellos para reemplazar al original, de modo que en la malla final solo queden los nuevos.
- Cada elemento nuevo se forma conectando el centroide de un elemento original, los nodos asociados a dos aristas consecutivas, y el vértice que comparten esas aristas.
  - Para encontrar el índice del centroide de un elemento, puede notar que si la malla original tenía  $N$  nodos (cuyos índices van de 0 a  $N-1$ , entonces los centroides en el arreglo de nodos de la malla comienzan en el índice  $N$  (en  $N$  está el del 1er elemento, en  $N+1$  el del segundo, en  $N+2$  el del tercero, etc).
  - No hay una fórmula simple para encontrar el nodo asociado a una arista. Por ello se recomienda registrar el índice en un mapa<sup>5</sup> al generarlo (o alternativamente en una matriz donde la fila sea el número de elemento, y la columna el de arista).
- Se sugiere ordenar los nodos dentro del nuevo elemento con algún criterio que luego le permita saber cuál era cada uno (por ej, poner siempre primero al que era centroide). Esto simplificará el paso 4.
- Dado que en este paso cambian las conectividades, la información de vecinos y frontera quedará desactualizada. Recuerde reconstruir esa información `makeVecinos` para evitar problemas en pasos o subdivisiones posteriores.



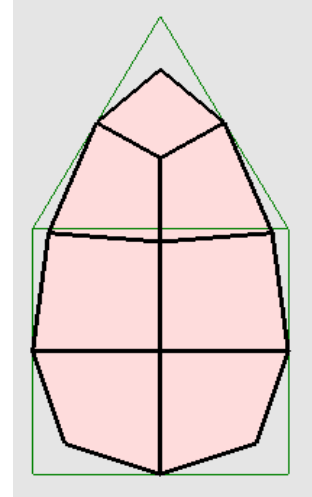
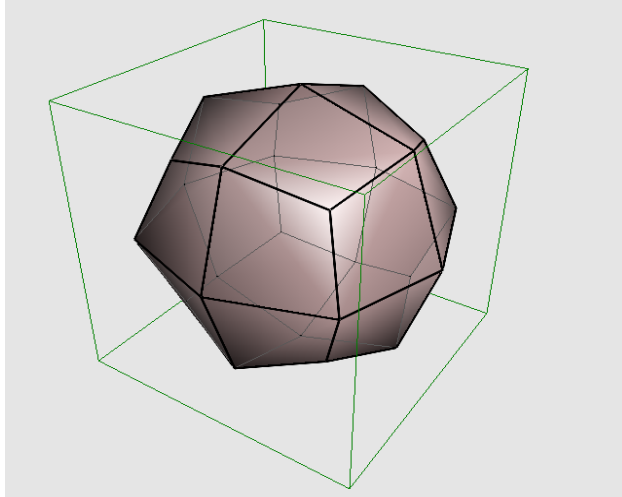
4. Calcular las nuevas posiciones para los nodos originales (no mover los que se agregaron en los pasos 1 y 2).

- La nueva posición debe ser:
  - Para nodos interiores:  $(4r - f + (n - 3)p)/n$ , donde
    - \*  $f$  es el promedio de los centroides de sus caras originales (los agregados en el paso 1)
    - \*  $r$  es el promedio de los pts asociados a sus aristas de las aristas (los agregados en el paso 2)
    - \*  $p$  es la posición original del nodo
    - \*  $n$  es la cantidad de elementos para ese nodo
  - Para nodos del borde:  $(r + p)/2$ , donde

<sup>4</sup>una de las ventajas de este método de subdivisión es que genera siempre elementos del mismo tipo (solo cuadriláteros) sin importar de qué tipo sean los elementos de partida.

<sup>5</sup>El main define el tipo Mapa como equivalente a un mapa estándar donde la clave es un instancia de Arista (su particularidad es que no importa el orden de sus nodos al comparar dos Aristas), y el valor un entero (índice del nodo asociado). Si declara un Mapa `m`, para asociar un índice (`i`) de nodo a una arista (`n1` a `n2`) puede utilizar `elmapa[Arista(n1,n2)]=i`, para saber si hay un índice asociado a una arista `elmapa.find(Arista(n1,n2))!=elmapa.end()`, y para recuperar el índice (`j`) asociado a una arista `int j=elmapa[Arista(n1,n2)]`.

- \*  $r$  es el promedio de los dos pts asociados a sus dos aristas (paso 2)
  - \*  $p$  es la posición original del nodo
- Notar que a esta altura del algoritmo ya no existen los elementos originales, sino que el arreglo de elementos debería contener solo los nuevos.



Se sugiere intentar primero subdividir el cubo, ya que es cerrado (no hay aristas/nodos de frontera) y solo tiene cuadriláteros. Luego la pirámide, ya que también es cerrada, pero tiene elementos diferentes (triángulos). Luego el ejemplo plano, ya que tiene fronteras. Y finalmente, si los tres ejemplos anteriores funcionan, debería funcionar correctamente el de *Suzanne*, que contiene todas las variantes anteriores juntas.

