EL ESPACIO DE LA IMAGEN

Aquí estudiaremos algunos métodos de trabajo que producen resultados visuales, pero a través del manejo de la información gráfica *per-píxel*, en lugar de manejar información geométrica *per-vertex*. En forma equivalente, se habla de métodos de imagen o *image-pecision* o *raster* en contraposición a los métodos geométricos o *model-precision* o *vector*.

Se suele distinguir un fragmento de un píxel. Originalmente, un píxel es una porción elemental de imagen; pero una imagen se almacena como un arreglo rectangular de WxH colores; por lo cual se generaliza el uso de la palabra "píxel" para una unidad en el espacio de almacenamiento (*buffer*) que contiene tres colores, un valor de alpha y otros valores asociados. Durante el procesamiento en el *pipeline* gráfico, las rutinas de rasterización muestrean valores de color y otros en cada fragmento de primitiva que corresponde a la posición de un píxel. Un fragmento puede resultar descartado o puede ser almacenado en el pixel, mezclando o no sus valores con los que tenía el pixel. Ambos (fragmento y píxel) contienen valores asociados a una determinada porción elemental (que se considera cuadrada y unitaria) de una imagen de WxH píxeles.

La cantidad de memoria disponible para cada píxel se suele denominar "profundidad" (*depth*), como si fuese una pila con un "plano" por cada bit (*bitplane*). La utilización de gráficos en computación comenzó con un bit por píxel, en los "*bitmaps*" que eran imágenes en blanco y negro.

Para simular los grises se usaba el *dithering* o puntillado, que aún puede verse con lupa en los diarios, los *comics* o en los carteles callejeros de cuatro colores. Las tintas no cambian su intensidad con la cantidad, poca tinta azul o mucha tinta azul da color azul. El efecto se logra puntillando: un área determinada se cubre parcialmente; vista desde lejos, la mezcla del azul y el fondo blanco atenúa el azul puro. La superposición de capas que hace la industria gráfica editorial, se trasladó, al menos en la jerga, a la computación gráfica. La ventaja del monitor es que la intensidad de cada color sí puede variarse, pero el problema estaba en el alto precio de la memoria.

Del blanco y negro (peor: ¡ámbar o verde! negro era más caro) se pasó a 16 y rápidamente a 256 colores organizados en una "paleta" (*color palette*) como la del pintor; que aún conservan las imágenes gif.

En OpenGL quedan los resabios de la manipulación de *bitmaps* que se suelen utilizar aún para los *raster fonts* o fuentes de tamaño fijo (a diferencia de los *true type fonts*) y de las paletas con *color index* en lugar de RGB, que ya no se utiliza casi para nada en OpenGL, pero sí en los gif animados.

Actualmente se utilizan 24bpp (bits por pixel) de profundidad para RGB (32 para RGBA). En la teoría del color vimos el exceso de llamar *true color* a esta paleta muy densa, que no cubre todos los colores visibles. Con 24 bits de color podemos manipular 8 para la intensidad de cada color, rojo, verde y azul. Con 8 bits, cada color varía su intensidad entre 0 y 255, aumentando de a uno; pero internamente, se almacena en formato *float* entre 0 y 1, aumentando de a 1/255. OpenGL especifica muchos otros modos posibles de almacenamiento de los colores (*image format*).

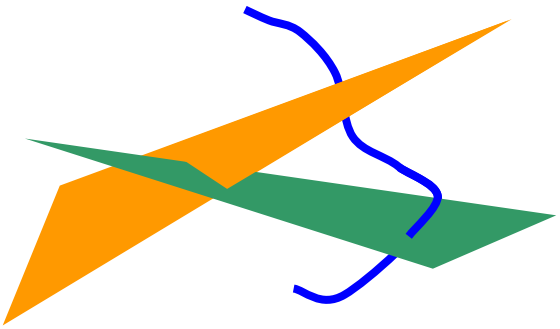
En todos los sistemas gráficos hay varios *buffers* gráficos "paralelos" al color, normalmente utilizan la memoria de la placa gráfica, pero a veces utilizan memoria RAM del PC, teléfono o consola de juegos.

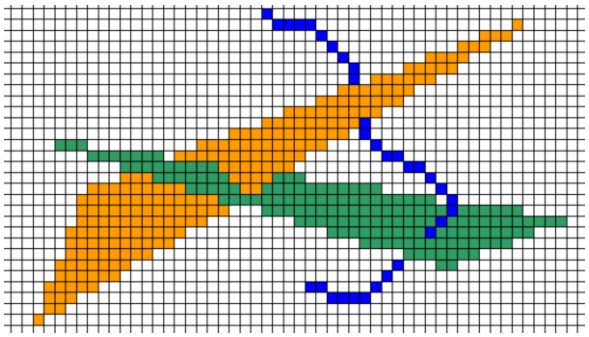
Primero veremos el *z-* o *depth-buffer*, como ejemplo introductorio; para una generalización posterior.

z-buffer

Al renderizar varios objetos en 3D, algunos tapan visualmente a otros. Hay varios métodos geométricos (*vector*) para resolver los problemas de oclusión visual, consisten en calcular intersecciones y dividir los objetos, para luego renderizar desde el fondo, de acuerdo a la distancia al observador (algoritmo del pintor). Pero aun cuando existen muchísimas técnicas que aceleran los cálculos y el ordenamiento, el problema es la velocidad. Cuando se dispone de suficiente memoria y hardware especializado para la rasterización (GPU), se utiliza el algoritmo *raster* basado en el *z-buffer* o *depth-buffer*. Básicamente consiste en "pintar" el píxel sólo si la distancia del fragmento al ojo (*z*) es menor que la almacenada en el *z-buffer*; en tal caso se actualiza además ese *buffer* con el valor de *z* del fragmento recién calculado.

El *color-buffer* almacena WxH píxeles de color, es el *buffer* que se actualiza cuando un fragmento es visible. El *depth-buffer* se mantiene en paralelo y consiste en otros WxH valores de punto flotante, normalmente de 24 o 32 bits; se muestra, a veces, como una imagen en escala de grises.





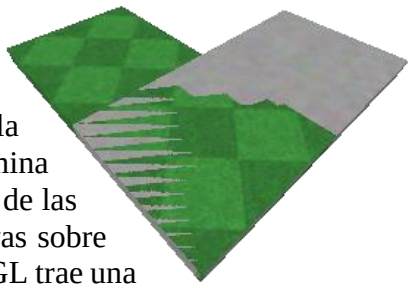
El algoritmo fue desarrollado por Ed Catmull (uno de los fundadores de Pixar) en 1974 y es algo así:

Inicializar el *z-buffer* con el *z* máximo, el del *far-plane*: $\forall\{x,y\}\in[0,W)x[0,H) \Rightarrow depth(x,y) = zfar$
Para cada primitiva dentro del espacio visual:
Rasterizar calculando *z* (la "profundidad" del fragmento entrante, su distancia al ojo).
Si $z < depth(x,y)$
Actualizar el *color-buffer* (pintar: $color(x,y)$ = RGBA del fragmento entrante)
Actualizar el *z-buffer* ($depth(x,y) = z$ del fragmento entrante)

En realidad, no se utiliza el *z* visual entre *znear* y *zfar*, sino un "z de la imagen", que depende del visual en forma no-lineal pero monótona ($z_i = a+b/z_v$) producido por la matriz de proyección; luego convertido al rango [0,1] y almacenado con la precisión numérica disponible; es decir que es aproximado.

Este es el algoritmo que utilizan OpenGL y la mayoría de las APIs gráficas a partir del abaratamiento de la memoria. Es muy rápido, eficiente y general, no se calcula ninguna intersección, no requiere ningún ordenamiento de los objetos ni subdivisión del espacio porque *rasteriza* todas las primitivas del volumen visual en cualquier orden. Pero adolece de algunos defectos que pueden ser importantes:

1) No sirve para manejar las transparencias múltiples: si el objeto es semitransparente, hay que combinar el color del fragmento con el color del píxel almacenado detrás. Pero, si aparece un tercer objeto, intermedio y también semitransparente, ya no se dispone de la información necesaria para calcular bien el color resultante. Cuando hay transparencias múltiples, el resultado depende el orden de rasterización. Una técnica utilizada consiste renderizar primero los objetos opacos y luego los (pocos) semitransparentes, pero ordenados desde los más lejanos a los más cercanos al ojo.



2) Precisión: Con 24 bits, los valores alojados, que son reales $\in [0,1]$, se cuantizan: $z_{alojado} = int((2^{24}-1)\, z_{calculado})/(2^{24}-1)$[1]. Debido a ello, los planos coincidentes (o casi) suelen provocar defectos visuales. Esto obliga a ajustar con precisión los planos *near* y *far* para aumentar la precisión del resultado, que aun así se verá mal. Este defecto se denomina *z-fighting* (pelea de *z*) y es muy visible por la mezcla ruidosa del color de las piezas. También aparece cuando se dibujan los bordes de las primitivas sobre las mismas (*wireframe* y *filled* superpuestos). Para resolver esto, OpenGL trae una función (glPolygonOffset) que perturba el *z-buffer* cuando se activa (miente el *z*).

3) No es posible aplicar *antialiasing* por fragmento: Cuando un píxel está formado por varias fuentes distintas (bordes) el *antialiasing* se realiza pintando el píxel con un promedio ponderado (*coverage*) de las fuentes. El *z-buffer* sólo dispone del fragmento entrante y el pixel alojado.

4) El algoritmo de *z-buffer* requiere *rasterizar* muchas primitivas (todas las que están dentro del volumen a visualizar) y realiza múltiples accesos no secuenciales a la memoria (*cache misses*). Para escenas especialmente complejas se puede combinar con distintas técnicas de ordenamiento y/o recorte (*clipping*) y descarte masivo (*culling*) de objetos o grupos de objetos invisibles, tanto para reducir la labor como para optimizar el uso de la memoria (*cache*). La más utilizada es el *face-culling*: dado un objeto (*b-rep*) cerrado, se descartan las caras cuya normal no apunta al ojo.

[1] Supongamos reales entre 0 y 1 de dos bits, los únicos valores posibles son: 0, 1/3, 2/3 y 1. Si quiero alojar 0.42, la máquina aloja: $int(0.42*(2^2-1))/(2^2-1) = int(0.42*3)/3 = 1/3$
Se pueden ver más detalles y matemática en el *RedBook* o en *Wikipedia*: http://en.wikipedia.org/wiki/Z-buffering

El *frame-buffer*

OpenGL trabaja con varios *buffers* o porciones de memoria, ya conocemos el *color-buffer*, donde se arma la imagen que vemos y el *depth-buffer*, que sirve para resolver las oclusiones visuales. Internamente no son más que espacios de memoria con valores asignados a los píxeles. Hay varios *buffers* y en conjunto conforman el *framebuffer*.

Los valores, ya sean individuales (píxel) o en bloques (pedazos de imagen) pueden leerse, manipularse y sobrescribirse desde el programa; OpenGL trae además algunas rutinas propias de manipulación.

Los *buffers* clásicos de OpenGL son los siguientes:

- *Color buffer"**s**": front-left*, *front-right*, *back-left*, *back-right* y algunos otros *auxiliary buffers*
- *Depth buffer*
- *Stencil buffer*
- *Accumulation buffer*

Se puede ver que hay más de un *buffer* de color. El *front* y el *back* se utilizan para *renderizar* con la técnica de *double-buffering*: se *renderiza* en *background* mientras se visualiza el *front*, cuando la imagen está completa se intercambian los roles de ambos *buffers* (*swap buffers)*, de esa forma se evita el parpadeo (*flickering*) que se ve en las animaciones con un solo *buffer*. *Left* y *right* son dos juegos de *buffers* de color utilizados para estereoscopía, en cada uno se genera la imagen para cada ojo. El resto de los *buffers* de color (*aux*) se pueden utilizar como *layers* o capas o también para el almacenamiento temporario de imágenes accesorias generadas.

Un sencillo truco con el *color-buffer* consiste en averiguar el color del píxel que hay "debajo" del cursor para saber si está sobre un objeto o sobre el fondo. Si se pretende saber sobre que objeto en particular está el cursor, hay que utilizar un color fijo distinto para cada uno. Si se utiliza iluminación, el color es variable; pero aun así se puede hacer un renderizado invisible en un *buffer* auxiliar (normalmente en el *back* y sin *swappear*) sin iluminación y con un color fijo designado para cada objeto.

Manipulando el *z-buffer* también se logran algunos trucos interesantes. Un ejemplo es la determinación de siluetas buscando altos gradientes de *z* entre píxeles vecinos. Además, el *z-buffer* se puede declarar *read-only* (no se actualiza) o se puede cambiar la función que compara los valores de *z* entrante y almacenado; con ello pueden hacerse varios pasos de *renderizado* para dibujar primero las líneas invisibles de una manera y, en el siguiente paso, las visibles de otro modo. Otro truco estándar es la proyección de sombras, se basa en una manipulación del *depth-buffer* que se genera al mirar la escena desde el foco de luz: los objetos y porciones visibles desde la luz son los que están iluminados. Otra aplicación permite conocer la posición del punto del modelo picado: el cursor define *x* e *y* de una línea que atraviesa el modelo, pero el z del punto 3D picado, el más cercano al ojo, puede conocerse leyendo el *z-buffer* en ese píxel, solo resta (anti)transformar al sistema del modelo (gluUnProject).

El *stencil buffer* se utiliza para muchísimos trucos y lo veremos en detalle al final. El nombre proviene del uso original, que consiste en enmascarar una zona de dibujo. Por ejemplo, en un juego de carreras de autos, el parabrisas sirve de máscara, para *renderizar* adentro toda la escena exterior; alrededor habrá partes fijas del auto y algunas variables, como el volante y algunos indicadores del tablero; los espejos definen otras máscaras para renderizar las vistas traseras. Otro uso muy interesante es para renderizar sólidos mediante las operaciones booleanas de un CSG *tree*. Hay ejemplos e instrucciones en la web.

El *buffer* de acumulación se utiliza para superponer imágenes, como si fuesen distintas manos de pintura con efecto acumulado sobre una misma superficie. Sirve por ejemplo para hacer el borroneado por movimiento (*motion blur*) dando la impresión de velocidad; para hacer *antialiasing*; para simular la profundidad de campo fotográfico (*depth of field*) o para hacer sombras blandas (*soft shadows*) que es el término usual para referirse a la penumbra: los bordes suaves de las sombras provocadas por fuentes luz extensas, no puntuales. El *buffer* de acumulación no puede manipularse píxel a píxel como el resto, se le pasa (varias veces) un buffer de color entero (atenuado o no) y el resultado de las operaciones se transfiere entero al buffer de color. Este *buffer* suele tener más bits disponibles que el de color; por lo tanto, las operaciones de acumulación se pueden hacer con más precisión y se "redondean" al transferir.

La presencia o no de los distintos *buffers* y la cantidad de bits disponibles dependen del hardware en cuestión. Se solicitan al inicializar y luego se consulta (glGet) cuantos bits hay disponibles.

Los *buffers* se escriben, borran y enmascaran mediante sencillas reglas que pueden verse en el *Red Book* u otro manual de OpenGL o DirectX. Lo que se debe saber es que los datos pueden manipularse logrando una miríada de efectos muy útiles; trabajando directamente con los píxeles y fragmentos, en modo *raster*.

Una imagen, parcial o completa, puede leerse desde un *buffer* como un rectángulo de píxeles; por el contrario, una imagen leída de un archivo u otro buffer, puede escribirse en forma directa sobre algún *buffer* o una porción del mismo. Las sentencias para hacerlo son glReadPixels y glWritePixels. OpenGL posee un *pipeline* especial para realizar eficientemente las operaciones costosas de *blitting* (*blit ~ block image transfer*), esto es lectura y escritura de rectángulos de píxeles contenidos en algún buffer.

Operaciones sobre fragmentos:

Después de rasterizar (generar un fragmento) y antes de escribir algo en algún *buffer*, OpenGL puede manipular el fragmento y actualizar el píxel de distintos modos. Los detalles de las funciones se pueden ver en el *Red Book*. Pueden dividirse en dos tipos: *test* y alteración. Por una cuestión de eficiencia los *tests* se hacen primero. Ya vimos el *test* del *z-buffer*: el fragmento se descarta si no pasa el *test* de *z*.

Es importante conocer el orden de las operaciones: *FRAGMENT → ownership → texturing*[2] *→ fog → scissor → alpha → stencil → depth → blending → dithering → logic op → masking → PIXEL*.

Test:

Los *test* son: *ownership*, *scissor*, *alpha*, *stencil* y *depth*. Si el fragmento no pasa un *test*, se descarta (aunque en el *stencil* suceden cosas aun si se debe descartar el fragmento).

El *pixel* ***ownership*** *test* verifica si el píxel está tapado por otra ventana. Si la ventana esta (parcialmente) tapada por otra, parece razonable que no se pierda tiempo haciendo cálculos cuyos resultados no se verán. Pero no siempre: Supongamos un largo proceso en *batch*, donde el procesador trabaja desatendido por el usuario y minimizado o detrás del Facebook. Cuando el trabajo termina, se pretende ver el resultado o leer los píxeles del *color buffer* para grabar una imagen. En tal caso el descarte sería un error. Desgraciadamente, si el fragmento se descarta o no, depende de la marca y modelo de la placa, la versión de OpenGL y el sistema operativo[3]. En tal caso hay que leer sobre éste *test* (aunque más simple es experimentar) pues puede que no trabaje bien en los píxeles tapados por otro programa.

Los siguientes *test* se habilitan con glEnable() o se deshabilitan con glDisable().

El ***scissor*** *test* (tijeras) define un rectángulo dentro del cual se dibuja y por fuera no. Permite, por ejemplo, restringir la escritura (y borrado) a una región variable (cuadro de diálogo u otro *widget*), cuando el resto es constante. Es más eficiente que usar el *stencil* o un *viewport* para la misma acción.

Todos los tests que siguen comparan valores para decidir el curso de acción. El conjunto de posibles comparadores es: GL_ALWAYS, GL_NEVER, GL_EQUAL, GL_NOTEQUAL, GL_LESS, GL_LEQUAL, GL_GREATER y GL_GEQUAL. Los *defaults* son GL_ALWAYS (pasa siempre) para *alpha* y GL_LESS para el *depth*; para el *stencil* no hay *default* pues sólo se usa si se define.

El ***alpha*** *test* compara el *alpha* que trae el fragmento (junto con el color) con un valor de referencia, que se define al habilitar el test. Sirve para realizar el descarte de ciertas partes. Por ejemplo, el fragmento se descarta si el *alpha* es igual (GL_EQUAL) a 0 (el valor de referencia); ese es el método estándar para definir partes invisibles. El *alpha test* (como todo *test*) es pasa / no pasa y no debe confundirse con la mezcla o *blending*, que veremos en breve y que también suele utilizar el *alpha* del fragmento. Otro uso es para renderizar transparencias múltiples en dos pasos: los objetos que sean o que tengan partes semitransparentes se ordenan desde el fondo hacia el ojo; en el primer paso se renderizan sólo los objetos opacos descartando los fragmentos con *alpha*<1; en el segundo paso se descartan los opacos (*alpha*=1), renderizando sólo fragmentos semitransparentes o invisibles.

[2] La aplicación de texturas y *fog* (niebla) las vemos en otros temas.

[3] Este test en particular no se encuentra bien explicado ni en el Red-Book ni en la especificación de OpenGL. Para garantizar resultados correctos en caso de oclusión de la ventana hay que usar *framebuffer objects*; de ese modo, el test pasa siempre.
www.opengl.org/discussion_boards/ubbthreads.php?ubb=showflat&Number=246762
www.opengl.org/resources/faq/technical/rasterization.htm.

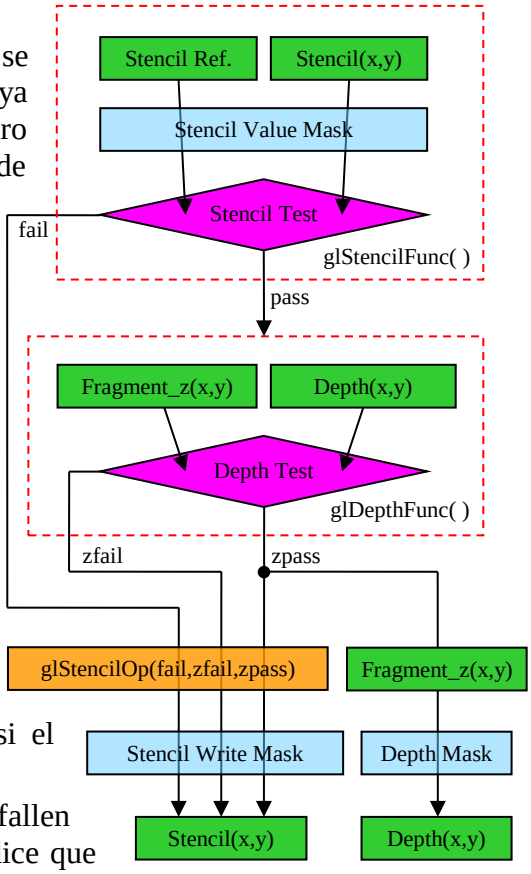
Al ***stencil*** *test* le daremos una importancia especial por lo complicado y útil que resulta.

Así como hay un *depth-buffer* paralelo al de color, también hay un *stencil buffer*; pero en éste se suele "dibujar". Consta típicamente de 8 bits por pixel y sirve para poner banderas o *flags* (números enteros que se interpretan bit a bit) en distintas regiones y así lograr distintos efectos dependiendo del *flag* en el pixel y de los filtros o máscaras utilizados. En principio, se podrían definir 256 acciones distintas en cada píxel, dependiendo del valor del *stencil* en ese píxel. Para definir una zona se renderiza un objeto "pintando" en el *stencil buffer* con un "color" que es el número [0-255] elegido.

El ejemplo típico es muy simple: se "dibuja" sobre el *stencil* una circunferencia (o cualquier cosa) rellena con color 1 sobre fondo 0; luego se renderiza una escena sobre el *color buffer* cuyos fragmentos solo pasan donde hay un uno en el *stencil* y otra escena distinta que solo pasa donde hay un cero; el efecto y el nombre del *stencil* se hacen obvios.

Otro ejemplo: se dibujan dos rectángulos solapados en el *stencil-buffer*, uno con "color" 1 y otro con color 2, siendo el fondo de color 0. De algún modo se establece que, cuando "pinta", solo enciende los bits del color que entra sin apagar los que estaban; es decir que habrá zonas con el primer bit (1) encendido, otras con el segundo (2), otra con ambos (3) y un "fondo" con ninguno (0). Luego se renderiza otra escena ("la" escena) sobre el *color-buffer*, pero cada fragmento se trata de distinto modo, de acuerdo al bit encendido en el correspondiente píxel del *stencil-buffer*.

Como se puede ver en el diagrama, el flujo de información es bastante complejo; pero está hecho de ese modo para aumentar las posibilidades de acción.



Si un fragmento no pasa el *test* de *stencil* o el de *z*, entonces se descarta, no sigue hacia el *color buffer*. El *z-bufer*, como ya sabemos, se actualiza sólo si el fragmento pasa el *test* de *z*; pero el *stencil buffer* se actualiza de distinto modo, dependiendo de que *tests* pasó o no.

Para el *stencil test*, hay una función (glStencilFunc) que define como se harán la comparación y el filtrado; define un valor de referencia (*ref*), una máscara (*stencil value mask*) y el comparador del *test* (*stencil test*). Primero se enmascaran (*bitwise and*) el valor de referencia y el almacenado en el pixel del *stencil buffer*; luego, ambos resultados se comparan usando el comparador solicitado. Si el *test* falla, el fragmento se descartará; si pasa va al test de profundidad.

Como ya se explicó, en el *depth test*, el *z* del fragmento se compara con el valor almacenado en el *z-buffer*, usando el comparador elegido mediante glDepthFunc. Si el fragmento no pasa, se descartará. Si pasa, sigue su camino al *color buffer*, mientras que se actualiza el *depth buffer*, pero previo paso por la máscara de escritura *depth mask*, que indica si el *z−buffer* se puede escribir (*true*) o no (*false*).

El *stencil buffer* se actualiza de distinto modo según que *tests* fallen o pasen. Para ello se provee otra función glStencilOp(), que dice que hacer en cada uno de los tres casos: falló el *stencil test*, pasó el *stencil test*, pero fallo el de profundidad, pasó ambos *tests*. En cada uno de los casos se decide si se escribe y qué se escribe en el *stencil buffer*; puede ser que se borre, que se incremente, que se invierta, que se escriba el valor de referencia, etc. Pero, para darle más versatilidad, primero se enmascara, lo que se deba escribir, con una máscara de escritura (*stencil write mask*) que se define aparte, mediante la función glStencilMask(). Lo que tiene que hacer, entonces lo hace, pero sólo donde la máscara de escritura tenga bits en uno.

¡Simple!

Quizás nadie utilice todas las funcionalidades en un mismo algoritmo, pero ahí están; para hacer trucos simples o muy elaborados, en forma muy eficiente. Hay muchos ejemplos disponibles en la web.

Alteración:

Las operaciones de alteración son las que rellenan un píxel de un *buffer*; eso puede hacerse en forma directa o mezclando el color del fragmento y el del píxel y pasando luego a través de filtros o máscaras.

El ***blending*** mezcla el color del fragmento entrante con el alojado en el correspondiente píxel del *color buffer*, dependiendo de la operación de mezcla seleccionada.

La operación de mezcla clásica es glBlendFunc(GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA), se habilita cuando hay fragmentos semitransparentes (0 < alfa < 1). Con esa operación, el color del píxel resulta: $C_d \leftarrow A_sC_s + (1-A_s)C_d$. Donde C_d (*destination*) es el color R, G o B del píxel en el color buffer, mientras que C_s (*source*) es R, G o B del fragmento entrante. A_s es el *alpha* del fragmento *source* o entrante, que se utiliza aquí como opacidad (1: opaco, 0: invisible, intermedio: semitransparente).

El *blending*, en general, se define mediante la función glBlendFunc(sfactor,dfactor); con un factor multiplicativo para C_s y otro para C_d. La tabla de factores es la siguiente (Tabla 6.1 del *Red Book*):

Constant	**Relevant Factor**	**Computed Blend Factor**
GL_ZERO	source or destination	(0, 0, 0, 0)
GL_ONE	source or destination	(1, 1, 1, 1)
GL_DST_COLOR	source	(R_d, G_d, B_d, A_d)
GL_SRC_COLOR	destination	(R_s, G_s, B_s, A_s)
GL_ONE_MINUS_DST_COLOR	source	(1, 1, 1, 1)-(R_d, G_d, B_d, A_d)
GL_ONE_MINUS_SRC_COLOR	destination	(1, 1, 1, 1)-(R_s, G_s, B_s, A_s)
GL_SRC_ALPHA	source or destination	(A_s, A_s, A_s, A_s)
GL_ONE_MINUS_SRC_ALPHA	source or destination	(1, 1, 1, 1)-(A_s, A_s, A_s, A_s)
GL_DST_ALPHA	source or destination	(A_d, A_d, A_d, A_d)
GL_ONE_MINUS_DST_ALPHA	source or destination	(1, 1, 1, 1)-(A_d, A_d, A_d, A_d)
GL_SRC_ALPHA_SATURATE	source	(f, f, f, 1); f=min(As, 1-Ad)

En el mismo *RedBook* o en otras fuentes se pueden ver algunos trucos y usos alternativos del *blending*; pero algo más veremos al estudiar texturas, que es donde más se aprovecha.

También se pueden hacer **operaciones lógicas** entre el fragmento entrante y el almacenado mediante glLogicOp() que compara bit a bit los valores y produce un resultado de acuerdo a la operación lógica seleccionada de las dieciséis posibles. El uso típico es para cursores o líneas temporarias, como por ejemplo los rectángulos de selección; se pretende que dibujando lo mismo dos veces reaparezca lo que había, para ello se puede usar *xor* (^ *exclusive or*) o *invert*; la idea es que $C_d \hat{} C_s \hat{} C_s = C_d$, es decir que, si el píxel era de color C_d y lo pinto con color C_s aplicando *xor*, el color cambia a $C_d \hat{} C_s$; si luego vuelvo a pintar el resultado con el mismo color C_s y *xor*, vuelve a aparecer el color C_d original.

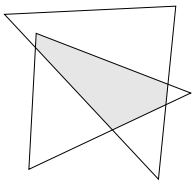
OpenGL prescribe que, al activar las operaciones lógicas, se desactiva el *blending*. Además, las operaciones lógicas actúan cuando el buffer de color está especificado en formato entero (normalmente 0-255) y no en formato *float* (0-1).

Una operación de alteración costosa es glClear(), que borra todo el contenido del/los *buffers* solicitados, es costosa porque recorre entero cada *buffer* que debe borrar, asignando el valor default píxel por píxel (aunque se realiza usando *blitting*). El default se define para cada caso con glClear<Buffer>; <Buffer> puede ser Color (para definir el color de fondo), Accum, Depth (el *default* es el 1 que corresponde a *zfar*) o Stencil.

También hay máscaras de escritura: gl<Buffer>Mask() para *color, depth y stencil*. Para *color* y *depth* son booleanas, indicando si el buffer se puede escribir (*true*) o no (*false*); para el *stencil* es un número del mismo tamaño en bits que el *stencil*, permite alterar sólo aquellos bits donde la máscara tiene 1. Cuidado: al borrar (glClear) se escribe el *clear-value* en todos los píxeles y también se aplican las máscaras.

Ejercicios:

- ¿**Qué es** el canal alpha? ¿Para qué sirve?



- Un programa permite que el usuario mueva los vértices de dos triángulos en el plano y, mientras lo hace, muestra las aristas y la intersección rellena.
  ¿Cómo logra ese efecto mediante una técnica raster? (image-precision y no vectorial).

- Describa un algoritmo que genera una imagen con sensación de movimiento rápido (*motion-blur*). Despreocúpese por la sintaxis y las posibilidades reales del *Accumulation Buffer* de OpenGL, puede utilizar un buffer suyo y manejarlo como quiera. Cuidado con el rango de los colores ([0,1] o [0,255]).

- Explique cómo lograr el efecto que se muestra en la figura a la derecha para diferenciar las líneas ocultas.
  a) Cuando los objetos tienen las caras rellenas
  b) Cuando los objetos no tienen las caras rellenas.