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, el fragmento es un candidato a píxel, mientras que el píxel es el valor alojado (o a veces el espacio que lo aloja) en un *buffer* de memoria. Los fragmentos pueden ser producidos por las rutinas de rasterización o en la aplicación de texturas. Ambos (fragmento y píxel) son valores que corresponden al punto de coordenadas enteras $\{x, y\}$ en la imagen de tamaño WxH: $0 \leq x < W$; $0 \leq y < H$.

La cantidad de memoria disponible para cada píxel se suele denominar "profundidad" (*depth*) como si fuese una pila de "planos" de imagen, uno por cada bit (*bitplane*) de memoria disponible por píxel. La utilización de gráficos en computación comenzó con un bit por píxel, en los llamados "*bitmaps*" que eran imágenes de puntos blancos o negros. Para simular los grises se usó el *dithering* o puntillado, que aún puede verse con lupa en los diarios 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 y, vista de 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 precio de la memoria.

Del blanco y negro (¡o peor, ámbar o verde!) se pasó a 16 y rápidamente a 256 colores organizados en una "paleta" (*palette*) y actualmente se utilizan 24 bits de profundidad para RGB (o 32 para RGBA), más adelante, cuando expliquemos la teoría del color, veremos el exceso de llamar *true color* a ésta paleta muy densa. Con 24 bits de color podemos manipular 8 para cada color, rojo, verde y azul. El mínimo rojo de 8 bits es cero y el máximo 255 aumentando de a uno, pero internamente, el mínimo es 0 y el máximo 1, aumentando de a 1/255.

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

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.

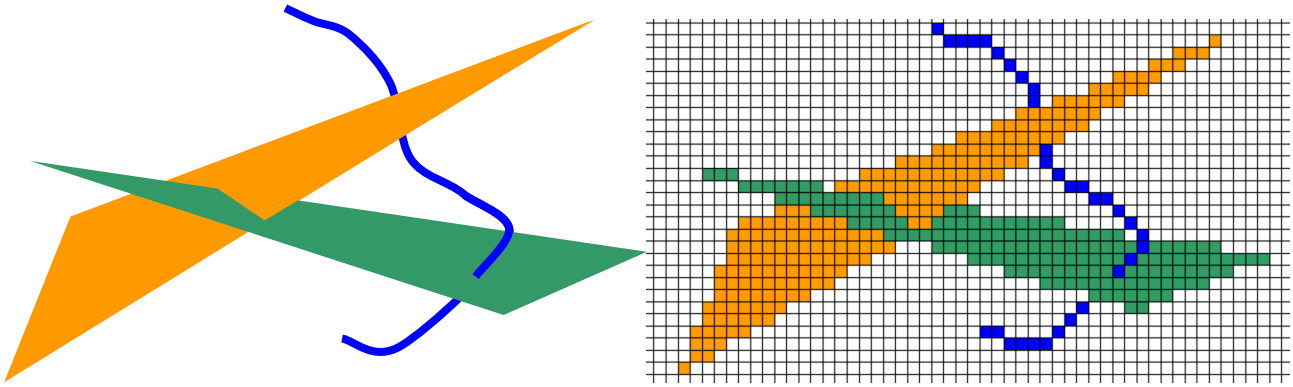
Primero veremos el *z-* o *depth-buffer*, como ejemplo introductorio para una generalización posterior.

z-buffer

Al renderizar varios objetos 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 en forma ordenada, de acuerdo a la distancia al observador. El problema es la velocidad, y hay muchísimas técnicas que aceleran los cálculos y el ordenamiento. Pero cuando se dispone de suficiente memoria y hardware especializado para la rasterización, se utiliza el algoritmo basado en el *z-buffer* o *depth-buffer*. Básicamente consiste en "pintar" el píxel 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 recién calculado.

El *color-buffer* es la porción de memoria que almacena los WxH píxeles de cada componente de color (en general, un *byte* por cada componente RGBA, seguidos). Ese es el *buffer* que se pinta o actualiza cuando un fragmento es visible. El *depth-buffer* se mantiene en paralelo y consiste en WxH valores de punto flotante, normalmente de 24 o 32 bits. La porción de espacio visible está limitada mediante un plano cercano al ojo y uno más alejado, las distancias *znear* y *zfar* del ojo a los planos sirven para cuantizar el rango de profundidades, ya que no hay precisión infinita para almacenar los valores de z.



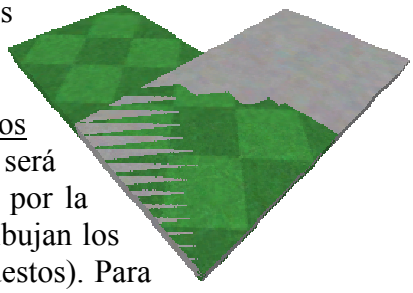
El algoritmo fue desarrollado por Ed Catmull (uno de los fundadores de Pixar) en 1974 y es algo así:

Inicializar el *z-buffer* con el *z* máximo, el del *far-plane*: $\forall \{x,y\} \in [0,W)\times[0,H) \Rightarrow depth(x,y) = zfar$
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)

El algoritmo no almacena el valor exacto de *z* entre *znear* y *zfar*, convierte ese valor en uno que está entre cero y uno, normalmente con precisión de 24bits.

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 que se va a dibujar es semi-transparente, debe combinarse su color con el color actual del píxel. Pero, si aparece un tercer objeto intermedio y también semi-transparente, ya no se dispone de la información necesaria para calcular bien el color resultante. Cuando hay transparencias, el resultado depende el orden de rasterización. Una técnica utilizada consiste renderizar primero los objetos opacos y luego los semi-transparentes, pero ordenados desde los mas lejanos a los mas cercanos al ojo.



2) Precisión: Con 24 bits, los valores alojados, que son números 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 resultar en visualizaciones defectuosas. Esto obliga a ajustar con precisión los planos *near* y *far* para aumentar la precisión del resultado, que aún así será malo. Este defecto se llama *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.

3) No es posible aplicar *antialiasing* al rasterizar: Cuando un píxel esta formado por varias fuentes distintas (bordes) el *antialiasing* se realiza pintando el píxel con un promedio ponderado de las fuentes. Con el *z-buffer* se pinta de acuerdo a una única fuente.

4) El algoritmo de *z-buffer* requiere *rasterizar* muchas primitivas (todas las 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 y descarte masivo (*clipping* y *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 unicos 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

Extensión a 3D del Algoritmo de Bresenham

Vimos que para *rasterizar* segmentos rectos con el algoritmo de Bresenham, se aprovecha la constancia de la derivada. En 3D sucede lo mismo, pero con las derivadas parciales, y eso permite calcular *z* en cada paso de *rasterización* y así rellenar el *depth-buffer*.

Si estamos *rasterizando* un segmento, avanzamos de a un píxel en *x* o *y*, la coordenada *z* del píxel actual es igual que la del anterior más una constante: la derivada parcial de *z* respecto de *x* o *y*. Haciendo el cálculo de antemano, en el *loop* de *rasterización* sólo se suma un factor constante.

En el caso de polígonos u otras figuras planas, que rasterizamos mediante *scan-lines*, el proceso es igual: siempre se suma la derivada parcial de *z* respecto de x, que es constante para el plano.[2]

El *frame-buffer*

OpenGL trabaja con varios *buffers*, ya conocemos el *color-buffer*, donde se arma la imagen que vemos y el *depth-buffer*, que sirve para realizar 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 pueden leerse, manipularse y sobrescribirse desde el programa, OpenGL trae además algunas rutinas propias de manipulación.

Los *buffers* estándar 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 utilizan como *layers* o capas o también se pueden utilizar para almacenamiento temporario de imágenes generadas.

Un ejemplo sencillo de utilización del *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 utiliza iluminación en lugar de un color fijo, el color es variable; en tal caso se puede hacer un renderizado invisible en un *buffer* auxiliar (normalmente en el *back* y sin *swappear*) con un color fijo y distinto para cada objeto, leyendo el color del píxel se puede conocer el objeto debajo del el cursor.

Se pueden lograr algunos trucos útiles manipulando el *z-buffer*. Un ejemplo es la determinación de siluetas buscando altos gradientes de *z* entre píxeles. También se puede declarar *read-only* al *z-buffer* (no se actualiza) o cambiar la función que compara los valores de *z* entrante y almacenado, con ello pueden hacerse varios pasos de *renderizado* para dibujar en uno las líneas invisibles de una manera y en el siguiente 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. La aplicación geométrica más interesante es para selección en 3D: la posición *x*, *y* del cursor define una línea que atraviesa el modelo, pero el punto 3D picado sólo puede conocerse leyendo el *z-buffer* en ese píxel.

El *stencil buffer* se utiliza para muchísmos trucos. El nombre proviene del uso por el cual se creó, que consiste en enmascarar la zona de dibujo. Por ejemplo en un juego de carreras de autos, el parabrisas sirve de máscara para *renderizar* dentro toda la escena externa variable y por fuera muchas partes constantes excepto el volante o algunos indicadores del tablero; los espejos definen otras máscaras

[2] **Nota importante**: La técnica de prever lo que va a suceder se conoce con el nombre de **coherencia** y se utiliza siempre en computación gráfica. En este caso, conociendo la derivada sabemos exactamente lo que va a suceder: la posición *z* del próximo fragmento, pero puede suceder que la derivada no sea constante, en tal caso podemos suponer que no será muy distinta de un píxel a otro. Saber aproximadamente lo que va a suceder es mejor que no tener ninguna idea. Esto se aprovecha en múltiples situaciones, por ejemplo en una animación: si las cosas son así, en el próximo cuadro no serán muy distintas, o mejor: si conozco como varió del anterior a éste, probablemente el mismo cambio se aplique al siguiente.

para renderizar la vista trasera. También se utiliza para realizar reflejos (simple) y sombras (complicado a muy complicado); hay ejemplos e instrucciones de la web, pero primero deberíamos aprender algo más sobre transformaciones. Otro uso muy interesante y muy complicado es para renderizar sólidos armados mediante las operaciones booleanas de un CSG *tree*.

El *buffer* de acumulación se utiliza para acumular datos de color, como si fuesen distintas manos de pintura con efecto acumulado (con ciertas reglas de enmascaramiento y sobreescritura) sobre una misma superficie. Sirve por ejemplo para hacer el borroneado por movimiento (*motion blur*) dando la impresión de velocidad o para hacer *antialiasing* o 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 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* que se utilizan se escriben, borran y enmascaran mediante sencillas reglas que pueden verse en el *Red Book* u otro manual de OpenGL. Lo que se debe saber es que los datos pueden manipularse logrando una miríada de efectos muy útiles, se trabaja directamente con los píxeles (o fragmentos) en modo *raster*.

Una imagen completa puede leerse desde un *buffer* como un rectángulo de píxeles; por el contrario, una imagen leída, por ejemplo de un archivo, puede escribirse en algún *buffer*. OpenGL trae un juego especial de funciones para leer, escribir y manipular imágenes y un *pipline* especial para hacer esas operaciones en forma eficiente.

Operaciones sobre fragmentos:

Después de rasterizar, OpenGL puede realizar varias operaciones de manipulación de los fragmentos antes de actualizar un píxel. 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 píxel se pinta si pasa el test de *z*.

Los tests son: *ownership*, *scissor*, *depth*, *alpha* y *stencil*. En cuanto el fragmento no pasa uno de la serie, inmediatamente se descarta.

El *pixel ownership test* verifica si el píxel está tapado por otra ventana. 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. Si la ventana esta tapada, es razonable que no se pierda tiempo haciendo cálculos cuyos resultados no se verán. Ahora, supongamos que después de un largo proceso en *batch*, donde el procesador trabaja mucho y desatendido por el usuario, cuando el trabajo termina, el programa lee los píxeles del *color buffer* y graba una imagen. En tal caso hay que leer sobre éste test (o experimentar) pues puede que ni dibuje, ni trabaje, ni lea bien, los píxeles que no son visibles por estar tapados por otro programa[3].

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

El *scissor test* (tijeras) es extremadamente simple, se define un rectángulo dentro del cual se dibuja y por fuera no. Se usa para renderizar por secciones. ~~Se puede ver como una versión reducida del *stencil*.~~

Todos los tests que siguen comparan valores para decidir el curso de acción. Los test de *alpha* y *stencil*, comparan el valor del fragmento con un valor de referencia, que se define junto con el test, el *depth* del fragmento se compara con el valor almacenado en el *depth-buffer*. Todos utilizan el mismo conjunto de comparadores entre valores: GL_ALWAYS, GL_NEVER, GL_EQUAL, GL_NOTEQUAL,

[3] Este test en particular no se encuentra explicado ni en el Red-Book ni en la referencia de Opengl. Para lograr resultados correctos en caso de oclusión de la ventana hay que usar *framebuffer objects*:
www.opengl.org/discussion_boards/ubbthreads.php?ubb=showflat&Number=246762
www.opengl.org/resources/faq/technical/rasterization.htm.

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* se suele utilizar con texturas: si una imagen que se aplica sobre una primitiva tiene, además de R, G y B un valor A, de *alpha*, ese valor compara con un valor de referencia para realizar efectos de difuminado o invisibilidad de ciertas partes de la imagen. Otro uso es para renderizar transparencias múltiples en dos pasos: primero pasan sólo los fragmentos con *alpha*=1 y luego los de *alpha*<1, pero sin alterar el contenido del *z-buffer*.

Las operaciones de alteración son las que rellenan un píxel, eso puede hacerse en forma directa o a través de filtros o funciones de combinación. Veremos dos funciones de combinación: mezcla y lógica.

El *blending* mezcla el fragmento entrante con el alojado en el píxel, dependiendo del *alpha* de cada uno de ellos y la operación de mezcla seleccionada.

La operación de mezcla clásica, cuando entra un fragmento semitransparente, es: D = As S + (1-As) D. En esa ecuación S (*source*) es cada uno de los valores R, G o B del fragmento entrante, mientras que D (*destination*) corresponde a la derecha al píxel almacenado en el color buffer y a la izquierda al color resultante que será almacenado en el píxel. As es el valor *alpha* del píxel entrante, que se utiliza aquí como nivel de opacidad del fragmento que entra.

En general, el *blending* se define mediante la función glBlendFunc(sfactor,dfactor); con un factor multiplicativo para S y otro para 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)

El ejemplo mostrado más arriba sería: glBlendFunc(GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA).

En el mismo *RedBook* o en otras fuentes se pueden ver algunos trucos y usos alternativos del *blending*.

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 existentes. La que más se utiliza es *xor* (*exclusive or*) para cursores o líneas temporarias, como por ejemplo los rectángulos de selección. La idea es que D^S^S=D, es decir que si el píxel era de color D y lo pinto con color S aplicando *xor*, el color cambia a D^S; si luego vuelvo a pintar el resultado con color S (y *xor*) vuelve a aparecer el color D original. Un efecto similar se logra con el operador *invert*, pues al invertir dos veces también queda lo que estaba originalmente.

Es importante conocer el orden de operaciones: *FRAGMENT → ownership → texturing*[4] *→ fog*[4] *→ scissor → alpha → stencil → depth → blending → dithering → logic op → masking → PIXEL*.

Una operación de alteración muy 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. El default se define para cada caso con glClear<Buffer>, donde <Buffer> puede ser Color (para definir el color de fondo), Accum, Depth (el *default* es el 1 que corresponde a *zfar*) o Stencil.

[4] La aplicación de texturas y *fog* (niebla) la veremos más adelante.

OpenGL posee un *pipline* especial para realizar eficientemente las operaciones costosas de *bitblt*: *bit block transfer*, esto es lectura y escritura de rectángulos de píxeles contenidos en algún buffer.

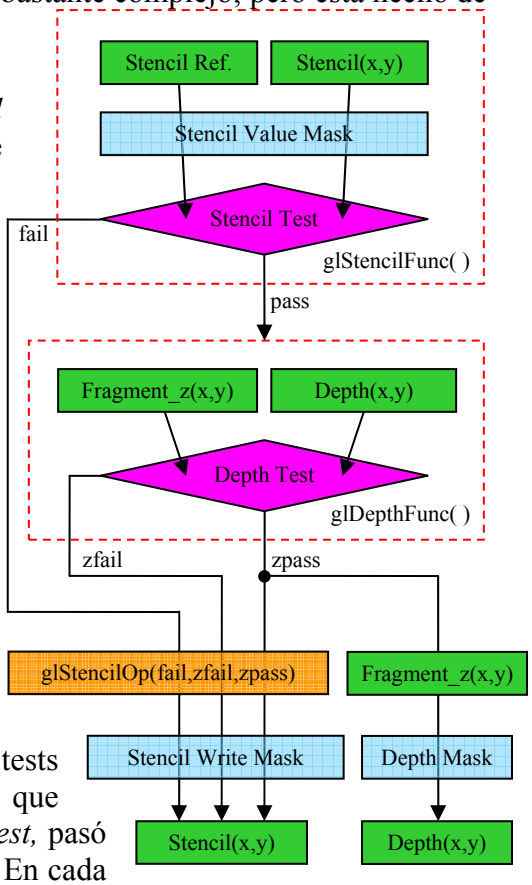
Hay máscaras de escritura: glBuffer>Mask() para *color, depth y stencil*. Para *color* y *depth* son Booleanas (un solo bit), indicando si el buffer se puede escribir o no; para el *stencil* tiene el mismo tamaño en bits que permite el *stencil*, indicando cuales bits pueden ser alterados. Cuidado: borrar (glClear) es escribir el *clear-value* en todos los píxeles; también se aplican las mascaras de escritura.

Las operaciones simples de lectura y escritura se pueden hacer con glReadPixels y glWritePixels, pero hay varias opciones más que permiten operaciones eficientes para *bitmaps* (*fonts*) o de mapeo (glPixelMap) de los valores, etc.

Stencil Buffer

El *stencil buffer* consta típicamente de 8 bits y sirve para poner banderas (*flags*, números enteros que se interpretan bit a bit) en distintas regiones y así lograr distintos efectos dependiendo de la bandera correspondiente al lugar donde se dibuja y a los filtros o máscaras utilizados. En principio, se podrían definir 256 acciones distintas en cada píxel, dependiendo del valor del *stencil* en ese píxel.

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 el fragmento no pasa el test de *stencil* o el de *z*, entonces se descarta, no sigue hacia el *color buffer*; pero aún así, el *stencil buffer* se actualiza, sólo que de distinto modo, dependiendo de que tests pasó o no el fragmento.

Hay una función (glStencilFunc) que define como se hará el filtrado; define un valor de referencia (*ref*), una máscara (*value mask*) y un *test* (*stencil test*). Primero se enmascaran (*bitwise and*) el valor de referencia y el valor almacenado en el *stencil buffer*; luego, ambos resultados se comparan usando el *test* solicitado. Si el *test* falla, el fragmento se descarta; 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 test elegido mediante glDepthFunc. Si el fragmento no pasa, se descarta; 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 o no.

El *stencil buffer* se actualiza de distinto modo según que tests fallen o pasen. Para ello se provee una 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 mascara de escritura tenga un bit en uno.

¡Simple!